Universitetet i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR THESIS

Study programme / specialisation: Computer Science	The spring semester, 2022 Open / ~~Confidential~~
Author: Anders Ringen	.....Anders Ringen..... (signature author)
Author: Lars Andreassen Jaatun	..... (signature author)
Supervisor(s): Chunming Rong Supervisor(s): Jiahui Geng	
Thesis title: Just Us: A Blockchain-based Privacy-friendly Social Network	
Credits (ECTS): 20	
Keywords: Blockchain, Social Network, Hyperledger Fabric, React Native, Golang, Mobile Application	Pages: 65 + appendix: 4 pages Stavanger, 15 May 2022

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Just Us: A Blockchain-based Privacy-friendly Social Network

Bachelor's Thesis in Computer Science
by

Anders Ringen
Lars Andreassen Jaatun

Internal Supervisors

Chunming Rong
Jiahui Geng

May 15, 2022

"If what you've done is stupid but it works, then it really isn't that stupid at all."

David Letterman

Abstract

This report details the creation of a social network using react-native and Hyperledger Fabric to enable a higher level of control over user data. The end result shows that there is a potential application for blockchain for social networks, though it cannot control how the service provider for the social network service treats the users' data on the way through the network.

Acknowledgements

We would like to thank Chunming Rong and Jiahui Geng for supervising the project and giving important advice.

We would also like to thank Martin Jaatun for proof reading and formatting help.

Contents

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
CA	**C**ertificate **A**uthority
CLI	**C**ommand **L**ine **I**nterface
ETH	**ETH**er
HLF	**H**yper**l**edger **F**abric
JSON	**J**ava**S**cript **O**bject **N**otation
JWT	**J**SON **W**eb **T**oken
REST	**RE**presentational **S**tate **T**ransfer
TTP	**T**rusted **T**hird **P**arty
UI	**U**ser **I**nterface

Chapter 1

Introduction

Social media today allows you to upload, share and comment on content of all kinds, whether it be a famous actor's bad take on proper etiquette or the latest arrest by your local police. When you post something on a social media platform, you essentially donate a whole load of personal data to the owner of the social medium. After you post something, there is no telling what happens to the data. Even though a user of a social network should own their own data, ownership is transferred to the owner of the network the moment you post it.

1.1 Problem Definition

This project aims to create a social network that allows users to have a higher level of control over the data they upload. Users should be able to track what path their data takes, and eventually moderate how far a post can propagate through the network.

1.2 Use Cases

We will use blockchain technology to enable the tracking of information. There are aspects of blockchain that can provide solutions to problems with regular centralized social apps. Blockchain technology can help verify the origin of data as well as restrict unauthorized access, or at the very least maintain a record of unauthorized access, and help social applications move away from a centralized model of operation. This is due to the fact that all operations on assets stored in the blockchain are recorded in the ledger as transactions, and can therefore be audited.

1.3 Challenges

Traditional Social media earns revenue from selling information on its users, and from letting other companies display advertisement to a targeted audience. An application with stricter control over personal data will not have this income, which makes the maintaining of servers and other expenses harder pay for. Moreover, this could necessitate direct payment to participate in the network. As is apparent in the amount of users of social media today, "people in general" do not seem to care enough about their personal data for it to be possible to implement payment for participation.

1.4 Contributions

The Project creates a foundation for social media supported by blockchain technology. We have created chaincode that can interact with the ledger to read both historical information and the current state of the ledger. This data is fetched to a mobile application written in react-native through an API that interfaces with the network. This enables users to track how their data travels through the network, and who has access to your posts.

1.5 Outline

The remainder of this thesis is structured as follows: Chapter 2 describes the background knowledge procured for the project. Chapter 3 describes the project development process, as well as detailing the chosen project design. Chapter 4 is an evaluation of the final product, and Chapter 5 is a discussion on the choices and strategies employed in the project. Chapter 6 contains the conclusion for the thesis.

Chapter 2

Background

Before we could start the project, we needed to establish a foundation of knowledge within blockchain technology, as well as exploring other projects to gain an understanding on how others have applied the concepts that we are basing this project on.

2.1 What is Blockchain?

To put it in simple terms, a blockchain is an unchangeable transaction history. Individual transactions between participants in the network known as peers are grouped together in blocks with a hash referring to the previous block and a timestamp for when the block was created. The transactions in question may consist of transferring assets from one member of the network to the other (like transferring bitcoins or Ether), or changing the state of some asset owned by a member (through custom smart contracts).

Bitcoin [1] and Ethereum [2] are two famous examples of public blockchains. Their blockchain is available to anyone who is interested, all members have equal access to the blockchain, and all members participate in validating the order of transactions (otherwise known as "consensus"). Some of the controversy tied to blockchain comes from consensus protocols, especially in the case of Bitcoin and Ethereum. Their current consensus protocol ("proof-of-work") uses a lot of processing power and rewards the users proportionally to how much processing power they allocate for the consensus protocol. This is what is referred to as mining. A study from 2020 estimated that for every USD of bitcoin created in the USA, there were environmental damages equivalent to almost 0.5 USD [3]. When a new block is added to the blockchain it contains a reference to the preceding block. If you wanted to change the transaction history within the previous block, due to the hash referring to the previous block you would have to generate a new

block for the subsequent block. In addition to this, in bitcoin the majority accepts the longest chain as the valid chain, so you would have to perform this work faster than the rest of the network; you would need more than 50% of all the physical mining resources in the network. This is known as the "51% attack". The reason this is generally accepted as a non-risk, is that it is deemed less profitable to spend your resources attacking the blockchain than actually just mining.

Other than the desire for a decentralized currency without the fees that come with centralized banking, the use cases of blockchain center around smart contracts which empower organizations and corporations to formulate contracts as code which fulfill themselves automatically. This could result in faster fulfillment (less bureaucracy = more efficacy), as well as decreased costs (no Trusted Third Party (TTP) = no extra fee).

2.1.1 What is “Proof-Of-Work”?

“Proof-of-work” is the consensus protocol used to validate, order and assemble Bitcoin and (at the time of writing) Ethereum transactions into blocks for the blockchain.

Bitcoin is a public permissionless blockchain. This means it is accessible to anyone wishing to connect to it, in contrast to a permissioned blockchain where you have to be added by an administrator of the network to gain access. When everyone is free to connect to and make transactions on the network, to avoid duplicate transactions and other malicious acts you need a way to be able to verify and validate correct transactions. This is what proof-of-work aims to do.

When proposing a block you first assemble some transactions, the hash of the previous block, and a nonce, which is a number that has no purpose other than to change the hash of the block. If we take bitcoin as an example, transactions would consist of transfers of bitcoins from one wallet to another. The hash is an identifier that is created by giving all the fields of the block, i.e. the transactions, hash for the previous block and the nonce, as input for a special kind of function called a hash function. The hash function produces a unique alphanumeric string. To create the block, you must find a hash for the block that begins with a certain number of 0 bytes. The amount of zeros is denoted as the difficulty of the hash. The more zeros, the heavier the computation. To find a hash that satisfies the difficulty requirement, the computer must increment the nonce, and run the proposed block through the hash function again. There might be multiple proposed blocks for each addition to the blockchain, but the first block that satisfies the difficulty is added. While it is possible to have multiple parallel blockchains, in the long run the longest chain will prevail. Parallel blockchains are known as forks, and are generally undesirable. This makes the act of adding a block to the blockchain a democratic process

between all members of the blockchain network, in that the majority of processing power decides the next block[1].

2.1.2 What is "Proof-Of-Stake"?

"Proof-of-stake" is another consensus mechanism for blockchain. "Proof-of-stake" is at the time of writing not in widespread use, but is included in the roadmap for Ethereum 2.0 [4].

As with "proof-of-work", "proof-of-stake" is a mechanism to establish a consensus about the order and validity of transactions within the blockchain, however the approach is different. Instead of awarding the block to the agent with the most computing power, the miners "stake" at least 48 ETH (at the time of writing equivalent to over €10 000) to gain the ability to validate blocks. The result of relying on a monetary stake of assets for block validation instead of processing power is to further de-incentivize fraud, as you can end up losing your stake. Additionally, the blockchain is designed to make it more lucrative play by the rules than against them. If you do not have enough ETH to fulfill this requirement, you can join a pool together with other miners.

"Proof-of-stake" differs from "proof-of-work" in that there is no significant computing power expended to create and validate blocks. Nodes that stake their ETH are randomly chosen to either create or validate blocks, without all the hassle of calculating a difficult hash.

The 51% attack is perhaps the biggest problem with public consensus mechanisms and there is at the time of writing no way to prevent this completely, multiple successful attacks have been committed on various public blockchains [5]. With "proof-of-stake", the 51% attack becomes less likely than with "proof-of-work" because of the penalties that can be doled out to peers who endorse malicious blocks. If a peer in the blockchain endorses a malicious block, the peer risks losing their entire stake, which as previously mentioned is a lot of money. In addition to this, "proof-of-stake" penalizes being disconnected, thereby increasing availability of the system.

These are the main arguments for switching over to "proof-of-stake" for Ethereum, but it has yet to be proven that it works in real life. At the time of writing, there are no blockchains on the scale of Ethereum and Bitcoin that use "proof-of-stake", so whether the switch pays off or not will be interesting to see.

2.1.3 What is a Smart Contract?

Within blockchain technology, one of the most attractive points for business applications are smart contracts. They enable businesses to write code instead of physical contracts, and these coded contracts are automatically fulfilled when conditions agreed upon by both parties are fulfilled. This has the potential to decrease costs for both parties, as well as time spent on bureaucracy. This however is not the only use case for smart contracts [6].

As well as being used for legal contracts between corporations, smart contracts can be applied in numerous fashions within blockchain networks to run distributed applications on multiple clients.

Smart contracts are not limited to transferring assets (e.g. cryptocurrency) from one party to the next, but can also be used to change states of programmatic objects defined within the application. A good example for this can be found in the hyperledger fabric documentation [7]. They describe a car ownership smart contract. In the application, there are multiple cars owned by different people. One person initiates a transaction to buy a car, and another person initiates a transaction to sell this car to the person. The smart contract takes the input from the peers and completes the trade by changing the state of the car object, changing ownership from one person to the next, all without making any physical transfers between users of the application.

2.2 Hyperledger Fabric

Hyperledger Fabric is an open source project that started development in 2015 and became an official part of the Hyperledger Foundation [8] in 2016 which again was started by the Linux Foundation [9]. Fabric was created to enable blockchain technology for use in a business context with strict criteria on privacy, confidentiality and auditability, which are difficult to combine within the same project. Nevertheless, the project turned out quite successful [10]. HLF differentiates itself from the aforementioned blockchain networks through being a permissioned network, requiring participants to be authenticated to join. HLF networks are built up of channels. Channels are separate sections of the network with their own ledger and their own members independent of the other channels in the network. Different members have different privileges depending on the role they have been assigned in the network, such as administrators with permissions to change configuration options and commit transactions, or members that have read-only permissions. Nodes that act as access portals to the channels for the members are known as peers, and they too have roles defined by the configuration that differ from the roles given to users.

2.2.1 Configuration

HLF is configurable to a large degree. One HLF network will most likely be different to any other network you can find. The configuration options range from endorsement policies within a channel, to orderer configuration, whether there are many orderers or a single orderer per channel, and how many channels you want for your network. The configuration isn't static however; if you have the right permissions(i.e. you are a channel administrator) you can send a proposal for a configuration change, and the change will be accepted depending on the existing policies.

Endorsement Policy

The endorsement policy describes which members from which organizations in the channel must "endorse" or approve smart contract transactions [11]. If the transaction is not endorsed as required, the transaction is not accepted and placed into a block.

