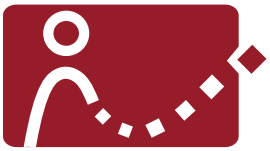
Hybrid Online Social Networks



Masterarbeit von Carsten Porth
Tag der Einreichung: 25. März 2019

1. Gutachter: Prof. Dr. Max Mühlhäuser
2. Gutachter: Dr. Jörg Daubert, Aidmar Wainakh

Darmstadt, den 25. März 2019

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Telekooperation
Prof. Dr. Max Mühlhäuser

Erklärung zur Abschlussarbeit gemäß § 23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Carsten Porth, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Masterarbeit stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Bei einer Masterarbeit des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, den 25. März 2019

Unterschrift

Abstract

Zusammenfassung

Contents

1 Introduction

1.1 Motivation

1.2 Challenge

1.3 Research Question

1.4 Outline

2 Background

2.1 Software System Architecture

The software system architecture describes the relationships and properties of individual software components. It is a model that describes a software on a high-level design. The structure of an architecture can be represented mathematically as a graph, with the nodes representing the individual software components and the edges their relationships to each other. Although the individual components can be executed on the same computer, they are usually interconnected via networks. In general, a distinction is made between centralized, decentralized and distributed architectures as shown in Figure 2.1.

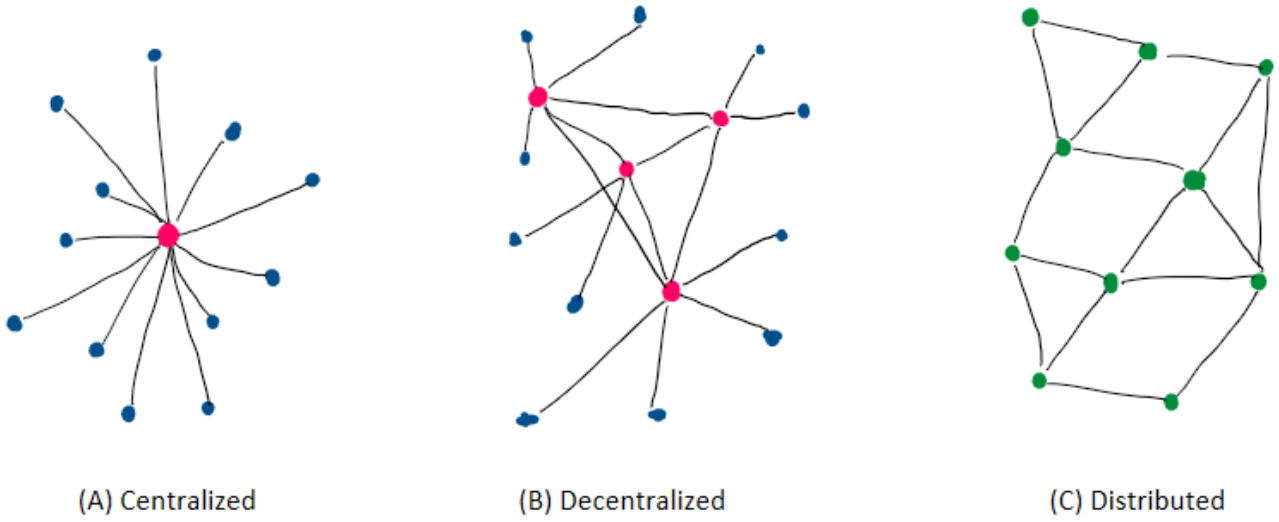


Figure 2.1: Schematic representation of various software system architectures. In centralized applications (A), all clients (blue) are in connection with a central server (red), while in decentralized applications (B) several servers provide for improved stability. In distributed applications (C), all nodes have the same role with the same tasks, which is why we also speak of peers (green).

In the following, the characteristics and peculiarities of the different architectures are described in detail.

2.1.1 Centralized Applications

In a centralized application, the software essentially runs on a central node with a static address with which all other nodes communicate. This central node is called a server and the nodes connected to it are called clients. The clients use the services of the server.

Typically, most web applications are designed as centralized applications. The clients are usually (mobile) apps or browsers that act as the interface to the user. Examples of this are social networks such as Facebook.

The advantages of such an architecture include that new clients can easily join the system simply by connecting to the server. Also, the maintenance of the software is easy to perform, since only the server node needs to be updated. Such structures are also advantageous for the speed and the associated user experience since servers are generally reachable via fast connections, have sufficient resources and do not need to use complicated routes across several nodes. Due to the special role of the server, the authentication of a client in the system is easy to perform. Only the server has to check whether the client only performs the actions to which it is authorized and can intervene if necessary. If a client is offline, this has no negative impact on the architecture. Because the software essentially runs on the server, clients only need to have low resources.

The biggest disadvantage with this architecture is that with the failure or the blockade of the server, the entire system collapses. So, the server is the single point of failure of the whole system. Due to the design, the server has all the data. This makes him not only a popular target for hackers but also brings the clients into total dependency on the server. Having access to all the data can be used to provide an improved user experience by adapting the software to the user's needs. But it can also be used for targeted advertising or sensible data can even get sold to third parties. Furthermore, the server decides which data to send to the client, which brings with it the danger of censorship. As the number of clients increases, the server must scale to have sufficient resources.

2.1.2 Decentralized Applications

What are the advantages and disadvantages of centralized applications, is largely reversed in decentralized systems. Unlike centralized applications, decentralized applications do not have a single point of failure. But there is no node in the system that has all the data which makes accessing information sometimes hard. There are any number of nodes that perform the tasks of a server and by exchanging with other servers in total form a complete system.

Thus, the advantages of decentralized applications include, in addition to the already mentioned increased reliability, even further advantages. Each server only records the number of users covered by its resources. New servers contribute new resources to the overall system. Distributed applications cannot be disabled easily because a new server can be started at any time that does not fall under a lock. Because the data are not centrally available, they cannot be processed, so that user data are better protected. Also, there is no longer a prominent attack target. So, hacking a single server loses lucrativeness.

The drawbacks include the difficulty of finding data because they are spread across multiple servers. Search functionalities are thus difficult to implement. If a server is taken offline, the data is no longer available, even if the system itself remains functional. Since there is no longer a central point that is managed by an operator, rolling out updates is difficult. This raises the challenge that server nodes of different versions can still work together.

Examples of decentralized applications are the social networks Diaspora and Mastodon. Each user can operate his own server in these networks. Unlike Facebook, the decision on the existence of the service is therefore not in the control of the operator, but the user. Bitcoin is also a decentralized application architecture. In doing so, transactions are carried out in a decentralized database (blockchain).

2.1.3 Distributed Applications

A feature of distributed applications is the distributed computation across all nodes. At the same time, a single task is executed in parallel on several nodes of a system, and the entire system delivers the result in total.

In distributed applications, there is no hierarchy with servers or clients. Each node is equal to the other nodes with everyone performing the same tasks. For this reason, a node in this architecture is called a peer. The structure is then referred to as peer to peer (P2P). With such structures, there are no scaling problems, since each node contributes the required resources itself. Thus, the resources of the entire system grow with each new node added.

BitTorrent is an application that belongs to this architecture type. It solves the problem of downloading large files efficiently. When downloading a file, it is offered by several nodes so that different pieces can be loaded in parallel from different sources in difference to HTTP where only one source provides the file. Each node not only downloads but also provides itself to other files. In order to avoid that nodes download only (so-called Leechers) but also contribute, they are penalized with slow download speeds. This principle is known as *tit for tat*.

2.1.4 Comparison

The following table 2.1 compares the main features of the different architectures as described before.

	Centralized Systems	Decentralized Systems	Distributed Systems
Scalability		+	++
Maintenance	++	+	
System Stability		+	++
Performance	++	+	
Data Availability	++	+	

Table 2.1: Comparison of different software system architectures on scalability, maintenance, system stability, performance, and data availability. The pluses indicate how positive something is relative to the other systems.

2.2 Peer-to-Peer Networks

The distinctive feature of peer to peer (P2P) systems is that each participant has the role of both a server and a client. The participants are therefore equal with each other and provide each other with services, which is reflected in the naming. P2P networks are usually characterized as overlay networks over the Internet. Concerning the structure of the overlay network, a distinction is made between structured and unstructured networks. The P2P principle became mainly well known in 1999 with Napster. With the file-sharing application Napster, it was possible to exchange (mainly copyrighted) songs among the participants without having to offer them from a central server.

Popular applications of P2P networks are file sharing (e.g., BitTorrent), instant messaging (e.g., Skype) and blockchain technology (e.g., Bitcoin).

Their independence particularly characterizes P2P networks: there are no control points and not necessarily a fixed infrastructure. This also has a positive effect on operating costs. Besides, P2P networks are self-organized and self-scaling, as each additional user contributes its resources.

However, there are also some challenges in P2P networks that need to be solved for successful operation. These include finding peers in the network (peer discovery) and finding resources (resource discovery). Especially in file sharing networks, solutions have to be found how to motivate users to upload data and not only use the download one-sidedly. The replication of data and the associated availability must also be taken into account in solutions. Another critical issue is the Internet connection of individual participants, which may not be powerful or permanent.

2.2.1 Unstructured P2P

In unstructured P2P networks there are no specifications for the overlay network, so the peers are only loosely connected. Due to the loose structures, the failure of one peer has no significant influence on the function of the rest of the network. Another advantage of the unstructured topology is the lower vulnerability.

While such loose structures are easy to create, the performance of the entire network suffers. A multicast request is sent to all connected peers, who forward the request again and flood the entire network. If a peer can respond to the request, it responds by the same route that the request used to reach it. Each request has a validity time (time to live, TTL) before it is discarded. Popular files with wide distribution can thus be found quickly. However, rare files are difficult to find because the TTL may have been reached before. A flooding search is not efficient and provides a large amount of signaling traffic. An example of this approach is Gnutella.

In order to counteract the problem of inefficient and complicated searching for resources, in other network implementations, central points are created to answer the search requests. These include Napster, FastTrack, and BitTorrent. Figure 2.2 shows a comparison of the search process in the respective networks.

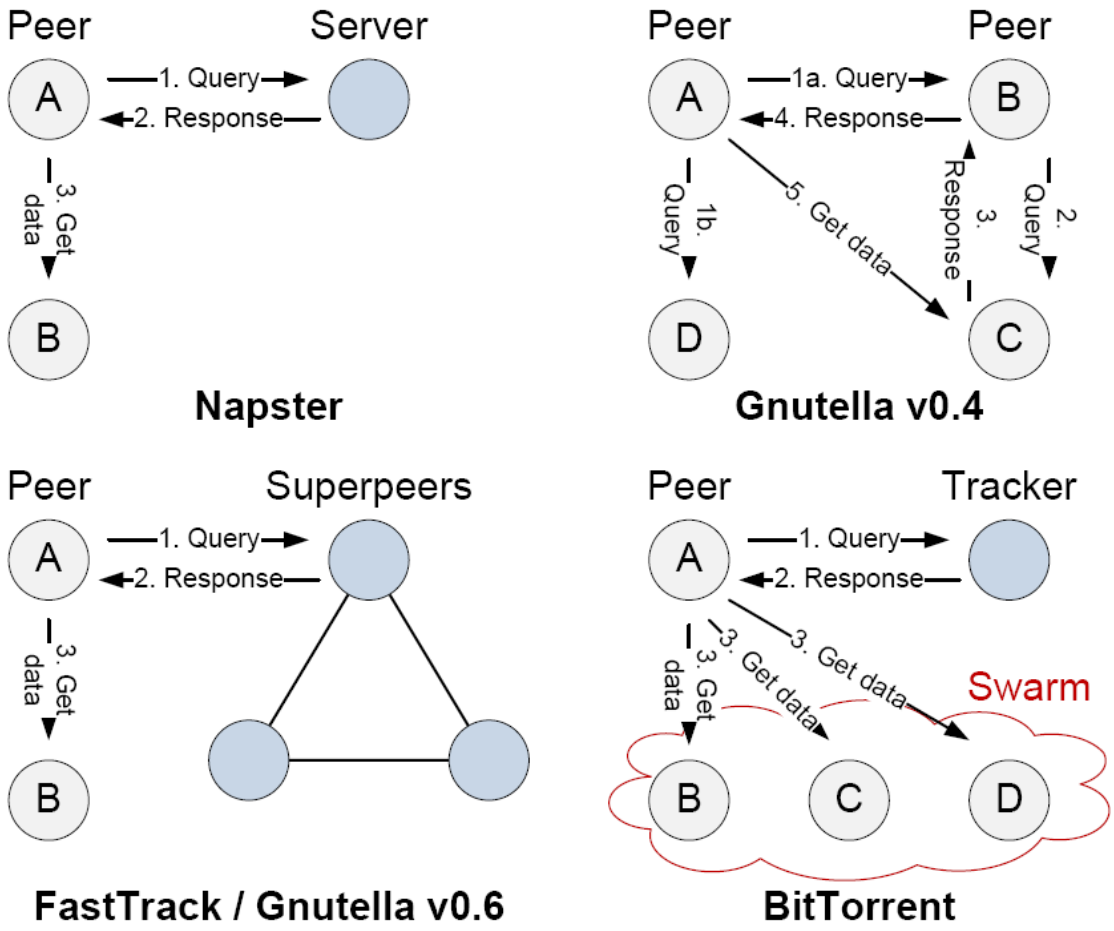


Figure 2.2: Search process in unstructured P2P systems [11]

2.2.2 Structured P2P

By defining a particular structure, for example a circle or a tree, search processes in the overlay network can be designed efficiently and deterministically. In such structured networks, compliance with the structure is strictly controlled. Routing algorithms determine how a node is arranged in the overlay network. The performance of the entire network depends directly on the arrangement of the nodes and how quick changes (joining and leaving nodes) are detected.

Usually, the routing algorithms are based on a Distributed Hash Table (DHT). Hash tables are data structures in which key-value pairs are stored, whereby the key must be unique. The corresponding value can then be queried via the key. The keys are ids, which are generated with a hash function (e.g., SHA-1). For the addresses of the nodes and the files, ids are created equally, so that they lie in the same address space. For finding a file, it is searched at the node with the same or the next larger id. If it is not available there, it does not exist on the network.

For joining a network, either one or more peers must be known as the entry point, or this information must be obtained from a bootstrap server. When entering a structured network, the

joining node is assigned a unique id and thus positions itself in the structure. The routing tables of the nodes affected by the structural change must then be updated.

When leaving a network, this happens either gracefully, and all affected nodes are informed to update their routing tables, or unexpectedly. Therefore, nodes must always check the correctness of their routing tables.

Known routing algorithms that use DHTs include Chrod[19], CAN[12], Pastry[15], Tapestry[21] and Kademlia[10]. Among other things, they differ in their distinct structure and the hash functions used.

2.3 Web 3.0 - Distributed Apps (dApps)

The term Web 1.0 refers to the beginnings of the Internet, which consisted of simple static web pages. The central idea was to present or consume content. The characteristic of Web 2.0 is the user's participation in the creation process. Thus, a series of platforms (blogs, social networks) was launched on which users can provide content and connect. Typically, Web 2.0 platforms have a centralized structure, which entails the problems mentioned in the previous chapter. On the occasion of the 30th anniversary of the World Wide Web in March 2019, the initiator Tim Berners-Lee summed up that the Internet was misused - partly due to the system design [5].

With the next version 3.0 of the web, more transparency, security, and fairness should be created. However, while there is broad agreement on what is meant by terms Web 1.0 and Web 2.0, there is no uniform definition of Web 3.0 that has prevailed to date. There are many ideas, but no final solution yet.

