Final Report

Data Anarchy: Building A Social Media Platform Where Users Control Their Data

CM3203 - One Semester Individual Project

40 Credits

School of Computer Science and Informatics

Cardiff University, 2022

Author: Tim Lane

Supervisor: Kathryn Jones

Moderator: Natasha Edwards

Abstract

This project explores possible ways to implement a peer-to-peer social media platform based upon data anarchy - a data storage model where the user's data is stored on their own device, rather than a server. This involves both the practical design and implementation of a prototype app for Android devices which aims to emulate common social media features with the data anarchy concept. Towards the end of the project, research was conducted to gather attitudes towards 3 main types of data storage model used in social media - centralised, federated and peer-to-peer - as well as performing user testing with the prototype app to see how usable it is for existing social media users. A wide range of concepts and ideas for additional features and improvements to the platform are discussed. The overarching goal of the project is to create a working proof-of-concept for future research and development to build upon.

Acknowledgements

I would like to express my thanks to Dr Kathryn Jones for taking on the role of my supervisor for this project, who encouraged and guided me throughout. I am also very grateful for those who took part in the research study and user testing, without whom this project would have failed to assess user standpoints on social media and the prototype. Finally, I would extend a special thanks to the local Cardiff takeaways for keeping me going in my time of need.

Contents

List of Figures

List of Tables

1 Introduction

With estimates of 3.96 billion MAU (Monthly Active Users) users worldwide during 2022 and continuing to grow [1], social media has never been more important. The behemoth Facebook is less than 20 years old [2] and yet boasted 2.936 billion monthly active users in the first quarter of 2022 [3]. Yet despite the prevalence and growth of these platforms, the client-server model allows for numerous problems to occur such as outages [4] [5], data breaches [6], account hijacks [7] and GDPR breaches [8]. By contrast, dynamic peer-to-peer networks are more resilient to faults [9], and do not rely on a central database that could be targeted for cyberattacks. The nodes of a peer-to-peer network are also responsible for the sharing of data and determine *who* is allowed access to *what* data. This means no central authority has ownership or licence over data in the network - instead, the end users are in full control of their data, which greatly enhances privacy.

Therefore, it seems sensible to build on what we have learned from the rise of existing social media platforms and resolve some of their issues by using a decentralised peer-to-peer networking and data storage model. By developing an alternative to popular social media platforms like Facebook in the form of a mobile app, we can directly compare the user experience and data privacy aspects and determine whether such a platform is viable.

1.1 Aims & Objectives

The primary objective of this project is to build a social media app where user data is stored on user devices, rather than in a server database. The aim is not to build a complete, production-ready platform but to create a prototype as a working proof-of-concept for future iteration. Additional aims include exploring how viable the primary aim is when scaled up and extended beyond the MVP (Minimum Viable Product), with further conceptualisation of how it could be improved.

The full list of specific aims and objectives from the initial report has been updated with some changes. These changes have generally been made to better relate objectives to the overarching aim of reproducing common social media features using a unique data storage model where data is not stored in a server, instead relying on the user's own device.

1. Identify key features and problems in popular centralised social media platforms, then compare the problems with decentralised alternatives.
2. Determine the requirements of a generic decentralised social media platform based on identified features and problems, and specify more specific MVP requirements for a mobile application.
3. Investigate peer-to-peer networking technology and how it can be used to emulate common features of social media platforms.
4. Build a peer-to-peer prototype MVP social media application for Android devices that:
    a. Does not store any user data on a server beyond the lifetime of a user connection.
    b. Uses a peer-to-peer network for sharing multimedia data between users, including text and images.
    c. Includes at least 3 common features of existing social media platforms.
5. Carry out user testing and survey the participants for feedback.
6. Identify ways to improve the platform, and briefly conceptualise ways to implement additional common social features in the future.
7. Evaluate the project, determining whether it meets these objectives.

1.2 Intended Audience & Beneficiaries

The primary intended audience for this project are the open-source community, fellow computer scientists and software developers. However, the main beneficiaries include anyone who wants a new social media platform that addresses a variety of serious problems with traditional platforms. The reasons for wanting an alternative platform are vast and varied, but some examples of problems users face with existing social media platforms include:

- Lack of data privacy
- Lack of control over content personalisation
- Dislike of third-party advertising
- Problems with lack of persistence in existing platforms (such as individual Mastodon server instances being taken down [10])
- Lack of controls to stop targeted harassment & cyberbullying campaigns

I propose that many of these problems can be addressed more readily in a decentralised, peer-to-peer platform where data is stored locally on user devices, rather than server databases. By exploring the potential of such a platform with a working open-source prototype, others can freely contribute as well as carry out further research and development on this kind of data storage model for use in applications including (but not limited to) social media.

1.3 Project Scope

This project will focus more on the technical and general usability aspects of developing such a platform than large-scale long-term use of such a platform as there is only enough scope to build a MVP. 12 weeks would not be sufficient to explore every aspect of using such a radically different kind of data storage model for a social media application, in contrast to what we commonly see with server databases or publicly distributed data storage architectures. As such, the project is aiming to be a basic proof-of-concept and initial exploration of the topic - a jumping off point for further research and development.

There are many possible features to integrate into the prototype itself; to avoid overscoping, these are limited to requirements detailed in section 3.2. Those requirements are primarily based on some of the common features of existing social media platforms. However, some exploration of additional features and considerations are detailed throughout the design, user interface and future work chapters as there is a lot to discuss on this topic which cannot be put into practice with the prototype due to the limited time allocated for this project.

1.4 Approach

A form of the Agile methodology mixed with the spiral prototype model is used for development of the prototype social media platform. This involves fortnightly sprints, using the initial plan and the design specification as guides for iterative rapid prototype development. This is necessary as it allows changes to be made as iterations are made over time, such as when features have to be cut due to unforeseen challenges forcing reprioritisation. This is useful as development can be readily used to inform further design decisions, in contrast to the waterfall development methodology which doesn't support rapid iteration and design improvements.

A basic design specification is used for guiding the overarching project, determining what the requirements of the application are before moving onto the rapid iterations and development of the design, user interface and implementation. Once the MVP has been reached, research can begin including some basic user testing to gauge the usability of the mobile app.

1.5 Assumptions

Due to the breadth of topics explored in this project, the scope has been quite challenging to define. As such, I have selected some features of social media for exploration and not others based upon a mixture of social UI design patterns [11] and my own experience using a range of social media platforms. These assumptions have largely informed the development of the Peerway platform, rather than first-hand research, as getting focus groups together would take a lot of time which could otherwise be spent on other aspects of the project. Furthermore, it is expected that the reader of this report is familiar with general terms such as "social media" as terms such as these have become sufficiently well-known in recent decades.

This brings me to the project itself. This project is not intended to be considered as a complete solution. Rather, it is a proof-of-concept research & development project that builds a basic working model, identifies a range of challenges and potential solutions and provides a jumping off point for further exploration.

Other assumptions relate more to the social media platform implementation, research and user testing. These include assumptions about users that:

- Have an Android device
- Have internet connectivity
- Have some experience using social media apps already

This project will not assess usability for individuals who have never used social media prior or those who use iOS, for example. This is in order to limit the scope and focus more on the main aims and objectives.

1.6 Key Outcomes

The results of the research combined with technical knowledge learned during development and examining the resultant prototype are fundamental to indicating how viable such an alternative social media platform could be. In particular, relating the use of the peer-to-peer data storage model described in section 1.1 to working reproduction of common popular social media features would be useful to see in practice. Gaining insight on ways in which such a platform could solve some of the issues users face with other platforms in relation to the data storage model would be very useful to inspire further research and development on the subject. Additionally of course, ideal outcomes would include meeting all or as many of the aims and objectives as possible.

Ultimately, this project acts as an initial exploration of a new kind of web application which takes inspiration from existing decentralised, open-source projects such as Solid [12] but with a focus on ease of access to users; social media being a broad application domain which is accessible to many without technical know-how. If it's possible to make a truly free, decentralised social media platform which serves the wants and needs of existing social media users, it would be very beneficial to know if this kind of data storage design paradigm could be a core component of that ambition - or if there are hurdles that would hold it back.

2 Background

2.1 Popular Social Media Platforms

As the aim of this project is to build a social media platform similar to existing platforms but utilising a different underlying data storage model, it seems sensible to identify the most popular existing platforms. I have selected the top 5 platforms by MAU [13] for consideration.

Facebook

Despite being banned in China [14], Facebook is the most widely used social media platform with over 2.89 billion monthly active users [13]. The platform's main mobile app is also among the most downloaded apps on Android, at over 5 billion downloads [15]. Despite the huge user base however, the main Facebook mobile app does not have great review ratings. With only 2.9 out of 5 stars on the Google Play store for Android [16] and only 1.8 out of 5 stars on the App Store for iOS [17], it appears that the platform is quite unpopular among it's users - though it should be noted that there are far less reviews than downloads, at least on Android.

Facebook has two main apps; the first goes by the same name and has similar features to the web app version but lacks user messaging. A second standalone app named Messenger supports messaging other Facebook users. This means users of the platform can choose which app they want to use to suit their needs.

YouTube

Owned by Alphabet, the parent company of Google, YouTube is a platform that is dedicated primarily to watching and uploading videos [18]. Although coming in second to Facebook in terms of MAU [13], the YouTube app has over 10 billion downloads on Android [19] compared to Facebook which has over 5 billion downloads [15]. However, this could be attributed in part to the fact that the YouTube app is installed by default on Android devices, whereas Facebook users have to manually install the Facebook app.

WhatsApp

WhatsApp is owned by the same company as Facebook - Meta [20]. As a dedicated messaging app, WhatsApp differs from Facebook and YouTube significantly as there is less focus on content, with no activity feeds or personalised content. The presence of WhatsApp in the top 5 social media platforms along with the fact that both Facebook and Instagram feature messaging functionality suggests that messaging is an important aspect of social media platforms.

Instagram

Instagram brings the focus back to content and activity feeds once again. Also owned by Meta [21], Instagram focuses on the sharing of photos and videos uploaded by users.

WeChat

WeChat is unique in that it is most popularly used in China [22]. Owned by Tencent, WeChat is a social media app with features that make it similar to Facebook in some aspects - although it has been described as a kind of "super-app" [23] that has a whole host of features built in. In this regard WeChat is more than just a social media platform, though it's roots lay in being a WhatsApp style dedicated chatting app originally.

2.2 Common Features of Popular Social Media Platforms

User Profiles

User profiles display important information about a user, usually including a text-based identifier and an avatar. They sometimes include the user's age or date of birth, location, links to websites and a user-customised section that can be used for other miscellaneous information. Certain information may not be provided, depending on the platform and the user's preferences. Often, these profiles will include an activity feed containing posts that the user has either created or otherwise interacted with in some way.

Messaging

Many forms of social media offer support for sending and receiving messages. This allows for private, real-time communication between users, without automatically including content from other sources. Messaging can happen between one or more users and these are often referred to as chats or direct messages. For this project, I shall refer to chats specifically between two individuals as private chats and all other chats as group chats (having multiple members).

WhatsApp, Signal and Discord are examples of social media applications that are dedicated to messaging. However, many social media platforms such as Facebook, Twitter, Mastodon and others include messaging as just a single component of a whole host of features.

Activity Feeds

With the exception of some types of social media platforms (such as messaging-only platforms like Whatsapp), activity feeds are commonplace. Also referred to as activity streams [24], activity feeds show content from one or more sources. These content sources may be individual users, groups of users, or third-party sources such as advertisers. Throughout this project the term “activity feed” and the shorthand “feed” will be used interchangeably

Activity feeds usually prioritise the consumption of content over user interactions, although they often feature ways to interact with the content and content source. Interactions with content and sources via activity feeds are possible on most social media platforms, though the precise terminology differs from platform to platform. These can be summarised as follows:

- Indicating positive interest in the content (reacting, liking).
- Sharing content with others (reposting, sharing).
- Commenting on or discussing the content (replying, commenting).

Users can usually share their own content or that of others, effectively becoming a content source, which is included in particular activity feeds of users who have subscribed. On some platforms, content doesn’t have to be generated by the user to share it - it may be content shared from another content source, such as a link preview [25].

2.3 Issues Face By Existing Social Media Platforms

Privacy concerns

A significant concern about some social media platforms relates to privacy - from misuse of personal data to harassment and surveillance [26]. Data shared on centralised or federated platforms is stored in a remote database which is outside of the user's physical control. Serious problems such as data breaches [27] and data privacy scandals [28] have occurred in the past.

Spam & bots

Various social media platforms have faced issues with bots (automated accounts) which are used for a range of questionable activities, from artificially boosting engagement metrics [29] to suppressing political opposition [30]. Not only are bots negatively affecting use of social media platforms, but the lack of transparency regarding how content personalisation algorithms work is concerning, especially when some social media algorithms are found to be amplifying extremist content [31].

Targeted harassment & extremism

One unfortunate aspect of the rise of public social media platforms is that the ability to reach and communicate with more people has simultaneously made it easier for organised, targeted harassment to take place online [32]. The rise of conspiracy theories such as QAnon and other far-right extremism has been perpetuated in online spaces such as social media in recent years [33]. These effects pose serious threats not only to individual users, but geopolitical stability globally.

Security flaws, bugs & outages

In addition to data breaches, account hijacking scams enable malicious actors to steal not just data but also gain control of user accounts, such as recent Discord scams [34]. Serious bugs also occur at times, affecting the content that users see [35]. On top of these problems, a number of network outages have occurred recently, such as the Facebook services outage in 2021 [4] and the 2020 Cloudflare outage taking down a number of platforms including Discord [5].

2.4 Using Decentralisation To Solve Issues

Decentralised social media platforms have been gaining increasing attention in recent years [36], with platforms such as Mastodon offering an alternative to mainstream centralised social media platforms. Notably, these platforms are not operated by any single individual or organisation and even for federated platforms which still rely on servers for storing data, some of the issues discussed in section 2.3 are either less impactful to the networks as a whole or are minimised. However, peer-to-peer social media platforms are relatively unexplored by comparison to Federated server networks, and I think these could offer additional advantages to solving some of the issues faced by traditional social media platforms.

Privacy concerns

One reason for moving to a decentralised data storage model is the advantage that user data doesn't necessarily have to be hosted by a single third party - for platforms like Mastodon, users can run their own server (called an instance) that connects to the rest of the network [37]. For users who don't run their own Fediverse server, there is still more choice as users can decide which instance they wish to use. These aspects enhance user privacy at a foundational level; peer-to-peer platforms also have potential as data is not tied to a particular server.

Spam, bots & poor content personalisation

With decentralised networking infrastructure, it may be more difficult for bots to spread spam and misinformation so easily as Federated instances don't have to interoperate with every other instance in the network. The same goes for peer-to-peer networks, but at an even more granular level - content spread by bots would need to find their way to individual peers rather than an easy to access central hub, without an easy way to artificially boost engagement for example.

Decentralisation also reduces opportunities for automated content personalisation. As the data on decentralised platforms is more fragmented, it's difficult to implement such systems. Although this seems like a disadvantage, it does prevent issues like automated recommendations of extremist content. Plus, users can still cater their content consumption to what they care about - but there is more control and transparency in how this works than fully automated personalisation algorithms.

Targeted harassment & cyberbullying

As mentioned previously, decentralisation has potential to slow down or reduce the spread of bots and misinformation. This also applies to targeted harassment and cyberbullying. Federated instances can block entire peer instances, which means nefarious communities can be prevented from engagement, while peer-to-peer platforms could make it easier for users to block harassers and potentially "hide" themselves without impacting the rest of their social network. Small-scale communities could also enable more effective moderation, unlike centralised social media platforms controlled by a single organisation which typically make all the decisions when it comes to banning or suspending users.

Security flaws, bugs & outages

Once again, decentralisation has the upper hand over centralised platforms even if we only consider the impact of security flaws. If a large, centralised social media platform has a security flaw that enables attackers to access user data, it's possible that all user data is compromised. However, for an attacker to do the same on federated platforms would require attacking multiple servers. Likewise for peer-to-peer networks - if, somehow, an attacker manages to breach a single node in the network, the impact of that on the whole network is likely small.

2.5 Data Anarchy: An Alternative Data Storage Model

I propose a data storage model for social media where instead of storing data on a server (centralised or Federated instance), or even storing data in a public distributed database like IPFS [38], instead data is stored primarily on the user's own device. In other words, data such as posts and messages produced by a user are stored on their device and only distributed to peers the user wants to share with. In other words, there is no entity governing data other than the user - data anarchy.

Advantages

This is a radically different approach to most data storage models, even decentralised ones as in this model devices do not need to remain online for data to persist. If some peers were to go offline, none of the user's data would be lost, which is not guaranteed by technology such as distributed hash tables. Data need only be transferred to peers that the user grants access to by default, and public content could be propagated by peers automatically forwarding content if necessary. As a result, such a platform seems inherently more performant which would make it suitable for deployment on mobile devices. With the proliferation of smartphones, better internet speeds [39] and greater device storage capacity [40] this kind of platform has potential to scale up well.

Additional advantages of this kind of data model include being able to delete or modify existing data, which is not possible with distributed blockchain models and is not necessarily easy to do with distributed hash tables. Of course, it is possible that a peer might have some data a user has shared with them after the user has deleted their own copy of the data, but in such a system peers would be discouraged from keeping data for a long period of time anyway as it takes up unnecessary space. As a result, it may be the case that users only store the data they really care about (such as their own original posts and messages) and temporarily cache other data.

Another advantage of this model is that servers do not need much permanent storage space, cutting operating costs. As various operations that are typically carried out by servers in centralised social media platforms can instead be carried out on client devices, there are less requests and less processing to deal with.

Disadvantages

One concern about this kind of anarchic data structure is whether such a platform can be effectively moderated - for instance, illegal content, misinformation and extremism may be able to spread on such a platform without interference due to the lack of any central authority. However, it should be noted that this is already a problem faced by chatting apps such as Telegram [59] and Discord [60] - it is not a unique issue. Implementing the data storage model at scale requires careful design as ultimately, the onus of moderation shifts from a third-party (as in centralised or federated platforms) to individual users. Therefore it is vital to ensure that users are provided with intuitive, easy to use or potentially user-configurable automated moderation tools to ease the burden.

A disadvantage of relying on the user's own device to store data is that if the user loses their device or damages it beyond repair, the data cannot be recovered and the user would have to start over on the platform. As centralised social media platforms host the data, they do not face this problem - as long as the user remembers their username and password (or at least has access to their email), it is generally possible for users to access their account even when they change devices. The lack of a built-in backup in this sense means that users are at higher risk of losing their data with this anarchic data storage model. However, there are ways to mitigate this; features could be added to such a platform that enable users to transfer a copy of their data to store on another device as a backup, either automatically or manually for example.

Another problem with this data storage model is that there is no central authority available to authenticate users and issue identifiers. This means that the platform will have to generate a sufficiently large random ID for users that minimises the likelihood of coming across a duplicate ID. Fortunately, UUIDs (Universally Unique Identifiers) already exist exactly for this kind of purpose [41].

3 Specification

3.1 Overview

This specification serves as a guide for developing a prototype social media platform that at least meets the set of MVP priority requirements, as well as meeting additional stretch goal requirements. This also includes identification of risks with mitigation strategies, and defining an abstract set of system architecture components to build.

Prototype related terminology

This specification introduces terminology that is relevant only in the context of the prototype; for this reason the terms have not been included in a separate terminology or glossary for the project as the meanings may differ depending on context. Some of these terms may not be used in the app user interface for similar reasons, as the context matters and so instead a range of synonymous terms may be utilised to improve the user experience at times (such as “user account” or “peer” instead of entity).

- **“The platform”**: Platform, or “the platform” when no explicit context is provided refers to the prototype developed for this project.
- **Entity**: Entities are analogous to user accounts in traditional social media platforms. In the app itself, these may also be referred to as users, accounts, groups, pages or other terms.
- **Peer:** A general term to refer to a user or entity on the platform, sometimes in the context of being users or entities other than the user being discussed.
- **Peerway:** The name of the platform.
- **Mutual**: Peers which have agreed to be (or have been accepted as) part of the group of peers associated with an entity. These may also be referred to as friends, members or other terms depending on the context.
- **Subscribe/Unsubscribe**: Subscribing is where an entity adds another entity to be used as a source of posts in the main activity feed. Unsubscribing is where an entity is removed as a source of posts in the main activity feed. These terms may also be referred to as following or unfollowing respectively, especially in the app itself.

3.2 Requirements

This project is non-profit and open source with a focus on free access for users so there are no business requirements - the focus is placed entirely on the end users.

TABLE 3.2.1: Functional requirements

ID	Requirement	Reason	Priority
F1	A profile with activity feed for each entity, including at least an avatar and name.	Users need a way to identify and follow other users, and profiles are a common feature in existing social media platforms, so they are an expected feature.	MVP
F2	Text-based messaging between at least 2 entities.	Messaging is a common feature of social media platforms, so it is an expected feature.	MVP
F3	Combined activity feed that shows posts from entities the user follows.	A system for showing content from a collection of entities is necessary for users to view posts from those they follow, which is a common feature in existing social media platforms, so it would be an expected feature.	MVP
F4	Message notifications.	These notifications help users respond to messages on the platform quickly and easily, an expected feature of real-time apps.	MVP
F5	Mutual entities.	Mutual entities can be used for groups and communities centred around a particular entity.	Should have
F6	Post interactions (reacting, sharing, commenting).	Interactions with posts enhance the social aspect of the platform.	Should have
F7	Posts and entities search.	A text-based search is useful for improving discoverability of content and other entities on the platform.	Could have

TABLE 3.2.2: Non-functional requirements

ID	Requirement	Reason	Priority
NF1	Content and data sharing permissions.	An important part of this project is putting users in control of content and data they share on the platform. Providing permissions allows users to more explicitly specify who can access their data.	Should have
NF2	Message and content caching.	When a user goes offline, their messages and content will not be available to retrieve. Therefore some caching should be used such that recent messages and shared content can be viewed at any time.	Should have
NF3	Peer authentication.	When a user interacts with a peer, they should be able to authenticate that the peer is genuine. This may not be possible in some scenarios such as first interactions, but at least from the point of trust establishment users should be able to verify peers they have previously interacted with are genuine.	Should have
NF4	Content distribution.	Combined with content caching, distribution means when a user goes offline their shared content can still be accessed by other users via users who have cached that content. This means content can be shared between users even when the source user is offline.	Could have

3.3 Risks

To assess the risks associated with development of the platform, we can consider the risks using PESTLE analysis. Although this project is free, open-source and non-profit it is useful to identify risks relating to these different areas in relation to end users and the development process.

Political risks

As decentralised applications and peer-to-peer networks do not have a single central point of governance it is difficult, if not impossible for governments to regulate such platforms. This is a serious concern of authoritarian policy makers, with countries such as China banning cryptocurrencies precisely because of the difficulty of regulation [42]. The built-in peer-to-peer and securely encrypted aspects of the prototype may result in a ban in some countries, as has occurred with many end-to-end encrypted apps and services like Signal [43].

Economic risks

There is fairly low risk associated with economics, as long as internet connectivity remains affordable to the masses. Supporting mobile devices as the main means of using the platform combined with the data storage model actually make the platform relatively affordable to use and run, as mobile devices are extremely common [44] and costs associated with data storage in servers are negated by design.

Social risks

The concept of decentralisation has sometimes been conflated with distribution [45], and certain distributed technologies such as Blockchain and cryptocurrencies which have attracted negative connotations [46]. As a result, society at large may not find such a platform as attractive as traditional centralised platforms. Attitudes over factors such as the lack of a central authority moderating content may put off some existing social media users.

Technological risks

Due to the lack of a centralised authority to authenticate users and provide unique identifiers, we have to rely on decentralised identification systems such as UUIDs and digital signatures to identify and verify peers directly. A concerning risk is identity theft. It would be easily possible for individuals to create fake accounts impersonating peers. However, this risk can be mitigated with careful platform design - for example, when two peers first interact with each other they can exchange certificates to authenticate each other in future meetings. Coupled with a privacy-first platform design where data such as posts are not shared publicly by default, it would be more difficult to carry out identity theft on the platform.

Other risks relate to the feasibility of the technical implementation of certain common social media features. For example, while peers are offline a user cannot retrieve the latest posts and messages or other data from those peers. Coupled with data storage and internet connectivity limitations, this has a very serious impact on the usability of the platform. However, as people worldwide are actively spending more time online using mobile devices [47] coupled with the continuous connectivity of mobile devices provided by wireless internet tech such as WiFi and 4G/5G increases the likelihood that a user will be online at any time. This reduces the likelihood of situations where users cannot retrieve data from peers, especially if features such as forwarding of data and content caching are implemented.

Legal risks

The data storage model for this platform ensures that user data is never shared with anyone who hasn't been granted access, and no third-party servers are storing data. This means that laws such as GDPR have little impact on this project, as any legal risks here tend to fall on the end users rather than developers. There are risks of illegal content being shared using the platform, but as this is only in control of the users the legal responsibility should usually fall on them.

Of course, there are still legal risks with deployment of the platform. If for instance the app is deployed to the Google Play Store, rules and regulations need to be followed there. Overall however the legal risks seem low impact and unlikely.

Environmental risks

The main risk associated with the environment is the energy consumption of the platform, which may risk driving up CO_2 emissions. Some decentralised technologies utilise proof-of-work algorithms, such as Bitcoin - which is notorious in part for its excessive energy usage [48]. This kind of wastage is not something you want in a social media platform which could potentially have millions or billions of users, and would have a serious impact on carbon emissions as more electricity needs to be generated. However, the platform being built does not have to rely on intensive networking technologies like proof-of-work based blockchains. It may be possible to design a minimally-distributed platform that has a relatively small carbon footprint with careful design, where network efficiency is improved by ensuring peers only send each other data when necessary. In this way peer devices would function a little bit like simple HTTP servers - responding to requests dynamically without users requiring a continuous connection to view the content.

3.4 System Architecture

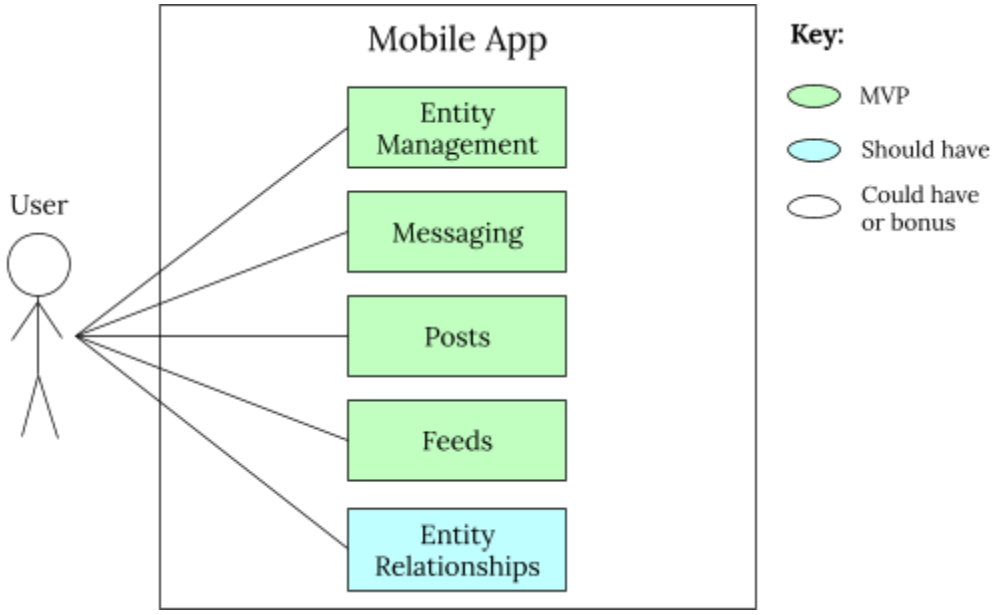

FIGURE 3.4.1: *Use case diagram for the mobile app*

Building a social media platform that functions as a viable generic alternative to existing popular social media platforms requires many components. For this reason, I have split the main use case diagram for the mobile app into 5 subsystems. For each of these subsystems at least one use case is required for the MVP, except for the Entity Relationships subsystem.

Entity Management

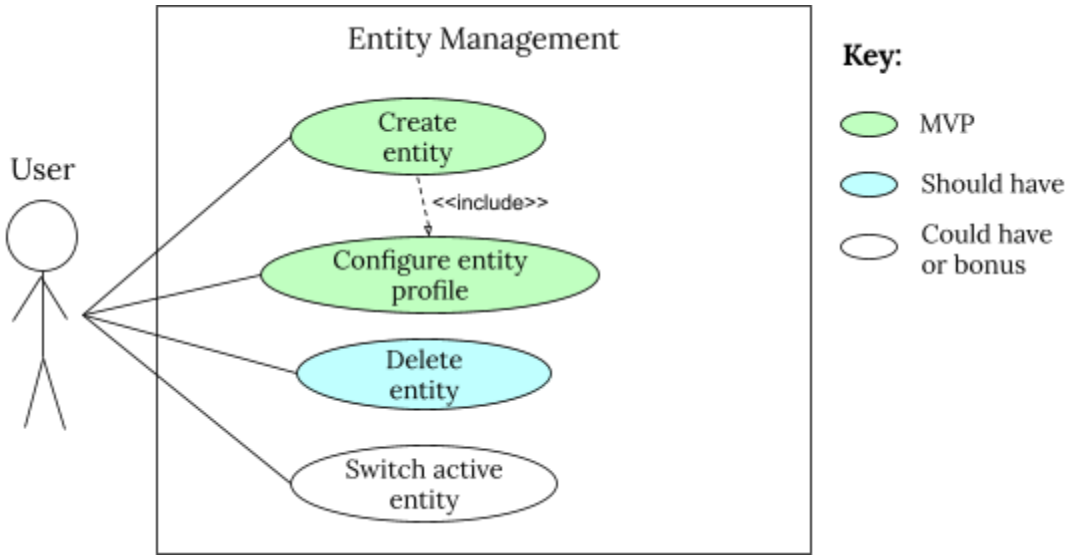

FIGURE 3.4.2: *Use case diagram for the entity management subsystem*

The Entity Management subsystem deals with high-level control over entities the user creates in the app. For the MVP, only a single entity can be created - but in a more advanced version of the app, users would be able to create and control multiple entities on the same device.

