Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

A Decentralized Social Network on IPFS

Ekjyot Singh
B.A.I. (Computer Engineering)
Final Year Project Aug 2020
Supervisor: Dr. Donal O'Mahony

School of Computer Science and Statistics
O'Reilly Institute, Trinity College, Dublin 2, Ireland

DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university

Ekjyot Singh	14th Aug, 2020
Name	Date

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Signed: Ekjyot Singh

Date: 14/08/20

Acknowledgements

This report was made possible with the contribution and support of many individuals:

First and foremost, I would like to thank my supervisor, Dr Donal O'Mahony, for his advice and guidance throughout this project.

I would also like to thank my friends and family for their constant support throughout my college years.

Abstract

Current social networks like Facebook and Twitter have become ubiquitous in our daily lives. Both of these platforms boast millions of active users around the world. Users in these platforms often exchange privacy for free use and, due to the client-server nature of these social networks, the platforms are susceptible to user censorship and single-point failure.

This project proposes a decentralized social network on IPFS which preserves privacy and anonymity of its users, enables users to have total control over who has access to their private data, is resilient against censorship and mass surveillance, provides access from multiple devices and still maintains all the features of current social networks.

The design makes extensive use of a data structure by OrbitDB called *ipfs-log* which is an immutable, operation-based conflict-free replicated data structure (CRDT) for distributed systems. These data structures maintain consistency of metadata and user data among multiple nodes.

Features implemented by this design include user profiles, adding friends, posts, direct and group messaging, discovery of other users, multi device support and login via username-password.

Table of Contents

List of Figures

1. Introduction

This chapter describes the motivation behind the project, problems with present social networking platforms and then arrives at a clear problem statement which this project aims to address.

1.1 Motivation

Current social networks like Facebook and Twitter have become ubiquitous in our daily lives. Both of these platforms boast millions of active users around the world, but users in these platforms often exchange privacy for free use and, due to the client-server nature, these platforms are susceptible to user censorship and single-point failure.

The extent of global surveillance by government bodies has been alarming in recent years. In 2013, Snowden-NSA leaks [1] revealed a massive effort by the US government agencies that were spying unlawfully on their own citizens. Another high-profile case involved the Chancellor of Germany [2], Angela Merkel whose personal phone was reportedly tapped by the NSA.

Recent revelations about the mishandling of private user data by Facebook and subsequent unethical use of that data by Cambridge Analytica [3] has brought privacy issues with the current social networks to the forefront. Moreover, due to the centralized nature of these platforms, they are also susceptible to censorship or being totally blocked by governments. The monopoly of companies like Facebook makes it harder for a user to move to another social network service because doing so would require moving every single friend to the new service, which requires setting up new accounts and downloading new applications for mobile devices.

There are a few “decentralized” social networks platforms like Mastodon, and Matrix which are made up of a collection of independently controlled “federated” servers instead of a single governing body. Although this federated approach is a great improvement over the traditional social network platforms, there are still several shortcomings as users still need to trust some third-party nodes.

The solution is a decentralized social network on IPFS. The utilization of a peer-to-peer network for a social network provides two main advantages:

- It is virtually impossible to block it or take it down since anyone can launch a node in a few seconds on their computer and two nodes within the same LAN are still able to communicate and operate without internet access, whereas in a centralized model, it is

easier to block access to the servers of the company concerned, or even to force it to shut down its servers and stop its activity.

- It is difficult to monitor as there is no central server to spy on nor a central directory to compromise, thus metadata collection is greatly minimized.

1.2. Aim

The overall aim of this project is to design and develop a decentralized social network on IPFS which preserves privacy and anonymity of its users, enables users to have total control over who has access to their private data, is resilient against censorship and mass surveillance, provides access from multiple devices yet maintaining all the features of current social networks.

2. State of The Art

This section examines the technologies and ideas which underlie the social network application designed. It starts with basic principles of cryptography, then talks about peer-to-peer and secure communication systems. Finally, it reviews CRDTs, PubSub, OrbitDB and IPFS, which are integral to the design proposed.

2.1 Cryptography

2.1.1 Symmetric Encryption

Symmetric encryption [4] [5] is a type of encryption where only one key is used to encrypt and decrypt data. The plaintext (data) is converted into an unreadable format (ciphertext) using an encryption algorithm called a cipher. Once the intended recipient who has the key receives the message, the algorithm reverses the encryption and converts the ciphertext into readable form. The secret key between the sender and the receiver can be a specific password or code or can be a random string of characters.
The security of this encryption depends on how difficult it is to guess the secret key using brute force. The longer the encryption key is, the harder it becomes to crack it.

2.1.2 Asymmetric Encryption

Asymmetric cryptography [6] or public-key cryptography is an encryption scheme that encrypts and decrypts the data using two separates yet mathematically connected cryptographic keys. The key pairs consist of a ‘Public Key’ and a ‘Private Key’. One key in the pair can be shared, that is the public key. The other key, as the name suggests, meant to remain confidential, is the private key. The public key is used for encryption and the private key, can only be used by the authenticated recipient to decrypt the message. And so, a message that is encrypted using a public key can only be decrypted using a private key, while a message encrypted using a private key can be decrypted using a public key.

The keys are generated in a pair by a cryptographic algorithm and hence both keys are mathematically connected with each other. The RSA [7] [8] (Rivest-Shamir-Adleman) algorithm is based on this principle, first published in 1978.

Another essential property of asymmetric cryptography is the ability to sign data, which gave rise to digital signatures. Digital signature is a “mathematical scheme for verifying the authenticity of

digital messages or documents". [9] It uses the described key-pair to verify the authenticity and integrity of transmitted data. The sender combines a message with their private key to create the digital signature. The *publicly* known public key is then used to verify whether the signature is valid. If validated, the recipient is assured that the message was created by a known sender, and the message was not altered in transit, thereby affirming authentication and integrity.

Consider a case where Alice wants to send Bob a message and Alice wants to make sure that Bob knows the message came from her. To do this, Alice needs to create a digest and sign it.
To create a digest, Alice uses a cryptographic hash function to convert the message into an ID number. To sign the digest, Alice needs a secret key that one else knows.
To verify her signature, Alice needs to publish her public key so that it is available to everyone. Using her private key, Alice signs the digest.
At this point, anyone who received Alice's message to Bob can use Alice's public key to verify that Alice created the message. However, Alice might only want Bob to read and verify her message. To accomplish this, Alice needs to use Bob's public key to encrypt the message. Alice uses Bob's published public key to encrypt her message. This ensures that only Bob (anyone in possession of Bob's private key) can read the message. Because her signature is part of the message, Bob will know that the message came from her. Bob uses his private key to decrypt the message and then uses Alice's public key to verify the signature.

2.1.3 Cryptographic Hashing

Hashing functions are an integral part of cybersecurity and cryptocurrency protocols. It is a mathematical algorithm that encrypts data of an arbitrary size into a unique string of a fixed size. It is a one-way function, i.e., encodes data such that the calculated hash cannot decipher the input data. Given that the same mathematical algorithm is used to encrypt data, the same hash will be produced for identical input.

Cryptographic hash functions add security features to typical hash functions, making it more difficult to detect the contents of a message or information about recipients and senders and are used in cases like checking the integrity of messages and authenticating information.

SHA (Secure Hash Algorithm) is a group of cryptographic functions designed by the United States National Security Agency (NSA). In 2014, the Certificate Authority Security Council published an article explaining the need to move all usages of SHA-1 to SHA-2. SHA-2 includes significant changes from its predecessor, SHA-1 (160 bits in length). The SHA-2 family consists of six hash functions with hash values that are 224, 256, 384 or 512 bits. Attacks against hash functions are

measured against the length of time required to perform a brute-force attack. As such, increases in hash function lengths are necessary to maintain an acceptable margin of security.

To represent the string “Hello” using SHA-1, we get the following output:

```
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
```

The exact same input string generates the following output using SHA-256:

```
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
```

The second hash is longer than the first one because SHA-1 creates a 160 bit hash, while SHA-256 creates a 256 bit hash.

Hashes can also be represented in different bases (base2, base16, base32, etc.). This can be indicated by prepending the hash. E.g.- 0x is an indicator prefix that tells us that a hash is represented as a base 16 (hexadecimal) number.

2.1.4 Elliptical Curve Cryptography

Elliptic Curve Cryptography or ECC is an alternate and optimized approach to public key cryptography, from the well-known RSA, based on elliptic curves over finite fields. An elliptical curve is illustrated by the following equation:

$$y^2 = x^3 + ax + b$$

Based on the values given to the constants, ***a*** and ***b***, the shape of the curve is determined. Elliptical curve cryptography uses these curves over finite fields to create a secret that only the private key holder is able to unlock. The larger the key size, the larger the curve, and the harder the problem is to solve. The biggest differentiator between ECC and RSA is key size compared to cryptographic strength. The optimization aspect of EEC is that it requires smaller keys compared to non-ECC cryptography while maintaining equivalent security. This results in reduced storage costs and transmission speeds of providing such an encryption scheme.

Given two points **A** and **B** on an elliptic curve, finding a number ***n*** such that **B=nA** (if it exists) can take an enormous amount of computing power, especially when ***n*** is large. Elliptic curve cryptography exploits this fact: The points **A** and **B** can be used as a public key, and the number n

as the private key. Anyone can encrypt a message using the public key, but only the intended recipient of the message, in possession of the private key, the number ***n*** can decrypt it. [10]

Curve25519, [11] one of the fastest ECC curves, offers 128 bits of security and designed for use with the elliptic curve Diffie–Hellman (ECDH) key agreement scheme. The original Curve25519 paper defined it as a Diffie–Hellman (DH) function. Daniel J. Bernstein has since proposed that the name Curve25519 be used for the underlying curve, and the name X25519 for the DH function.

2.1.5 Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange is one of the first public-key protocols conceived by Ralph Merkle and named after Whitfield Diffie and Martin Hellman. It is a way of securely establishing a shared secret between two parties over a public channel, in a way that the secret cannot be seen by observing the communication. The communicating parties, as a result, create a key together. [12]

Traditionally, to facilitate a secure encrypted communication between two parties, it was required for them to exchange keys physically. With the Diffie-Helman exchange method, parties with no prior knowledge of each other can safely develop and exchange keys over an insecure channel.

If Alice and Bob wish to establish a shared secret using Diffie-Helman's protocol, the following exchange takes place:

1. Alice and Bob publicly settle on a prime number *p* and a number *g* which is coprime to *p-1*.
2. Alice picks secret number (*a*) and computes $g^a \ mod \ p$ and sends that result back to Bob. Let the result be A.
3. Bob does the same thing; picks a secret number (*b*) and computes $g^b \ mod \ p$ and sends it to Alice. Let the result be B.
4. Now, Alice takes the number Bob sent and computes the exact same operation with it. So that's $B^a \ mod \ p$.
5. Bob does the same operation with the result Alice sent, so: $A^b \ mod \ p$.

The resultant answer in step 4 and 5 is equivalent:
$(g^a \bmod p)^b \bmod p = g^{ab} \bmod p$
$(g^b \bmod p)^a \bmod p = g^{ba} \bmod p$

Alice and Bob never know what secret number each used to get to the result but arrive at the same result.

p and *g* should be chosen suitably, in particular, the order of *p* should be large. The base, *g*, can be a relatively small number like 2, but it needs to come from an order of *p* that has a large prime factor.

2.2 Directed Acyclic Graph (DAG)

A directed acyclic graph is a graph where it is impossible to come back to the same node by traversing the edges; the edges of this graph move only in one direction. Information is passed from one node a circle to another along the different edges. Example: a linked list data structure, A→B→C, is an instance of a DAG where A references B references C. B is the *child* or *descendant* of A and C is the child of B.