An understanding of what Web 3.0 is, is all about decentralization, which is why it is also called the decentralized Web (dWeb). In this context, Web 3.0 is considered an umbrella term for a group of emerging technologies such as blockchain, crypto currencies, and distributed systems that are interconnected to create novel applications, so-called dApps (decentralized apps). Although decentralized applications have existed for a long time (e.g., BitTorrent), these applications do not meet the criteria of a dApp (see next section).

In the following, the characteristics of a dApp are described and individual, essential components are examined in more detail.

2.3.1 Characteristics of a dApp

Just as with client-server applications, a dApp is also divided into front and back end. The main difference is that the back end is not represented by a centralized server and thus a single point of failure, but by code that is executed in a decentralized P2P network. In the back end so-called smart contracts are used to perform logical operations, and instead of a database, a blockchain is used. There are no special requirements for the front end so that it can be displayed as an app or website. It must only be possible to execute calls from the front end to the back end.

Johnston et al. and Siraj Raval name the following four criteria for a dApp [9, 13]:

- **Open Source**: Trust and transparency are created by the disclosure of the source code. This also enables improvement through contributions from other developers.
- **Blockchain based**: The data of the application is stored cryptographically in a public, decentralized blockchain. The immutability of the blockchain can be used in conjunction with smart contracts to build consensus and trust.
- **Cryptographic token**: Certain actions in the network can only be performed by paying with a token. This token can either be purchased or given to the user in exchange for data, storage space or similar. Especially as a reward for positive actions, users should be rewarded with tokens. However, no entity should have control over the majority of the tokens.
- **Token generation**: The application must generate tokens according to a standard cryptographic algorithm. (e.g. Proof of Work Algorithm in Bitcoin)

In addition to the criteria of a dApp, Johnston et al. also describe a classification system for dApps [9]:

- **Type 1**: Use their own blockchain (e.g. Bitcoin, Ethereum, EOS).
- **Type 2**: Protocols, to use another blockchain of type 1 with own tokens (e.g. Omni Protocol).
- **Type 3**: Protocols with own tokens, which in turn use protocols of a dApp of type 2.

The advantages of a dApp include not only reliability but also the avoidance of censorship and manipulation by design. Furthermore, there are no dependencies towards a service provider. However, the development of a dApp is complex and it is difficult to install updates and bugfixes. Currently, there are still very few dApps, so interactions between different dApps are not able to create synergy effects.

2.3.2 Ethereum

2.3.3 InterPlanetary File System (IPFS)

2.4 Summary

3 Related Work

This chapter gives a comprehensive overview of different projects trying to protect the users' personal data in online social networks. Hereby, six different approaches are presented. First, extensions are considered that make the use of established social networks more secure. Then, alternative social networks will be presented that have placed the protection of personal data at the center. After that two next-generation social networks will be considered, which take advantage of the blockchain technology and belong to the group of dApps. Finally, the ActivityPub protocol is presented, which maps the communication in decentralized platforms. The chapter concludes with a summary of the related work.

3.1 Privacy through Extensions

Existing connections to other people and already created content can bind users to platforms. This so-called lock-in effect prevents users from switching to another platform. If switching to another platform is not an option, how can the use of the platforms be made more secure and user data better protected? In the following, two approaches (Twitterize and FaceCloak) are presented which try to increase the privacy and anonymity on Facebook and Twitter.

3.1.1 Twitterize

With Twitterize, Daubert et al. present an approach on how a particular overlay network can protect data and anonymously exchanged within a social network[8]. The proposal refers to micro-blogging social networks such as Twitter. As proof of concept, they created an Android app.

Design Principles

Daubert et al. made various demands on the proposed solution. Concerning the protection of privacy in general:

- **Confidentiality**: Messages exchanged between sender and receiver should be transmitted secretly.
- **Anonymity**: A individual user should not be identifiable within a set of users (anonymity set).

Also, concerning the design of the implementation:

- **Understandable privacy**: The user should decide for himself which level of privacy protection he chooses for his messages. The system should reasonably communicate this.

- **Simple group management**: Group memberships should be managed quickly and easily.
- **Low overhead**: The client should have the full functionality of the original client and have as little overhead as possible for the extended functionality.

Architecture

The basic idea behind Twitterize is the encrypted, anonymous exchange of messages within a group via an overlay network. There is a group-specific hashtag for this. The messages are forwarded in the underlay network by users until they reach the recipient.

In order to establish communication within a group, the three following phases must be passed through one after the other:

1. Generation and exchange of the keys of a hashtag
2. Flooding and subscription to build an overlay network
3. Exchange of messages

First, a user must define the hashtag and generate a key for symmetric encryption. This key is used later to encrypt the messages as well as the hashtag itself. From the encrypted hashtag, only a hash value is used (so-called pseudonym) so that the actual hashtag stays private.

In order to join the group, other users need the key and knowledge of the name of the hashtag. However, the key exchange should not be carried out via the social network used. An exchange via e-mail, Near Field Communication (NFC), QR codes is conceivable.

If the users of a group with a common hashtag are all aware of the key and the hashtag, the second phase of building an overlay network begins. Here the private flooding mechanism is used [7]. A publisher creates an advertisement tweet consisting of a pseudonym and the first element of a hash chain and posts it in the underlay network. This tweet appears in the timelines of his followers. If not already done, each follower distributes the advertisement tweet on his timeline and thus reaches his followers. However, this tweet differs from the original tweet by extending the hash chain. It generates a hash of the previous hash chain and thus receives the next element of the chain. According to this principle, the entire network is flooded, and as a result, each user saves a triplet consisting of the pseudonym, the hash chain and the user previous to him in the chain in a database table.

If the advertisement tweet reaches a user who is aware of the hashtag and the associated key, the user (so-called subscriber) responds with a subscription tweet. This subscription message is addressed to the user from whom the advertisement tweet was received and contains the user name (@user) and the pseudonym. The user thus addressed in turn posts a message consisting of the user name of the user from whom he received the advertisement tweet and the pseudonym. This happens until the subscription message reaches the original publisher. In this process, the pseudonym and the sending user are stored in another routing table. The overlay network is thus formed. Since each user only has a local view of the

message flow, no conclusions about the sender and recipient can be drawn from the message flow.

In the third phase, users can exchange messages using the previously shared hashtag. The messages are encrypted with the key belonging to the hashtag and then posted together with the hashtag to the timeline. By posting the message again, forwarders ensure that the message is forwarded to the recipient. The exact procedure is shown schematically in Figure 3.1.

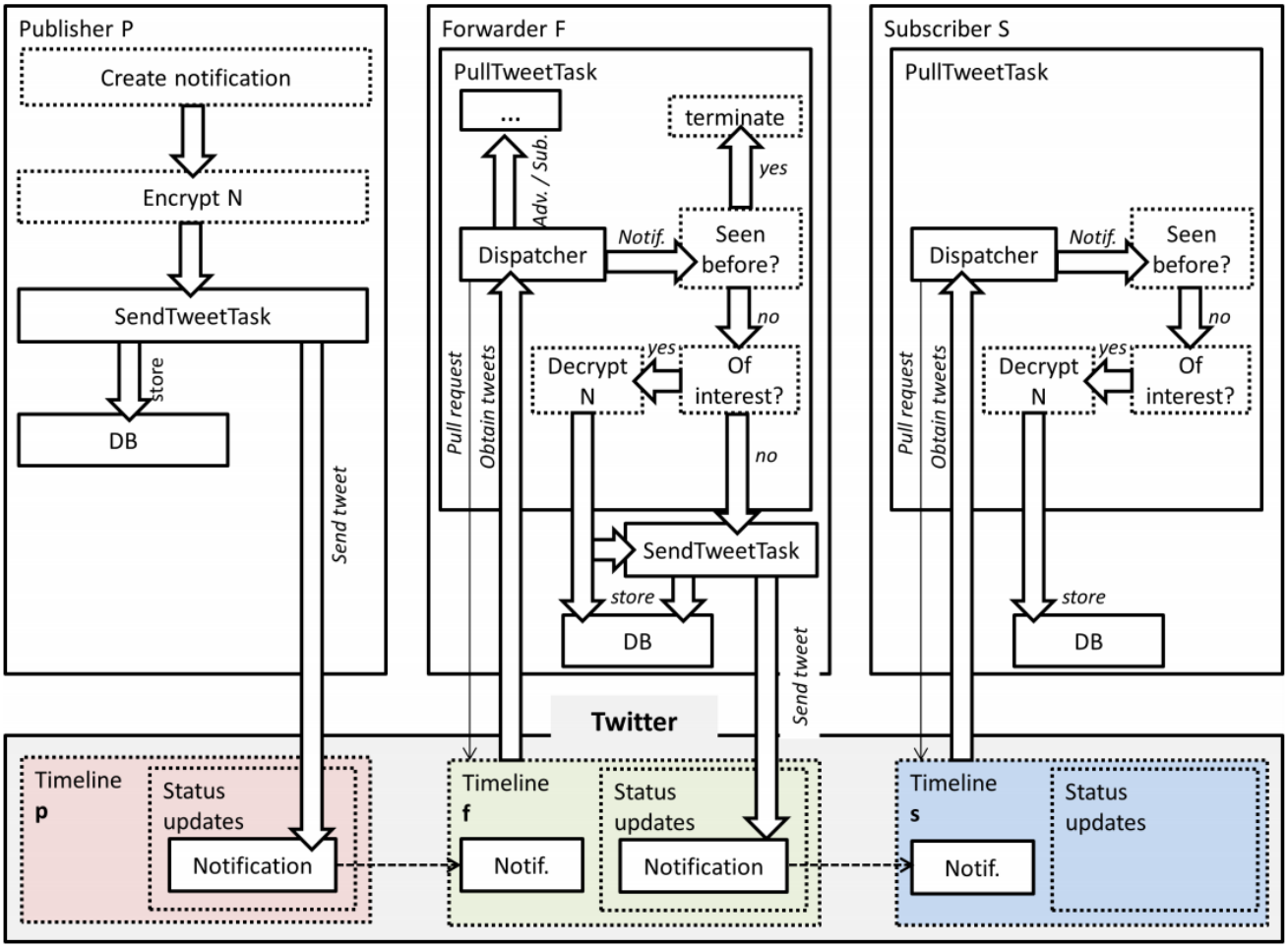


Figure 3.1: Information flow of a notification. The notification is created, encrypted and posted to the publisher's timeline. The notification is then picked up by a forwarder who follows the publisher. Next, the status update is being processed and finally posted on the forwarder's timeline. Lastly, the subscriber who follows the forwarder receives the notification. [8]

Proof of Concept

As proof of concept, Daubert et al. implemented Twitterize as an app for Android. The app was written in Java, the code remained closed source, and the app is not available from the Google Play Store. The representation of the data in the app takes place on different timelines. In

addition to the standard home and user timelines, each hashtag gets its timeline. Each timeline is accessible via a tab.

For encryption 128bit keys with AES CBC is used. The keys are exchanged between the users via NFC or QR codes. Since data structures, such as JSON, are too verbose, the tweets are encoded using Base64. To make message types distinguishable, rarely used UTF-8 symbols are used to give the messages a rough structure. The formation of the overlay network and the exchange of messages then works as described above.

The design of the Twitterize architecture meets the privacy requirements. The other requirements for implementation were also well received during the development of the app. With the exchange of keys via NFC or QR codes an easy way for key management was found and implemented. Just a few seconds of physical proximity is enough to form a group.

The avoidance of overheads was also successful, although the timeline is updated in the background every minute. CPU usage and power consumption were only slightly above the values of the original Twitter app. For the storage of the additional data, only a little space is necessary, since each hashtag occupies only 48 bytes. Assuming that in the Twitter Social Graph two arbitrary users are connected via 4.71 following users in between, an average delivery time of 142 seconds was calculated for a message.

Restrictions arise on the one hand due to limits on the use of the Twitter API and on the other hand because the application must always be online to get the best user experience.

3.1.2 FaceCloak

Researchers Luo, Xiu, and Hengartner of the University of Waterloo in Ontario propose an architecture to protect personal information from social networking platforms. Protection is achieved by transmitting fake data to the social network server and storing the correct data encrypted on a third party server. Authorized users can then replace the fake data with the correct data when they visit the site containing protected data. The prerequisite is that all users use a specific browser extension that communicates with the third party server and replaces content. In concrete terms, this was implemented for Facebook and both a server and an extension for the Firefox browser were developed and successfully tested.

Design Principles

FaceCloak's design is based on the following four principles:

- **Preservation of normal browsing experience**: In order to provide the best possible user experience, the solution should function largely automatically and require only a minimum of interaction.
- **No server-side changes**: The mechanism for protecting personal data should not require any server-side changes.

- **Self-containment and minimal user configuration**: Regardless of the technical abilities of a user, the configuration effort (e.g. the installation of a certain software) should be limited to a minimum and be feasible by everyone.
- **Incremental deployment**: Compatibility between users with and without using the special extension should always be ensured and should never prevent users from no longer being able to contact each other.

FaceCloak Architecture

After validating several available solutions for personal data protection, the researchers decided that a client-side architecture was the best solution for automatic protection. Figure 3.2 shows this architecture schematically.

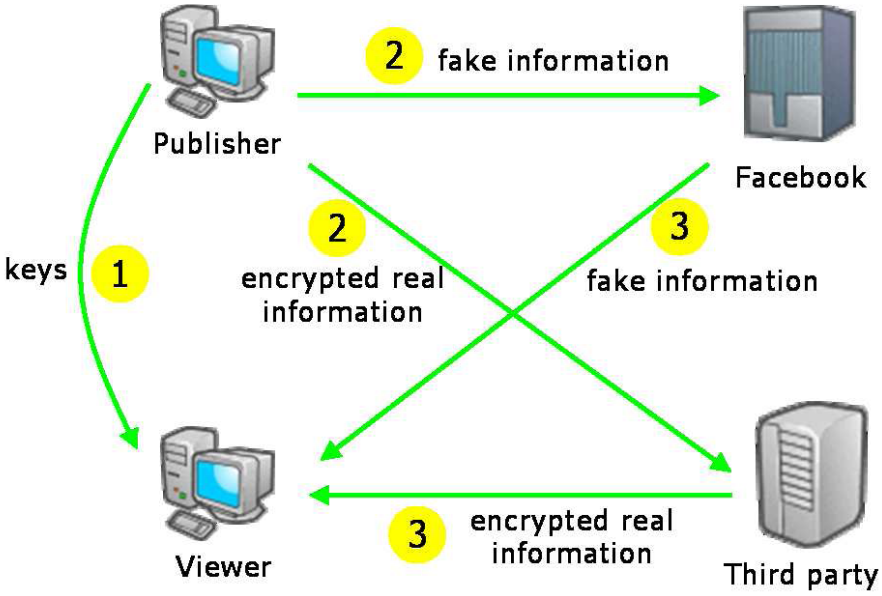


Figure 3.2: Schematic representation of the Setup Phase (1), Encryption Phase (2) and Decryption Phase (3) and the data flow taking place between the entities.

During the setup phase, the browser extension is installed, and the encryption keys generated. Afterward, the keys for decryption are shared with the trusted contacts. In phase two, when data worthy of protection is stored, it is transmitted in encrypted form to a third party server and stored there. Only fake data is transmitted to the social network server. In phase three, whenever an authorized contact calls up a profile page and fake data is transmitted by the social network, the extension takes care of the replacement with the real data.

In addition to adhering to the above design principles, the proposed architecture makes the following contributions:

- The functionality of the service and the interface is not limited by the use of FaceCloak.
- The user decides which information should be protected and which not.
- The architecture can be applied to any social network.

FaceCloak for Facebook

To protect the privacy of Facebook users, Luo, Xiu, and Hengartner have developed a Firefox browser extension according to the previously described architecture, as well as a server application for storing encrypted real data.[1]

The extension uses AES and a key length of 128 bits to encrypt the data. The indices for the encrypted data are calculated using SHA-1. The authors propose an e-mail for the key exchange. For this purpose, the browser extension automatically generates e-mail texts and recipient lists and forwards them to the standard e-mail program. The recipients then have to store the received keys in the extension manually.