Messaging

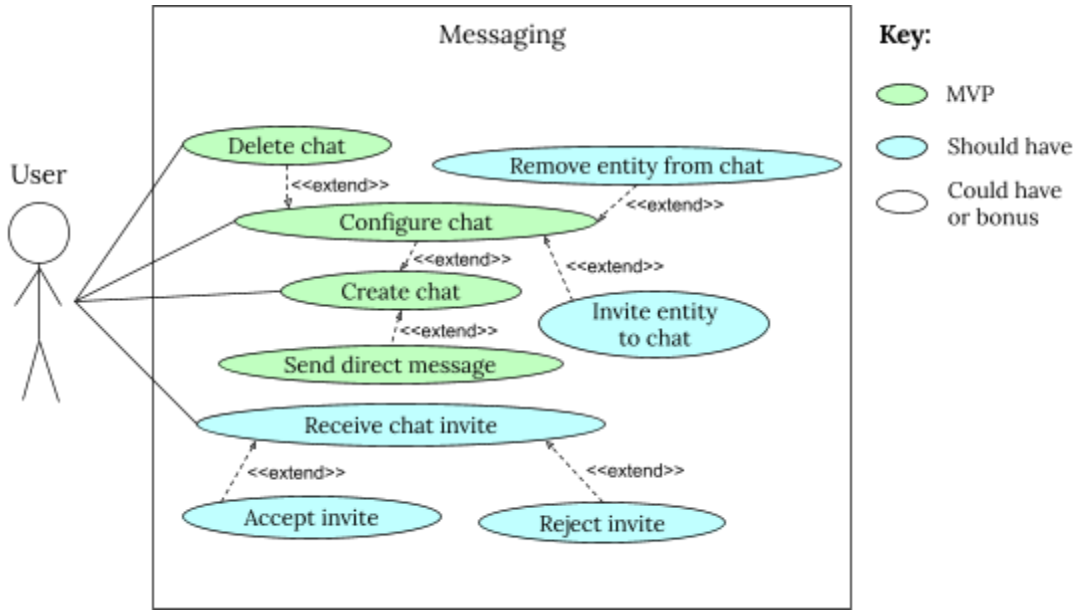

FIGURE 3.4.3: *Use case diagram for the messaging subsystem*

The messaging system allows for a user to send or receive messages from peers - in typical use cases this will be an entity created and controlled by a different user on the platform. In a more advanced version of the app, users will be able to “invite” entities to chats for messaging, which could include multiple entities, forming group chats.

Posts

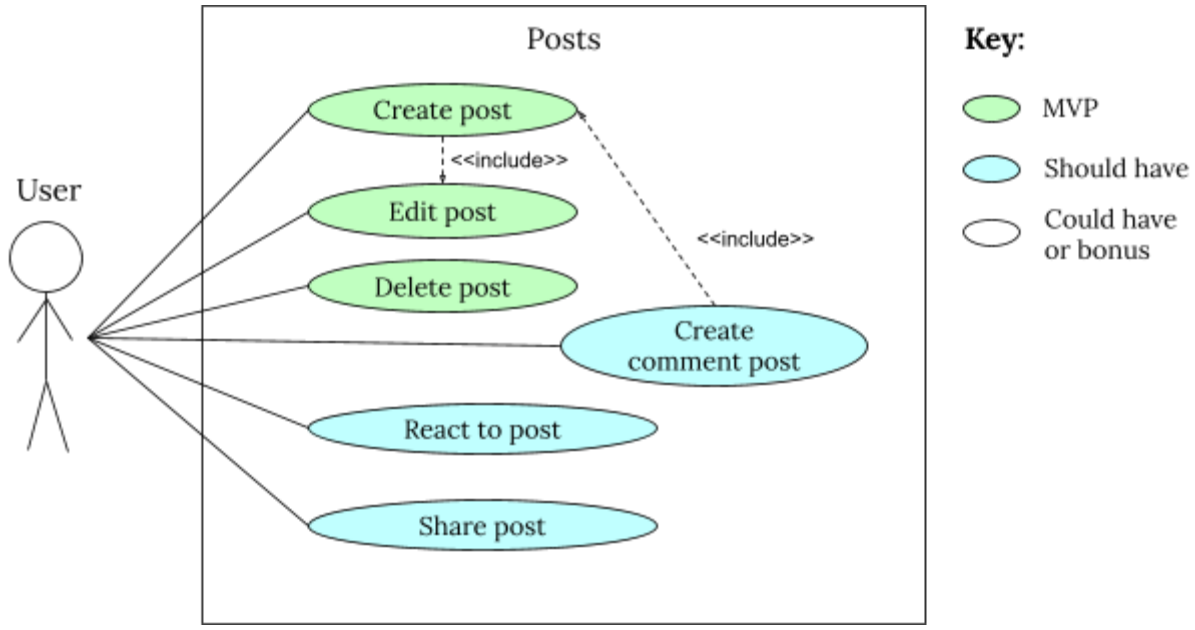

FIGURE 3.4.4: *Use case diagram for post interactions*

Although directly related to feeds, I have separated the use case diagram for posts. Posts themselves are a relatively complicated part of the mobile app, as their creation and viewing can include use of multimedia data and also offer various ways for entities to interact with them.

Feeds

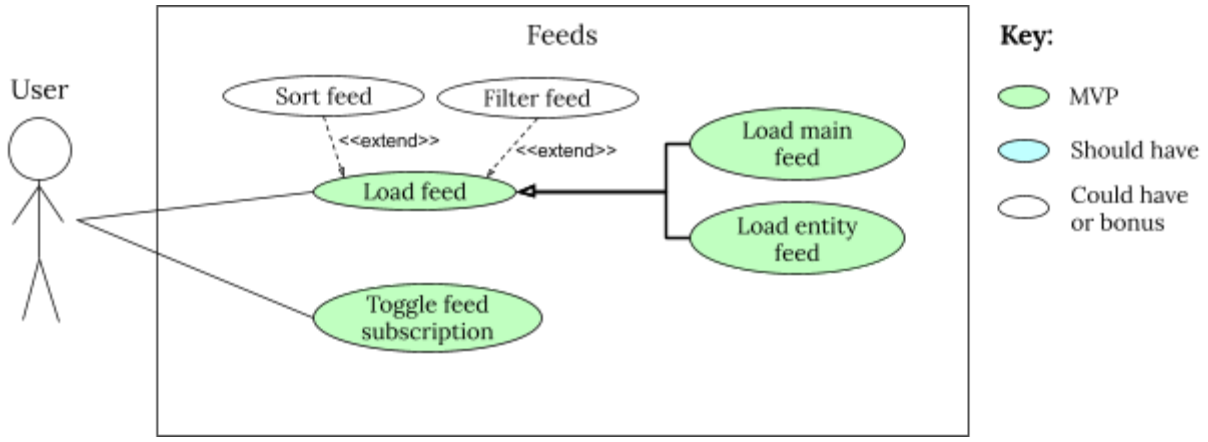

FIGURE 3.4.5: *Use case diagram for feed interactions*

Feeds are a key component in the app because these will enable users to view and interact with posts on the platform. These feeds will be implemented in the form of a scrolling “infinite” list. In other words, when you scroll to the end of the list, additional content is loaded and added to the list. It is not truly infinite, rather it is limited by the number of posts that are available to the feed. However, to the user who may have many posts shown in their feed over time, it may appear “infinite” as the user could scroll for a long time without running out of posts.

There are two key types of feed for the app; firstly, there are entity feeds. These feeds are displayed on entity profiles and show posts created by those entities. The other key type of feed is the main feed that combines the feeds of multiple entities into one. This will only include feeds of entities that the user has subscribed to.

Entity Relationships

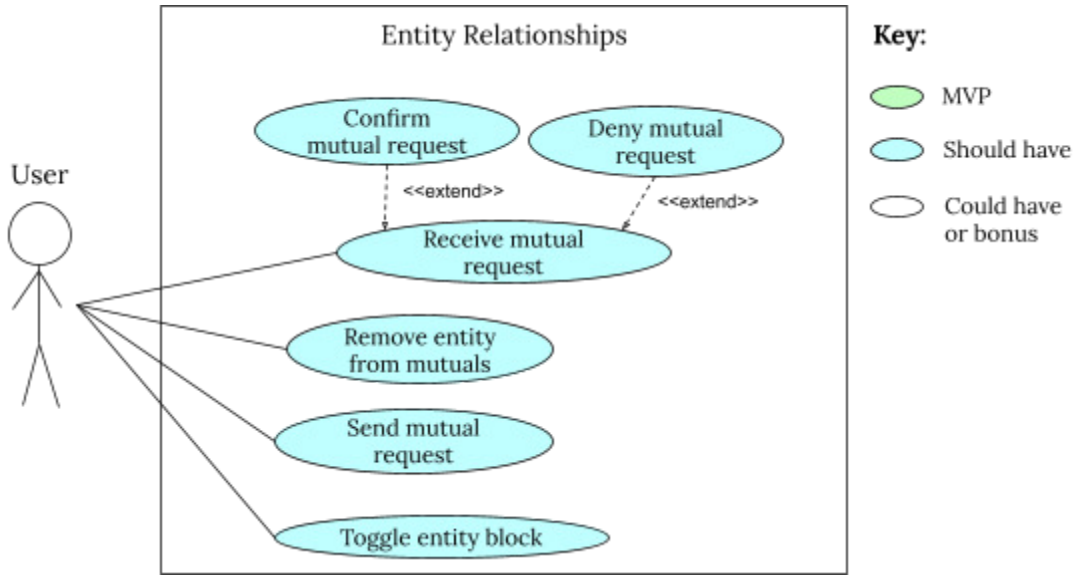

FIGURE 3.4.6: *Use case diagram for managing entity relationships*

The entity relationships system isn't a core component of the MVP, but it will be a key addition to the app in a more advanced version. This system gives users a way to relate entities, forming groups or "friendships" with other entities. This is a very important part of community building.

4 Platform Design

4.1 Peer-to-peer Networking Considerations

Wireless networking

WiFi, Bluetooth and cellular data systems all rely on physical broadcasting of information via electromagnetic waves. Of these, only Bluetooth is really suitable for a pure peer-to-peer network as it does not require any centralised routing infrastructure, with the added benefit that all modern mobile devices ship with Bluetooth support [49]. Bluetooth is not a long range protocol with 10 metres typically cited as a reliable working distance - though it is heavily affected by environmental conditions, so in practice it varies a lot [50]. However, this variability in range is not ideal and usage would be limited to specific locations - you simply can't connect to someone in Australia from the UK using Bluetooth for instance.

For situations where centralised infrastructure is unavailable, Bluetooth can provide connectivity where there would otherwise be none, which is great. Typically however, social media users can interact with others all over the world, not just locally, as the distance internet infrastructure covers is so much greater than standard Bluetooth chips can manage. Existing internet infrastructure is really the most efficient and best way to do that right now, so I will focus primarily on ways that a peer-to-peer network can utilise that infrastructure.

DNS

When using DNS (domain name system), internet clients must know the domain name of at least one server they can connect to (such as example.com). This is forwarded through DNS servers until one is found that knows the IP associated with the domain [51]. If an address is found, then it will be sent back to the client which can then form a connection with the server.

DNS at a glance seems like the solution - why not use domain names for nodes in the network? Unfortunately, domain names typically cost money [52]. Added to that, technology that exists to overcome the limitations of IPv4 such as NAT (Network Address Translation) [53] and dynamic DNS [54] makes configuration hard for users.

Despite the issues, there are projects out there such as Solid [12], which seeks to build upon existing internet infrastructure with a data storage model where users are in full control of their data and how it is shared. Unfortunately this is simply not very accessible to end users - unless you can run your own pod server (which requires technical knowledge and costs money to keep running with a domain name), users have to rely on third-party servers to host their data pods. This has already resulted in problems, with a popular Solid server failing back in 2020 [55]. Ideally, a decentralised peer-to-peer social media platform shouldn't require technical know-how or backwards step into a centralised third-party data storage model.

WebRTC

As our existing internet infrastructure and design is not well suited to physical peer-to-peer connections by default, the next best thing is use of centralised servers that act purely as intermediaries for setting up connections between peers. This is a compromise, but the data storage model can remain intact - where users store data on their own devices, rather than using the server to store data. This can be achieved using WebRTC.

WebRTC is a protocol for real-time peer-to-peer communication that is widely supported in browsers and React Native. It uses the ICE (Internet Connectivity Establishment) standard to do this, with STUN (Session Traversal Utilities for NAT) and TURN (Traversal Using Relay around NAT) servers, as well as a custom signalling server to set up the peer connections. For this purpose, NodeJS can be used to run a simple HTTP server which will connect peers and signal them. This approach is well established and used frequently in applications ranging from video calls to file sharing. This approach seems most appropriate for a peer-to-peer social media app as it enables rapid transfer of data directly between peers over the internet, so this is the underlying networking technology chosen for use in this project.

4.2 Signalling Server

Having settled on the use of WebRTC for establishing peer-to-peer connections, a signalling server is necessary. For example, in order to send a message to a peer, the user must be first connected to that peer - which cannot be easily done unless the IP address of the peer is known prior *and* the peer is actually accessible, which is often not the case due to NAT. A signalling server that users connect to independently can be used to overcome these obstacles, establishing direct connections between users with WebRTC.

Another reason for using a signalling server is such that users can discover new peers and access their public content, send messages and so on even if they have never interacted with each other before. This is important in activity feeds, where public posts are shared and the authors want these to be visible to peers who may not have interacted with the author before.

Furthermore, a signalling server could be used to send push notifications to peers. Push notifications are special kinds of notifications that can be received by users even if the associated app is not running, and even if the user's device goes offline they will receive the notification(s) when reconnected to the internet. This does require use of proprietary mobile network infrastructure such as Firebase Cloud Messaging or the Apple Push Notification service (for Android and iOS devices respectively), though fortunately these services are free. As an example of the role the server plays here, when a user sends a message to a peer who is unavailable, the server can be used to send a push notification instead which will be received by the peer when they become available again.

4.4 Communicating with peers

Once peers have been discovered on the network, we run into a new problem: Which peers should we connect to, and when? In a naive approach, one could connect to *all* peers in the network. That way, whenever a peer posts an update or sends a message, you'll be sure to receive it as fast as possible. However, in a large-scale social media network that could span millions or even billions of users who are interacting with each other, this clearly isn't sustainable. Ideally nodes in the network should follow a more classic client-server approach where the majority of the data is only downloaded when content is first requested, or there is an update. These are the only circumstances in which a connection to another peer should be required for the MVP.

Lazy connections

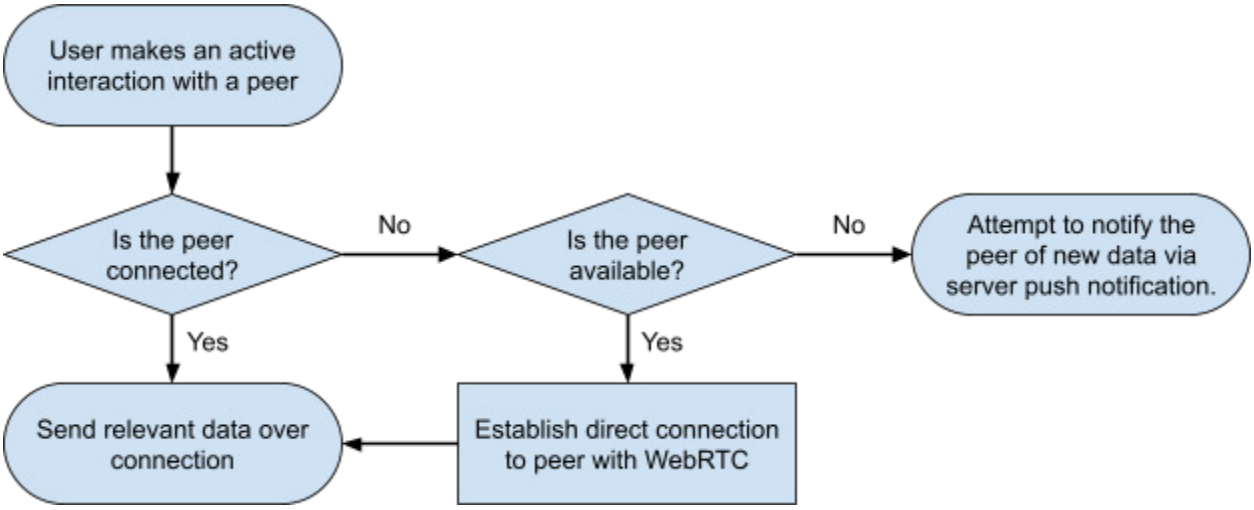

FIGURE 4.4.1: *Lazy connection process*

Ideally, the server should be doing as little work as possible. Messages and posts can be sent between peers by direct connection with WebRTC. However, any active connections will be severed when a peer closes the app; these disconnections need to be handled carefully. Additionally, we don't want to connect to every peer unnecessarily - not all peers will be sending the user messages or creating relevant posts when they are using the app. Therefore we should connect to peers in a lazy manner, where the user only connects to a peer when the peer has relevant new data to share or the user initiates an interaction with the peer.

Figure 4.4.1 demonstrates how this process would work. If a peer is not connected to the user, then a connection is established such that data can be sent directly. With the addition of push notifications, even if a peer is not available they may still be notified via the signalling server.

Offline peers & synchronisation

In an application like this when one or more peers are offline, other peers who are online can still connect to each other - but the offline peers will not receive any updates. For a social media application, there are various use cases besides group chats where users will be interacting with multiple other users simultaneously such as publishing a content post for followers to see. This means that the application must support these complicated scenarios where peers are not always available so not all peers can be kept up to date with the latest data, at least until they become available again.

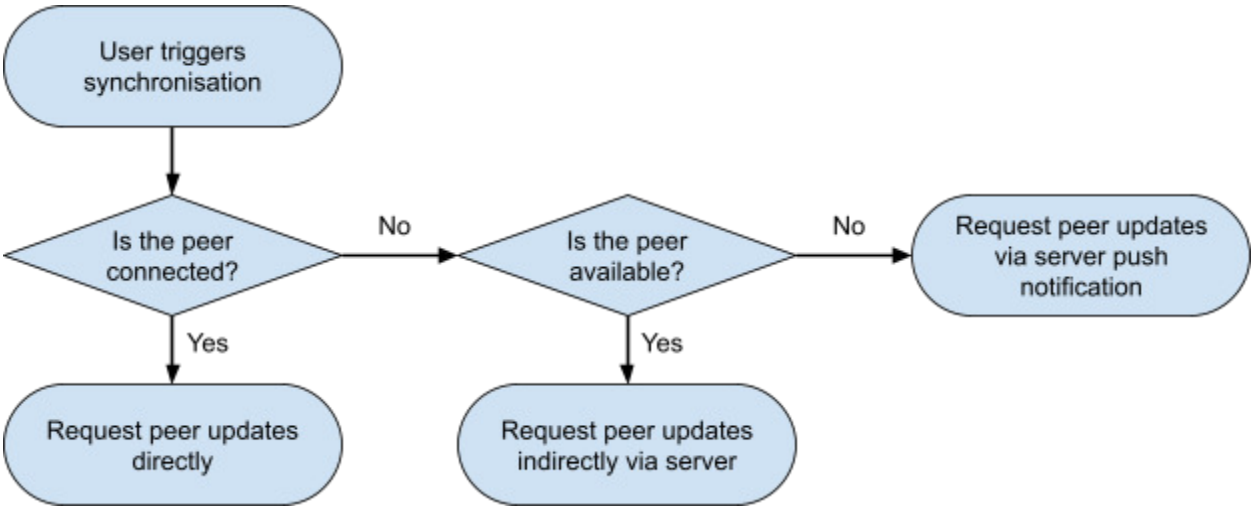

FIGURE 4.4.2: *Synchronisation process*

Fortunately, this can be solved with synchronisation. In essence, when a user comes online and first connects to the signalling server, we should send out a request to peers checking for updates as shown in figure 4.4.2. If those peers have any new data to share since the user was last online, then the peers can connect to the user and provide the updates. We don't want to connect to every single peer the user knows as this could be quite inefficient and waste mobile device resources. Instead, the signal server should be used to check the other peers for new data. This way, peers can respond and only create connections with the user as necessary - if a peer has no updates for the user, they simply don't bother to connect.

5 User Interface

5.1 UI Flow

The mobile app provides a user interface for the platform which should be intuitive to use for existing social media users and easy to learn for those who have never used such platforms. Due to the peer-to-peer networking and data storage model at the core of Peerway however, there are some complications to consider. Although many aspects of existing social media user interface design can be replicated to some extent, key differences in the way data flows through the platform make this difficult and so requires careful design. Please note that the colour scheme of the mockup diagrams does not necessarily correspond to the app.

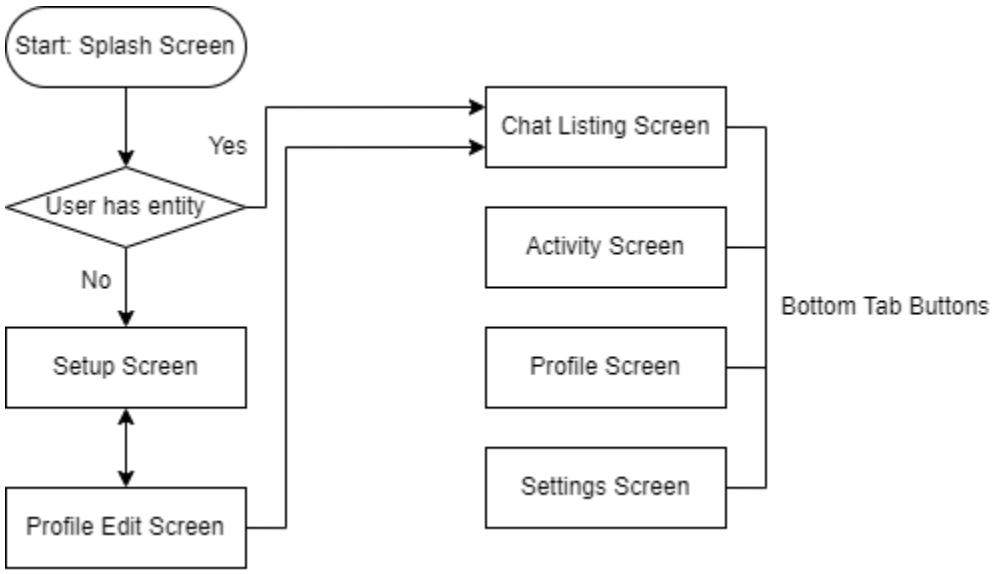

FIGURE 5.1.1: *Top-level UI flow*

Figure 5.1.1 shows the top-level UI flow, from the entry point of launching the app when it is not running. A splash screen displays the app logo while logic runs to determine whether the user has already created an entity or not. If not, the UI guides the user to create a new entity. Otherwise, the existing entity is loaded. In either case, both flow paths lead to the chats listing screen with bottom tab buttons granting access to the other main screens in the app.

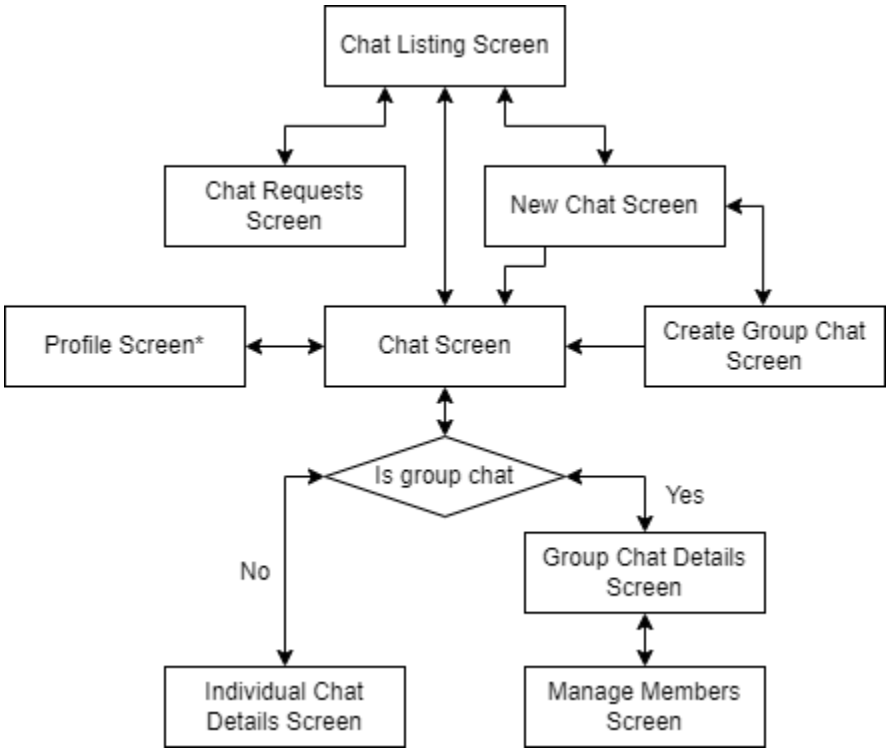

FIGURE 5.1.2: *UI flow from the chats listing screen*

The chats listing screen has the most complex UI flow, as you can see in figure 5.1.2. This is because there are a large number of features relating to chats compared to the other bottom tab screens, including chat requests, creating new chats and of course chats themselves which have information screens too. The asterisk is on the profile screen to note that the profile screen shown there is not necessarily the user's entity profile - it could show peer profiles in some cases. The same holds true for figures 5.1.3 and 5.1.4.

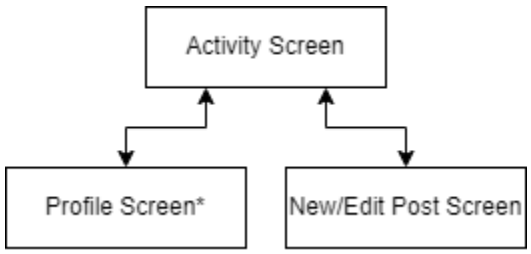

FIGURE 5.1.3: *UI flow from the activity screen*

The activity screen is relatively simple; just one screen is accessible at all times from the activity screen (besides the other tabs), which is the new post screen. However, if the activity feed on the activity screen displays any posts, the profiles of entities can be accessed via tapping on their avatars, or the user can edit posts they have created with the post editing context menu option.

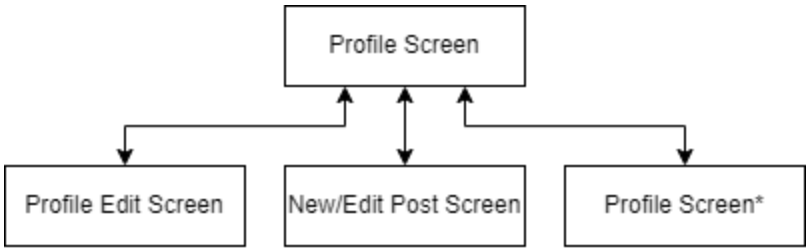

FIGURE 5.1.4: *UI flow from the profile screen*

As mentioned previously, the profile screen with an asterisk indicates that the profile screens for different entities can be accessed that way. In fact, you can theoretically recurse through profile screens infinitely as profile feeds display posts including replies in the app implementation. Though that specific use is not an intended use case, it's useful to keep as it allows users to step through several entities as they naturally come across them - for instance, if a user is trying to find several people they know of via the profiles of peers they already follow.

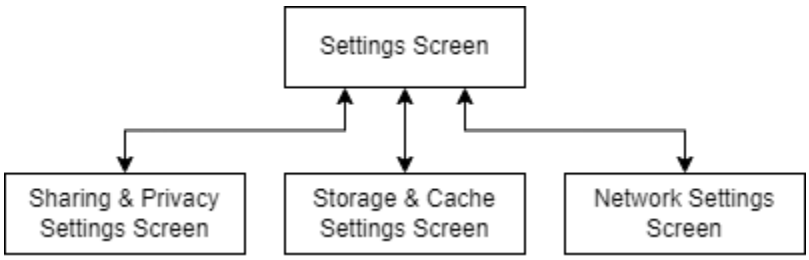

FIGURE 5.1.5: *UI flow from the settings screen*

The settings screen itself doesn't actually feature any settings - but it does provide access to specific groupings of settings, which is what the additional screens are for as you can see in figure 5.1.5.