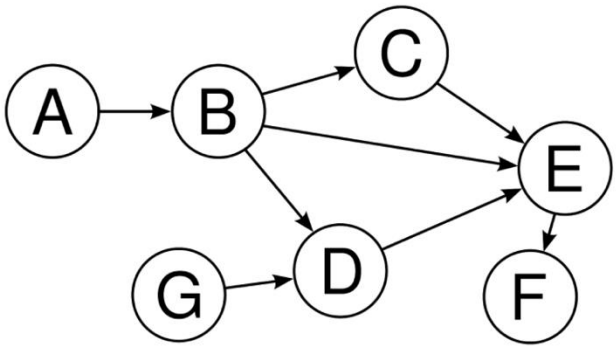

Figure 1: DAG. [13]

A DAG represents a series of ordered activities. The order is defined by the graph topology, where each node, depicted by a circle (called vertex), represents an activity. The connected lines (edges) and the arrow represent the flow from one activity to another.

2.3 Merkle DAG

Merkle DAGs are self-verified structures used to provide causality information that can be easily and efficiently shared in peer-to-peer environments.

In Merkle DAGs, each node has a unique ID which is the hash value of the node's contents (payload carried by the node and the list of identifiers of its children). Identifying a Merkle DAG node by the value of its hash is referred to as 'content addressing'. This unique identifier is named as content identifier (CID).

The CID of a node is "linked to the contents of its payload and those of all its descendants." As a result, Merkle DAGs can only be constructed from the nodes without children (leaf nodes, in graph terminology). Parent nodes are added *after* the children nodes because the children's identifiers must be computed in advance to be able to link them.

Another important consideration is that Merkle-DAG nodes are immutable. Any change in a node will alter its identifier and thus affect all the ascendants in the DAG, resulting in a completely different DAG.

Merkle DAGs are very widely used in source control systems like Git "use them to efficiently store the repository history, in a way that enables de-duplicating the objects and detecting conflicts between branches."

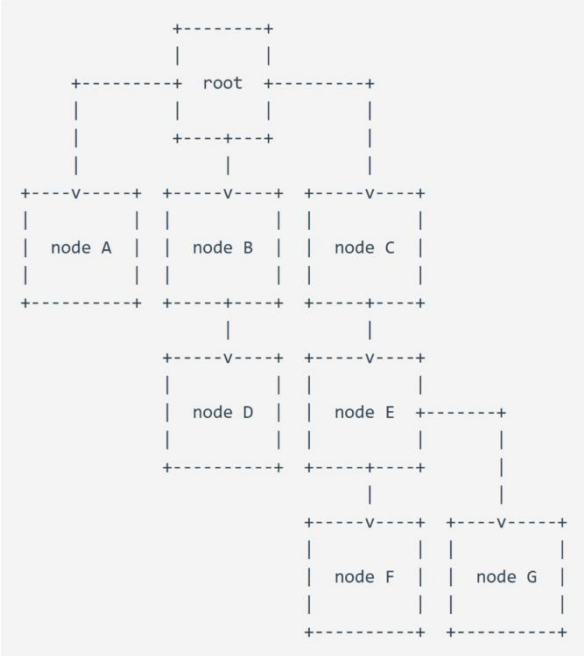

Figure 2: Merkle tree diagram. [14]

2.4 Peer-to-Peer

Peer-to-peer, or P2P, is a distributed computer network made up of computers and devices, referred to as peers, that communicate by sharing and exchanging data. Each peer in such a network is equal to the other peers. There are no privileged peers, no central or primary administrator device in the center of the network; all peers have the same rights and duties as the others. A pure P2P is the most egalitarian of computer networks.
Not only is data shared, but every resource available in the network is shared between peers- processor usage, disk storage capacity, or network bandwidth.
Generally, a peer communicates directly with their known neighbors (connected peers) who in turn, communicate with their neighbors. In this way, information is transmitted through the network. Unlike a traditional client server approach, peers are both clients and servers at the same time and as a result, peers consume services from other peers while also providing services simultaneously. Hence, peer-to-peer networks are ideal for file sharing since they allow peers to receive files and send files simultaneously.

Many networks exist as hybrid P2Ps (combining features of a P2P network with the features of a client-server network), with BitTorrent being the most widely known example. BitTorrent is a protocol for P2P file sharing. When we download a file from using a BitTorrent platform, the file is downloaded onto our computers in bits and parts that come from many other computers, that are also connected to the same P2P network and already have that file, or at least parts of it. At the same time, the file is also sent (uploaded) from our computer to other devices that are asking for it. The larger the P2P network, the faster the file sharing.

Unstructured peer-to-peer networks do not impose a particular structure on the network, but rather are formed by nodes that randomly form connections to each other. On the other hand, structured P2P networks are generally based on a distributed hash table (DHT), which is used to assign ownership of each file to a particular peer. This enables peers to search for resources on the network using a hash table (key-value pairs) and any participating node can effectively retrieve the value associated with a given key. [13] [14] [15] [16]

2.4.1 Distributed Hash Table

A distributed hash table (DHT) is a distributed system that provides a lookup service similar to a hash table: key-value pairs are stored in a DHT, and any participating node can efficiently use the key to find the respective value. Keys are unique identifiers which map to particular values, which in turn can be anything from addresses, to documents, to arbitrary data. A DHT is the basis for

supporting core P2P services since their item-to-location mapping can be used to (i) place data on specific peers and (ii) efficiently locate the data by identifying the responsible peer. [17] [18] [19]

Keys are converted to hashes with an underlying hashing function, which are used as indices into the hash table. A DHT gives a dictionary-like interface, but the nodes are distributed across the network and so each node is responsible for a subset of the DHT. The nodes are arranged in an overlay network, allowing them to find the owner of any given key in the network. [20]

When a node receives a request, it can either return the requested data or the request is carried on to another peer until the node that has the requested data is found. There are several implementations that dictate how the requests are passed on if the first node can't answer it. A request can be passed from one node to another, with the requested data returned following the same path; it can be passed on from node to node with the last node contacting the requesting node or it can be answered following the hash table of the node that is most likely to have the requested data.

Consider a simple DHT example shown in figure 3. This assumes a Chord DHT that uses consistent hashing, which treats nodes as points on a circle. The basic approach is to pass the query to a node's successor, if it cannot find the key locally.

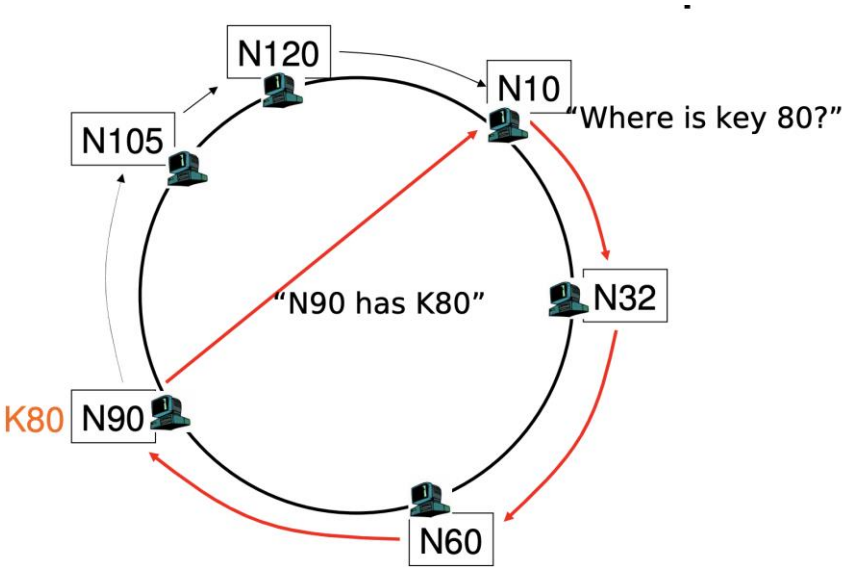

Figure 3: DHT Chord Basic Lookup. [19]

2.5 Secure Communication

Secure communication is when communicating entities have a level of assurance that a third party cannot listen in. The transmission is not susceptible to eavesdropping or interception.

There are various tools used to obtain security. The most basic one being encryption by which data is rendered almost impossible to read by an unauthorized party. Encryption itself does not prohibit interception. And so, even if the transmission is intercepted, a third party, without the means to decrypt the encrypted data, will not be able to comprehend it. It is possible to decrypt the message without possessing the key, but, for a well-designed encryption scheme, considerable computational resources and skills are required.

Another tool to secure communication is an anonymous network. Such a network is made up of a large number of users running the system and where it is ultimately impossible to transparently see which pieces of data are coming from users and where the destination is. Examples are Tor and anonymous P2P. Anonymity of participants is achieved by special routing overlay networks that hide the physical location of each node from others. [21]

2.5.1 Attacks

In network attacks, attackers are focused on infiltrating the network perimeter and gaining unauthorized access to the internal system. Very often, attackers will combine different types of attacks, for example compromising an endpoint, exploiting network vulnerabilities, injecting malware, with the objective of stealing data or performing other malicious activity. This section discusses man-in-the-middle attacks and replay attacks.

A **man-in-the-middle attack** (MITM) is an attack where a malicious hacker inserts itself between the communications of a client and a server. Broadly speaking, there are two kinds of MITM attacks, one that involves physical proximity to the intended target. Traditionally, the hacker will gain access to a poorly secured Wi-Fi network, most likely found in public places with free Wi-Fi. Once the attacker breaks into the vulnerable network, they can insert tools between the victim's device and the internet, intercept and read any transmitted data.
The second type of MITM involves malicious software, or malware. This second form is also called a man-in-the-browser attack. Through phishing, the attacker can trick the user into downloading malicious software. While the user is completely oblivious of such an injection, the

malware records the data sent between the victim and specific targeted websites, such as banks, and transmits it to the attacker.

A common example is session hijacking. In this scenario, the hacker hijacks a session between the client and the server and continues the session where the server believes that it is still communicating with the client.

(a) A client connects to a server.
(b) The attacker's computer gains control of the client.
(c) The attacker's computer disconnects the client from the server.
(d) The attacker's computer replaces the client's IP address with its own IP address and spoofs the client's sequence numbers.
(e) The attacker's computer continues to communicate with the server and the server believes it is still communicating with the client.

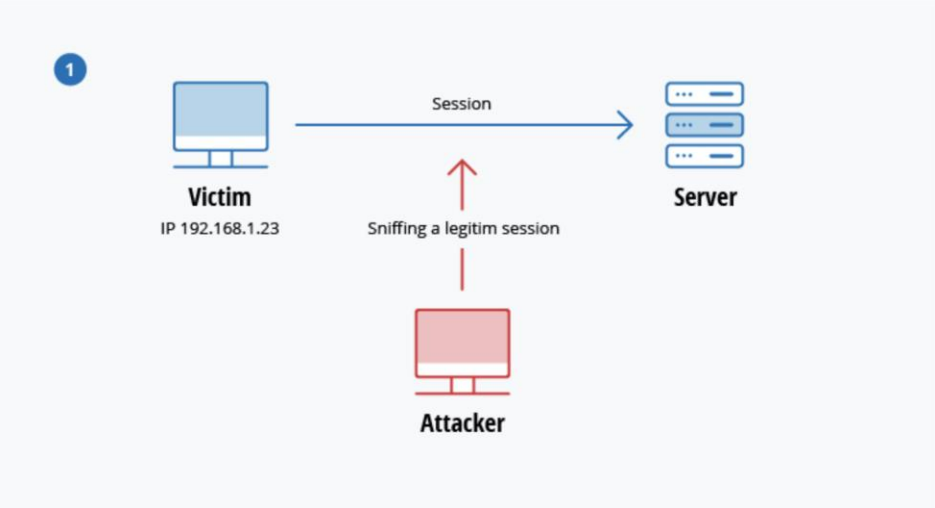

Figure 4: MITM Attack 1. [25]