In order to protect data with FaceCloak, the prefix @@ must be added to the information in a text field. For other form elements such as dropdowns, radio buttons or checkboxes, the extension creates additional options that also start with @@. When submitting the form, the extension intervenes and replaces the data marked with @@ with fake data. The data to be protected is encrypted with the stored keys and transferred as a key-value pair to the third party server where it is stored. FaceCloak can protect all profile information, but only for name, birthday, and gender algorithms for the meaningful creation of fake data are implemented.

In addition to profile information, the extension can also protect Facebook Wall and Facebook Notes data. To avoid attracting attention with random, unusual character strings, the contents of random Wikipedia articles are transmitted as fake data.

When loading a profile page that contains protected data, the extension with asynchronous HTTP requests retrieves the information from the third party server, decrypts it, and replaces the fake data. A large part of the replacement can thus be performed during the load process so that the user does not see the fake data. However, since Facebook also loads content asynchronously, some replacements can only be performed with a time delay and the fake data is shortly visible.

To use the same account on multiple devices, the keys must be transferred to all devices and stored in the extension. It is not possible to use multiple accounts with the same Firefox profile, as all data is stored in the extension and these are always bound to exactly one Facebook account.

The latest version 0.6 from August 2010 cannot be installed in the current Firefox (version 65). Furthermore, it is not known if the server is still running. Therefore it is not possible to check if the extension still works. Due to the numerous updates and sometimes serious changes that Facebook has experienced in the last 8 years, it is very unlikely that the extension will still work today. At that time, however, it was successfully applied and proved that the proposed architecture worked.

[1] Download: https://crysp.uwaterloo.ca/software/facecloak/download.html

3.2 Privacy-protecting Social Networks

In the business models of the large, popular OSNs, user data plays an essential role. The data is evaluated and used to make a profit, for example through personalized advertising. Anonymity and the protection of privacy are not among the overriding objectives.

In the following, two social networks, diaspora* and LifeSocial, are presented which have placed the protection of data at the center.

3.2.1 diaspora*

Inspired by a lecture on surveillance in centralized social networks by Eben Moglen on February 5, 2010, the four mathematics students from New York University Grippi, Salzberg, Sofaer and Zhitomirskiy had the idea for diaspora*[16]. Diaspora* is a decentralized social network. To its special features count:

- **Decentralization**: Everyone can start their server with the diaspora* software and be part of the network. Alternatively, there are public servers accepting registrations from everyone, so there is no need to set up an own instance of diaspora* to participate in the network.
- **Privacy**: By running a server, the data remains with the user. Furthermore, it is possible to determine which users can see content.
- **Open Source**: The source code is disclosed and hosted on GitHub[2]. The transparency created in this way ensures trust in diaspora*.

For funding the development of diaspora*, $ 10,000 should be crowdfunded on Kickstarter[3]. The project was very well received so that after 14 days the target was reached[17] and in the end, a total of $ 200,641 was donated. In November 2010, the first alpha version of diaspora* was released. One year later there was a prominent feature update. In May 2012 it was announced that diaspora* should be further developed within the Y Combinator start-up program[4]. Due to the commercial influence, there were fears that diaspora* could lose its independence. In August 2012, the developers announced that diaspora* is henceforth a community project[6].

The diaspora* back end is written in Ruby, the front end to the user is a website. A server running diaspora* is called pod. Each pod has its own domain, so users of a pod have a username similar to an e-mail address (for example, username@podname.org). Diaspora* has the typical functionalities of a social network (hashtags, @ mentions, likes, comments, private messages). What marked a peculiarity at the time of diaspora*'s appearance are so-called aspects. Aspects are groupings of contacts that can be specified as a target audience when posting content. Only the contacts associated with the aspect can see the post.

[2] https://github.com/diaspora/diaspora
[3] https://www.kickstarter.com/projects/mbs348/diaspora-the-personally-controlled-do-it-all-distr/description

For staying in contact with friends on other platforms like social networks (Facebook, Twitter) or blogs (Tumblr, Wordpress), the initial idea was to connect these platforms. Data exchange should work both ways. Posts published on diaspora* should also appear on other platforms at the same time. Also, posts from the other networks should be viewed in diaspora*. Diaspora* should play the role of a social media hub. Unfortunately, the APIs of some platforms have become increasingly limited as instances of misuse of the interfaces have become public.

The data of the users are unencrypted on a pod so that someone having access to the database can see them[2]. In order to protect his own data in the best possible way, the operation of a separate diaspora* instance is necessary. The communication between the pods is encrypted with SSL[2]. Furthermore, the exchanged messages are first signed (Salmon Magic Signatures), then symmetrically encrypted with AES-256-CBC[3]. The AES key is encrypted with the public key of the recipient and sent together with the encrypted message.

Diaspora* does not use the ActivityPub protocol, but its own diaspora* federation protocol[1]. Other platforms such as Friendica[4], Hubzilla[5] or Socialhome[6] can also communicate via the diaspora* federation protocol. There is no official API, which makes app development difficult. Diaspora* points out that the website is also usable on mobile devices, so there is no need for a native application[2].

According to the statistics of the-federation.info[7] on February 24, 2019, 679723 users were registered on a total of 251 pods. Over the last 12 months, 19591 new users have joined the network. In January 2019, only 4.4% of all users were active with 30042 users. However, the numbers are incomplete, as some pods do not share information and there may be more than the 251 listed pods.

3.2.2 LifeSocial.KOM

3.3 dApps - The Next Generation Social Networks

3.3.1 Akasha

In early 2015, Mihai Alisie (co-founder of Ethereum) had the idea for AKASHA. AKASHA is a social network that differs from other known social networks mainly in its decentralization. The absence of a central server meant that censorship was ruled out by design alone. This is realized by the two technologies Ethereum and the Inter-Planetary File System (IPFS). Electron, React, Redux, and NodeJS complete the technology stack so that the primary programming language is JavaScript. In addition to Mihai Alisie, 12 other employees now work at AKASHA. Furthermore, the founders of Ethereum (Vitalik Buterin) and IPFS (Juan Benet) advise the project.

4 https://friendi.ca/
5 https://zotlabs.org/page/hubzilla/hubzilla-project
6 https://socialhome.network/
7 https://the-federation.info/diaspora

Alisie sees AKASHA as „the missing puzzle piece that will enable us to tackle two of the most critical challenges we face today as a modern information-based society: freedom of expression and creative perpetuity“. The central goal is therefore to prevent censorship and to obtain information over a long period.

«Describe what AKASHA can do»

After a proof of concept had validated the idea, the technology stack mentioned above was defined, and development began at the beginning of 2016. The first goal was to develop a client based on Electron for Windows, Linux, and MacOS. In January 2017, the first functional alpha version was completed and tested with a closed circle of users. Over time, additional functions were added, bugs fixed, and performance optimized. In November 2017, a web version of AKASHA[8] was introduced. This was a big step towards a better user experience since the web client does not need to download the Rinkeby blockchain which took around 30 minutes on the first run. The public beta phase started in February 2018 with the primary goal to see how the application behaves under heavy load.

With the announcement of the web version, the team behind AKASHA also released plans for their AETH token. This token should have the unique feature that it can take different states. The state transition from one state to the other is only possible as shown in Figure 3.3. The developers describe the states as follows:

- **AETH** is a transferable, ERC 20 compatible token, living on the Rinkeby test network
- **Mana** is non-transferable and is obtained by locking AETH for X time at Y ratio (Manafied AETH). The Mana amount regenerates every day for as long as AETH remains locked, in a „Manafied“ state.
- **Essence** is non-transferable and is obtained through positive contributions. It can be burned to mint new AETH into existence. When people use their Mana to vote on artifacts, the authors can collect the burned Mana as Essence.
- **Karma** is not a state, but rather a score tracking user contributions. For every unit of Essence collected, the user receives also Karma. Karma is used for defining milestones, thresholds and unlocking functionality within the dapp.

After a long period of silence, the entire project was converted in January 2019, almost three years after the start. The domain changed from akasha.world to akasha.org, and the focus shifted from a social network to an umbrella organization that unites several projects. In the AKASHA blog, Alisie writes about metamorphosis and compares the change from a caterpillar to a butterfly. The alpha and beta phases are said to have corresponded to the caterpillar phase, the second half of 2018 to the chrysalis stage, and now the butterfly is supposed to unfold with all its beauty. However, it is left open how the new orientation will look like and how the social network will continue. On the website, there is a software section as well as a hardware section. However, there is no content available yet. The public launch of the social network in version 1.0 was planned for the fourth quarter of 2018. With this launch, the change from the Rinkeby test network to the Ethereum main network should also be completed. It is currently not known when the public launch will take place.

[8] https://beta.akasha.world/

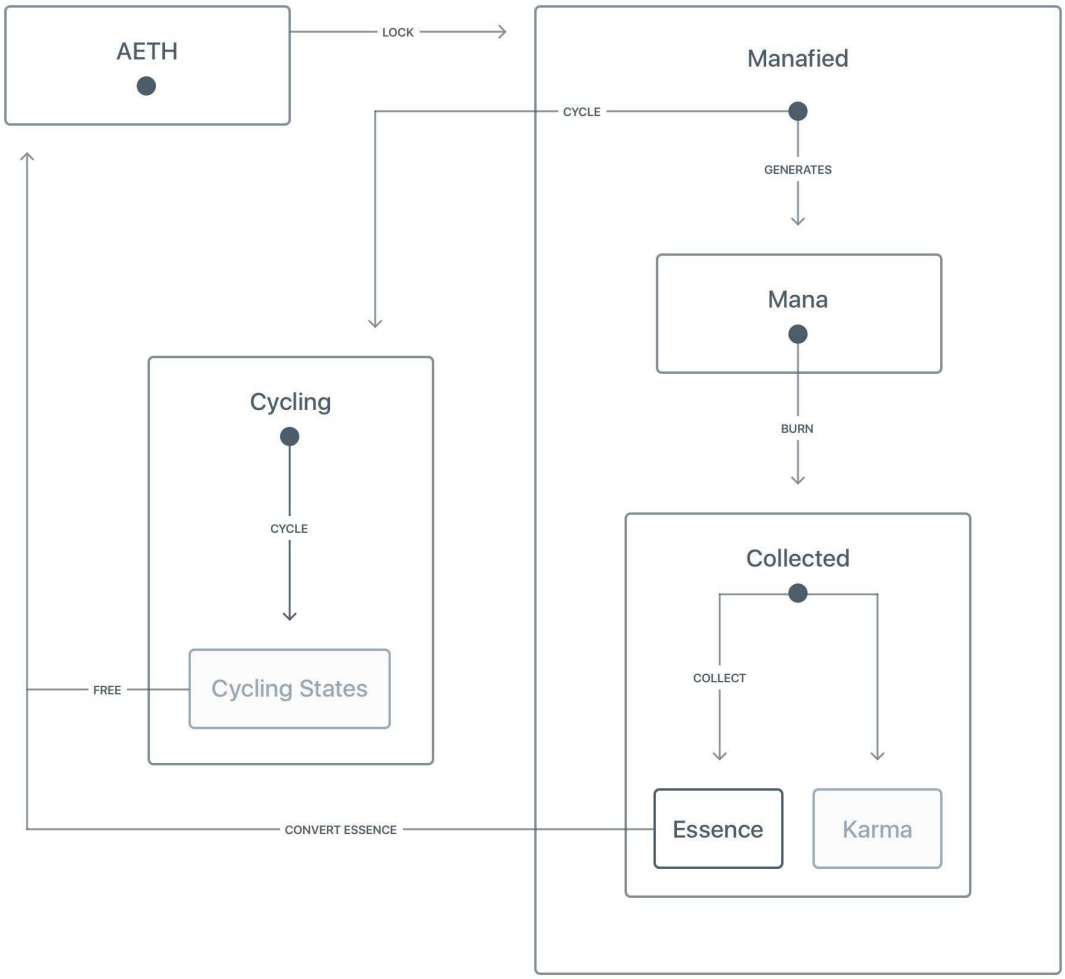


Figure 3.3: AETH flow

3.3.2 Peepeth

Peepeth is a microblogging platform that is very similar to Twitter in functionality and design. There are also other parallels to Twitter: Instead of a blue bird in the logo, Peepeth uses a penguin and instead of *tweeting* users *peep*. The maximum post length is limited to 280 characters. This has no technical cause but was taken over by Twitter. The main difference to Twitter is the decentralization. Behind Peepeth's development is Bevan Barton, who launched the platform in March 2018.

From Peepeth's point of view, social media is broken. The fault lies with the operators of social networks, who control the online identities of users, sell their data and violate their privacy. The news feeds are manipulated to drive the user to a higher level of interaction at any price. Besides, the platforms are teeming with trolls, bullying and flame wars. Barton wants to counter these grievances. Therefore there is no advertising on Peepeth.

The website Peepeth.com is the front end of a decentralized app (dApp), which uses the Etherum blockchain and the Inter-Planetary File System (IPFS). This front end can theoretically be exchanged arbitrarily and Peepeth's data can be read and written because the blockchain protocol is open. No Ethereum test network is used, but the main network. The execution of

transactions on the Ethereum blockchain is associated with costs. Peepeth bears the costs for its users. The necessary capital was collected via a crowdfunding campaign. However, when accounts distribute spam, Peepeth no longer bears the cost of writing to the blockchain. The resulting costs should make spamming unattractive and reduce it to a minimum because technically it is still possible. Peepeth requires a dApp browser (e.g., Opera) or a browser that has been extended by a wallet (e.g., using MetaMask extension) for use. Although Peepeth covers the costs, the user has to sign the transactions, which is why the browser has to contain the corresponding extension.

In order to keep transaction fees low, the actions executed on Peepeth are collected on the server hosting the front end and written to the blockchain in batches every hour. Several actions are bundled in one file and transaction. The actual contents end up as a JSON file in IPFS and only the reference hash in the blockchain.

While the smart contracts are open source, the front end is closed source. So it is impossible to understand what is happening on the server hosting the front end Peepeth.com. Image files are not only stored in IPFS but also mirrored at Amazon AWS to provide a better user experience. The client does not communicate directly with IPFS, but the server behind the front end communicates with the two back end technologies IPFS and Ethereum Blockchain, as shown in Figure 3.4.