Channel Configuration

The channel configuration is stored in a collection of transactions that is normally abbreviated as configtx [12]. The channel configuration contains settings for the organizations present in the channel, for the peers that will participate from the different organizations, as well as configuration for the ordering service in the channel.

2.2.2 The Ordering Service

As previously mentioned, permissionless blockchain networks typically have consensus mechanisms that not only require the participation of a large proportion of network members, but also take time, are very power-hungry, and typically result in some nodes doing a lot of work for no payoff. HLF instead uses orderer nodes for ordering and assembling blocks that can be added to the blockchain. A group of orderer nodes form an ordering service, that in addition to assembling blocks to the blockchain manage basic access control for channels, and enforce policies for validation of transactions created by the administrators of the organizations participating in the network [13]. When the ordering service receives enough transaction proposals that are endorsed according to the endorsement policy to fulfill the minimum requirements to constitute a block, it packages it into a block which is then distributed to all peers in the channel. Using an ordering service instead of using all peers for endorsement and ordering of transactions results in higher throughput of transactions, as well as completely avoiding forks in the ledger.

As with all other nodes, orderers are owned by the organizations participating in the channel.

Raft Protocol

HLF supports multiple protocols for consensus among multiple ordering nodes, but recommends the use of the Raft protocol, due to it being simpler to set up and manage for network architects[13]. The implementation is crash fault tolerant, which means that the service can reach consensus even if some orderer nodes fail, but it is not Byzantine fault tolerant [14], which means the service cannot reach (reliable) consensus with malicious nodes in the ordering service. Therefore one should avoid sharing a channel with untrusted organizations.

2.2.3 Permissioned

HLF is a permissioned blockchain network. That means that instead of being openly accessible to anyone, you need to be authenticated to gain access, typically with an identity issued by an administrator in the network. In addition to users having to be authenticated, users can be given roles that span beyond just whether you are verified or not; for instance whether you have administrative rights in a certain channel, or that you have read access to certain channels but not the right to submit transactions.

2.2.4 Identity

Identity in HLF is most commonly managed using digital identities in the form of digital certificates issued by a trusted certificate authority (CA). Fabric provides binaries for a certificate authority called fabric-CA. Within channels, there are membership service providers that decide what certificates have access to the ledger. The HLF documentation likens the relationship between certificate authorities and membership service providers to credit cards and credit card terminals, respectively. The example is that some card terminals may accept only Visa, while others only American Express, and others again may accept both American Express as well as Visa [15].

2.2.5 Consensus

While other blockchain networks use protocols like proof-of-work and proof-of-stake to verify and commit blocks to the blockchain, HLF uses an ordering system with specialized

nodes in the network called orderers. They create blocks from transactions according to a policy formulated by the respective businesses represented in the network, such as "organization 1 and organization 2 must endorse these transactions for them to be valid". When all conditions listed in the policy are fulfilled, the orderers are free to create a new block and add it to the blockchain.

2.2.6 Smart Contracts

While smart contracts in Fabric function in a similar manner to what was outlined above, Fabric makes some additional definitions in the documentation [7]. Chaincode is used interchangeably with the word smart contract, but they have slightly different meanings. A smart contract is a collection of functions that carry out terms agreed upon by all parties participating in the transaction. A chaincode on the other hand is a package of multiple smart contracts, and is what you deploy to a channel.

2.3 Existing Approaches

There are currently multiple social networks that tout a blockchain-backed technology stack. Steemit [16], Somee [17] and Sapien [18] are examples of social networks that are all based on blockchain-technology. While Steemit and Sapien seem to be reputable, or at the very least have good intentions, there is evidence pointing towards Somee being a scam [19]. This is also important to showcase because of the prevalence of cryptocurrency related scams based on encouraging large investments into the product while promising huge payouts, only to liquidate their own (often substantial) stake, causing the value to drop significantly. Steem, Sapien and Somee all have the same focus which is the monetization of content through their own cryptocurrency.

Both Sapien and Steemit boast a virtually censorship-free social network as an alternative to censorship-heavy centralized platforms. However, in their respective white papers outlining the technology stack, there is barely a mention of privacy, or a user's right to their own data [20, 21]. While Sapien's marketing lead of 2020, Jonathan Goodwin, wrote a piece on how users are entitled to their own data [22], the piece is accompanied by a disclaimer that clarifies that the piece does not reflect the official position of the Sapien social network, even though it is written to sound like it is. So while the platforms may better accommodate for free speech, users still do not own their data.

2.3.1 Ushare

Ushare is a proposition for a decentralized social network described by Antorweep Chakravorty and Chunming Rong [23]. This design also utilizes blockchain technology, but in ways that differs from the approaches described above. The most significant usage of blockchain technology here is not the monetization of social interactions, but not unlike our project the tracking and restriction of data sharing throughout the network. To achieve this, they create a token associated with each post uploaded to the network. On every sharing operation, this token decreases, until it reaches zero whereupon it cannot be shared further.

Ushare argues that the inherent properties of public blockchains like anonymity, resistance to censorship and inherent decentralization makes for an excellent candidate to host a social network with. In addition to this, the paper describes a system for distributed storage of larger files for predictable growth of the blockchain. This load-sharing technology would mainly be for video posts. To control the sharing process itself, the paper describes a "turing-complete relationship system", which according to their description is equivalent to a smart contract as described above. Ushare introduces interesting ideas that are applicable to our project. While storing the data off the chain might be good for performance, it could also be beneficial from a security standpoint in that it could make it possible to store data with the user instead of in a server that is publicly available.

Chapter 3

Project Description

Our primary focus when beginning the project was to create a system that enabled the tracking of data. By using blockchain to track data, the application would give the user a fundamentally different level of control with regard to personal data, i.e. the data shared with others through the network. After a brief investigation of available documentation and functionality we concluded that HLF would suit the project. Considering that the project does not seek to create a completely peer-to-peer solution, the architecture of HLF suits the applications requirements. While we briefly considered hyperledger Sawtooth [24] as an alternative framework for the project, we quickly decided against it due to the lack of support and documentation presently available for Sawtooth.

The project development was separated into frontend and backend development, and each team member was assigned a main area of responsibility from the two branches.

3.1 Development Process

The project plan followed an agile process. This means that the participants of the project decided together on tasks that needed to be done within a predefined time frame in a planning phase and "pulled" tasks as needed, either when stuck or finished with another task. Whenever the defined tasks had been reached, the participants would need to have a new meeting to discuss and plan the next step, and continue pulling tasks as needed. To help with this, the project is hosted on Github, and is therefore maintained using git. Github already has infrastructure to create tasks (called issues on Github), as well as assignment and completion of these tasks.

To speed up development, the project aims to make use of other development tools made available by Github, like Github Actions for continuous integration and continuous

deployment. In a production environment, Github Actions enable developers to seamlessly test code on a fresh machine, as well as automatically creating requests to merge development branches into stable branches if tests pass, and then eventually with human approval allow you to deploy a new stable build with only the minimum necessary human interaction.

We decided not to aim for complete test coverage, instead opting to write fewer tests in total, but instead more integration tests.

3.2 Proposed Solutions: Just Us

We propose the application **Just Us** as a solution to the problem statement of the thesis. **Just Us** will use Hyperledger Fabric to enable users to both claim a higher level of ownership over their own data, as well as determining the path their data travels throughout the network.

Learning how to use HLF has many facets, like the obvious design of smart contracts and discovering how to create an interface between the blockchain and an external application. In addition to this, there is the question of the network design; what policies to define for organizations, endorsement policies in channels, and ordering service design.

To alleviate some of these concerns, the project utilizes the test network provided by the HLF developer team. The test network contains two peer organizations and one ordering organization. It is deployed to an isolated docker compose network and is designed to test new applications and smart contracts. There are not many choices that can be made when building the test network, but one thing developers can control is the database used to address the ledger. You can choose between LevelDB and CouchDB, the difference being in pure performance and the complexity of queries supported. For our project we opted to go with LevelDB, as it is the fastest of the two options, though it only supports key-value queries.

For consistency purposes, both the chaincode and any other components for the backend portion of the application should be written in Go. The application should expose an API that a mobile application can interface with, as well as a HLF network that communicates with the API.

The application should be able to track posts made by the users, and manage access control to different users' posts. How other users access a users post should be recorded in the ledger as transactions, and access to data should be managed by the user that

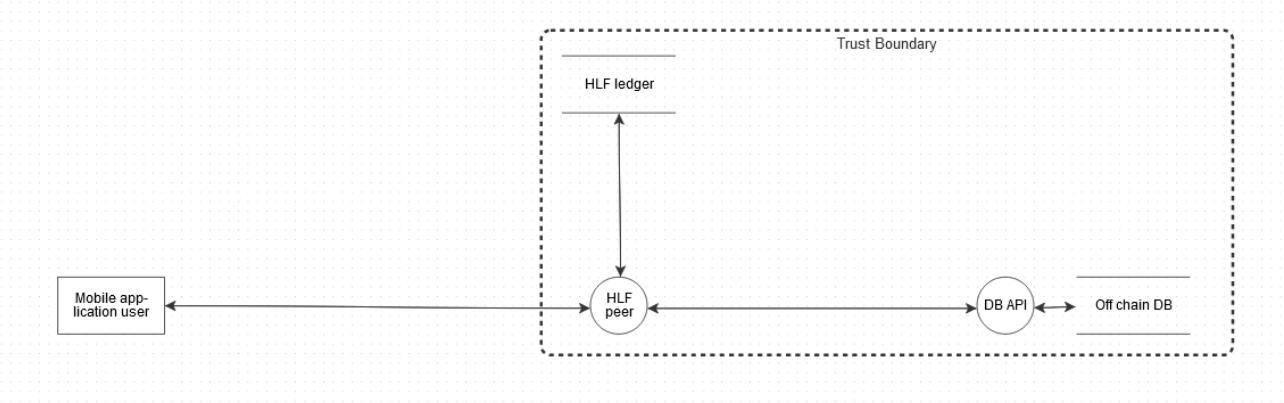


Figure 3.1: Data flow diagram for proposition 1

owns the data. Users should be able to retract another user's permissions to view their content, and otherwise accurately regulate access to their data.

Instead of storing all data on the blockchain, data should be stored off-chain on an external database only accessible to the backend server. This creates more predictable growth of the size of the blockchain since all assets persisted on the ledger are of the same size rather than ranging from the size of a photo to the size of a video, as well as making it easier to upload different types of content such as photos or videos to the application due to the blockchain only storing a hash of the post, and seeing no difference between a text post and a video.

Below are three propositions for an application structure that would fulfill the requirements detailed above.

3.2.1 Proposition 1

The first proposition is to create a backend application that is completely decentralized, in that it can function separately from a larger, global network. This would enable anyone with enough physical resources and adequate availability to create a local "station" that will act as a hub for other users of the application. A data flow diagram for a setup consisting of one user is shown in figure 3.1. The integration between the API and the fabric chaincode would be very tight, potentially allowing for chaincode execution directly from the mobile application. This could be achieved with the use of a library created by Bityoga[25] for invoking chaincode from a mobile application written in react native. Identity for chaincode access would be managed directly on the mobile device, meaning certificates that authenticate the user would be stored on the phone. The fabric network would be backed by an API that had no other job than to serve data stored outside the blockchain, such that access to data is dictated by the certificates stored in the frontend

application. The local "station" would also be associated to a local database that is set up to communicate with the local fabric nodes. All data on the database would be encrypted with the certificates of the owner, so all data on the database is worthless if the owner does not give any other user access. Data for a single user such as hashes referring to posts would be stored in a personal channel where the user is appointed as admin, giving the user the right to assign read permissions for other people as well as revoking permissions using only the built in features of HLF. Access to individual posts (such as through the sharing of posts) could be made through an access control list generated by a smart contract. A smart contract could investigate whether the person accessing the post either has direct access to the channel (i.e. is a follower of the owner of the channel), or follows another user that has been allowed to share the post.

