POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica - Data Science

Tesi di Laurea Magistrale

Supporting the portability of profiles using the blockchain in the Mastodon social network

Relatore
prof. Giovanni Squillero

Candidato
Alessandra Rossaro

Anno Accademico 2018-2019

UCLouvain

epl

École polytechnique de Louvain

Supporting the portability of profiles using the blockchain in the Mastodon social network

Authors: **Alessandra Rossaro, Corentin Surquin**
Supervisors: **Etienne Riviere, Ramin Sadre**
Readers: **Lionel Dricot, Axel Legay, Giovanni Squillero**
Academic year 2018–2019
Master [120] in Computer Science

Acknowledgements

We would like to thank anyone who made the writing of this thesis possible, directly or indirectly.

First of all, we would like to thank our supervisors, Prof. Etienne Riviere and Prof. Ramin Sadre for their continous support and advice during the year. We would never have gone this far without them.

Secondly, we thank Lionel Dricot, Prof. Axel Legay and Prof. Giovanni Squillero for accepting to be the readers of this thesis.

Alessandra First of all, I would like to thank my family, my parents Claudia and Alberto, my brother Stefano and my sister Eleonora, that from the beginning of my studies believed in me, every time urging me to give more and sustaining me each time that I had difficulties. They are my strength and I feel really lucky to have them in my life. Another thanks is to my friends, to Soraya, Beatrice, Sinto and Stefano and especially to Matteo and Edoardo that each time that I needed, remember me to believe in myself and don't give up. Thank you, sincerely! I would like to thank also my partner, Corentin, because we were a great team, sometimes with some misunderstandings, but I appreciated to work at this project with him!

Corentin I must express my deep gratitude to my family and friends for their moral support. I have particular thoughts for my partner Asseline for her faith in me, her patience and her reassuring words and presence during the worst moments. I would like to thank her parents and her sister as well. They were always present for me and have been a great help until the end. Finally, I thank my parents who never let me down and allow me to end my studies at University. Thank you.

Abstract

This Master Thesis has the objective to present a possible solution, using the Blockchain, to solve the fact that Mastodon social network does not support the portability of profiles. Mastodon is one of the most famous DOSN - Decentralized Online Social Network, free, open source, that offers a microblogging service as Twitter. Mastodon supports the interoperability and portability of data across sites, using the ActivityPub open protocol. This allows users connected to one site to follow the users connected to other sites, and to share posts across communities, but if for some reasons, a user decides to migrate to another instance, he/she has to recreate a new account on the new instance with, for example, the unpleasant consequence of loss of followers. Our solution is a prototype based on Hyperledger Fabric, that uses a microservice-like approach. A Server module provides a RESTful API and acts as a gateway between the Mastodon application and the Fabric network, thus the management of certicates and the connection to the network are independent of the Mastodon application. This makes the implementation easier to maintain on the long term. This solution allows the users to register to different instances of Mastodon with the credentials stored on the Blockchain, to perform the Login and other few operations. This works includes also a dissertation about the Decentralization, the Mastodon social network and the Blockchain.

Contents

Chapter 1

Introduction

1.1 Context of the thesis

In a context where the improper use of personal data by centralized social networks such as Twitter or Facebook is making the headlines (we remember the scandal that involved Facebook and Cambridge Analytica on the 17th March 2018, when the New York Times reported that in 2014 contractors and employees of Cambridge Analytica, eager to sell psychological profiles of American voters to political campaigners, acquired the private Facebook data of tens of millions of users — the largest known leak in Facebook history [1][2][3]), social networks based on the principles of decentralization, openness and privacy are gaining significant interest. These online social networks (OSN) do not centralize users' data (posts, comments, links) in a single data center but rather federate a number of decentralized sites, operated by different administrative entities. In this kind of social network data are no more under the control of one single entity. They follow management rules focusing on the respect of users' privacy and control over their data – as opposed to Twitter or Facebook whose business models depend on the collection of these data.

Mastodon [4] is one of the most popular decentralized OSN. It is a free and open-source self-hosted social networking service which has microblogging features similar to Twitter. It reached 2 million users in May 2019. [5] It is a federation, which means that thousands of independent communities running Mastodon form a coherent network.

1.2 Problem definition

Mastodon supports the interoperability and portability of data across sites, using the ActivityPub open protocol [more details at 2.2.3]. This allows users connected to one site to follow the 'toots' (the name of posts in Mastodon) of users connected to other sites, and share these toots across communities. However, it does not support the portability of profiles: if for some reason a user wishes to switch to another site (e.g. because the rules for the site he uses have changed), he has to recreate a full new profile on another instance, with his connections (following of other users), parameters, and posted toots and replies. In practice, if a user has an account X on one instance and an account Y on another instance, this user will have to either use X to follow some people and Y to follow others, or randomly interact with some things from X and others from Y depending on which the user opens first. Those options are disorganized or create an additional burden for the user.

In addition this issue also arises whenever a site is closed (due to a server failure, or to a decision of its owner). The lack of profile portability leads to a tragedy of the commons, where users tend to connect preferentially to large instances of Mastodon. This leads to a new form of centralization that goes against the decentralization principles of the platform. In fact, their goal is to have users separated out across multiple federated instances.

Another issue tied to these problems is the "supporting username". In fact, if a user wants to migrate his account from Mastodon.xyz to another instance, he might want to keep the same username, but if it is already used by another user, it is not possible to choose it.

1.3 Organization of this work

This Master Thesis is structured in six chapters:

1. Introduction where we describe in details what is the Identity Migration issue, the state of the art regarding the use of the Blockchain for the Identity Management, the goal of the thesis and we report some solutions proposed by the community to solve this issue. We conclude the chapter with a brief description of the current situation and the additions that were made in these years to fix the situation and our proposed solution.
2. Technical Background, in which we explain what Decentralization is, we describe Mastodon in all its aspects and we conclude the chapter talking

about the Blockchain to help the reader to understand the technology that we used in our work.

3. Design and Implementation, this is the main chapter of this thesis in fact here we describe what we did more in details, concentrating on the several parts of the project: Mastodon side, Server side and Blockchain side. We conclude the chapter with a comparison of the processing times of Login action, Sign up action, Change password action and Update biography action in case we use the Blockchain and in case we do not use it.
4. Further Development in which we sketch out what implementations could be developed in the future to improve our solution.
5. Related Works that use the technology of Blockchain in the social network.
6. Conclusions

There are also four appendixes:

- A - Definitions, where we collected short definitions of concepts mentioned in the thesis.
- B - Solutions proposed by the community, a collection of the main ideas. In the first chapter we tried to summarize and report only few solutions, letting the reader free to deepen the topic, if he is interested.
- C - Mastodon side, where the files with the code that we modified in Mastodon are reported.
- D - Blockchain side, where the smart contract (chaincode) that we wrote to interact with the Blockchain is reported.

1.4 Identity Migration

In the previous section we described how the identity migration represents an issue: current email, identity and file migration solutions fail to provide a fully robust migration of users to new email, identity and file platforms [6]. These solutions have a difficult time identifying individuals whose data and/or identity should be migrated together. The most common way to solve this problem is by tabulating a master object (for example, mailbox or identity object) list through manual lists and spreadsheets, but it is very difficult to manually manage a spreadsheet that may have many thousands of rows. Plus it is very difficult to collate an accurate and appropriate list of objects for each migration, when the necessary relationship information between mailboxes (or other types of migrating objects) is not included in the spreadsheet or even readily available. Thus, it is difficult to determine which objects should migrate together to minimize the impact on the organization.

Consequently it is necessary a standard for solving the identity migration, because domains are a fragile identity system. Particularly in a world where it is more and more obvious people do not want to set up their own domain for their own identity. To achieve a distributed identity migration you need:

- proof that the user owns the original account (accounts on Mastodon and other OStatus [A] sites should already have a private key because that is how the Salmon protocol [A], used to coordinate discussion between members belonging to different servers, works)
- a list of current identities that subscribe to user's feed (the followers in this case, who should already have user's public key)

Then the user needs to update those subscribers by sending a notification, probably a profile-update style notification that is signed by his original key or the original site could send the signed notification for the user (signing a new public key with the former private key and sending from the original server is a pretty reasonable practice, when it is possible). For these reasons a complete identity migration requires:

- all instances remember the public keys of off-site identities
- ability to export identity (which you can do as a backup precaution if your current Mastodon instance is destroyed)
- ability to export followers list
- ability to import identity
- ability to import followers list

- ability to send a signed notification

The described issue is what we tried to solve with our work. We found interesting and very challenging this topic because, as daily users, we noted that it represents an obstacle, not only in the social networks' environment, but in several applications: each time that you have to register on a website, for instance. It would be useful to have a way to automatically insert credentials and all the information that a user retains necessary for the specific situation. Thanks to the new technologies that are present nowadays, it becomes possible. The challenge is to select the best solution that fits this kind of unpleasant condition.

1.5 State of the art

Nowadays there are several solutions that use the Blockchain to solve the Identity Management: in the real world there exist some systems for establishing personal identity, for instance the identity documents, driver's licenses, passports, ... but in the online world there is no equivalent system for ensuring the personal identities or the identity of digital entities. The two problems to solve when we talk about the verification of digital credentials are the two following [7]:

- Format Standardisation: the machine that reads the digital credentials needs to be able to understand the format in which the credentials are written.
- Standard way to verify the source and integrity: the digital signatures are composed by a public key and a private key that are cryptographically linked (in this way every private key has only one public key and viceversa). It is necessary to find a standard way to verify the public key of the issuer, which would then prove the authenticity of the credential. The current way to perform this activity, that involves a PKY - public key infrastructure, is complex, costly and centralized. In fact a user that has a private key gives his public key to a CA (certificate authorities) who signs it with their own private key and issues a public key certificate, but certificates from reputable CAs take real time and effort to obtain and the centralization can lead to a single points of failure.

Blockchain technology, without the need of a trusted, central authority, may offer a secure solution, creating an identity on the Blockchain, a user could simply use an application for the authentication instead of using traditional methods (as username and password). For individuals, it could be easier to manage who has their personal information and how they access it, according to their own terms. A digital ID can be created and used as a digital watermark, assigning it to every

online transaction and allowing the organizations to check the identity on every transaction in real time.
Among all the solutions, according to our problem and our personal opinion, the most interesting solutions present on the market are the following ones:

- **Blockstack** [8]: "Blockstack ID provides user-controlled login and storage that enable the user to take back control of his identity and data. Creating a Blockstack ID is easy, free, and secure. This Blockchain implements services for identity, discovery, and storage and can survive failures of underlying blockchains. Blockstack gives comparable performance to traditional internet services and enables a much-needed security and reliability upgrade to the traditional internet."

- **Sovrin** [7]:"The Sovrin Network is the new standard for digital identity – designed to bring the trust, personal control, and ease-of-use of analog IDs – like driver's licenses and ID cards – to the Internet. They have designed Sovrin as a metasystem for any application to use, giving people, organizations, and things the freedom to prove things about themselves using trustworthy, verifiable digital credentials. The Sovrin blockchain has been designed ONLY for self-sovereign identity and verifiable claims. "Self-sovereign" means the individual identity holder controls their credentials, using them whenever and however they please, without being forced to request permission of an intermediary."

- **uPort** [9]: "uPort is a secure system for self-sovereign identity, built on Ethereum. It represents the next generation of identity systems: the first identity system to enable self-sovereign identity, allowing the user to be in complete control of their identity and personal information. uPort identities can take many forms: individuals, devices, entities, or institutions."

- **DIONS - I/O Digital** [10]: "DIONS is a fully AES 256, encrypted, decentralized name system, messaging, data storage and a decentralized "GPGTOOLS"-like system which offers a distinct advantage over Bitcoin. The DIONS Blockchain also enables identity storage, avatar creation and encrypted document storage capabilities that are transferable between users. All the features are readily hard coded into a user-friendly HTML5 wallet system."

- **MyData** [11]: "The term MyData refers 1) to a new approach, a paradigm shift in personal data management and processing that seeks to transform the current organization centric system to a human centric system, 2) to personal data as a resource that the individual can access and control. The aim is to provide individuals with the practical means to access, obtain, and use datasets containing their personal information, such as purchasing data,

traffic data, telecommunications data, medical records, financial information and data derived from various online services and to encourage organizations holding personal data to give individuals control over this data, extending beyond their minimum legal requirements to do so. "

There is a consistent number of Blockchains, thought to solve the Identity Management and in our opinion there will be others in the future, since every day, anyone has access to Internet and he/she is requested to use digital credentials.

1.6 Goals

This work has the aim to find a way to solve the Identity Migration issue, to let people that already have an account registered on an instance of Mastodon, to create a new account on a new instance, leveraging the information (as basic personal information, toots, favorites, followers, ...) that are already present in the previous account. This should allow users;

- to move an account to another instance
- to create a new account on another instance in both cases: when a user wants to join to another instance, that maybe fits better his interests or in case the server goes down, as a sort of backup that makes easier the migration.

Since the structure of data in Mastodon was already defined, we did not evaluate a solution to solve the issue related to the uniqueness of username. If a username is no more available on the instance he wants to register on, at the moment, he has to use another one, but continuing to use the same email as on the other account which is the first parameter to be checked: when you decide to sign up on a new instance a test is performed on the presence of the email within the instance's Database . If the email is already present, Mastodon informs the user about his presence on that instance. In addition if the username that the user chose is already in use, Mastodon reports an error and it asks the user for inserting a different username.

1.7 GitHub open issue

The open issue has involved several members of the community since 2016, finding a good solution took them a long time because of the load involved in switching, both cognitive and technical.

"People could be worried about impersonation, but signaling that an identity is the same is the easy part: servers can authenticate with each other quite easily and do so all the time, for instance using two factor authentication.

The hard part concerns porting the content and followers:

- content is tied to URLs, which are necessarily going to change as part of the move;
- followers have to be updated of the move, which puts strain on all the remote servers to update their links and databases as well.

Porting original post seems quite easy and it would not necessitate a full migration, whereas porting Boosts and Favourites is going to be tricky because you have to redeliver potentially thousands or millions of Announce/Like activities. Thus, they proposed different solutions".

The reported solutions have been taken from the open issue #177 about *supporting account migration*[1]. Here are the most relevant ones that address the problem of a full migration. They are in a chronological order.

According to the community "Mastodon needs to support two things:

- Account import across providers, which should be authenticated on both ends to prevent people bulk copying another person's account.
- A "redirected account" field against user accounts, which indicates the full URL of the new user account.

When users configure a redirect location against their account the instance on which the account is, should implicitly and automatically redirect followers. The hardest part is porting the content and the followers: content is tied to URLs, which are necessarily going to change as part of the move and followers have to be updated of the move, which puts strain on all the remote servers to update their links and databases as well".

In a row, there are collected the main ideas of solutions proposed by the community:

[1] `https://github.com/tootsuite/Mastodon/issues/177`

- Mastodon Name Resolver (MNR): *"For the unique-usernames we would have several ways to go. In order to create unique names across the network we could give the MNR numbers/names/handles. And each MNR would then simply have to perform the check of uniqueness themselves and that number/name/handle will be part of that username for its lifetime. [...] Example: The mastodon.social MNR has the handle odon and hence my unique username could be nocksock.odon. [...] So people would never have to know on which instance I currently am, they can always address me using @nocksock.odon."*
  The downside is that MNR is like an identities decentralized blockchain that would be really hard to set up and probably plenty of bugs. Plus, a lot of people have already created multiple accounts with the same username on different instances. Unique ID system seems to be too complex for Mastodon also because this social network is a federation, not a big network.

- Another proposal was to buy your own domain name, point your Mail eXchanger-equivalent DNS record; *"it might be also possible to setup an alias via WebFinger A, instead of DNS, to point to the correct instance, then register the domain as an alias on that instance so that you can register yourname@yourdomain.com. Later, if you want to move instances for whatever reason, you can repeat the DNS/alias process on another server, and migrate your toots/following list over. Everyone still follows you and addresses you correctly without needing an explicit "hey I moved instances" notice."*
  Unlike email providers, instances have different moderation rules, antiharassment policies, they will gather different communities, some will choose not to federate with other instance that they consider harmful.

- The "easy" method to achieve the result, using the keys:
  - *User wants to migrate from @bob@roddenberry.zone to @robby@abrams.website*
  - *User goes to abrams.website and has to give it a "migration key" (internally signed by @robby)*
  - *User goes to (original node) roddenberry.zone and presses "Migrate Account" and pastes in the key.*
  - *roddenberry.zone sends a signed notification to subscribers of @bob*
  - *Each subscriber sees the signed notification and verifies it is @bob, and discovers @robby@abrams.website, verifying that the key in the notification has been signed there by @robby*
  - *Each subscriber, once they verify, can select to unsubscribe from @bob*

and subscribe to @robby

The downside with the keys is that this makes the private key very sensitive; if it leaks, you are screwed.

- Mastodon could use OAuth A between instances to allow one instance to copy/migrate the account and associated content over from the original instance, in a TCP way. The flow could look like:
  - *User from Old Instance arrives at New instance.*
  - *User activates the "Sign up" flow.*
  - *After creating credentials, a new step: "Migrate Account".*
  - *Enter the original instance URL.*
  - *OAuth A flow is triggered, New Instance asks for read privileges on Old Instance (read-only prevents bad-actor instances from posting/deleting content on the original instance).*
  - *With read privileges, New Instance duplicates any account settings (bio, etc, perhaps this is configurable in the wizard).*
  - *New Instance queues a full content migration, in offline.*
  - *After content is migrated, user is notified that they can now delete their account on Old Instance.*

  The only downside is replies, etc could be broken unless the migration process literally updates the toots of all users on all instances that replied to your toots, that is impractical.

- Another proposal is related to the history/development of Friendica/Hubzilla: *Hubzilla was split off from Friendica because of a desire to create a decentralized permissions system. Friendica, for the most part, is a social network that includes profile features like albums/calendars/etc, similar to Facebook. Hubzilla, on the other hand, is a permissions/ID system that happens to allow social networking. This means that account migration is easily achievable using Hubzilla's concept of "nomadic identity": your ID is referred to as a "channel" and takes the form handle@site.tld, similar to what OStatus A / ActivityPub 2.2.3 uses for its accounts. But the key difference is that your content is not tied to the DNS; accounts and channels are separate entities. In effect: you can import a "channel" (identity) from site1.tld to site2.tld seamlessly, either by uploading an export of your data, or by entering your account credentials on another site. This has the benefit of allowing live mi-*

gration, as well as also syncing content back to the other connected accounts. In order for an identity to persist across locations, one must be able to provide or recover:

- *the globally unique ID (GUID) for that identity*
- *the private key assigned to that identity*
- *(if the original server no longer exists) an address book of contacts for that identity.*

This information will be exportable from the original server via API, and/or downloadable to disk or thumb-drive.
In terms of advantages and disadvantages: this is very useful in being resistant to service outages or DNS censorship. It also allows for complete identity portability, so you can move from one hub to another seamlessly. On the other hand, it might also encourage people to create accounts across many different hubs simply to clone the same channel, which is not really necessary unless you want people to be able to access your content from multiple URLs. The Zot A identity is managed by a primary hub, and if that hub goes offline, any hub with the private key can declare itself to be the new primary hub (initiated by the user). Users can import their GUID and private key to any server by either uploading their backup, or by live mirroring from the current primary hub. Your followers' primary hubs are notified of your new hub.