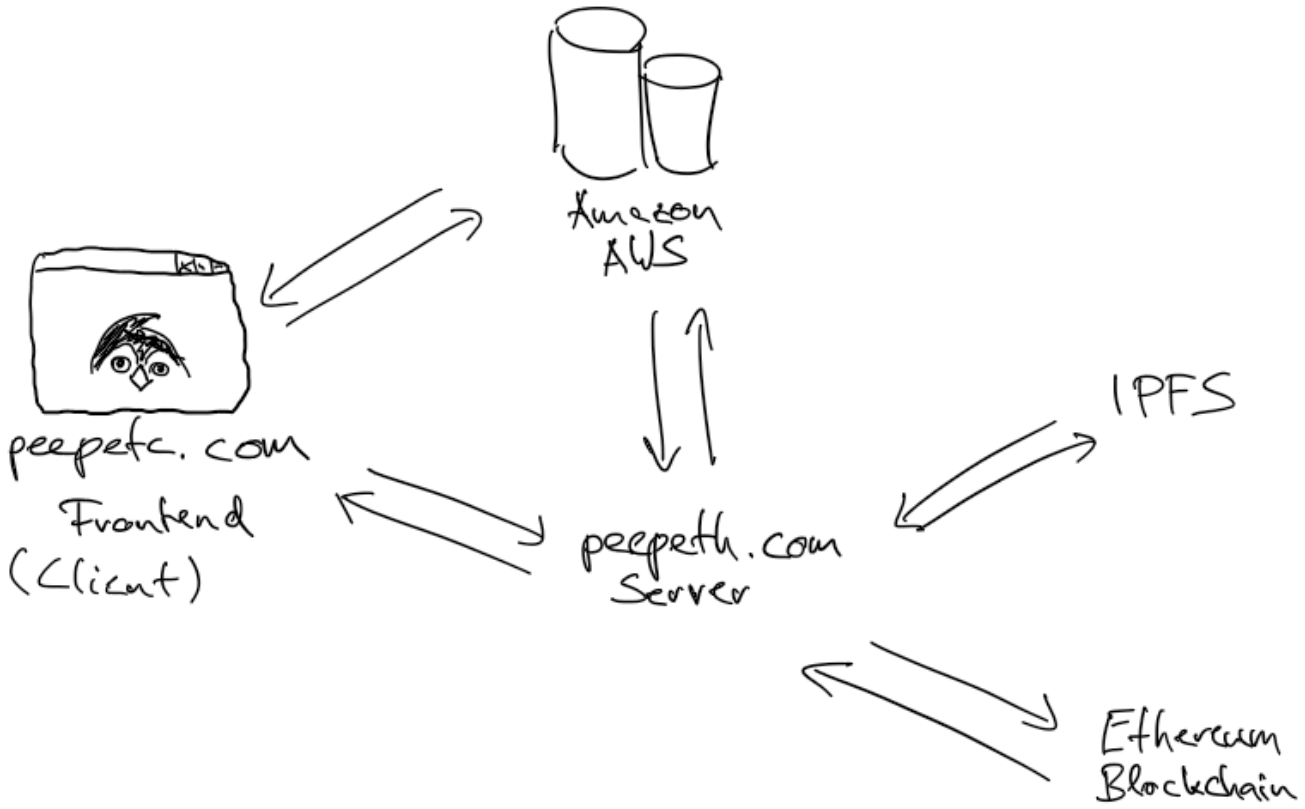


Figure 3.4: Peepeth architecture

The data written to the Ethereum blockchain cannot be deleted or modified. For Barton, this is a primary advantage since everyone is forced to be aware of his actions and the self-confidence for his actions is sharpened. Furthermore, this fact of immutability is the main

argument for freedom of expression and against censorship. However, not all messages are presented in the Peepeth front end. If they violate Peepeth's terms of use, a filter will sort them out. Peepeth calls this procedure „moderation" and argues that this is by no means to be understood as censorship, but much more „on cultivating mindful engagement and positive contribution".

In addition to writing short messages, it is possible to like posts. However, posts cannot be liked infinitely. There is only one like per day available, a so-called *Ensō*. The resulting rarity should express the particular appreciation of a contribution. „Ensō (Japanese for ‚circle') is a circle that is hand-drawn in one uninhibited brushstroke. It represents creativity, freedom of expression, and unity". Furthermore, good content from other users can be rewarded with a tip. 10% of the tip goes to Peepeth and serve to finance. Also for the verification of an account and special badges go 10% of the fees to Peepeth.

On 29th January 2019, Peepeth had 4055 users who posted a total of 66262 Peeps. For an account, future users have to apply first and receive a sign-up link by email to join the platform after some time. On invitation of an active user, new users can join directly without waiting time. Users can verify themselves with their existing Github and Twitter accounts. In the future, it will be possible to use further platforms for the verification of an account. In order to verify an account, the user must post a „special message", which also contains his Ethereum address. The link to this post must then be handed over to a Smart Contract, which confirms the ownership of the account.

Peepeth communicated the next milestones to increase the user experience as part of the crowdfunding campaign. The first milestone has already been reached. The fact that the possession of a cryptocurrency requirement for the use and procurement of such a currency is difficult was to be eliminated. Peepeth bears the costs for its users. The next steps are the use without special software requirements (renouncement of particular browsers or MetaMask) and the development of an iOS app. However, only 140.56 ETH could be collected from the required 1000 ETH. It is unclear to what extent the desired goals will now be achieved.

3.4 Protocols

3.4.1 ActivityPub

ActivityPub is a protocol published by the World Wide Web Consortium (W3C) in January 2018 as an official standard[9]. The protocol regulates communication within an open, decentralized social network. There are two levels: client to server (Social API) and server to server (Federation Protocol). The two protocols are designed in such a way that they can be used independently of each other. If one of them is implemented, it is easy to implement the other. The Activity Streams[10] data format is used to describe activities in JSON-LD format. This data format is also an official W3C standard with the aim to record meta data of an action in a human-friendly but machine-processable syntax.

[9] https://www.w3.org/TR/activitypub
[10] https://www.w3.org/TR/activitystreams-core

The principle behind Activity Pub is similar to that of e-mail. Servers can be uniquely identified via the domain. Within a server, each mailbox is accessible via a unique name. Thus, users can communicate with each other via different servers by having their messages forwarded to their mailbox.

Client to Server (Social API)

Users are called actors in ActivityPub and are represented by an associated account on the server. Since there can be several servers, it is important to emphasize that an account is only ever located on one server and user names must always be unique only within a server. Each actor has an inbox and an outbox on the server on which he is registered with his account. These are the two endpoints with which the client application communicates via HTTP requests. More is not necessary, because the server ensures that all messages and information for the user end up in his inbox and that the messages in his outbox are forwarded to the desired recipients (see 3.4.1). For ensuring that only authorized clients store content in an outbox, the sender must sign the posts.

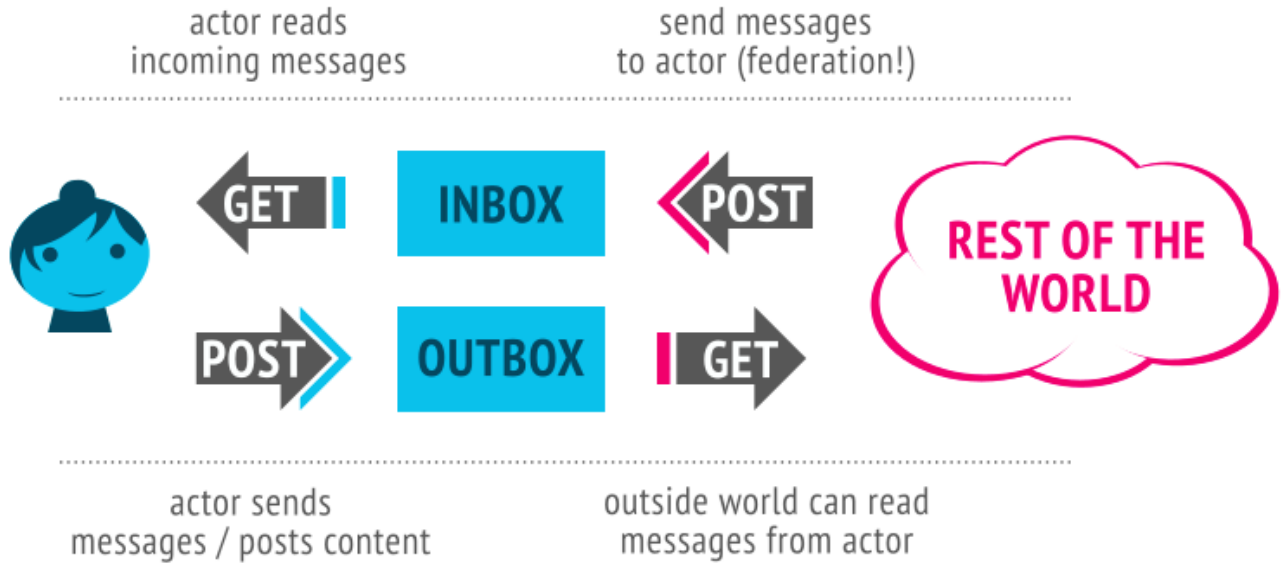


Figure 3.5: Concept behind ActivityPub [20]

The outbox of an actor holds all his published posts. When accessing the outbox of an actor without authorization, the server delivers all public posts of the actor. If access with authorization occurs, the explicitly shared content is also transmitted.

The corresponding actors can only access their own inbox. On access, the messages are downloaded from the server. New messages get in the inbox per HTTP POST request from another server.

Each actor has some so-called collections. To these collections, objects can be added and removed. The collections are used to store information related to an actor. These are the collections each actor has:

- **Followers Collection**: List of actors who have posted the actor a follow-activity and follow him. Can be used as addressee when sharing a post.
- **Following Collection**: List of actors followed by an actor and whose content he is interested in
- **Liked Collection**: List of all objects the actor has liked

The following interactions are defined between the client and the server, but some of them are optional to implement: Create, Update, Delete, Follow, Add (add an object to a collection), Remove (remove an object from a collection), Like, Block, Undo. To address the activities, *to*, *bto*, *cc*, *bcc* provide the same possibilities as are known from e-mail. The server then has to ensure that the activity also reaches the addressed actors.

Server to Server (Federation Protocol)

This protocol defines the exchange of activities between actors of different servers. The network between the servers with all actors is called a social graph. The interactions defined in the Social API must be implemented by the server so that they reach the addressed actors. The starting point is always the outbox of an actor. If a new activity is created using an HTTP POST request and, for example, the follower collection of the actor is selected as the addressee, the server must ensure that every actor in the follower collection receives the activity in its inbox. The recipients and their addresses can be found by following the links in the activities. It is also up to the server to ensure that there is no duplication of content.

Application Examples

The most popular application example is the social network Mastodon[11]. Mastodon is a decentralized network based on free and open source software. Everyone is invited to host their own platform. With currently 1.6 million users it is the most significant implementation of the ActivityPub protocol. The Federation Protocol is used for communication between the individual servers. The Social API is not used, instead Mastodon offers its own API[12] for communication with the client.

Networking with other users is not limited to mastodon instances. Each service that has implemented the ActivityPub protocol allows its actors to network with actors of entirely different applications because the communication is standardized. So it is possible without problems to follow an actor of the video platform PeerTube[13] as Mastodon actor and be notified when he uploads a new video there.

In addition to cross-platform networking, one big advantage is that it has no significant impact no matter what happens to Mastodon. The network can still exist and thanks to the open

[11] https://joinmastodon.org/
[12] https://docs.joinmastodon.org/api/guidelines/
[13] https://joinpeertube.org/en/

protocols and open source software it can be developed and used without restrictions. If Facebook were to go offline, all contacts would be lost, and the platform would never be accessible again.

3.5 Summary

4 Concept of a Hybrid Online Social Network

The criticism of the protection of privacy on the Internet, especially in social networks, is not new. As early as 2010, the founders of diaspora* discovered that there was no social network that sufficiently protected the privacy of users[16]. Their idea of a decentralized network that protected user data by design convinced so many people that even before the start of the development people donated $200,000 instead of the required $10,000 to a Kickstarter campaign.

The reason for the inadequate protection of personal data lies in the centralized system structure used by all leading social platforms. With this structure, the data is stored centrally and mostly unencrypted. The service provider therefore inevitably has access to this data. Which data is collected during use and what happens to the data is not transparent to the user. On the one hand, the user data is evaluated in order to improve the user experience (suggestions for content matching the user's preferences using recommender systems), but on the other hand also in order to make a profit. Revenues can be generated through personalized advertising or, in the worst case, through the sale of data. Furthermore, the protection of data against access by third parties via official interfaces (harvesting) or unauthorized hacking cannot be ruled out. Last but not least, due to applicable law, it may be necessary for data to be transferred to secret services or government agencies.

Although the problems and dangers are known for a long time and new scandals regularly become known to the public, the users remain mostly loyal to the respective social networks. Alternative social networks, which focus on privacy protection (e.g., Vero[1], Ello[2]), lack attractiveness so that they gain only a few users and often fail after a short time. The connection to the respective social network is so strong that the barrier to switching to another, more secure social network is not overcome. The amount of content already created, the social network built up, and a large number of contacts using the same platform all create this so-called lock-in effect.

If switching to another platform is not an alternative, it is necessary to look for ways to better protect users and their data on existing platforms. The Researcher Training Group (RTG) „Privacy and Trust for Mobile Users“[3] in research area B „Privacy and Trust in Social Networks“ is dealing with problems like this. Subarea B.2 deals specifically with the protection of privacy in hybrid social networks.

In the following, a concept for a hybrid social network will be developed that tries to take into account the interests of the different stakeholders. Besides, functionality requirements and potential limitations are listed. Finally, a solution strategy is shown, and a possible architecture is presented. The concept should apply to all social networks. A specific implementation will be presented in the next chapter for Twitter.

1 https://www.vero.co
2 https://ello.co
3 https://www.informatik.tu-darmstadt.de/privacy-trust/privacy_and_trust/index.en.jsp

4.1 Requirements to the Hybrid OSN

The solution should meet specific functional and non-functional requirements so that users are willing to use another client.

Functional requirements:

- The standard functionality of the OSN is usable without restrictions.
- The user can see which way (private P2P or public OSN) the data comes from and where it goes.
- Data access should only be possible for authorized users.
- The data exchange should be automatically encrypted so that the data is worthless for third parties.
- The data format is flexible in order to map changes and new OSN functionalities.
- The solution is client-side since there is no control over the OSN server.
- The OSN Service Provider can retrieve anonymized data relevant to him.
- The solution is platform independent.

Non-functional requirements:

- Data exchange over the private network is fast and secure.
- The user interface is simple and understandable.
- The design of the app is modern and appealing.
- No violation of the guidelines and terms of use/service of the OSN.
- No restrictions for standard users who do not use the hybrid solution.
- The additional effort for the user when using the hybrid solution should be minimal.

4.2 Stakeholders

Even if the stakeholders are not necessarily directly involved, it is still essential to understand the views and interests of the various parties and take them into account in the design. Table 4.1 shows the interests and points of view of the parties involved directly and indirectly.

Steakholder	Interests	Attitude towards hybrid OSN	Influence on hybrid OSN
Service Provider	• Collect as much user data as possible • Binding users to the platform • Profit maximization	• Negative, as undesirable	• No direct influence on the development of the hybrid solution • By identification of the hybrid OSN users their exclusion • Blocking the hybrid OSN • Great influence
OSN User	• Unrestricted use of the OSN • No disadvantages due to hybrid OSN users	• Neutral	• No influence
Hybrid OSN User	• Unrestricted use of the OSN • Decision-making sovereignty over what happens with data	• Positive, as desired	• Feedback for improvement • Low influence
Developer hybrid OSN	• Scalable, secure, fast data exchange solution • No costs for infrastructure and development	• Positive	• Great influence, since decision-making sovereignty
Governments	• Compliance with laws • Supervision • Censorship	• Depending on policy and legislation • At best, positive and supportive • In the worst case, negative and preventing	• At best, promotion and financial support - great influence • In the worst case legal action - great influence • No other influence

Table 4.1: The table shows the interests, attitudes towards a hybrid solution and influence on the hybrid solution of several stakeholders.