The assets stored on the ledger are posts, with the minimal data fields for these posts being "owner","ACL-list" and "postId". "owner" would be the original poster of the post, "ACL-list" would be a list of users that have been granted sharing rights to the post and "postId" is the hash referring to the data stored in the off-chain application.

3.2.2 Proposition 2

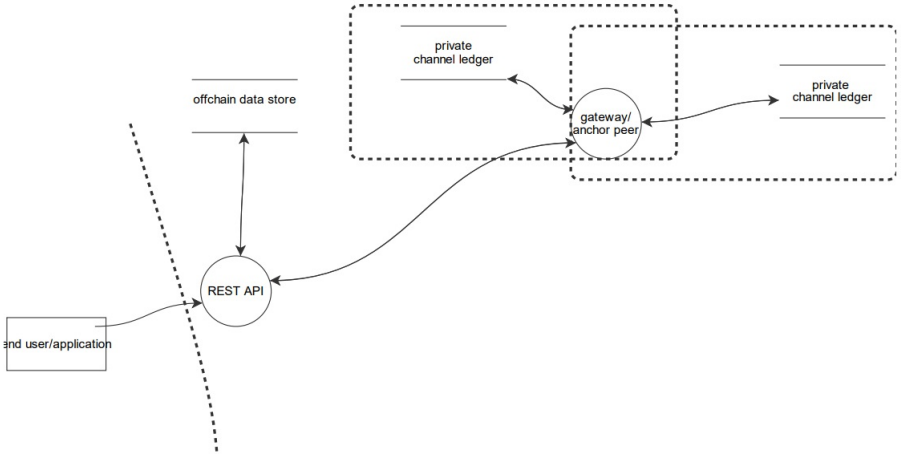


Figure 3.2: Data flow diagram for proposition 2

The second proposition is to create a REST API for interfacing with a mobile application and interacting with a database for storage of data, backed by a HLF network. Instead of invoking smart contracts directly from the mobile device, the user sends a request to the REST API which then furthers a new request to the network through a so-called fabric-gateway-enabled peer. While the creation of a new user does not create a new peer in the network, a new user creates a new channel where the user's data is stored. Access to the application is managed through JSON web tokens [26] between the mobile application user, and using certificates generated by the HLF certificate authority. The

certificate used to communicate with the ledger depends on what user is authenticated to the server. Access to channels is, as in proposition 1, decided by the owner of the channel, which is the user associated with the certificate that was used to generate the channel. This approach does not remove the possibility to create smaller local "stations" that either can operate independently or interconnect with a larger network. The Assets stored on the ledger would also be of the same nature as in proposition 1 (Section 3.2.1).

3.2.3 Proposition 3

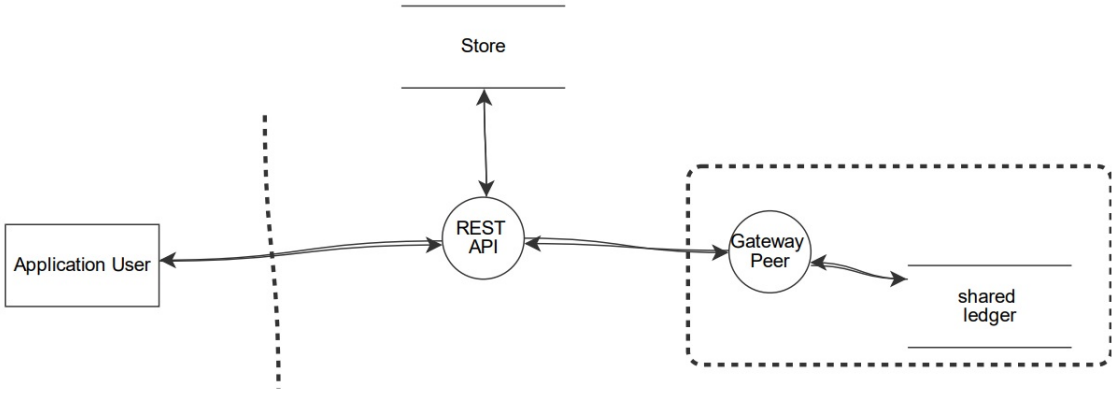


Figure 3.3: Data flow diagram for proposition 3

The third proposition uses a REST API together with an HLF network. The users again authenticate to the REST API using JSON web tokens, and access to the HLF network is done through an HLF gateway enabled peer using a single certificate authenticating the server. A single channel holds all users' posts, and access to a user profile is controlled through the API. Assets on the ledger are profiles, each representing a single user. A minimal profile asset includes the data fields "username", "followers", "followed users", "pending followers" and "posts". Username being, of course, the username, followers a list of verified followers, followed users a list of users that the user follows, pending followers a list of users that have requested to follow the user, and posts a list of the posts the user has either created or shared. The list of posts are data objects consisting of the fields "owner", "post id" and "sharing history". Sharing history is a list of users that have shared a post, and is used to verify whether another user has access to read the post. This proposition aims for simplicity, as something simple is often relatively simple to protect. How easy an application is to secure is often linked to how well the developers understand the framework.

3.2.4 Frontend Proposition

The proposition for the design of the UI is shown in Figure 3.4. The requirements for the frontend were to be able to navigate from one screen to another, style components like we wanted and make API calls to the backend server. More specific functionality included having a home screen with posts of followed users, being able to follow and accept follow requests, being able to forward posts in a private chat between users, and create posts and users. Other proposed functionality was being able to see your own posts, track where they ended up and control it by restricting certain users access. We proposed the use of the React Native [27] framework. This is because it has a good reputation after being used by Meta [28] in their biggest mobile applications. The code written works the same for both Android and IOS, making deployment to the two major platforms a lot easier. A part of this proposal is using Expo CLI [29] as React Native recommends it because of the simplicity and speed of the project setup. Another proposition was using Java [30] for the development. An advantage with Java is that its been around for a long time, which means that it has had long time to improve, and that the there is a larger quantity of help resources from other users that have experiences the same problems.

User Interface Design

Diagram 3.4 shows the proposed user interface design for the mobile application.

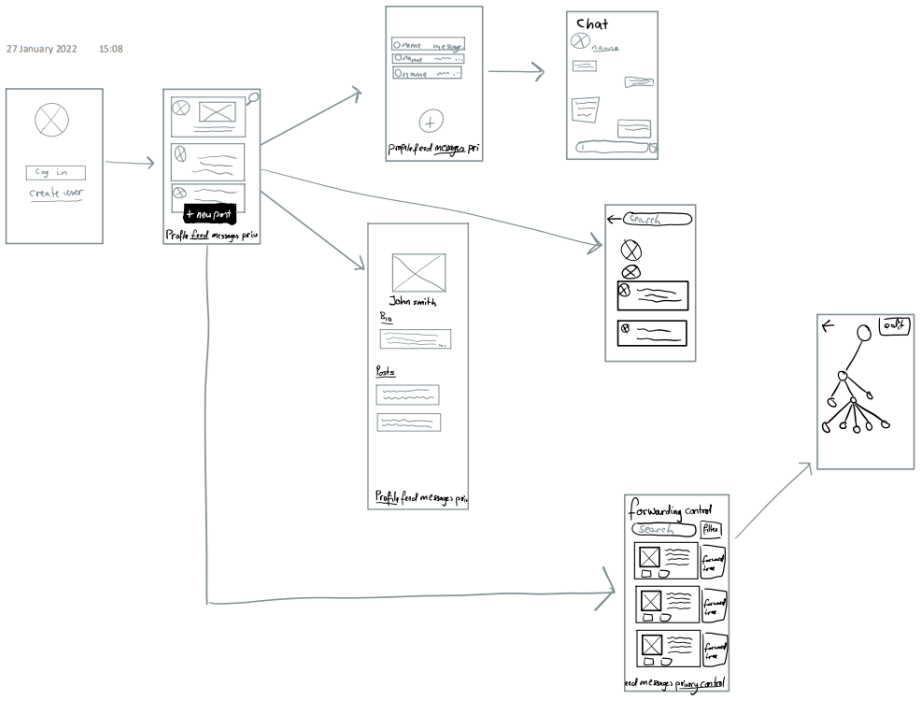


Figure 3.4: UI flow diagram

3.3 Frontend Development

Before we started making the UI of the mobile application, we downloaded Android Studio [31]. It was used to emulate an Android mobile phone on the computer (see Figure 3.5). The reason for the use of an Android mobile phone emulator is that this ensures that all developers can use the same type of device, and makes up for the lack of a physical Android mobile device. It was also more efficient and structured to have the complete development environment on the a single device.

Figure 3.5: This is the Android Studio Emulator

We then started setting up the development environment. One have to install node.js, but as this already had been done on the computer. The "node -v" command was ran in the terminal to see if the version of node.js that was installed was of version 14 or newer. Further in the environment setup, the installing Expo CLI globally was done using the "npm install –global expo-cli" command. The project was then initiated with the "expo init just-us" command. The last command made a folder with the project, which then was opened in the code editor Visual Studio Code [32].

3.3.1 The First Screens and Tab Navigation

Now it was time for creating the different screens of the application and the navigation between them. In the beginning the screens were created with only a title, and maybe a button for redirection to other screens. This was done by returning "<Text>" and

"<TouchableOpacity>" components wrapped inside "<View>" components, as shown in line 31 and onwards in Figure 3.10. We then startet making the tab navigator displayed in Figure 3.6.

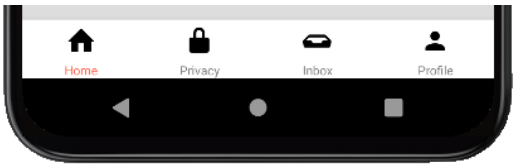


Figure 3.6: The navigation bar on the bottom of the screen

This navigator is a bar on the bottom of the screen with four different touchable icons which navigates the user to its respective screen. Having it at the bottom of the screen makes it easy to go from the home screen to the privacy screen, as well as to the inbox and profile page. This is because most of the users of a mobile application are using its thumb to touch buttons, and that is why the tab navigator is at the bottom. It was possible to use this because of the React Native library React Navigation[33]. It also contains other different solutions to navigation between screens, like the stack navigator and drawer navigator. The stack navigator was used on the four tab screens to further navigate to other screens, stacking upon the original screen which gives you an arrow in the top left corner to return to the previous screen.
The mobile application has support for instant messaging, however it wasn't implemented in the backend portion of the project. The inbox screen works as an example of a potential future implementation if this application was not only a proof-of-work.