- The key concept that makes migration feasible is having a backup account at another instance, set up in advance. *Like with computer data backups, this should be a thing that people are encourage to do if they are using the system for anything serious, and hopefully it should be pretty easy to do. So, the primary account is @bob@w3c.social, and the backup could be the older account, @bob@mastodon.social. There should be a way for the user to point the accounts at each other, so every subscribing system has recorded that one is the backup for the other, and the servers should be set up do be replicating my data. If the primary server goes away, even suddenly, all the followers can automatically use the backup. If the backup goes away suddenly, then the user picks a new backup. The only catastrophic failure would be if both went away at about the same time. Maybe the system could even support multiple backup servers at once, for the truly paranoid.Thus, some accounts are explicit backup accounts that cannot be posted to; they only get content via sync from their master. The only thing the user can do at the backup server is point it to a new master if the old master dies. To keep network and storage load sane, you could store follow lists on some number of instances n considered sufficient to provide redundancy (say n=3) but then you have*

the problem of which instances store what, how do you retrieve the info when needed, and how do you distribute updates when follow lists or passwords change. Focusing on just the follow lists for the moment, one could store them in a Distributed hash table (DHT) with the key being the global UUID (Unique User ID) of the user, and have some sort of election to pick the n instances to store the data at the time the user account is first created. When one of the n instances replicating the follow lists goes away for long enough (i.e. availability on the DHT drops below n for too long), some process needs to notice this and elect a new instance to host replication data such that DHT availability is brought back up to n. When the user's follow/following lists change, updates need to be replicated.
The major costs of this scheme are development cost - lines of code to write, debug, and maintain - and server load - mostly additional disk space for storing user authentication tuples and follow/following lists, and network traffic replicating updates to follow/following lists.

1.8 Current situation

The last stable version of Mastodon is 2.9.3, it was released on the 9th August 2019. We worked on the 2.9.2 version, realized on the 22nd July 2019 but when we started to work at the beginning of March 2019, the version was the 2.7.3, realized on the 23rd February 2019.

On the 18th January 2019, the founder of Mastodon, Eugen Rochko alias Gargron, wrote a post on GitHub where he described which improvements have been done and which ones will be done:

- In the version 2.1, addition profile redirect notes, in this way the user can configure the account to say "Now moved to example@example.com". It also prevents the old account from getting followers. Unfortunately this is not a complete solution because the existing followers still have to be manually notified to re-follow the new account and it is also full of risks, such as stressing the network with increased traffic, selling followers, etc.
- The version 2.7 will contain a handler for a *Move* activity, the account given as *target* will be checked for the old account in its *alsoKnownAs* property (to ensure the user cannot just send followers to random people without consent), and if it is there, the followers on the server will be re-assigned to the new account. It is necessary to split the implementation of receiving/sending of this activity between releases. First, most of the servers need to upgrade to

> 2.7 (once it is out) before it makes sense allowing people to send the *Move* activity, otherwise it would just get ignored. Thus, the programmer has proposed to implement the sending part in 2.8 or 2.9. And the action will likely be behind a long cooldown to prevent people from stressing the network through constant hopping between accounts.

He commented:

> "[...] This is not the final solution to the migration issue, unfortunately posting a status is computationally expensive. Importing tens of thousands of posts is a huge task, and it is really hard to design it in such a way that both low-end servers and high-end servers can handle it without impacting the experience of active users."

From the date of this post (the version at that moment was 2.6.5) until today the programmers did not find a definitive solution, but they added the following features to temporary fix the issue and to prepare the system to support the future account migration service[2]:

- **v2.7.0**:
  - Add CSV export for lists and domain blocks
- **v2.7.1**:
  - Fix slow fallback of CopyAccountStats migration setting stats to 0
  - Fix wrong command in migration error message
- **v2.7.4**:
  - Fix lists export
- **v2.8.0**:
  - Add option to overwrite imported data instead of merging
  - Fix race conditions when creating backups
  - Change format of CSV exports of follows and mutes to include extra settings
- **v2.9.3**:
  - Fix backup service crashing when an attachment is missing

[2]https://github.com/tootsuite/mastodon/releases

1.9 Practical work

As practical work, the goal is to bypass the database of Mastodon and store data on a distributed ledger in order to share them between multiple instances. In particular, this thesis addresses a way to perform a complete Sign up/Sign in process where a user is able to register an account on one instance and log in on an other one.
In order to ensure the availability of the service both for old users registered on the Database both for new users that want to create an account on the Blockchain, our solution registers the credentials of a new user on the Blockchain and on the Database. Our idea is to drop out the use of the Database in the future.

In the following picture the mechanism according to which the Registration and the Login are performed is described more in details.

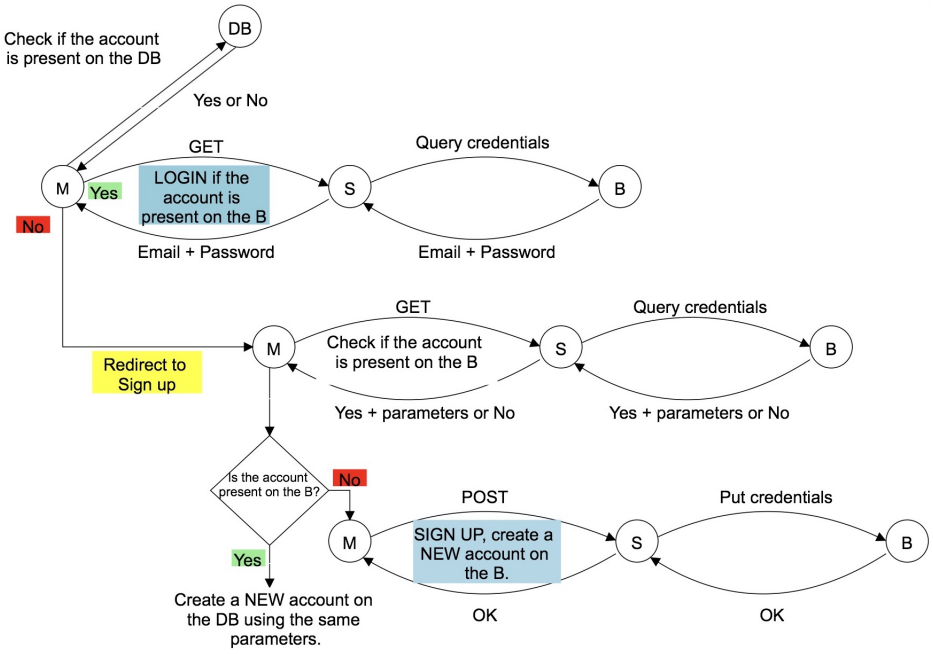


Figure: A flow chart to describe the process of Login and Sign up
Legend: B stays for Blockchain, DB stays for Database, M stays for Mastodon, S stays for Server

At the moment, in Mastodon-M, you can decide to allow the two factors authentication, it means that each time you want to Login you receive a message on your phone. In the normal conditions you just have to insert your email and password. Our solution, as first step, checks if the user has an account on that instance (if he is present on the Database-DB), in the positive case, it checks if the user is on the

Blockchain-B and returns the credentials of the users. The password is checked by Mastodon side. If the user is not on the Blockchain the Login is performed as I described before. In the case the user is not on the Database it means that he is on another instance where an account is not present. Mastodon redirects the user on the Sign up page. At this point he can register himself on the new instance, if he is present on the Blockchain he will be able to create a new account with the same credentials stored on it, otherwise he will create a new account and the parameters will be stored on the Database and on the Blockchain.

Details about the design and the implementation will be discussed in chapter 3.

Chapter 2

Technical background

2.1 Decentralization

The theoretical definition of Decentralization affirms that it is the process by which the activities of an organization, particularly those regarding planning and decision making, are distributed or delegated away from a central, authoritative location or group. [12]

The best known model is the centralized one, in this type of network all users are connected to a central server, which is the acting agent for all communications.

Another type is the distributed one: this kind of network is spread over different networks. This provides a single data communication network, which can be managed jointly or separately by each network. The decentralized networks' node can interoperate without a centralized source of decision making and management.

As we will deeply see in section 2.3, the Blockchain is a distributed and decentralized register, i.e. the central node is replaced by several nodes (distributed) and each node belongs to a different owner (decentralized).

2.1.1 DOSN: Decentralized Online Social Networks

Nowadays, Online Social Networks (OSNs) are really popular and in the last years they changed the way people communicate and interact. With 2.38 billion monthly active users as of the first quarter of 2019, Facebook is the biggest social network worldwide. [13] OSNs provide several services offering to their users the opportunity of building a public profile, looking up new friends among the registered users,

establishing relationships, and sharing content and information, also within groups of users and the possibility of building communities of users characterized by common interests. [14]

One of the most critic problem that each centralized platform has to face, concerns the privacy of the users' data. Unfortunately, the OSNs are mainly developed in this way: social data are stored in centralized servers, and the companies running the OSNs use these data for commercial goals. These issues have led researchers to work on different solutions based on the decentralization of OSN services.

A Decentralized Online Social Network (DOSN) [15] is an online social network implemented on a distributed platform. In this type of architecture the single service provider is replaced by a set of nodes that cooperate to guarantee all the functionalities offered by a centralized OSN. [16] Thus, these new solutions have the objective to provide similar online socializing functionality without the need of any one single central trusted entity. This is achieved by architecture of multiple independent federated servers that provide the same OSN functionality, from which users can freely choose which to join and whom to trust, and between which users can freely and seamlessly switch without losing any of their advantages or functionality, building some peer-to-peer (P2P) networks of end users devices, with direct one-to-one interactions between them. [17] In this way the privacy can benefit of the decentralization.

2.1.2 Decentralized Identifiers: DIDs

The spread of Distributed Ledger Technologies (DLT), sometimes referred to as Blockchain technologies, provides the opportunity for fully decentralized identity management. Decentralized Identifiers (DIDs) [18] "are a new type of identifier for verifiable, "self-sovereign" digital identity, they identify entities that may authenticate via proofs (e.g., digital signatures, privacy-preserving biometric protocols, etc.). DIDs point to DID Documents which contain a set of service endpoints for interacting with the entity the DID identifies (aka the DID subject). Any entity may have as many DIDs as necessary.

DIDs achieve global uniqueness without the need for a central registration authority, but the algorithms capable of generating globally unique identifiers automatically produce random strings of characters that have no human meaning. Zooko [19] claims that any naming system can only fulfill two of the following three desirable properties:

- Secure: only one , unique and specific entity to which the name maps and nobody can successfully pretend to be the owner of someone else's domain

name.

- Decentralized: Names correctly resolve to their respective entities without the use of a central authority or service.
- Human-meaningful: Meaningful and memorable names are provided to the users.

These three properties, collectively called Zooko's Triangle, leave three possible choices to implement a naming system: the combination of two of them for each possibility.

Authentication is the mechanism by which a DID subject can cryptographically prove that they are associated with a DID. Authentication is separate from Authorization because the subject may wish to enable others to update their DID Document. Authorization is the mechanism used to state how operations may be performed on behalf of the DID subject. Delegation is the mechanism that the subject may use to authorize others to act on their behalf.

The DLTs hosting DIDs and DID Documents have special security properties for preventing active attacks. Their design uses public/private key cryptography to allow operation on passively monitored networks without risking compromise of private keys. There are two methods for proving control of the private key corresponding to a public key description in the DID Document: static and dynamic. The static method is to sign the DID Document with the private key. This proves control of the private key at a time no later than the DID Document was registered. If the DID Document is not signed, control of a public key described in the DID Document may still be proven dynamically as follows:

- First step: send a challenge message containing a public key description from the DID Document and a nonce to an appropriate service endpoint described in the DID Document.
- Second step: verify the signature of the response message against the public key description.

The anti-correlation protections of pseudonymous DIDs are easily defeated if the data in the corresponding DID Documents can be correlated. For example, using same public key descriptions or bespoke service endpoints in multiple DID Documents can provide as much correlation information as using the same DID. Therefore the DID Document for a pseudonymous DID should also use pairwise-unique public keys. It might seem natural to also use pairwise-unique service endpoints in the DID Document for a pseudonymous DID.

However, unique endpoints allow all traffic between to DIDs to be isolated perfectly

into unique buckets, where timing correlation and similar analysis is easy. Therefore, a better strategy for endpoint privacy may be to share an endpoint among thousands or millions of DIDs controlled by many different subjects".

2.2 Mastodon

2.2.1 What is it?

Mastodon is a DOSN, open source that offers a microblogging service [500 characters] with functionalities similar to Twitter [140 characters]. This social network reached 2 million users in May 2019. [5] It is a federation, in this way thousands of independent communities running Mastodon form a coherent network, where while every planet is different, being part of one is being part of the whole. Mastodon comes with effective anti-abuse tools to help protect users. Thanks to the network's spread out and independent nature there are more moderators who you can approach for personal help, and servers with strict codes of conduct. Your feed is chronological, ad-free and non-algorithmic. All you need to do to sign up is choose a server. Just like when signing up for an e-mail address, one server is going to be hosting your account and be part of your identity, you can follow and talk to anyone from any server, regardless of your choice. You can also host your own social media platform on your own infrastructure. [4] As in each OSN you can follow other users, post status messages in a timeline, where hashtags can be used and other users can be mentioned, status posts can be favorited and replied to. It supports restricting the audience of individual status posts to specified users, allowing private conversations. Furthermore, Mastodon supports uploading and managing of image and video files. Finally, Mastodon offers a fine-grained management of privacy settings and access control that allows users to restrict access to individual parts or the entire user profile.

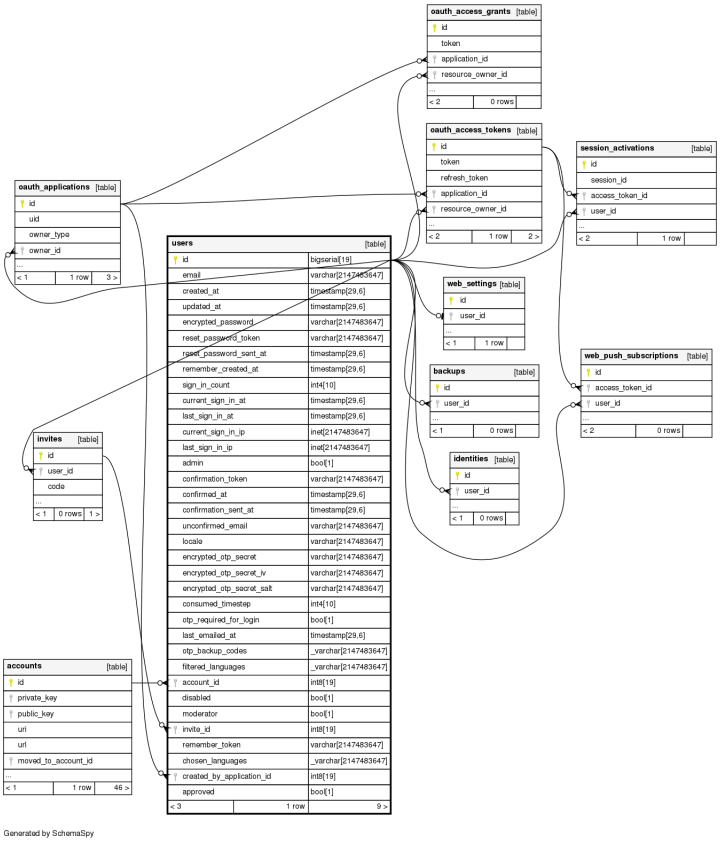
2.2.2 How does it work?

The main components of Mastodon consist of:

- **Ruby on Rails** Rails is a web application development framework written in the Ruby programming language. It is designed to make programming web applications easier by making assumptions about what every developer needs to get started. It allows you to write less code while accomplishing more than

many other languages and frameworks. Experienced Rails developers also report that it makes web application development more fun.[20]

- **ActivityPub protocol** The ActivityPub protocol is a decentralized social networking protocol based upon the ActivityStreams 2.0 [A] data format. It provides a client to server API for creating, updating and deleting content, as well as a federated server to server API for delivering notifications and content. [21]
- **PostgreSQL** also known as Postgres, is a free and open-source relational database management system (RDBMS) emphasizing extensibility and technical standards compliance. It is designed to handle a range of workloads, from single machines to data warehouses or Web services with many concurrent users.[22]
- **Schema of the database** In the following picture there is an example of how the tables interact in Mastodon, the main one is the Users' table.



Example of a table of the database: Users' table.

2.2.3 ActivityPub protocol

The ActivityPub protocol, as we can read from the technical documentation [21], "is a decentralized social networking protocol based upon the ActivityStreams 2.0 [A] data format.

Main features

The ActivityPub protocol provides two layers:

- a **client to server protocol**, or **"Social API"** for creating, updating and deleting content (so users, including realworld users, bots, and other automated processes, can communicate with ActivityPub using their accounts on servers, from a phone, desktop, web application, or any other device).
- a **server to server protocol**, or **"Federation Protocol"** for delivering notifications and content (so decentralized websites can share information).