4.3 Restrictions

When designing the hybrid OSN, there are a few limitations that need to be considered, for which appropriate solutions have to be found. These restrictions include:

- **Interfaces of the OSN**: Ideally the OSN offers a public API with full functionality. Since the user's data should be protected as best as possible, access via the API is usually restricted. The restriction may be due to a limited number of requests per time interval or a limited range of offered functions.
- **Crawling the OSN web pages**: If there is no official API or if it is sharply restricted, the contents can theoretically also be extracted by crawling. However, this brings with it several challenges. Modern web pages load many contents asynchronously so that the initial HTML does not yet contain these contents. Furthermore, there are sophisticated mechanisms that notice crawling and lock out crawlers. Likewise, it may be difficult to add data to the OSN. For security reasons, in most cases, special tokens are sent along with each request to detect and prevent abuse and fake requests.
- **Development, operation and licensing costs**: Costs for the development, operation and licensing of third-party software may incurred. At best, conscious decisions lead to the avoidance of expenses.
- **Operating system or runtime environment**: Nowadays OSNs can be used on almost all devices; independent of their operating system. In order to achieve the same user experience, the hybrid OSN should be usable on the same platforms. Any restrictions imposed by the operating system (user and application rights, connectivity) must be taken into account during development.
- **Resources**: The devices running the hybrid OSN may have limited resources (storage space, processing power, Internet connection/data volume, battery). When making design decisions, it is important to plan as resource-conserving as possible and to find scalable solutions. Overall, the overhead for the hybrid extension should be as low as possible compared to the original application.
- **Availability of data**: The data that is exchanged securely and not via the OSN's servers must always be available. Whether a user is offline or how old the data is must not affect its availability.

While the restrictions on the hybrid client itself can be actively influenced and resolved, the restrictions on the OSN cannot be controlled. If the OSN does not provide any interfaces and the hurdle of data exchange with the servers is insurmountable, this can completely prevent the development of a hybrid client.

4.4 Quality Goals

The following quality objectives (Table 4.2) have been defined to ensure that the quality of implementation is as high as possible. By adhering to the quality goals, errors and problems

should be avoided, the application should remain maintainable, and new developers should be enabled to get started quickly.

Quality Goal	Motivation
Analyzability	- Module, class and method names in English - Detailed documentation of the public interfaces - Compliance with the Clean Code principles
Changeability	- Common programming language - Program modules against interfaces to keep them interchangeable
Testability	- The architecture should allow easy testing of all building blocks
Transparency	- The application should be Open Source

Table 4.2: The quality goals are the relevant requirements and the driving forces that software architects and developers should consider.

4.5 Solution Strategy - Architecture

Various models can be used to implement a secure data exchange between the users of an OSN via an add-on. The solution strategies shown below differ primarily in the question of where data is stored and how it can be found.

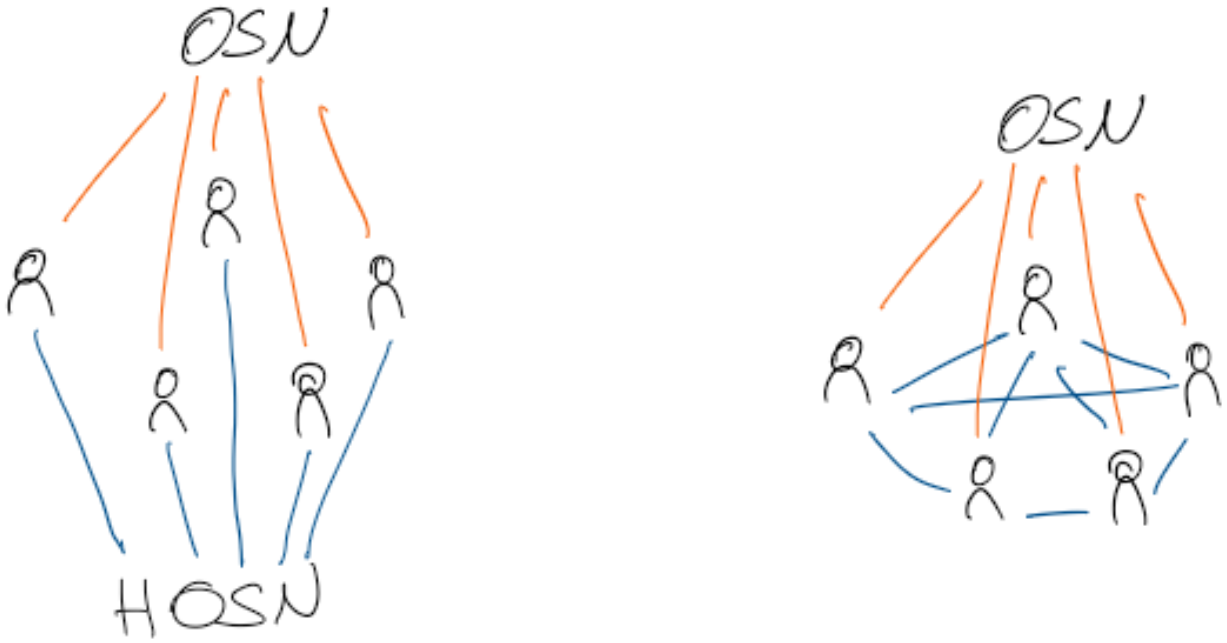


Figure 4.1: Different architectures: a) Use of a central server to which all hybrid OSN users connect to, b) Creation of a P2P network among the users for data exchange.

One possibility is to use an extra infrastructure to store the data, as shown in Figure 4.1.a. Additional servers are used to store and distribute the private data to be protected. Using additional servers has the advantage that the data is always available and there are no dependencies to

other hybrid OSN users. Furthermore, resources must only be available centrally and not locally for every user. At the central location, the data can be indexed and explicitly queried. However, the operation and maintenance of one or more such servers are problematic. In principle, the question of the operators must be clarified, because the infrastructure must function reliably. An architecture based on this proposal was used by FaceCloak.

In contrast, a decentralized or distributed solution strategy would create a network among users of the hybrid application. This strategy is depicted in Figure 4.1.b. No extra infrastructure would have to be operated. The users would then have a typical peer role. With this model, solutions must be found for how data is always available and can be found, even if a user is temporarily or permanently offline. Furthermore, the resources on the end devices are limited, so that effective, economical solutions are needed. Another challenge is the addressing of peers. Since they typically do not have a static IP address, but the IP address changes frequently, solutions must be found for accessibility. Since there is no central, global index, finding data is even more difficult.

An interim solution is also conceivable, in which an existing infrastructure, e.g., an already existing P2P network or the blockchain, is used for storing and exchanging data. Since no influence can be exerted on existing infrastructure, its use entails further restrictions and potential risks.

Table 4.3 lists the advantages and disadvantages of the different strategies.

4.6 Solution Strategy - Client

Concerning the implementation of the hybrid approach, two possibilities are conceivable. On the one hand, the extension of the original OSN client (app or web front end) as an addon. On the other hand, the development of an entirely new client.

When an addon extends the OSN, there is no need to take care of the standard functionality of the OSN. Therefore, the development can entirely focus on the addon and the private extension. At crucial points, the add-on extends the interface by additional elements that enable secure data exchange. The service providers usually do not offer developers an interface to extend the OSN with such own functionalities. With web front ends it is possible to manipulate the website content using browser add-ons. One example for doing so is FaceCloak. Since such browser extensions manipulate the Document Object Model (DOM), knowledge about the document structure is necessary for the successful function in order to make changes in the right places. The short release cycles of OSNs and the associated frequent changes to this DOM structure make it difficult to keep up with the changes. When consuming the OSN via the official apps on mobile devices, an extension or manipulation is not possible.

The alternative to the extension approach described above is a new hybrid client app. The entire functional range of the OSN must be implemented and kept up to date. As already mentioned with the restrictions (see 4.1), the functions are usually not entirely provided via an API, and the crawling of the content also brings with it some challenges. By having complete control over the development of the client, the additional protected, secure communication can be added at

	Advantages	Disadvantages
Own infrastructure (centralized)	• Availability of data • Finding the data • Resources only have to be available centrally • No dependencies among hybrid OSN users	• Expenses • Who operates the infrastructure? • Compliance with legal requirements
Own network (decentralized/distributed)	• Resources scale with increasing number of users	• Availability of data • Finding the data • Addressing the peers • Local resources limited
External infrastructure	• Ideally no costs • Resources are provided by the external infrastructure	• No influence on future development • Dependence on infrastructure entails risks

Table 4.3: Advantages and disadvantages of the different solution strategies for the hybrid OSN architecture.

the appropriate points. In the best-case scenario, at least one hybrid app is available for every operating system for which an official OSN app exists.

Both approaches can be combined by displaying the (mobile) web page of the OSN in a WebView in a separate app and executing DOM manipulations via injected JavaScript code. For example, there are some alternative clients for Facebook (e.g. „Friendly for Facebook“[4] [5], „Metal Pro“[6]) that use this approach.

4.7 Summary

[4] https://play.google.com/store/apps/details?id=io.friendly
[5] https://itunes.apple.com/de/app/friendly-for-facebook/id400169658
[6] https://play.google.com/store/apps/details?id=com.nam.fbwrapper.pro

5 Proof of Concept

After working out the concept of a hybrid OSN in the previous chapter 4, this chapter presents a proof of concept in the form of an individual Twitter client for Android. The previously described solution strategies, as well as functional and non-functional requirements, actively influenced the development of the client, and attention was also paid to compliance with the defined quality goals. In the following, the decisions made are explained, and the resulting architecture is presented.

5.1 Objective

With the proof of concept, the basic feasibility of the idea of an extension of an established OSN by a secure data exchange should be proven. Within the framework of this thesis, the task was to provide the proof of concept as a native Android app. Concerning the architecture, the focus from the beginning was on a P2P solution, which is why a solution with its additional servers was not pursued further in the development of the prototype. Furthermore, an interface should be available for the service provider of the OSN, through which anonymized information can be obtained from the privately exchanged data.

Even though the implementation as an add-on, as shown in the previous chapter as a possible solution strategy, was thus fundamentally ruled out, it nevertheless influenced the made decisions. It was always considered how the architecture could be this open and flexible to enable all kinds of extensions and clients.

Since it is only a proof of concept, the mapping of the complete functionality of the OSN was not the highest priority. However, again, this was taken into account by considerations and decisions, how for example data formats can be arranged so flexible that every function can be mapped.

Also, the focus was on compliance with the quality goals and implementation of the functional and non-functional requirements as defined in the previous chapter. How well this has been achieved will be evaluated in the following chapter for evaluation and discussion.

5.2 Selection of the OSN

When selecting a suitable OSN for the development of a hybrid client, Facebook was the obvious first choice due to the numerous negative headlines about data protection. With over two billion users per month, it is currently the most widely used social network in the world. In the recent past, it has often been criticized for its handling of its users' data. In particular, the scandal surrounding the data analysis company Cambridge Analytica, which had access to the data of up to 87 million users, hit Facebook hard. As a result, CEO Mark Zuckerberg had to face the US Congress and the EU Parliament in question rounds and did not leave a good impression

by avoiding many questions. As a result of this scandal, there were further restrictions to the Facebook API.

However, the Facebook API is not suitable for developing a new client. The functionalities offered by the API offer the possibility to develop an app that can be used within Facebook, for example for a game. So, it is not possible to make a like for a post through this API, which is part of the core functionality of a Facebook client. As discussed in Chapter 4, it is possible to access the data through crawling. However, the constant and rapid development would make this an arduous undertaking. Facebook writes in a blog post[14] that the code changes every few hours. Therefore it is almost impossible to adjust the crawler fast enough and roll out the adjusted code.

Even the mixed version of displaying and manipulating the mobile website in a WebView in a container app does not seem to be an option due to the short release cycles and frequent changes. Apps like „Friendly for Facebook“ do not manage to keep up with the changes as reported in various user ratings on the Google Play Store. The false representations and bugs worsen the user experience and result in the frustration of users.

For this number of reasons, Facebook dropped out as an OSN candidate for the prototype despite the particular interest. As a further candidate, the OSN Google Plus was dropped, as Google announced in October 2018 that it would discontinue its OSN[18].

In the end, Twitter was chosen for the prototype. With 336 million active users per month, it is one of the largest social networks. It is particularly well suited for the development of a hybrid client for two reasons: on the one hand, it has a comprehensive API that provides almost full functionality free of charge, and on the other hand, compared to Facebook, it offers only a few simple functions. These are the ideal prerequisites for the first proof of concept.

Twitter offers different APIs for developers that serve different purposes. The current APIs are:

- **Standard API**: the free and public API offering basic query functionality and foundational access to Twitter data.
- **Premium API**: introduced in November 2017 to close the gap between Standard and Entrprise API. Improvements over the Standard API: „more Tweets per request, higher rate limits, a counts endpoint that returns time-series counts of Tweets, more complex queries and metadata enrichments, such as expanded URLs and improved profile geo information“[1]. Prices to use this API start at 149$/month.
- **Enterprise API**: tailored packages with annual contracts for those who depend on Twitter data.
- **Ads API**: this API is only of interest for creating and managing ad campaigns.
- **Twitter for websites**: this is more a suite of tools than an API. It's free to use and enables people to embed tweets and tweet buttons on their website.

[1] https://blog.twitter.com/developer/en_us/topics/tools/2017/introducing-twitter-premium-apis.html

In the case of the hybrid client, the standard API is the right one. For using the API, first, the registration of a „Twitter App“ is necessary to receive a consumer key and access token. These two authentication tokens are required to log in users via the hybrid app and successfully communicate with the API.

Twitter offers almost the entire range of functions via the API. The missing functionality (e.g., the targeted retrieval of replies to a tweet) is not so critical for building a client app. A significant limitation is a restriction on the number of requests. Twitter argues that this restriction is necessary to avoid the exposure of the system to too much load. It also aims to prevent bots from abusing Twitter. The exact limits can be found on a help page[2]. In the app stores of Google and Apple, there are a number of alternative Twitter clients (Twitterific[3], Talon for Twitter[4], Fenix 2 for Twitter[5]), which are also subject to these restrictions in terms of functionality and request restrictions.

The API can be accessed using HTTP requests. The data exchanged are in JSON format. Furthermore, there are also various libraries (e.g., Twit[6]), some of which are developed directly by Twitter (see Twitter Kit for Android[7] or iOS[8]) and simplify the use of the API.

5.3 Technology Decisions

In order to implement the concept in the best possible way, some technology decisions had to be made. These included the implementation of the data exchange on a peer-to-peer basis and the choice of a framework for the development of the app. In the following, different technologies and libraries are presented that were considered for the implementation.

5.3.1 Peer 2 Peer Network

Basically, there are two conceivable approaches for the P2P network:

- Creation of a separate P2P network between the hybrid clients
- Use of an existing P2P network for own purpose

In the following, solutions for both ways are presented and compared with each other.

2 https://developer.twitter.com/en/docs/basics/rate-limits
3 https://itunes.apple.com/de/app/twitterrific-5-for-twitter/id580311103?mt=8
4 https://play.google.com/store/apps/details?id=com.klinker.android.twitter_l
5 https://play.google.com/store/apps/details?id=it.mvilla.android.fenix2
6 https://github.com/ttezel/twit
7 https://github.com/twitter/twitter-kit-android
8 https://github.com/twitter/twitter-kit-ios

Creation of a P2P Network

The advantage of having an extra P2P network is that it is completely under control. Accordingly, it can be designed to fit exactly to the use case and require little or no compromise. However, setting up a P2P network is a big challenge and some hurdles must be overcome. These challenges include peer discovery (how peers find each other), global data exchange over the Internet and data storage, and availability of the stored data. And of course, all these requirements must scale. It should work for P2P networks with only a few peers and also for a few thousand or even more peers. Two approaches are conceivable here: the use of an established standard such as Wi-Fi Dircet or WebRTC or the use of a library (for example Hive2Hive, Y-Js) to create a dedicated P2P network.

Wi-Fi Direct

Wi-Fi Direct is a standard (IEEE 802.11) for data transmission between two WLAN terminals. There is no need for an access point between the two devices. However, the distance that may lie between the two peers is thus limited. Without obstacles, which contribute to the attenuation of the signal, a distance of up to 95m is possible. In buildings, it sinks to 32m or less. However, the requirement for the P2P network to expand Twitter is different. Users can be scattered around the world and may be online through the mobile network. There is no P2P connection via Wi-Fi Dircet in this case.