5.2 Entity Management

Setup Screen

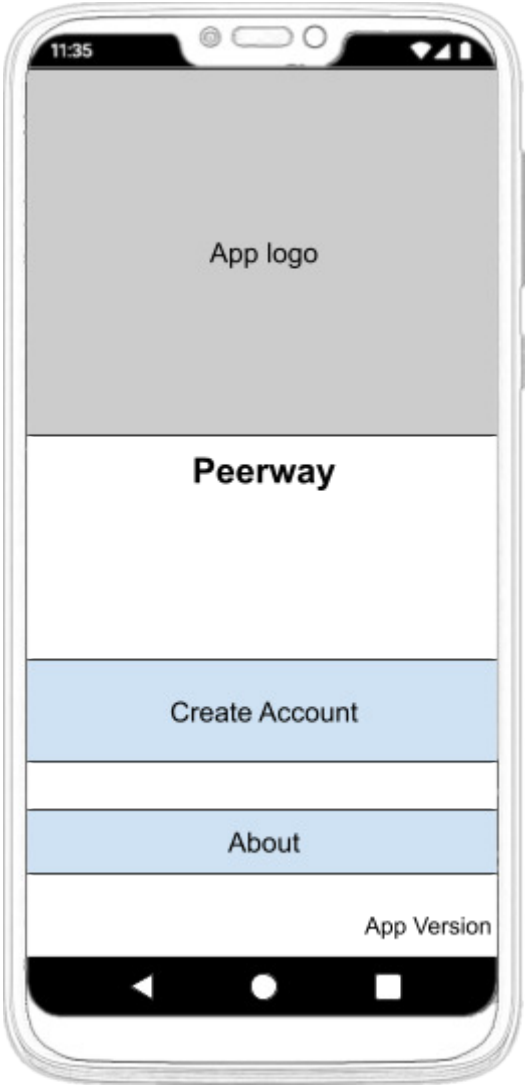

FIGURE 5.2.1: *Mockup of the app setup screen*

The screen mocked up in figure 5.2.1 is shown in the app when the user opens it for the first time or there is no previously created entity available. There is a button that enables the user to create an account (entity) named “Create Account” and a second “About” button that opens the project website in the default web browser. When the “Create Account” button is pressed, the profile edit screen is opened with all fields empty such that the user can create a local entity.

Profile Edit Screen

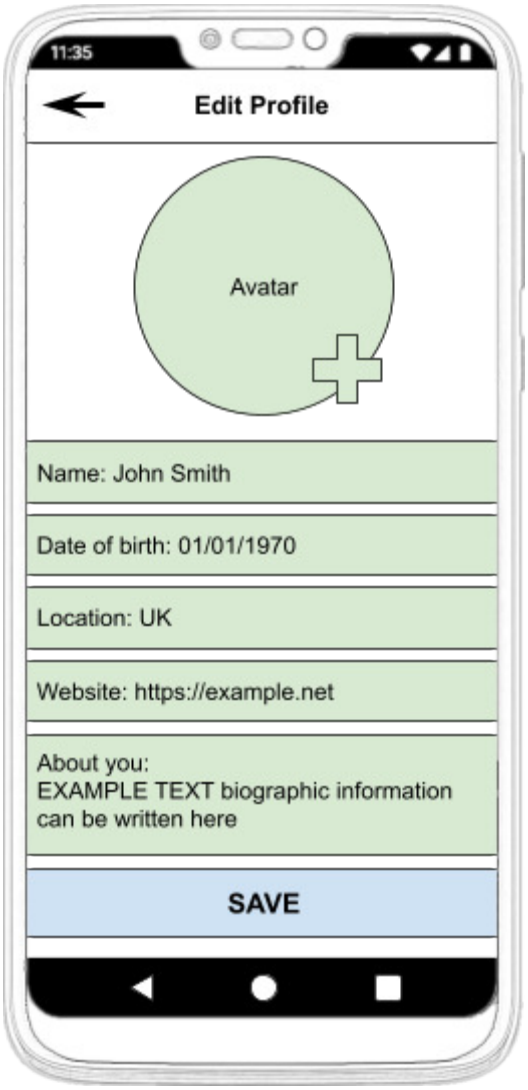

FIGURE 5.2.2: *Mockup of the profile edit screen*

Figure 5.2.2 shows the profile edit screen. The user is required to set the name of the entity. The user can optionally specify other profile information, including an avatar image, date of birth, location, website link and “About you” bio. The user can continue by pressing the “Save” button to store the updated entity profile locally, but this button will be disabled until the name has one or more non-whitespace characters. When choosing the avatar image, a third party library will be used that provides a user interface for image selection and cropping. If this screen is opened from the setup screen, saving the profile will create a new entity on the device.

App Settings Screen

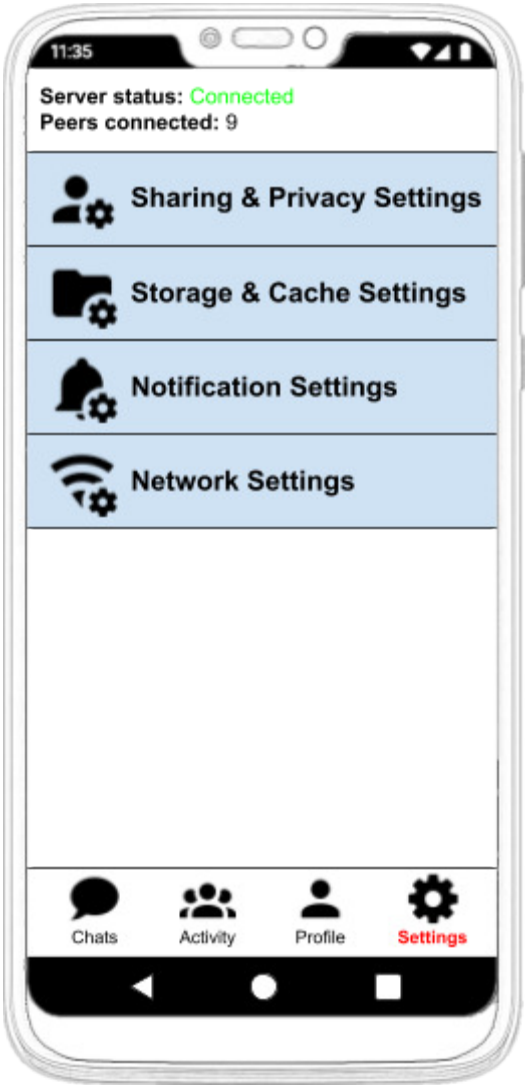

FIGURE 5.2.3: *Mockup for the settings screen*

The settings screen shown in figure 5.2.3 is a hub for all app settings. Each button opens the relevant settings screen. These include screens for:

- Global data sharing/privacy settings.
- Storage and cache settings relating to the local database instance and multimedia files.
- Notification settings relating to how notifications should be received.
- Network settings, such as the URL of the signalling server to use.

Sharing & Privacy Settings Screen

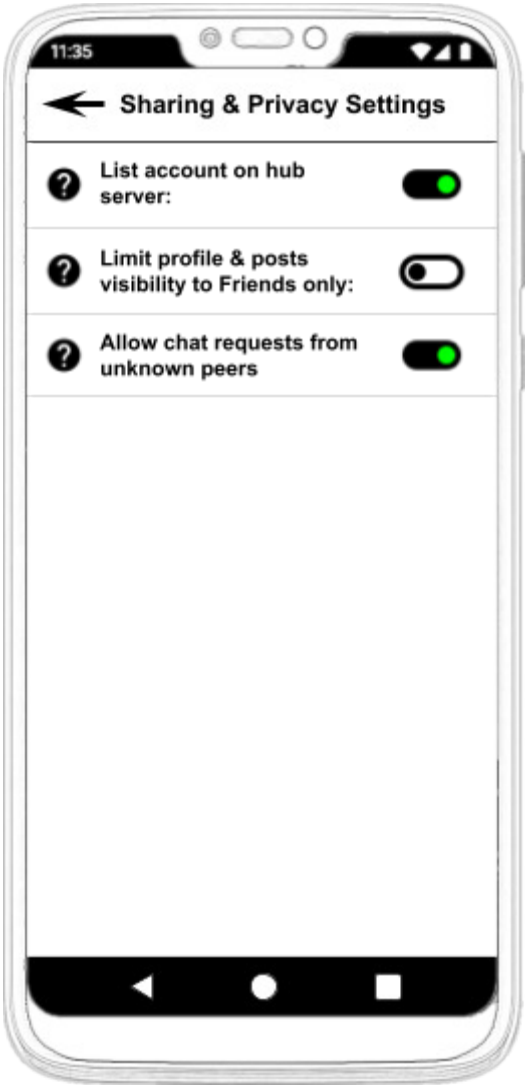

FIGURE 5.2.4: *Mockup for sharing & privacy settings screen*

The sharing & privacy settings consist primarily of toggle-based options, as you can see in figure 5.2.4. Enabling “List account on hub server” means that when peers search for a particular entity currently connected to the signalling server, the user’s entity avatar and name are listed in the results. Enabling “Limit profile & posts visibility to Friends only” makes all public profile information (except for name and avatar) visible only to mutuals. This setting also overrides the visibility of all public posts, limiting them to mutuals only or private until the setting is disabled. “Enable chat requests” allows peers that the user does not know to create a group chat or private chat request with the user.

Storage & Cache Settings Screen

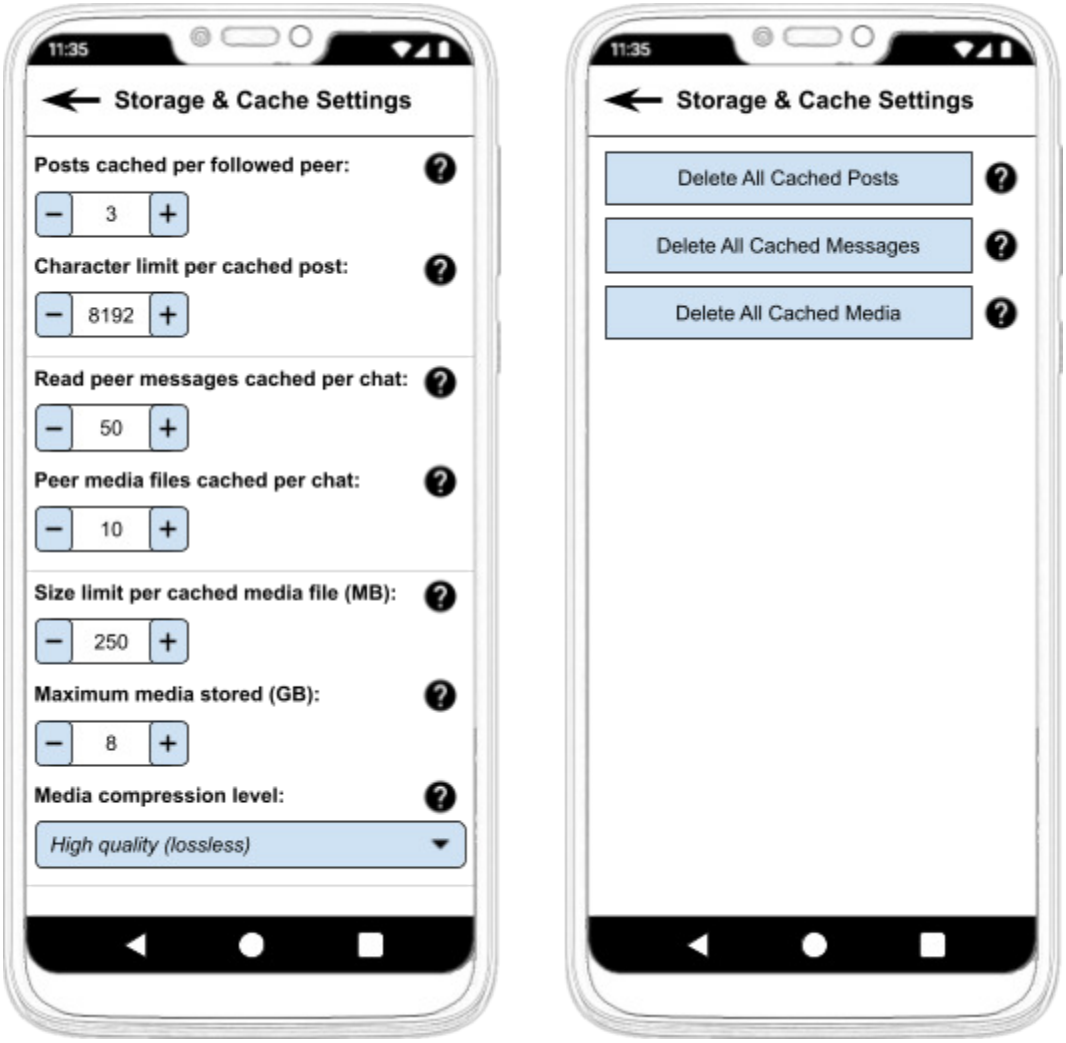

FIGURE 5.2.5: *Mockup for storage & cache settings screen*

The storage & cache settings pertain to how much data is stored on the user's device. As there are so many settings, the mockup is split into two images within figure 5.2.5 - the first showing the start of the screen, the second showing the end after the user has scrolled down. These settings are designed to be intuitive to users, but the help button provides more detail on what each setting does. The functionality of each setting is explained below in table 5.2.1.

TABLE 5.2.1: Storage and Cache Settings

Setting Name	Setting Type	Description of functionality
Posts cached per followed peer	Integer	Specifies the maximum number of posts that are automatically cached for each peer you are following.
Character limit per cached post	Integer	Posts over this limit are not cached.
Read peer messages cached per chat	Integer	Specifies the maximum number of messages cached that have already been read, per chat.
Peer media files cached per chat	Integer	Specifies the maximum number of media files cached per chat.
Size limit per cached media file (MB)	Integer	Specifies the maximum size of each media file in MB (megabytes).
Maximum media stored (GB)	Floating point (decimal)	Specifies the maximum amount of storage used for media files in GB (gigabytes).
Media compression level	Dropdown options	Cached media compression options: - High quality (lossless): No quality loss. - High quality (lossy): Media size is reduced more, but with some small losses in quality. - Low quality (lossy): Media size and quality are reduced significantly.
Delete All Cached Posts	Action button	Deletes all cached posts received from peers.
Delete All Cached Messages	Action button	Deletes all cached messages received from peers.
Delete All Cached Media	Action button	Deletes all cached media files received from peers.

Notification Settings Screen

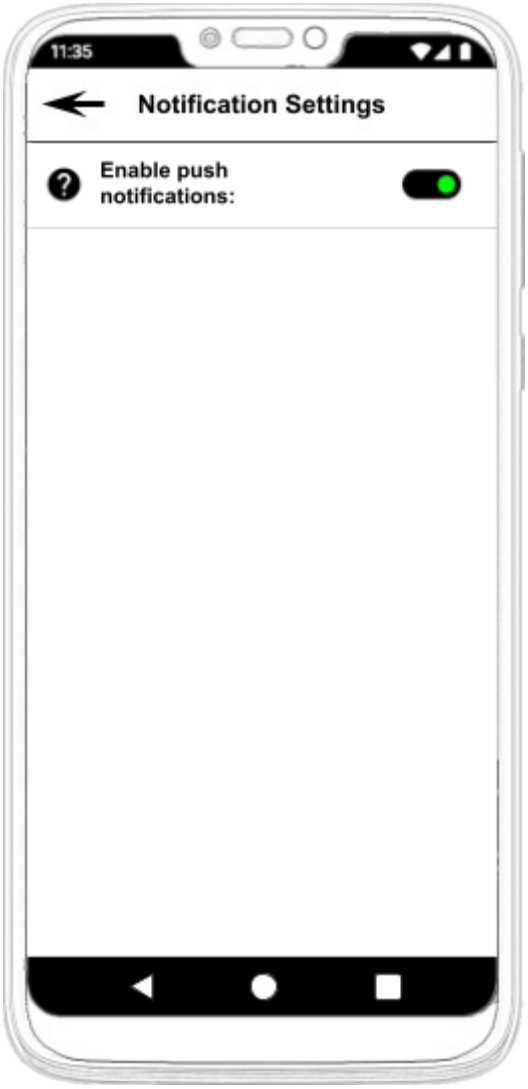

FIGURE 5.2.6: *Mockup of the notification settings screen*

As shown in figure 5.2.6, the notifications screen only has one option - the setting to enable or disable push notifications. General notification settings for the app can be managed externally, but as push notifications have an effect on the long-term usability of the app it is important to include it with a help button. The help button should show the following text: “Push notifications enable you to receive notifications even when the app is not running. It also enables relevant peers to retrieve data you have shared with them when the app is not running.”

Network Settings Screen

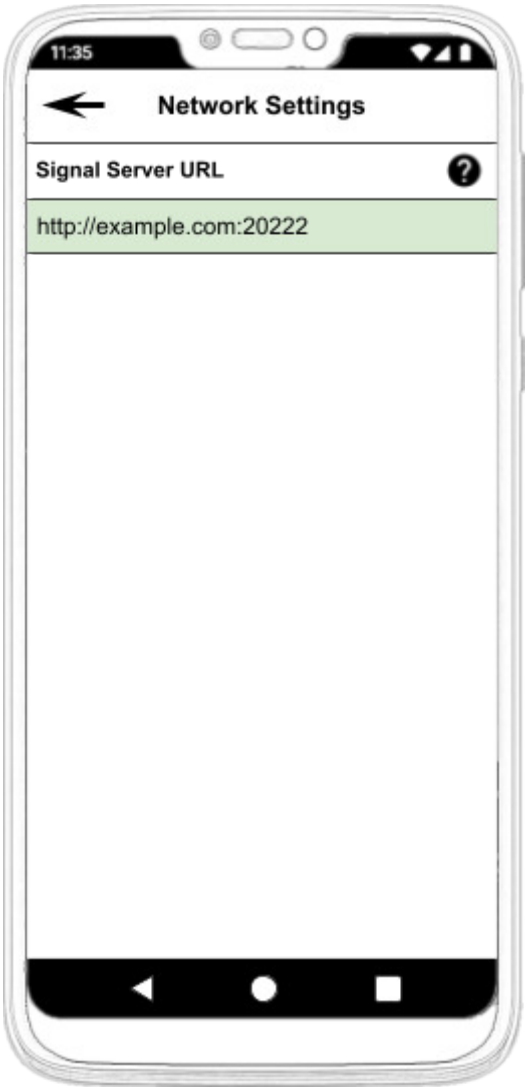

FIGURE 5.2.7: *Mockup of the network settings screen*

The network settings shown in figure 5.2.7 screen only features one setting that allows users to specify the URL of the signalling server they wish to use. There are no other network settings to specify here, though in future iterations these may include data saving settings that reduce the amount of data transferred (e.g. a setting to only download an uncompressed image when the user taps the image).

5.3 Messaging

Chat Listing Screen

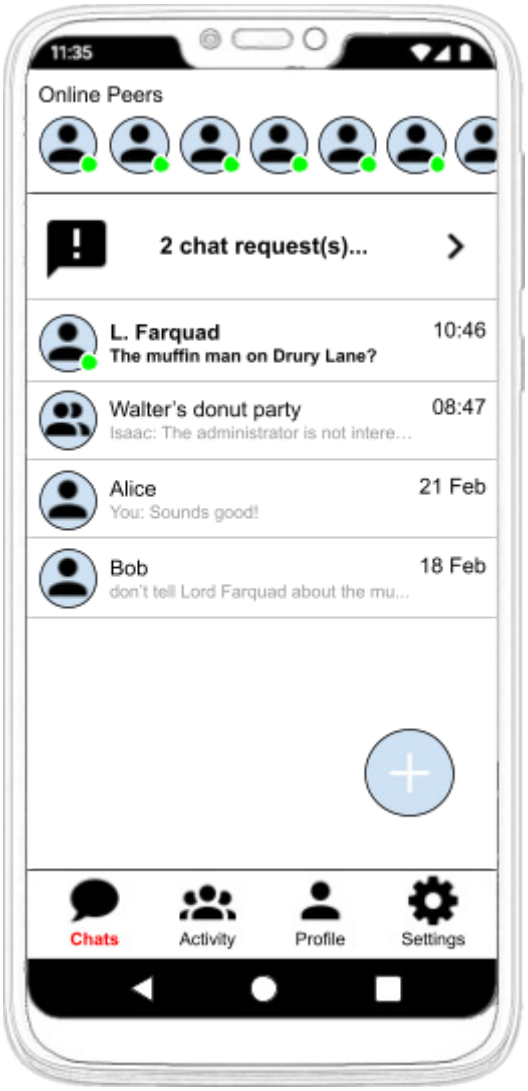

FIGURE 5.3.1: *Mockup of the chat listing screen*

The chat listing screen shown in figure 5.3.1 is the default screen shown to the user whenever they run the app once an entity has been created. At the top of the screen, there is a horizontal scrolling list of known entities that are actively connected to the signalling server (online peers). This is here for quick access to chats with available peers that the user interacts with. If the user has any unanswered chat requests, a navigation button is shown below the online peers section for viewing and responding to chat requests.

The rest of the screen comprises a scrolling vertical list of all existing direct message chats the user has created or been invited to, including group chats. The user can tap on a chat listing to open the associated chat. Each chat listing shows the last message sent (and who sent it), the time of the last message sent, the name of the chat and a chat icon. For private chats, the name of the chat is always the name of the other chat member and the chat icon is that member's avatar. For group chats a custom or default name and icon is set which is the same for all members of the chat. Chats that have received messages since last being opened are marked as unread (with the chat name and last message text in bold) until they are next opened. The user can long-press a chat to bring up a context menu that provides options to delete the chat, or toggle it marked as read/unread.

Note that this screen features tab buttons at the bottom. These tabs are accessible across 4 screens - the chat listing screen, the activity feed screen, the user profile screen and the settings screen. They provide quick access to the main areas of the app which users are expected to use often. The chats tab is placed first on the left and this is the default tab shown upon opening the app, by design. This is in contrast to social media platforms where content is put front and centre; while feeds are an important aspect, by nature of the peer-to-peer aspect the platform is intended to focus more on direct interactions between people than media consumption. To create a new chat or locate an existing private chat with another entity, the user can press the new chat button (plus symbol) in the bottom right corner above the tab buttons.

New Chat Screens

FIGURE 5.3.2: *Mockups of new chat screens*

The new chat screens shown in figure 5.3.2 allow the user to create a private or group chat If the user selects a peer in the initial screen, a private chat with that peer is created if it doesn't exist already and then it is shown in the chat screen. However, if the user selects "Create a group chat" from the initial screen, they will be presented with a checkbox list of peers which they should select from before pressing the "Create Group Chat" button to create a chat with those selected peers as members. Both screens feature a search bar for finding a specific peer. Only peers that the user has previously interacted with are listed.

Chat Screen

FIGURE 5.3.3: *Mockup of the chat screen*

Opened chats have features similar to most messaging applications. The bottom toolbar shown in figure 5.3.3 contains a file attachments button, a message text input section and a button to send messages. The chat displays messages in chronological order with the most recent messages at the bottom of a scroll view. As the user scrolls up the chat history, earlier messages are loaded until there are no more messages to load. The layout of this screen is designed to be familiar to existing social media users as much as possible. The user can open the chat details screen by tapping the "*i*" (information) button located at the very top right.

Private Chat Details Screen

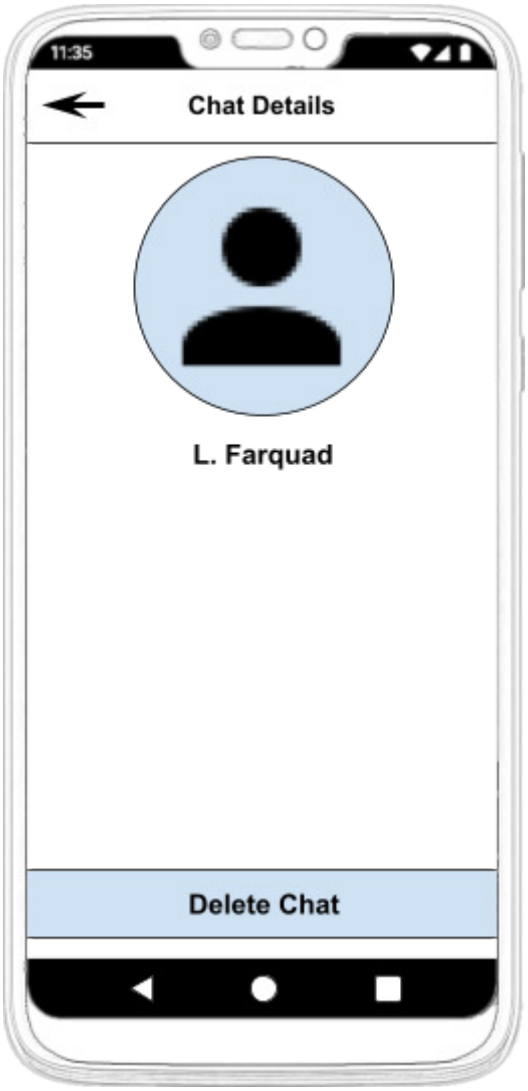

FIGURE 5.3.4: *Mockup of the private chat details screen*

The chat details screen for private chats (i.e. not group chats) shown in figure 5.3.4 is very simple. The peer's avatar and name are displayed, along with a button to delete the chat locally. Pressing the peer avatar image opens the peer's profile. There isn't much to configure in the details of private chats as these are intended to be private messages with a specific peer rather than a group of peers.

Group Chat Details Screen

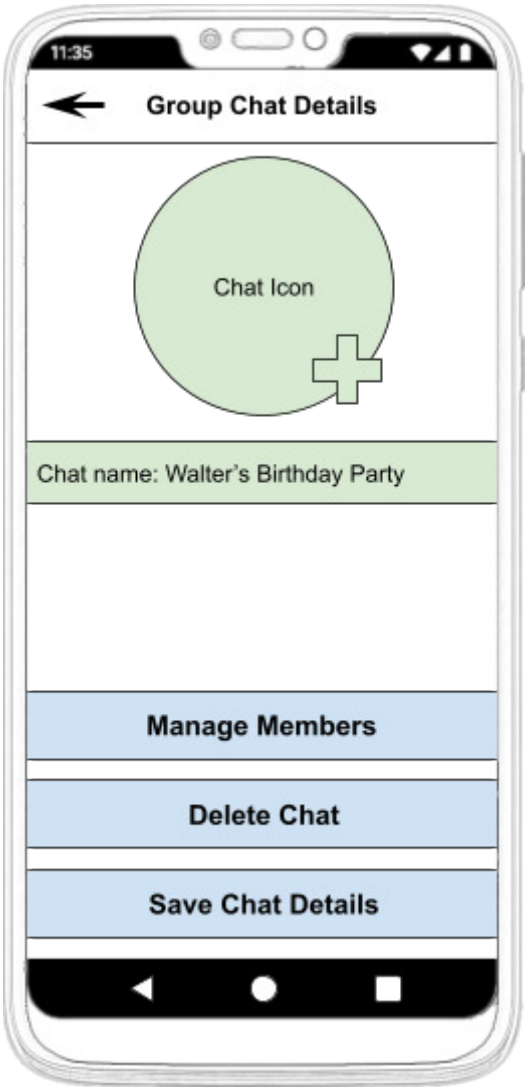

FIGURE 5.3.5: *Mockup of the group chat details screen*

For group chats, opening the details screen shows the chat name, chat icon and some action buttons as seen in figure 5.3.5. “Manage Members” opens a screen listing the chat members for selection. “Delete Chat” opens a popup asking the user whether they wish to delete the chat just for themselves (and leave the chat as a result), or request deletion of the chat across all members as well. If the popup is not closed and an option is chosen, a confirmation popup is shown before deletion takes place. Finally, the “Save Chat Details” button will be disabled unless the chat name or icon changes, in which case pressing the button will save those changes.

Manage Members Screen

FIGURE 5.3.6: *Mockup of the group chat manage members screen*

Figure 5.3.6 shows the screen for removing members of a group chat. Future iterations of this screen may add options such as giving administrator privileges to chat members.