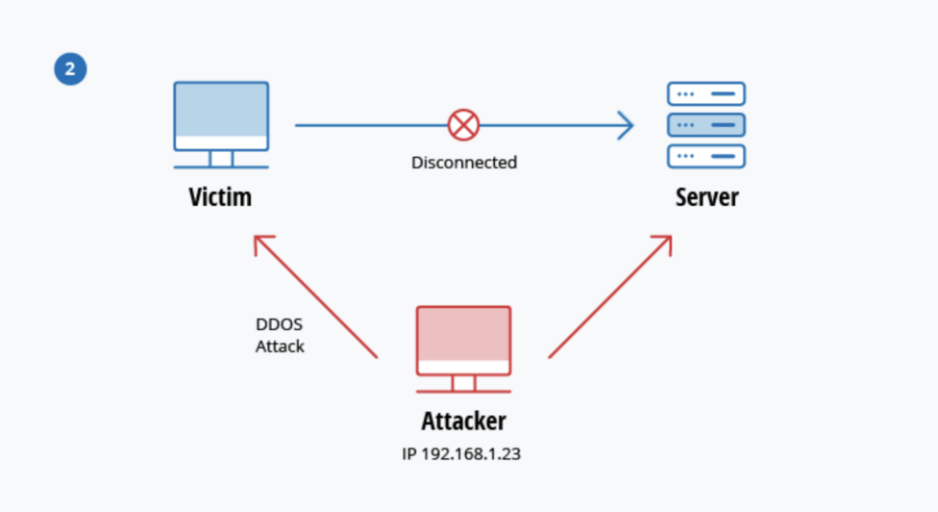

Figure 5: MITM Attack 2. [25]

Another common attack is **replay attack**. Such an attack occurs when valid data transmission is maliciously repeated or delayed. The hacker eavesdrops on the network communication, intercepts it and fraudulently misleads the victim into doing what the hacker wants.
This attack can also be described as "an attack on a security protocol using replay of messages from a different context into the intended (or original and expected) context, thereby fooling the honest participant(s) into thinking they have successfully completed the protocol run." [22]

Consider Alice and Bob communicating over a network and Bob trying to prove his identity to Alice. Alice requests Bob's password to confirm his identity which Bob sends. Consider a third person, E, who has been eavesdropping and obtains the password Bob sent. E can then pose as Bob to Alice and when Alice asks for Bob's password, E is able to give the password that was

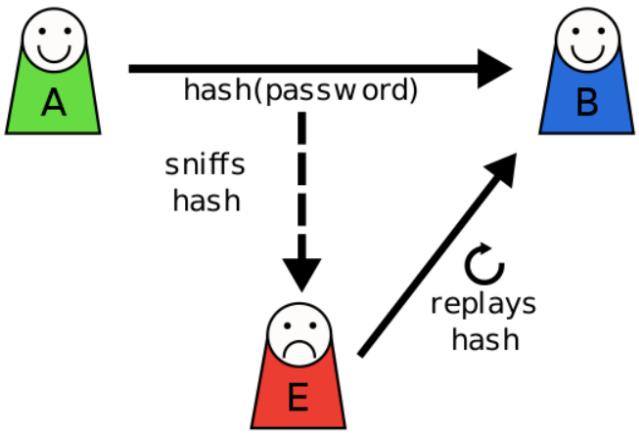

Figure 6: Replay Attack. [22]

earlier intercepted. Alice is likely to respond to this request positively unless she has a good reason to be suspicious. And hence, E is granted access as Bob.

2.5.2 End-to-End Encrypted Messaging

End-to-end encryption (E2EE) is a system of communication where only the communicating users can read the messages. It prevents potential eavesdroppers and interceptors from being able to access the cryptographic keys needed to decrypt the conversation. [23]

In many messaging systems, the messages pass through an intermediary and are stored by third parties from which the receiver retrieves them. The messages are 'encrypted', however, only in transit and thereby accessible by the server. This allows third parties to view the data and can be misused by anyone who has access to the stored data on third party systems.

E2EE prevents data from being read or modified, other than by the true sender and recipient(s). End-to-end encryption takes place on either end of a communication. A message is encrypted on a sender's device, sent to the recipient's device in an unreadable format, then decoded for the recipient.

Currently, Apple, Facebook, and WhatsApp are among the biggest services to offer E2EE. Among bigger webmail providers, Outlook is the only one with the proper encryption.
At the time of writing this paper, Zoom, the video-conferencing specialist faces legal challenges over end-to-end encryption. Zoom had previously asserted that it offers end-to-end encryption, but this claim came into question revealing that the platform only uses TLS encryption between individual users and service providers as opposed to directly between the communicating users. This would allow the service to access user data if it chose to and leave it open to potential eavesdropping by third parties. [24] [25] [26] [17] [27]

2.6 Conflict-free Replicated Data Type

CRDT, or Conflict-Free Replicated Data Type, is a data structure used to achieve strong eventual consistency (SEC) and monotonicity (absence of rollbacks). "Conflict-freedom ensures safety and liveness despite any number of failures. An update does not require synchronization, and CRDT replicas converge to a correct common state." [28]

CRDTs can be duplicated across multiple nodes in a network. The replicas can be concurrently updated, independently, without any need of synchronization between each other. Any

inconsistency that does come up, is ultimately resolved, avoiding complex roll-back and conflict resolution mechanisms.
They are of two types:

a. Operation-based CRDTs are also called **commutative** replicated data types. CmRDTs make use of the operation-based synchronization model, by simply transmitting the update operation that changed the local state, to every other replica. CmRDTs require all operations to be delivered according to some specific order, causal order being the most common. To finally converge to a correct common state, operations need to be commutative. CmRDTs prove to be effective because of the propagation of update information only along with the causal order, hence allowing larger states. [29] [30]

b. State-based CRDTs are called **convergent** replicated data types, or CvRDTs. In contrast to CmRDTs, CvRDTs employ a state-based synchronization model, where replicas synchronize by periodically exchanging their full local state. When a replica receives state information from another replica, it updates its current local state to match the received state by using a merge function. The merge function must be commutative, associative, and idempotent. It provides a join for any pair of replica states.
   One drawback of this approach is the very heavy communication overhead that propagates to all the replicas. However, state based CRDTs are easier to implement as all information is carried by the state. [30]

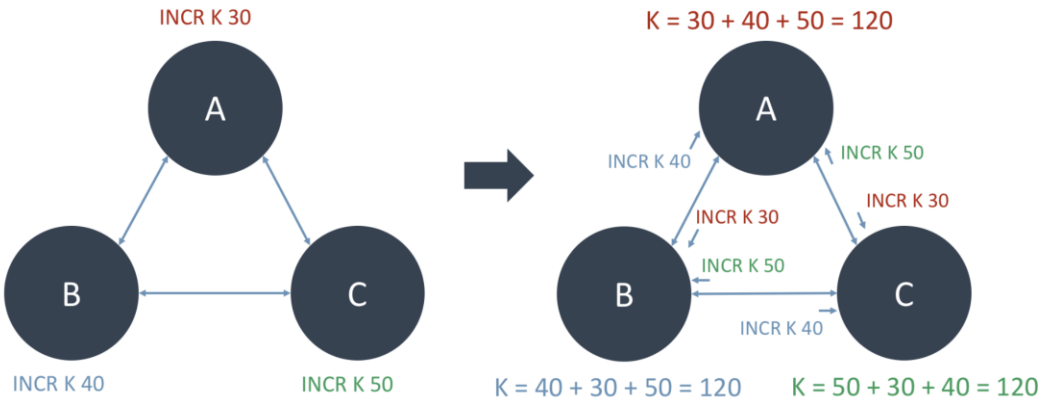

Figure 7: Example of CRDT. [30]

The above figure shows three nodes, each incrementing K by a certain amount. Starting with Node A, A is connected to B and C which relay their updates to A. The ultimate state of A is at K = 30

(update at A) + 40 (update from B) + 50 (update from C). The same process happens at B and C. A and C propagate their updates to B and likewise, A and B do the same to C. As a result, all three nodes settle at a common state with K = 120.

2.7 Pub-Sub

PubSub is a Publish/Subscribe system where peers attach themselves to topics, they are interested in. Any peer engaged with a topic is said to be subscribed to that topic.

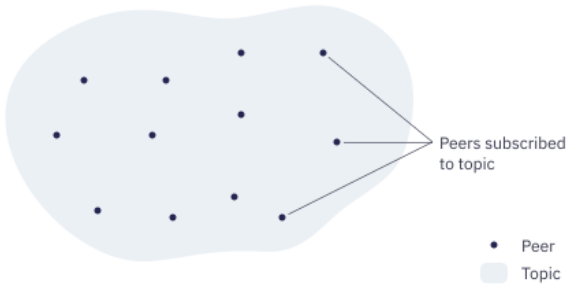

Figure 8: Peers subscribed to a topic. [35]

Peers can send messages to topics. Each message gets delivered to all peers subscribed to the topic.

Figure 9: Peers sending messaging within a topic. The message propagates to all peers in that topic. [35]

The concept of Pub/Sub is used in (a) chat rooms, where each chat group is a pub/sub topic and members of the group can send and receive messages to/from other members in the same group; (b) file sharing, where each file is a pub/sub topic that can be downloaded; uploaders and

downloaders publicize the pieces of the file they have in the pub/sub topic and coordinate downloads that happen outside the pub/sub system.

Types of Peering

Peers connect with other peers through full-message peerings or metadata-only peerings.

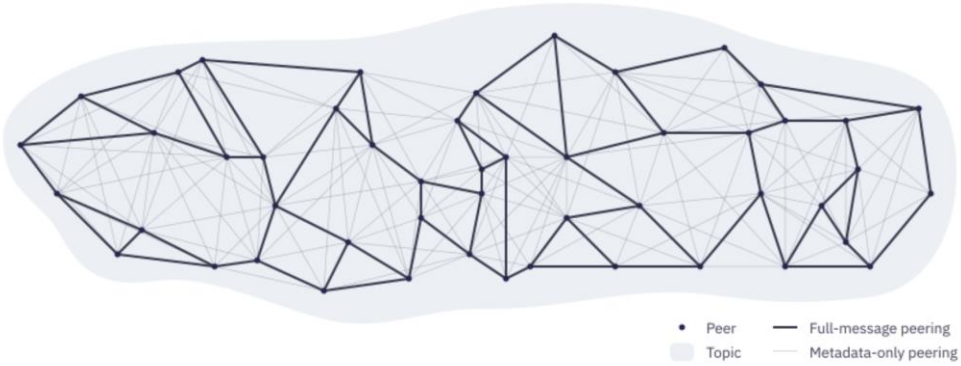

Figure 10: Types of peerings. [35]

Full message peerings transmit the full content of messages throughout the network. Such connections are sparse in the network to be able to control the network traffic; each peer only forwards messages to a few other peers, rather than all of them.

On the other hand, the metadata-only peerings is a dense network of peers that are not a part of full-message peering. This network mainly performs functions to maintain the network of full-message peers.

Sending Messages:

When a peer wants to send a message, it sends a copy to all full-message peers it is connected to. Similarly, when a peer receives a new message from another peer, it stores the message and redirects a copy to all other full-message peers it is connected to.

Subscribing and Unsubscribing

Peers keep a track of the topics that their immediate (directly connected) peers are subscribed to. Using this information, each peer is able to build a picture of the topics around them and which peers are subscribed to each topic. This is maintained by sending messages. When a new

connection is made between two peers, messages of what topics each is subscribed to or topics that a peer has unsubscribed to are exchanged. This message propagates to all connected peers.

2.8 InterPlanetary File System

IPFS is a peer-to-peer network for storing and sharing data in a distributed file system. IPFS borrows many ideas from the past peer-to-peer systems, including DHTs, Git, BitTorrent and Self-Certified Filesystems and connects these proven technologies into a single cohesive system.

At its core, IPFS relies on three basic underlying principles which build upon each other to enable the IPFS ecosystem: Unique Identification via Content addressing, Content Linking via Directed Acyclic Graphs, and Content Discovery via Distributed Hash Tables.

IPFS uses a *Content Identifier (CID)* to identify content based on what's in a file as opposed to location-based addressing used by the centralized web. All the files that use IPFS protocol have a content identifier, that is its cryptographic hash. This hash is unique to the content that it came from, even though it may look short compared to the original content. This means that any difference in the contents of the files will produce a totally different CID and the two files with the same content added to two different nodes will produce the exact same CID.