WebRTC

WebRTC (Web Real-Time Communication) is an open standard that provides various communication protocols for real-time communication between two or more peers. Above all, WebRTC is popular for its ability to easily perform video calls, as known from Skype, in the browser without a server. But also the file transfer between peers is possible. In a separate P2P network, data could be exchanged between the clients. However, the connection between the clients is complex, since in addition to the two peers also a STUN server and optionally a TURN server is involved. Furthermore, it is unclear whether the system scales. At least for Google Chrome browser is known that a maximum of 265 connections to other peers can be maintained in parallel.

Y-JS

The JavaScript library Y-JS describes itself as „a framework for offline-first P2P shared editing on structured data-like text, richtext, json, or XML.“ The library takes care of solving synchronization conflicts when editing distributed files. By choosing a connector, you can set the protocol for the communication between the peers.There is the possibility to use WebRTC, XMPP or Websockets, but with some connectors running a server is a prerequisite. Further extensions can be used to supplement a database and data types, but the focus here is on the joint editing of

data by multiple peers. For all connectors, the authors of the library recommend using an own server.

Hive2Hive

Hive2Hive is a Java library for „secure, distributed, P2P-based file synchronization and sharing“. There is no less promise than „a free and open-sourced, distributed and scalable solution that focuses on maximum security and privacy of both users and data“. In order to be able to use Hive2Hive globally via the Internet, it is necessary to operate at least one relay peer - five are recommended[9]. Since a permanent TCP connection between peer and relay peer is maintained, the power consumption is quite high. Constant TCP connection can be avoided by using Google Cloud Messaging (GCM). However, Google has already discontinued this service. And since the development of Hive2Hive was discontinued in March 2015 too, there is currently no solution to this problem.

GUN

GUN is a graph database that keeps a state in sync across multiple instances. GUN is written in JavaScript and unlike Hive2Hive and Y-JS, the project is still being worked on.

todo...

Using an Existing P2P Network

As mentioned in the previous section, setting up your own P2P network involves a number of challenges. If you use an already existing P2P network for your own purpose, these challenges can be elegantly avoided. For this purpose, however, a suitable P2P network must be found whose properties and possible uses meet the requirements of the hybrid OSN client.

In the following, various P2P networks are considered and examined for their usability for a potential deployment to extend an OSN.

Filesharing P2P Networks

Blockchain

IPFS

[9] https://github.com/Hive2Hive/Android/wiki/Guide-for-System-Admins

5.3.2 Application Framework

[Standard Android Java App vs. Ionic]

5.4 Building Block View

In this section, the context in which Hybrid OSN is located is first considered, and then a breakdown into the individual components is carried out. The function of the respective blocks is then described in more detail. Finally, the function of individual components in interaction is explained using the examples of displaying the home timeline and posting a new tweet.

5.4.1 Scope and Context

Figure 5.1 shows a black box view of which other systems Hybrid OSN communicates with via interfaces. The systems are:

- Twitter API
- Gun
- IPFS via Infura
- User

Infura[10] is a service that provides access to Ethereum and IPFS via a simple interface. Communication with the API happens using HTTP requests. The connection of IPFS in Hybrid OSN can thus be carried out in an uncomplicated way. The use of an additional system entails an additional risk typically. However, there is a JavaScript client for IPFS, which can be integrated into hybrid OSN and thus the dependency on Infura would be omitted. At present and for the development of the prototype, the decision was made to use Infura for reasons of simplicity. Infura can be used for IPFS free of charge and without registration.

5.4.2 White Box View

The used Ionic framework uses Angular in the core, in the concrete case of Hybrid OSN Angular 5.2 is used. Accordingly, the Hybrid OSN app is in principle an Angular application. The essential building blocks are components, pages, and providers. In the following, these components are described in detail and examples are given of where they are used in hybrid OSN.

[10] https://infura.io/

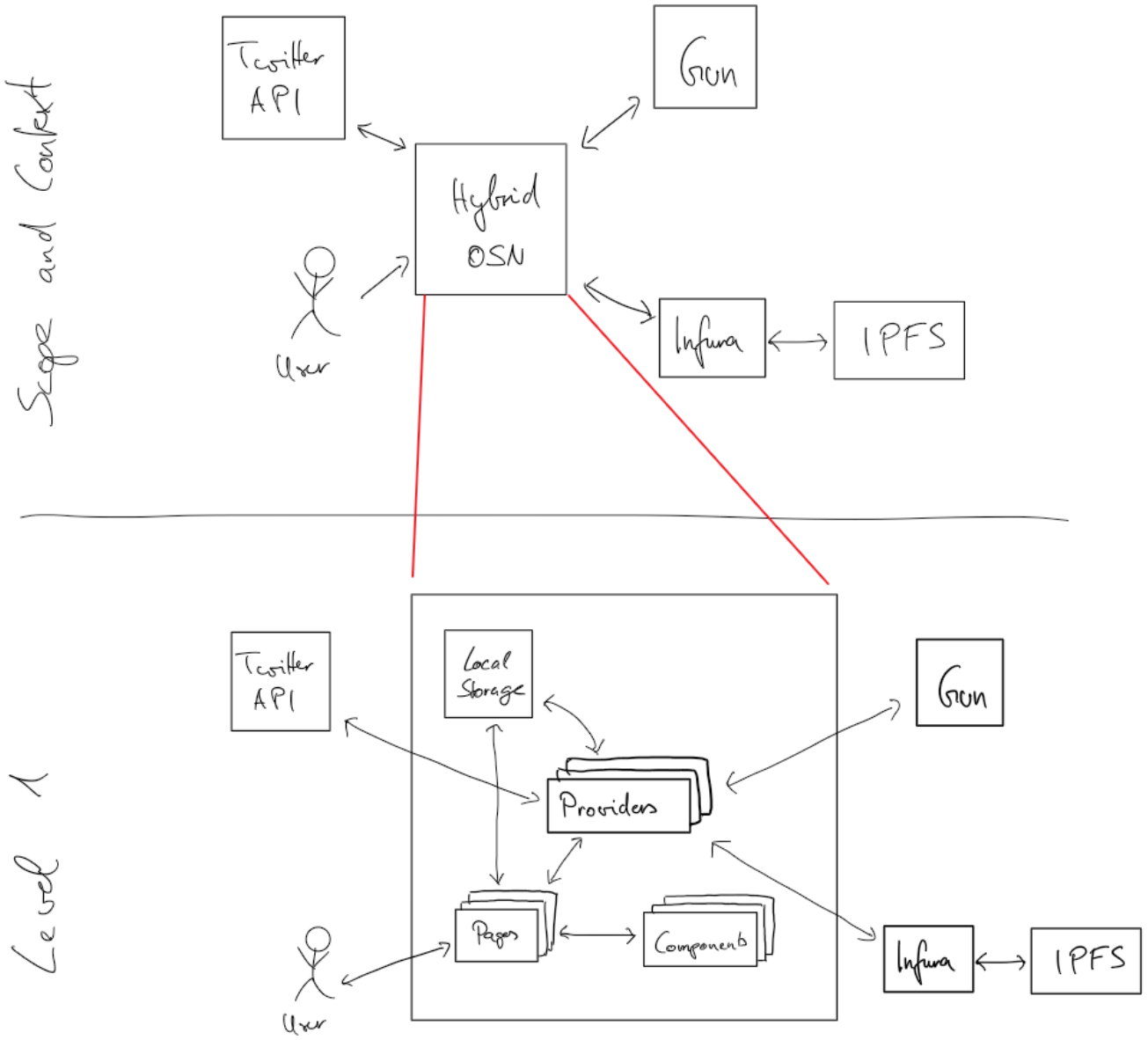


Figure 5.1: .

Providers

Data access is performed using providers (known as services in Angular). For the external services (Twitter API, P2P database, P2P storage), there is one provider each to handle the communication. Besides, providers are used as helper classes that provide specific functionality that is used several times. This functionality includes, for example, encryption and decryption and the compilation of aggregated timelines. Providers are injected into components using the constructor. Table 5.1 lists all providers used in hybrid OSN and their functional descriptions.

Provider	Purpose
Auth	Manage and perform authentication against the Twitter API. Responsible for login and logout.
Crypto	Provides methods for encryption, decryption, and key generation
Feed	Aggregation of private (P2P) and public (Twitter) tweets to compose a chronological timeline
P2P-Database-Gun	Interface for data exchange with Gun
P2P-Storage-IPFS	Interface for data exchange with IPFS via Infura
Twitter-API	Interface to use the Twitter API using the Twit package

Table 5.1: Providers used in the hybrid OSN app in alphabetical order with their purpose.

Components

Components are the basic building blocks of a user interface. Figure 5.2 shows an example of the representation of a tweet in Hybrid OSN using various components. A component consists of an HTML template, CSS styling, and JavaScript logic, whereby the logic is typically limited to a minimum. Components can be used as elements in other components or pages. A component receives the data it is supposed to visualize. Furthermore, components can process events or return them to parent components for handling.

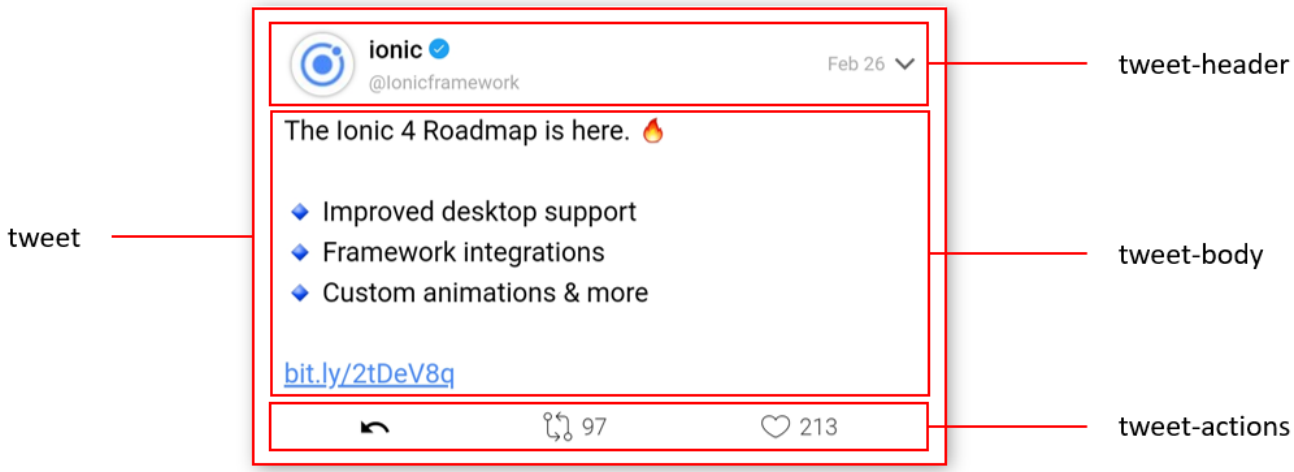


Figure 5.2: Composition of the tweet component from three other components. Several tweet components are in turn combined to form a feed component.

Pages

Pages are specialized components that are used as a holistic view. A page is made up of several other components. The data to be displayed is loaded using providers. To be able to navigate between the individual pages within the app, the model of a stack is used (implemented by the NavController). The currently visible page is at the top of the stack. When another page

is called, it is pushed onto the stack. Pressing „Back“ removes the top page from the stack and displays the page below it.

Table 5.2 lists all pages and their purpose. When the app is opened, it checks whether the user is already logged in. Depending on this, the user starts with the Login Page or the Home Page.

Page	**Purpose**
About	Information about the app, which can be accessed via the login page to get basic information about the app before logging in
Home	Chronological view of the latest tweets from Twitter and the private network
Login	Authentication against Twitter to use the Twitter API
Profile	Presentation of a user profile consisting of the user data (profile picture, brief description, location, website) and the user timeline
Search	Container page for searching for tweets and users, where tweets are also divided into popular and recent (see Search-Results-Tweets-Tab)
Search-Results-Tweets-Popular	Search results of currently popular tweets for a given keyword
Search-Results-Tweets-Recent	Search results of recent tweets for a given keyword
Search-Results-Tweets-Tab	Container page for the search results for tweets (recent and popular) in tabs
Search-Results-Users	Search results of users for a given keyword
Settings	Configuration of keywords that trigger the private mode and settings regarding encryption
Write-Tweet	Form for writing a tweet

Table 5.2: Pages used in the Hybrid OSN app in alphabetical order with their purpose.

Local Storage

As the name suggests, this is a local storage that is accessible by the app. With hybrid OSN, this memory is used to store essential information for usage. These include the Twitter user id, the two tokens for accessing the Twitter API, the keywords that trigger the private mode, and private and public keys for encryption. Log out completely deletes the local storage.

5.5 Runtime View

This section describes particular processes and relationships between building blocks for two selected scenarios. The first is to write and post a tweet and the second is to display the

home timeline. In addition to the description, the two scenarios are also documented by flowcharts.

The goal is to make clear how the building blocks of the system perform their respective tasks and communicate with each other at runtime. The two scenarios were therefore deliberately selected because they are of particular importance and require special documentation due to their complexity.

5.5.1 Post a Tweet

On the write-tweet page, new tweets can be written and posted using a form. In addition to the input field for the message text, there is also a status display that informs the user about the character limit, a switch that decides about the target network, and a button to send. The process of writing and publishing a tweet is shown in the flow chart in Figure 5.3. Figure 5.4 shows the involved components in the block view.

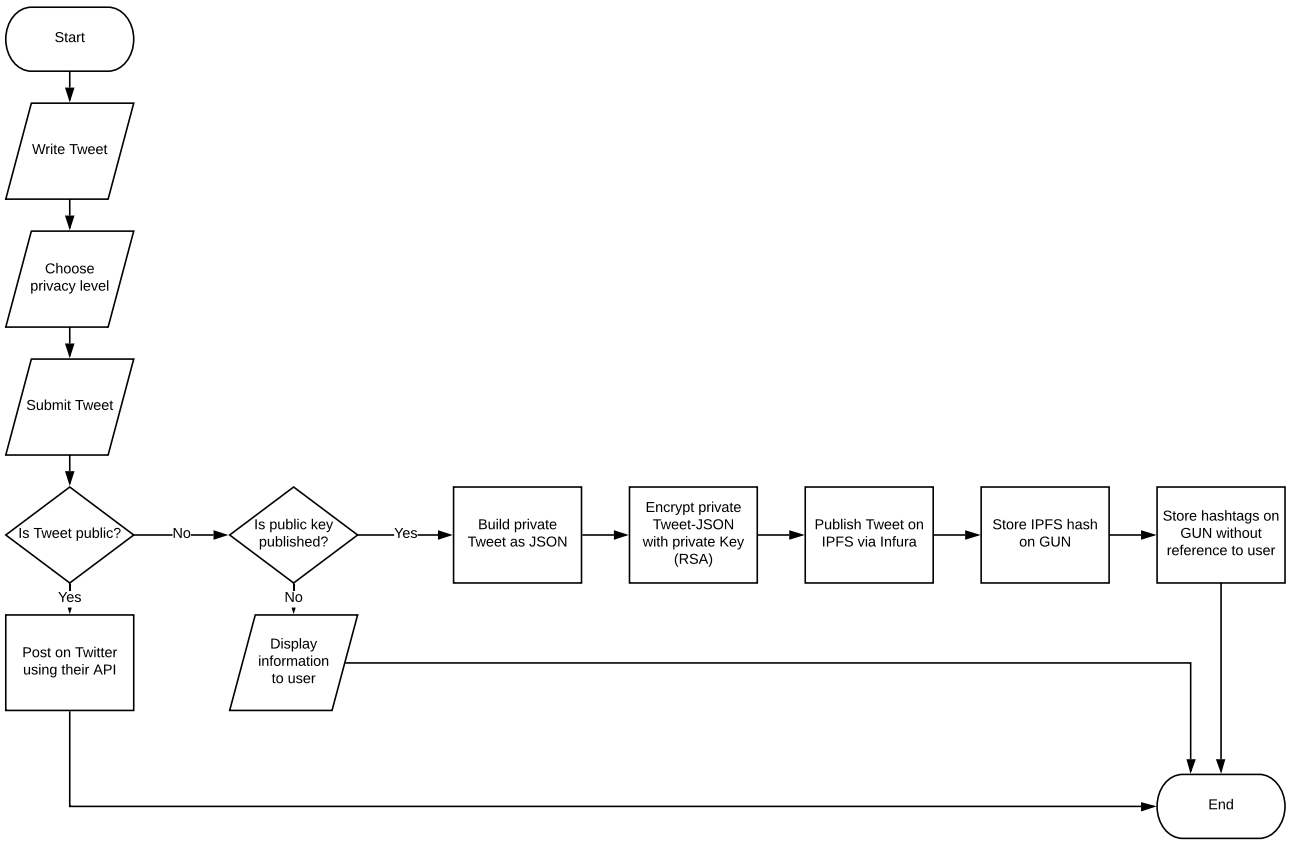


Figure 5.3: Flow chart of the process for posting a new tweet either on the public Twitter network or on the private P2P network