ActivityPub implementations can implement just one of these things or both of them. However, servers may still implement one without the other. This gives three conformance classes:

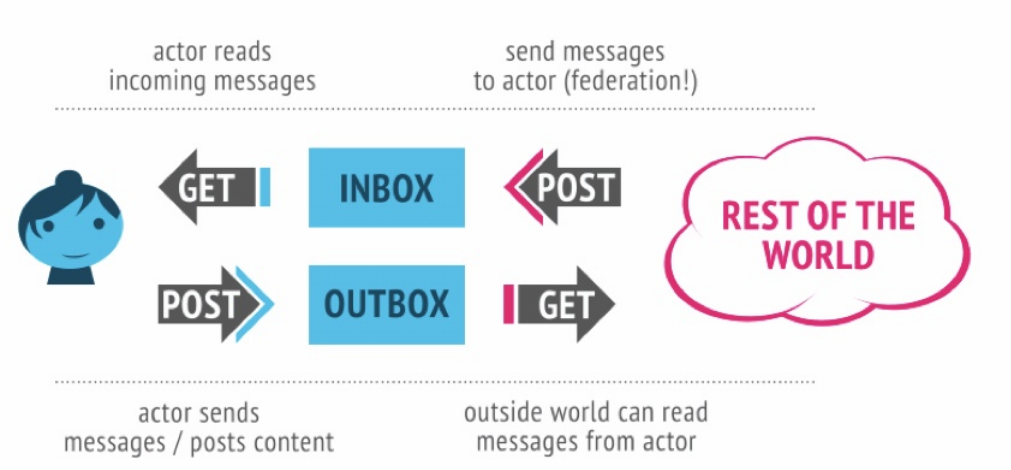
- **ActivityPub conformant Client**: This designation applies to any implementation of the entirety of the client portion of the client to server protocol.
- **ActivityPub conformant Server**: This designation applies to any implementation of the entirety of the server portion of the client to server protocol.
- **ActivityPub conformant Federated Server**: This designation applies to any implementation of the entirety of the federation protocols.

Objects are the core concept. They are often wrapped in Activities and are contained in streams of Collections, which are themselves subclasses of Objects. They must have unique global identifiers, unless they are intentionally transient (short lived activities that are not meant to be looked up, such as some kinds of chat messages or game notifications).

In this protocol a user is represented by **"Actors"** via the user's accounts on servers. User accounts on different servers correspond to different actors. Actors are retrieved like any other Object in ActivityPub. Like other ActivityStreams objects, actors have an ID, which is a URI (Uniform Resource Identifier). When entered directly into a user interface (for example on a login form), it is desirable to support simplified naming. Every Actor has:

- an inbox: How they get messages from the world
- an outbox: How they send messages to others

ActivityPub uses ActivityStreams for its vocabulary. It includes all the common terms needed to represent all the activities and content flowing around a social network. ActivityStreams can be extended via [JSON-LD: JavaScript Object Notation for Linked Data, a method of encoding Linked Data using JSON]. These are endpoints, or URLs which are listed in the ActivityPub actor's ActivityStreams description.



Behaviour of GET and POST using inbox and outbox in ActivityPub [21]

- You can POST to someone's inbox to send them a message (server-to-server/federation only).
- You can GET from your inbox to read your latest messages (client-to-server; this is like reading your social network stream).
- You can POST to your outbox to send messages to the world (client-to-server).
- You can GET from someone's outbox to see what messages they have posted (or at least the ones you are authorized to see). (client-to-server and/or server-to-server).

ActivityPub defines several **Collections** with special behavior. Some of these collections are designed to be of type OrderedCollection specifically, while others are permitted to be either a Collection or an OrderedCollection. An OrderedCollection must be presented consistently in reverse chronological order. Examples include Following and Followed collections.

Client to Server Interactions

Activities, as defined by ActivityStreams, are the core mechanisms for creating, modifying and sharing content within the social graph. Client to server interaction takes place through clients posting Activities to an actor's outbox. The body of the POST request must contain a single Activity (which may contain embedded objects), or a single non-Activity object which will be wrapped in a Create activity by the server. Clients are responsible for addressing new Activities appropriately. The Followers Collection and/or the Public Collection are good choices for the default addressing of new Activities.

The types of Activities allowed are:

- The **Create** activity, used when posting a new object.
- The **Update** activity, used when updating an already existing object.
- The **Delete** activity, used to delete an already existing object.
- The **Follow** activity, used to subscribe to the activities of another actor.
- Upon receipt of an **Add** activity into the outbox, the server should add the object to the collection specified in the target property.
- Upon receipt of a **Remove** activity into the outbox, the server should remove the object from the collection specified in the target property.
- The **Like** activity which indicates the actor likes the object.
- The **Block** activity, used to indicate that the posting actor does not want an other actor (defined in the object property) to be able to interact with objects they post.
- The **Undo** activity, used to undo a previous activity.

Federated servers must perform delivery on all Activities posted to the outbox according to outbox delivery.

Server to Server Interactions

Servers communicate with other servers and propagate information across the social graph by posting activities to actors' inbox endpoints. An Activity sent over the network should have an ID, unless it is intended to be transient.

In order to propagate updates throughout the social graph, Activities are sent to the appropriate recipients. First, these recipients are determined through following the

appropriate links between objects until you reach an actor, and then the Activity is inserted into the actor's inbox (delivery). This allows recipient servers to:

- conduct any side effect related to the Activity, e.g. a notification that an actor has liked an object is used to update the object's like count;
- deliver the Activity to recipients of the original object, to ensure updates are propagated to the whole social graph (see inbox delivery).

An Activity is delivered to its targets (which are actors) by first looking up the targets' inboxes and then posting the activity to those inboxes: an HTTP POST request (with authorization of the submitting user) is made to the inbox, with the Activity as the body of the request. This Activity is added by the receiver as an item in the inbox OrderedCollection. Attempts to deliver to an inbox on a non-federated server should result in a 405 Method Not Allowed response.

For federated servers performing delivery to a third party server, delivery should be performed asynchronously, and should additionally retry delivery to recipients if it fails due to network error.

For servers which support both Client to Server interactions and Server to Server Interactions, the objects are received in the outbox and the server must target and deliver them (**Outbox Delivery Requirements for Server to Server**). When Activities are received in the inbox, the server needs to forward these to recipients to whom the origin was unable to deliver (**Forwarding from Inbox**).

For servers hosting many actors, delivery to all followers can result in an overwhelming number of messages sent. Some servers would also like to display a list of all messages posted publicly to the "known network". Thus ActivityPub provides an optional mechanism responding to these issues. When an object is being delivered to the originating actor's followers, a server may reduce the number of receiving actors by identifying all followers sharing the same sharedInbox, and by delivering objects to the latter instead of delivering them to every followers' individual inbox. Thus in this scenario, the remote/receiving server participates in determining targeting and performing delivery to specific inboxes. Additionally, if an object is addressed to the Public special collection, a server may deliver that object to all known sharedInbox endpoints on the network (**Shared Inbox Delivery**).

The types of Activities allowed are:

- The **Create** activity, which should appear in the actor's inbox. It is likely that the server will want to locally store a representation of this activity and its accompanying object.

- The **Update** activity, which means that the receiving server should update its copy of the object of the same id to the copy supplied in the Update activity. Unlike the client to server handling of the Update activity, this is not a partial update but a complete replacement of the object.
- The **Delete** activity: the server should remove its representation of the object with the same ID, and may replace that representation with a Tombstone object.
- The **Follow** activity: the server should generate either an **Accept** or **Reject** activity with the Follow as the object and deliver it to the actor of the Follow. In the case of receiving an Accept referencing this Follow as the object, the server should add the actor to the object actor's *follower* collection. In the case of a Reject, the server must not add the actor to the object actor's *follower* collection.
- The **Add** activity: the server should add the object to the collection specified in the target property.
- The **Remove** activity: the server should remove the object from the collection specified in the target property.
- The **Like** activity: the server should increment the object's count of likes by adding the received activity to the *like* collection, if this collection is present.
- The **Announce** activity: a server should increment the object's count of shares by adding the received activity to the *share* collection, if this collection is present. This activity is effectively what is known as "sharing", "reposting", or "boosting" on other social networks.
- The **Undo** activity is used to undo the side effects of previous activities.

Security Considerations

- **Authentication and Authorization**: ActivityPub uses authentication for two purposes: first, to authenticate clients to servers, and secondly, in federated implementations, to authenticate servers to each other. Unfortunately at the time of standardization, there is no strong consensus on what mechanisms to use for authentication.
- **Verification**: Servers should be careful to verify that new content is really posted by the actor that claims to be posting it, and that the actor has permission to update the resources it is attempting to.

- **Accessing localhost URIs**: If your ActivityPub server or client permits requests to localhost URIs for development purposes, consider making it a configuration option which defaults to off. It could be dangerous: making requests to URIs on localhost which do not require authorization may unintentionally access or modify resources assumed to be protected to be usable by localhost-only.
- **URI Schemes**: Client and server authors should carefully check how their libraries handle requests, and potentially whitelist only certain safe URI types, such as http and https.
- **Recursive Objects**: Servers should set a limit on how deep to recurse while resolving objects, or otherwise specially handle ActivityStreams objects with recursive references.
- **Spam**: No specific mechanism for combating spam is provided in ActivityPub. It is recommended that servers filter incoming content both by local untrusted users and any remote users through some sort of spam filter.
- **Federation denial-of-service**: Servers should implement protections against denial-of-service attacks from other, federated servers. This can be done using, for example, some kind of ratelimiting mechanism.
- **Client-to-server ratelimiting**: Servers should ratelimit API client submissions. This serves two purposes:

    1. It prevents malicious clients from conducting denial-of-service attacks on the server.

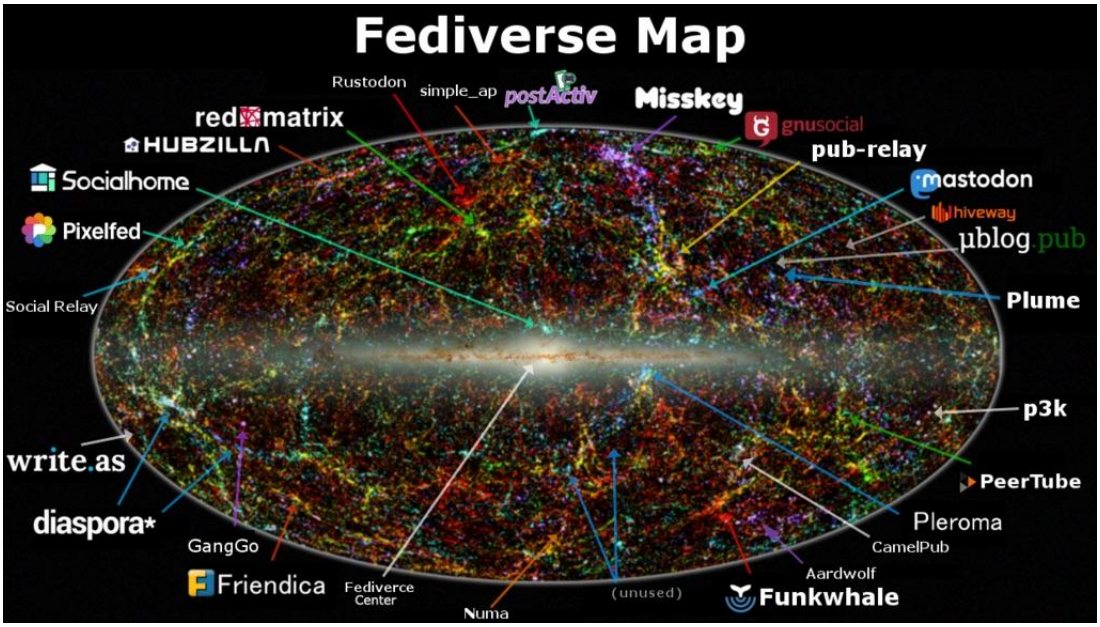
    2. It ensures that the server will not distribute so many activities that it triggers another server's denial-of-service protections.

- **Client-to-server response denial-of-service**: In order to prevent a client from being overloaded by oversized Collections, servers should limit the size of Collection pages they return to clients. Clients should still be prepared to limit the size of responses they are willing to handle in case they connect to malicious or compromised servers, for example by timing out and generating an error.
- **Sanitizing Content**: Any activity field being rendered for browsers should take care to sanitize fields containing markup to prevent cross site scripting attacks.
- **Not displaying bto and bcc properties**: bto and bcc properties (only

intended to be known/seen by the original author of the object/activity) must already be removed for delivery, but servers are free to decide how to represent the object in their own storage systems".

2.2.4 Fediverse Network

Mastodon is a part of the wider Fediverse network, allowing its users to also interact with users on different open platforms that support the same protocol.



The Fediverse (the union of "federation" and "universe") is a group of federated (i.e. interconnected) servers that are used for web publishing (i.e. social networking, microblogging, blogging, or websites) and file hosting. Users can create accounts that are linked to an identity on different servers (instances). Thanks to these identities, users are able to communicate over the boundaries of the instances. It could be compared with the email service rather than other social media sites. With email clients, it does not matter what site is used to make your account, because anyone having an email address can receive emails. Similarly, in the Fediverse, the account is created on one instance, but it is nonetheless possible to reach users with accounts on other instances.

In the case of federated instances, when one user ("Alice," for example) follows another user ("Bob") from a different instance, the instance Alice is from subscribes to Bob's posts. This means that when Bob makes a post, it is not only sent to

users from their own instance, but also to the instances of users that are subscribed to their posts, like Alice.

Not all instances are federated with every other instance in the Fediverse. For example, awoo.space is an instance that is only federated with mastodon.social. This means that only users on mastodon.social and on awoo.space will be delivered posts made by users to which they are subscribed on either platform. [23]

As an identity in the Fediverse, one is able to post text and other media, or to follow posts by other identities. In some cases, one can even show or share data (video, audio, text and other files) publicly or to a selected group of identities, and allow other identities to edit one's data (i.e. a calendar or an address book). [24]

2.2.5 The open issue

The open issue we worked on was: "**Support account migration #177**". The original post was posted on 22 November 2016 and stated:

> A lot of people seem to be jumping on https://mastodon.social right now, even though the end goal is to have users separated out across multiple federated instances. However, if people start putting up a lot of content and getting followers on the primary instance, this will be a disincentive to move providers. I think this largely means that Mastodon needs to support two things:
>
> - Account import across providers, which should be authenticated on both ends to prevent people bulk copying another person's account.
> - A "redirected account" field against user accounts, which indicates the full URL of the new user account.
>
> When users configure a redirect location against their account (whether explicitly on an account page, or implicitly set during a cross-provider account import), the instance on which the account is should implicitly and automatically redirect followers.
>
> That is, if I have the account @hq on the primary instance (which I do), and I set up the account @hach-que on another Mastodon service, the @hq should:
>
> - Remain on the primary instance, and not be disabled in any way.
> - Show posts from @hach-que after the redirection is set up.

- Disallow posting from the @hq account while ever the redirection is in place.
- Existing followers of @hq should start seeing posts from @hach-que instead.
- New followers of @hq should be allowed to follow the account (and internally, they are following @hq, but see posts from @hach-que).
- Followers should not actually have their lists updated to follow @hach-que - in the event of a mistaken redirect, removing the redirect should act exactly as if there never was a redirect in the first place.

  I think this should work, but I'm interested to hear other people's thoughts.

After reading all the comments of users on GitHub, it can be stated that this is an issue for several reasons:

- a lot of people use this social network as a working tool. They could want to move their account on another instance for personal or business reasons, but this service is not allowed. The only solution is to create a new account from scratch on another instance. On this topic, @JoshuaACNewman writes the following:

  > I'd agree that, while losing posts would be annoying, losing followers would be debilitating. Remember that a lot of us have no marketing outside of our social media. This isn't a hobby or pastime. It's our job in a society that doesn't otherwise value our work, whether it's art or community journalism or whatever. [...]

- a normal user could want to move his account, but he should recreate his account (followers, posts, etc.) On this topic @Sadzeih writes: "[...] Account migration is necessary. The most important is toots and followers for me. Followings too but they can already be exported. [...]"
- in the unfortunate events that the server goes down, someone using Mastodon for working reasons would had big troubles because he would lose everything.
- in the unfortunate events that the server goes down, a normal user would lose everything and could have no time or desire to recreate a new account on another instance.

On 28th March, Mircea Kitsune published:

> Today brought us the sad news that one of the biggest Mastodon instances, friends.nico, will be shutting down due to financial difficulties. I understand this is the first time an important server goes down with potential to have a powerful impact on the fediverse. I sincerely believe this is a good reminder of why migration will rapidly become essential... as over time some instances inevitably disappear, and with them so will the activity of users who won't be able to take their history to any other server.
>
> People fleeing a dying instance can download the full data of their account. But without a way to import it, that data will forever be lost somewhere on their hard drive. Users will likely start a new on another instance... everything they've done up to that point will never be read or interacted with again.

This testimony demonstrates how important and necessary it is to find a solution to this issue.

2.3 Blockchain

In this section we present the Blockchain technology, giving a description of what it is and how it works in the subsections 2.3.1 and 2.3.2. After these ones, in the 2.3.3, we talk about Proof-of-Work and in the 2.3.4, we compare the two different types of Blockchain: Permissionless and Permissioned. In the 2.3.5 we present the first Blockchain: Bitcoin. In the following subsection, 2.3.6 we describe another type of Blockchain: Ethereum. Afterwards, in the 2.3.7 we discuss Hyperledger, the Blockchain that we used in our work, concentrating on Hyperledger Fabric and its main features. We conclude the section with an explanation of two important concepts: the Consensus in the 2.3.8 and the Smart Contracts in the 2.3.9. In the last subsection, the 2.3.10, we discuss the reason why we need a Blockchain.

2.3.1 What is it?

A Blockchain, also called Distributed Ledger, is essentially an append-only data structure maintained by a set of nodes which do not fully trust each other. [25] A ledger consists of an ordered list of transactions replicated over all the nodes. A transaction, a sequence of operations applied on some states, requires the ACID (Atomicity, Consistency, Isolation, and Durability) semantics, as in traditional databases. All nodes in the system agree on the transactions and their order.

A Blockchain starts with some initial states, and the entire history of update operations made to the states is recorded by the ledger.

2.3.2 How does it work?

The name Blockchain derives from the fact that it is comprised by a continuously growing list of records called blocks that contain transactions, that are created and exchanged by peers of the Blockchain network and modify the state of the Blockchain. [26][27]

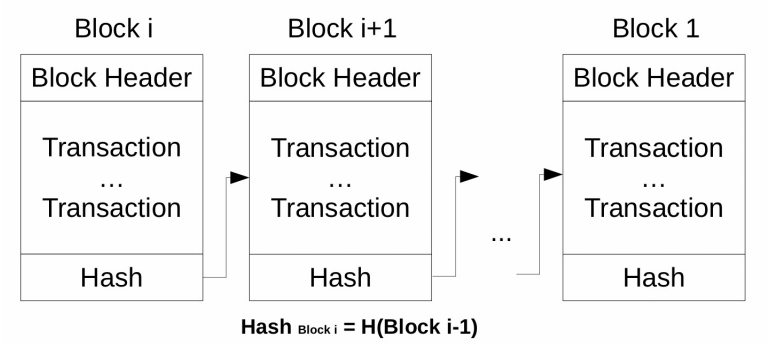


Figure: Blockchain structure [26]

Blocks are protected from tampering by cryptographic hashes and a consensus mechanism (explained in details in the subsection 2.3.8). A Blockchain, as can be seen in the Figure, consists of a sequence of blocks in which each one contains the cryptographic hash of the previous block in the chain. Because of this specific structure, block j cannot be forged without also forging all subsequent blocks $j+1...i$. The system is secure as long as honest nodes collectively control more CPU power than any cooperating group of attacker nodes. [28]

The following list of steps to add a block in the Bitcoin Blockchain provides an example of how the network is run:

1. New transactions are broadcast to all nodes.
2. Each node collects new transactions into a block.
3. Each node works on finding a difficult Proof-of-Work for its block.
4. When a node finds a Proof-of-Work, it broadcasts the block to all nodes.
5. Nodes accept the block only if all transactions in it are valid and not already spent.

6. Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

The longest chain is always considered to be the correct one by nodes and they will keep working on extending it.
New transaction broadcasts do not necessarily need to reach all nodes. If a node does not receive a block, it will request it when it receives the next block and realizes it missed one.
Since the system is decentralised and all parties have an opportunity to create a new block on some older pre-existing block, the resultant structure is necessarily a tree of blocks [29]: Merkle Tree [30], with only the root included in the block's hash. Old blocks can then be compacted by stubbing off branches of the tree.

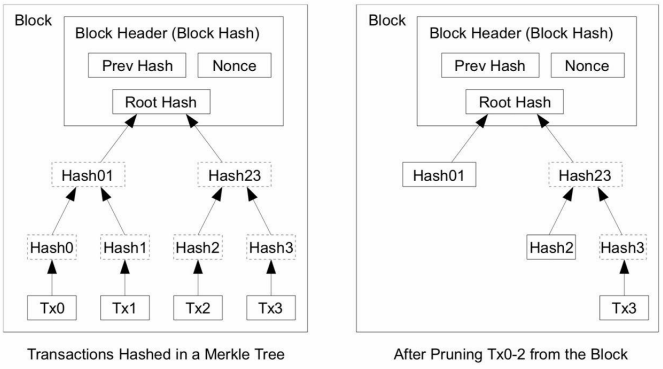


Figure: Merkle Tree [28]

2.3.3 Proof-of-Work (PoW)

The Proof-of-Work is a protocol for consensus (more details in subsection 2.3.8). It was invented by Cynthia Dwork and Moni Naor that presented it in a 1993 journal article. [31] It was used to deter denial of service attacks and other service abuses such as spam on a network by requiring some work from the service requester, usually meaning processing time by a computer. The mining PoW exists as a cryptographically secure nonce, a random, one-time, whole number that proves that a particular amount of computation has been expended in the determination of it. Miners compete with each other to find a nonce that produces a hash with a value lower than or equal to that set by the network difficulty. [32] If a miner finds such a nonce, called a *golden nonce*, then they win the right to add that block to the blockchain and receive the block reward. It involves scanning for a value that, when hashed, begins with a number of zero bits, as is the case with SHA-256. The average work required is exponential in the number of zero bits required and can

be verified by executing a single hash. [28]

2.3.4 Blockchain's Types

We can divide the Blockchains in two different classes: Permissionless (or Public) and Permissioned ones. These can be further divided in Public or Private, according to the public verifiability requirement [27]: it allows anyone to verify the correctness of the state of the system. In a distributed ledger, each state transition is confirmed by verifiers, which can be a restricted set of participants.

- **Permissionless**: any nodes can join and leave. [25] A set of transactions is broadcasted by each node that wants to perform them. At this point, miners, some special nodes group the transactions into blocks and they check for their validity. To append the blocks on the Blockchain, miners use a consensus protocol, the most employed is PoW that works well in the public settings because it guards against Sybil attacks [33]. However, being non-deterministic and computationally expensive in the identification of who can update the ledger, it is unsuitable for applications which must handle large volumes of transactions in a deterministic manner, such as banking and finance applications.
  There is no central entity which manages the membership,or which could ban illegitimate readers or writers. This openness implies that the written content is readable by any peer[27].
  In public settings the need for transaction privacy (the transactions cannot be linked from one to another and the transaction content is known only to its participants) is driven by two factors:
  - deanonymization attacks have successfully recovered the underlying structure of the Bitcoin network [34], and even linked Bitcoin addresses to real-world identities [35].
  - transaction linkability can undermine the currency's fungibility, rendering some coins more valuable than others due to their histories.

  The most widely known instance of permissionless Blockchains are Bitcoin and Ethereum.

  Actually there are also Private Permissionless solutions: anyone can decide to join the Blockchain. However, contrary to a public Blockchain, other nodes will only acknowledge its existence, but not share any data. The best known platform in this space is Holochain. These projects are quite recent.

- **Permissioned**: the node identities are known in the private settings, thus most Blockchains adopt one of the existing distributed consensus protocols (more details in subsection 2.3.8). They support smart contracts, the mean to express complex transaction logics. [25]
  There is a central entity decides and attributes the right to individual peers to participate in the write or read operations of the blockchain[27].
  In private settings, complete transparency of transaction history may not be a problem. Either transparency is desirable for the applications, such as financial auditing, or it is straightforward to add an access control layer to protect the Blockchain data.
  The most widely known instance of permissioned blockchains is Hyperledger Fabric.

In the following picture the four different types of Blockchain previously described are reported.

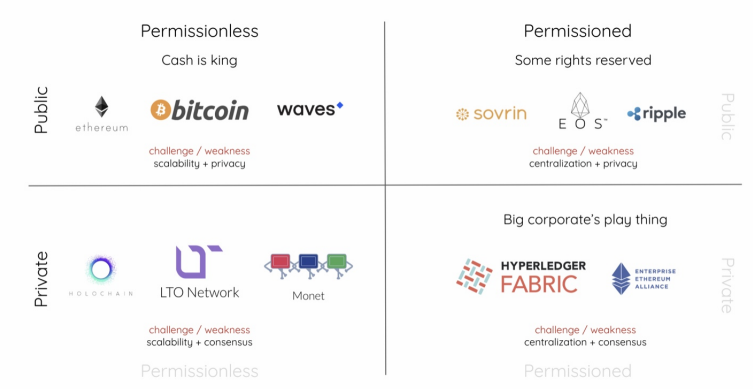


Figure: Blockchain's Type[36]

2.3.5 Bitcoin

In 2008 Satoshi Nakamoto [28] presented in a white paper a purely peer-to-peer version of electronic cash that would allow online payments to be sent directly from one party to another without going through a financial institution (TTP - trusted third party), offering a transparent and integrity protected data storage. In this way, what is needed is an electronic payment system based on cryptographic proof instead of trust. As we saw in 2.3.2 the network timestamps transactions by hashing them into an ongoing chain of hash-based PoW, forming a record that cannot be changed without recomputing it. The longest chain is both the proof of the sequence of events witnessed and the proof that it came from the largest pool of CPU power. Until a majority of CPU power is controlled by nodes that are

not cooperating to attack the network, they will generate the longest chain and outpace attackers. Nodes can leave and rejoin the network at will, accepting the longest PoW chain as proof of what happened while they were gone.

2.3.6 Ethereum

Ethereum [29] is a public Blockchain. It is a project which attempts to build the generalised technology: can be viewed as a transaction-based state machine. We begin with a genesis state and incrementally execute transactions to morph it into some final state. The state can include any information that can currently be represented by a computer is admissible. Transactions thus represent a valid arc between two states. A valid state transition is one which comes about through a transaction. Transactions are collated into blocks that are chained together using a cryptographic hash as a means of reference. Mining is the process of dedicating effort (working) to bolster one series of transactions (a block) over any other potential competitor block. It is achieved thanks to a cryptographically secure proof, PoW. . In order to incentivise computation within the network, there needs to be an agreed method for transmitting value. To address this issue, Ethereum has an intrinsic currency: Ether.
The resultant structure is a tree of blocks: a modified Merkle Patricia tree.
All programmable computation in Ethereum is subject to fees, in order to avoid issues of network abuse. The fee schedule is specified in units of gas: every transaction has a specific amount of gas associated with it (*gasLimit*). This is the amount of gas which is implicitly purchased from the sender's account balance. The purchase happens at the according *gasPrice*, also specified in the transaction. The transaction is considered valid if and only if the account balance can support such a purchase. To execute a transaction we have to define the amount of gas this transaction requires to be paid. For a valid transaction, the execution begins with an irrevocable change made to the state. The Ether for the gas is given to the miner, whose address is specified as the beneficiary of the present block.
The process of finalising a block is composed by four steps:

1. Validate (or, if mining, determine) ommer (gender-neutral term to mean "sibling of parent") headers;
2. Validate (or, if mining, determine) transactions;
3. Apply rewards;
4. Verify (or, if mining, compute a valid) state and nonce.

In Ethereum the execution model (how the system state is altered given a series of

bytecode instructions and a small tuple of environmental data) is specified through a formal model of a virtual state machine, known as the Ethereum Virtual Machine (EVM).
Ethereum is among the first blockchains offering Turing-complete smart contracts. Users write their contracts in either Solidity, Serpent or LLC language, which then get compiled to EVM bytecodes. EVM executes normal crypto-currency transactions, and it treats smart contract bytecodes as a special transaction. Specifically, each smart contract is given its own memory to store local states. Resources consumed during execution of the contract, both in terms of CPU and memory, are tracked by EVM and charged to the transaction sender's account. EVM also keeps track of intermediate state changes and reverse them if there are insufficient funds to pay for the execution.

2.3.7 Hyperledger

Hyperledger [37] is an open source collaborative effort created to advance cross-industry blockchain technologies. It is a global collaboration, hosted by The Linux Foundation, including leaders in finance, banking, IoT, supply chain, manufacturing and technology.
It supports running Turing-complete code and it offers key-value data model, with which the applications can create and update key-value tuples on the Blockchain. The latest release of Hyperledger (v1.4) outsources the consensus component to Kafka — another building block often found in distributed database systems. More specifically, transactions are sent to a centralized Kafka service which orders them into a stream of events. Every node subscribes to the same Kafka stream and therefore is notified of new transactions in the same order as they are published. Since there is only one Kafka service, the observed transaction sequence is the same at every node. Hyperledger does not have its own bytecotes. Instead, it runs its language-agnostic smart contracts inside Docker containers. Specifically, a contract can be written in any language, which is then compiled into native code and packed into a Docker image. When the contract is uploaded, each node starts a new container with that image. Invoking the contract is done via Docker APIs. The contract can access the blockchain states via two methods *getState* and *putState* exposed by a shim layer. One benefit of Hyperledger is that it supports multiple high-level programming languages like Go and Java. However, its key-value interface with the Blockchain necessitates extra application logics for mapping high-level data structures into key-value tuples.
Hyperledger has no APIs for querying historical states. To support historical data lookup, the contract appends a counter to the key of each account.

Hyperledger incubates and promotes a range of business blockchain technologies, including distributed ledger frameworks, smart contract engines, client libraries, graphical interfaces, utility libraries and sample applications. The Hyperledger greenhouse strategy encourages the re-use of common building blocks and enables rapid innovation of DLT components. In the following picture we can observe all the products that Hyperldger offers.

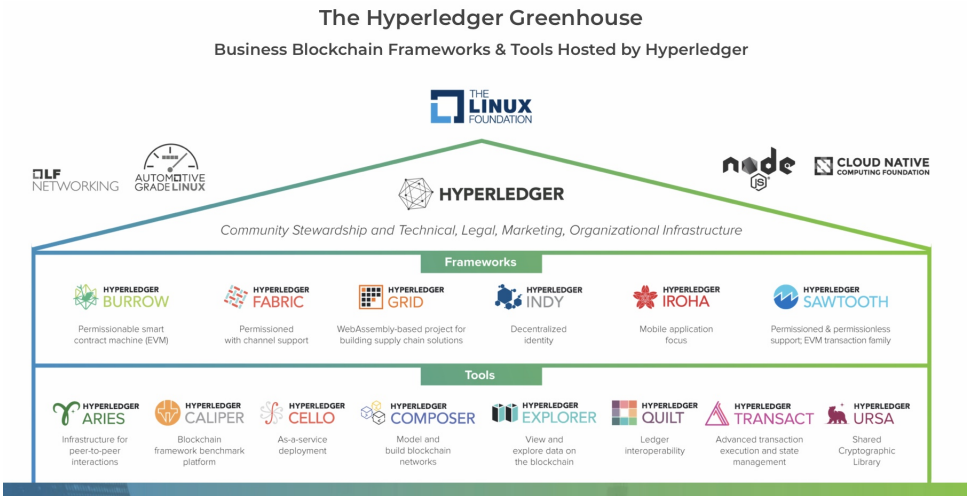


Figure: Hyperldger products [37]

In the following table, the main features of the three Blockchains just described are summarized.

Blockchain	**Application**	**Smart contract execution**	**Smart contract language**	**Data model**	**Consensus**
Bitcoin	Crypto-currency	Native	Golang,C++	Transaction-based	PoW
Ethereum	General	EVM	Solidity,Serpent,LLL	Account-based	PoW
Hyperledger(v0.6.0)	General	Dockers	Golang,Java	Key-value	PBFT
Hyperledger((v1.0.0))	General	Dockers	Golang,Java	Key-value	Ordering service (Kafka)

Table 2.1: Comparison of Bitcoin, Ethereum and Hyperledger

Hyperledger Fabric

We decided to describe more in details Hyperledger Fabric, since we chose this product in our work. Fabric [38] "is an open-source Blockchain platform that overcomes the limitations of generic Permissioned Blockchains:

- Consensus is hard-coded within the platform.
- The trust model of transaction validation is determined by the consensus protocol and cannot be adapted to the requirements of the smart contract.
- Smart contracts must be written in a fixed, non-standard, or domain-specific language, which hinders wide-spread adoption and may lead to programming errors.
- The sequential execution of all transactions by all peers limits performance, and complex measures are needed to prevent denial-of-service attacks against the platform originating from untrusted contracts (such as accounting for runtime with “gas” in Ethereum).
- Transactions must be deterministic, which can be difficult to ensure programmatically.
- Every smart contract runs on all peers, which is at odds with confidentiality, and prohibits the dissemination of contract code and state to a subset of peers.

Fabric introduces a new Blockchain architecture aiming at resiliency, flexibility, scalability, and confidentiality. Designed as a modular and extensible general-purpose Permissioned Blockchain, without systemic dependency on a native cryptocurrency. It is the first Blockchain system to support the execution of distributed applications written in standard programming languages (e.g., Go, Java, Node.js), in a way that allows them to be executed consistently across many nodes, giving impression of execution on a single globally-distributed Blockchain computer. Fabric securely tracks its execution history in an append-only replicated ledger data structure.

How Fabric works

Fabric introduces the *execute-order-validate* Blockchain architecture (described in the following picture).

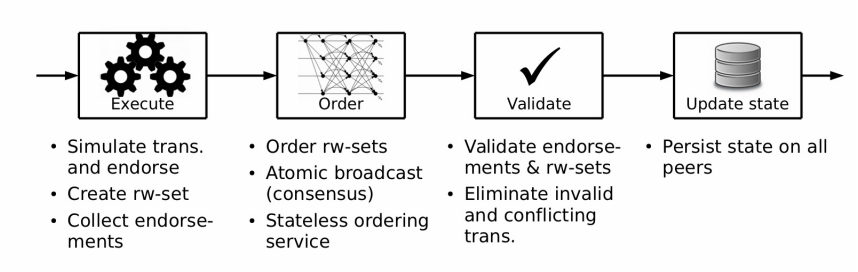


Figure: Execute-order-validate architecture of Fabric [38]

The Ledger component at each peer maintains the ledger and the state on persistent storage and enables simulation, validation, and ledger-update phases.
A distributed application for Fabric consists of two parts:

- A smart contract, called *chaincode*, which is a program code that implements the application logic and runs during the *execution* phase.
- An endorsement policy that is evaluated in the *validation* phase. Only designated administrators may have a permission to modify endorsement policies through system management functions.

In the **execution phase**, clients sign and send the transaction proposal to one or more peers specified by the endorsement policy. A transaction proposal is composed by:

- submitting client ID
- payload:
  - operation
  - parameters
  - chaincode ID
  - nonce
  - transaction ID (derived from the client ID and the nonce)

The endorsers simulate the proposal without synchronization with other peers, by executing the operation on the specified chaincode (this runs in a Docker container, isolated from the main endorser process), which has been installed on the Blockchain. The state of the Blockchain is maintained by the Peer Transaction Manager (PTM) in the form of a versioned key-value store (the version number is monotonically increased). The chaincode does not maintain the local state in the program code, but only in the blockchain state that is accessed with *GetState*, *PutState*, and *DelState* operations. After the simulation, the endorser cryptographically signs a

message called *endorsement*, which contains readset (version dependencies of the proposal simulation) and writeset (state updates produced by simulation), together with metadata such as transaction ID, endorser ID, and endorser signature, and sends it back to the client in a proposal response. The client collects endorsements until they satisfy the endorsement policy of the chaincode. Then, the client proceeds to create the transaction and passes it to the ordering service.
The **ordering phase** establishes a total order on all submitted transactions per channel: it orders atomically broadcast endorsements and thereby establishes consensus on transactions, despite faulty orderers. The ordering service groups multiple transactions into blocks and outputs a hash-chained sequence of blocks containing transactions and ensures that the delivered blocks on one channel are totally ordered.
A new block then enters the **validation phase** which consists of three sequential steps:

1. The endorsement policy evaluation occurs in parallel for all transactions within the block. The evaluation is the task of the Validation System Chaincode (VSCC), a static library that is responsible for validating the endorsement with respect to the endorsement policy configured for the chaincode. If the endorsement is not satisfied, the transaction is marked as invalid and its effects are disregarded.

2. A read-write conflict check is computed for all transactions in the block sequentially, comparing the versions of the keys in the readset field to those in the current state of the ledger (locally stored by the peer), and ensuring they are still the same. If the versions do not match, the transaction is marked as invalid and its effects are disregarded.

3. In the ledger update phase the block is appended to the locally stored ledger and the Blockchain state is updated. All state updates are applied by writing all key-value pairs in writeset to the local state.

The ledger of Fabric contains all transactions, including those that are deemed invalid because the validation is done by the peers' post-consensus. This is useful in those applications where it is required to track the invalid transactions. Since Fabric is a Permissioned Blockchain, detecting clients that try to mount a DoS attack by flooding the network with invalid transactions is easy. Hereunder we summarized the Fabric general transaction processing protocol:

1. Clients create a transaction and send it to endorsing peers.

2. Endorsing peers simulate transactions and produce an endorsement signature.

3. Clients collect and assemble endorsements into a transaction: the transaction

proposal.

4. Clients broadcast the transaction proposal to the ordering service.
5. The blocks of envelopes are delivered to the peers on the channel.
6. Peers append the received block to the channel's Blockchain".