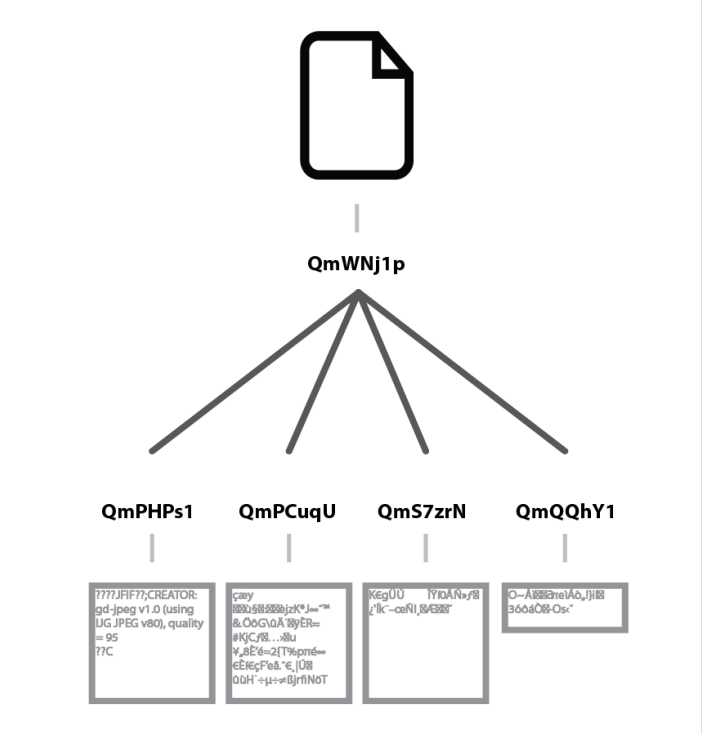

Figure 11: Merkle DAG. [36]

IPFS stores data in an *Object Merkle DAG*, which is a directed acyclic graph where each node has a unique identifier that is a hash of the node's contents. Just like CIDs discussed above, identifying data objects by the hash of its contents is content addressing.
All the data stored by IPFS is split into multiple 256 kb sized blocks which are then used to build a Merkle tree. Splitting a file into small blocks means that different chunks of a file can come from different sources therefore, two similar files stored on IPFS can share the blocks of Merkle DAG which they have in common. This provides deduplication as all objects that have exact same contents are equal and only need to be stored once.

Every block stored in the Merkle DAG has its own CID. Which means that if a file is made up of 20 different blocks, each one of those 20 blocks will also have their own CIDs and the CID of the file will be a hash of the CIDs from all the blocks. This provides tamper resistance as changing a file even slightly results in a totally different CID therefore making IPFS immune to data tampering or corruption.

```
type IPFSLink struct {
        Name string

        Hash Multihash

        Size int
        //size of target
}
```

```
type IPFSObject struct {

        links [ ]IPFSLink
        //array of links

        data [ ]byte
}
```

Figure 12: IPFS object format

To find participating peers that are in ownership of a particular file, a distributed hash table (DHT) is used to search for resources on the network using key-value pairs in the table; any node can retrieve the value associated with a given key. The libp2p project is the part of the IPFS ecosystem that provides the DHT and handles peers communicating with each other.

Using the DHT, the first step involves knowing which peers are responsible for storing the required pieces of data. Using the table, those peers can be located. In order to get to content, DHT is queried twice with libp2p. Next, a connection to established with peer to get the data. For this exchange of data, IPFS uses a module called Bitswap.

Bitswap is responsible for connecting peers to each other, requesting blocks of content from other peers and having those peers send back the requested blocks. The received blocks can be verified by hashing their content to get CIDs and compare them to the CIDs requested. [31]

```
type NodeId Multihash
type Multihash []byte
//self-describing cryptographic hash digest

type PublicKey []byte
type PrivateKey []byte

type Node struct {
        NodeId NodeID
        PubKey PublicKey
        PriKey PrivateKey
}
```

Figure 13: Node data structures

2.8.1 Interplanetary Naming System

As discussed in the last section, IPFS uses content-based addressing which means that any file, image or website on IPFS is represented by a long string of random letters and numbers like the one below.

```
/ipfs/QmbezGequPwcsWo8UL4wDF6a8hYwM1hmbzYv2mnKkEWaUp
```

Not only is it hard to read, every time something in that file or website is changed, it results in a totally different IPFS hash. The InterPlanetary Name System (IPNS) solves these issues by

creating human-readable addresses that can be updated and hence can be made to always point at the latest hash of a website resulting in a constant address.

```
NodeId = hash(node.PubKey)
```

A name in IPNS (the hash follows `/ipns/` in a link) is the hash of a public key. It is associated with a record containing information about the hash it links to that is signed by the corresponding private key. New records can be signed and published at any time. In other words, IPNS is a global namespace based on Public Key Infrastructure (or PKI) which enables building of trust chains so that one can follow a public key to its route peer. [32]

```
/ipns/QmSrPmbaUKA3ZodhzPWZnpFgcPMFWF4QsxXbkWfEptTBJd
```

2.8.2 Mutable File System (MFS)

Files in IPFS are content-addressed and immutable. This means that when files change, the content identifier (CID) of those files changes too. Also, if files are too big to fit in a single block, IPFS splits the data into multiple blocks and uses metadata to link it all together. Because of the aforementioned reason files can be very complicated to edit on IPFS.

Mutable File System (MFS) is a tool built into IPFS that allows users to treat files like a regular name-based filesystem. It allows normal functions of a filesystem like adding, removing, moving or editing of files and handles all the complex operations like updating links and hashes in the background.

2.9 OrbitDB

Orbit-db [33] is a serverless, distributed, peer-to-peer database that uses IPFS for its data storage and IPFS Pubsub to automatically sync databases with peers. It’s an eventually consistent database that uses CRDTs for conflict-free database merges.

Being a peer-to-peer database, each peer has its own instance of a specific database. If Twitter were using OrbitDB then instead of a single global database of tweets where millions of users write concurrently, each user would have their own database for their tweets. To see another’s tweets, one would subscribe to a user's feed, i.e. replicate their feed database, thereby ‘following’ them. [34]

OrbitDB creates and manages mutable databases centered around the IPFS `get` and `set` functions. OrbitDB stored the databases in a distributed manner on IPFD by using CRDTs (discussed in section 2.4). In the context of OrbitDB, CRDTs are logs with specifically formatted "clock" allowing multiple users to perform independent and asynchronous operations on the same distributed database. When the peers share these logs with each other, the clock values ensure that any inconsistencies are resolved. Therefore, it is guaranteed that every peer will eventually reach a common state of the database. [35]

Due to its distributed nature and is stored locally, it is offline-compatible, offering more accessibility but will be outdated the longer the app is offline. Peers can also be granted/denied certain permissions based on a special hash key, unique for each peer.

OrbitDB offers a number of databases for a variety of use cases:

- **log**: an immutable (append-only) log with traversable history. Can be used for messages.
- **feed**: a mutable (entries can be added and removed) log with traversable history. Can be used for "shopping cart" or for a "blog feed".
- **keyvalue**: a key-value database, the most basic type of database
- **docs**: a document database to store JSON documents and can be indexed by a specified key. Can be used for building search indices or version controlling documents and data or describing products.
- **counter**: Useful for counting events from **log**/**feed** data.

2.9.1 Ipfs-log

All databases in OrbitDB are implemented on top of ipfs-log. It is an immutable, operation based CRDT for distributed systems. It's an append-only log that can be used to model a mutable, shared state between peers in p2p applications.

Each entry in the log is saved in IPFS and each points to a hash of the previous entry(ies) thereby forming a graph. Logs can be divided and joined back together. [36]

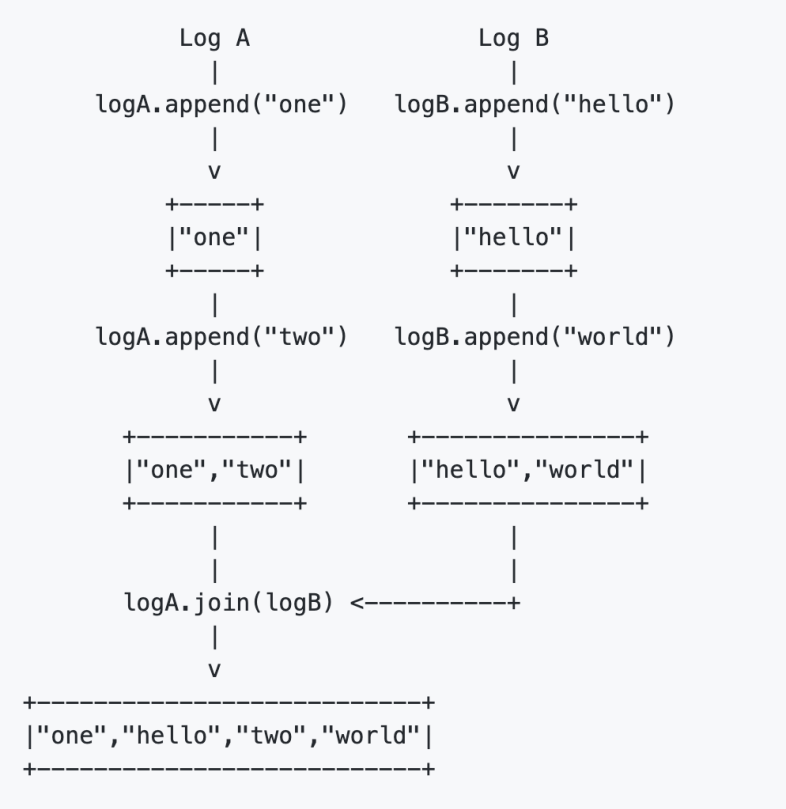

Figure 14: ipsf-log. [42]

2.10 Scrypt - Password-Based Key Derivation

Scrypt Password Based Key Derivation Function, PBKDF, is an algorithm designed to make it highly memory intensive to perform large-scale custom hardware attacks. Scrypt is also used in many cryptocurrencies as a proof-of-work algorithm. [37]

A key derivation function (KDF) is a cryptographic hash function that derives secret key(s) from passwords or other data sources using a pseudorandom function. KDFs are used to stretch keys into longer keys, thereby increasing the computational work to crack passwords.

HMAC is an example of a pseudorandom function used by PBKDF2, which is applied to the input password along with a salt value. This process is repeated to create a derived key of a specified length.

Password-based key derivation functions provide a solution to situations:

First, if an attacker gets access to a password file, they do not immediately possess the original passwords and second to create cryptographic keys to be used for encrypting and authenticating data. [38]

Scrypt Parameters:

- **Passphrase, P** is the user password; a string of characters to be hashed.
- **Salt, S** is a uniquely and randomly generated string of characters that modifies the hash to protect against Rainbow table attacks.
- Parameter **r** ("blockSizeFactor") specifies the block size (8 is commonly used).
- **N** ("costParameter") is the CPU/Memory cost parameter, must be a power of 2 and satisfy $N < 2^{(\frac{128 * r}{8})}$
- Parallelization parameter **p** ("parallelizationParameter") is a positive integer satisfying
$$p \leq \frac{((2^{32} - 1) * 32)}{(128 * r)}$$
- The desired output length **dkLen** is the length in octets of the key to be derived ("keyLength"); $dkLen \leq (2^{32} - 1) * 32$

The PBKDF2 Scrypt function takes in (P, S, c, dkLen) as is defined in RFC 2898 [39], where *c* is an iteration count. The iteration count is used to slow down the computation, and the salt is used to make pre-computation costlier. The notation for specifying a usage of PBKDF2 with *c = 1* is used by RFC 7914.

3. Solution Design

As with any social network, the effectiveness of a platform depends vastly on the number of active users. It is therefore very important to provide an easy to use and familiar user experience without sacrificing on the decentralization or privacy.