Chat Requests Screen

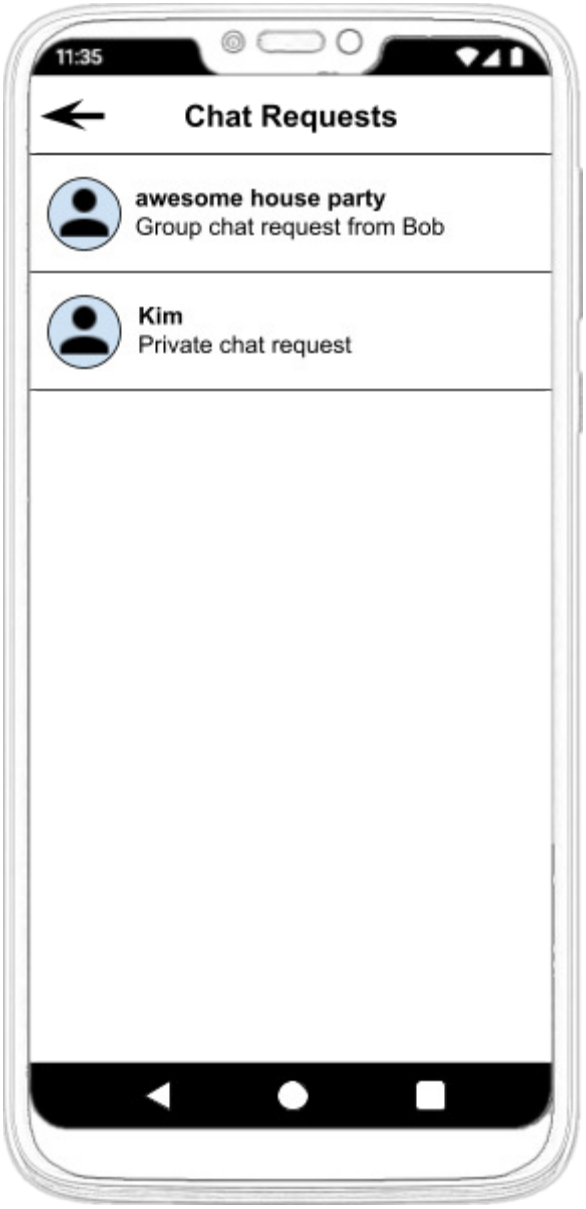

FIGURE 5.3.7: *Mockup of the chat requests screen with example requests*

The chat requests screen seen in figure 5.3.7 shows a list of chats that the user has been invited to. Tapping on one of these listings shows a confirmation popup, asking the user to accept or reject the chat request. Accepting a chat request implicitly registers the peer who created the request and issues that peer an authentication certificate. Next time the user interacts with the peer, the peer can be verified before the user carries out any other interactions with the peer.

5.4 Feeds & Posts

Activity Feed

FIGURE 5.4.1: *Mockup of the activity screen with empty activity feed*

The entirety of the Activity screen is an activity feed. This shows posts from peers the user follows, if there are any available. When the user scrolls down the feed, additional posts are loaded. The posts are ordered chronologically with newer posts at the top. In future iterations, additional UI elements may be added to filter and sort the feed. A button in the bottom right above the tab buttons opens the post creation screen when tapped.

When a peer followed by the user creates a new post while the user is viewing the activity feed, a button will automatically appear at the top of the screen that enables the user to scroll to the top of the feed so they can load and view any new posts.

FIGURE 5.4.2: *Mockup of the activity screen with example posts*

The user can tap the 3 dots button on a post to open the context menu. If the post is one the user authored, then they have the options to delete or edit the post. Tapping on the entity avatar in the top part of a post opens that entity's profile. Users can create a reply post with the speech bubble button in the bottom part of a post. Users can also "like" a post with the thumbs up button.

Profile Screen

Profiles show general information associated with entities. The profile screen also contains an activity feed. However, unlike the main app activity feed, profile feeds only show posts authored by the entity that the profile is associated with.

FIGURE 5.4.3: *Mockup of the profile screen with example posts*

Like the activity screen, the profile screen features a button in the bottom right for the user to create a new post. If the user is viewing the profile of their own entity, the “Edit Profile” button is shown in the entity information section which navigates to the profile edit screen. Otherwise, the button shows “Follow” or “Unfollow” depending on whether the user is following the entity. The bio section is truncated if it exceeds 3 lines, but the user can tap the bio to expand it in this case.

New Post & Edit Post Screen

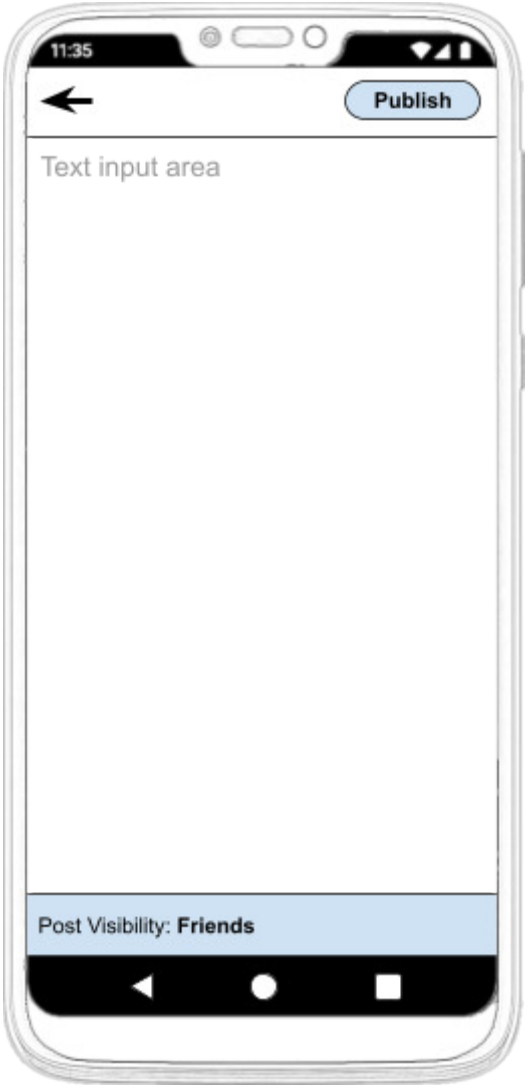

FIGURE 5.4.4: *Mockup of the new/edit post screen*

When creating a new post, or editing an existing post, the user can provide some text content and specify the visibility of the post using the dropdown. The visibility settings include, but are not limited to “Public”, “Friends” and “Private”. Public posts can be viewed by anyone, while only friends (mutuals) can view posts with Friends visibility and Private posts are only visible to the user. When the publish button is pressed, the post is saved in the local database and appears on the user’s profile. Only those who meet the post visibility criteria will be able to view the post.

6 Implementation

6.1 Overview

The platform has been developed entirely using JavaScript. The signalling server uses NodeJS while the Android app is a React Native project. There are quite a few benefits of using JavaScript; it is a very high level language where all objects can be easily serialised thanks to JSON, it is widely supported on many platforms and has an excellent selection of libraries available. Thanks to being just-in-time compiled it's also possible to quickly reload code, which is a massive benefit of React Native over writing in Java or native C++ code for Android.

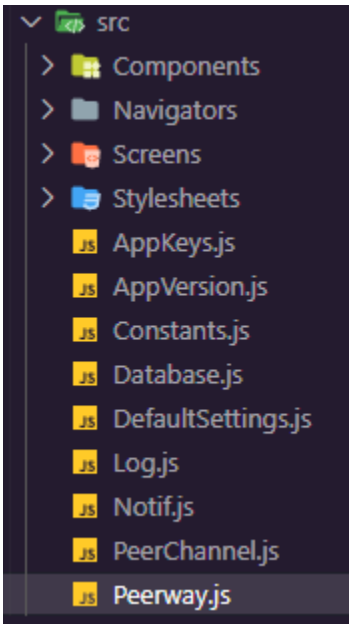

FIGURE 6.1.1: *Directory structure of the app source files*

I have split up code into subfolders consisting of React Native components, the UI navigation configuration, UI screens and React Native stylesheets (similar to CSS in some ways, but instead these are entirely JSON format). The rest of the source files make up the underlying systems powering the platform including database storage, WebRTC connection handling, chat message notifications, communicating with peers and the signalling server. A single instance of the class defined in Peerway.js is used throughout the project. Overall this is a very large project as the data storage model means the app has to do most of the heavy lifting.

On the other hand, the signalling server has very little code as it primarily acts as a hub for finding peers and connecting to them via WebRTC. A single file, SignalServer.js contains all of the server code. This is one advantage of designing a social media platform where data is stored only on user devices - the server doesn't need to do very much as a result.

The source code for this project is linked in Appendix A. Git was used as the version control system, and the project is available on GitHub under the MIT licence in the spirit of open-source development. Please note that line numbers shown in screenshots and referenced in text may not correspond exactly to the code in the latest commits. Also note that a number of TODO comments remain in the code - these are left in for future development but were not needed to produce the MVP.

6.2 Data Storage Systems

As the design of Peerway relies upon storing data on user devices, the way this data is stored is very important. There are 3 key parts:

1. Fast access key-value storage. This is useful for frequently accessed data such as user information and settings.
2. A relational database to allow storage and querying of large quantities of data including posts, messages, peer entity information and so on.
3. The native file system. This is necessary for storing large chunks of binary data like media files, as this data could reduce performance if it were stored in the database.

I found React Native libraries for each of these components. For the fast access key-value storage, I made use of MMKV storage developed by WeChat. This does exactly what I need - there's not much to say about it other than it provides very quick access to small amounts of data. I opted to use SQLite for the relational database, but I specifically used an implementation package designed to run quickly (react-native-sqlite-quick) due to the dependence of the platform on local data storage. The file system of course is already provided on Android, but I decided to use the RNFS (react-native-file-system) library as it provides all the file management utilities I need, from creating new directories to creating new files.

MMKV storage

```
11
12  export default class Database {
13
14      // MMKV storage instances for all entities, by entity ID.
15      static entities = {};
16
17      // MMKV storage instance for the current "active" entity.
18      static active = null;
19
20      // Local user data, tracking entities and general app settings.
21      static userdata = new MMKV({ id: "userdata", encryptionKey: AppKeys.userdata });
22
```

FIGURE 6.2.1: *MMKV storage instances in the Database class*

There are two main uses of MMKV storage in the app. The first is for storing basic information about any entities the user has created, primarily profile information such as entity names. These MMKV instances are stored in the `entities` object, keyed by entity IDs. The second is the `userdata` object, which is for storing app settings and the ID of the current active entity, i.e. the entity which the user was last using. Although the current implementation does not support use of multiple entities, in principle this means that the active entity could be switched easily in future. This is what the `active` variable is for - it's just a reference that gets set to the entity MMKV object instantiated in the `entities` object when the user creates an entity or the app loads the active entity specified in `userdata`.

```
284                 // Extract other state into profile
285                 let timeNow = (new Date()).toISOString();
286                 var profile = {
287                     name: this.state.name,
288                     dob: (this.state.selectedDate != "" ?
289                         (new Date(Date.parse(this.state.selectedDate))).toISOString() : timeNow),
290                     location: this.state.location,
291                     website: this.state.website,
292                     bio: this.state.bio,
293                     avatar: this.state.avatar,
294                     updated: timeNow
295                 };
296                 // Save profile in database
297                 Database.active.set("profile", JSON.stringify(profile));
298
```

FIGURE 6.2.2: *Example storing entity profile data with MMKV*

The MMKV storage library supports strings, booleans and numbers - most commonly strings are used, as they can be used to represent objects as well thanks to JSON serialisation. Figure 6.2.2 shows how the user's entity profile is stored this way after creating or editing the profile in the app (see Screens/ProfileEdit.js lines 284-297). The actual storage of the data just requires a single call to the MMKV `set` method, with the key and data specified. Figure 6.2.3 shows how data can be retrieved in a similar way on line 47, where the `active` entity is set by loading the corresponding MMKV instance (lines 42-44).

```
26      // Changes the current active entity storage slot.
27      static SwitchActiveEntity(id) {
28          if (this.db) {
29              sqlite.close(this.db);
30              this.db = null;
31          }
32
33          if (id == null || id == "") {
34              this.userdata.set("active", "");
35              this.active = null;
36          } else {
37              this.userdata.set("active", id);
38              // Load the MMKV instance if not already loaded
39              let key = this.userdata.getString(id);
40              if (!(id in this.entities)) {
41                  console.log("Loading active entity " + id + "...");
42                  this.entities[id] = new MMKV({
43                      id: id,
44                      encryptionKey: key
45                  });
46              }
47              this.active = this.entities[id];
48              // Open SQLite database
49              let location = RNFS.DocumentDirectoryPath;
50              let result = sqlite.open(id, location);
51              if (result.status) {
52                  Log.Error("Failed to open SQLite database " + id + " at " + location);
53              } else {
54                  this.db = id;
55              }
56          }
57          return this.active;
58      }
```

FIGURE 6.2.3: *A method for switching the active entity*

SQLite database

Figure 6.2.3 also shows the opening of the SQLite database. This takes a name (I use the entity ID for this) and a file directory, for which I use the standard Android document directory path provided by the RNFS library. However, setting up the database tables takes place in a different method when creating an entity.

```
60      // Creates a new entity storage slot and automatically sets the active slot.
61      static CreateEntity() {
62          // Generated ID for this entity, based on timestamp & MAC address.
63          let id = uuidv1();
64
65          // Randomly generated encryption key for the entity data.
66          // This way, if another user gets access to the entity data, they need the key to access it.
67          // In future, this could be a user password or pin instead of RNG UUID.
68          let key = uuidv4();
69
70          // Create new entity storage slot
71          this.entities[id] = new MMKV({
72              id: id,
73              encryptionKey: key
74          });
75          this.entities[id].set("id", id);
76
77          // Add entity to userdata.
78          this.userdata.set(id, key);
79
80          // Initialise the settings
81          Object.keys(DefaultSettings).forEach((key) => {
82              this.userdata.set(key, DefaultSettings[key].toString());
83          });
84
85          // Setup SQLite database tables
86          let location = RNFS.DocumentDirectoryPath;
87          let result = sqlite.open(id, location);
88          if (result.status) {
89              Log.Error("Failed to open SQLite database " + id + " at " + location);
90          } else {
91              // Setup the database tables
```

FIGURE 6.2.4: *The start of the method for creating a new entity*

When an entity is created, a UUID is generated for the entity ID as well as an encryption key for the MMKV instance as figure 6.2.4 shows. The key isn't much use in the current implementation as it's stored in the unencrypted `userdata` instance, but the idea would be that if the user wanted to transfer or copy their entity data to another device in future versions the key would be needed to access the copy.

```
200
201     // Execute some arbitrary SQL command (synchronous)
202     static Execute(sql, params=[]) {
203         //Log.Debug("Executing SQL command:\n" + sql);
204         let query = sqlite.executeSql(this.db, sql, params);
205         if (query.status) {
206             Log.Error("Failed to execute SQL command \"" + sql + "\":\n" + query.message);
207             return { success: false, data: [] };
208         }
209         return { success: true, data: "rows" in query ? query.rows._array : [] };
210     }
211
```

FIGURE 6.2.5: *A wrapper method for executing database queries*

The SQLite database tables are created after line 91 of figure 6.2.5. I shall not include a figure showing every table here due to readability but the code is commented with information regarding what the purpose of each table and field is in lines 91-186 of Database.js. Please note that some fields are unused as some features of the implementation were cut to meet project deadlines.

```
1026     // Handle updated peer data
1027     _OnUpdatePeer(from, data) {
1028         Log.Debug("Received update for peer." + from.id);
1029         let avatarExt = "avatar" in data.profile && "ext" in data.profile.avatar ?
1030             data.profile.avatar.ext : "";
1031         // Update in the database
1032         Database.Execute(
1033             "UPDATE Peers SET name=?, avatar=?, updated=? WHERE id=?",
1034             [data.profile.name, avatarExt, data.profile.updated, from.id]
1035         );
```

FIGURE 6.2.6: *Updating peer data in Peerway.js*

When it comes to actually executing SQL queries to find data, I created a wrapper method such that the data returned was always consistent and any errors would be logged. This wrapper method is shown in figure 6.2.5 and I use it in many places throughout the project to access and store data. Figure 6.2.6 shows an example query where the name, avatar file extension and synchronisation timestamp are updated upon receiving new profile data from the peer.

File system

```
321   else if (data.filename.length > 0) {
322     // Save as binary file (base64 string)
323     // TODO avoid gross base64 conversion. Should just be able to save directly to binary.
324     let content = this._pendingData.toString("base64");
325     // Save media types in media, other files in generic download directory
326     let dir = data.mime.startsWith("image/") || data.mime.startsWith("video/") ?
327         Constants.GetMediaPath(this._activeId) : Constants.GetDownloadPath(this._activeId);
328     let path = dir + "/" + data.filename;
329
330     RNFS.mkdir(dir).then(() => {
331         return RNFS.writeFile(path, content, "base64");
332     }).then(() => {
333         return RNFS.stat(path);
334     }).then((res) => {
335         Log.Info("Saved file from peer." + this.id + " at " + path + " successfully with size " + res.size);
336     }).catch((e) => { Log.Error("Failed to save data to " + path + " due to " + e); });
```

FIGURE 6.2.7: *Saving binary data received from a peer with RNFS*

The RNFS library provides utilities for managing files. Besides the SQLite database file, the only other files that the app has to deal with are media files such as avatar images. RNFS supports binary file reading and writing using base64 strings as React Native unfortunately does not have very good built-in binary data structure support. One of the primary use cases for RNFS to handle is storing peer avatar images. Figure 6.2.7 shows some code written in the _OnDataEnd() method of Peerway.js which saves some binary data received from a peer in a media or downloads directory.

6.3 Web Servers

There are a few kinds of servers that are required to facilitate peer-to-peer connections with WebRTC. These include the signalling, STUN and TURN servers. The signalling server is a custom server required for peers to find each other and connect. The STUN server is used to get the public IP of a peer. The TURN server acts as a fast relay between peers. Initially during development, I relied upon free STUN and TURN servers but had to resort to running my own as the free ones restrict usage. I am using an old Raspberry Pi model to run a coturn server, which functions as both the STUN and TURN servers. No programming was necessary to set this up, installing a package and setting up a basic configuration was all that was needed so I will not go into much detail on this. I also had to forward some ports on my internet router so incoming connections could connect to these servers.

Signalling server

I used a library called socket.io to handle HTTP requests to and from the signalling server. I set up a number of request handlers to handle user connections, disconnections, provide information about other peers connected to the server (including their name, avatar and temporary client ID), setting up WebRTC connections and entity synchronisation for peers who are not currently connected to each other over WebRTC.

```
269     // Relay a call request
270     socket.on("Call", payload => {
271         Log.Info(
272             "Relaying peer connection request to client " + payload.target +
273             " (entity: " + payload.remote + ") from client " + payload.caller +
274             " (entity: " + payload.local + ")"
275         );
276         io.to(payload.target).emit("Call", payload);
277     });
```

FIGURE 6.3.1: *Server request handler for initiating a call*

Creating a WebRTC connection between two peers is quite straightforward. The peer who wants to initiate the connection does so by sending a call request to the server as shown in figure 6.3.1.

```
279     // Relay an answer to a call
280     socket.on("Answer", payload => {
281         Log.Info("Answered connection request from client " + payload.target);
282         io.to(payload.target).emit("Answer/" + payload.local, payload);
283     });
```

FIGURE 6.3.2: *Server request handler for relaying an answer to a call*

This request is then relayed to the target peer, who upon receiving the request automatically responds with an answering request which the server relays back to the caller as shown in figure 6.3.2.

```
285     // This is part of the ICE process for connecting peers once a request is accepted.
286     socket.on('ice-candidate', incoming => {
287         Log.Info(
288             "Sending ice candidate to target client " + incoming.target +
289             " (entity: " + incoming.remote + ") from entity " + incoming.local
290         );
291         io.to(incoming.target).emit("ICE/" + incoming.local, incoming);
292     });
```

FIGURE 6.3.3: *Server request handler for relaying ICE candidates*

The peers then use the server to send each other ICE candidates - messages containing information about the different WebRTC connection configurations each peer can use. Figure 6.3.3 shows the server-side request handler that relays an ICE candidate from one peer to another in the connection process. Section 6.4 goes into more detail on how these WebRTC connections are created and configured on the client side.

6.4 Managing Peer Connections

Lazy connections

As detailed in platform design section 4.5, lazy connections seem to be the most efficient way of deciding when to connect peers together. I have implemented the pseudocode flowchart from figure 4.5.1 in Peerway.js.

```
677     // Send a request to a peer or list of peers, even if they're not connected.
678     SendRequest(id, data) {
679         let peer = this.GetPeerChannel(id);
680         if (peer.connected) {
681             peer.SendRequest(data);
682         } else {
683             // Try to connect before sending the request
684             let listener = peer.addListener("connected", (result) => {
685                 listener.remove();
686                 if ("available" in result && !result.available) {
687                     // TODO try sending as a push notification if possible
688                 } else if (result.error) {
689                     // An error occurred during the connection attempt
690                     Log.Error(result.error);
691                 } else {
692                     // Connection successful, send the request directly.
693                     peer.SendRequest(data);
694                 }
695             });
696             // Attempt to connect to the peer
697             this.ConnectToPeer(id);
698         }
699     }
```

FIGURE 6.4.1: *The core of the lazy connection process*

The SendRequest method shown in figure 6.4.1 is used for all requests sent between peers *except* synchronisation requests. If the target peer is already connected to the user over WebRTC, data is sent directly. Otherwise, the connection is created. Figure 6.4.2 shows an example of this method in use.

```
83                  Peerway.SendRequest(peer, {
84                      type: "peer.sub"
85                  });
```

FIGURE 6.4.2: *Sending a subscription request*

Figure 6.4.3 shows the code for handling a subscription request. Later in this chapter you can see where that subscription request handler is actually set up in figure 6.4.7.

```
1067    // Handle peer subscribing or unsubscribing
1068    _OnPeerSubToggle(from, sub) {
1069        Log.Debug("Received " + (sub ? "subscription" : "unsubscription") + " from peer." + from.id);
1070        if (sub) {
1071            Database.Execute(
1072                "INSERT OR IGNORE INTO Subscriptions (pub,sub) VALUES (?,?)",
1073                [this._activeId, from.id]
1074            );
1075        } else {
1076            Database.Execute("DELETE FROM Subscriptions WHERE pub=? AND sub=?", [this._activeId, from.id]);
1077        }
1078    }
```

FIGURE 6.4.3: *Handling a subscription request*

The only part of the original design missing from this lazy connection process is using the server to send a push notification if the peer is offline. Due to time constraints push notifications were not implemented, but in future development this could be easily added to the algorithm.

Synchronisation requests

Besides simple cases where two peers simply connect to each other when they need to send data, synchronisation deals with cases where a peer might be missing new data due to being offline when the data was first sent. Although push notifications were not, a pair of peers will always be able to stay up to date as long as they are at some point both online at the same time thanks to the synchronisation implementation. The logic for sending a synchronisation request is contained within Peerway.js, with the SyncPeers() method.

```
574         // Send off the sync requests
575         let peers = Object.keys(requests);
576         for (let i = 0, counti = peers.length; i < counti; i++) {
577             let id = peers[i];
578             let data = requests[id];
579             data.type = "sync";
580             if (options.ts) {
581                 data.ts = options.ts;
582             }
583             if (options.cachePostLimitPerUser) {
584                 data.cachePostLimitPerUser = options.cachePostLimitPerUser;
585             }
586 
587             // Not using SendRequest here as we don't want to force connection unless necessary
588             let peer = this.GetPeerChannel(id);
589             let logMessage = "Syncing peer." + id;
590             if (peer.connected) {
591                 Log.Debug(logMessage + " via existing direct connection");
592                 peer.SendRequest(data);
593             } else {
594                 Log.Debug(logMessage + " via server");
595                 // Try the sync request via the server
596                 this.server.emit("Sync", { target: id, data: data });
597             }
598         }
```

FIGURE 6.4.4: *Part of the SyncPeers() method, in the file Peerway.js*

This method takes an object which maps timestamps of the latest data to the particular peers they correspond to. The data includes the timestamps of the latest known entity profile updates, chat messages, posts, and also the current subscription state of the active entity in relation to the various peers the sync request is for. This data is then converted into individual synchronisation requests and sent to the peers via the server, or directly if they happen to be connected already. Importantly, if the peer making a sync request is not connected to the peer intended to receive it, no connection is formed between them as per the pseudocode flowchart in figure 4.5.2. Figure 6.4.4 shows just the part of the SyncPeers() method where the requests are actually sent - the rest of the method is concerned with converting the data into the sync request format.

```
885         if (didSync && !("ts" in data)) {
886             // Store time of this sync with the peer
887             let ts = (new Date()).toISOString();
888             Database.Execute("UPDATE Peers SET sync=? WHERE id=?", [ts, from.id]);
889             // Sync the specific peer back
890             let options = this.GetSyncOptions(
891                 "general" in data,
892                 "chats" in data,
893                 "posts" in data,
894                 "subs" in data,
895                 from.id
896             );
897             options.ts = ts;
898             this.SyncPeers(options);
899         } else if ("ts" in data) {
900             Database.Execute("UPDATE Peers SET sync=? WHERE id=?", [data.ts, from.id]);
901         }
```

FIGURE 6.4.5: *Part of the _OnPeerSync() method, in the file Peerway.js*

Once a sync request is received by a target peer, that peer compares the timestamps against their local data and checks to see if they have created any data that is newer - whether those are new posts, new messages, or anything else specified in the sync request. The method that does this is named _OnPeerSync() in Peerway.js; if there are any new data points then the target peer will respond to the synchronisation request, attempting to connect to the source peer and send the updates. Otherwise, the target peer doesn't need to do anything and no connection is required. Figure 6.4.5 shows code where if the target peer did need to respond with updates, then it will send a sync request right back at the original peer to check that it has all the updates it needs from that peer.

Setting up a WebRTC connection

The first step in forming a WebRTC connection is for an app user to initiate a call via the server. A wrapper method named ConnectToPeer() written in Peerway.js does this, setting up a callback to handle successful connection and initiating the connection. To enhance modularity of the code however the actual connection and configuration logic is all handled by a class named PeerChannel, written in PeerChannel.js. Each PeerChannel instance corresponds to a particular peer, keyed by entity ID. Figure 6.4.6 shows the wrapper method that retrieves a PeerChannel instance and starts the connection process.

```
397     // Attempt to connect to a specified peer
398     // Returns false if connectionState is "connected" or "connecting".
399     ConnectToPeer(id, onConnected=null) {
400         let peer = this.GetPeerChannel(id);
401         if (peer.connecting || peer.connected) {
402             Log.Debug("Already " + (peer.connecting ? "connecting" : "connected") + " to peer." + id);
403         } else {
404             // Setup connection handler
405             peer.onCallSuccess = () => {
406                 // Sync profile data first
407                 let query = Database.Execute(
408                     "SELECT * FROM Subscriptions WHERE pub=? AND sub=?",
409                     [id, this._activeId]
410                 );
411 
412                 this.SyncPeers(this.GetSyncOptions(true, false, false, true, id));
413 
414                 if (onConnected) {
415                     onConnected(peer.id);
416                 }
417             }
418             peer.Connect();
419             return true;
420         }
421         return false;
422     }
```

FIGURE 6.4.6: *The method used to initiate a WebRTC connection*

The GetPeerChannel() method is a getter that creates a PeerChannel instance for the given peer ID if it doesn't exist already. This method also configures the PeerChannel with various request handlers used throughout the app for things like messaging, responding to synchronisation requests and so on. Figure 6.4.7 shows each of these request handlers grouped by type of request. All of these requests are made as JSON strings under the hood, which are easily parsed into actual JavaScript objects.

```
271     // Get an existing peer channel, or create one with callbacks configured
272     GetPeerChannel(id) {
273         if (!(id in this.peers)) {
274             // First create a communication channel for the peer, then configure request handlers
275             let peer = new PeerChannel(id, this._activeId, this.server);
276 
277             // Peer authentication
278             peer.requests.cert.issue = (data) => this._OnCertificateIssued(peer, data);
279             peer.requests.cert.present = (data) => this._OnCertificatePresented(peer, data);
280             peer.requests.cert.verify = (data) => this._OnVerifyRequest(peer);
281 
282             // Chats
283             peer.requests.chat.message = (data) => this._OnChatMessage(peer, data);
284             peer.requests.chat.update = (data) => this._OnUpdateChat(peer, data);
285             peer.requests.chat.request = (data) => this._OnChatRequest(peer, data);
286 
287             // Media files request
288             peer.requests.media.request = (data) => this._OnMediaRequest(peer, data);
289 
290             // Peer sub/unsub and profile updates
291             peer.requests.peer.sub = (data) => this._OnPeerSubToggle(peer, true);
292             peer.requests.peer.unsub = (data) => this._OnPeerSubToggle(peer, false);
293             peer.requests.peer.update = (data) => this._OnUpdatePeer(peer, data);
294 
295             // Posts
296             peer.requests.post.publish = (data) => this._OnPostPublished(peer, data);
297             peer.requests.post.request = (data) => this._OnPostRequest(peer, data);
298             peer.requests.post.response.begin = (data) => this._OnPostResponse(data);
299             peer.requests.post.response.end = (data) => this.emit(data.type, data);
300 
301             // Synchronisation
302             peer.requests.sync = (data) => this._OnPeerSync(peer, data);
303 
304             this.peers[id] = peer;
305         }
306         return this.peers[id];
307     }
```

FIGURE 6.4.7: *The method used to get a PeerChannel for a given peer ID*

Figure 6.4.8 shows how the call to create a WebRTC connection begins. In order to get the correct peer from the server, the temporary client ID assigned to the peer when they are first connected to the server must be known. This is different to the entity ID, which remains the same once an entity is created. Therefore it is necessary to get this ID from the server using the “GetEntityMeta” request if the client ID is not already known.

```
125     // Attempt to connect to the peer with the specified server clientId.
126     Connect(clientId=null) {
127         this.connected = false;
128         this.connecting = true;
129         // Try and find the peer on the server just using the ID
130         if (!clientId) {
131             Log.Debug("Requesting entity meta for peer." + this.id);
132             this.server.emit("GetEntityMeta", { id: this.id });
133         } else {
134             this.online = true;
135             this._SendConnectionRequest(clientId);
136         }
137     }
```

FIGURE 6.4.8: *The method which starts the peer connection process*

Once retrieved, the connection process can continue as described in section 6.3 where a peer calls, the target peer answers and then they provide each other with ICE candidates.

```
350     // Creates a WebRTC connection object.
351     _CreatePeerConnection() {
352         // TODO: Allow customisation of these parameters in settings (maybe reuse signal server IP)
353         let peercon = new RTCPeerConnection({
354             iceServers: [
355                 {
356                     urls: "stun:" + Constants.server_ip + ":3478"
357                 },
358                 {
359                     urls: "turn:" + Constants.server_ip + ":3478",
360                     username: "peerway",
361                     credential: "peerway"
362                 },
363             ]
364         });
365         peercon.oniceconnectionstatechange = () => {
366             if (peercon.iceConnectionState === "disconnected") {
367                 Log.Debug("Disconnected from peer." + this.id);
368                 this.connected = false;
369                 this.connecting = false;
370                 this.online = false;
371                 peercon.oniceconnectionstatechange = () => {};
372             }
373         }
374         return peercon;
375     }
```

FIGURE 6.4.9: *The method which configures the WebRTC servers*

Before the peers can connect to each other using the ICE information, they need to be configured to know what servers they can use to connect. Figure 6.4.9 shows the method which configures the information about the STUN and TURN servers (both running with coturn on my Raspberry Pi in this case). This method also sets up a callback to handle disconnection of the peer, which updates the state of the PeerChannel instance.