3.3.2 Nests and Stack Navigation

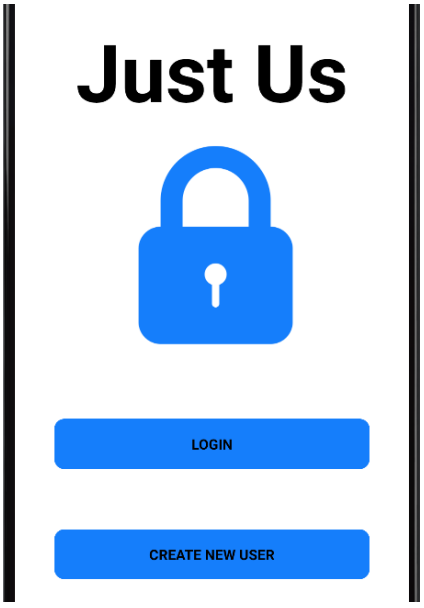


Figure 3.7: The first screen of the application

We then made the index screen, which is shown in Figure 3.7. This is the first screen the user sees when opening up the application. It is also where the navigation became challenging for the developers because of their lack of React Native experience. With the help of the React Navigation documentation [33], the solution was to make a nest. A nest is creating a navigation system within a navigation system. Most of the four tab screens got their own nest which had a stack navigation within the tab navigation, shown in the code of Figure 3.8.

```
export default function FeedNest() {
    return ( // This the nest for the Home screen
             // that enable navigation between these screen
        <Stack.Navigator>
            <Stack.Screen name="Feed" component={Feed}/>
            <Stack.Screen name="Search for people" component={VisitUserNest}
                          options={{headerShown: false}}/>
            <Stack.Screen name="Create a post" component={CreatePost}/>
        </Stack.Navigator>
    )
}
```

```
  <Tab.Screen name="Home" component={FeedNest}
              options={{headerShown: false}}/>
  <Tab.Screen name="Privacy" component={ControlNest}
              options={{headerShown: false}}/>
  <Tab.Screen name="Inbox" component={InboxNest}
              options={({route}) => ({headerShown: false})}/>
  <Tab.Screen name="Profile" component={Profile}
              options={{headerShown: false}} />
</Tab.Navigator>
```

Figure 3.8: The upper image is the stack navigator for the home screen with the post feed, and the lower code shows that stack as the first component of the tab navigator, together making a nest.

To make the navigation from the login screen to the main content of the application, a nest enfolding the four nests had to be made. This became a stack navigator within a tab navigator, within a stack navigator where you hide the return arrow when you navigate from the login screen to the main application. The login functionality was made by globally storing the logged in user with AsyncStorage [34], and setting it to an empty string when logged out. We later found out AsyncStorage is now deprecated, so for future development it would have to be replaced with a maintained solution to avoid the application breaking upon a new update. The screen displaying posts from other users is called the home screen, and can be seen in Figure 3.9. The home screen was made by displaying a list of a custom component. This component was made as a function that returns what to be displayed, similar to "CreatePost()" from Figure 3.10. But this function takes in the post as input and displays its content with the help of the "item"

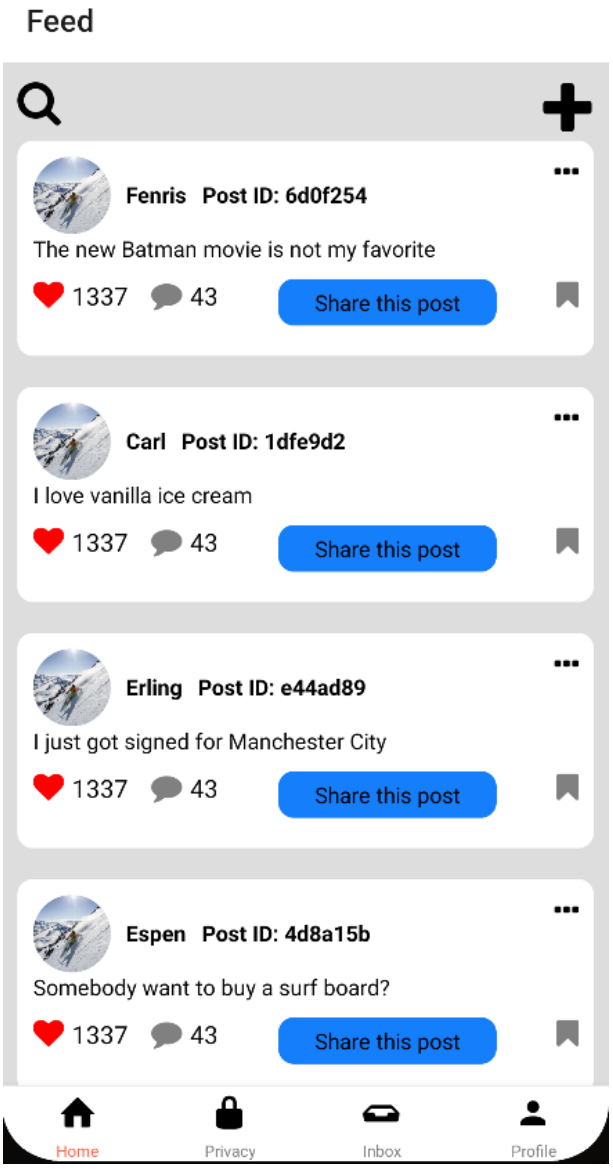


Figure 3.9: The home screen with the post feed. The like and comment count are static values.

parameter that is the post object, the same technique used in the code example of 3.15. The list was made with the React Native performant interface called Flatlist[35]. And with the help of the React hook (feature) Effect[36], it was possible to fetch the data belonging to the logged in user when entering the screen. Data in this case being the created posts of followed users.

3.3.3 Connecting with the API

At this point it was time for the frontend to start communicating with the API. The first instances were the creation of a new user and post. This was made with a POST-request with data sent to the backend server. The UI of this feature is displayed in

figure 3.11. Both the user and post creation was made with two main steps: store the input as a variable on the frontend, and then using React Native's fetch request to send the data to the endpoint of the backend. The creation of users send the fetch-request to "http://152.94.171.1:8080/User", where the creation of a post is sent to "http://152.94.171.1:8080/Post". The method is marked with "POST" so the API know how to query the data. In contrast, the home screen fetches a "GET"-request to "'...:8080/Posts?username='+user" on loading the screen, which returns the posts to be displayed, as shown Figure 3.9.
The way the local variable gets its value is through the a property of "<TextInput>" called "onChangeText". This sets the value of the variable every time anything in the input field is changed. It is used in the code screenshot 3.10. In this case the username variable is set to whatever is in the input field every time its changed, e.g., when a new character is added. The button under has the property "onPress", which calls a function of your choice when the button is pressed. In this case the function "Submit()" is called. This runs a try-catch statement which tries to send a fetch-request to the endpoint address, while catch handles the exception if it fails. The fetch-request includes the method as "POST", and a body containing the variables as data. This data is converted from a JavaScript-object to a JSON String using "JSON.Stringify()", making it appropriate to receive for the backend. This method of transporting data to and from the database, as used in the code of 3.10, is nearly the same for every API call made in the all the other screens. The only difference is the endpoint address and the body of the "POST" requests. One can also see that the "Submit()" function is asynchronous, enabling the fetch method to use "await". This prevents influence from other API calls happening at the same time.

3.3.4 Following Another User

The next objective in the development process was the following feature. For a user to be able to follow another user, there had to be implemented a way to search for a user and send them a follow request. This follow request would then have to be accepted by the receiving user. On the home screen with the post feed, a magnifying glass icon was placed in the top left corner, as shown in Figure 3.9. This had an "onPress" option that navigated the user to the search screen with the use of React Native's built-in navigation feature and a stack navigator. This search screen has a search bar that takes in a text input, and sends a POST-request with the search phrase to the "...:8080/Users/Search" endpoint which responds with a boolean value if the search phrase exists in the user database or not. If the user exists, it will appear a "send follow request" button which will do exactly that when pressed. If the search phrase does not match any created users,

```
14  export default function CreatePost() {
15
16      const [data, setData] = useState('');
17
18      async function Submit()
19      {
20          const user = await AsyncStorage.getItem('storageUsername')
21          try{
22              fetch('http://152.94.171.1:8080/Post', {
23                  method: 'POST',
24                  body: JSON.stringify({data: data["data"], owner: user})
25              });
26          }catch{
27              console.log("This did not go as planned")
28          }
29      };
30
31      return (
32          <View style={styles.container}>
33              <Text style={styles.header}>Create post</Text>      You, 2 months ago • Inbox, p
34
35              <TextInput style={styles.textinput} placeholder="Content of the post"
36               onChangeText={(text)=> setData({data: text})}
37               underlineColorAndroid={'transparent'} ></TextInput>
38
39              <TouchableOpacity style={styles.buttonContainer} onPress={()=>{Submit()}}>
40                  <Text style={styles.btntxt}>Post</Text>
41              </TouchableOpacity>
42          </View>
43      )
44  }
```

Figure 3.10: The main code for the "Create Post" functionality

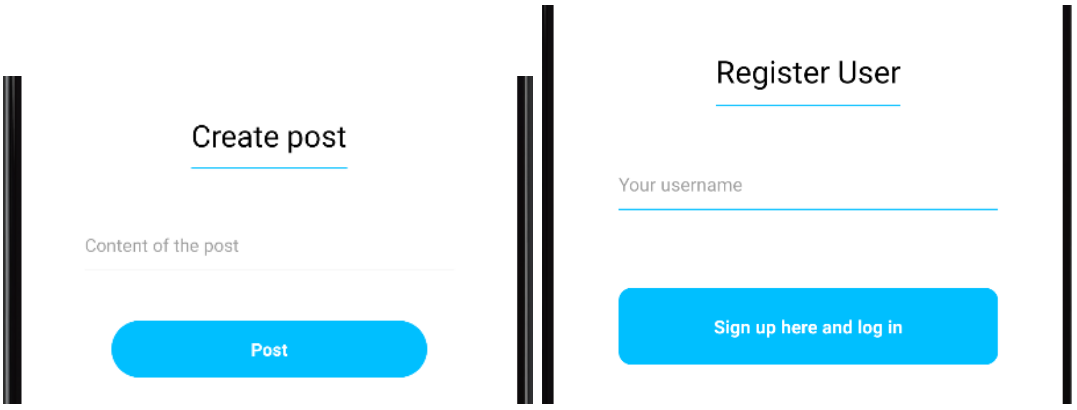


Figure 3.11: The UI of user and post creation using the same template

the user will be alerted of it. The "send follow request"-button have the ability to appear because of the technique of conditional rendering. This is done with the &&-operator which works like an if-statement, explained further in 3.15. In this case the follow button only appeared if the response from the "search for user"-request was true. Sending the follow request is quite similar, with just a POST-request to the "...:8080/User/Follow", and with the body consisting of the logged in user and the search phrase as the target user.

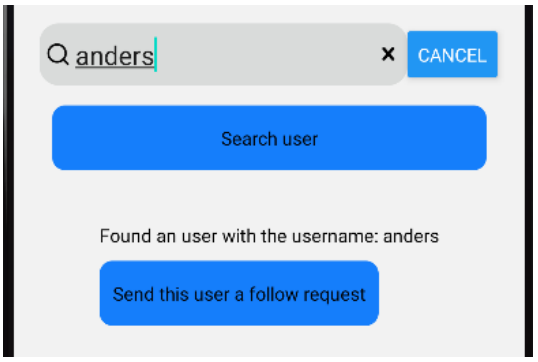


Figure 3.12: User search with follow functionality

3.3.5 Profile and Accepting Followers

We were now going to making it possible for a user to accept the follow request. This was done by adding a button on the profile screen that displayed the list of all users that have sent a follow request. This list of users is obtained by fetching data from the "'/Profile?username='+user" endpoint that returns all data associated with the user, including its posts, followers and follow requests. With the help of the React hook(feature) Effect[36], the data fetch happens as you enter the screen and the list of pending followers is saved in a state variable. The display of the pending followers has a pressable green check symbol that accepts the follow request. This is done by sending a POST-request to "...:8080/User/AcceptFollow" together with the user and the future follower as data in the body.