The utilization of a peer-to-peer network for a social network makes it impossible for the network to be taken down, blocked or monitored since there is no central server or any single point of failure. IPFS provides all the basic ingredients and tools necessary to develop such a decentralized social network. However, designing a social network on a peer-to-peer network does bring several design challenges. Since there is no central storage, all the data is stored on the user's devices and it is impossible to access content stored on a device that is offline. Without any central server to maintain the consistency of data, it becomes very difficult to maintain consistency in data among different peers. The solution aims to solve this issue by using a combination of CRDTs and PubSub to replicate data among peers.

Current "decentralized" social network implementations like Mastodon fall short in providing complete decentralization. These platforms provide a federated approach to decentralization where users still have to trust a central node. This solution will try to implement a fully decentralized social network where users don't have to trust any central node with all the features of current social networks.

The final solution aims to contain the following features:

- Preserves privacy and anonymity of its users
- Provides all the basic features of current social networks, e.g. messaging, posts, groups, profile, comments
- Resilient against censorship and mass surveillance by minimizing metadata leakage
- Provides users control over who can see what in their profile with fine granularity
- Multi device support

3.1 Design

This section aims to explain how different technologies and concepts were used from the research to develop a solution that achieves all the design goals specified in the previous section.

3.1.1 System Overview

The *base identity* of each account is a public/private Ed25519 key pair. This key pair serves as the permanent identity of an account and is referred to in this system as the *Account ID* key pair. The *Account ID* key pair also serves as an account's IPNS identity, which is published in the IPNS DHT once per account. The IPNS id of a user account always points to the latest hash of a user's profile and allows anyone to resolve the account's public key to the user's profile stored in IPFS.

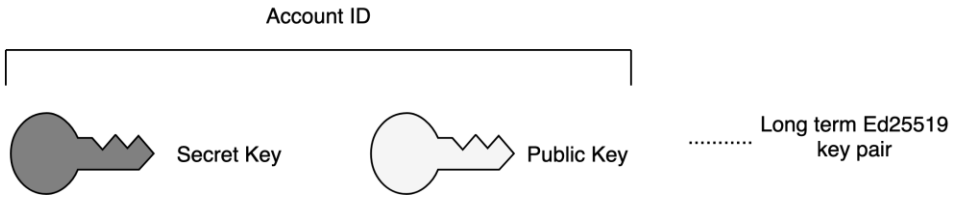

Figure 15: Account ID key-pair

The most basic element of this design is a user *profile*, which stores all the public and private data for an account. A profile is simply an IPFS object and can be thought of as a file directory with multiple subfolders. The *profile* object links to other files on a user's profile that are stored on IPFS and can therefore include any amount of data.

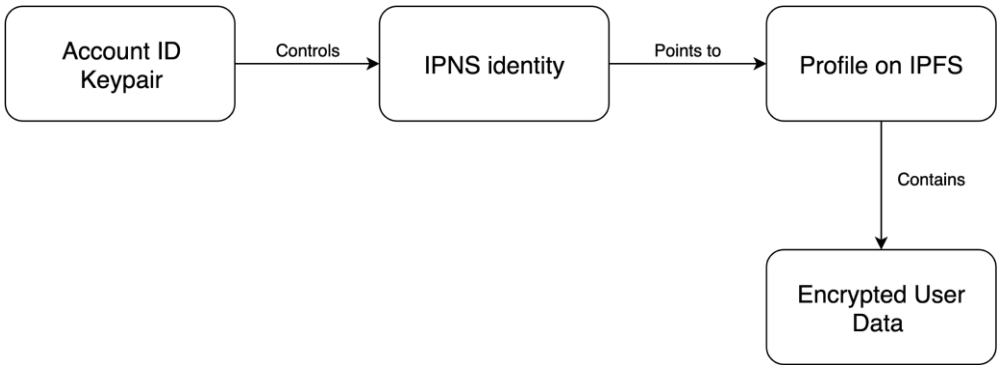

Figure 16: Profile architecture

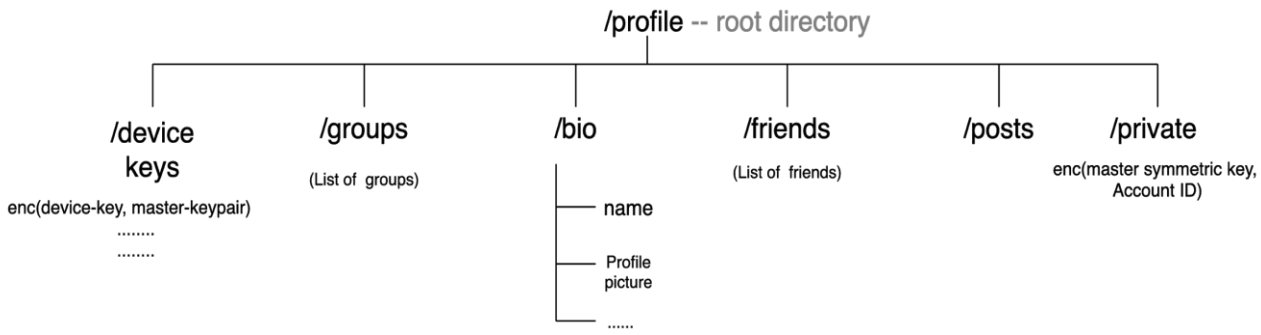

Figure 17: Profile description

Ed25519 keys were used for identity in this design instead of RSA because of the following reasons:

1. They are smaller than RSA key pairs for the same level of security which means, less data to store and smaller payload to send over the network.
2. Elliptic Curve Cryptography is also faster than the RSA algorithm, especially on private key operations, which means less processing power.

3.1.2 Adding Friends

Adding someone as a friend is synonymous with establishing a *shared secret* between the two accounts, which in this design, is a symmetric key. Once a shared secret has been established between two accounts, users can securely exchange further messages.
In this system, shared secrets are used for encrypting posts, messages and even metadata meant for different audiences. Users can publish data on their profiles only for a specific audience by encrypting it with a shared symmetric key known only by the members of that group.

In the subsequent subsections, the handshake process used in this system to establish a shared secret will be discussed in detail. The following handshake is greatly inspired by Scuttlebutt's Capability-based Handshake [40] and is designed to provide the following security features:

- After a successful handshake the peers have verified each other's public keys.
- The handshake produces a shared secret.
- The client must know the server's public key before connecting. The server learns the client's public key during the handshake.

- Once the client has proven their identity the server can decide they don't want to talk to this client and disconnect without confirming their own identity.

- A man-in-the-middle cannot learn the public key of either peer.

- Handshakes provide **forward secrecy**. Recording a user's network traffic and then later stealing their secret key will not allow an attacker to decrypt their past handshakes.

3.1.2.1 Handshake

A handshake process starts when the Requester sends a Friend Request to the Responder. Before a user (Requester) can send a friend request to another user (Responder), it needs to know responder's Account ID public key. Once, the requester has acquired the Account ID public key, the following 5-step handshake can begin:

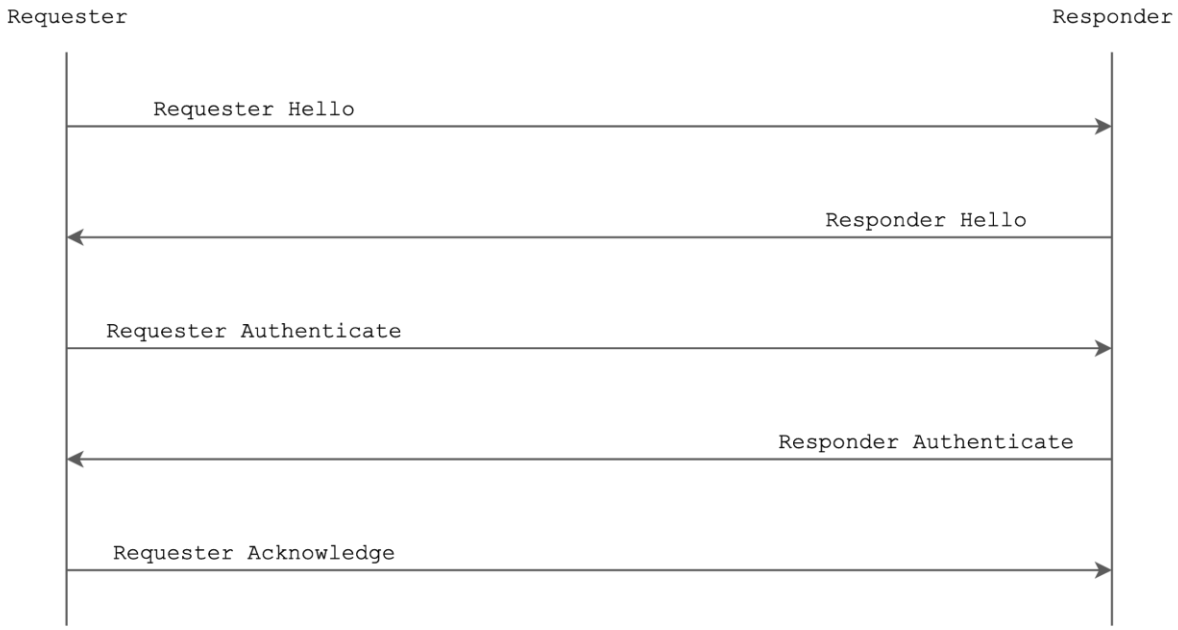

Figure 18: Handshake

Starting Keys

Upon starting the handshaking process, the requester and the responder know the following keys:

Requester (A) knows:
Requester's Account ID key pair: $\mathbf{A_S}$ and $\mathbf{A_P}$
Requester's ephemeral key pair: $\mathbf{a_S}$ and $\mathbf{a_P}$
Responder's Account ID public key: $\mathbf{B_P}$

Responder (B) knows:
Responder's Account ID key pair: $\mathbf{B_S}$ and $\mathbf{B_P}$
Responder's ephemeral key pair: $\mathbf{b_S}$ and $\mathbf{b_P}$

Step 1. Requester Hello

First the Requester sends their ephemeral public key $\mathbf{a_P}$ to the Responder. Ephemeral keys are only used for a single handshake and then discarded. This ensures the freshness of the messages to avoid replay attacks.

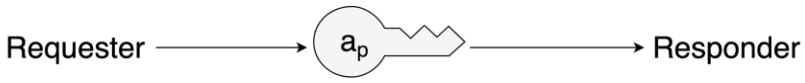

Figure 19: Requester Hello

Step 2. Responder Hello

The Responder replies with their own ephemeral public key $\mathbf{b_P}$ to the Requester.

Figure 20: Responder Hello

After, exchanging the ephemeral keys, both ends use them to derive a shared secret **a.b** using scalar multiplication. Both the parties combine their ephemeral secret key with the other's ephemeral public key to derive the same shared secret. Since an attacker doesn't know either of the secret ephemeral keys, therefore can't generate the shared secret. A man-in-the-middle can still intercept the ephemeral keys and exchange them. This means that the system cannot rely solely on **a.b**, and another derived secret **a.B** must also be used in the exchanges that follow.

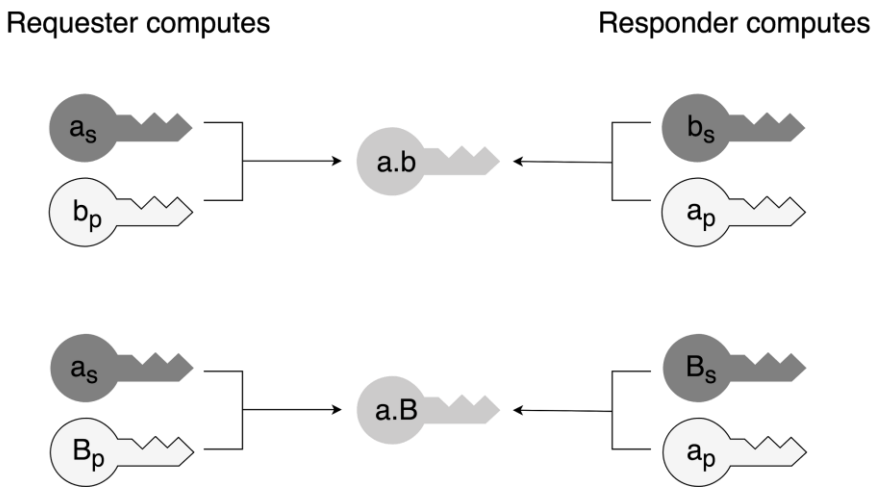

Figure 21: Shared key derivation

Step 3. Requester Authenticate

The Requester reveals their identity to the Responder by sending their Account ID public key along with a signature using their Account ID secret key. By signing the shared key derived in the previous step of the handshake, the requester successfully proves their identity.