6.5 Chats

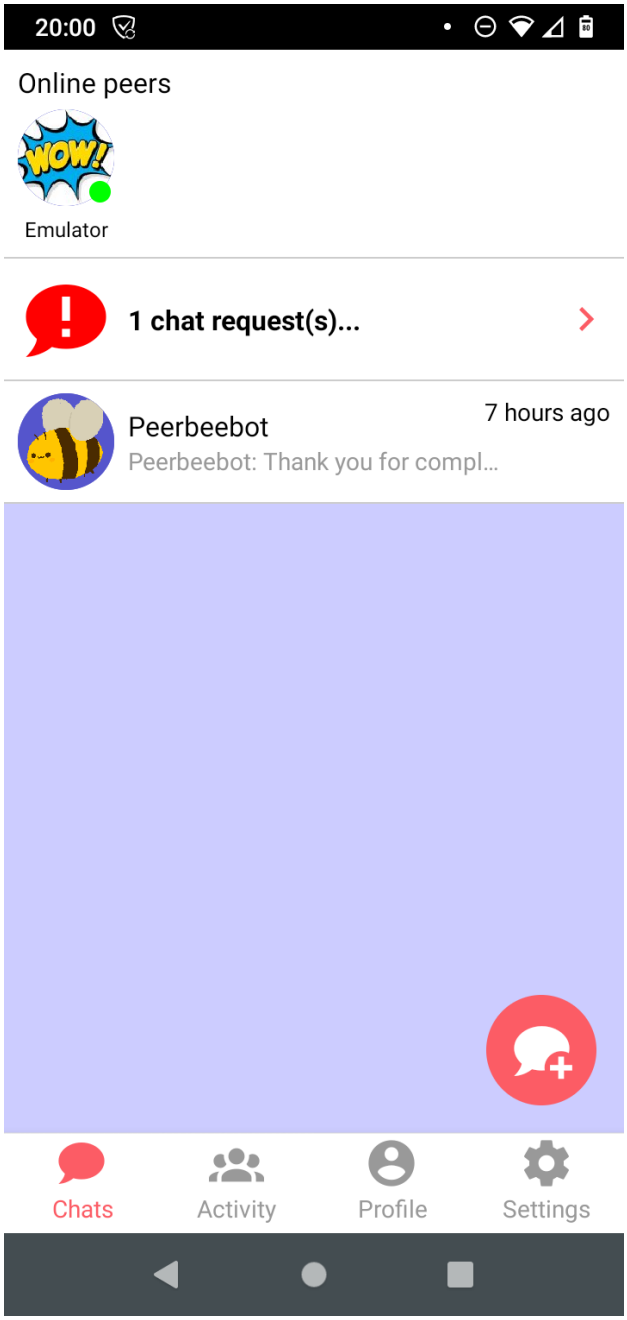

FIGURE 6.5.1: *Screenshot of the chats listing screen*

Moving onto general app functionality from a user standpoint, we'll first look at how messaging has been implemented. Figure 6.5.1 shows the chats listing screen, featuring an existing chat with an entity named Peerbeebot.

Peerbeebot is a fake peer used for the research study tasks detailed in chapter 7. The screenshot also showcases the chat requests button and online peers section. Note: from this point onwards, all app screenshots will only be shown in appendix B to improve readability.

Creating a chat & handling chat requests

Creating a private chat or group chat is implemented as per the mockups shown in figure 5.3.2, albeit without the entity search functionality. You can see both the new chat screens in figures 6.5.2 and 6.5.3, in appendix B. The search bars were cut from the prototype due to time constraints and that feature isn't necessary for creating a chat. When the user creates a new chat (whether a private or group chat), a new entry is added to the Chats database table locally and a chat request is sent to all chat member peers. When the peers receive this request, it is shown in the chat requests screen accessible via the chat requests button that appears on the chats listing screen, as you can see in figure 6.5.1.

As soon as the chat request is received by a peer, an entry in the Chats table of the local database is created. However, the peer will only be able to see the chat listed in the chats listing screen once they have accepted the request via the chat requests screen. Figure 6.5.4 shows the accept or delete chat request popup after a user has tapped on a chat request in the chat requests screen. Deleting a chat request deletes the chat entry, so the user cannot receive messages from that chat. Otherwise, accepting the chat request enables the chat so it can receive messages properly and is displayed in the chats listing screen. It should be noted that the "Online peers" section seen in figure 6.5.1 allows the user to quickly open (or create if not already existing) a private chat with the peer who's avatar they tap on. The online peers displayed are simply those peers the user has previously interacted with who happen to be connected to the signalling server. Currently this uses a server request to check the availability of those peers, but in future this can be optimised by using the relevant PeerChannel instance to check if a peer is online without needing to check the server every time.

Sending & receiving messages

Once a chat is created (or a chat request has been accepted), a user can open the chat from the chats listing screen by tapping on the particular chat entry they wish to open. Doing so will open the chat screen, which loads the most recent messages for the chat as you can see in figure 6.5.5. Rather than craft the entire chat UI myself, I utilised a library named react-native-gifted-chat which is a very customisable UI framework specifically built for messaging applications. I then plugged in code for loading messages and handling newly received messages from the relevant peer(s), as well as sending messages to the relevant peer(s). Messages are loaded chronologically with an SQL query to load the last 50 messages whenever the user scrolls to the top of the chat, with new messages displayed at the bottom like the mockup in figure 5.3.3.

The code for sending new messages to a peer relies on the Peerway.SendRequest() detailed in section 6.4, which attempts to establish a connection if not already connected. Handling messages from the peer relies upon a “chat.message” event emitted in Peerway.js upon a request of the same type, using a simple EventListener class I created (see EventListener.js). The implementation also supports sending images using the toolbar action button, next to the message text input in the chat screen. As you can see, in figure 6.5.5 I have used this functionality to send a picture of a (very hard plastic) duck. This meets functional requirement F2.

When receiving messages from peers, they will automatically appear in the chat. If the user is viewing the chat listing screen, the chats list will update with the chat that received the most recent message at the top, marked in bold to show it is unread by the user. When the user is viewing other screens in the app or has the app minimised, local notifications are shown with the name of the chat and the message, meeting functional requirement F4.

6.6 Feeds & Posts

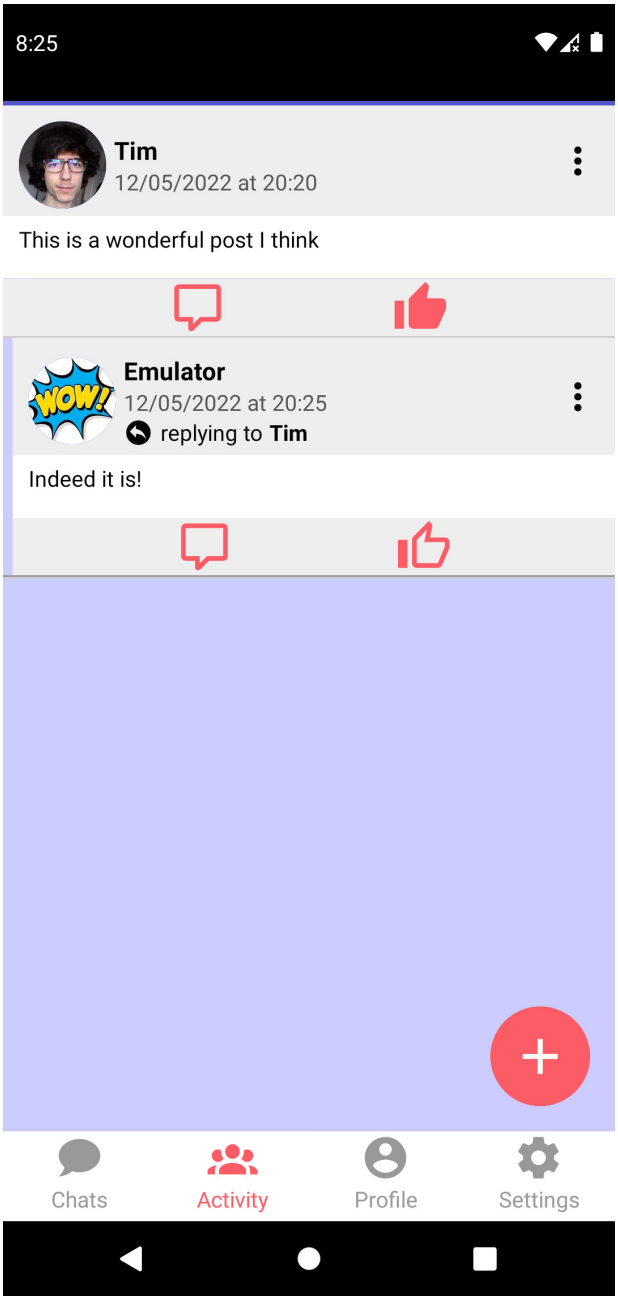

FIGURE 6.6.1: *Screenshot of the activity screen with a post and a reply*

The activity screen is the simplest screen in the app in terms of UI flow as the only sub-screen is the new/edit post screen. However, it contains the combined activity feed - which displays posts from every peer the user follows (as well as themself).

Loading posts

Activity feeds in the app feature “pull-to-refresh” functionality - so when the user makes a pull gesture while at the top of the scroll view, a refresh of the content is triggered. When this happens, or when the activity screen is first opened, a synchronisation request is sent out to every peer the user follows to check for new posts. While this is taking place, posts already stored on the device are loaded from the SQLite database in chronological order, with newer posts at the top of the scroll view. As the user scrolls down, older posts are loaded automatically until there are no more posts to load, such as in figure 6.6.1 where there are only 2 posts.

If a new post is received from a peer, whether as a result of sending the synchronisation request or as a result of that peer creating a new post at that moment, a “Load new posts” button is shown after the received post is cached in the database. When this button is pressed, the feed scrolls to the top if not already there and then reloads the most recent posts from the database.

Creating & editing posts

Posts can be created by pressing the button with the plus sign in the bottom right corner of the screen. This opens the new/edit post screen, allowing the user to type out a text post and publish it. Publishing the post sends out a request to peers following the user’s entity such that they can automatically retrieve the post if online. Pressing the 3 dots button on a post the user has created opens a context menu to edit or delete the post. Editing the post will open the new/edit post screen again, prefilled with the content of the post being edited.

The alternative way to create a post is to reply to an existing post by pressing the speech bubble icon at the bottom of every post, opening the new/edit post screen as seen in figure 6.6.2 (see appendix B). Unlike other posts, reply posts are associated with a parent post - showing a reply icon followed by the text “replying to ” and the name of the parent post’s author. These posts are also indented to visually show which post the reply pertains to, as seen in figure 6.6.1. The thumbs-up like button doesn’t actually do anything; in future iterations this button could be used to notify the author of the post that a peer likes the content. The complete combined activity feed meets functional requirement F3, with functional requirement F6 partially met by the inclusion of post replies.

6.7 Profiles

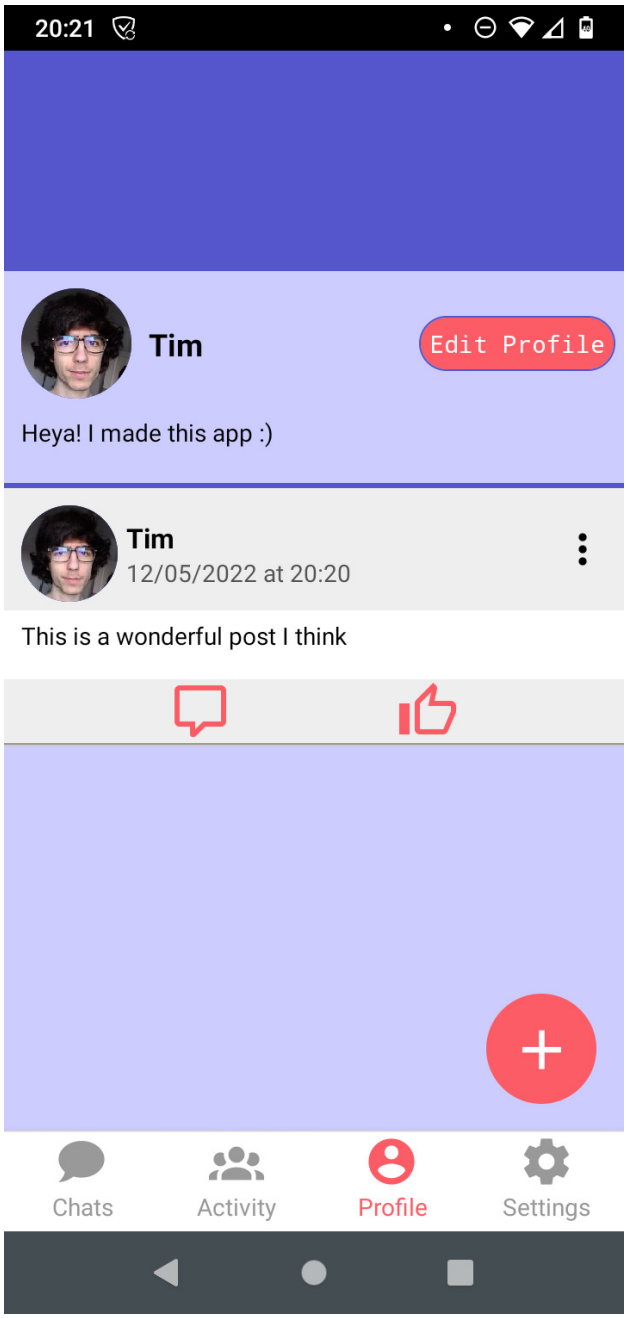

FIGURE 6.7.1: *Screenshot of the profile screen*

The profile screen displays the user's public entity information, such as their name, avatar and bio. Like the activity screen, it is possible to create and interact with posts in an activity feed - but only posts related to the user's entity are displayed.

Editing the profile

Besides activity feed functions available to the user on the profile screen (which are much the same as the activity screen), the user can click on the “Edit Profile” button to go to the profile editing screen. Note that for profiles of peers, this button is replaced with a follow/unfollow button, allowing the user to toggle their subscription to a peer so the user can view that peer’s posts in the activity feed.

When the profile editing screen is opened, it is prefilled with the entity’s current profile data including the name, avatar image, bio, website, location and date of birth as you can see in figure 6.7.2 (appendix A). In the case of editing an existing profile, the user must modify one of the fields before the button will be enabled to save the changes.

Creating a profile

The profile editing screen is also shown when the user is first creating an entity, after the initial setup screen as you see in figure 6.7.3 (appendix A); however, the only difference then is seen in the section 6.1 flow diagram, an entity is created when the profile is saved and the user’s screen is set to the chats listing screen by default - and of course, the fields are all blank when first creating an entity. The name field must have at least one or more non-whitespace characters, but all other fields are optional - the date of birth will be set to the date the entity is created on by default. The implementation of entity profiles meets functional requirement F1.

6.8 Issues & Ongoing Development

As the Peerway platform has been developed in an iterative, combined spiral-prototyping and Agile methodology, the app is still only a prototype rather than a fully fledged application ready for release. However, the app does hit all of the MVP requirements defined in the specification along with objective 4 detailed in the introduction. Throughout development various issues have arisen, with some being solved and some relegated on the backlog for future development. Some aspects of the design and user interface are partially implemented or not yet implemented due to time constraints, but many of the core systems are in place to build these features upon. For example, post visibility options - the code is partially implemented, along with peer authentication certificates and data encryption - but there are still some parts to hook up before it can function.

A full list of these ongoing tasks are tracked using Asana, a task management system akin to systems like Trello and Jira. Figure 6.8.1 shows a screenshot of all the completed tasks, while figure 6.8.2 shows the current backlog of ongoing development tasks for future work.

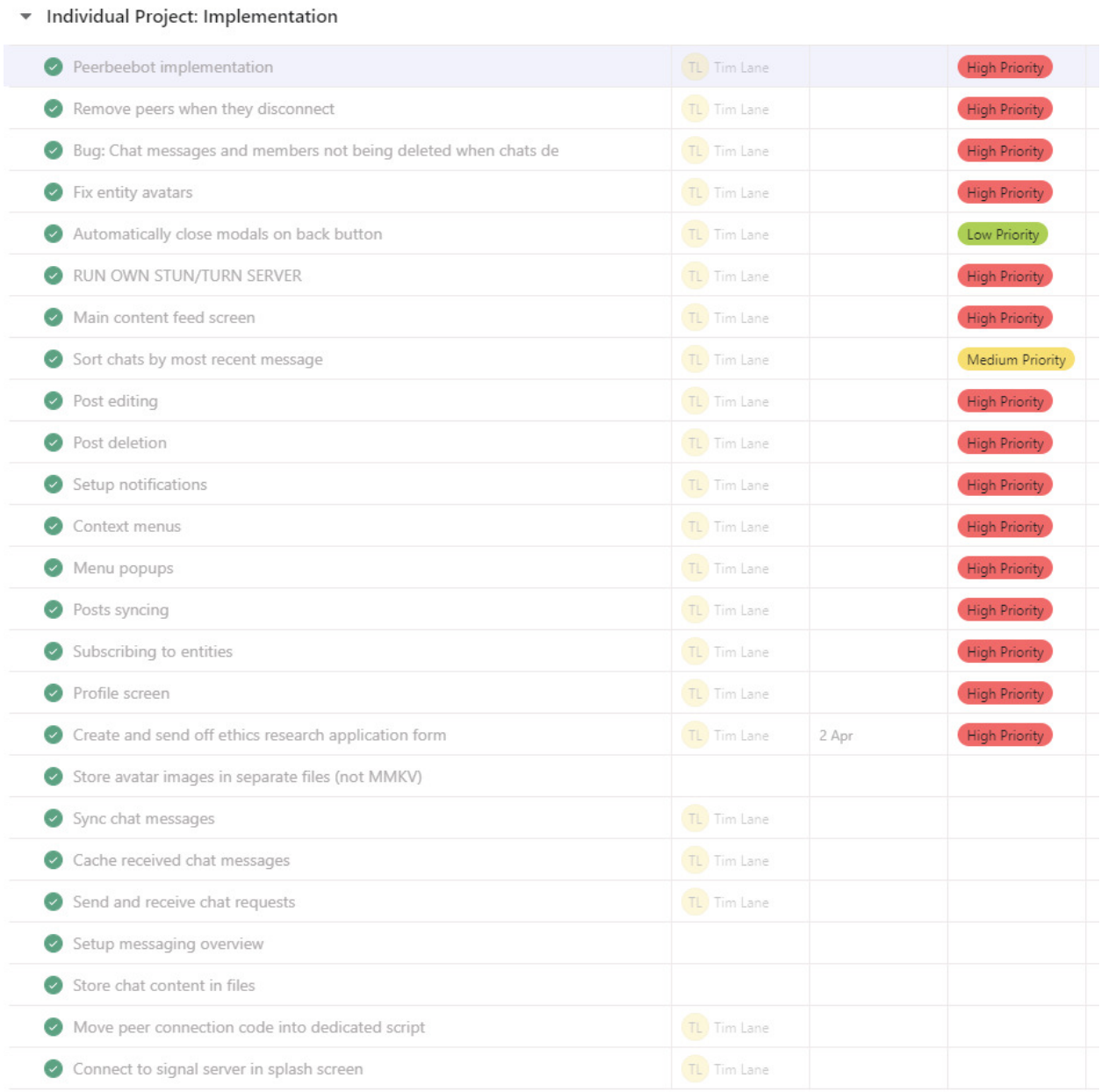

FIGURE 6.8.1: *Screenshot of the completed Asana task list*

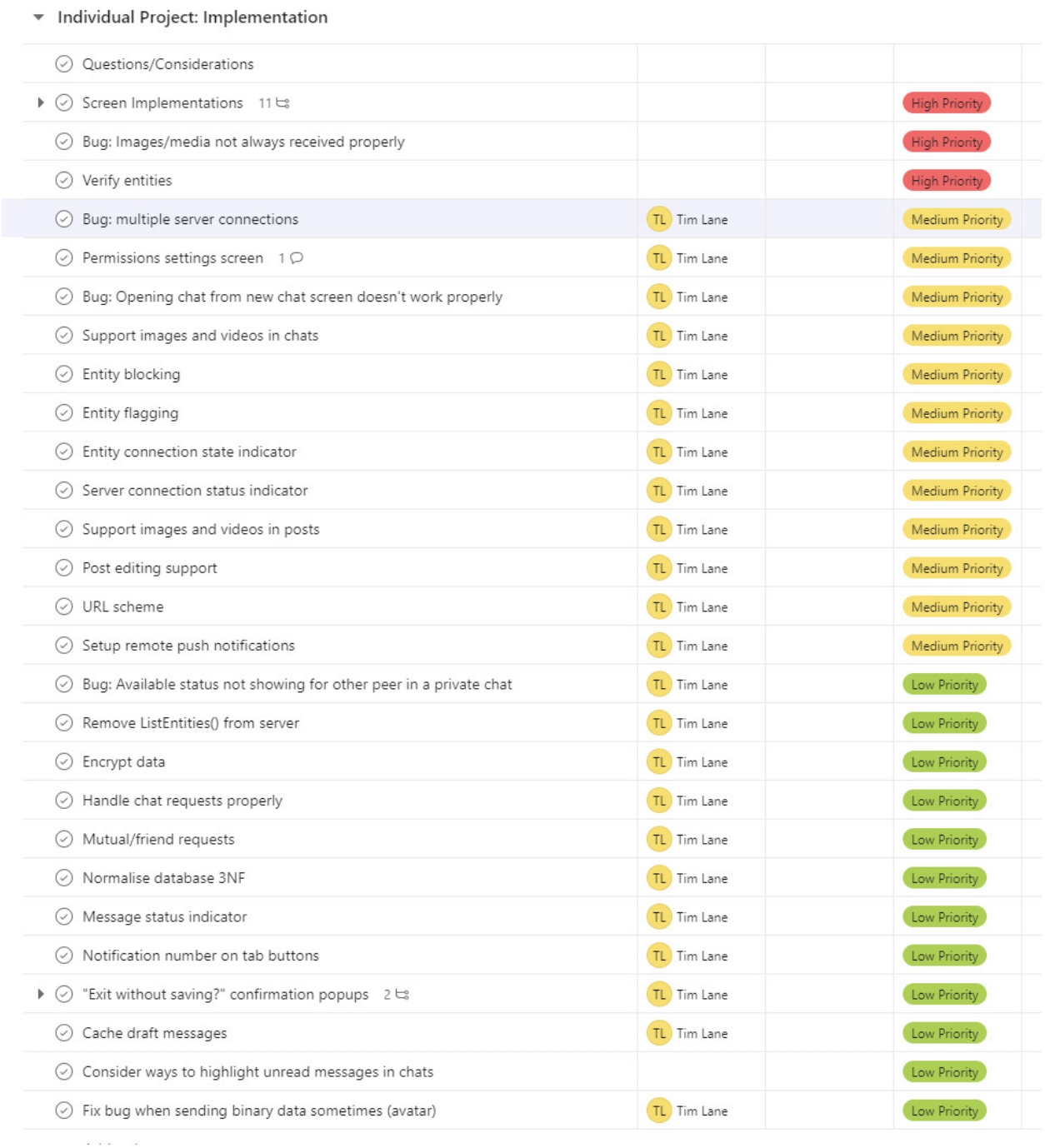

FIGURE 6.8.2: *Screenshot of the current Asana task list*

As you can see, there are a whole range of interesting tasks to work on beyond the publishing of this project to continue improving the app until such a stage where it can grow beyond a prototype into a full release. Some conceptualisation of how some of these might be implemented is detailed in chapter 9.

Unimplemented features

Of the design elements that are yet to be implemented, the one closest to being complete is defined in the specification as requirement NF3. The code for peer authentication exists, as shown in figure 6.8.3 with RSA certificate generation; however, due to time constraints this has not been fully implemented.

```
175     // Create a certificate to use for comms authentication (returns a promise).
176     CreateCertificate() {
177         return RSA.generateKeys(4096).then(keys => {
178             let data = {
179                 private: keys.private, // Private key
180                 certificate: {
181                     public: keys.public, // Public key
182                     issuer: this._activeId, // Who issued this certificate?
183                     created: (new Date()).toISOString(), // The time when this certificate was issued
184                     version: Constants.authVersion // What version is this certificate?
185                 }
186             };
187 
188             return data;
189         });
190     }
191 
192     IssueCertificate(id) {
193         // Issue certificate
194         return this.CreateCertificate().then((cert) => {
195             Log.Debug("Issuing certificate...");
196             // Save the private key
197             Database.Execute(
198                 "UPDATE Peers SET verifier=?, issued=? WHERE id=?",
199                 [cert.private, JSON.stringify(cert.certificate), id]
200             );
201 
202             // Issue the certificate
203             // TODO send only to the specific client!
204             this.SendRequest(id, {
205                 type: "cert.issue",
206                 cert: cert.certificate
207             });
208         }).catch((e) => {
209             Log.Error("Failed to create certificate. " + e);
210         });
211     }
```

FIGURE 6.8.3: *Code for issuing a peer authentication certificate*

Other aspects of the design that are missing are mostly data storage related settings for optimisation purposes, such as those settings mocked up in figure 5.2.5. However, now the core systems are in place, future iterations should go faster. Early on in development there were various issues that had to be overcome such as getting WebRTC working - the code for which was majorly refactored twice after initially getting it up and running.

7 Research Study & User Testing

7.1 Methodology

To better determine whether or not a peer-to-peer mobile application would be viewed as a viable alternative to existing social media, I have conducted a research study on the 3 common types of data storage model (centralised, federated and peer-to-peer) as well as some user testing of the MVP. To that end, I have designed a questionnaire and interview to see how real world social media users view the concept and the app itself.

Questionnaire

I have created a survey questionnaire with Google Forms - see Appendix C. As the form is online and has a built in consent section, it is easy for participants to access without additional interaction with me beyond viewing my initial advertisement and the participant information sheet via my website - see Appendix C. This was done such that I could improve my potential pool of respondents beyond interviewees.

Interviews

Using the same questions as those in the questionnaire, I also invited participants to take part in an interview format as an alternative to filling out the Google Forms survey. The reason for this is for obtaining a more detailed response from participants who can take part in an interview, as opposed to questionnaire respondents who may not dedicate as much time to answering some questions.

7.2 Research Questions & Results

Besides the consent form, the survey is split up into 3 parts. The first part gathers information about the participants' experience, understanding and attitudes on social media, especially in relation to the differences between the different data storage models social media platforms use. In the second part, participants are asked to complete 6 tasks within the app, providing the approximate time it took to complete each task. The third part focuses on how difficult the user found the tasks, as well as general questions about their experience using the app to gauge usability.

Part 1

The first 2 questions provide some basic background information about the participant. Knowing the age and occupation is useful to correlate against the other questions in the survey, as there may be differences in social media experience between younger and older people, as well as understanding of how they work relating to technical knowledge certain occupations may entail.

The survey then briefly explains the differences between 3 main types of data storage model, showing a simple diagram illustrating this before the participant is asked about their understanding of these differences. Following on from the explanation of those key data storage model differences, participants are asked what social media platforms they use (if any), divided by data storage model type. Additional questions ask how long the participants have been using social media, and to what degree of competence they consider themselves to use social media. These questions exist to gauge participant experience with social media in general, as well as experience with specific types of social media by data storage model.

The next question focuses on how the participants primarily use social media; they are asked to order 4 different kinds of common social media interactions. These are consuming content, messaging, organising groups and posting/sharing. This question is useful to gauge which aspects of social media are most important to users on average, such that future work can prioritise some aspects over others.

The next few questions all pertain to participants' attitude to the 3 different types of social media in relation to aspects including:

- Control over data that participants share.
- Automated content personalisation.
- How securely stored participants feel their data is.
- How much participants oppose or support content moderation and censorship.

Knowing how existing social media users feel about these aspects is important for informing development of social media platforms and identifying key topics of contention in regards to the different types of social media platforms.