A Byzantine Fault-Tolerant Ordering Service

As we saw a key property of Fabric is its extensibility, and in particular the support for multiple ordering services to build the blockchain. Nonetheless, the version 1.0 was launched in early 2017 without an implementation of a Byzantine fault-tolerant (BFT) [39] ordering service, that guarantees that eventually there exists a round with a correct proposer that will bring the system into a univalent configuration [40]. A Byzantine fault is a condition of a computer system, particularly in distributed computing systems, where components may fail and there is imperfect information on whether a component has failed. To overcome this limitation, Sousa et al. [26] designed, implemented, and evaluated a BFT ordering service for Fabric, implementing also optimizations for wide-area deployment. Their results show that Fabric with their ordering service can achieve up to ten thousand transactions per second and write a transaction irrevocably in the Blockchain in half a second, even with peers spread on different continents.

2.3.8 Consensus

The content of the Ledger is composed by historical and current states maintained by the Blockchain. Since the ledger is replicated, updates to it must be agreed on by all parties: it means that multiple parties must come to a consensus.
As we saw, one characteristic of a Blockchain system is that the nodes do not trust each other: some nodes may behave in Byzantine manners. The consensus protocol must therefore tolerate Byzantine failures. In the literature there is a vast choice among distributed consensus protocols, and there are many variants of previously proposed ones [25]. In extreme cases, we find purely computation based protocols that use proof of computation to randomly select a node which single-handedly decides the next operation. For instance Bitcoin uses proof-of-work (PoW), used in public settings. In other extreme cases, we find purely communication based protocols in which nodes have equal votes and go through multiple rounds of communication to reach consensus. These protocols, for instance PBFT, are used in private settings because they assume authenticated nodes. In between these

extremes are hybrid protocols which aim to improve performance of PoW and PBFT. For instance:

- Proof-of-Elapsed-Time (PoET) [41] which eliminates expensive mining in PoW by leveraging trusted hardware, used in private settings.
- Proof-of-Authority (PoA) [42] which improves PBFT by pre-selecting a small set of trusted nodes that vote among themselves to reach consensus, used in private settings.
- Stellar [43] and Ripple [44] improve PBFT by executing consensus in smaller networks.

2.3.9 Smart Contracts

Blockchains may execute arbitrary, programmable transaction logic in the form of smart contracts, as exemplified by Ethereum [29]. The scripts in Bitcoin were a predecessor of the concept. A smart contract functions as a trusted distributed application and gains its security from the Blockchain and the underlying consensus among the peers. A smart contract [38] refers to the computation executed when a transaction is performed. It can be regarded as a stored procedure invoked upon a transaction. The inputs, outputs and states affected by the smart contract execution are agreed on by every node.
A good description is taken by [25]: "All blockchains have built-in smart contracts that implement their transaction logics. In crypto-currencies, for example, the built-in smart contract first verifies transaction inputs by checking their signatures. Next, it verifies that the balance of the output addresses matches that of the inputs. Finally, it applies changes to the states".

One way to categorize a smart contract system is by its language, another one is by its runtime environment.

2.3.10 Why we need a Blockchain?

In general [27], "using a permissionless or permissioned Blockchain only makes sense when multiple mutually mistrusting entities want to interact and change the state of a system, and are not willing to agree on an online trusted third party. In the following picture, a useful flow chart to ease the decision making process is presented: if no data need to be stored, no database is required at all and consequently neither a Blockchain as a form of database. The same reasoning is valid if only one writer exists: a regular database provides better performance in terms of throughput

and latency. If a Trusted Third Party (TTP) is available and it is always online, write operations can be delegated to it and it can function as verifier for state transitions. Otherwise, if the TTP is usually offline, it can function as a certificate authority in the setting of a Permissioned Blockchain. In case the writers all mutually trust each other, a database with shared write access is likely the best solution. On the contrary, if they do not trust each other, using a Permissioned Blockchain makes sense. According to the public verifiability requirement anyone can be allowed to read the state (Public Permissioned Blockchain), the set of readers may also be restricted (Private Permissioned Blockchain). If the set of writers is not fixed and known by the participants a suitable solution could be a Permissionless Blockchain. In a centralized systems, the performance in terms of latency and throughput is generally much better than in Blockchain systems, since Blockchains add additional complexity through their consensus mechanism. There is a tradeoff between decentralization (how well a system scales to a large number of writers without mutual trust) and throughput (how many state updates a system can handle in a given amount of time). In the decision process whether to use a Blockchain system or not, this tradeoff should be taken into account as well".

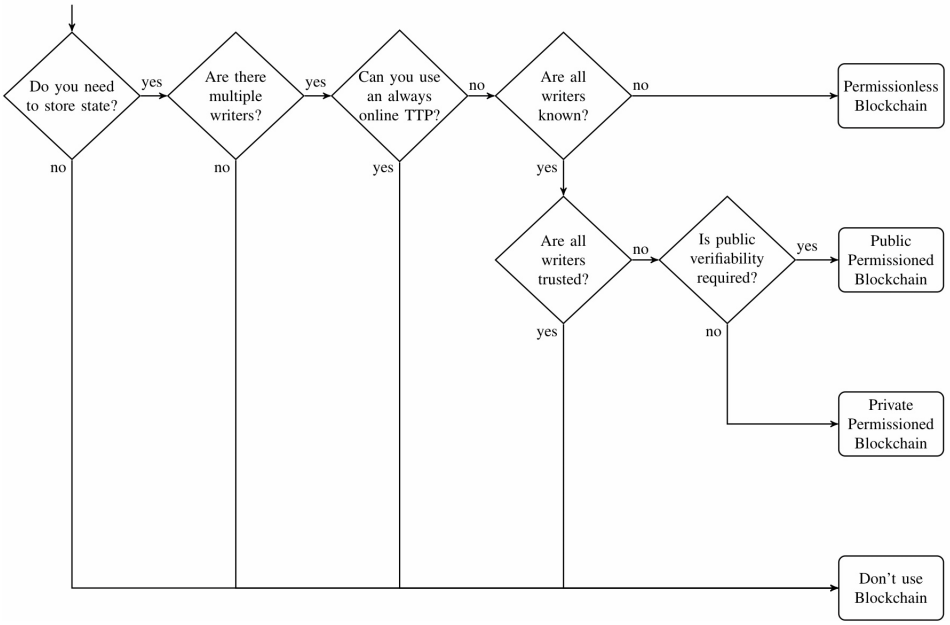


Figure: A flow chart to determine whether a Blockchain is the appropriate technical solution to solve a problem [27]

Below are the reasons why we would need a Blockchain [27]:

- **Public Verifiability**: in this way anyone can verify the correctness of the system's state. In a distributed ledger, each state transition is confirmed bya

restricted set of participants: verifier. Eventually, the same view of the ledger will be observed by all participants.

- **Transparency**: to guarantee the public verifiability, this characteristic has be present in the data and in the process of updating the state.
- **Privacy**:one of the most property in any system. There is an inherent tension between privacy and transparency.
- **Integrity**: it is necessary to ensure that information is protected from unauthorized modifications. This means that if a system provides public verifiability, anyone can verify the integrity of the data.
- **Redundancy**: this feature characterizes the data and it is reached through replication across the writers.
- **Trust Anchor**: this role represents the highest authority of a given system, having the power to grant and revoke read and write access to a system.

Chapter 3

Design and implementation

This chapter explains the architecture design of the prototype and the implementation of the different parts. section 3.5 concludes with a benchmark of the performance of a Login action, a Sign up action, a change password action and a change biography action without the use of the Blockchain in comparison with the same actions performed with the use of the Blockchain.

3.1 Architecture design

At first, the prototype was composed of two parts: the Mastodon instances and the Fabric network. The goal was the process of invoking the chaincodes in the Mastodon instances. However, this approach was quickly abandonned because of the complexity it added to the Mastodon application.

Instead, a microservice-like approach has been considered. A server module provides a RESTful API and acts as a gateway between the Mastodon application and the Fabric network, as reported inthe following picture.

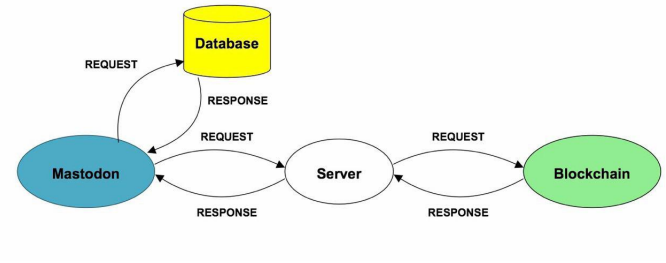


Figure: Architecture

In this way, the management of certificates and the connection to the network are independent of the Mastodon application which makes the implementation easier to maintain on the long term. Furthermore, it allows to write the server in Node.js and to use the native API provided by the Hyperledger Fabric project.

The prototype is then composed of three parts: the Mastodon application, the server and the Fabric network. There is a distinction between the Mastodon application and an instance. An instance groups the application, an implementation of the server and the peer that is the link with the network, i.e. a copy of the ledger and the world state.

3.2 Mastodon

The longest part of this work was the part related to Mastodon. As it always happens when you have to deal with a source code already written by someone else, it took time to understand and find the useful parts where to insert our modification.

The main difficulties were:

- Mastodon is designed in Ruby on Rails [20], a web application development framework written in the Ruby programming language. It is really useful when you want to create your web application, but at the same time if you never used it before, as in our case, it is complex to understand which parts were automatically created and which ones were written by a developer.
- We did not attend any courses on Ruby at University, thus we had to follow some tutorials to learn the syntax.
- In Ruby on Rails you can use the *Gems* [45], a sort of libraries: gems are plugins, additional functionalities of a product designed to fulfill specific goals. They are really useful because they let you perform many activities to customize your own application, but as already said in the first point, they make difficult to find what you need, especially when you are a beginner.
- Especially for the part of authentication that deals with the Login section and Sign up section (the parts that we needed to modify), they used a Gem called '*devise*' [46]. This Gem provides several controllers, what we needed were:
  - *registration_controller* to manage the Sign up action.

  - *confirmation_controller* to manage the activation of the new account, after the Sign up action, through the activation link on the email.
  - *session_controller* to manage the Login action.
- The Mastodon developers modified and customized these three controllers, it took time to understand the right behavior of the authentication process, since it mixed the original behavior of the *devise* controller with the customized one.

Thanks to the help of the logs, testing different situation like correct Login, Login with an incorrect password, new Sign up, etc. and analyzing in details the logs we finally found the correct places where to insert our modification. This section describes what we did, which additional gems we used and why we needed to modify these files. The files that we changed are the following four controllers:

- `confirmation_controller (in the folder mastodon\app\controllers\auth)`
- `registration_controller (in the folder mastodon\app\controllers\auth)`
- `session_controller (in the folder mastodon\app\controllers\auth)`
- `profiles_controller (in the folder mastodon\app\controllers\settings)`

For each API request (the API is described in section 3.3) on the Blockchain we used a gem: *faraday* that gives the possibility to perform API calls in parallel, too. [47] In the following subsections we describe for each controller what we modified and what the aim of each one is.

3.2.1 confirmation_controller

The source code is accessible at C.1.

When a user decides to Sign up on Mastodon he has to insert a username, an email, a password and a confirmation password. After the submission, an email is sent to his email account and a new account a new user are created, thanks to the confirmation link in the email, the user activates the account and he is redirected to his new profile.

What we have modified in this controller is the fact that, when the user clicks on the confirmation link, a GET request is sent to the Blockchain to test if the user is already present on the Blockchain. If he already exists, the parameters of the new account are not stored again on the Blockchain. Otherwise, in case he does not exist yet, the parameters are saved on the Blockchain.

3.2.2 registration_controller

The source code is accessible at C.2.

This controller is used in two different cases that we were interested in:

1. Build resources for the creation of a new account (as we saw in the previous subsection, before the activation, the new account is already set).
2. Change password.

What we added in this controller, to fit the first case to our solution, was a method that checks if the user is present on the Blockchain before creating a new account. If this is true, the new account is created with the same parameters that are present on the Blockchain - so far in our work, we store on it the email, the password, the displayed name and the biography. It would even be possible to store all the information related to an account. In case the user is not present on the Blockchain, the registration process is the original one and the account parameters will be stored on the Blockchain in the confirmation phase, as described in the previous subsection.

In the original Change password action, after having checked that this action is allowed, the password is modified and stored on the DB. We inserted a GET request that checks if the user is present on the Blockchain. Only if that is the case, the new password is encrypted using *bcrypt* gem, a hash algorithm for hashing passwords, and is sent to the Blockchain with a PUT request and stored on it. [48]

3.2.3 session_controller

The source code is accessible at C.3.

To better explain this part we use a flowchart that describes the Login process.

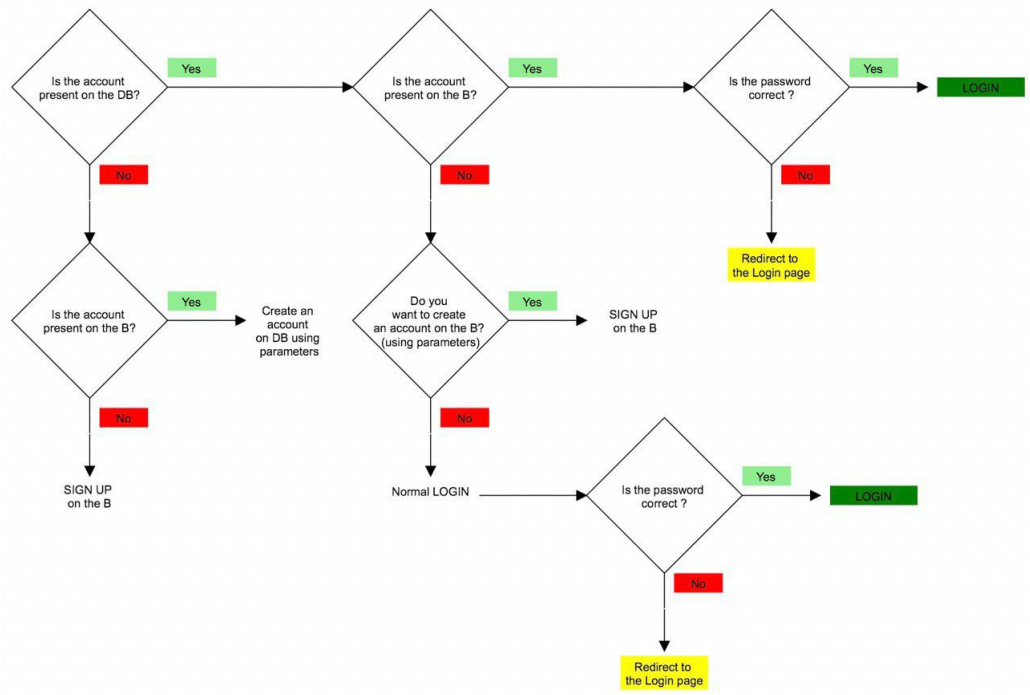


Figure: A flow chart to describe the process of Login
Legend: DB stays for Database, B stays for Blockchain

As we can note observing the flowchart, the first check is if the account is present on the Database (DB). If that is the case, it means that the user is trying to log in on an instance where his account already exists. Thus, a second check is performed to see if the account is present on the Blockchain (B). If this is true and the password inserted is correct, the user is available to access his profile. The second check is performed by a GET request that returns the email and the password, the check on the password is computed on the Mastodon side. If the password is incorrect, the user is redirected to the Login page, as usual when there is an incorrect password.

If the second check returns that the account is not present on the Blockchain, the user performs the normal login and on his page appears a message in red that suggests the user to move his account onto the Blockchain, and if he wants, to perform a new Sign up with the same username, email, password and confirmation password.

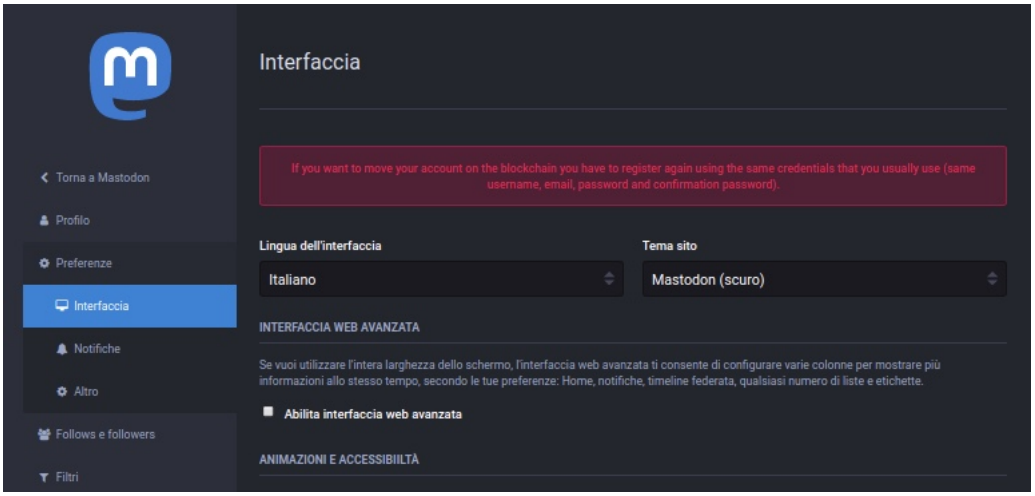


Figure: Advertising

In the current version of our project, to do this, the user has to create a new account from another instance, because he is already present on the Database. This could be a future feature and we explain better how this could be implemented in chapter 4.

In case the account is not present on the Database, this means that the user is trying to login from another instance for the first time. If he already has got an account stored on the Blockchain, he can create a new account on the Database of the new instance using the same parameters stored on the Blockchain (email, password, displayed name and biography), otherwise he will create a new account that will be stored on the Blockchain and consequently also on the Database.

3.2.4 profiles_controller

The source code is accessible at C.4.

This controller is used to update some fields of the profile, for instance the biography or the displayed name. We decided to take these two fields to store on the Blockchain, as already mentioned in the previous subsections. At this point we implemented also the part about the updating them. In this way, when a user registered on the Blockchain decides to modify the displayed name and/or the biography, these modifications are also updated on the Blockchain.

3.3 Server

This section describes the implementation of the server. When a call occurs, the server establishes a connection with the network and gets the smart contract. Then, it is able to evaluate or submit a transaction to the ledger according to the request.

The code is written in JavaScript on the platform Node.js with the framework Express.js. The module `fabric-network` from the Hyperledger Fabric SDK for Node.js [49] provides an API to interact with the network.

Here are the API calls allowed by the gateway.

Get a user's data Get a user's data stored on the ledger given their email.