The Requester's message is enclosed in a secret box which is sealed with **a.b** and **a.B** to ensure that only the Responder can read it.

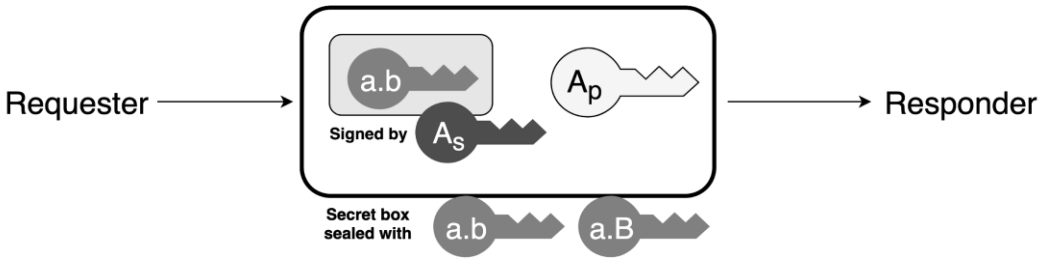

Figure 22: Requester Authenticate

At this stage, the Responder also knows the Requester's Account ID public key, therefore another shared secret **A.B** is derived by both parties. From this step, a man-in-the-middle is no longer able to intercept the exchanges regardless of what they have done in the previous steps.

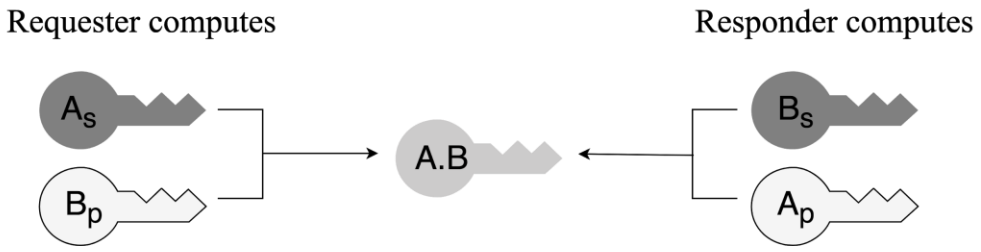

Figure 23:Second key derivation

Step 4. Responder Accept

The Responder accepts the handshake by signing the **a.b** with their Account ID secret key. The Responder's signature is sealed in a secret box using the **a.b** and **A.B** shared secrets, which proves that the Responder has effectively received and decrypted the message in the previous step.

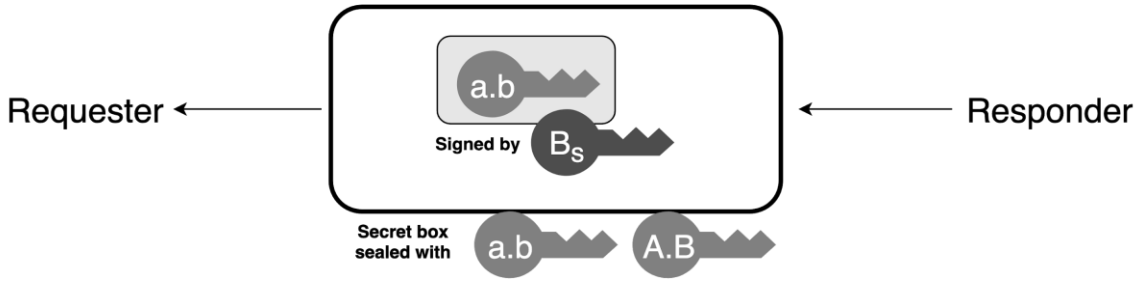

Figure 24: Responder Accept

Step 5. Requester Acknowledge

The Requester then sends an acknowledgement message to the Responder, proving that they successfully received the previous message. After this the handshake is considered successful. Both the parties have proven their identities to each other and established a shared secret.

Figure 25: Requester Acknowledge

3.1.3 Groups

A group is essentially a group of accounts exchanging messages and metadata. The metadata can be of various types, for example to inform the group of a newly added member, to exchange encryption keys between members and so on. When a group is created, a *Group ID* key pair is generated, which is an asymmetric keypair and uniquely identifies the group. Once generated, the *Group ID* key pair cannot be changed. Each group also has a randomly generated *group-secret*, which is a shared symmetric key shared among all the group members. All the data published in a group is encrypted under the most recent *group-secret* and therefore can only be decrypted by the group members.

Every time a new group member is added, or an existing group member is removed, a new *group-secret* must be derived and shared with all the group members. This ensures forward and backward secrecy in the system since a removed group member can't decrypt the new messages and a newly added member can't decrypt the messages posted before they joined a group.

For the groups to work, it is essential that all the group members have the same copy of the metadata and messages posted in the group. This poses a serious challenge in a decentralized system since there is no central server to store and maintain a consistent state among all the peers. This design solves this problem by maintaining two immutable logs provided by OrbitDB - *ipfs-logs*. These logs are responsible for exchanging metadata and messages among group members.

3.1.3.1 Group Structure

A Group is made up of two logs - a *message log* and a *metadata log*. [41]

Message log: This log contains all the messages and other payloads exchanged within the group. Group members can choose to download only a part of this log if they want to. Although group members cannot decrypt any messages posted before they joined the group because of the backward secrecy design of this system.

Metadata log: This log is responsible for exchanging all the metadata among the group members. Since essential metadata such as new shared secrets and arrival of new members is published in this log, if a member doesn't download the entire metadata log, they might not be able to see the full list of group members or miss a newly derived *group-secret*. For these reasons, all group members must download the entire metadata log.

3.1.3.2 Types of Groups

To allow users to interact with other users directly or in a group, two types of groups are used in this system: *Multi-member Groups* and *Two-member Groups*.

1. **Multi-member Groups**: These types of groups can contain any number of members who can or cannot be friends with each other. When a user creates a multi-member group, the *Group ID keypair* and the *group-secret* are randomly generated. Once the secrets have been generated, the group creator posts an INIT message in the metadata log. The INIT message consists of a secret box sealed with the *group-secret* and contains the *Account ID* public key of the creator member along with a signature of *Account ID* public key of the creator member by the *Group ID* private key.

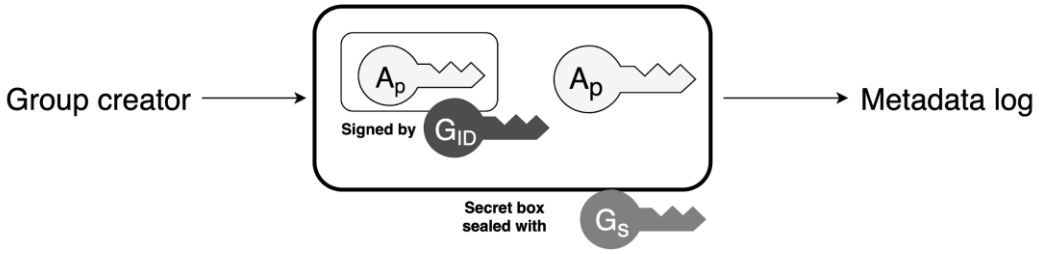

Figure 26: Multi-member groups

After the INIT message is posted in the metadata log, the group creator discards the *Group ID* secret key which ensures that all of the group members have equal rights and capabilities.

2. **Two-member Groups**: These groups consist of exactly two users who are friends. When two accounts participate in the handshaking process to add each other as a friend, a Two-member group is automatically created. These groups are different from the multi-member groups as they don't have a *Group ID* keypair since it is between two friend accounts. The *group-secret* of a two-member group is the same shared secret that is derived during the handshaking process.

3.1.2.3 Joining a group

To join a multi-member group, a user must have an invitation from one of the group members. An invitation contains the *Group ID* and the *group-secret.* Once a user possesses an invitation, they can download the entire metadata log of the group and get the list of group members. After the metadata log has been successfully downloaded, the new member announces their arrival with an arrival post in the metadata log.

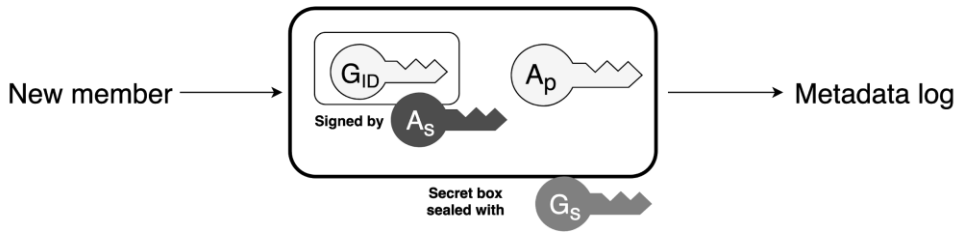

Figure 27: Joining a group

Figure 27 shows the format of an arrival post by a new member (A) in the metadata log. An arrival post consists of a secret box sealed with the *group-secret* and contains the Account ID public key of the new member along with a signature of the *Group ID* by the Account ID private key of the new member.

3.1.4 Posts

Posts are essentially a list of messages posted by an account in the `./posts` folder of their profile. When a user *posts* some content, a post object encrypted with a shared symmetric key is published in the `./posts` folder of their profile and is decryptable only by the accounts that are established as friends of that user. The posts objects are also signed by the account's *Account ID* secret key which verifies the post's validity.

Posts are fundamentally different from group messaging as they are not replicated on all friends' accounts and are only stored in the profile that publishes that post. One other distinction between a post and a group message is that posts can only be seen by users that are friends with the author's account whereas a message posted on a multi member group can be seen by all the group members who don't necessarily have to be friends with the author of the message.

Figure 26 shows the format of a *Post* object which is greatly inspired by Scuttlebutt's *Feeds* [42]. A post object consists of two separately encrypted parts - the metadata and the content. This feature allows friend accounts to only download the metadata which is necessary to make sense of their friends' profile without having to download large payloads like images or videos.

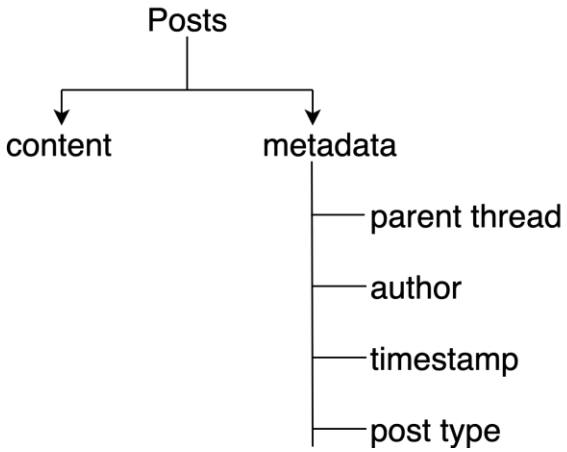

Figure 28: Post format

The metadata associated with a post includes *parent thread, author, timestamp* and *post type. Parent thread* is an optional field, which stores the hash of another Post object that the post is responding to, otherwise, it is left empty. This allows users to start a *chain* of posts by referring to a parent post. The UI can then group together all these messages together in the same *thread.*
Author contains the public key of the account who is publishing that particular post. This allows other users to identify the author of a post.
Timestamp stores time when the message was posted. This enables sequencing of post objects so that friends can see the newest posts first.
A *post type* field allows users to post different kinds of posts such as posts, reacts, read-receipt, etc. All these types of posts are treated the same in our design, the UI can then differentiate between different types and treat them differently.