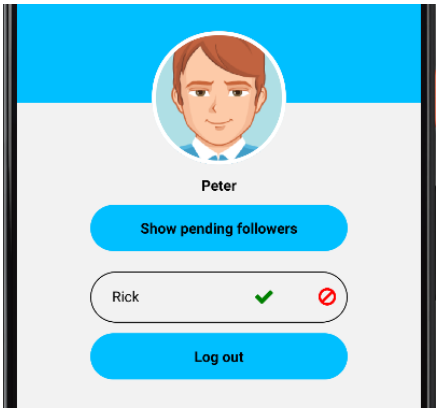


Figure 3.13: The profile tab with pending follower requests and logout

Further development included adding a button on all posts displayed on the home screen. This was a share button that would have the functionality to share the post further to the followers of the user sharing the post. In simpler terms, when a user shares a post, it posts it to its own followers, with the original creator and content. This is by sending the user, owner and post ID to "...:8080/Post/Share'". This potentially exposes the post to users not following the original creator, and giving them the opportunity to share the post even further.

3.3.6 Privacy

When there is a possibility to share posts, we thought the original creator deserves to know who shares the post and where it ends up. This is where the privacy screen, shown in Figure 3.14, comes in to play. This screen also uses Flatlist[35] to display the post ID of a post and the users that have shared the post.

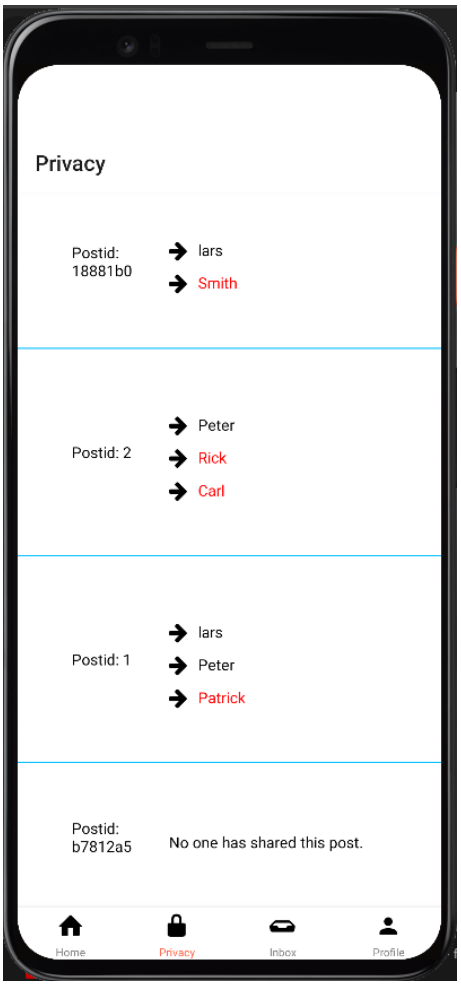


Figure 3.14: The privacy tab

This was done by first fetching all the posts of the logged in user, and sending the list of posts into another function called "makeShareList()". This function loops through the post ID of every post in the list and sends a "GET"-request that return the transaction history of the post, which includes every user that has shared that post. This data is pushed onto another list, as an object containing the sharers of the post and the post ID. This list is used when rendering the item that is displayed in the Flatlist [35] on the privacy screen. It includes conditional rendering, as shown in 3.15. It is rendering different components depending on whether or not there exists any sharers of the post. The last step is distinguishing the sharers of the post that follow the original creator and

not. This is done by fetching the follower list of the user, and checking if the sharer is included in the follower list. The sharers are then conditional rendered with different colors. The followers of the original creator are displaying with normal black font color, and the sharers that do not follow the original creator will be displayed with red font color. This is done by giving the latter's "<Text>" tag its own style component.

```
<FlatList
data={userSharedList}
keyExtractor={item => item.postId}
renderItem={({item}) => ( // This is where the rendered component is defined.      You, 1 second ago • Uncommitted
    <View style={styles.container}>
        <View style={styles.postIdField} numberOfLines={1}>
            <Text>Postid: {item.postId.substring(0,7)}</Text>
        </View>
        {item.sharers.length > 0 &&    // The content after && is displayed if the condition is true.
            <View style={styles.userCard}>
                {item.sharers.map((username) => { // Map is looping through the sharers list and retrieves the user
                    return(
                        <View style={styles.arrowUser}>
                            <Icon name="arrow-right" size={20} style={styles.arrow}/>
                            {!followerList.includes(username) ? // A new if-statement with a condition.
                            //  This is where users that are in the share list and not in the followerlist
                            //  gets red font and the follower gets normal black font.
                            <View>
                                <Text style={styles.imposter}>{username}</Text>
                            </View>
                            : // This is the else-statement. Because of this, the ?-operator is used instead of &&.
                            <View>
                                <Text>{username}</Text>
                            </View>}
                        </View>
                    )
                })}
            </View>
        }
        {item.sharers.length < 1 && // Another conditional rendering that check if the post has any sharers.
        <View style={styles.arrowUser}>
            <Text>No one has shared this post.</Text>
        </View>}
    </View>
)}></FlatList>
```

Figure 3.15: The Flatlist component that is rendered in the privacy screen, along with descriptive comments in the code

3.4 Chosen Solution

3.4.1 Frontend

We chose to use React Native as the frontend framework rather than Java for a multitude of reasons. One of them was because it seemed simple to create the project with the help of the Expo CLI [29]. We also found it useful to make the code compatible with both an Android, iPhone and multiple other operating systems for the mobile device. We chose

to partially follow the UI design diagram of Figure 3.4. This is because the backend had more limitations than expected. For that reason, there is implemented inbox and chatting on the frontend, but lacks backend functionality. That means the inbox tab of the application will only contain static components. We also chose to not prioritize being able to restrict access for users that have forwarded posts. This is because the chosen functionality of being able to see and differentiate between followers or not followers in the sharers of the post is sufficient to show the concept.

3.4.2 Backend

The beginning of the project was characterized by indecision. Development started on proposition 1 (Section 3.2.1), but after several months of scouring the internet for reference material and very low code output the group decided to go with proposition 2 (Section 3.2.2), then proposition 3 (Section 3.2.3) when progress on proposition 2 stagnated as well. To ensure that we had a project that we could present within the time limit, we opted for the third and most simple solution.

Backend Components

The chosen solution consists of three main components: The chaincode, the chaincode tests, and the API, according to proposition 3. All components are written in Go for consistency.

Chaincode

The chaincode uses the smart contract API created by hyperledger fabric to facilitate creation of smart contracts[37]. Code snippet 3.1 is representative for the functions in the chaincode as all the functions that make up the chaincode follow similar patterns.

```
func (s *SmartContract) GenerateSharingTree(ctx contractapi.TransactionContextInterface,
userId string, postId string) ([]Post, error) {
        iterableHistory, err := ctx.GetStub().GetHistoryForKey(userId)
        if err != nil {
                return make([]Post, 0), err
        }
        var profileStates []Profile

        for iterableHistory.HasNext() {
                queryResult, err := iterableHistory.Next()
                if err != nil {
                        return make([]Post, 0), err
                }
                var profile Profile
```

```
                err = json.Unmarshal(queryResult.Value, &profile)
                if err != nil {
                        return make([]Post, 0), err
                }
                profileStates = append(profileStates, profile)
        }
        var postHistory []Post
        for _, val := range profileStates {
                for _, value := range val.Posts {
                        if value.PostId == postId {
                                postHistory = append(postHistory, value)
                        }
                }
        }
        return postHistory, nil
}
```

Listing 3.1: chaincode function for getting complete history for an asset

All chaincode functions are methods on the struct SmartContract defined earlier in the script, as well as taking in the transaction context. Taking in the context like this makes the function more testable. Whenever functions in the chaincode need to access the ledger, they use the transaction context to pass the function getStub() which gives access to functions that operate on the ledger as well as the world state, in this case the function creates a query for the complete history of the profile with the corresponding userId. All queries are simple key-value queries.

Chaincode Tests

Tests for chaincode was a necessity for speedy development. The virtual machine used more than 2 minutes to rebuild the network and install new chaincode on the peers in the network, which made it near impossible to weed out small bugs without having the tests.

The tests use mocks generated by counterfeiter to be able to pass a fake transaction context to the chaincode functions. As per usual, the tests aim to test for edge cases. As an example is the test for the follow function in code snippet 3.2.

```
func TestFollowProfile(t *testing.T) {
        chaincodeStub := &mocks.ChaincodeStub{}
        transactionContext := &mocks.TransactionContext{}
        transactionContext.GetStubReturns(chaincodeStub)
        userProfileJson := createProfile()
        smartContract := chaincode.SmartContract{}

        chaincodeStub.GetStateReturns(userProfileJson, nil)
        err := smartContract.FollowProfile(transactionContext, "anders", "jorban")
        require.NoError(t, err)
```

```
        err = smartContract.FollowProfile(transactionContext, "lars", "lars")
        require.EqualError(t, err, "users cannot follow themselves")

        err = smartContract.FollowProfile(transactionContext, "anders", "lars")
        require.EqualError(t, err, "user anders already following user lars")
}
```

Listing 3.2: Follow profile test

API

The final component of the backend is the API. The API creates the interface between the mobile application and the fabric network. The connection is made with a gateway enabled peer. To authenticate the API to the network, a certificate is created from one of the pre-made identities in the HLF test network. This identity is stored in a wallet folder in the same folder level as the server. The server uses the gorilla/mux [38] library to enable the passing of parameters in the url.

When contact has been made between the gateway enabled peer and the api, the gateway can be used to fetch an instance of the chaincode running on the network. from that instance, the API can call chaincode functions. An example call is shown in code snippet 3.3

```
func followUser(w http.ResponseWriter, r *http.Request) {
        log.Println("--> Endpoint Hit: followUser")
        body, _ := ioutil.ReadAll(r.Body)

        var followRequest UserQuery
        json.Unmarshal(body, &followRequest)
        log.Printf("sending follow request from %v to %v",
        followRequest.UserId, followRequest.QueryId)
        _, err := contract.SubmitTransaction("FollowProfile",
        followRequest.UserId, followRequest.QueryId)
        if err != nil {
                log.Printf("Failed to submit transaction: %v", err)
        }

}
```

Listing 3.3: Share post in API

The call to the chaincode function FollowProfile is made through the chaincode instance. parameters must be passed as strings. The function followUser is called when the endpoint /User/Follow is hit.

3.4.3 Running Environment

The chaincode and API development was done on a remote virtual machine running linux. This was because HLF plays nicer with linux systems than other environments. In addition to this, using the VM provided us with a static IP address reachable from within the UiS network, thus we could easily host the website for the mobile application.

3.4.4 Frameworks/Tools Used

For development of chaincode and API we used the Visual Studio Code [32] programming environment together with its SSH extension. The API was written using the gorilla/mux[39] package to be able to fetch parameters from the URL. This is mostly a feature for quick prototyping, as passing parameters in the URL is generally considered unsafe. To achieve off chain data storage we adapted a project created by Mohammad Hanif Tadjik[40]. We also make use of Tadjik's setup script (start.sh) in the folder, albeit with some small changes. The database used for off chain storage was mysql.

3.4.5 Diagrams

Sequence Diagrams

To gain an Idea of what the application calls would look like, we created sequence diagrams for the most important operations, namely "create post" in figure 3.17, "share post" in figure 3.18, "view sharing history" in figure 3.19 and "change sharing permissions" in figure 3.20. These were made together with a state diagram, shown in figure 3.16, for proposition 1.