URL	`/authentication/:email`
Method	`GET`
URL Params	`email=[string]`
Success	200
Error	404 (Not Found)

Success response payload:

```
{
  "success": true,
  "message": "Transaction has been evaluated",
  "result": { ... }
}
```

The `result` field has the following attributes:

- `display_name`: the username displayed in Mastodon
- `docType`: the type of the data stored on the ledger. At the moment, only the document type "credentials" is implemented.
- `email`
- `note`: the bio of the user
- `password`: the encrypted password

Register a user Register a user on the ledger.

URL	`/authentication/new/:email`
Method	`POST`
URL Params	`email=[string]`
Success	200
Error	404

The request must contain a body in the JSON format with the following fields:

- `display_name`: the username displayed in Mastodon
- `note`: the bio of the user
- `encrypted_password`: the encrypted password

The fields `display_name` and `note` can be left empty.

Change password Change a user's password.

URL	`/authentication/:email`
Method	`PUT`
URL Params	`email=[string]`
Success	200
Error	404

The request must contain a body in the JSON format with the field `encrypted_password`.

Update information Update a user's information.

URL	`/account/:field/:email`
Method	`PUT`
URL Params	`field=["updateDisplayName" | "updateNote"]` `email=[string]`
Success	200
Error	404

The request must contain a body in the JSON format with one of the following fields according to the value of the parameter `field`:

- `display_name`: the username displayed in Mastodon
- `note`: the bio of the user

3.4 Fabric network

The third component of the architecture is the Hyperledger Fabric network, let's call it *Mastochain*. The development consists of three parts:

- the configuration: this includes the topology of the network and the generation of the cryptographic materials;
- the smart contract: this is an interface that defines a data structure and methods to interacts with;
- deployment and maintenance.

3.4.1 Configuration

For the configuration two YAML files are needed: `crypto-config.yaml` and `configtx.yaml`.

The first one is used to generate cryptographic materials, i.e. certificates that will identify the several entities on the network. It is possible to generate a static amount of certificates for users or use the Hyperledger Fabric Certificate Authority (Fabric CA) to generate certificates dynamically. In the context of this thesis, the network is static because two instances are enough to demonstrate the concept. However, it will be critical to use Fabric CA in a live system where the topology of the network is not known.

In the same context, other implementation choices are made. Since the network is running locally, one orderer is enough but this must never be the case with a production purpose. The reason is that any information that has not been written yet to the blockchain stays in the orderer memory. Thus, if the orderer fails, information is lost. Note that using one orderer is called solo ordering in the configuration. In production, Kafka ordering is used. This applies to the number of peers as well. In the implementation each organization has one peer but it is highly recommended to have more to avoid a single point of failure.

The cryptographic materials are generated thanks to the `cryptogen` tool.

The second file is used to generate the Genesis block of the ledger and channel configuration transactions. This composes the shared configuration of the network and those transactions are generated using the `configtxgen` tool.

3.4.2 Smart Contract

In Hyperledger Fabric, smart contract defines the transaction logic that controls the lifecycle of an object contained in the world state. It is then packaged into a chaincode which is deployed to the network.

In the *Mastochain* network, the object that represents the credentials of a user is defined as a JSON object with the fields `email`, `password`, `display_name` and `note` already described in section 3.3. Each entry is identified by a key that must be unique on the network. The email of the user fits this requirement of uniqueness. Then, it is used as a key to retrieve the record.

Here are the methods defined in the chaincode:

- `initLedger`: init the ledger with one or more object. In the scope of the demo, the admin user defined in Mastodon is added.
- `signUp`: register a user on the ledger.
- `queryCredentials`: get a user's data, i.e. all this user's information stored on the ledger.
- `changePassword`: change the password of a user.
- `updateDisplayName`: update the display name of a user.
- `updateNote`: update the biography of a user.

The API for chaincode development in Node.js provided by Hyperledger Fabric [50] defined two methods in the chaincode interface: the `Init` and `Invoke` methods.

Their implementation is mandatory and their role is to init the ledger as explicitly mentioned in the name of the former and to process requests to evaluate or submit a transaction for the latter.

3.4.3 Deployment and maintenance

Once cryptographic materials have been generated, the orderer, the peers and their associated CouchDB representing the world state are deployed in Docker containers. Scripts are used to automate the process to make it easier and faster.

3.5 Time evaluation with vs without Blockchain

This section has the aim to compare the time used to perform four different actions using the Blockchain technology that we proposed in our solution and without it, as it works in the current original version. The four actions that we tested are:

- Confirmation of the account, after Sign up
- Login
- Change password
- Update biography

The tests were performed with an HP-250-G4-Notebook-PC, with a processor Intel® Core™ i5-5200U CPU @ 2.20GHz x 4 using a Ubuntu 18.04.2 LTS as operating system. The laptop has a disk of 250GB, partitioned by Windows(200GB)/Ubuntu(50GB) and a RAM of 8GB. The connection leveraged an hotspot 4G TIM - Telecom Italia Mobile.

As we can observe from the tables the average times of the solution with Blockchain are higher than the others, but these results are expected because for each action it is necessary to check if the account is already in the Blockchain. The user experience, by the way, is not reduced.

3.5.1 Confirmation of the account, after Sign up

Registered on the B	HomeController#index	Emojis Controller	HomeController#show	NotificationController	Lists Controller	Filters Controller	Web Settings	Total Time (ms)	Average Time (ms)
3190	483	29	231	263	403	407	80	5086	3550,4
2607	219	19	29	26	25	13	55	2993	
2536	159	22	20	116	43	33	326	3255	
2741	186	33	52	114	70	20	47	3263	
2711	229	15	30	23	42	60	45	3155	

Figure: Confirmation of the account, after Sign up with Blockchain

ConfirmationsController	HomeController#index	EmojisController	ListsController	NotificationsController	HomeController#show	FiltersController	Web Settings	Total Time (ms)	Average Time (ms)
239	568	51	47	27	244	136	187	1499	664,6
58	254	27	14	19	28	12	99	511	
62	24	23	23	24	34	16	39	245	
58	168	13	55	209	41	46	52	642	
60	141	21	27	60	60	18	39	426	

Figure: Confirmation of the account, after Sign up without Blockchain

3.5.2 Login

Session Controllers	HomeController#index	Emojis Controller	HomeController#show	NotificationController	Lists Controller	Filters Controller	ManifestController	HomeController#show	NotificationsController	Total Time (ms)	Average Time (ms)
467	223	15	42	46	18	23	18	25	28	905	855
478	184	15	44	30	16	22	33	29	27	878	
464	176	22	24	32	12	19	11	32	32	824	
554	189	11	36	22	19	18	14	30	24	917	
403	147	13	24	32	25	22	20	37	28	751	

Figure:Login with Blockchain

SessionsController#create	HomeController#index	EmojisController	NotificationsController	HomeController#show	ListsController	FiltersController	Total Time (ms)	Average Time (ms)
219	130	14	50	91	17	16	537	529,2
221	204	15	27	42	16	16	541	
254	194	16	27	33	16	17	557	
201	173	11	26	29	20	14	474	
267	166	15	29	25	18	17	537	

Figure: Login without Blockchain

3.5.3 Change password

Registered on the B	registrationsController	Total Time (ms)	Average Time (ms)
3304	257	3561	3275,6
2922	164	3086	
2992	205	3197	
2988	232	3220	
3097	217	3314	

Figure:Change password with Blockchain

RegistrationsController#update	RegistrationsController#edit	Total Time (ms)	Average Time (ms)
310	145	455	529,2
248	201	449	
261	196	457	
247	169	416	
332	184	516	

Figure: Change password without Blockchain

3.5.4 Update bibliography

Updated on B	ProfilesController	Total Time (ms)	Average Time (ms)
5391	189	5580	5459,8
5313	288	5601	
5059	271	5330	
5232	187	5419	
5179	190	5369	

Figure:Update bibliography with Blockchain

RegistrationsController#update	ProfilesController#show	Total Time (ms)	Average Time (ms)
163	310	473	410,8
73	227	300	
206	371	577	
62	205	267	
69	368	437	

Figure: Update bibliography without Blockchain

3.5.5 Comparison of average times

In the following table we compared the time of the previous four actions. As we can observe, the longest is related to the update of the bibliography. On the contrary the Login action is the fastest and only one time and a half slower than usual Login.

With Blockchain	Without Blockchain	Ratio
3550,4	664,6	5,3
855	529,2	1,6
3275,6	529,2	6,2
5459,8	410,8	13,3

Figure: Comparison of average times

Chapter 4

Further development

This chapter gathers a collection of potential enhancements of the prototype described in this thesis.

4.1 Feedback from the Mastodon community

The strength of open source projects is the multitude of creative individuals which composes the community. Submitting this prototype to them could lead to meaningful feedbacks for further development that would combine decentralized social networks and blockchain-based identity management.

This could help to highlight the flaws in the prototype and in which way it does not resolve the open issue that this project aimed to solve.

4.2 Uploading existing accounts on the ledger

Taking into account that users could already have an account on the Database of the instances where they are registered, we possibly face a problem. So far, if a user has got an account only on a Database, but not on the Blockchain, to move his credentials on the Blockchain he has to create a new account from another instance, because otherwise he gets an error message that says that he is already present on Mastodon. A feature to implement is how to fix this inconvenient: to do this we could implement a mechanism that authenticates that the user is the one who he says to be. In this case, the system could let him simulate a Sign up action with the same credentials of his account that does not create a new one

on the Database, but sends the parameters to the Blockchain. The mechanism to create the account on the Blockchain already works, so what we would need is a way to skip another creation of the account on the Database, but only if the user is authenticated in some way.

4.3 Account deletion

Another feature is related to the deletion of an account from the Blockchain. In Mastodon, they use a field called *AccountStat* that defines the status of an account, for instance if it is blocked or not confirmed yet. We could use a similar strategy, storing the status of the account. In this way, if a user decides in the future to create again an account on an instance, he will be able to use the same credentials already stored on the Blockchain, setting the field to "*Functional*".

4.4 Using an other key than the email

In the initial design, we thought about identifying a user by a unique public key and use a DNS-like system to link it to a readable name. This idea was put aside to focus on an actual prototype that will allow further iteration but we lack time to improve the system and continue on this path.

This implies challenges like the uniqueness of a username. It is possible to constraint the user to choose a name that does not exist yet but this could be frustrating if the user wants a specific name. Furthermore, since the ledger is immutable, usernames that are not used anymore could be locked forever. An idea could be to add a suffix automatically generated that is easy to remember like a four digit number. This number could be derived from the public key with a custom hash function, etc. Then, a user could be identified as <username>#<number> where the username is not unique but the concatenation with the number is.

Chapter 5

Related works

Nowadays the problem of the privacy of personal data on Social networks is gaining in importance and some people started to use the Blockchain to struggle with it. The recent increase in reported incidents of surveillance and security breaches compromising users' privacy call into question the current model, in which third-parties collect and control massive amounts of personal data. MIT Media Lab employed Blockchain to describe a decentralized personal data management system (i.e. Decentralizing Privacy) that ensures users own and control their data without authentication from a third party. [51]

They implemented a protocol that turns a Blockchain into an automated access-control manager that does not require trust in a third party. Unlike Bitcoin, transactions in their system are not strictly financial – they are used to carry instructions, such as storing, querying and sharing data. Finally, they discussed possible future extensions to block chains that could harness them into a well-rounded solution for trusted computing problems in society.

Fu et al. employed a better encryption algorithm from NTT (Nippon Telegraph and Telephone) Service Evolution Laboratory to enforce the "Decentralizing Privacy". [52] Instead of using Proof-of-Work (PoW) for protection, they employed Proof-of-Credibility Score to improve the previous system and analyzed attack situations.

Since this issue is quite new and challenging it is not so easy to find many other works related to ours.

Chapter 6

Conclusion

In this thesis, we hope to have contributed to the promissing open source project that is Mastodon, and to have developed solutions to existing problems with a cutting-edge technology, the blockchain.

In chapter 1, we have introduced the problem of identity migration in Mastodon and the challenges that the developers who are working on the issue are facing. We have gathered and summarized their thoughts to provide a clear definition of the problem.

Then in chapter 2, we have made a state of the art of distributed technologies and have explained why those technologies could help to solve the issue.

In chapter 3, we have developed a prototype based on Hyperledger Fabric that allows to authenticate to different instances of Mastodon with credentials stored on the blockchain.

Further development of the prototype are exposed in chapter 4. We have provided an overview of related work on the topics of this thesis in chapter 5.

Finally, we are aware that the prototype developed in this thesis is yet to be further developed in order to resolve the initial use case. However, we hope that our work provided an interesting use case of these technologies, on which to base further work.

Bibliography

1. CONFESSORE, Nicholas. Cambridge Analytica and Facebook: The Scandal and the Fallout So Far. *The New York Times* [online]. 2018 [visited on 2019-08-16]. Available from: https://www.nytimes.com/2018/04/04/us/politics/cambridge-analytica-scandal-fallout.html.
2. ROSENBERG, Matthew; CONFESSORE, Nicholas; CADWALLADR, Carole. How Trump Consultants Exploited the Facebook Data of Millions. *The New York Times* [online]. 2018 [visited on 2019-08-16]. Available from: https://www.nytimes.com/2018/03/17/us/politics/cambridge-analytica-trump-campaign.html?module=inline.
3. CADWALLADR, Carole; GRAHAM-HARRISON, Emma. Revealed: 50 million Facebook profiles harvested for Cambridge Analytica in major data breach. *The Guardian* [online]. 2018 [visited on 2019-08-16]. Available from: https://www.theguardian.com/news/2018/mar/17/cambridge-analytica-facebook-influence-us-election.
4. ROCHKON, Eugen. *Mastodon* [online] [visited on 2019-08-16]. Available from: https://joinmastodon.org/.
5. CHENET, Carl. You Should Not Ignore the Mastodon Social Network Any More. *Carl Chenet's Blog* [online]. 2019 [visited on 2019-08-16]. Available from: https://carlchenet.com/do-not-ignore-the-mastodon-social-network/.
6. WILLIS, Tomas Charles; CASSEL, Brian Donald; MIELKE, Adam. *Email and identity migration based on multifactor relationship data provided by users and systems*. Patent, US10109022B2. [Visited on 2019-08-16]. Available from: https://patents.google.com/patent/US10109022B2/en.
7. *Sovrin* [https://sovrin.org/].
8. *Blockstack* [https://blockstack.org/].
9. *uPort: Tools for Decentralized Identity and Trusted Data* [https://www.uport.me/].

10. *Decentralized Input Output System* [https://iodigital.io/dions/].

11. *MyData* [https://mydata.org/].

12. *Decentralization* [https://www.merriam-webster.com/dictionary/decentralization]. Merriam-Webster.

13. *Facebook users worldwide 2019* [https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/].

14. BOYD, Danah M.; ELLISON, Nicole B. Social Network Sites: Definition, History, and Scholarship. *Journal of Computer-Mediated Communication*. 2007, vol. 13, no. 1, pp. 210–230. ISSN 1083-6101. Available from DOI: 10.1111/j.1083-6101.2007.00393.x.

15. DATTA, Anwitaman; BUCHEGGER, Sonja; VU, Le-Hung; STRUFE, Thorsten; RZADCA, Krzysztof. Decentralized Online Social Networks. In: *Handbook of Social Network Technologies and Applications*. Ed. by FURHT, Borko. Boston, MA: Springer US, 2010, pp. 349–378. ISBN 978-1-4419-7142-5. Available from DOI: 10.1007/978-1-4419-7142-5_17.

16. GUIDI, Barbara; CONTI, Marco; PASSARELLA, Andrea; RICCI, Laura. Managing social contents in Decentralized Online Social Networks: A survey. *Online Social Networks and Media*. 2018, vol. 7, pp. 12 –29. ISSN 2468-6964. Available from DOI: https://doi.org/10.1016/j.osnem.2018.07.001.

17. BAHRI, Leila; CARMINATI, Barbara; FERRARI, Elena. Decentralized privacy preserving services for Online Social Networks. *Online Social Networks and Media*. 2018, vol. 6, pp. 18 –25. ISSN 2468-6964. Available from DOI: https://doi.org/10.1016/j.osnem.2018.02.001.

18. *Decentralized Identifiers (DIDs) v0.13* [https://w3c-ccg.github.io/did-spec/].

19. MUNEEB, Ali; RYAN, Shea; NELSON, Jude. *Decentralized processing of global naming systems*. Patent, US20170236123A1. [Visited on 2019-08-16]. Available from: https://patents.google.com/patent/US20170236123A1/en.

20. HEINEMEIER HANSSON, David et al. *Ruby on Rails* [online]. 2005. Version 2.6.1 [visited on 2019-08-17]. Available from: https://rubyonrails.org.

21. *ActivityPub W3C Recommendation 23 January 2018* [https://www.w3.org/TR/activitypub/].

22. *Postgresql: The World's Most Advanced Open Source Relational Database* [https://www.postgresql.org/].

23. *Mastodon User Guide* [https://web.archive.org/web/20170409030653/http://mastoguide.info/Pages/fedFAQ.html].

24. HOLLOWAY, James. *What on Earth is the fediverse and why does it matter?* [https://newatlas.com/what-is-the-fediverse/56385/]. New Atlas, 2018.

25. DINH, Tien Tuan Anh; LIU, Rui; ZHANG, Meihui; CHEN, Gang; OOI, Beng Chin; WANG, Ji. Untangling blockchain: A data processing view of blockchain systems. *IEEE Transactions on Knowledge and Data Engineering*. 2018, vol. 30, no. 7, pp. 1366–1385.

26. SOUSA, Joao; BESSANI, Alysson; VUKOLIC, Marko. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2018, pp. 51–58.

27. WÜST, Karl; GERVAIS, Arthur. Do you need a Blockchain? In: *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. 2018, pp. 45–54.

28. NAKAMOTO, Satoshi et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

29. WOOD, Gavin. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*. 2014, vol. 151, pp. 1–32.

30. BECKER, Georg. Merkle signature schemes, merkle trees and their cryptanalysis.

31. DWORK, Cynthia; NAOR, Moni. Pricing via Processing or Combatting Junk Mail. In: BRICKELL, Ernest F. (ed.). *Advances in Cryptology — CRYPTO' 92*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 139–147. ISBN 978-3-540-48071-6.

32. WHITTLE, Ben. *What Is a Nonce? A No-Nonsense Dive into Proof of Work* [https://www.bitcoininsider.org/article/52287/what-nonce-no-nonsense-dive-proof-work]. 2018.

33. VU, Quang Hieu; LUPU, Mihai; OOI, Beng Chin. *Peer-to-peer computing: Principles and applications*. Springer Science & Business Media, 2009.