3.1.5 Discovery

A major component of any social network is the ability to discover other users on the platform. This however poses a major design challenge in a decentralized system like this as there is no central server that contains the account details for all the users. This system offers two ways for peers to discover each other:

- **Discovery log**: This is essentially a log containing *Account ID* public keys for users to find each other. This log uses an immutable *ipfs-log* by orbitDB, which is the same data structure used by the groups to exchange and replicate data in this design. Since, an *ipfs-log* is an append only log, users can only append their entry in this log and once it is

appended, it cannot be removed by anyone. Each entry contains a user's *Account ID* public key along with a *unique username* which allows users to find each other easily.
Since, these logs implement CRDTs to maintain a consistent state by replicating data among peers, each user can store their own copy of it which keeps getting updated as more users publish their Account IDs.

- **Invite code:** Invite codes can also be shared by peers through other messaging or social media platforms. An invite code contains the *Account ID* public key of a user. Which means that anyone who possesses an invite code can send a friend request to the account that generated that particular invite code.

3.1.6 Multi device support

Users often use multiple devices to access their social network identities, therefore the system must enable access to a user account via multiple devices. However managing long random strings of public/private keypairs is not feasible for most of the users. A combination of username and passphrase is more manageable and familiar for most users and will allow for a smooth transition from current social network platforms.

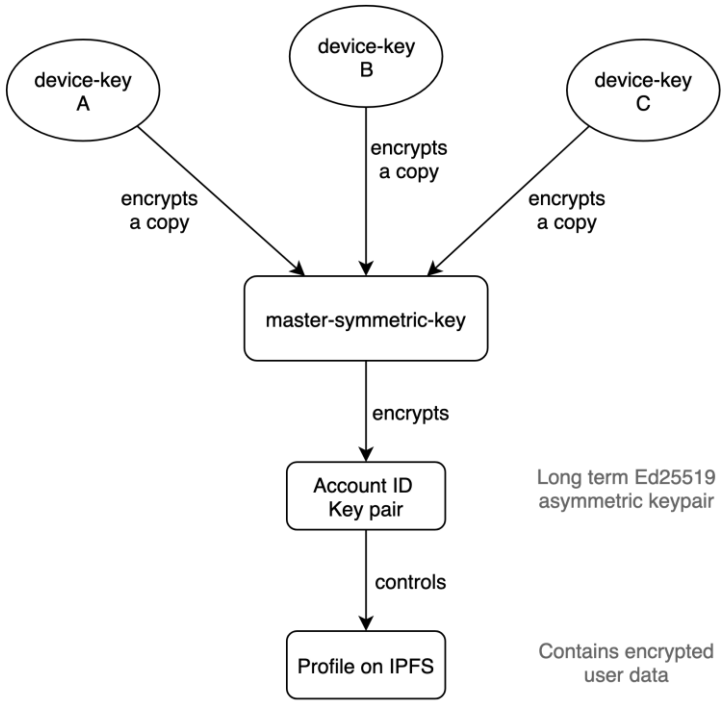

Figure 29: Multi-device support

When an account is first created, a combination of *Username* and *Password* is chosen by the creator of the account. This password salted with the username is passed through the **Scrypt** key derivation function to derive a symmetric key which is referred to as *master-symmetric key* in this design. A random Ed25519 key pair (*Account ID*) is also automatically generated which serves as the permanent identity of an account. Once the keys are generated, the Account ID keypair is encrypted with the *master-symmetric key* and is stored in the user's profile under `./private`. The operation of key generation for the two aforementioned keys i.e. Account ID key pair and the master-symmetric key, only takes place once and is not repeated. Finally, the master-symmetric key is encrypted with a randomly generated *device key* and is stored in the `./device-keys` folder in the user profile. This operation is repeated on every new device and a device key provides a unique identity to each device.

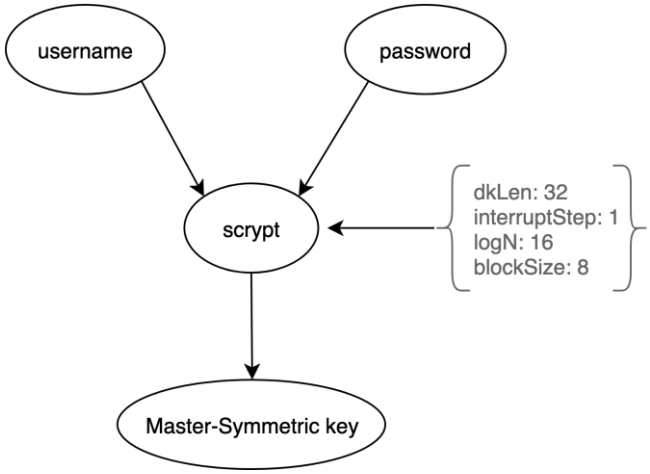

Figure 30: Scrypt master-symmetric key derivation

In order to access their account from a new device, a user must enter their Username-Password combination. This information is passed again through the Scrypt key derivation function to derive the master-symmetric key for the account which can then be used to read the encrypted Device ID keypair. Another copy of the account's master-symmetric key is encrypted with a newly generated device key and stored in the `./device-keys` folder as described above for providing access for the current device in future. After this a user is logged in.

By virtue of being decentralized, the number of attempts an attacker has to crack the password cannot be limited, hence it is imperative to choose a strong password. A password of at least 14 random alphanumeric characters is recommended. Scrypt also offers an extra layer of protection since it is a memory-hard function, which makes it much more expensive to attack the system using custom hardware.

3.2 Implementation

This section provides an overview of how the design stated in the previous section was implemented. The development tools and technologies used in the development of the application will also be explained.

3.2.1 Application Architecture

Figure 31 shows the application architecture and the interaction of individual components in the design. The user interacts with the application via a user interface in the browser which is a static file served over HTTP. This interface interacts with the locally running Node.js application which handles all of the interface's requests. Node.js has an IPFS node running locally using js-ipfs. This IPFS instance is also passed to the orbit-db constructor which exposes the orbit-db api. This IPFS node interacts and is responsible for interacting with the rest of the IPFS network.

As most of the components of the application are running locally, it makes the whole system more secure and eliminates the need to trust a website or a server for the users which are susceptible to being hacked or breached.

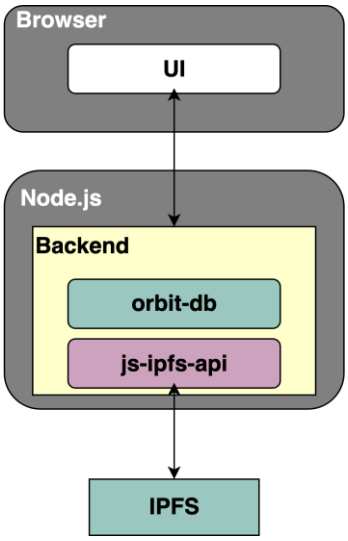

Figure 31: Application architecture

3.2.1.1 Node.js

Node.js [43] is a cross-platform JavaScript run-time environment which executes server side. It is lightweight and efficient to run. It also has extensive documentation and a large package ecosystem, named npm, which makes development much simpler and efficient. Node.js supports a wide range of platforms which makes it a great choice for this design as this application must support a wide range of systems.

Node.js was used in this application to import the js-ipfs library and spawn a local IPFS node. A host range of libraries required to build the application were used via npm. For example, `Scrypt-async` package was used to implement Scrypt key derivation function, `tweetnacl` was used as the main cryptography library.

3.2.1.2 JS IPFS

Js-ipfs [44] is the JavaScript implementation of IPFS which is capable of running in a Browser, a Service Worker, a Web Extension and Node.js. In this application, js-ipfs is used in Node.js to start an IPFS node locally.

JS IPFS has built-in support for a wide range of sub packages including PubSub, IPNS, DAG, FILES, multihashing, etc. which were essential in building this application. Although a lot of development has been done on js-ipfs already, the project is still in Alpha which makes some things break sometimes.

3.2.1.3 Libp2p

Libp2p [45] is a modular system of protocols, specifications and libraries that enable the development of peer-to-peer network applications. It provides many components required to create a peer-to-peer application, some of them are: DHT, PubSub, Nat Traversal, a collection of transports, etc.

More specifically, the `Peer-id` package provided by libp2p was used to create identities for each account.

3.2.2 Application Interface

A simple user interface was developed using HTML to show the basic functionality of the application. A static page is served to the browser over HTTP and the source code is bundled using browserify so that the web page can use it. The application screen is divided into multiple parts and will be described below.

Figure 32: Application interface

The first part of the UI describes the basic information about an account, namely Node information and the Account ID keypair of the account. It also shows the latest profile hash of the account which changes anytime the profile is changed. There is also a field where the user can look-up the profile of another account by entering its Account ID public key.

Fig 33 shows the final section of the homepage which lists all the directories stored in the profile. The type field in the metadata in the below image differentiates between a file and a folder. Type 1 represents folders while type 0 represents files.

Path: / load


```
{
  "bio": {
    "hash": "QmddsgZcysEhEoEkdUA7HetdLPN7dSCy76Z3WaVtMerc1D",
    "name": "bio",
    "type": 1,
    "size": 0,
    "contents": {
      "public.json": {
        "hash": "QmcFYhjiMEoEuPRSjKqHWkKGmy5kB38G4qvJfUDtqNjM8i",
        "name": "public.json",
        "type": 0,
        "size": 82
      },
      "salt": {
        "hash": "Qma2eG3LLgFAAQVL6DqNNzYtAbA87RZeWvsNMQ8Va1upTZ",
        "name": "salt",
        "type": 0,
        "size": 24
      }
    }
  },
  "device-keys": {
    "hash": "QmbBTeZzwdrkWB8H2Z1o7SDuqEiFu9h5PgjGH9NJzVdbwJ",
    "name": "device-keys",
    "type": 1,
    "size": 0,
    "contents": {
      "87NLhJj5GhbBPg": {
        "hash": "QmaAQhtmNVAb8DoXNJVQq8APmwbKdf4Pq623BaEKzp4QCP",
        "name": "87NLhJj5GhbBPg",
        "type": 0,
        "size": 83
      },
      "info.json.enc": {
        "hash": "QmcFHSCPJDvsSMbhYVbncbJJjkD41VE8DWo2LHVvxiVD5d",
        "name": "info.json.enc",
        "type": 0,
        "size": 70
      }
    }
  },
  "private": {
    "hash": "QmZDmBxBL13NWv2Xy1m9zv316xoK7NGRZpTR543NbSLRva",
    "name": "private",
    "type": 1,
    "size": 0,
    "contents": {
      "key": {
        "hash": "QmTcMovQDLBg25NTD68AGeYZaoLAyGPuPniAM5Ma5fnoGp",
        "name": "key",
        "type": 0,
        "size": 174
      }
    }
  },
  "version.txt": {
    "hash": "QmbbJqnmggZavEvEiY3cMK8JYmi78b5YADumSMUgHaV19D",
    "name": "version.txt",
    "type": 0,
    "size": 5
  }
}
```


Figure 33: Homepage listing directories stored in the profile

4. Conclusions

The following chapter will conclude this dissertation by refiling on the goals achieved by this project. First the application developed for this dissertation and its potential use cases will be discussed. Next, some of the difficulties that were faced while developing this application will be discussed followed by the limitations of the design developed in this project. Finally, this dissertation closes with a section on future work where possible improvements and future development plans are discussed.

Revisiting the original goals for this project, which was to develop a decentralized social network on IPFS which preserves privacy and anonymity of its users, enables users to have total control over who has access to their private data, is resilient against censorship and mass surveillance, provides access from multiple devices and still provides all the features of current social networks, e.g. messaging, posts, groups, profile, comments, etc.