Part 2

Part 2 of the survey requires the participant to install the Peerway app on their Android device. They are then asked to carry out 6 tasks using basic features of the app, including creating a user account (entity), accepting a chat request, sending a message, navigating to the data privacy settings, creating a post and commenting (replying) to a post. The non-interviewed participants are asked to time themselves for these tasks, while the interviewed participants are timed by the interviewer. The purpose of these tasks are two-fold. Firstly, these tasks help gauge the usability of the app between participants with different levels of experience. Secondly, the tasks introduce the participants to the app for the first time and ensures a consistent base-level of experience between participants using the app to better compare how participants feel about using the app overall. At the end of the tasks, users are permitted to "free roam" and explore the app in more detail if they wish before moving onto the final part.

Part 3

The final part of the survey involves questions on how difficult participants found the app to use, whether they experienced any particular issues in the app or felt about the features (or lack thereof). This part also asks participants whether they preferred any features over existing alternative social media platforms before checking if the participants themselves have any questions. Finally, participants are asked to rate their overall experience using the app and how often participants would want to use this kind of app in their daily life.

The purpose of this part of the survey is entirely to gather feedback on the Peerway app developed for this project, and in interviews generate some discussion about the app between the interviewer and the interviewee to better help understand participants' opinions.

7.3 Results

The research received a total of 7 respondents, of whom 6 participated by filling out the Google Forms survey and 1 participated in an interview. Although this results in a rather small sample size, there is still significant value in the data - especially when it comes to the user testing of the app. The interviewed participant's results are combined with the online survey results to give a slightly better sample size, the main difference being that the interviewee often gave reasons for their answers.

Part 1

100% of the respondents were under the age of 35, the majority of whom were in the age range of 18-24. One of the reasons for asking the question "What is your main occupation?" was to identify any potential bias or correlation related to technical knowledge, as participants in technical fields may have more understanding of social media technology than average social media users. This does not seem to be the case, with a wide range of career fields.

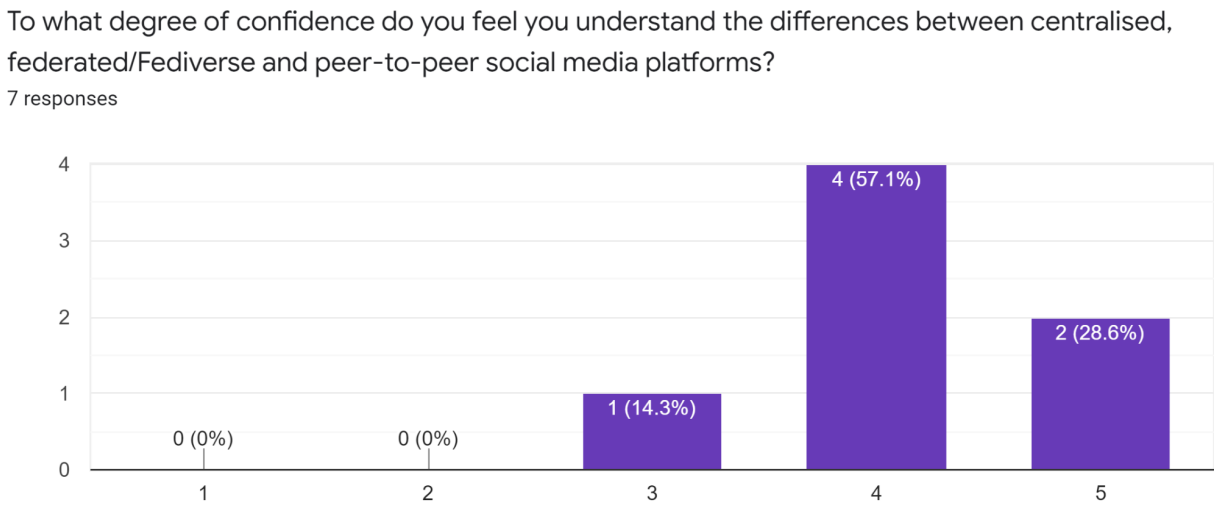

FIGURE 7.3.1: *Answer distribution for the third question of part 1*

Participants generally seemed to understand the differences between the 3 types of data storage models in relation to social media platforms. Every respondent claimed to have medium confidence or higher with the majority having a high or very high degree of confidence in their understanding.

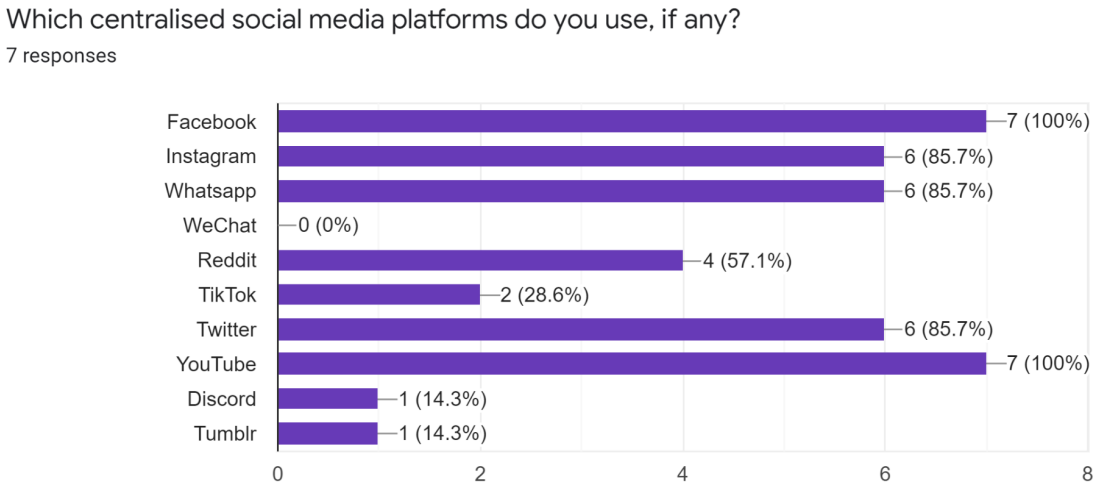

FIGURE 7.3.2: *Answer distribution for the fourth question of part 1*

In regards to what centralised social media platforms participants used, figure 7.3.2 shows that the majority used Facebook and YouTube, followed up by WhatsApp and Instagram which matches up with the popularity of those platforms seen in other research. However, a majority of participants use Twitter while none of the participants use WeChat. The latter is somewhat expected as WeChat is primarily used in China, with this survey conducted entirely in English based at a UK university. The large proportion of Twitter and Reddit users is unexpected, but given the small sample size it may well be chance or related to participant demographics. It does however suggest that the majority of the respondents have a wide range of experience of social media beyond the most popular platforms.

By contrast, only a single participant used a federated social media platform - Mastodon - while no participants used any of the listed peer-to-peer platforms. These results are not very surprising, as decentralised social media platforms are generally not as popular as centralised platforms [61].

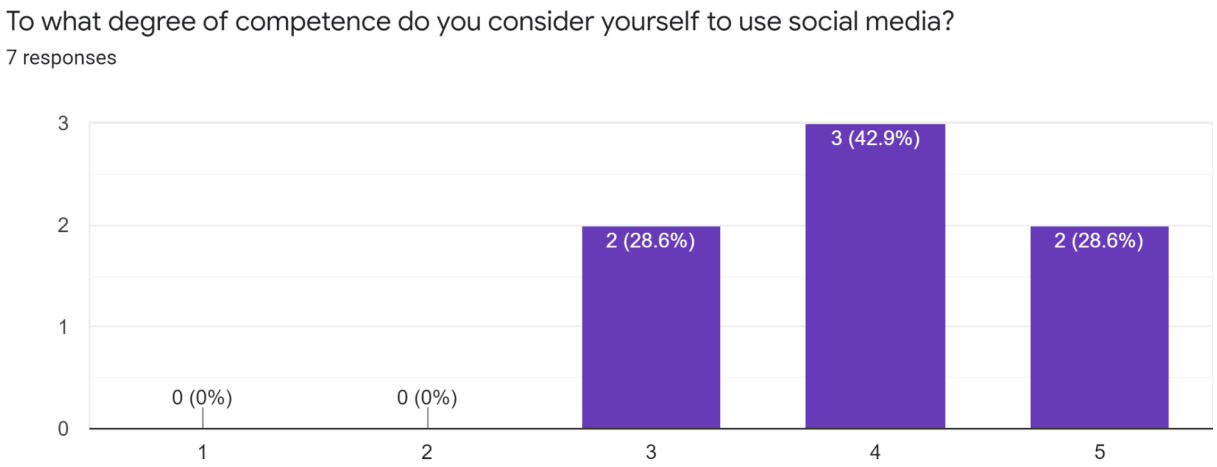

FIGURE 7.3.3: *Distribution of self-ascribed competence*

All participants had experience using social media for 7 years or longer, with the majority stating 10 years. Coupled with a high degree of self perceived competence using social media on average as shown in figure 7.3.3, this gives more evidence to suggest that the majority of the participants are well experienced in use of social media, although mostly centralised social media platforms.

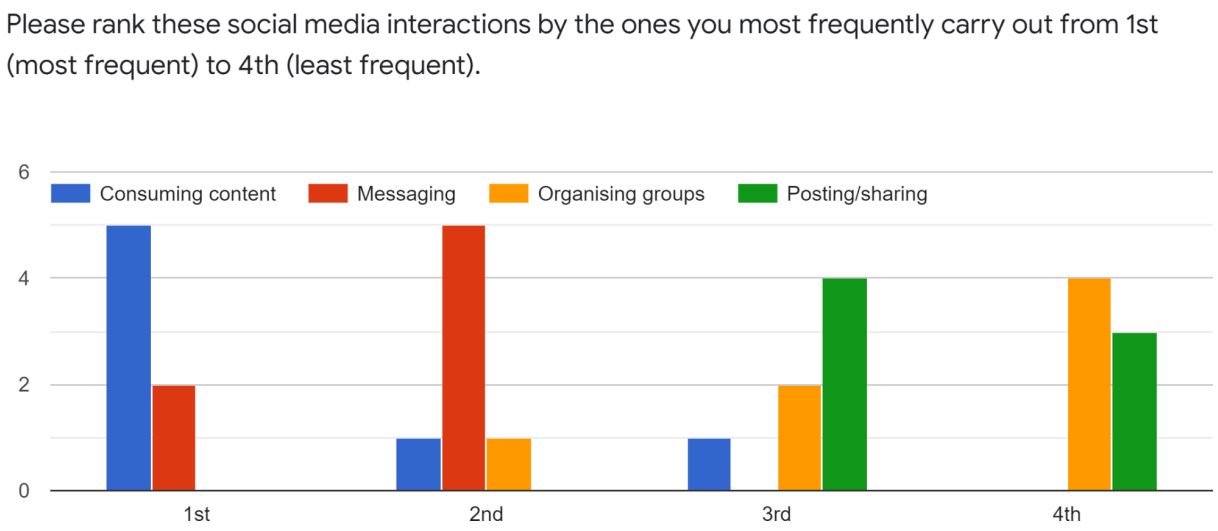

FIGURE 7.3.4: *Ranking distribution of types of social media interactions*

Figure 7.3.4 shows the answers to the next question indicating a trend on how the participants most frequently use social media platforms, with content consumption taking the lead followed by messaging, then posting/sharing and finally organising groups. These results are interesting and show a definite trend towards content consumption and messaging as key components of social media, but it is difficult to theorise whether content consumption is ranked above messaging overall due to the design of the platforms those users are using or other reasons. It would be beneficial to have more participants who use decentralised platforms to compare those who use such platforms against those who use centralised platforms.

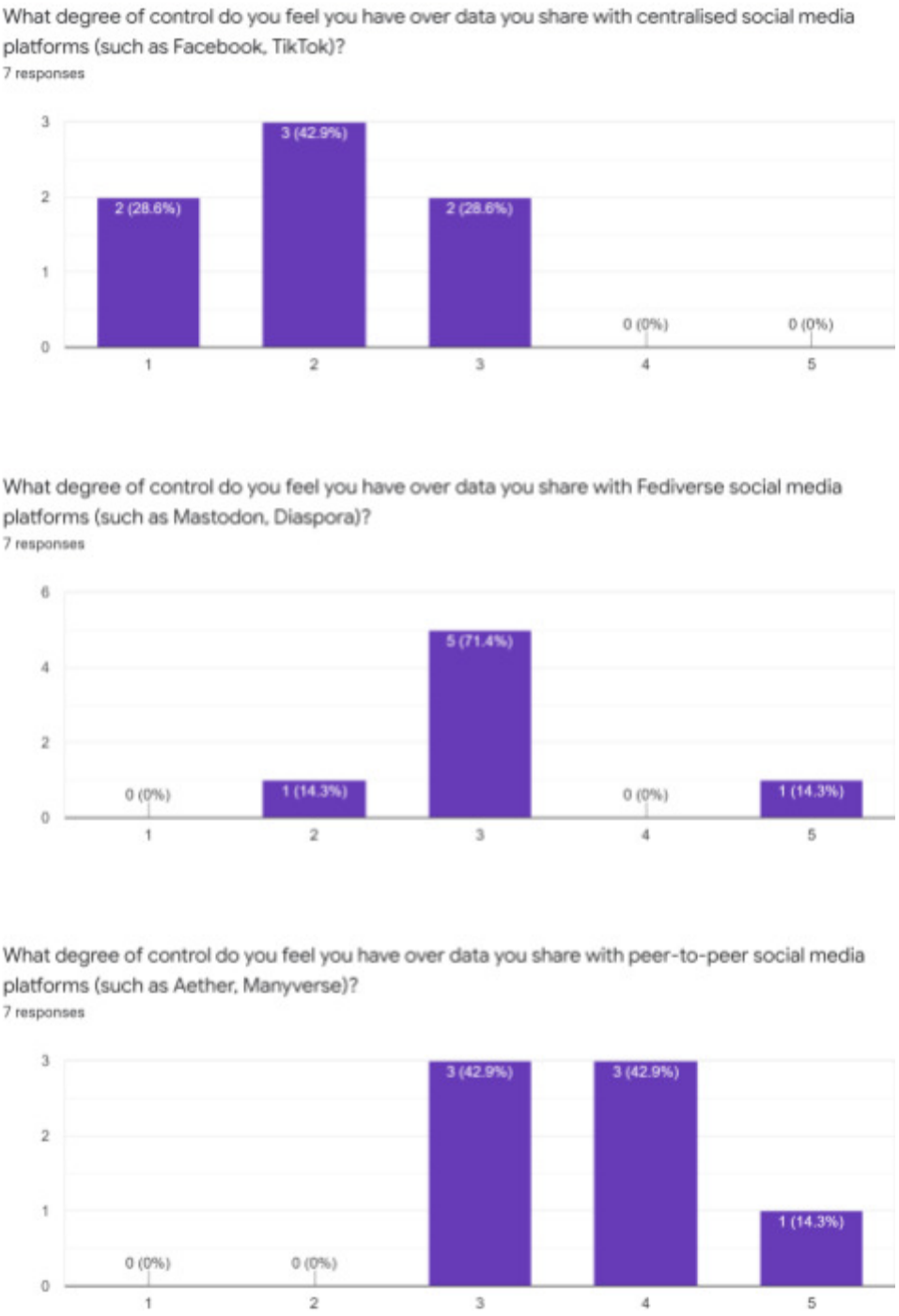

FIGURE 7.3.5: *Feelings of control over data shared*

Figure 7.3.5 shows the results of questions that ask about the degree of control participants feel they have over data they share, with each question associated with one of the 3 social media data storage models. As the data storage model becomes more decentralised the participants indicate they generally feel that they have a higher degree of control, with participants feeling they have the lowest degree of control with centralised platforms overall. This is despite the majority of the participants using centralised social media platforms, generally having little or no recent experience using decentralised social media platforms. Overall this result seems to strengthen the evidence in support of decentralisation in relation to giving users more control over their data.

The next set of questions were more consistent between the different platforms as shown in figure 7.3.6. These asked users how they felt about automated content personalisation, ranging from “strongly dislike” to “strongly like”. Generally there wasn’t much change between the 3 types of platform, with an overall net dislike for automated content personalisation across the board. This makes sense as most of the participants have little experience using decentralised platforms and so are more likely to associate any form of automated content personalisation with the kind seen on centralised platforms.

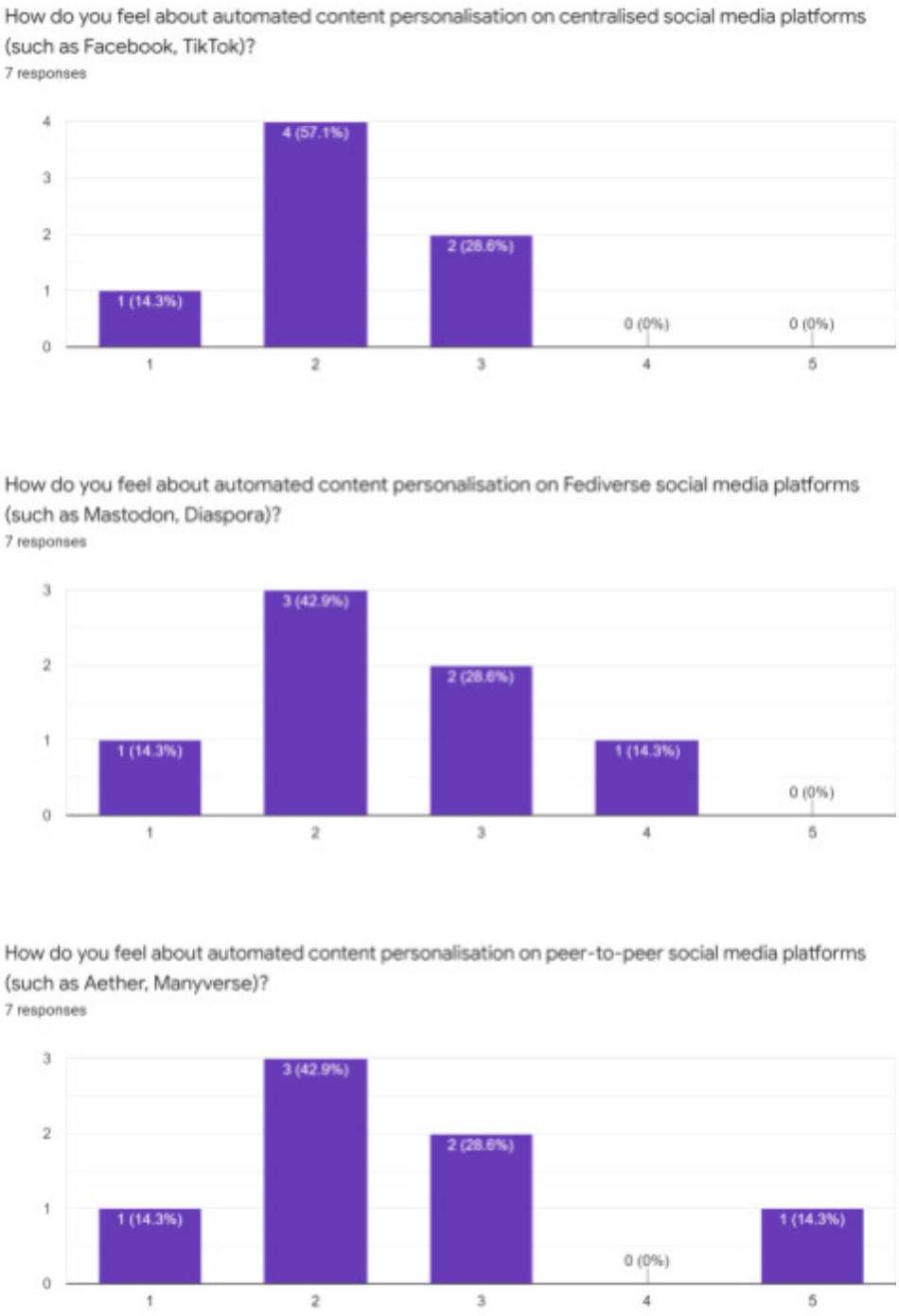

FIGURE 7.3.6: *Feelings on automated content personalisation*

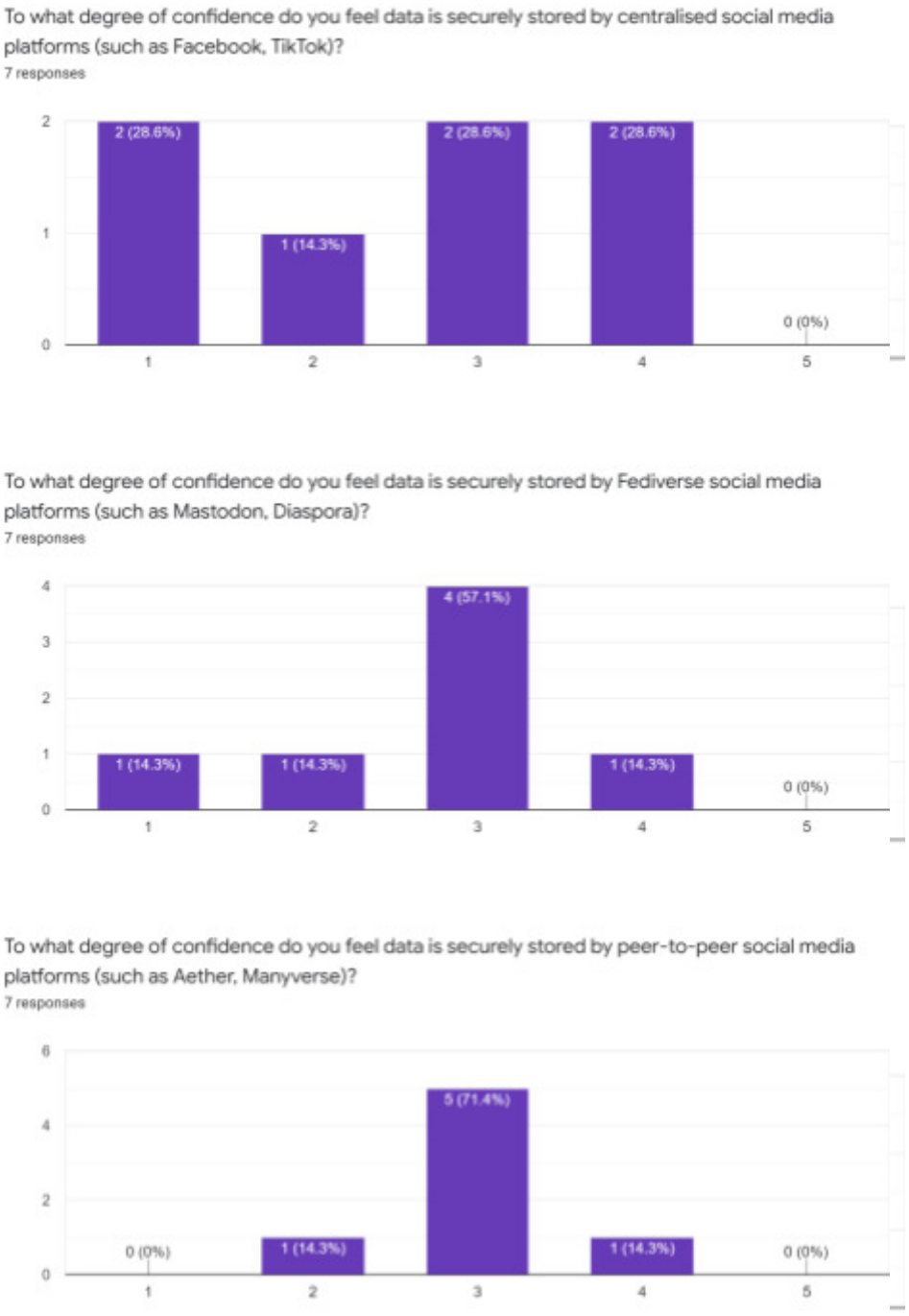

FIGURE 7.3.7: *Feelings on how securely data is stored*

When it comes to how secure participants feel data is stored on the 3 different types of platforms, the results were somewhat mixed as seen in figure 7.3.7 - with centralised platforms having a relatively even answer distribution, while decentralised and peer-to-peer platforms indicated a trend towards neutral feelings. Though it should be noted that none of the participants felt any of the 3 types of social media platforms were highly confident about how securely data is stored. It's possible that the questions were too open to interpretation; in fact the interviewed participant asked for clarification on this:

"When you say securely are we discussing, just simply, not likely to be accessed by someone not authorised by the platform, or held in a way that only people I would expect to access to [stored data] have access [...]?"

In other words, does "secure" mean secure in a way that data can only be accessed by the user, the platform and third-parties who are granted authorisation to use the data - or does the term mean secure from *anyone* other than those the user authorises, so third-parties cannot access the data? This is an interesting consideration that I originally approached in the first sense of the word, but may explain why there are mixed answers to the question for centralised social media platforms at least. For federated and peer-to-peer platforms, generally participants tended towards neutral feelings on the subject, again this may be due to lack of experience or knowledge around decentralised social media platforms.

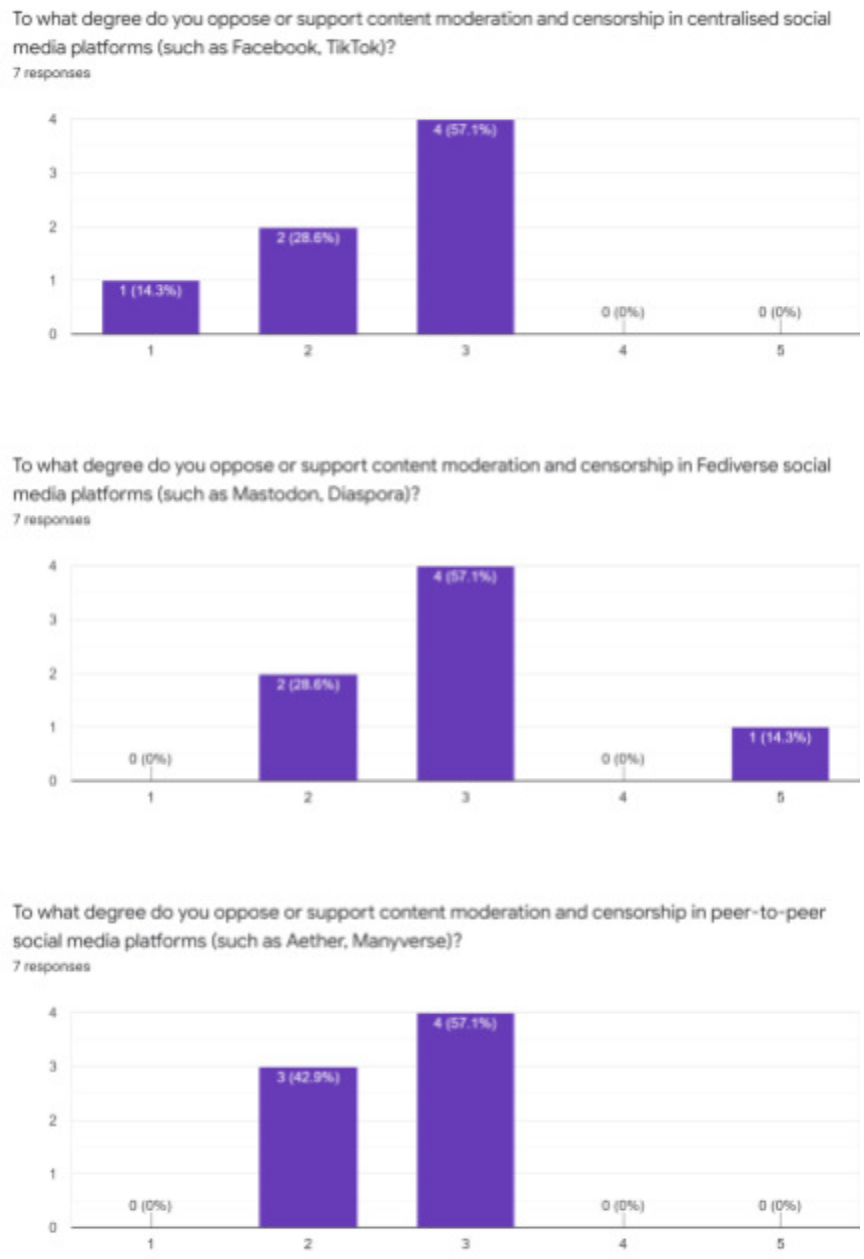

FIGURE 7.3.8: *Opinions on moderation and censorship*

The next set of 3 questions shown in figure 7.3.8 asked to what degree participants opposed or supported content moderation and censorship on the different platforms. These had fairly consistent answers, with the majority having neutral opinion and most of the minority slightly opposing moderation and censorship. This may be indicative of uncertain or slightly negative opinions on how and in what ways platforms enact moderation and censorship.

Part 2

The 6 user testing tasks tested the general usability of the core app features. As both the questionnaire and the interview survey formats were conducted online rather than in person, it was not possible to measure the error rate of participants but approximate timings were obtained for completion of each task. For most tasks users responded with times in seconds or minutes, but in cases where a numeric time was not specified those results were omitted from average time calculations. All participants successfully completed each of the 6 tasks, so there was a 100% success rate.