34. MEIKLEJOHN, Sarah; POMAROLE, Marjori; JORDAN, Grant; LEVCHENKO, Kirill; MCCOY, Damon; VOELKER, Geoffrey M; SAVAGE, Stefan. A fistful of bitcoins: characterizing payments among men with no names. In: *Proceedings of the 2013 conference on Internet measurement conference*. 2013, pp. 127–140.

35. INC., Chainalysis. *Blockchain analysis* [https://www.chainalysis.com/].

36. DANIELS, Arnold. *The rise of private permissionless blockchains - part 1* [https://medium.com/ltonetwork/the-rise-of-private-permissionless-blockchains-part-1-4c39bea2e2be]. LTO Network, 2018.

37. *Hyperledger: Open Source Blockchain Technologies* [https://www.hyperledger.org/].

38. ANDROULAKI, Elli et al. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In: *Proceedings of the Thirteenth EuroSys Conference*. 2018, p. 30.

39. CASTRO, Miguel; LISKOV, Barbara. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*. 2002, vol. 20, no. 4, pp. 398–461.

40. BUCHMAN, Ethan; KWON, Jae; MILOSEVIC, Zarko. The latest gossip on BFT consensus. *arXiv preprint arXiv:1807.04938*. 2018.

41. HYPERLEDGER. *hyperledger/sawtooth-core* [https://github.com/hyperledger/sawtooth-core/blob/master/docs/source/architecture/poet.rst].

42. PARITYTECH. *paritytech/wiki* [https://github.com/paritytech/wiki/blob/master/Proof-of-Authority-Chains.md].

43. *Stellar: Make Money Better* [https://www.stellar.org/].

44. RIPPLE, Team. *Ripple: One Frictionless Experience To Send Money Globally* [https://www.ripple.com/].

45. *What is a gem* [https://guides.rubygems.org/what-is-a-gem/].

46. PLATAFORMATEC. *Devise* [online]. 2019. Version 4.6 [visited on 2019-08-17]. Available from: https://github.com/plataformatec/devise.

47. *Faraday*. Available also from: https://lostisland.github.io/faraday/.

48. *bcrypt* [https://rubygems.org/gems/bcrypt/versions/3.1.12].

49. FOUNDATION, The Linux [online]. Version 1.4 [visited on 2019-08-17]. Available from: https://fabric-sdk-node.github.io/release-1.4/.

50. FOUNDATION, The Linux [online]. Version 1.4 [visited on 2019-08-18]. Available from: https://fabric-shim.github.io/release-1.4/.

51. ZYSKIND, Guy; NATHAN, Oz, et al. Decentralizing privacy: Using blockchain to protect personal data. In: *2015 IEEE Security and Privacy Workshops*. 2015, pp. 180–184.

52. FU, Dongqi; FANG, Liri. Blockchain-based trusted computing in social network. In: *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*. 2016, pp. 19–22.

53. *Activity Streams 2.0* [https://www.w3.org/TR/activitystreams-core/].

54. *OStatus* [https://www.w3.org/community/ostatus/wiki/Main_Page].

55. *Salmon Protocol* [http://www.salmon-protocol.org/home].

56. *The Salmon Protocol - specifications* [http://deadspecs.work/salmon-protocol/draft-panzer-salmon-00.html].

57. *Zot Communications Protocol* [https://wiki.p2pfoundation.net/Zot_Communications_Protocol].

58. The New York Times. A. G. Sulzberger. Available also from: https://www.nytimes.com/.

Appendix A

Definitions

- **ActivityStream 2.0**" is suitable as a social data syntax. It forms part of the Social Web Protocols suite of related standards.
  The JSON Activity Streams 1.0 specification was published in May of 2011 and provided a baseline extensible syntax for the expression of completed activities. This specification builds upon that initial foundation by incorporating lessons learned through extensive implementation, community feedback and related ongoing work from a variety of other communities. Some issues motivated the evolution of Activity Streams 2.0 from Activity Streams 1.0." [53]
- **OAuth** "is an open standard for access delegation, commonly used as a way for Internet users to grant websites or applications access to their information on other websites but without giving them the passwords. This mechanism is used by companies such as Amazon, Google, Facebook, Microsoft and Twitter to permit the users to share information about their accounts with third party applications or websites. Generally, OAuth provides to clients a "secure delegated access" to server resources on behalf of a resource owner. It specifies a process for resource owners to authorize third-party access to their server resources without sharing their credentials. Designed specifically to work with Hypertext Transfer Protocol (HTTP), OAuth essentially allows access tokens to be issued to third-party clients by an authorization server, with the approval of the resource owner. The third party then uses the access token to access the protected resources hosted by the resource server."
- **OStatus** [54] "is an open standard for federated microblogging, allowing users on one website to send and receive status updates with users on another website.The standard describes how a suite of open protocols, including Atom, Activity Streams, PubSubHubbub, Salmon, and WebFinger, can be used

together, which enables different microblogging server implementations to route status updates between their users back-and-forth, in near real-time."

- **Pleroma** "is a microblogging server software that can federate (= exchange messages with) other servers that support the same federation standards (OStatus and ActivityPub). What that means is that you can host a server for yourself or your friends and stay in control of your online identity, but still exchange messages with people on larger servers. Pleroma will federate with all servers that implement either OStatus or ActivityPub, like Friendica, GNU Social, Hubzilla, Mastodon, Misskey, Peertube, and Pixelfed. Pleroma is written in Elixir, high-performance and can run on small devices like a Raspberry Pi."
- **PubSubHubbub** "is an open protocol for distributed publish/subscribe communication on the Internet. It generalizes the concept of webhooks and allows data producers and data consumers to work in a decoupled way. This protocol provides a way to subscribe, unsubscribe and receive updates from a resource, whether it's an RSS or Atom feed or any web accessible document (JSON...)."
- **Salmon** [55] "Conversations are becoming distributed and fragmented on the Web. Content is increasingly syndicated and re-aggregated beyond its original context. Technologies such as RSS, Atom, and PubSubHubbub allow for a real time flow of updates to readers, but this leads to a fragmentation of conversations. The comments, ratings, and annotations increasingly happen at the aggregator and are invisible to the original source. The Salmon Protocol is an open, simple, standards-based solution that lets aggregators and sources unify the conversations. It focuses initially on public conversations around public content. Federated social networks such as GNU Social and Diaspora use Salmon as defined in the OStatus specification to coordinate discussion between members belonging to different servers. A member of one server can publish an article which is disseminated to other users over the network via Salmon who in turn can comment back in a similar fashion. The main characteristics of this protocol are [56]:
  - Salmon: a signed entry.
  - Salmon generator:a service or agent that creates new salmon on behalf of a user.
  - Salmon endpoint: a URL to which a salmon is POSTed via HTTP.
  - Aggregator: a service or client which aggregates multiple streams of content from other services in order to present a unified view.

- Parent entry: an entry which can be the target of a reply, comment, or activity salmon
- Reply feed: a feed of entries which are replies, comments, or activities such as likes which depend for their context and semantics on a parent entry; a feed identified by a link rel="replies" on a parent entry.

The full flow for signing a request in this most general case is then:

- Generator obtains OAuth token for signing service via standard mechanisms (outside the scope of this document)
- Generator discovers the salmon-signer endpoint
- Generator POSTs an unsigned Atom Entry to the salmon-signer endpoint, with OAuth credentials.
- IdP checks credentials and content and signs the Salmon with the user's private key.
- IdP returns the signed application/magic-envelope salmon to the generator with a 200 OK response.
- Generator immediately sends the salmon to the desired final destination.

Salmon generators sign salmon. In the most common case, the Salmon generator is also the identity provider for the author URI. The Salmon generator may maintain keypairs on behalf of its users or use additional mechanisms to allow users to maintain their own private keys (while still publishing the public keys)."

- **WebFinger** "is a protocol specified by the Internet Engineering Task Force IETF that allows for discovery of information about people and things identified by a URI.Information about a person might be discovered via an "acct:" URI, for example, which is a URI that looks like an email address. The WebFinger protocol is used by the federated social networks GNU social, StatusNet and Diaspora] to discover users on federated nodes and pods as well as the remoteStorage protocol."
- **Zot!** [57] "is a communications protocol for social communications on the web. The protocol consists of two basic functions: send-message and remote-access. These functions are built on top of other web standards, such as webfinger/lrdd and atom/activitystreams. Communications are encrypted and both sides of the communication verified through crytpographic means before communication is allowed. Zot! does not prove identity. It verifies communications endpoints, and secures messages between those endpoints."

Appendix B

Proposed solutions by the community

This section is a collection of all the solutions proposed by the community.We grouped these in different sections:

- Solutions to the moving issue.
- How to notify to the followers the moving.
- Unique usernames

B.1 Solutions to the moving issue

- GitHub way, where the old username is now forwarded to the new username. The downside is that if the server goes down before a client has got the redirect they will be unavailable.
- Matrix way, in which you have identity servers that decouple the "real" ID (@username@instance) from the person whose ID it belongs to.
  The downside is that Matrix implements this in a very centralized way.
- A decoupled notion of identity. For example, a keybase.io profile aggregates different accounts for a single person so that they can be easily identified across all networks. If this were true, you could imagine a system where you follow someone on Mastodon by following an identity and all addresses (that is, instance/profile combinations) that are associated with it.

The downside is that it is not easy to maintain a trusted database of identities that is shareable across all federated nodes. Blockstack might be a good candidate for providing a decentralised, global identity. With Blockstack you could keep the same ID (and thus your followers) even if the instance you are currently using goes down permanently.
The problem with having separated identities is that it would not be compatible with other OStatus implementations (Mastodon is not the only implementation, GNU Social is another and there is quite a few more). Migration should be something that someone following a Mastodon user from GNU Social should be able to support.

- The two accounts can just reference one another, and that by itself is verification that they are under the control of the same person.

  User wants to migrate from @bob@roddenberry.zone to @robby@abrams.website User logs on to @robby@abrams.website and flips a setting saying they are preparing to migrate from @bob@roddenberry.zone @robby@abrams.website puts out a special toot that says "I'm migrating from @bob@roddenberry.zone"; most clients would just hide this toot. User logs on to roddenberry.zone and pushes the button on migrating to @abrams.website roddenberry.zone sends abrams.website a request saying "@bob, on my server, wants to migrate to @robby on yours, is that true?" abrams.website says "@robby on my server says he is migrating from @bob on yours" @bob@roddenberry.zone puts out a special toot that says "I'm migrating to @robby@abrams.website" Clients see that toot, check that its counterpart exists in the toot history of @robby@abrams.website, and automatically follow the new account. Optional extra steps:

  - Users that see the migration toot see a special notification, rather than a regular toot, and are given the choice to opt out.
  - Clients don't unfollow the "old" account for some period of time (24? 48 hours?). At that point, they verify that both origin and destination "migration toots" still exist; otherwise, they unfollow the new account. That means you can cancel a migration by deleting the special migration toot. Thus if an account is stolen, users can reset their password and prevent account migrations from happening.
  - roddenberry.zone exports bob's follow list and sends it along to abrams.website, which imports it.

  They do not solve the problem of migration, especially in the case of a server suddenly goes down.

- Mastodon could use HTTP, the equivalent of a 301 redirect. To be clear:
  - I choose to move from me@mastodon.social to ohai@mydomain.elsewhere
  - I tell the Mastodon instance at Mastodon.social to stop allowing posts there and to provide (in my mastondon.social profile) a redirect to ohai@mydomain.elsewhere.
  - Any other instances where people follow me at me@mastodon.social should update their subscription to point to ohai@mydomain.elsewhere
  - If I choose to migrate follower / followed lists, or even toots, that can be a separate API but (to the point of many here) migration is less important than portability, at least to start. Portability is a fundamental protocol requirement, portability is something that can be added later without core protocol support.

  No new crypto is needed, all the security boundaries continue to work. There is an obvious hijacking exploit possible in the event that someone gains control of an account, so it would probably make sense to have a SHOULD clause for following user redirects that instructs federated servers to check back with the original server for some period of time (e.g., 30 days) to allow the original user to regain control and cancel the redirect.

  The drawback of this method is that it does not work if an instance permanently goes down.
  To have a more complete feature, cryptography is required, so that a user is able to prove that he owns the account on the defunct instance. Other instances would know the account public key, only the user knows his private key. And as long as the instance is up, he should be able to update the key pair (in case it gets lost or compromised), which means the public key cannot be the account unique ID.

- Another proposal was to buy your own domain name, point your Mail eXchanger-equivalent DNS record; it might be also possible to setup an alias via WebFinger [Appendix A], instead of DNS, to point to the correct instance, then register the domain as an alias on that instance so that you can register yourname@yourdomain.com.

  Later, if you want to move instances for whatever reason, you can repeat the DNS/alias process on another server, and migrate your toots/following list over. Everyone still follows you and addresses you correctly without needing

an explicit "hey I moved instances" notice.

WebFinger would help for instances that are alive, but once they are taken down, that information will need to be stored and retrieved from somewhere else, for example using a mechanism that, when an instance is planning to shut down, the users could generate and send a token to the instance of their choice which would then become, once they log in with their token, their instance of record and they could migrate directly.

In this way you only need to change server while keeping your identity, and all the links would stay the same. Server migration then is just a DNS change (and some data transferring).
Unlike email providers, instances have different moderation rules, anti-harassment policies, they will gather different communities, some will choose not to federate with other instance that they consider harmful.

- The "easy" method to achieve the result, using the keys:
  - User wants to migrate from `@bob@roddenberry.zone` to `@robby@abrams.website`
  - User goes to abrams.website and has it give it a "migration key" (internally signed by @robby)
  - User goes to (original node) roddenberry.zone and presses "Migrate Account" and pastes in the key.
  - roddenberry.zone sends a signed notification to subscribers of @bob
  - Each subscriber sees the signed notification and verifies it is @bob, and discovers @robby@abrams.website, verifying that the key in the notification has been signed there by @robby
  - Each subscriber, once they verify, can elect to unsubscribe from @bob and subscribe to @robby

  Assuming you have the roddenberry.zone "migration key" and followers-list already backed-up, in case of the worst:

  - User backed up migration data from roddenberry.zone consistening of a "migration key" and followers list.
  - User wants to migrate from defunct @bob@roddenberry.zone to @robby@abrams.website
  - User goes to abrams.website and gives it the "migration key" from roddenberry.zone and the followers-list
  - abrams.website sends a signed notification to those on the list

  - Each subscriber sees the signed notification and verifies the "migration key" using a public key they already have on file
  - Each subscriber, once they verify, can elect to unsubscribe from @bob and subscribe to @robby.

  . The downside with the keys is that this makes the private key very sensitive ; if it leaks, you are screwed.

- Mastodon could use OAuth between instances to allow one instance to copy-/migrate the account and associated content over from the original instance, in a TCP way (it makes the migration a lot more secure):
  Connection:
  https://en.wikipedia.org/wiki/Transmission_Control_Protocol#Connection_establishment
  Migration:
  https://en.wikipedia.org/wiki/Transmission_Control_Protocol#Data_transfer
  Termination:
  https://en.wikipedia.org/wiki/Transmission_Control_Protocol#Connection_termination
  As with TCP, both ends can see the proxy, since there are two connections and they are both terminated at the proxy. This makes blocking such proxies trivial. The NAT, Network Address Translation, unlike proxying, is transparent to one end of the connection (but not both). This makes it harder to filter, but at the same time it is not an issue: it requires DNS spoofing which is trivially defeated by, among other things, TLS/HTTPS and DNSSEC. The flow could look like:
  - User from Old Instance arrives at New instance.
  - User activates the "Sign up" flow.
  - After creating credentials, a new step: "Migrate Account".
  - Enter the original instance URL.
  - OAuth flow is triggered, New Instance asks for read privileges on Old Instance (read-only prevents bad-actor instances from posting/deleting content on the original instance).
  - With read privileges, New Instance duplicates any account settings (bio, etc, perhaps this is configurable in the wizard).
  - New Instance queues a full content migration, in offline.
  - After content is migrated, user is notified that they can now delete their account on Old Instance.

The only downside is replies, etc could be broken unless the migration process literally updates the toots of all users on all instances that replied to your toots, that is impractical.

- Another proposal is related to the history/development of Friendica/Hubzilla: Hubzilla was split off from Friendica because of a desire to create a decentralized permissions system. Friendica, for the most part, is a social network that includes profile features like albums/calendars/etc, similar to Facebook. Hubzilla, on the other hand, is a permissions/ID system that happens to allow social networking. This means that account migration is easily achievable using Hubzilla's concept of "nomadic identity": your ID is referred to as a "channel" and takes the form handle@site.tld, similar to what OStatus / ActivityPub uses for its accounts. But the key difference is that your content is not tied to the DNS; accounts and channels are separate entities. In effect: you can import a "channel" (identity) from site1.tld to site2.tld seamlessly, either by uploading an export of your data, or by entering your account credentials on another site. This has the benefit of allowing live migration, as well as also syncing content back to the other connected accounts. In order for an identity to persist across locations, one must be able to provide or recover:
  - the globally unique ID (GUID) for that identity
  - the private key assigned to that identity
  - (if the original server no longer exists) an address book of contacts for that identity.

  This information will be exportable from the original server via API, and/or downloadable to disk or thumb-drive.

  - Bob signs up at site1.tld, a "hub" run by one of his friends.
  - Bob creates a personal channel bob@site1.tld and starts posting life updates.
  - Bob's friend announces that site1.tld will be shutting down temporarily in a week, and will be unavailable for a few months.
  - Bob wants to migrate to his personal server instead.
  - Bob creates his own hub at site2.tld, and during account signup, he chooses to import the channel bob@site1.tld.
  - Bob enters his credentials for site1.tld, and clones bob@site1.tld to bob@site2.tld seamlessly. Alternatively: Bob exports his channel bob@site1.tld to a file, which he uploads to site2.tld and his account there.

– Bob continues to make personal posts to bob@site2.tld, and those updates are sent back to site1.tld as well. When site1.tld goes offline, Bob can continue to post to bob@site2.tld uninterrupted. When site1.tld comes back online, it syncs all of the content from the downtime period.

– Bob might decide he does not want to use his site1.tld account anymore, so he can turn off syncing on site1.tld and delete his account there. Bob now lives at site2.tld, and publishes at bob@site2.tld. All of the channels that have previously connected with bob are, in actuality, transparently linked to the channel "bob", regardless of the domain name. All of the permissions remain intact.