After entering the message in the input field, determining the destination network, and pressing the „TWEET!“ button, processing starts in the WriteTweetPage class. If the message is destined for Twitter, the Twitter API provider sends an HTTP POST request with the data to the

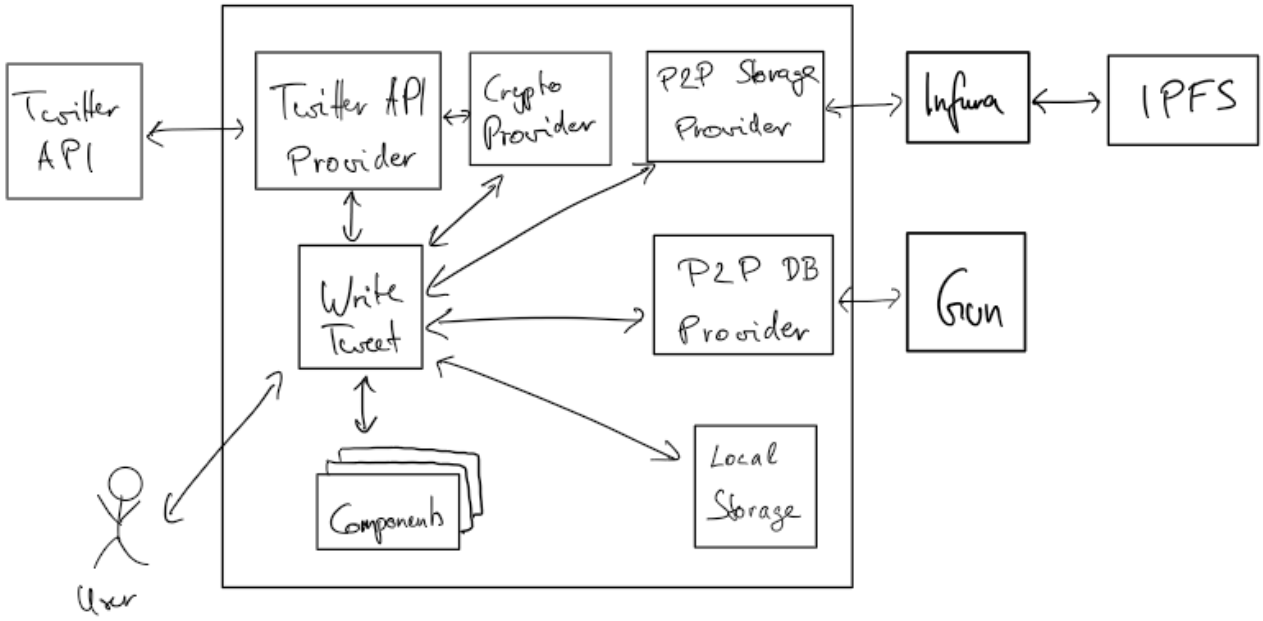


Figure 5.4: Building block view of the interaction between the different modules when posting a new tweet

`statuses/update`[11] interface. In this case, nothing more needs to be done, as the Twitter API takes over the preparation of the data and extracts, for example, hashtags, mentions, and URLs.

When publishing in the private network, the system first checks whether the public key has already been published. The Crypto provider performs this check using the Twitter API provider (for further information about the handling of public keys see the section about security 5.6). If the public key has not yet been published, the user receives a warning, and the posting process is aborted. Otherwise, the private tweet will be constructed. The entered text is converted into a simplified tweet object (see Twitter documentation for original tweet object[12]) that contains the essential information.

Beside the message (`full_text`) the Twitter user id (`user_id`) and the timestamp (`created_at`) are set. In addition, there is a flag (`private_tweet: true`) to distinguish the private tweet, which later influences the design. The `display_text_range` indicates the beginning and end of the relevant part in `full_text`. For private tweets, this is always the entire text, for tweets from the Twitter API an additional, not needed URL may be appended, which is cut off by clipping. Furthermore, the tweet entities are extracted. The extraction includes URLs, hashtags and user mentions. An example of a private tweet is shown in Listing 5.1.

Listing 5.1: Private Tweet in JSON format

```
1 {
2   "full_text": "Hello @twitter #hi",
3   "user_id": "3237049834",
4   "created_at": 1552290414930,
```

[11] https://developer.twitter.com/en/docs/tweets/post-and-engage/api-reference/post-statuses-update
[12] https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object.html

```
  "private_tweet": true,
  "display_text_range": [0, 18],
  "entities": {
    "hashtags": [
      {
        "hashtag": "hi",
        "indices": [15, 18]
      }
    ],
    "urls": [],
    "user_mentions": [
      {
        "indices": [6, 14],
        "id_str": "783214",
        "screen_name": "twitter"
      }
    ]
  }
}
```

The crypto provider performs the encryption of the private tweet data. For asymmetrical encryption, the RSA algorithm is used. The P2P storage provider sends the encrypted data via an HTTP POST request to Infura for storage in IPFS. The response contains the hash which addresses the data in IPFS. This hash is stored in Gun together with the timestamp and the author's Twitter user id. For saving to Gun, the P2P DB provider is used. Besides, the previously extracted hashtags with the timestamp are also stored in Gun with the same provider so that the data in the dashboard is accessible to the service provider without having to conclude individual users.

5.5.2 Load the Home Timeline

When opening the home page, the logged in user gets the latest tweets of the accounts he follows chronologically presented. The tweets are loaded in batches of 20 tweets from the Twitter API and enriched with the private tweets for this period. If the user scrolls to the end of the feed, the reloading of the next batch is triggered and automatically inserted at the end. At the top of the feed, a „pull to refresh“ action intents the feed reloading. The loading process is shown in Figure 5.5 as a flow chart and in Figure 5.6 as a building block view of the interacting components.

The starting point is the home page, which is accessed by the user. Several components display the data obtained from the feed provider. Using the Twitter API provider, the feed provider loads the latest 20 timeline tweets via the corresponding interface (`statuses/home_timeline`[13]) via an HTTP GET request.

The next step is to load the private tweets that match the period marked by the current timestamp and timestamp of the 20th (the oldest) tweet. Furthermore, the user ids of the users the user follows (so-called friends) are required. These must initially be requested from the Twitter

[13] https://developer.twitter.com/en/docs/tweets/timelines/yapi-reference/get-statuses-home_timeline.html

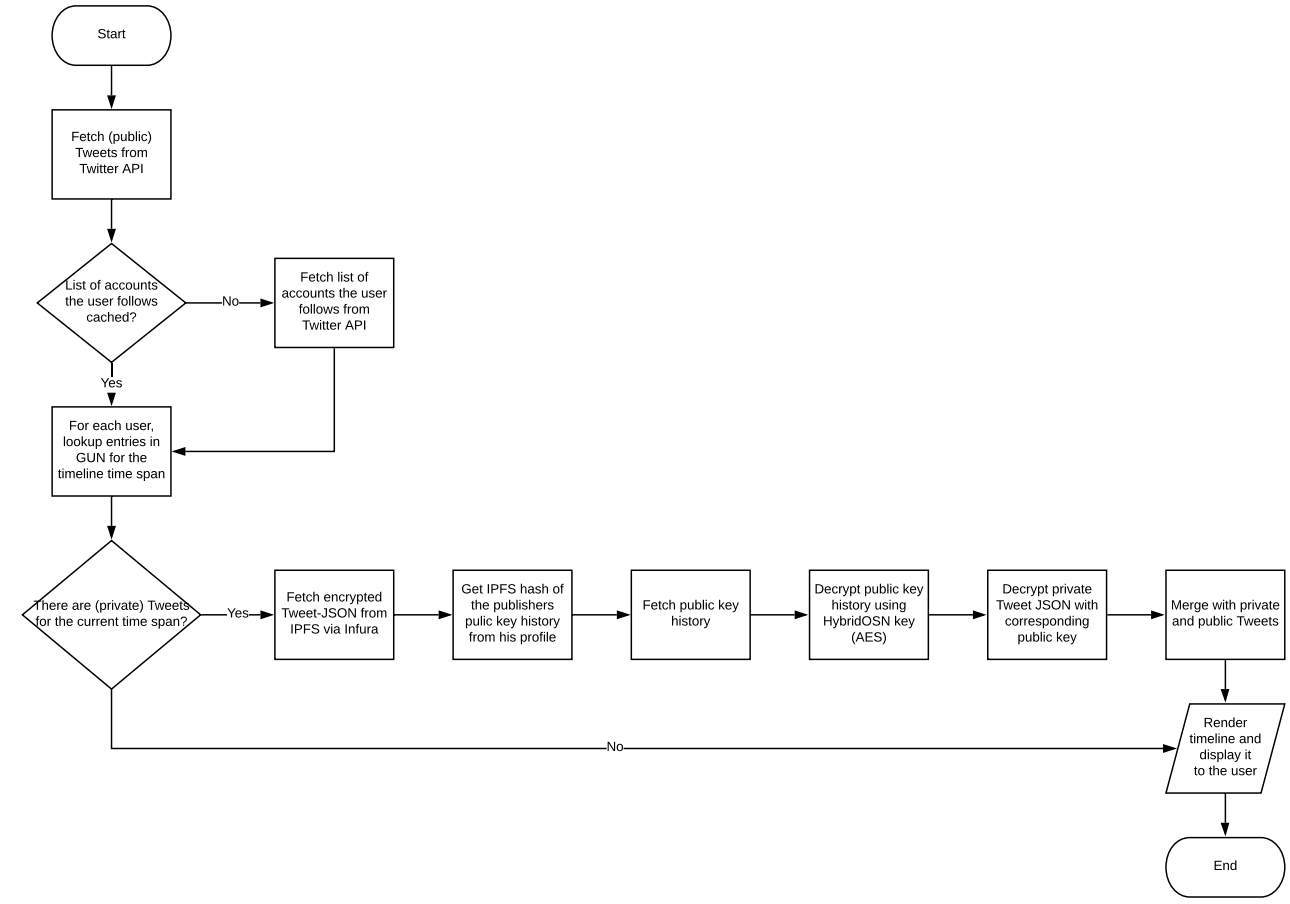


Figure 5.5: Flow chart of the process for loading the home timeline for a user

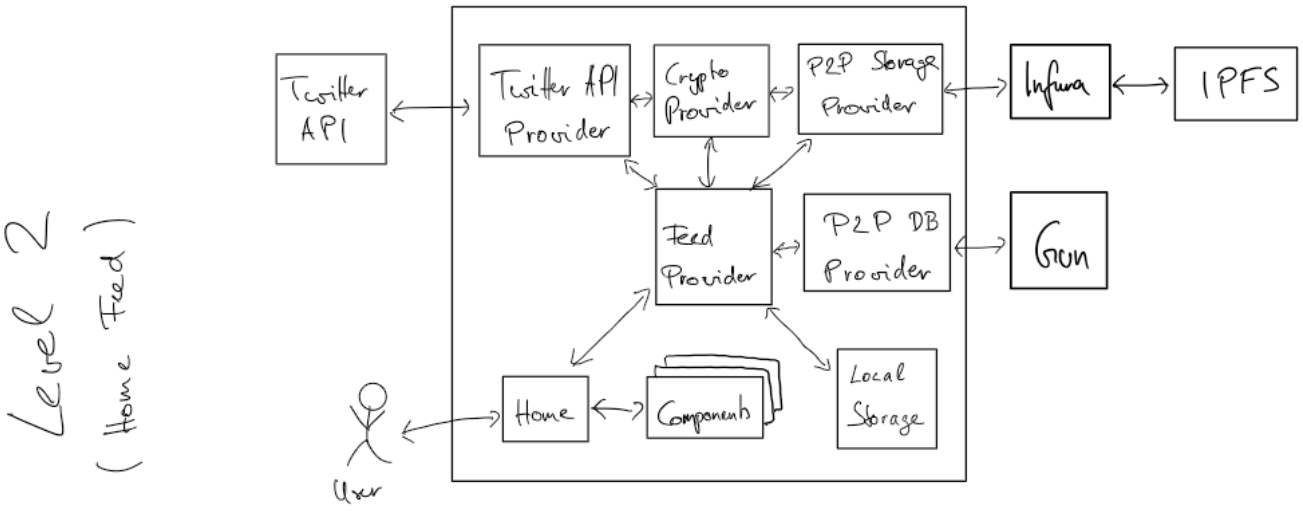


Figure 5.6: Building block view of the interaction between the different modules when loading the home timeline of a user

API (`friends/list`[14]) via the Twitter Feed provider using an HTTP GET request. The loaded

[14] https://developer.twitter.com/en/docs/accounts-and-users/follow-search-get-users/api-reference/get-friends-list

user ids are cached in order to keep the number of requests to the Twitter API to a minimum for later loading processes. For each user id, a lookup for private tweets in the given period is performed. The P2P DB provider queries Gun. If there are private tweets, the hashes for IPFS are returned together with the `created_at` timestamp. If no private tweets are available for the period, the feed provider returns the data of the public tweets to the home page. Otherwise, next, the private tweets are loaded and decrypted. First, the P2P storage provider is used to load the data behind the hash addresses from IPFS via Infura. For this purpose, the hash is transferred to Infura with an HTTP GET request, and the data is received from IPFS as the response. The author's public key, which can be obtained from the user's public key history, is needed for decryption. The public key history is loaded and decrypted via the Crypto provider, which in turn uses the Twitter API provider. Afterward, the private tweet is decrypted.

Finally, the private and public tweets are merged and sorted according to their `created_at` timestamp in descending order. This data is returned to the home page. If the user triggers a reload by reaching the end of the feed or by „pull to refresh“, the previously described process starts again.

5.6 Security

Usually, asymmetric or a combination of asymmetric and symmetric encryption methods are used for secure message exchange. The advantage of asymmetric encryption methods is that messages can be encrypted with the known public key and then only the owner of the private key can decrypt them. It is not possible to determine the private key from the public key within a reasonable time. However, asymmetric encryption methods are more computationally intensive and therefore more time-consuming than symmetric encryption methods. For this reason, a combination of both methods can be used first to exchange a symmetric key using asymmetric encryption. The asymmetric encryption ensures that only the two parties are aware of the symmetric key, which can then be used for symmetric encryption of the communication.

Tweets are usually posted publicly on Twitter. Only those who explicitly set their profile to „private“ can decide whom they allow as followers and thus recipients of their tweets. This type of publication and visibility should also apply to private networks. Since it is unclear who is the recipient of a private tweet, it is not possible to encrypt the message with the recipient's public keys.