For task 1, the average completion time was ~94 seconds. There was significant variation, ranging from 20 seconds to 3 minutes. This is somewhat expected as the UI does not require users to fill out all the fields of the profile editing screen when creating an user account (entity). As a result, it is likely that some users took more time filling out fields than others.

Task 2 was much shorter to complete on average at ~21 seconds. The variation for this task was also much less, although there was still some significant relative difference between the shortest time at 3 seconds and the longest time at 1 minute. Generally this is positive, as it shows that users can quickly locate a chat request and accept it.

Task 3 was even shorter at ~6 seconds on average to complete, with a range from 2 seconds to 10 seconds. This is positive, as it's indicative that the UI for opening a chat and then sending a message feels intuitive - as a core real-time communications feature of the app, ease of use is important.

Task 4 took an average time of 11.5 seconds to complete, with a range from 5 seconds to a minute. Generally it seems that locating the privacy settings is a relatively easy task, which is beneficial for users who may move to the platform for the sake of improved privacy.

Task 5 took ~21 seconds to complete on average, similar to task 2. Once again there was some variation, with the shortest time at 10 seconds and the longest time at 1 minute. This variation is somewhat expected in this case due to the open-ended nature of the task, allowing users to type anything in and post it. Despite this however the task took significantly less time than task 1, likely because there are less UI fields to interact with.

Task 6, the final task, took ~18 seconds. This isn't much different from the time taken to create a post as in task 5, which is expected as the UI for creating a comment post is identical except for the button used to open the post creation/edit screen. The range for this task was from 5 seconds to 1 minute, likewise similar to the variation seen in task 5.

Part 3

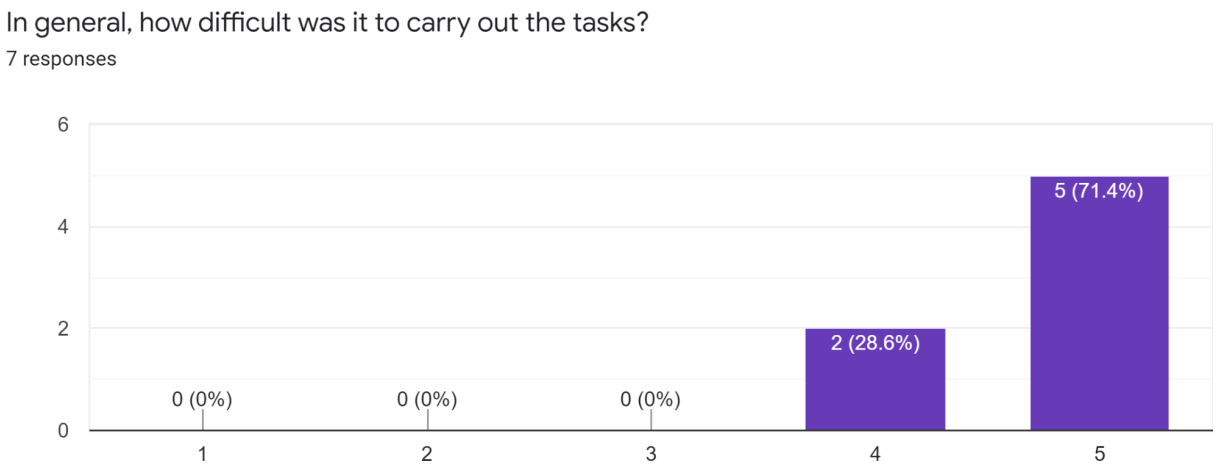

FIGURE 7.3.9: *Results on difficulty of completing the tasks*

It should be noted that for the first question of part 3, "very difficult" is lowest and "very easy" is highest - so a score of 4 or 5 indicates the tasks being easy to complete. The results for this question, shown in figure 7.3.9, are very promising - all participants indicated that the tasks were easy or very easy to carry out, which suggests the UI is generally intuitive to use.

The next few questions focus on user feedback. When it came to issues faced when completing the tasks, a couple of participants found that finding the button to create a new post was unintuitive. In the question where participants were asked if they had any questions, two participants also asked how they could start a new chat. Considering the responses to these questions, it's possible that the location or styling of the new chat and new post buttons are not very intuitive - both are located in the same area of the screen, with the same colour scheme and style but with different icons. Perhaps repositioning these buttons or using text such as "New Chat" or "Create Post", rather than icons, would be more intuitive for users.

A couple of participants were confused by the profile page, as after creating a reply post it seems that the post did not immediately appear until those users refreshed the screen. From testing this myself in the build, it seems to be a minor visual bug where creating a reply post doesn't make it appear without reloading the posts manually. Besides that, it seems that there were no significant bugs or issues with the app, which is quite promising.

Several participants felt that the app was lacking features seen in other social media platforms. The most common feature participants felt was missing is the ability to add images or videos to posts. Besides this, there were a mixture of features participants wanted, such as a search box for finding others on the platform, better profile customisation, customisation of the app's appearance, and publicly accessible group chats.

In general participants found the app colour scheme and layout preferable to alternative social media platforms, as well as the general usage with comments such as "it just works", "the quick reaction times" and "Layout is simple and easy to use". These are very positive remarks, suggesting that the app is quite user friendly, intuitive and performant. When asked to rate their experience using the app, all participants gave a good or "very good" rating. In terms of how frequently participants would want to use the app in their daily life, all participants except one chose often (but less than very often or all the time) with the exception choosing not often (but more than rarely or never). Overall, the user testing has proved positive with good regard for the usability of the app, though with room for improvement.

8 Results & Evaluation

8.1 Project Successes

The project as a whole has achieved a number of successes. A full listing of platform development successes include:

- Meeting every MVP requirement of the specification.
- Partially meeting additional requirements and design concepts, such as requirement F6 with the implementation of post replies.
- Completion of the research study and user testing, gaining valuable insight for future work.
- Meeting every project aim and objective defined in section 1.1.

Besides these technical and project development achievements, the project has managed to cover a lot of ground in a short space of time - with a significant number of implementation concepts informed by the iterative design process for future research and development. Having seen the results of the research study and user testing, it seems that not only are the technical aspects of such a platform plausible but from a usability standpoint initial outlooks are very promising.

8.2 Project Challenges

Of course, a number of challenges arose during development of the project. Delays caused by unforeseen issues occurred, including:

- Problems configuring WebRTC servers, eventually forcing me to set up my own STUN/TURN server.
- Switching from the Metro development environment to the React-Native CLI (command line interface) early on due to problems with running debug builds.
- An unexpected amount of work required to submit the ethics application, taking several days to do so and then about 3 weeks before receiving favourable opinion forcing a short timespan for data collection.

These events along with the usual unwanted development slow downs such as bugs meant not as many features could be implemented as I would have wished, but despite this all of the core project goals were successfully met.

8.3 Overall Results

TABLE 8.3.1: Measuring completion of aims & objectives

Objective	How has the objective been met?
1	Chapter 2 covers the common social media features (partially assumed as stated in section 1.5) with section 2.2, identifies key problems with popular centralised social media platforms in section 2.3 and then directly compares those problems with decentralised alternatives in section 2.4.
2	The requirements set out in the specification section 3.2 cover the MVP and beyond.
3	Chapter 4 covers all of the core platform components, considering how data can be communicated between peers in practice in an efficient way that can enable the emulation of social media platform features. These details are covered more in chapters 5 and 6, with specifics on how such systems have been implemented.
4	Each sub-objective has been met here: a. A working prototype implemented that doesn't store data on the signalling server beyond the lifetime of a connection (instead stored on the user device locally) b. The platform utilises WebRTC for peer-to-peer connections that can be used to transfer data, including posts, messages and images as well as other arbitrary data. c. The 3 common social media features implemented are chats/messaging, activity feeds with posts, and entity profiles.
5	Chapter 7 covers the entirety of the research project and user testing.
6	Chapter 9 covers future work including improvements to the platform, though other sections also go over potential improvements.
7	With the completion of this chapter, the final objective is met.

As table 8.3.1 demonstrates, every objective of the project has been met successfully. While there are parts of the implementation that could be improved, the scope of this project has been filled. Trying to cover as much ground as possible in the space of 3-4 months has proven challenging but not infeasible.
The research study and user testing could be much improved; as there was a relatively small sample size there was little opportunity to compare the experiences and attitudes of federated and peer-to-peer platform users as centralised platform users are more common. However, the user testing was extremely valuable even with the small number of participants - being able to see which issues need to be prioritised and what aspects of the app work well in practice is useful to know for future work.

Despite the broadness of the project, with so many possible routes to go down with the design, implementation and research as a whole everything has gone remarkably well. As a proof-of-concept, Peerway, the peer-to-peer social media platform works. Not only that, but the iterative development cycle has enabled and informed rapid conceptualisation of potential features for future work, with a wide exploration of many aspects of social media. Each part of the process has informed the next, with gradual refinements making pragmatic progress on using the local device-based data storage model at the heart of this project. There is still so much room to grow too - with a large body of concepts and considerations for future work, this project could act as a stepping stone to greater things.

9 Future Work

9.1 Platform Improvements

There are some extra features included in the design section of this report that did not make it into the prototype. In addition, there are a multitude of potential features and improvements that could be added to bring better parity with existing popular social media platforms. This section shall detail a few of these proposed features and concepts, briefly detailing how they could be implemented effectively.

Push notifications

While push notifications were not deemed necessary for the MVP, they are a key feature of mobile apps as they enable notifications to be received even when the app is not running. These kinds of notifications can be used for things like chat messages. These could be especially useful for a platform like Peerway, as they can be used with synchronisation requests to ensure that peers are always kept up to date with the latest posts and messages from the user.

Mutuals/Friends

Another feature that has been discussed previously but did not constitute the MVP is the implementation of entity relationships, such as mutuals. The concept is an important one for reproducing social media features such as friends and groups. In essence, we define a mutual entity as an entity which has agreed with another entity to be part of the same group. For many users, that grouping could be "friends" as platforms like Facebook represent the mutuals concept [56].

The beauty of this abstract concept of mutuals however is that the concept of what a mutual group is for can be easily extended and customised. For instance, a user could create an entity they use specifically for discussing a particular topic and all the mutuals of that entity could be referred to as "members" of the group. Users could even charge other users wanting to join their mutual group with a subscription fee. There are a whole range of exciting possibilities for this, but the underlying system can operate in exactly the same way for each of them, allowing users access to the same base features such as setting permissions to include (or exclude) mutuals from viewing certain posts.

Custom URL scheme

One very important aspect of any social media platform is the ability to follow other users or add them as friends. However, before this can take place, a user must first find the entity they wish to follow or add. While this can be done by searching for peers connected to a signalling server instance, there is no guarantee that the peer a user is interested in finding is connected to the server. In fact, it is especially unlikely that the peer will be available as the app will not maintain an active server connection while in the background. However, if the entity ID of the peer a user is interested in is already known, then the signalling server could be used to send a push notification to that peer. Centralised and federated social media platforms typically provide a web link for user profiles that can be easily shared both on and off the platform [57]. Although a web link per user is not possible for Peerway as users are not tied to a specific web server, URL schemes can be created for Android (and iOS) apps which could serve similar functionality.

Instead of using the **http** protocol prefix in a URL, the prefix **peerway** is used to open the Peerway app and execute a specific command. For example, the URL **peerway://peer/follow/123e4567-e89b-12d3-a456-426614174000** could open a popup in the app asking the user to confirm whether they wish to follow an entity. When the user confirms, a push notification can be sent to the entity via the signal server to notify them that they have been followed. This scheme unlocks the world wide web as a medium for Peerway users to share their entities for discovery by others. This also supports a feature of some social media apps where a QR code is generated for user accounts so that people can easily scan the code with their phone camera to add a user as a friend, similar to Snapchat “Snapcodes” [58].

Encryption & peer verification

For a full release of any kind of social media platform, it’s important that data remains secure and that users can trust each other. Some of these systems have been discussed and partially implemented in Peerway, but there are still a lot of things to consider and improve upon. For instance, removing members from group chats wasn’t implemented in the prototype - doing so would require changing the encryption key members use to send messages such that ex-members can’t view or send new messages in a chat. Other factors such as enabling HTTPS rather than using clear text HTTP are important to consider beyond the prototype for security.

Peer verification is also important - using the existing but unused certification code in the prototype implementation to do this would be quite trivial, ensuring that users don't unknowingly send data to fake peers.

Another reason to get peer verification implemented in future work is that it can be used as the mechanism for setting sharing permissions on data such as posts. The UI design for creating or editing a post shown in figure 5.4.4 illustrates how the user would utilise this in practice, such that only specific groups of entities can view particular posts.

Data storage optimisations

Also explored in the design were the storage and cache settings. As storage space is always limited and the data storage model relies entirely on user devices, it is important to try and reduce the storage required as much as possible while keeping the platform usable. There are several angles to approach this from; perhaps the most obvious is use of compression. Allowing users to configure how data received from peers is compressed could help support those who don't have much space available by letting them more heavily compress media files for instance.

Another approach is to limit how much data can be cached from peers on a user's device. It seems unlikely that a user would need access to the entire history of a peer's posts at any one time except in rare circumstances, for instance. By only caching a finite number of each peer's most recent posts, messages and media files, the app should be able to reduce the amount of space used quite significantly. By implementing settings for the user to control these parameters, the platform should be able to support a wide range of user devices.

Entity backups

One concept that was not very explored in this project was the concept of data backups. One advantage of popular social media platforms is that server databases can be replicated as backups in case of system failures. With the data storage model used for this project, we are relying entirely upon the local database stored on a user's device. This presents challenges as it's possible for people to damage or lose their phone, potentially causing loss of entities and all their data in the app. To mitigate the impact of this risk, backup solutions are needed.

One way to backup data is to use peer devices as backup storage. However, as mentioned previously storage space is limited and so we do not want to waste more than necessary - most of the time, we hope that backups would not be required. Of course, some data can be implicitly backed up this way - as peers cache the user's most recent posts and messages, at least some of those can be retrieved in event of loss of an entity. However, this is obviously not sufficient when considering other systems in place such as entity verification - we need to backup more than just data shared with peers, such as private encryption keys. Perhaps the most straightforward data backup solution is to provide users with a way to export their data - which they can then store on a separate device. This could be done automatically if users provide a device which the app can export to periodically.

Cross-platform deployment

Although the prototype has been developed primarily for Android devices, React Native has built in support for iOS and the general purpose React framework is available for web applications. It may be possible to create a website or web extension, which provides the user interface and JavaScript code required for the client to function in a similar way to the mobile app. This would enable other users to access the platform, although as desktop devices tend not to be online at times this could reduce usability of the platform substantially (as users will be less likely to be available at the same time to synchronise). However, this issue could be overcome by implementing a web extension or web application that relies on the user's mobile device in a similar way to WhatsApp web [62]. This way the user's mobile device is still required but it allows an alternative interface for using the platform.

Federated signalling servers

The prototype currently relies upon a single signalling server instance to connect peers together. This means that there is still a central point of failure for platform users - although at least with the data storage model being decentralised, the impacts of a server being shutdown or an outage occurring are minimised. It's also possible for users to simply change the server they are using at any time in the app network settings, so the platform has some resilience built in. Ideally however, the user should not be required to manually change the server address whenever there is a fault. Furthermore, there is no connectivity between individual servers - so users will only be able to interact with peers on the server they are currently using.

To solve these issues and improve global connectivity of users on the platform, a form of server federation may be useful. Of course, this would not be applied to the data storage model - but if server instances could interoperate, then users could access every peer using the platform as long as those peers are connected to one of the servers in that federated network. Because the servers do not store any data, it doesn't matter which server a user connects to. This is an ideal situation where anyone can run a server and have it connected to the network, then clients can automatically determine the best server to connect to - in many ways the servers would act like specialised routers for the platform, forwarding requests to where they need to go. As long as peer verification and data encryption are well implemented, this should be a secure solution.

Entity searching

A common feature of popular social media platforms is the ability to search for other users. This would be feasible to implement with the signalling server for users who are online; although it would not be possible to search for users not connected to the server, unless a hybrid data storage model were used where the server caches some data about the entities that connect to it. This might be worth exploring, although doing so introduces significant complexity and considerations such as how the search should be ordered - should the most relevant peers appear first, or the most popular? Peerway is designed to put users in control of the data aspects of the platform as much as possible without requiring much technical knowledge, and introducing systems like server caches may affect this with unforeseen consequences later on.

User onboarding

One feature that may help new users is the addition of onboarding hints. Essentially, these would function as a basic tutorial to help a new user find everything in the app. Arguably this could be redundant for most existing users of social media who are likely to understand the user interface somewhat intuitively, but it may be useful down the line if the platform ever becomes popular enough to attract non-users of social media.

9.2 Further Research & Development

Preventing identity theft

As noted in section 4.3 when considering the impacts of Peerway's data storage model, identity theft is still a potential cause for concern. The partial solution noted could be further improved with some features designed to mitigate bad behaviour. For instance, if several mutuals have flagged an entity as an identity thief, or simply blocked the entity, then that can be made clear to other mutuals with a warning popup before they decide whether to trust the entity or not. Combined with privacy features that enable users to hide data from peers who don't fit particular conditions (e.g. must be a mutual), it becomes more difficult for bad actors to impersonate an entity. In practice, this isn't much different from existing social media platforms, where it's possible to create a fake account or change your account to pretend to be a different individual or organisation. However, by providing proper tools that inform users in a way that lets them make explicit decisions to trust or distrust entities, the platform will be more resilient against various forms of bad behaviour, not just identity theft.

Mitigating bad behaviour

Bad behaviour on social media is not new. Twitter is notorious for problems with targeted harassment campaigns that exploit features such as public lists, trending hashtags, user tagging and keyword searches in combination with bots (automated user accounts) to spam hate towards groups, organisations and individuals using the platform. Despite features such as blocking and reporting individual user accounts, due to the public-facing nature of Twitter by default it's not very hard for perpetrators to evade blocks and suspensions/bans by using alternative accounts or getting their friends to continue harassment. Users who are subjected to targeted harassment often use third-party tools to block perpetrators enmasse, or simply lock their account (the "protect your tweets" feature) so that only followers can view their tweets and follow requests have to be reviewed by the account owner.

Contrast this to Facebook, where the default privacy of posts is set to friends only and it is possible to customise the visibility of a post at a granular level. There are additional privacy settings for choosing who can send you friend requests, whether or not search engines should list your profile page, and how messages are handled from users who haven't messaged before. The design of these features makes a huge impact on how difficult it is for strangers to engage in targeted harassment. Both Facebook and Twitter have a somewhat similar blocking system. After blocking an account, the user can no longer interact with the blocked account and the blocked account cannot interact with the user. This does nothing to prevent users from harassing other users. Reporting can have this effect, sometimes resulting in a suspension or ban from the platforms.

Reporting entities & content

While freedom of speech is important, it is also important to note that it does not entitle people's views to be platformed or perpetuated. If someone is banned from a social media platform, they can go to another platform (or set up their own) and make the same statements; when considered as a "negative right" it is not a violation of free speech for a platform to take down content or accounts. This is both a good thing and a bad thing, as governments sometimes use this as a justification to enact censorship but it also means platforms are able to remove harmful content such as harassment campaigns.

Content moderation has long been a bone of contention for social media platforms. With a fully decentralised data storage model however, these decisions do not need to be put in the hands of a central authority - they cannot be, by design. This means that the traditional kind of reporting is not viable and alternative methods are required to help users control the content they see.

One way to put moderation power in the hands of users is to use a flagging system. When a user wants to warn others of harmful content or peers, they can flag the content or entity along with particular reasons (chosen from a finite list of options, or specify a custom reason). These flags are then distributed to mutuals or other peers who interact with the user such that they are warned of the flag and reasoning when they interact with the flagged peer. Beyond that, the users can decide how they wish to proceed - they may ignore the warning, or use it to decide to block the flagged peer for example.

There are also plenty of issues around topics such as misinformation being rapidly spread around social media platforms and third-parties influencing election results. It is worth exploring ways to prevent these situations occurring in every social media platform, but it is especially pertinent in a peer-to-peer platform like Peerway where there is no centralised automatic content moderation to help users.

Some suggestions for exploration in further research:

- Compare moderation tools across different platforms and assess how effective they are, surveying existing users for their experience and ideas.
- Consider how peer-to-peer moderation could work; perhaps automatic blocking and warnings about peers who have been blocked by other peers are necessary tools to mitigate these issues.
- Build a prototype or simulation using different moderation techniques to handle a variety of problematic scenarios.

Interaction metrics

Popular social media sites such as Facebook and Twitter often feature public facing metrics on how many likes, comments and shares occur with posts. Some platforms go even further than these with additional metrics for content creators to use. However, these pose a problem with the data storage model developed in this project. For one thing, how do you verify that the number of likes on a post is genuine? One solution is to check this against every peer recorded to have liked the post, and those peers can verify their interactions. However, it's not feasible to check against every peer and peers may not always be available to verify this.

Distributed technology such as public blockchains may prove their worth in this aspect of a peer-to-peer social media platform, as these offer means for verification of transactions at scale. However, it may not be the most efficient solution available - for example, proof-of-work based blockchains such as that used in Bitcoin [63]. It's also not foolproof as it's perfectly possible that someone could automatically generate bot accounts and artificially affect metrics. This is a problem already faced by existing social media platforms [64].

Of course, it could be argued that these metrics don't need to be public - if only the creator of a post has access to these metrics, there is less reason to falsify the metrics and the user can at least verify interactions from peers they already know. We should also consider the implications of implementing metrics like these in the first place. There are various research studies and articles on social media interactions and how they affect users like [65] and [66]. Identifying the best course of action should consider existing scientific literature and may benefit from additional research.

Long-term, large scale usability of the data storage model

This project has not tested how viable the data storage model is at large scales. With millions or even billions of users and large quantities of data being transferred between peers, can mobile devices really cope with the storage requirements? Or perhaps some aspects of existing social media platform designs are just not viable to replicate with the data storage model long-term. We need more research on how effectively these systems can scale up in practice - from the results of this project, I remain optimistic that this is possible. Nonetheless, more testing and simulations should be conducted to explore problems and solutions in this space.

One issue with the current prototype as a proof-of-concept is that for users who might have thousands or millions of followers, their device would be constantly bombarded by peers attempting to access their data. To mitigate this issue, it may be possible to utilise peers who have already cached a user's data as forwarders - building a kind of naturally distributed network to propagate data.

Wider research on social media attitudes and usage

While the research study for this project brought a little insight into attitudes towards the different types of social media data storage models, it would be beneficial to study a wider range of participant demographics and more social media users with experience of decentralised social media platforms. Studying the attitudes of users from a variety of countries may be useful as some social media platforms are unavailable (at least, without a VPN) in certain countries, such as Facebook being banned in China [14]. One interesting aspect of federated platforms is that currently even if one federated server is blocked, others servers are available so the entire network does not fail - possibly even allowing federated content from blocked servers to be accessed, like built in proxies [67]. As to whether existing social media users of countries that block some social media platforms would want to use decentralised alternatives, that is an important matter for further research.

Another key focus of research going forward should be the long-term user experience with the platform. With additional improvements and features that bring the prototype on par with existing popular social media platforms in terms of features, trialling the platform for several weeks or months of use among existing social media communities would provide excellent insight into the true viability of the platform as an alternative. It would also be an opportunity to test features unique to the data storage model, such as adding user configurable settings to optimise data storage and moderation tools.

10 Conclusions

This project explores using a peer-to-peer data storage model for social media by building a proof-of-concept prototype for Android where user-generated data is primarily stored on the user's own device. The aim of the prototype is to emulate common features of social media in a way that existing social media users would find familiar, while adhering to the unique data storage model where no data is stored on a server beyond the lifetime of a user's connection to the server. Each of the aims and objectives were met successfully, including the prototype MVP.

A research study examined experience and attitudes with the main types of social media data storage models, as well as conducting user testing with the prototype platform to gauge the usability and viability as an alternative platform to existing social media platforms. Although the sample size of participants was small, overall the results were positive and indicate that the developed prototype is generally user-friendly and intuitive for existing social media users. This suggests that as a proof-of-concept, the project is successful in showing that the core aspects of the data storage model are possible to implement in ways which do not negatively impact user experience. There are a wide range of ideas and potential features that have yet to be explored, some of which are identified in this report. Further research and development is necessary however to determine if the concepts can be improved upon and scaled up long-term, such as the premise that modern mobile devices are capable of storing enough data to make such a social media platform viable.

11 Reflection On Learning

When I came up with the initial concept for this project, I had some vague idea on how certain aspects of the internet works but little understanding of its limitations. Having now explored a range of peer-to-peer techniques, it has become clear to me for example that the physical internet structure we have just isn't sufficient for making direct connections between peers without an intermediary server as the physical internet infrastructure doesn't support broadcasting very well.

Despite these limitations, learning how WebRTC works has unveiled a whole host of potential applications for development that I would not know how to implement if not for this project. For example, now I know how to set up a STUN/TURN server and WebRTC signalling server, I have the tools I need to create things like multiplayer games, video calling apps and so on which don't need costly server storage or high performance to operate. Not to mention, now knowing the basics of React Native (and by extension, React) I have an alternative way to develop mobile apps easily without the pains of traditional native development. This is something I may consider for use in other projects, especially if I ever need to quickly mockup a prototype.

Besides the technical knowledge learned that I can apply elsewhere, I once again discovered my own limitations and undersights. For instance, the initial plan for this project was significantly more optimistic than realistic, despite my thinking early on that it was reasonably scoped. Due to unforeseen difficulties with the implementation, delays happened - instead of conducting the research study and user testing in week 9, I ended up conducting it over 3 days in week 12. Thankfully, as I made the initial plan without any planned work for the 3 week Easter break, this time period allowed development to catch up and for the ethics application to get under way. In the end, everything worked out but now I know better than to assume things I consider easy will be completed within a week.

One useful tool that I used increasingly towards the end of the project was Asana, for task management; it ensured that even on days where I felt directionless, I could step through the tasks by priority. The greatest challenge was envisioning how this project would end and the overall structure of this report. Determining what is relevant and what is irrelevant or beyond the scope is difficult in broad projects like this, but the iterative Agile and spiral-prototyping design methodologies gradually solidified things. This is something I have found very reliable, as deadlines crawl closer I am forced to decide which aspects are most important and which have to be cut. With every project I undertake, I too iterate and improve upon my own sense of the time it takes to complete tasks. The practical experience of undertaking such projects makes the process clearer each time, allowing me to do more with the time I have - for instance, I could not have imagined fitting in the time for the research study on top of everything else a couple of years ago. Going forward I can take this experience to better plan and scope my work such that I can account better for potential delays and unforeseen consequences.

When it comes to meeting the aims of this project, I have shown that the prototype is generally usable, and contains many of the features of a social media platform existing users would expect. This is what I set out to do, getting a better idea of whether this concept would even be viable from both a user and technical standpoint. I think the concepts I explored in the implementation were appropriate for the scope of this project, but I could have gone further with things that make such a platform unique, such as fully implementing post visibility permissions.

Potentially it would have been better to narrow the scope further and just select a single core feature of social media platforms, such as messaging or activity feeds - and then iterate on that single area. This would have allowed me to get into more depth - but then again, I also wouldn't have been able to explore as many areas. I feel that this project has given me a stable foundation to build upon in more than a single area, which enables practical application of my new skills and knowledge in so many ways. This project is something that I can see myself working on more and still enjoying too - at least, after a break! This has been a great learning experience that has stimulated my interest in how the design of systems can impact society, with social media being just one of many aspects.

References

[1] Statista 2022. *Number of social network users worldwide from 2017 to 2025.* Available at: https://www.statista.com/statistics/278414/number-of-worldwide-social-network-users/ [Accessed: 13 May 2022].

[2] History.com Editors 2019. *Facebook launches.* Available at: https://www.history.com/this-day-in-history/facebook-launches-mark-zuckerberg [Accessed: 13 May 2022].

[3] Statista 2022. *Number of monthly active Facebook users worldwide.* Available at: https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/ [Accessed: 13 May 2022].

[4] Madory, D. 2021. *Facebook's historic outage, explained.* Available at: https://www.kentik.com/blog/facebooks-historic-outage-explained/ [Accessed: 13 May 2022].

[5] Lawler, R. 2020. *Cloudflare outage cuts off connections to Discord, DownDetector and others.* Available at: https://www.engadget.com/cloudflare-dns-problem-214413940.html [Accessed: 13 May 2022].

[6] SelfKey 2022. *Facebook's Data Breaches - A Timeline.* Available at: https://selfkey.org/facebooks-data-breaches-a-timeline/ [Accessed: 13 May 2022].

[7] Thomas, K. and Moscicki, A. 2017. *New research: Understanding the root cause of account takeover.* Available at: https://security.googleblog.com/2017/11/new-research-understanding-root-cause.html [Accessed: 13 May 2022].

[8] Tessian 2022. *30 Biggest GDPR Fines So Far (2020, 2021, 2022).* Available at: https://www.tessian.com/blog/biggest-gdpr-fines-2020/ [Accessed: 13 May 2022].

[9] Leonard, D. et al. 2007. On Lifetime-Based Node Failure and Stochastic Resilience of Decentralized Peer-to-Peer Networks. *IEEE/ACM Transactions on Networking* 15(3), pp. 644-656.