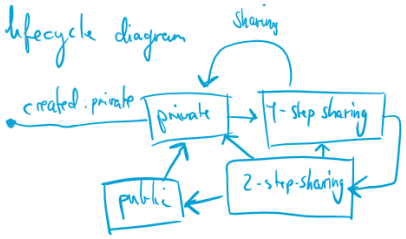


Figure 3.16: Lifecycle of a post asset

These operations alone are not enough to constitute a social network. What is missing from these operations are functions to connect users, without which you cannot create a social network. Connections between users would be made through configuration changes in the user's channel, such that for one user to follow another user the latter would create

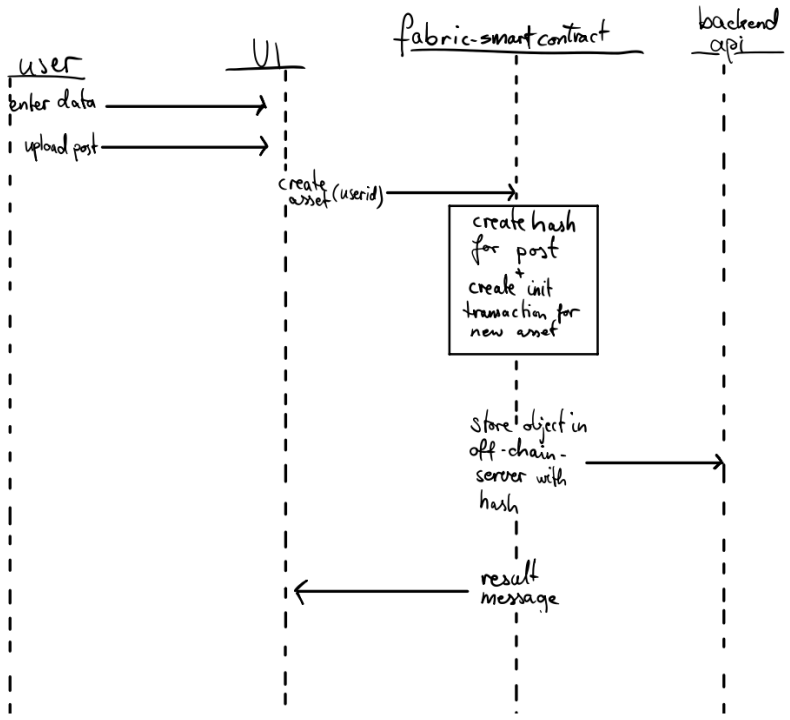


Figure 3.17: Sequence diagram for create post call

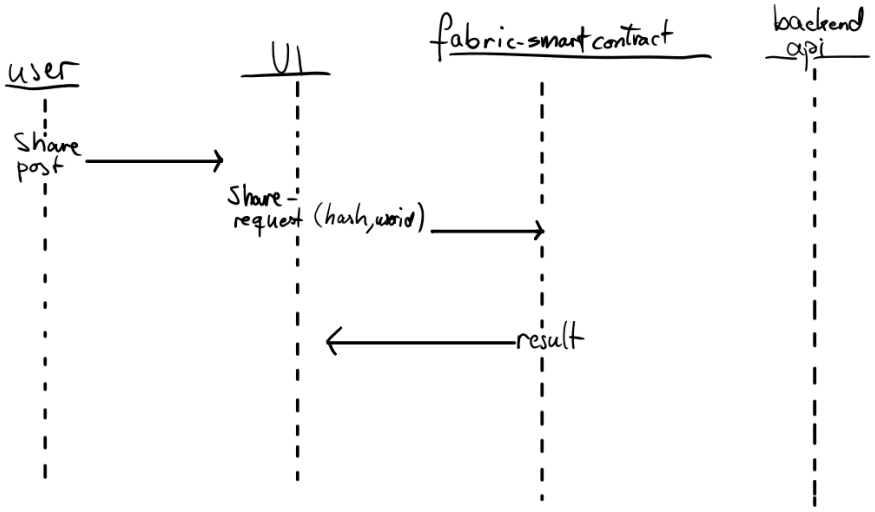


Figure 3.18: Sequence diagram for share post call

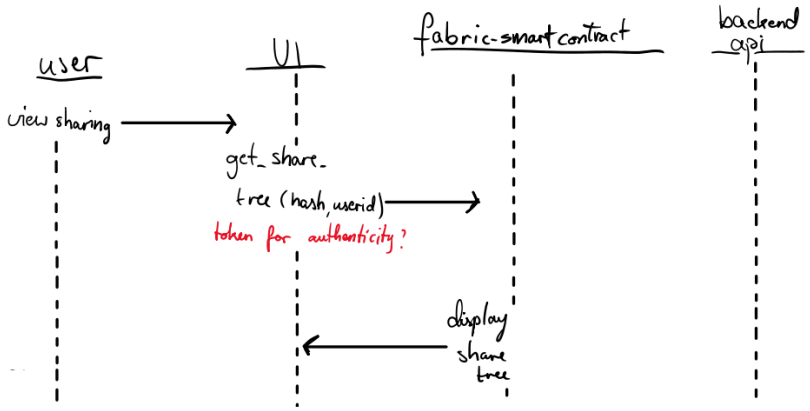


Figure 3.19: Sequence diagram for viewing the sharing sequence

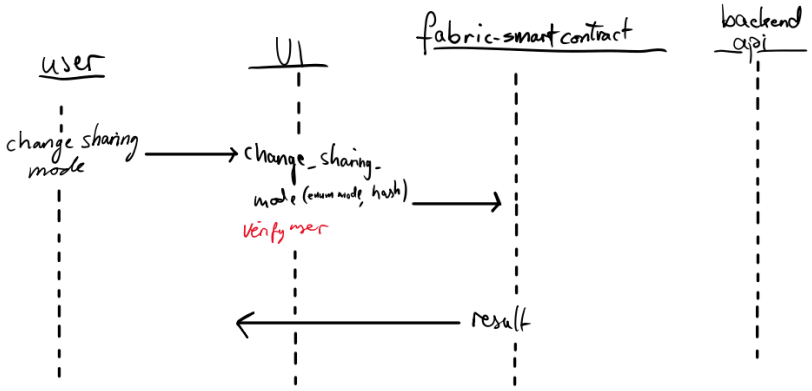


Figure 3.20: Sequence diagram for changing the options for sharing permissions

a configuration change in their own channel such that the latter is given read access to the channel. Admittedly, this is incomplete and handling adding a user directly in the API is a necessity for a well-functioning application.

3.5 Development Plan

We created a rough plan with major milestones for each month, and by the end of each week we would have a meeting and decide on tasks for the following week.

3.5.1 January

We decided to use January to begin learning the frameworks we were using, as well as creating this rough plan.

3.5.2 February

In February we aimed to finalize the design for the user interface of the mobile application. We estimated that we would be ready to write the backend using HLF by the end of the month, so February would be used for a lot of reading of documentation, and investigating other HLF projects. To reduce the amount of work we would have to do later on, we aimed to have 20 pages written in the thesis by the end of the month.

3.5.3 March

By the end of March, the goal was to have written 40 pages in the thesis. Though we decided that the final report should be around 40 pages, we didn't aim to finish the first draft in March. In a similar vein, we decided that app development (both frontend and backend) should be well underway by the end of March, but we didn't describe very accurately the deadline for an alpha build.

3.5.4 April

By the end of April, the app should be completed. We also aimed to finish the thesis, in other words finish both a first draft and final draft.

3.5.5 May

May we decided to use as a buffer month. We allocated the first seven days for eventual cleanup of code and other adjustments to the thesis, and set the deadline for the finished project at the seventh of May. The official deadline for the Thesis is the 15. of May.

Chapter 4

Solution Evaluation

The application in its current state serves as a test of the usage of blockchain in a social network, specifically to improve control over personal data. The application that we present at the time of writing does not have all the features that were described in the project outline, and also lacks features that improve functionality such as better handling of slow execution in both the mobile application and the server. Most of the functionality from the application is obtainable with a normal centralized approach, save for the transaction history, but this, in our opinion, is the most important part.

4.1 Execution Time

Chaincode execution in HLF is very fast, relative to public blockchains like Bitcoin, but still proved to be sluggish in our application. The likely cause for this is the rate at which the orderer was configured to assemble and verify blocks. Normally, the orderer creates blocks for a certain amount of transactions, but it is also configured to create blocks on a timeout. Therefore the theory is for the moment that the system created more blocks relative to the amount of transactions due to the low throughput, and this slowed the system down. If the application was written as a regular REST API without blockchain, a call to create a post would simply take the data from the request, package it into a new post data object and store it in the database. When using Fabric, the data itself is sent to the database, but the post object is added to the blockchain, and for this operation to succeed it must be packaged within a transaction on the ledger, which must be verified according to the policy in the channel.

4.2 Data Sharing

Our application demonstrates the concept of tracking data based on querying the transaction history. To obtain the sharing path of the data object, the application gets the history of the post in question. Though the application does not use the complete history in its current iteration, there is the potential to automatically audit this history of events to see if there is something that does not match what the history of events should be. Using the transaction history is more valuable than simply checking a state variable, as a state variable does not provide a history of all their past states and thereby if access to the account is compromised it is more difficult to discover. The current implementation shows the user which of their followers who shared their post, and which users that do not follow the user who also shared the post further.

4.3 Query Efficiency

HLF supports two different types of state databases for storing assets (in this case the profile objects): LevelDB and CouchDB [41]. While levelDB is faster, couchDB supports more advanced queries if the assets are stored in JSON format. The application only uses key-value queries for the application, which potentially could be improved using more specific JSON queries. While for the current configuration LevelDB seems to be more appropriate, as the chaincode only addresses the ledger using key-value queries, due to the complexity of the data object, more complex queries could speed up certain transactions that take a lot of time due to having to filter out the fields of interest instead of receiving them as raw data.

4.4 Network Design

For network design the project largely relied on the test network provided by the HLF development team. This is not a problem for testing of smart contracts and developing a basic proof-of-concept. However, for experimenting with different policies and different network configurations and topologies, it is necessary to design a bespoke network. In the documentation for the test network, the HLF developers state that the test network is prone to breakage upon changes. This is most likely due to the accompanying script for deploying the network which automatically sets up all facets of the network, from creating a certificate authority, to setting up organizations, to creating channels for those organizations by hard-coding.

4.5 Data Uploads

For the moment, data uploads are limited to text uploads. This does not need to be the case, as the data is not stored on the blockchain, only in the off-chain database. Therefore to enable the uploading of photos and/or videos, only small changes in the database as well as in the API to handle sending and receiving videos would need to be made.

4.6 Decentralization

In its current state the application does not support decentralization. It would be possible to deploy the social network chaincode to another channel, Users from one channel to the other would not be discoverable to each other. This is an inherent flaw with the design of the application, and could be solved by adopting one of the other propositions.

4.7 Security

In the original plan, JSON web tokens were used to authenticate users and verify origins of the request. In the final product, implementing basic features took precedence. Implementing JWT [26] for security is not a very big undertaking but improves security dramatically. If we had changed approach earlier, implementing JWT would be highly prioritized, as it significantly closes the distance between this application being a proof of concept to a beta version.

4.8 Scalability

The solution approach likely does not scale well. While improving the design of the data objects could result in a sharp decrease in the amount of queries needed to find a result and likely speed up the existing queries, in the end the design is a more centralized approach and is largely limited by the hardware supporting it. Eventually with a larger user base, the application would need the kind of processing power that is available to other social media.

4.9 Encryption

Data is uploaded as-is without encryption to the database. This is not secure, as if someone gains access to the credentials for database connection, they may compromise the privacy of the users. Enabling the encryption of posts is decidedly an important step for improving the security of the application. Posts could be encrypted using the keys distributed by the fabric CA. For the moment, accessing the posts of others is prevented through creating a unique hash for unique posts, and the hash is stored together with the post object in the ledger. However this hashing is also flawed, as the hash is based upon the data which means two identical text posts receives the same hash.