The following three requirements apply to encryption:

1. The author is verifiable. It is not possible to distribute tweets on behalf of another user on the P2P network.
2. A private tweet should have the same visibility to other users as a standard tweet on Twitter.
3. The service provider (Twitter) must not be able to decrypt private tweets or associate them with a user.

Concrete actions can be concluded to meet these requirements:

1. Private tweets must be signed or asymmetrically encrypted so that the author is identifiable.

2. Distribution of the public key for decryption must take place via the user's profile. The profile is the only place that guarantees that only authorized users can access the key and that the public key belongs to a specific user without any doubt.

3. The Hybrid OSN application must encrypt the public keys so that Twitter cannot read them and therefore cannot decrypt the private tweets.

5.6.1 Realization

In the app settings, an asymmetric key pair can be stored or generated, which is used to encrypt the private tweets. The RSA-OAEP algorithm is used here. Furthermore, by clicking on a particular button, the public key is published. The new key together with the current timestamp is recorded in the public key history of the user and stored in IPFS. Listing 5.2 shows an example for the public key history. In JSON format, public key and validity start are stored in an array. The address at which the public key history can be retrieved is posted as a regular tweet in the user's timeline during publication. So that this tweet can be found quickly and easily; the id of this tweet is saved in the profile description of the user.

Listing 5.2: Public key history in JSON format. The file is symmetrically encrypted before storing on IPFS.

```
{
    "keys":[
        {
            "key":"MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCNZiscdaUuJ8pUFkPkTAh6oq...",
            "validFrom":1548591435051
        },
        {
            "key":"MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC+14hMkrRfSwCTfbmkS+m6MQ...",
            "validFrom":1548193838409
        },
        {
            "key":"MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC5rYHZljdPXozCNCX86N8CTY...",
            "validFrom":1546382473017
        },
        {
            "key":"MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCjssLPklHgNX/KyPACLOQAmB...",
            "validFrom":1545945585132
        }
    ]
}
```

By saving this information on the user's profile, it is ensured that only users who can read the user's regular tweets have access to the user's public keys and hence to his private tweets. The history allows the changing of the key pair and still ensure that older private tweets are still decryptable.

Since Twitter has access to all the data stored on its servers, it can also find the link to the public key history. Therefore, it is necessary to prevent Twitter from decrypting the private tweets of the user by retrieving the public key history. For this reason, the public key history is additionally encrypted symmetrically with the AES-GCM algorithm. The key is stored in the Hybrid OSN app and therefore unknown to Twitter.

When writing a new, private tweet, the system checks whether the public key has been published before posting. Only if this is fulfilled, the private tweet will be encrypted with the private key and posted.

5.7 Providing Insights to the Service Provider

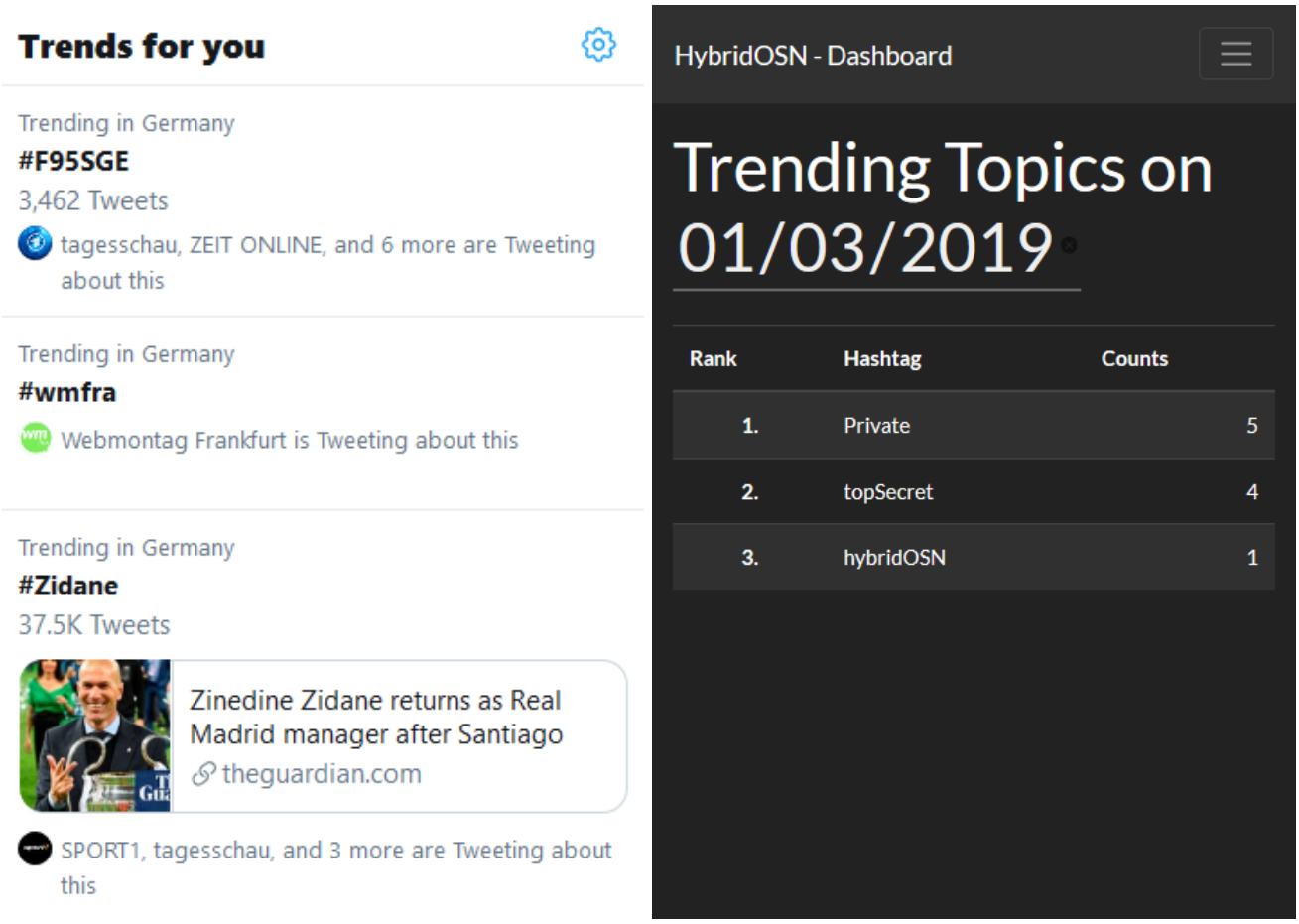
The requirements mentioned in 4.1 also include the provision of anonymized data for the OSN service provider. Since the business model of Twitter is based on personal data, and therefore the interests of hybrid OSN are contrary to those of Twitter, the fulfillment of this requirement is extremely complex.

A prominent feature of Twitter is the analysis and promotion of trends (see Figure 5.7a). The trends are identified through frequently used hashtags and presented in a ranking. Such data can also be collected and evaluated in the private network without having to establish a connection to the users.

To collect this information, when a new tweet is posted to the private network, the contained hashtags are extracted and stored separately (see flowchart Figure 5.3). Similar to the presentation of trends on Twitter, the trends in the private network are also aggregated on a daily basis and presented on a website (see Figure 5.7b).

Because Gun is JavaScript-based and therefore executable in the web browser, access to the data from a simple HTML web page can be performed using JavaScript code. The raw data is loaded and then aggregated and displayed.

5.8 Summary



(a) Twitter trends **(b)** Hybrid OSN trends

Figure 5.7: Trending hashtags in Twitter and the private network side by side

6 Evaluation

I will discuss,

- how the requirements defined in chapter 4 were met in the prototype,
- limitations,
- benefits in comparison with the projects mentioned in related work

6.1 Threat Model

In the threat model of Hybrid OSN the potential threats for different sub-areas are shown, and the particular risk discussed. The worst would be if private data could be decrypted and assigned to a user or if identity abuse were possible. However, other dangers such as identification by the service provider or manipulation of data must also be analyzed.

6.1.1 Service Provider – Twitter

Hybrid OSN users can be easily identified by the service provider Twitter, even if they only use Hybrid OSN passively to read private tweets of other users and do not write private tweets themselves.

For using the Twitter API, it is essential to register an app to get an app token. This app token is attached to all requests sent to the Twitter API. When logging in on Hybrid OSN for the first time, the user accepts to use the app to access Twitter.

So far not implemented, but theoretically possible is that each user creates an app for the use of the API on their own. The obtained app token could then be stored in the Hybrid OSN app, and the use of the application could be obscured. In this case, the identification possibility via the Hybrid OSN app token is omitted, and the passive use would be possible without danger.

Active use requires a public tweet and a reference in the profile description for the distribution of the public key history. Although the contents are inconspicuous, they are still sufficient for the identification of a Hybrid OSN user.

6.1.2 Gun

In Hybrid OSN, Gun takes the role of a database shared by the peers. The dashboard also establishes a direct connection. The data is publicly accessible and editable.

The stored data is a combination of hashtag and timestamp, which serve as information for the trends in the Hybrid OSN dashboard. For every private tweet of a user, there is an entry

consisting of Twitter user id, IPFS address hash, and timestamp. Also, there are the private likes, for which there is a counter to the tweet id.

For preventing the hashtag and private tweet timestamps from connecting, the time of the hashtag timestamp is set to 00:01. The trends in the dashboard are aggregated by the day, so the exact time is not essential.

The greatest threat is that an attacker may modify or delete data. By deleting entries, private tweets would no longer be found and thus no longer displayed. Changing the IPFS hash would mean that the data could not be found and would also not be displayed. Manipulation of the timestamp would result in private tweets being loaded at the wrong time interval when the feed is loaded and thus positioned at the wrong place. Furthermore, the timestamp in Gun is used to use the appropriate public key from the public key history for decryption. Under certain circumstances, the wrong key would be selected and the private tweet could not be decrypted.

Creating entries for private tweets does not have a significant effect because the associated content stored in IPFS must be encrypted with the private key, which is unknown to a third party. Adding wrong or modifying existing hashtag entries for trend detection is also possible and poses a significant risk as it allows manipulation of the trends. Ultimately, it is not possible to verify which hashtags were used and how often. The same applies to private likes. Since in this case the complete information is stored in Gun and can be changed, it is not possible to determine whether data has been manipulated.

6.1.3 IPFS

Since IPFS is publicly accessible, anyone can add and retrieve data. However, it is not possible to change or delete data. A hash of the content addresses the data stored in IPFS. Since the content is entirely unknown (especially by encrypting the plaintext content), it is not possible to conclude the hash. A targeted search for private data in IPFS is therefore impossible. The encrypted data also does not contain any clues that allow conclusions to be drawn about Hybrid OSN.

In combination with the publicly available information from Gun, all private tweet data could be found in IPFS. However, because of the encryption of the content, the data is worthless.

Due to the decentralization of the system, the availability of IPFS is always guaranteed. However, only as long as there are peers who make the service possible. If a peer leaves the network, its data is also lost if not reproduced beforehand. Therefore, there is no guarantee for the permanent availability of data.

6.1.4 Encryption – Leakage of Keys

On the one hand, the public key history is symmetrically encrypted; on the other hand, the private tweets are asymmetrically encrypted. The keys for asymmetric encryption are generated

independently by each user and are therefore individual for each user. With symmetric encryption, only one key is used, which is stored in the source code of Hybrid OSN. In this way, only the Hybrid OSN app can decrypt the public key history of a user and therefore decrypt its private tweets.

Disclosure of the source code would reveal the symmetric key. The service provider would then have all the necessary information and access to all data to read private tweets and assign them to users.

6.1.5 Authorized Access

A user's private tweets should be readable by all users who can also read the public tweets - except for the service provider Twitter. Therefore, a user posts the IPFS hash that leads to the public key history on his timeline.

There is a threat that authorized users may create a copy of the decrypted public key history and pass it on to third parties. Since the data in IPFS is permanent and therefore not erasable, it can be decrypted at any time later with the appropriate public key.

If a user decides to change his profile to „private“ in the account settings, the profile will no longer be publicly accessible. Only accepted followers should then be able to read public and private tweets. A non-approved twitter user is still able to fetch the encrypted private tweets from IPFS. However, since the link to the public key history is no longer accessible, the private tweets decryption is not possible. If non-approved users or third parties already have the link to or a backup of the public key history from the past, all private tweets of the past can still be decrypted. Whenever a profile is changed to „private“ a new pair of keys should be generated to ensure that future private tweets are only readable to approved users.

7 Conclusions

7.1 Summary

7.2 Contributions

7.3 Future Work

List of Figures

List of Tables

Bibliography

[1] diaspora* federation protocol. https://diaspora.github.io/diaspora_federation/index.html.

[2] FAQ for users - diaspora* project wiki. https://wiki.diasporafoundation.org/FAQ_for_users.

[3] Magic Signatures - diaspora* federation protocol. https://diaspora.github.io/diaspora_federation/f

[4] Eine neue Ära beginnt: diaspora plant mit Y Combinator den großen Neustart, May 2012. https://www.foerderland.de/digitale-wirtschaft/netzwertig/news/artikel/eine-neue-aera-beginnt-diaspora-plant-mit-y-combinator-den-grossen-neustart/.

[5] Tim Berners-Lee. 30 Years On, What's Next #ForTheWeb?, March 2019. https://onezero.medium.com/30-years-on-whats-next-fortheweb-6ce844ed147f.

[6] Maxwell Salzberg Daniel Grippi. Announcement: Diaspora* Will Now Be A Community Project, August 2012. https://web.archive.org/web/20121109114722/http://blog.diasporafoundation.org/2012/08/27/diaspora-will-now-be-a-community-project.html.

[7] Jörg Daubert, Mathias Fischer, Stefan Schiffner, and Max Mühlhäuser. Distributed and anonymous publish-subscribe. In Javier Lopez, Xinyi Huang, and Ravi Sandhu, editors, *Network and System Security*, pages 685–691, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[8] Jörg Daubert, Leon Bock, Panayotis Kikirasy, Max Mühlhauser, and Mathias Fischer. Twitterize: Anonymous micro-blogging. In *2014 IEEE/ACS 11th International Conference on Computer Systems and Applications (AICCSA)*, pages 817–823. IEEE, 2014.

[9] David Johnston, Sam Onat Yilmaz, Jeremy Kandah, Nikos Bentenitis, Farzad Hashemi, Ron Gross, Shawn Wilkinson, and Steven Mason. The General Theory of Decentralized Applications, Dapps, February 2015. https://github.com/DavidJohnstonCEO/DecentralizedApplications.

[10] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[11] Dmitri Moltchanov. Selected DHT algorithms: Chord and Pastry, October 2014.

[12] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A scalable content-addressable network*, volume 31. ACM, 2001.

[13] Siraj Raval. *Decentralized applications: harnessing Bitcoin's blockchain technology*. O'Reilly Media, Inc., 2016.

[14] Chuck Rossi. Rapid release at massive scale - Facebook Code, August 2017. https://code.fb.com/web/rapid-release-at-massive-scale/.

[15] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[16] Maxwell Salzberg. Kickstarter Pitch, April 2010. https://web.archive.org/web/20110814222702/http://blog.joindiaspora.com/2010/04/27/kickstar pitch.html.

[17] Maxwell Salzberg. We Made It!, May 2010. https://web.archive.org/web/20110713115706/http://b did-it.html.

[18] Ben Smith. Project Strobe: Protecting your data, improving our third-party APIs, and sunsetting consumer Google+, October 2018. https://www.blog.google/technology/safety-security/project-strobe/.

[19] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[20] Christopher Lemmer Webber, Jessica Tallon, Erin Shepherd, Amy Guy, and Evan Prodromou. ActivityPub - W3C Recommendation, January 2018. https://www.w3.org/TR/activitypub/.

[21] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, 22(1):41–53, 2004.