[10] Cole, S. 2022. *Switter, the Twitter for Sex Workers, Is Shutting Down.* Available at: https://www.vice.com/en/article/7kb7vx/switter-the-twitter-for-sex-workers-is-shutting-down [Accessed: 13 May 2022].

[11] UI-Patterns.com [no date]. *Design patterns: Social.* Available at: https://ui-patterns.com/patterns/social/list [Accessed: 13 May 2022].

[12] Solid 2022. *About Solid.* Available at: https://solidproject.org/about [Accessed: 13 May 2022].

[13] Statista 2022. *Most popular social networks worldwide as of January 2022, ranked by number of monthly active users.* Available at: https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/ [Accessed: 13 May 2022].

[14] Barry, E. 2022. *These Are the Countries Where Twitter and Facebook Are Banned.* Available at: https://time.com/6139988/countries-where-twitter-facebook-tiktok-banned/ [Accessed: 13 May 2022].

[15] Androidrank.org 2022. *Facebook.* Available at: https://www.androidrank.org/application/facebook/com.facebook.katana [Accessed: 13 May 2022].

[16] Google 2022. *Facebook.* Available at: https://play.google.com/store/apps/details?id=com.facebook.katana [Accessed: 13 May 2022].

[17] Apple 2022. *Facebook*. Available at: https://apps.apple.com/gb/app/facebook/id284882215 [Accessed: 13 May 2022].

[18] Johnston, M. 2022. 7 *Companies Owned by Google* (*Alphabet*). Available at: https://www.investopedia.com/investing/companies-owned-by-google [Accessed: 13 May 2022].

[19] Androidrank.org 2022. *YouTube*. Available at: https://www.androidrank.org/application/youtube/com.google.android.youtube [Accessed: 13 May 2022].

[20] Quinn, R. 2022. *Who Is The Owner Of WhatsApp? (Updated 2022)*. Available at: https://www.thecoldwire.com/who-is-the-owner-of-whatsapp/ [Accessed: 13 May 2022].

[21] Meta 2022. *Instagram from Meta*. Available at: https://about.facebook.com/technologies/instagram/ [Accessed: 13 May 2022].

[22] Kharpal, A. 2019. *Everything you need to know about WeChat — China's billion-user messaging app*. Available at: https://www.cnbc.com/2019/02/04/what-is-wechat-china-biggest-messaging-app.html [Accessed: 13 May 2022].

[23] Sapra, B. 2019. *This Chinese super-app is Apple's biggest threat in China and could be a blueprint for Facebook's future. Here's what it's like to use WeChat, which helps a billion users order food and hail rides*. Available at: https://www.businessinsider.com/chinese-superapp-wechat-best-feature-walkthrough-2019-12 [Accessed: 13 May 2022].

[24] UI-patterns.com 2022. *Activity Stream design pattern*. Available at: https://ui-patterns.com/patterns/ActivityStream [Accessed: 13 May 2022].

[25] Piar.io 2021. *How to Create a Link Preview for Social Media to Attract More Attention to Your Post*. Available at: https://piar.io/blog/how-to-create-a-link-preview-for-social-media-to-attract-more-attention-to-your-post [Accessed: 13 May 2022].

[26] Debatin, B. et al. 2009. Facebook and Online Privacy: Attitudes, Behaviors, and Unintended Consequences. *Journal of Computer-Mediated Communication* 15(1), pp. 83-108. doi: 10.1111/j.1083-6101.2009.01494.x.

[27] Cimpanu, C. 2020. *Hacker selling data of 538 million Weibo users*. Available at: https://www.zdnet.com/article/hacker-selling-data-of-538-million-weibo-users/ [Accessed: 13 May 2022].

[28] Criddle, C. 2020. *Facebook sued over Cambridge Analytica data scandal.* Available at: https://www.bbc.co.uk/news/technology-54722362 [Accessed: 13 May 2022].

[29] Whittaker, Z. 2017. *Twitter has a spam bot problem — and it's getting worse.* Available at: https://www.zdnet.com/article/twitter-spam-bot-problem-on-the-rise/ [Accessed: 13 May 2022].

[30] STUKAL, D. et al. 2022. Why Botter: How Pro-Government Bots Fight Opposition in Russia. *American Political Science Review*, pp. 1-15. doi: 10.1017/s0003055421001507.

[31] Whittaker, J. et al. 2021. Recommender systems and the amplification of extremist content. *Internet Policy Review* 10(2). doi: 10.14763/2021.2.1565.

[32] Chambers, A. 2021. *Twitter's new privacy policy was abused in predictable ways, experts say*. Available at: https://abcnews.go.com/Business/twitters-privacy-policy-abused-predictable-ways-experts/story?id=81515729 [Accessed: 13 May 2022].

[33] Kildiş, H. 2020. *Post-Truth and Far-Right Politics on Social Media.* Available at: https://www.e-ir.info/2020/11/17/post-truth-and-far-right-politics-on-social-media/ [Accessed: 13 May 2022].

[34] Boyd, C. 2021. *Discord scammers lure victims with promise of free Nitro subscriptions.* Available at: https://blog.malwarebytes.com/scams/2021/10/discord-scammers-lure-victims-with-promise-of-free-nitro-subscriptions/ [Accessed: 13 May 2022].

[35] Heath, A. 2022. *A Facebook bug led to increased views of harmful content over six months.* Available at: https://www.theverge.com/2022/3/31/23004326/facebook-news-feed-downranking-integrity-bug [Accessed: 13 May 2022].

[36] lostinlight 2022. *Fediverse in 2021.* Available at: https://fediverse.party/en/post/fediverse-in-2021/ [Accessed: 13 May 2022].

[37] joinmastodon.org 2021. *Running your own server.* Available at: https://docs.joinmastodon.org/user/run-your-own/ [Accessed: 13 May 2022].

[38] IPFS.io [no date]. *What is IPFS?* Available at: https://docs.ipfs.io/concepts/what-is-ipfs/ [Accessed: 13 May 2022].

[39] McKetta, I. 2021. *Despite All Odds, Global Internet Speeds Continue Impressive Increase.* Available at: https://www.ookla.com/articles/world-internet-speeds-july-2021 [Accessed: 13 May 2022].

[40] Lim, S. 2019. *Average Storage Capacity in Smartphones to Cross 80GB by End-2019.* Available at: https://www.counterpointresearch.com/average-storage-capacity-smartphones-cross-80gb-end-2019/ [Accessed: 13 May 2022].

[41] Cluster, C. 2021. *What is a UUID, and Why Should You Care?* Available at: https://www.cockroachlabs.com/blog/what-is-a-uuid/ [Accessed: 13 May 2022].

[42] Shin, F. 2022. *What's behind China's cryptocurrency ban?* Available at: https://www.weforum.org/agenda/2022/01/what-s-behind-china-s-cryptocurrency-ban/ [Accessed: 13 May 2022].

[43] Associated Press 2021. *Signal: China appears to have blocked encrypted messaging app.* Available at: https://www.theguardian.com/world/2021/mar/16/signal-blocked-china-encrypted-messaging-app [Accessed: 13 May 2022].

[44] bankmycell 2022. *How Many People Have Smartphones Worldwide (May 2022).* Available at: https://www.bankmycell.com/blog/how-many-phones-are-in-the-world [Accessed: 13 May 2022].

[45] Vergne, J. 2020. Decentralized vs. Distributed Organization: Blockchain, Machine Learning and the Future of the Digital Platform. *Organization Theory* 1(4) doi: 10.1177/2631787720977052.

[46] Riff, N. 2019. *How Bitcoin's Shaky Reputation Is Limiting The Spread of Blockchain.* Available at: https://www.investopedia.com/how-bitcoin-s-shaky-reputation-is-limiting-the-spread-of-blockchain-4690557 [Accessed: 13 May 2022].

[47] Statista 2019. *Daily time spent online by device 2021.* Available at: https://www.statista.com/statistics/319732/daily-time-spent-online-device/ [Accessed: 13 May 2022].

[48] Kim, P. 2022. *What are the environmental impacts of cryptocurrencies?* Available at: https://www.businessinsider.com/personal-finance/cryptocurrency-environmental-impact [Accessed: 13 May 2022].

[49] Bluetooth.com 2018. *Bluetooth market update.* Available at: https://www.bluetooth.com/wp-content/uploads/2019/03/Bluetooth_Market_Update_2018.pdf [Accessed: 13 May 2022].

[50] Gavrikov, P. 2021. *No, Bluetooth's range is not 10m.* Available at: https://medium.com/geekculture/no-bluetooths-range-is-not-10m-393b322de31e [Accessed: 13 May 2022].

[51] Cloudflare [no date]. *What is DNS?* Available at: https://www.cloudflare.com/en-gb/learning/dns/what-is-dns/ [Accessed: 13 May 2022].

[52] Cloudflare [no date]. *What is a domain name registrar?* Available at: https://www.cloudflare.com/en-gb/learning/dns/glossary/what-is-a-domain-name-registrar/ [Accessed: 13 May 2022].

[53] Yu, L. 2018. NAT: *Why Do We Need It?* Available at: https://medium.com/@laurayu_653/nat-why-do-we-need-it-f0230bb7d06f [Accessed: 13 May 2022].

[54] Vaughan-Nichols, S. 2019. *Static vs. Dynamic IP Addresses.* Available at: https://www.avast.com/c-static-vs-dynamic-ip-addresses [Accessed: 13 May 2022].

[55] Berners-Lee, T. 2020. *solid.community and solidcommunity.org.* Available at: https://lists.w3.org/Archives/Public/public-solid/2020Oct/0020.html [Accessed: 13 May 2022].

[56] Facebook [no date]. *Friending.* Available at: https://www.facebook.com/help/1540345696275090 [Accessed: 13 May 2022].

[57] Olang, K. [no date]. How to Give Someone a Link to Your Facebook Profile. Available at: https://smallbusiness.chron.com/give-someone-facebook-profile-30149.html [Accessed: 13 May 2022].

[58] Moreau, E. 2021. *How to Add Friends to Snapchat by Scanning Their Snapcodes.* Available at: https://www.lifewire.com/add-friends-by-scanning-their-snapcodes-3486002 [Accessed: 13 May 2022].

[59] Vasile, C. 2018. *The reason Apple pulled Telegram was more serious than we thought.* Available at: https://www.phonearena.com/news/Telegram-pulled-due-to-distribution-of-illegal-content_id102246 [Accessed: 13 May 2022].

[60] Nast, C. 2021. *How the far right took over Steam and Discord.* Available at: https://www.wired.co.uk/article/steam-discord-far-right [Accessed: 13 May 2022].

[61] Karanje, A. 2021. *Decentralized Social Media Platforms Aren't New, They Just Aren't Popular.* Available at: https://medium.com/swlh/decentralized-social-media-platforms-arent-new-they-just-aren-t-popular-2a3bf64a9d51 [Accessed: 13 May 2022].

[62] Opera Team 2020. *WhatsApp Web Knowledge Hub and FAQ.* Available at: https://blogs.opera.com/tips-and-tricks/2020/05/whatsapp-knowledge-hub-and-faq [Accessed: 13 May 2022].

[63] Pappalardo, G. et al. 2018. Blockchain inefficiency in the Bitcoin peers network. *EPJ Data Science* 7(1). doi: 10.1140/epjds/s13688-018-0159-3.

[64] Target Internet [no date]. *Social Media Spam Bots and Fake Engagement.* Available at: https://www.targetinternet.com/social-media-spam-bots-and-fake-engagement/ [Accessed: 13 May 2022].

[65] Berryman, C. et al. 2017. Social Media Use and Mental Health among Young Adults. *Psychiatric Quarterly* 89(2), pp. 307-314. doi: 10.1007/s11126-017-9535-6.

[66] Knispel, S. 2020. *Getting fewer 'likes' on social media can make teens anxious and depressed.* Available at: https://www.rochester.edu/newscenter/getting-fewer-likes-on-social-media-can-make-teens-anxious-and-depressed-453482/ [Accessed: 13 May 2022].

[67] cypherpunk 2019. *Mastodon.social is blocked in China.* Available at: https://mastodon.social/@cypherpunk/102085326935549688 [Accessed: 13 May 2022].

Appendices

Appendix A

Source code & app build

See the separate archive file named “Source.zip”.

GitHub code repository

https://github.com/SpectralCascade/peerway

Last relevant GitHub commit

All code written in this commit and previous commits are relevant to this report. Any newer code commits are not relevant to this report.

SHA code: ded585104b5ca7adcdc36e40bcdbcb0343296dd2
Link:
https://github.com/SpectralCascade/peerway/commit/ded585104b5ca7adcdc36e40bcdbcb0343296dd2

Appendix B

Implementation screenshots

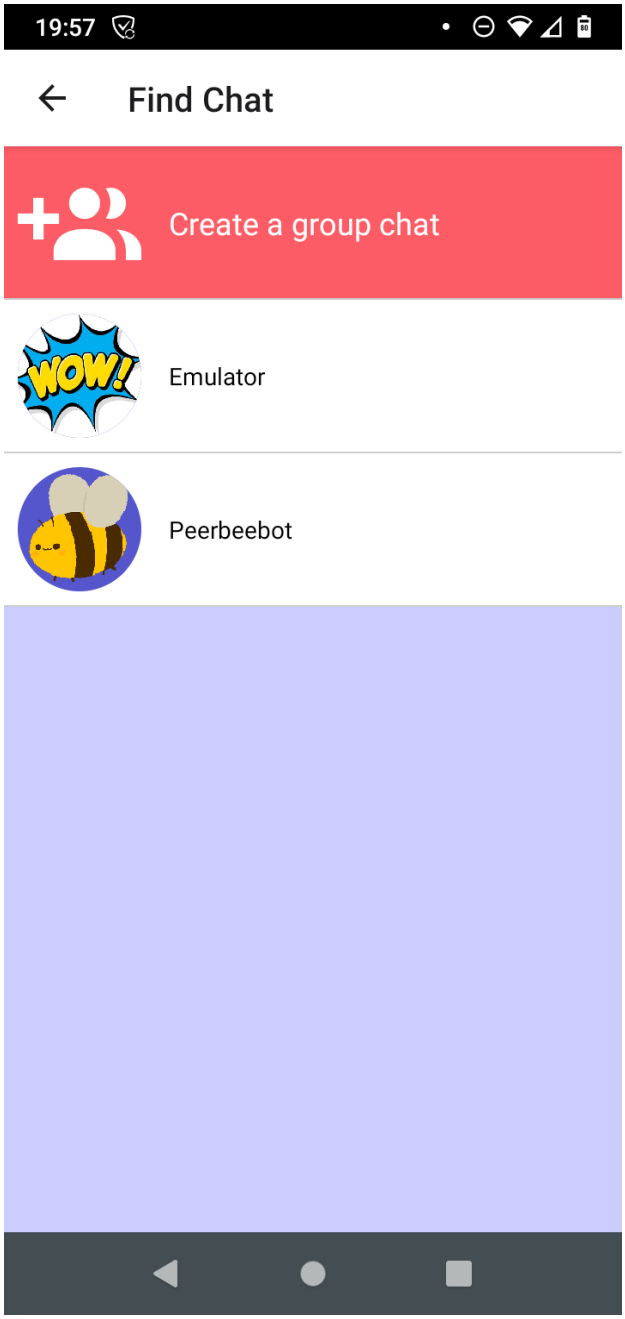

FIGURE 6.5.2: *Screenshot of the new private chats screen*

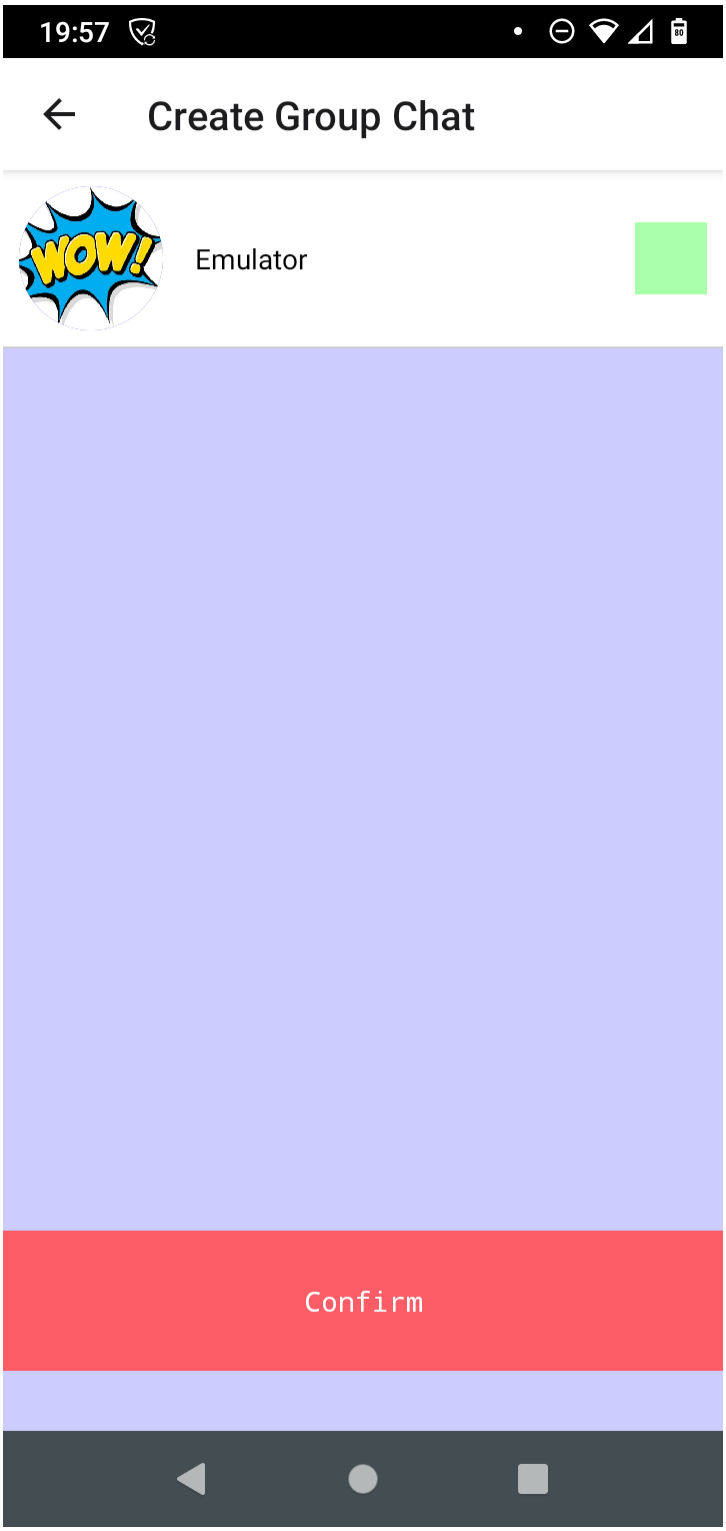

FIGURE 6.5.3: *Screenshot of the new group chat screen*

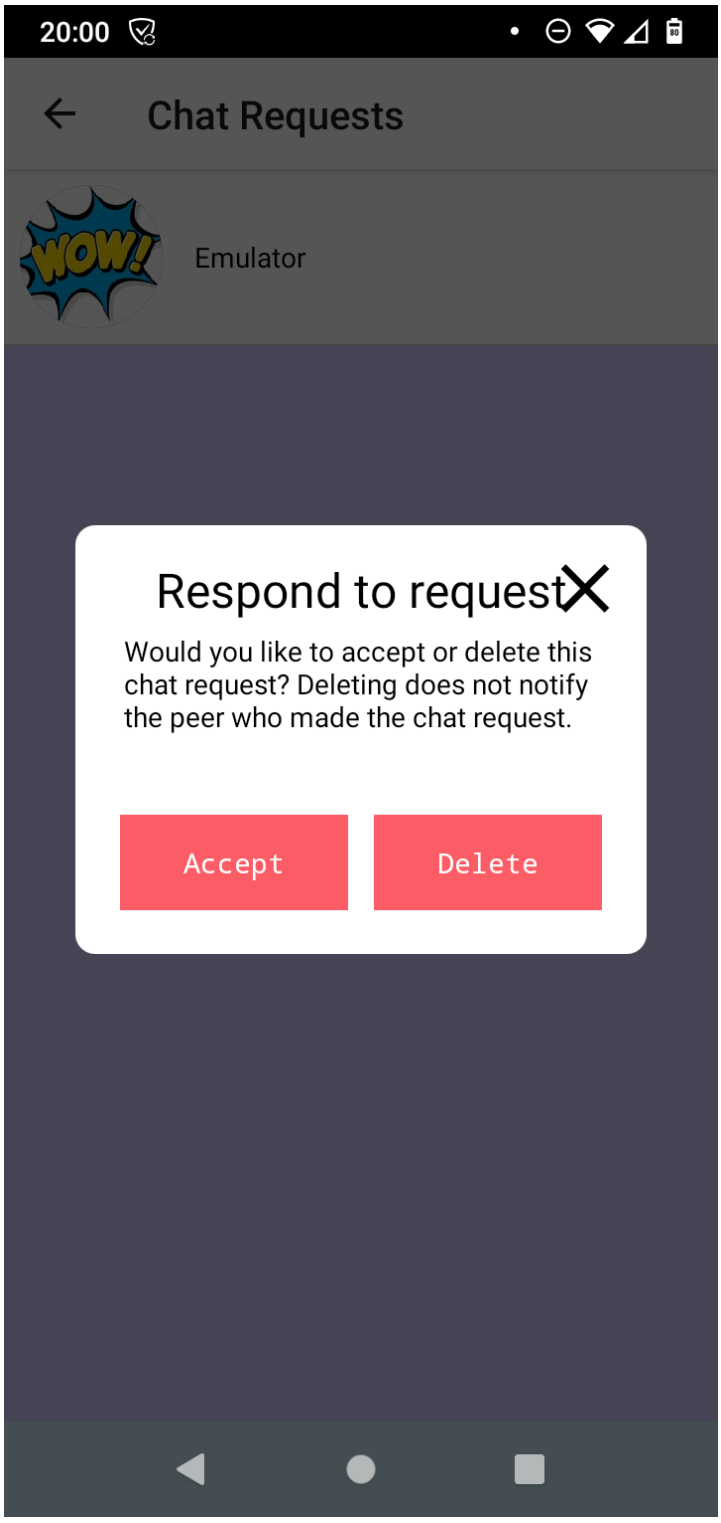

FIGURE 6.5.4: *Screenshot of the accept or delete chat request popup*

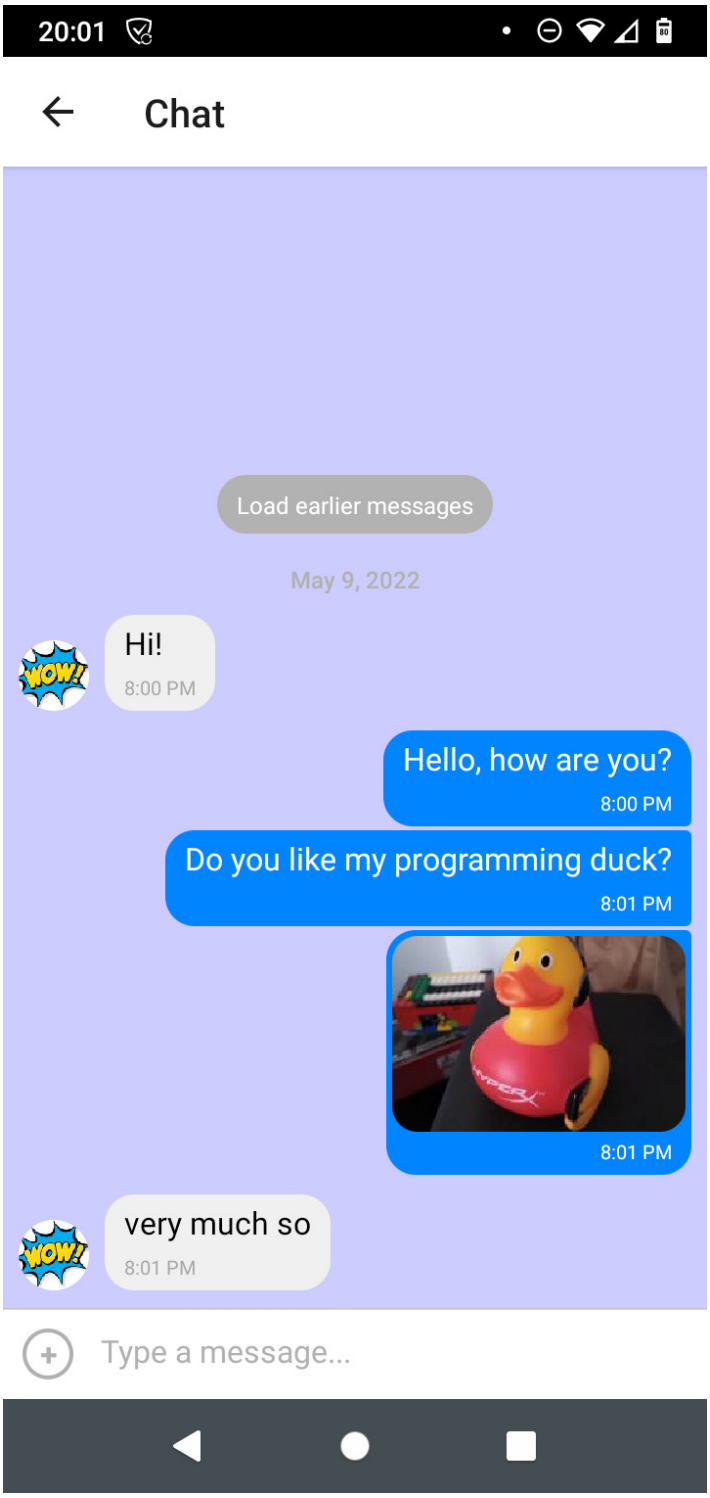

FIGURE 6.5.5: *Screenshot of the chat screen with some messages*

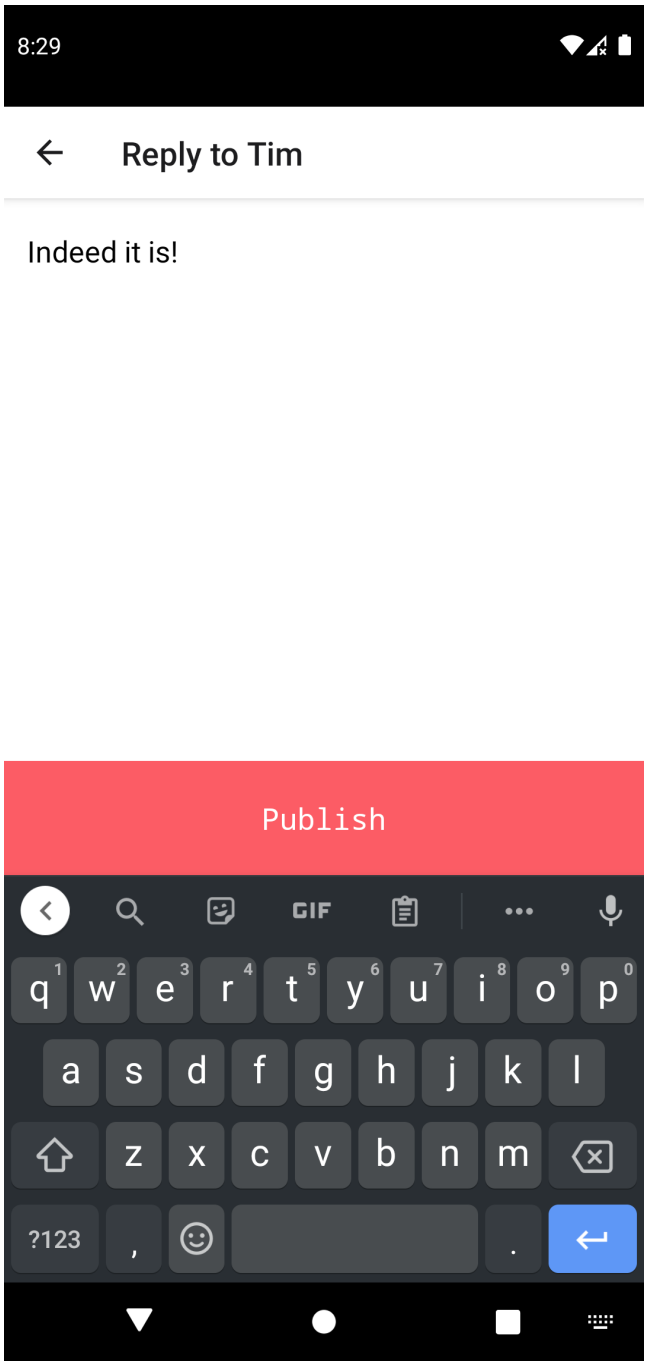

FIGURE 6.6.2: *Screenshot of the new/edit post screen*

FIGURE 6.7.2: *The profile editing screen*

FIGURE 6.7.3: *The app setup screen*

Appendix C

Research webpage, participant information sheet and user testing build

https://timlanesoftware.com/peerway/

The participant information sheet is linked in the above webpage. The webpage also links to a download of the Android app build used for the user testing. Please note that the Google Forms link on the webpage is no longer accepting responses.

Research & user testing questions

See separate appendix file named “Appendix C - Research Questions.pdf”.

Research integrity training certificate

See separate appendix file named “Appendix C - Research Training Certificate.pdf”.