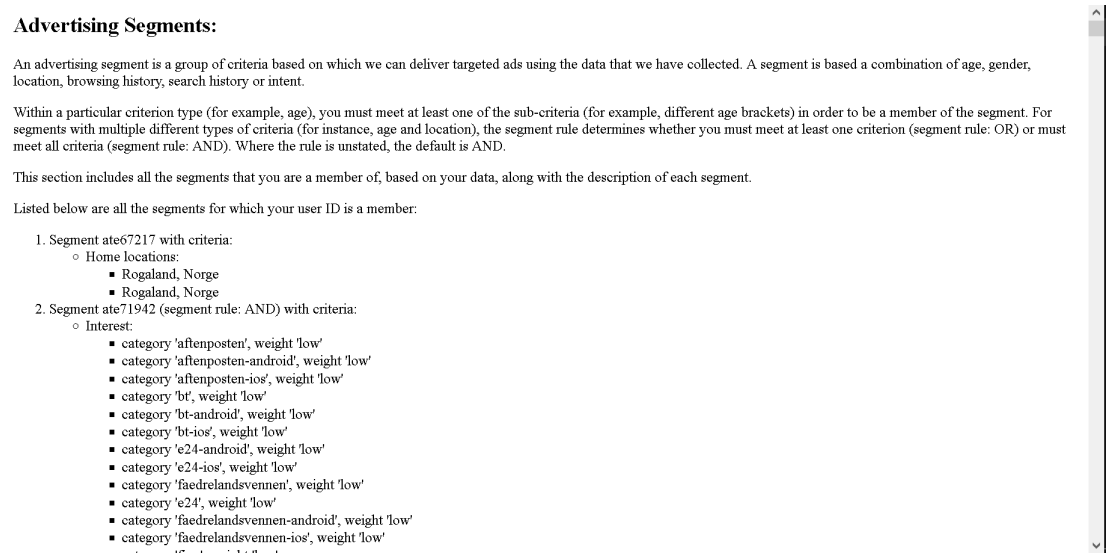
Chapter 5

Discussion

5.1 A Case of Finding a Nail For a Hammer?

At the moment, blockchain technology is very much in the wind. New non-fungible token frameworks and crypto-currencies are being promoted by influencers of all types on many different platforms without ever presenting any novel ideas, while promising a huge payoff for those brave enough to invest. Because of the popularity of blockchain technologies like Bitcoin, the rate at which "blockchain" is suggested as a solution when either not applicable or when another solution may work better is most likely too high. This quandry has spawned articles like "Five Things You Should Not Use Blockchain For" [42]. However, in the case of this project there is clearly potential. Many of the problems associated with social media are related to incorrect or unlawful treatment of personal data. With an HLF based system, it could all but eliminate the human component in the treatment of data, in the way that the only administrator of the data uploaded to the network is the creator, save for administrative rights to, e.g., remove channels and/or users that violate the terms of service for the application, as well as opening up for a decentralized approach where users do not depend on a single centralized service provider. It also opens the door for actually owning your own data, giving you the opportunity to host your own data and thereby have complete control over your data.

There is a lot of money being made in advertising using personal data that creates a frighteningly accurate profile of the end user to show advertisements that the end user is more likely to click. Figure 5.1 shows an example of an ad profile of one of the authors of the thesis downloaded from Schibsted. Note the size of the scrollbar, it is a large document.



Advertising Segments:

An advertising segment is a group of criteria based on which we can deliver targeted ads using the data that we have collected. A segment is based a combination of age, gender, location, browsing history, search history or intent.

Within a particular criterion type (for example, age), you must meet at least one of the sub-criteria (for example, different age brackets) in order to be a member of the segment. For segments with multiple different types of criteria (for instance, age and location), the segment rule determines whether you must meet at least one criterion (segment rule: OR) or must meet all criteria (segment rule: AND). Where the rule is unstated, the default is AND.

This section includes all the segments that you are a member of, based on your data, along with the description of each segment.

Listed below are all the segments for which your user ID is a member:

1. Segment ate67217 with criteria:
   - Home locations:
     - Rogaland, Norge
     - Rogaland, Norge
2. Segment ate71942 (segment rule: AND) with criteria:
   - Interest:
     - category 'aftenposten', weight 'low'
     - category 'aftenposten-android', weight 'low'
     - category 'aftenposten-ios', weight 'low'
     - category 'bt', weight 'low'
     - category 'bt-android', weight 'low'
     - category 'bt-ios', weight 'low'
     - category 'e24-android', weight 'low'
     - category 'e24-ios', weight 'low'
     - category 'faedrelandsvennen', weight 'low'
     - category 'e24', weight 'low'
     - category 'faedrelandsvennen-android', weight 'low'
     - category 'faedrelandsvennen-ios', weight 'low'

Figure 5.1: Example of an ad profile

5.2 Data Tracking

The function that shows tracking information for a post is very slow, as well as not persisted, so whenever you want the sharing history for a post it has to be generated from the transaction history. A way to mitigate this would be to persist a graph data structure within the post object that maintains lists of nodes and edges, and is updated upon sharing operations. For security, this data object could be regularly audited using the transaction history for both the profile and the post in question. This would catch any malicious attempts at accessing content.

5.3 Limitations

5.3.1 Learning Sources

Even if we wanted to start programming from the beginning, we could not. We were unfamiliar with writing smart contracts, even less with designing and configuring a blockchain network. Additionally, the lack of knowledge within hyperledger fabric hampered our ability to make a pragmatic choice of which proposition to invest time in.

There are limited resources to learn from apart from the official documentation. Projects in HLF are often for inter-enterprise solutions, which in turn limits the amount of available source code. While this prevents cherry-picking of code, which is generally seen to be bad, development slows down when there are no other relevant examples that apply concepts in a similar manner to what you aim to do.

The documentation may have been a cause for confusion, as there was very little difference between the documentation for different versions, and selecting the newest documentation wasn't done automatically for some reason. Different versions contained different best practices, and this may not have been noticed due to accessing the documentation through a search engine for the different modules/functionalities.

5.3.2 Division of Labour

The strategy of the development process was to divide the work between the two developers. This work was divided with one doing the frontend, and the other working on the backend. The intention with this strategy was to maximize efficiency by not learning the same knowledge and prevent it from overlapping. This became problematic because it prevented the opportunity to help each other. The consequences this mistake brought with it was the frustration of working alone without the input of another human being. The psychological improvement of just talking about a problem and not suppressing it inside was something we missed during this development process.

There were instances where the development process got stuck with the lack of a solution to a problem. If this problem paralyzes the entire development, it would have been great to use both brains to look for a solution. This inconvenience could have been avoided through better collaboration between areas of responsibility. The overlapping knowledge makes it possible to do some pair programming, which could bear fruit in the long run due to less time spent stuck with a problem. At the same time the motivation of the developers would maybe increase due to the lack of loneliness.

5.3.3 Initial Progress

As mentioned in Chapter 3, the development changed directions midway in the timeline. The initial plan was to implement proposition 1, but it proved to be challenging. We used the asset-transfer-basic example from HLF[43] as reference for the basic chaincode. This proved to be an unwise choice. Asset-transfer-basic as well as most of the other projects in the fabric samples repository use the CLI directly with the peers' docker images, with no examples on how to invoke smart contracts from outside the peers themselves. After some time we gained access to Tadjik's project [40] which helped a lot. It is a simple implementation of a smart contract which stores data in both an HLF network as well as in an external database, thereby solving the problem of interfacing with the HLF network. Therefore we attempted to make use of an example application created by IBM for a conference, fitcoin. The application seemed to fit what we were doing, and we attempted to adapt their strategy for generating certificates, but we encountered

an error with the config provided when attempting to generate new certificates. While we didn't find the solution to this, we might have encountered errors because we were attempting to create certificates without having the right permissions in the channel, and also because we attempted to use deprecated calls to the Go SDK together with technology that was introduced in the latest long term release, the application gateway.

When we saw that their approach wasn't working for us, we changed strategy to proposition two, to attempt to create and store certificates on the server and authenticate users with JWT to the API. While we could generate certificates for users described in the configuration file as well as connect to the network with them, we didn't figure out how to generate new certificates directly from the API, i.e. without using the fabric-ca CLI. At this point we were nearing April, and according to the plan the application development should have been "well underway", see A.3 in Appendix A. While the mobile application had seen progress, the backend had been practically standing still for the duration of the project. Therefore the decision was made to scrap proposition two, and attempt to implement proposition three. This was done because it didn't depend on certificates generated by the HLF CA, which is what had slowed development to a standstill.

5.3.4 Testing

The project ended up not following the testing regime described in Chapter 3. This was owing to the fact that the project significantly changed directions over the course of too much time. The testing strategy that was proposed works very well for an iterative development approach, such that whenever you add another layer of functionality you create a new integration test for this feature, as well as running the tests that you have created for the other features to make sure the new feature does not break any preexisting functionality, instead of having to weed through code haphazardly without any idea of where the eventual error is stemming from. The only tests written for the application was the chaincode tests, and they were unit tests instead of integration tests. This made coverage low, but the tests written saved a lot of time. Instead of having to build the entire network again upon every change (which took almost three minutes), we could run unit tests for the function in question until the code passed the tests, then build the network with the new code and test with the API and the mobile network.

5.4 Further Development Proposition

HLF has support for decentralization due to the Raft protocol backing the ordering service. Different organizations may have orderer nodes in the same channel, meaning

that no one organization would have to depend on a single centralized host for transaction processing and consensus, and would also be able to handle one of the other organizations' hosts crashing. However, as mentioned in Chapter 2 the Raft ordering service is not a byzantine fault tolerant system, meaning malicious orderer nodes may affect the transaction ordering. Using a modified version of the second design proposition that was introduced in Chapter 3 would mitigate this through giving each user a single orderer for their own channel, as well as open for a decentralized application.

The modification would center around the process for joining the network. Whenever a new user joins the network, they will be presented with the choice between downloading the network application and API for usage on a personal server (for advanced users), and a choice between service providers who can host your API and corresponding HLF gateway peer. The data is stored in the same location as the rest of the components that constitute the new user's membership in the network, which isolates it from unauthorized access as well as giving the user complete control over the data. The cost of participating in the network would therefore be the eventual hosting fee you would have to pay to the external service provider. And this way, the cost of participation in the network is completely transparent in the way that you pay for the resources needed to host the application, and not with your personal data.

An important factor when creating any software product is the value proposition: is there any money to be made off the product? While serving ads is possible on the network, non-tailored ads are not as attractive to buyers of ad space compared to tailored ad spaces, and buyers are thus not willing to pay as much for the advertisement space as they would otherwise. Therefore ad revenue would likely be minimal. For income, the developers of the network could offer a Software-as-a-service package where users can pay the network for hosting the peer associated with their account. Payment could vary depending on the resources needed for the user that is signing up; a high profile celebrity signing up will likely require more bandwidth and performance than a regular citizen.

5.5 What Blockchain Does Not Solve

To avoid selling blockchain as the be-all and end-all when it comes to the treatment of personal data, there are some assumptions that must be addressed and avoided. Yes, sharing may be better controlled by the owner and yes, malicious access may be less of a problem, but using blockchain doesn't automatically mean that you own your data. **Just Us** does not collect and store user data for analysis in its current iteration and neither will it if development continues, but the other options for social media backed by blockchain technology do not necessarily adhere to the same principles. In practice, the

usage of blockchain should be viewed with the same scepticism that something labeled as "water-resistant" gets versus something labeled "water-proof"; while blockchain may help with controlling user data, it is certainly not enough on its own. It must be combined with other technologies or principles so that everyone who uses the app can be certain that the application only consumes the data that it is intended to consume. One such design principle is open source. In the early stages of Elon Musk's proposed acquisition of Twitter, Musk claims that he will open source the algorithms used in Twitter for increased trust [44], thereby enabling individuals that hold the applicable competence to inspect how user data is treated by the network. Note that Musk never mentions anything about not collecting user data.