As the design developed by this dissertation fulfills all of the goals for this project, this project can be deemed a success. Many novel solutions were developed to deal with the challenges that come with decentralization. The application developed, successfully implemented much of the basic functionality of a social network, more development will be carried out in future to implement more advanced features.

4.1 Use Cases

This section will discuss some of the use cases for the system developed in this dissertation. Since the social network application developed is fully decentralized and therefore doesn’t require any central servers, it can be used in regions of the world where social networking sites like Facebook have been banned by the government or where freedom of speech is restricted.

Due to the centralized nature of the current social networking platforms like Facebook and twitter, they are susceptible to user censorship, single-point failure and misuse of private user data. The solution designed in this dissertation can provide a better and more secure alternative to these platforms while still keeping the user experience very similar.

Although a decentralized social network can provide security and privacy to users, it can also enable bad actors to use it to facilitate crime and illegal activities without any risk of government surveillance. Since a user can create unlimited identities on such a decentralized social platform, there is no way to know if an account belongs to the person they are representing.

4.2 Difficulties

Although the work done for this dissertation was very rewarding in terms of understanding of new concepts and working with cutting edge technology, there were some difficulties faced. As IPFS and most of the decentralized technology is relatively new and is in constant development the API changes sometimes didn't work with the older written code and had to be rewritten using the latest API versions.

Many of the documentation which turned out to be very important for this project wasn't even published when the research was begun which led to some last-minute design changes. Lack of online guides and limited discussions on the topic by the developer community meant that some of the development issues didn't have clear answers anywhere.

With no prior development experience with Node.js and limited background in cryptography, there was a steep learning curve at the beginning but with the help of many helpful online guides by the community and many great packages in npm, this issue was resolved.

4.3 Limitations

This section will discuss in detail, some of the limitations in this design.

The biggest limitation with the design developed in this project is *persistence* of user data. As the design relies on IPFS which is a peer-to-peer network, users are responsible for making sure their data is hosted somewhere. If a user is offline, none of their friends can view data in their profiles. To solve this problem, users can host their data using a decentralized storage service like filecoin. [46]

Another limitation is *key recovery* in case a user loses their Account ID keypair. Only the owner of an account knows the Account ID private key. When logging from a new device, a user retrieves the Account ID keypair by entering their Username and password. Given the users use strong passwords, it is almost impossible for a computationally bounded adversary to crack their login details. In a case where a user forgets their username or password, there is no way to retrieve these login credentials as they are not stored anywhere. Therefore, the only possible choice is to create a new account with new Account ID keys.

Current group invitations mechanism also presents a design limitation. Since there is no expiration time for a group invitation due to its decentralized nature, invitations can be used any number of times. And if shared irresponsibly by group members can allow anyone to join a group. Although such a member will still not be able to see older messages due to backward secrecy in this design.

4.4 Future Work

Decentralized identifiers (DIDs) are a new type of identifier that enables verifiable, decentralized digital identity. A DID identifies any subject (e.g., a person, organization, thing, data model, abstract entity, etc.) that the controller of the DID decides that it identifies. Unlike typical federated identifiers, DIDs have been designed so that they may be decoupled from centralized registries, identity providers, and certificate authorities.

IPID is an implementation of the DIDs specification over the IPFS network using the IPNS cryptographic namespace resolution service. IPIDs can be used to provide a verifiable identity to use accounts on a decentralized social network.

As mentioned previously IPFS and most of other decentralized technologies are in a constant development state. As these technologies mature and get better, decentralized social networks will eventually replace the present centralized social networks. The development on this project will continue to develop a more full-fledged application with all the features mentioned in this report.

Bibliography

[1] T. Guardian, "Boundless Informant: the NSA's secret tool to track global surveillance data," [Online]. Available: https://www.theguardian.com/world/2013/jun/08/nsa-boundless-informant-global-datamining.

[2] T. Guardian, "NSA tapped German Chancellery for decades, WikiLeaks claims," [Online]. Available: https://www.theguardian.com/us-news/2015/jul/08/nsa-tapped-german-chancellery-decades-wikileaks-claims-merkel.

[3] T. Guardian, "Facebook says Cambridge Analytica may have gained 37m more users' data," [Online]. Available: https://www.theguardian.com/technology/2018/apr/04/facebook-cambridge-analytica-user-data-latest-more-than-thought.

[4] P. S. &. D. M. Turner, "Symmetric Key Encryption - why, where and how it’s used in banking," [Online]. Available: https://www.cryptomathic.com/news-events/blog/symmetric-key-encryption-why-where-and-how-its-used-in-banking#:~:text=Symmetric%20encryption%20is%20a%20type,used%20in%20the%20decryption%20process..

[5] "What Is Symmetric Key Cryptography?," [Online]. Available: https://academy.binance.com/security/what-is-symmetric-key-cryptography.

[6] S. Information, "Symmetric vs. Asymmetric Encryption – What are differences?," [Online]. Available: https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences.

[7] A. N. P. B. Alexander Katz, "RSA Encryption," [Online]. Available: https://brilliant.org/wiki/rsa-encryption/.

[8] geeksforgeeks, "RSA Algorithm in Cryptography," [Online]. Available: https://www.geeksforgeeks.org/rsa-algorithm-cryptography/.

[9] E. Paul, "What is Digital Signature- How it works, Benefits, Objectives, Concept," [Online]. Available: https://www.emptrust.com/blog/benefits-of-using-digital-signatures.

[10] A. Chambers, "Elliptic cryptography," [Online]. Available: https://plus.maths.org/content/elliptic-cryptography.

[11] D. J. Bernstein, "Curve25519: new Diffie-Hellman speed records," [Online]. Available: https://cr.yp.to/ecdh/curve25519-20060209.pdf.

[12] J. LAKE, "What is the Diffie–Hellman key exchange and how does it work?," [Online]. Available: https://www.comparitech.com/blog/information-security/diffie-hellman-key-exchange/.

[13] C. Neagu, "What are P2P (peer-to-peer) networks and what are they used for?," [Online]. Available: https://www.digitalcitizen.life/what-is-p2p-peer-to-peer.

[14] R. Schollmeier, "A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications," in *Proceedings First International Conference on Peer-to-Peer Computing*, Linkoping, Sweden, Sweden.

[15] A. P. J. H. M. N. Dilum Bandara, "Collaborative Applications over Peer-to-Peer Systems - Challenges and Solutions," [Online]. Available: https://arxiv.org/abs/1207.0790.

[16] D. Barkai, Peer-to-Peer Computing: Technologies for Sharing and Collaborating on the Net, Intel Press, October 2001.

[17] R. R. Daniel Stutzbach, "Improving Lookup Performance over a Widely-Deployed DHT," *University of Oregon.*

[18] R. M. D. K. M. F. K. H. B. Ion Stoica, "Chord: A Scalable Peer-to-peer Lookup Service for Internet".

[19] "What is a distributed hash table?," [Online]. Available: https://www.educative.io/edpresso/what-is-a-distributed-hash-table.

[20] V. TRƯƠNG, "Testing implementations of Distributed Hash Tables," [Online]. Available: https://core.ac.uk/download/pdf/16312626.pdf.

[21] "A branch hash function as a method of message synchronization in anonymous P2P conversations," 02 July 2016. [Online]. Available: https://content.sciendo.com/view/journals/amcs/26/2/article-p479.xml.

[22] J. A.-F. R. B. H. Sreekanth Malladi, "On Preventing Replay Attacks on Security Protocols".

[23] A. Greenberg, "Hacker Lexicon: What Is End-to-End Encryption?," [Online]. Available: https://www.wired.com/2014/11/hacker-lexicon-end-to-end-encryption/.

[24] N. Perlroth, "What Is End-to-End Encryption? Another Bull's-Eye on Big Tech," [Online]. Available: https://www.nytimes.com/2019/11/19/technology/end-to-end-encryption.html.

[25] M. Tillman, "WhatsApp end-to-end encryption: What does it mean?," [Online]. Available: https://www.pocket-lint.com/apps/news/whatsapp/137221-whatsapp-rolls-out-end-to-end-encryption-what-does-that-mean.

[26] Y. G. Micah Lee, "ZOOM MEETINGS AREN'T END-TO-END ENCRYPTED, DESPITE MISLEADING MARKETING," [Online]. Available: https://theintercept.com/2020/03/31/zoom-meeting-encryption/.

[27] B. Jovanović, "What is end-to-end encryption?," [Online]. Available: https://dataprot.net/articles/what-is-end-to-end-encryption/.

[28] N. P. C. B. M. Z. Marc Shapiro, "Conflict-free Replicated Data Types," [Online]. Available: https://pages.lip6.fr/Marc.Shapiro/papers/RR-7687.pdf.

[29] "Operation-based CRDTs," [Online]. Available: https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type#Operation-based_CRDTs.

[30] H. M. C. Paulino, "Conflict-Free Replicated Data Types in Dynamic Environments," [Online]. Available: https://run.unl.pt/bitstream/10362/93770/1/Barreto_2019.pdf.

[31] IPFS, "How IPFS works," [Online]. Available: https://docs.ipfs.io/concepts/how-ipfs-works/.

[32] "InterPlanetary Name System (IPNS)," [Online]. Available: https://docs.ipfs.io/concepts/ipns/#example-ipns-setup.

[33] "About OrbitDB," [Online]. Available: https://orbitdb.org/about/#:~:text=OrbitDB%20uses%20IPFS%20as%20its,and%20offline%2Dfirst%20web%20applications..

[34] orbit-db, "Getting Started with OrbitDB," [Online]. Available: https://github.com/orbitdb/orbit-db/blob/master/GUIDE.md.

[35] R. L. M. R. H. Péter Huba, "What is OrbitDB?," [Online]. Available: https://github.com/orbitdb/field-manual/blob/master/00_Introduction/02_What_is_OrbitDB.md.

[36] "ipfs-log," [Online]. Available: https://github.com/orbitdb/ipfs-log.

[37] C. Percival, "The scrypt Password-Based Key Derivation Function," [Online]. Available: https://scuttlebot.io/more/protocols/shs.pdf.

[38] C. Percival, "The scrypt Password-Based Key Derivation Function," [Online]. Available: https://tools.ietf.org/html/rfc7914#page-2.

[39] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification," [Online]. Available: https://tools.ietf.org/html/rfc2898.

[40] D. Tarr, "Designing a Secret Handshake: Authenticated," [Online]. Available: https://scuttlebot.io/more/protocols/shs.pdf.

[41] "Berty protocol," [Online]. Available: https://berty.tech/docs/protocol/.

[42] Scuttlebutt, "Scuttlebutt Protocol Guide," [Online]. Available: https://ssbc.github.io/scuttlebutt-protocol-guide/?ref=hackernoon.com#feeds.

[43] [Online]. Available: https://nodejs.org/en/.

[44] "API resources for js-ipfs," [Online]. Available: https://docs.ipfs.io/reference/js/api/.

[45] " libp2p documentation," [Online]. Available: https://docs.libp2p.io/.

[46] Filecoin. [Online]. Available: https://filecoin.io/.

[47] K. Team, "Guide: What is Directed Acyclic Graph?," [Online]. Available: https://medium.com/kriptapp/guide-what-is-directed-acyclic-graph-364c04662609.

[48] ProtoSchool, "Merkle trees and directed acyclic graphs (DAG)," [Online]. Available: https://proto.school/data-structures/05/.

[49] "Distributed Hash Tables," [Online]. Available: https://www.cs.cmu.edu/~dga/15-744/S07/lectures/16-dht.pdf.

[50] "Top 10 Most Common Types of Cyber Attacks," [Online]. Available: https://blog.netwrix.com/2018/05/15/top-10-most-common-types-of-cyber-attacks/.

[51] libp2p, "PUBLISH/SUBSCRIBE," [Online]. Available: https://docs.libp2p.io/concepts/publish-subscribe/.

[52] IPFS, "Merkle Distributed Acyclic Graphs (DAGs)," [Online]. Available: https://docs.ipfs.io/concepts/merkle-dag/.