Chapter 6

Conclusion

Personal data is valuable. It is easy to claim that not opting out of having your private information recorded does not affect you because you cannot feel the physical effects. However, if data collection to this degree is allowed to run as wild as it currently is, the large companies stand free to use the information to further their own agenda. One needs to look no further than the Cambridge Analytica scandal to see the effects of this. All humans have an inherent right to privacy, and the amount of data that is collected is tantamount to constant surveillance. All this data in the wrong hands can result in the loss of personal freedom.

While the finished application does not fully exploit the capabilities of Hyperledger Fabric, it shows that it is possible to create and maintain a social network using Fabric. Personal data is protected from users that should not have it, and all operations on data is recorded in the ledger, so you can audit how your data has been used by others.

Despite spending the thesis making a case for using blockchain in social media applications, there must be room for the consideration that blockchain might not be the best solution to the problem. While knowing where posts are stored, as well as being able to track where posts are traveling through the network can be of interest to the owner of the posts, the solution might be overkill for a simple social network, and instead be more applicable in a business context. Most people using social media aren't too concerned with who reads their posts, hence the massive user base of Twitter and Facebook. In a business context, the data you share with other members might be a lot more sensitive, and users as well as the companies they act on behalf of may have a stronger interest in maintaining a log of how the data shared between companies is used. The biggest problem with social media today is not being unable to track who shares the posts you share publicly with friends, but rather how the companies that provide the social network processes all the other data that you indirectly give away through actions such

as scrolling, liking and sharing posts. This is a problem that implementing blockchain alone does not solve. Though some claim it could be solved with simply open-sourcing the code base associated with the social media, simply showing what information you are collecting is not the same as not collecting it.

Appendix A

Installation instructions and code repository

A.1 Github Repository

The complete project repository can be found at:
https://github.com/lars-and-anders-bachelor-thesis/just-us.

A.1.1 Frontend build instructions

The frontend build instructions are as follows:

```
    Running the frontend of Just Us
Requirements

One of the following versions of Node.js must be installed to run npm:

    14.x.x >= 14.15.0
    16.0.0 or higher

You can use the following command to check your version:

node -v

You need a mobile device to run the application on.
Android Studio's Emulator is the Android Emulator that
```

```
has been used to develop and test this application.
Cloning the repository

To clone the repository you have to run the following
command where you want your cloned repository:

git clone --branch frontend https://github.com/lars-and-anders-bachelor-thesis/just-us

Installation

This application is based on Expo CLI, which needs
to be installed with the following command:

npm install -g expo-cli

Connection with the API

To make the application send the API calls to the
right endpoint address, you have to change the value
of the global address variable. This file is found
in \frontend\assets\globalVariable.js. This file has
one line, with one variable that have to be changed to fit your setup.

If you run it on ubuntu, you just have to change it
to "localhost". If you run the server on wsl you
have to find the ip address associated with the wsl
2 distribution with the following command:

wsl -- ip -o -4 -json addr list eth0 ‘
| ConvertFrom-Json ‘
| %{ $_.addr_info.local } ‘
| ?{ $_ }

The backend server needs to be running for the
API calls to work. The setup for the backend
server is found here.

Running the application
```

```
If you are using the android emulator, start android
studio, and press the "start" icon next to the phone
description (e.g. pixel 3a).

In the just-us folder, use the following commands
to start the local server of the application:

npm install
npm start
```

A.1.2 Backend Build Instructions

The backend build instructions are as follows:

```
    Just Us: A Blockchain-based Privacy-friendly Social Network
Backend Build Instructions

For the frontend build instructions, go here. (note: only tested on WSL 2
ubuntu and ubuntu 18.02LTS,
need linux environment to run)
Prerequisites:

Mysql, Docker and Docker-Compose needs to be installed on your computer.
Step-by-step build

    Clone the Just Us repository
    Clone fabric-samples repository
    and binaries using the following command from within the just-us/backend folder:

curl -sSL https://bit.ly/2ysbOFE | bash -s

In WSL 2, make sure Docker Desktop is running as
well as the WSL 2 integration being turned on in Docker Desktop.

    Move the my-simple-offchain folder inside the fabric-samples folder.

    Run the following command inside the backend folder to initialize database
    with the required user and tables:
```

```
sudo ./databaseSetup.sh

If you get a Mysql error, run the following command and try again:

sudo service mysql start

    Navigate to the my-simple-offchain folder and run the following command:

start.sh

    Follow server build instructions from output of start.sh
```

A.2 Demo Video

A link to a demo video of the application running on a local machine with an Android emulator can be found here: https://youtu.be/PMpSpNsFWq0

A.3 Development Plan

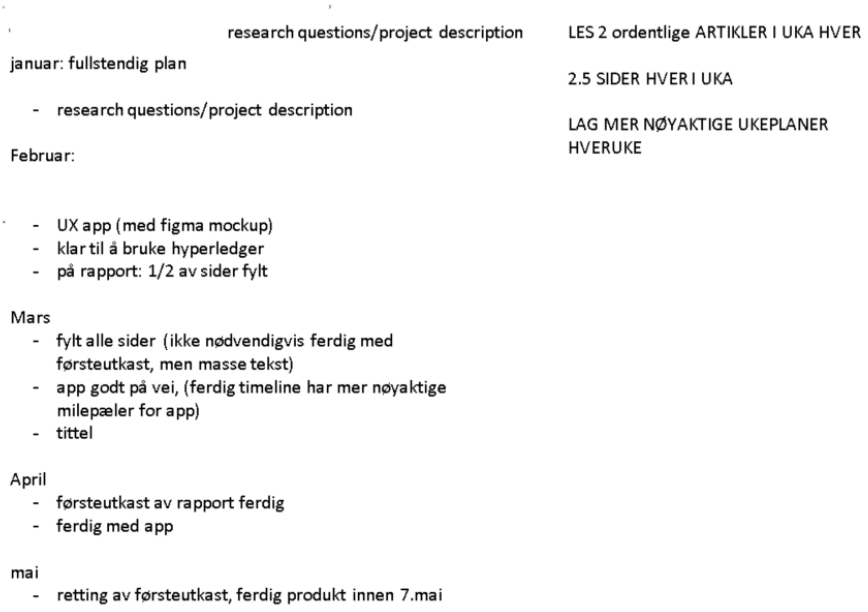


Figure A.1: Long term development plan created at the beginning of project

List of Figures

Bibliography

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[2] Ethereum, March 2022. URL https://ethereum.org/en/.

[3] Andrew L Goodkind, Benjamin A Jones, and Robert P Berrens. Cryptodamages: Monetary value estimates of the air pollution and human health impacts of cryptocurrency mining. *Energy Research & Social Science*, 59:101281, 2020.

[4] A digital future on a global scale. https://ethereum.org/en/upgrades/vision.

[5] 51% attacks, March 2022. URL https://dci.mit.edu/51-attacks.

[6] What are smart contracts on blockchain? URL https://www.ibm.com/topics/smart-contracts.

[7] Smart contracts and chaincode. https://hyperledger-fabric.readthedocs.io/en/latest/smartcontract/smartcontract.html.

[8] Hyperledger foundation, January 2022. URL https://www.hyperledger.org/.

[9] https://www.linuxfoundation.org/tools/hyperledger/. https://www.linuxfoundation.org/tools/hyperledger/.

[10] Hyperledger fabric - a brief history. URL https://www.linkedin.com/pulse/hyperledger-fabric-brief-history-binh-nguyen.

[11] Endorsement policies. https://hyperledger-fabric.readthedocs.io/en/latest/endorsement-policies.html.

[12] Channel configuration (configtx). https://hyperledger-fabric.readthedocs.io/en/latest/configtx.html.

[13] The ordering service. https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html, .

[14] Artem Barger, Yacov Manevich, Hagar Meir, and Yoav Tock. A byzantine fault-tolerant consensus library for hyperledger fabric. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, 2021. doi: 10.1109/ICBC51069.2021.9461099.

[15] Identity. https://hyperledger-fabric.readthedocs.io/en/latest/identity/identity.html, .

[16] Steem: Powering communities and opportunities. URL https://steem.com/.

[17] Somee: Social media redefined for privacy, end user control and monetization, March 2022. URL https://somee.social/.

[18] Sapien: Social network for privacy and intependence. URL https://www.sapien.network/.

[19] Scott Cunningham. Somee's last effort to take your money, February 2022. URL https://www.publish0x.com/at-scottcbusiness/somee-s-last-effort-to-take-your-money-xyyrlgg.

[20] Sapien. Sapien white paper. https://coinpare.io/whitepaper/sapien-wallet.pdf, 2020.

[21] Steem. Steem white paper. https://steem.com/steem-whitepaper.pdf, 2018.

[22] Jonathan Goodwin. Social media shouldn't run on personal data. https://blog.sapien.network/social-media-shouldnt-run-on-personal-data-44fcd2fc29ad, 2020.

[23] Antorweep Chakravorty and Chunming Rong. Ushare: user controlled social media based on blockchain. In *Proceedings of the 11th international conference on ubiquitous information management and communication*, pages 1–6, 2017.

[24] Hyperledger sawtooth. https://sawtooth.hyperledger.org/.

[25] react-native-hlf-wrapper, April 2022. URL https://github.com/bityoga/react-native-hlf-wrapper.

[26] Jwt. https://jwt.io/.

[27] React native · learn once, write anywhere, January 2022. URL https://reactnative.dev/.

[28] Meta, January 2022. URL https://about.facebook.com/meta//.

[29] Make any app. run it everywhere, January 2022. URL https://expo.dev/.

[30] Java, January 2022. URL https://www.java.com/en/.

[31] Download android studio and sdk tools, January 2022. URL https://developer.android.com/studio?gclid=Cj0KCQjwsdiTBhD5ARIsAIpW8CJUMuqQEm_VgnB56y1Hboe9Ov6h2A18P3DlKXbxyb8bM4sMvEuXMrsaAmFdEALw_wcB&gclsrc=aw.ds.

[32] Code editing. redefined, February 2022. URL https://code.visualstudio.com/.

[33] React navigation: React navigation, February 2022. URL https://reactnavigation.org/.

[34] Asyncstorage · react native, March 2022. URL https://reactnative.dev/docs/asyncstorage.

[35] Flatlist · react native, February 2022. URL https://reactnative.dev/docs/flatlist.

[36] Using the effect hook, March 2022. URL https://reactjs.org/docs/hooks-effect.html.

[37] Hyperledger fabric go contract api. https://pkg.go.dev/github.com/hyperledger/fabric-contract-api-go.

[38] gorilla/mux. https://github.com/gorilla/mux.

[39] Gorilla mux, February 2022. URL https://pkg.go.dev/github.com/gorilla/mux.

[40] Mohammad Hanif Tadjik. Blockchain empowered data sharing. https://github.com/Hanifff/Blockchain-empowered-data-sharing, 2021.

[41] Hyperledger fabric: Using couchdb, March 2022. URL https://hyperledger-fabric.readthedocs.io/en/release-2.2/couchdb_tutorial.html.

[42] Martin Gilje Jaatun, Peter Halland Haro, and Christian Frøystad. Five things you should not use blockchain for. In *2020 IEEE Cloud Summit*, pages 167–169, 2020. doi: 10.1109/IEEECloudSummit48914.2020.00032.

[43] Asset transfer basic sample. https://github.com/hyperledger/fabric-samples/tree/main/asset-transfer-basic.

[44] Elon musk to acquire twitter. https://www.prnewswire.com/news-releases/elon-musk-to-acquire-twitter-301532245.html, April 2022.