All connections are stored in a JSON format and are linked to the channels themselves, not the accounts that own the channels. Additionally, it is completely transparent to every other channel that follows you, since they can comment or reply to your posts the same as before, and in fact, they can do this irrespective of domain name. In the example above, Bob posts to site2.tld, and the content is cloned across bob@site1.tld and bob@site2.tld with the zot protocol. Channels that comment on bob@site1.tld will show up to Bob on bob@site2.tld, and the same channel data is accessible through both DNS entries.
In terms of advantages and disadvantages: this is very useful in being resistant to service outages or DNS censorship. It also allows for complete identity portability, so you can move from one hub to another seamlessly. On the other hand, it might also encourage people to create accounts across many different hubs simply to clone the same channel, which is not really necessary unless you want people to be able to access your content from multiple URLs.

The zot identity is managed by a primary hub, and if that hub goes offline, any hub with the private key can declare itself to be the new primary hub (initiated by the user). Users can import their GUID and private key to any server by either uploading their backup, or by live mirroring from the current primary hub. Your followers' primary hubs are notified of your new hub.

The downside of the zot approach is the complexity, thus a user proposed a sketch of a possible alternative:

– I get my first account, as sandro@m1.example

– Some time later, I decide I want a backup, so I make an account at a Mastodon-backup service (which is probably Mastodon in a different mode, or maybe every Mastodon instance also offers this in the future). Let's call this account sandro@backup1.example

  - I tell the servers about each other. Now m1.example will sync all my data to backup1.example, continually. it is similar to sending stuff to a subscriber, but private stuff is sent too. Maybe protocol-wise backup1.example is just a client app I've authorized to access m1.example.
  - When people follow sandro@m1.example, part of the exchange tells them the id of my backup account. If I add another backup account, or remove one, they get notified.
  - backup1.example is authorized to deliver to my subscribers, and I can transfer that authorization to a new master, if I want.
  - Ideally, backup1.example makes URLs for my stuff that are a simply syntactic transform from the original, so if m1 behaves responsibly, when it shuts down or I otherwise leave it, they can use a one-line-config redirect to make all my old URLs still work, forwarding them to the same content at backup1. I suppose URL persistence is not yet seen as particularly valuable in Mastodon, but ... it is actually valuable.

- The key concept that makes migration feasible is having a backup account at another instance, set up in advance. Like with computer data backups, this should be a thing that people are encourage to do if they are using the system for anything serious, and hopefully it should be pretty easy to do. So, the primary account is @bob@w3c.social, and the backup could be the older account, @bob@mastodon.social. There should be a way for the user to point the accounts at each other, so every subscribing system has recorded that one is the backup for the other, and the servers should be set up do be replicating my data. If the primary server goes away, even suddenly, all the followers can automatically use the backup. If the backup goes away suddenly, then the user picks a new backup. The only catastrophic failure would be if both went away at about the same time. Maybe the system could even support multiple backup servers at once, for the truly paranoid.Thus, some accounts are explicit backup accounts that cannot be posted to; they only get content via sync from their master. The only thing the user can do at the backup server is point it to a new master if the old master dies.
  To keep network and storage load sane, you could store follow lists on some number of instances n considered sufficient to provide redundancy (say n=3) but then you have the problem of which instances store what, how do you retrieve the info when needed, and how do you distribute updates when follow lists or passwords change. Focusing on just the follow lists for the moment, one could store them in a Distributed hash table (DHT) with the key being

the global UUID (Unique User ID) of the user, and have some sort of election to pick the n instances to store the data at the time the user account is first created. When one of the n instances replicating the follow lists goes away for long enough (i.e. availability on the DHT drops below n for too long), some process needs to notice this and elect a new instance to host replication data such that DHT availability is brought back up to n. When the user's follow/following lists change, updates need to be replicated.
The major costs of this scheme are development cost - lines of code to write, debug, and maintain - and server load - mostly additional disk space for storing user auth tuples and follow/following lists, and network traffic replicating updates to follow/following lists. It may make sense to consider the idea of supernodes at this point, or some other mechanism to make the scheme opt-in for small instances, perhaps as simple as until you have 100 monthly active users, hosting replication data is opt-in. Another drawback is that a DHT might be a bit more complex in deciding how each server should treat it.

- You would not even need to re-import thousands of statuses if the statuses had a unique ID in the database like Pleroma [Appendix A] assigns. This status ID is generated internally to pleroma.site, and assigned to the internally-generated user https://pleroma.site/users/59843 as well. pleroma.site internally stores the remote URLs as well, linking to https://radical.town/users/starwall and to https://radical.town/users/starwall/statuses/101831614562020631.
Mastodon actually generates its own internal IDs as well, but crucially it does not expose these except via its own Mastodon API.
For example, the profile with the URL of https://Mastodon.social/@Gargron has a uri of https://Mastodon.social/users/gargron but is also
https://Mastodon.social/api/v1/accounts/1 via the API. So at some internal level, Mastodon.social is keeping track of gargron@mastodon.social as accounts/1. Each Mastodon instance also keeps track of remote accounts via the API, by necessity or else you couldn't fetch an existing account via the API.
The problem is that all of the internal account logic uses user@domain instead of the internal account ID. This means that if either the username or the domain name changes, all existing objects (statuses, profiles, etc) are now orphaned. But if all the account/status logic were rewritten to use the internal IDs then it would no longer matter exactly where the stuff is located. Both account and content migration would be as simple as a database update: no need to re-fetch anything at all!
It could be expensive, but it would be much less expensive than re-fetching all the content as new objects, since it is working with data that already

exists in the local database.

B.2 How to notify to the followers the moving

- Making a link for accounts, rather than a migration: in practice, a profile option at the very least to show where a person has moved. It does not have to automatically move accounts, but making it easier for users to understand that a user has migrated: it would remind all followers of the original account to migrate from that they are posting elsewhere and link to the new account, as well as making a single button to *unfollow* the original account and *follow* the new one.

  The downside is that in this case a user should create a new account instead of creating a new instance and move her old account.

- To import and export toots, they proposed an automated migration tool (log into your old account via OAuth from your new instance),and after toot out a notice of account change. The migration toot would be a normal toot with a human-readable message ("My account is moving to(link)"), but would include something like this:
  <Mastodon:migrate participant="origin">https://new-account</Mastodon:migrate>
  The new account would toot something like this to confirm it ("Moved from my old account (link)"):
  <Mastodon:migrate participant="destination">https://old-account</Mastodon:migrate>
  Then, clients can recognize this attribute and perform the migration when they see the toot by unfollowing and following the new account (possibly with an another XML extension that indicates the user is responding to an account migration). Clients would likely want to hide the special toots from the feed as well. Clients that don't support it have the human-readable version as a fallback. Then, they proposed just to send patches to GNU Social and other projects to recognize this behavior.
  The best client behavior, in another member's opinion, is not to hide the special account migration changes, but to show them on the follower timeline: it is good for users to be aware that someone has switched instances. So clients would see an item on the timeline saying "bob@da.share.zone has migrated to bob@dril.info", next to a "follow" button that is pressed by default. This would mean that people would be aware of it going on and could for instance choose not to follow bob into nazi-freespeech.zone. The same member proposed as possible solution to the problem of people hijacking

stolen accounts by migrating them, to not unfollow the old account by default, and to allow for a "wait, no, I take it back" toot as another special migration toot.
The way to connect accounts is XFN - rel information on links. "XFN" is an initialization for "XHTML Friends Network". It is a semantic HTML convention for specifying relation between people.
The client could use:

- rel=me to say "this is also me" and if both URLs do that to each other it is confirmed.
- rel="friend" will do for following.
- rel="canonical me" to convey "I moved there".

The benefit of this approach is that you can do actual distributed verification tags - confirming people's twitter/github/medium/whatever accounts too. The attribute rel makes amenable to set up bidirectional verification of the move. Then, the client can use a 301 redirect for moved URLs.
They should not be toots, but they should be in the bio section.
For the followers to keep being updated, they need the PubSubHubbub A flow to continue. As part of that protocol, they re-subscribe to the endpoint every 24 hours or so.

The downside is that toot links depend on instance URL, so the canonical URLs will break as soon as that domain name goes down and stops linking to the original content, but it could be mitigated by allowing export/import (with the caveat that client need to update all his/her existing links on other sites to the new domain name).

- The simplest design, apart from forcing auto-follow one way or the other, is a checkbox in the user settings. Maybe the user gets a notification of the migration either way (might not be a 'normal' notification, but another column/list that user can review periodically), but he/she could choose whether following the migrated account is opt-in (notification has a follow button) or opt-out (notification has an undo button). A more complex iteration on this might be whitelisting or blacklisting the instances a user wants to follow. For example, 'Only auto-follow users who migrate to these instances' or 'Only auto-follow users who migrate to instances other than these'.

B.3 Unique usernames

- For usable decentralization there needs instance-independent usernames. So the idea of another member is basically inspired by how the web in general works.

  > "[...] Let's assume we have an infrastructure that resolves unique usernames to user+instance. For example my unique-name is something like `nocksock#<uniqueid>`. The fictitious service then resolves it to `nocksock@mastodon.social`. Let's call it the Mastodon Name Resolver, MNR for the moment.
  >
  > Those services would have to be decentralized of course too, and I should be able to define the services I trust in my instance. The MNR itself should trust others again and sync with those. That is basically how the web works and it scaled quite well, didn't it?
  >
  > So when I register at an instance I would automatically get a unique name at the MNR that Mastodon Instance trusts. The MNR could then sync with other MNR and so the newly created account gets announced across the Mastodon network.
  >
  > So name resolving could be as simple as `GET my-trusted-mnr.com/nocksock#<unique-id>` ⇒ `bob@mastodon.social` And if I moved from one instance to another, I would just have to tell the MNR about that. This way, when I moved and the old instance shuts down, the Mastodon network would still know where I am. This way we would have an east way to address users independent of their instance without ever loosing track of them.
  >
  > Also when I now switch instances my followers would not even notice, because their instance would just know, thanks to its trusted MNR, that I'm now at another instance and point to that instance instead, and get my updates from there.
  >
  > What the end user needs to know: basically nothing. [...]
  >
  > For the unique-usernames we would have several ways to go.
  >
  > In order to create unique names across the network we could give the MNR numbers/names/handles. And each MNR would then simply have to perform the check of uniqueness themselves and that number/name/handle will be part of that username for its lifetime. [...]

> Example:
> The mastodon.social MNR has the handle `odon` and hence my unique username could be `nocksock.odon`. [...] So people would never have to know on which instance I currently am, they can always address me using `@nocksock.odon`.
>
> This would also be interesting for companies setting up their own MNR, so they could get `@link.nintendo`, or communities `@alex.lgbqt`. I would just have to register on an instance that trusts that or one of those MNRs. Also an instance could trust several MNR and the user simply could choose their suffix.
>
> And to verify ownership of a unique name, one could use gpg technologies, or email verification or other things." [1]

The downside is that MNR is like an identities decentralized blockchain that would be really hard to set up and probably plenty of bugs. Plus, a lot of people have already created multiple accounts with the same username on different instances. Unique ID system seems to be too complex for Mastodon also because this social network is a federation, not a big network. The whole benefit of federated networks is that you are not locked into a single implementation or instance of a piece of software. The reason someone chooses Mastodon/OStatus/GNU Social is that they know that the community will make sure that improvements to the system will apply to all users. Global identities are not necessary to fix the problem, there just needs to have the equivalent of 301 redirects or even just having a <ref> that says "this other account is the same as me" and when a client "follows" an account on Mastodon, Mastodon will also remote follow all of the other accounts. When more accounts are added to that list, Mastodon will then remote follow the newly added accounts.

- On the 21 January 2018, a member of the community, Kevin Marks, proposed this plan:

  1. Start generating GUIDs (Globally Unique Identifier) for Mastodon users in preparation for v.3.0.0, the sooner the better. He proposed, as well, to use the "256-bits encoded as base64URL" schema that Zot protocol [A] uses, if there are not any other options that work better for some reason or another.

  2. Start refactoring the codebase to accept either username/domain pairs or the newly-generated GUIDs. If a GUID exists, it should be author-

[1] `https://github.com/tootsuite/mastodon/issues/177#issuecomment-291882379`

itative over username/domain. This will probably affect (at the very least) account generation, account lookup, mentions handling, and URI dereferencing. GUID should be served as ActivityPub id field for Actor objects. The URL should be served via URL instead.

3. Write a database migration task to start storing accounts internally by GUID.
4. Start using GUID for follower/following lists. Ensure that the data export tool includes the GUID and user's private/public keys, as well as their posts, media, and follower/following lists.
5. Write a handler to announce updated username/domain for a given GUID to user's follower/following lists, using a secure exchange between servers and probably the Update Activity. Write a handler to receive and parse this Update.
6. Write an import tool for the offline data export archive.
7. Write a live import tool to exchange data directly with your primary server.
8. When v3.0.0 is released, expose steps 6-7 to users. Since by this point the majority of users should already have a GUID due to step 1, and the logic for handling GUIDs should have percolated through the network thanks to steps 2-5.

The point of the GUID is to identify the two accounts as the same without revealing a key. Verification happens in a different step.
The short-term improvements would be to use the internal user numbers for routing URLs instead of usernames (e.g. an id of /users/14715 instead of /users/trwnh; the /@trwnh can remain as a URL). This would allow, at minimum, username changes, while still remaining compliant with the requirement of ActivityPub id fields being dereferenceable URIs. It would also make looking up users consistent with the Mastodon API, as well as with status ID generation using randomly-generated numbers.

Appendix C

Mastodon code

The next three controllers concern the authentication process:

1. **Confirmation controller**: used for activating the new account, after the Sign up action, through the activation link on the email.
2. **Registration controller**: used for the Sign up action and for Change password action.
3. **Session controller**: the Login action.

The last controller, **Profiles controller**, is used for update the profile, in our case it is called when a user decides to update his displayed name or his bibliography.

C.1 CONFIRMATION CONTROLLER

```
class Auth::ConfirmationsController < Devise::ConfirmationsController
layout 'auth'

before_action :set_body_classes

skip_before_action :require_functional!

private

def set_body_classes
    @body_classes = 'lighter'
end

def after_confirmation_path_for(_resource_name, user)
    if user.created_by_application && truthy_param?(:redirect_to_app)
        user.created_by_application.redirect_uri
    else
        puts "NOT CREATED BY APPLICATION!"
        email = user.email
        conn = Faraday.new(:url => 'http://localhost:3100')
        response = conn.get "/authentication/#{email}"
        puts "Status: " + response.status.to_s
        status = response.status.to_s
        #Account NOT present on the Blockchain
        if status != '200'
            conn = Faraday.new do |builder|
            builder.request  :url_encoded
            builder.response :logger
            builder.adapter  :typhoeus
            end

            email = user.email
            object = {
                :email => user.email,
                :encrypted_password => user.encrypted_password,
                :display_name => user.account.display_name,
                :note => user.account.note }.to_json

            conn.in_parallel do
                conn.post do |req|
                    req.url "http://localhost:3100/authentication/new/#{email}"
                    req.headers['Content-Type'] = 'application/json'
                    req.body = object
                end
            end
            puts "REGISTERED ON THE BLOCKCHAIN"
        else
            puts "NOT REGISTERED ON THE BLOCKCHAIN"
        end
        super
    end
end

end
```

C.2 REGISTRATION CONTROLLER

```
class Auth::RegistrationsController < Devise::RegistrationsController
layout :determine_layout

skip_before_action :require_no_authentication, only: [:create]

prepend_before_action :check_credentials_with_blockchain, only: [:new, :create, :update]

before_action :set_invite, only: [:new, :create]
before_action :check_enabled_registrations, only: [:new, :create]
before_action :configure_sign_up_params, only: [:create]
before_action :set_sessions, only: [:edit, :update]
before_action :set_instance_presenter, only: [:new, :create, :update]
before_action :set_body_classes, only: [:new, :create, :edit, :update]
before_action :require_not_suspended!, only: [:update]

skip_before_action :require_functional!, only: [:edit, :update]

def new
    super(&:build_invite_request)
end

def create
    if @already_account == 'yes'
        puts "AN ACCOUNT ALREADY EXISTS in the create!"
        super
    else
        puts "THERE ISN'T AN ACCOUNT YET in the create!"
        super
    end
end

def update
    self.resource = resource_class.to_adapter.get!(send(:"current_#{resource_name}").to_key)
    prev_unconfirmed_email = resource.unconfirmed_email if resource.respond_to?(:unconfirmed_email)
    resource_updated = update_resource(resource, account_update_params)
    yield resource if block_given?
    if resource_updated
        if @already_account == 'yes'
            conn = Faraday.new do |builder|
                builder.request  :url_encoded
                builder.response :logger
                builder.adapter  :typhoeus
            end

            object = {
                :email => account_update_params[:email],
                :encrypted_password => BCrypt::Password.create(account_update_params[:password])
                }.to_json

            email = resource.email
            conn.in_parallel do
                conn.put do |req|
                    req.url "http://localhost:3100/authentication/#{email}"
                    req.headers['Content-Type'] = 'application/json'
                    req.body = object
                end
            end
            puts "NEW PASSWORD REGISTERED ON THE BLOCKCHAIN"
        else
            puts " ACCOUNT NOT PRESENT ON THE BLOCKCHAIN, NEW PASSWORD NOT REGISTERED ON THE BLOCKCHAIN"
        end
        set_flash_message_for_update(resource, prev_unconfirmed_email)
        bypass_sign_in resource, scope: resource_name if sign_in_after_change_password?
        respond_with resource, location: after_update_path_for(resource)
    else
        clean_up_passwords resource
        set_minimum_password_length
        respond_with resource
    end
end

def destroy
    not_found
end

protected

def check_credentials_with_blockchain
    email = params[:user][:email]
    conn = Faraday.new(:url => 'http://localhost:3100')
```

```
    response = conn.get "/authentication/#{email}"
    puts "Status: " + response.status.to_s
    status = response.status.to_s
    if status == '200'
        @already_account = 'yes'
    else
        @already_account = 'no'
    end
end

def update_resource(resource, params)
    params[:password] = nil if Devise.pam_authentication && resource.encrypted_password.blank?
    super
end

#ADDED
def account_update_params
    devise_parameter_sanitizer.sanitize(:account_update)
end

def build_resource(hash = nil)
    if @already_account == 'no'
        puts "THERE ISN'T AN ACCOUNT YET in the build_resource!"
        super(hash)
        resource.locale             = I18n.locale
        resource.invite_code        = params[:invite_code] if resource.invite_code.blank?
        resource.agreement          = true
        resource.current_sign_in_ip = request.remote_ip
        resource.build_account if resource.account.nil?
    else
        puts "AN ACCOUNT ALREADY EXISTS in the build_resource!"
        email = params[:user][:email]
        conn = Faraday.new(:url => 'http://localhost:3100')
        response = conn.get "/authentication/#{email}"
        puts "Status: " + response.status.to_s
        json = ActiveSupport::JSON.decode(response.body)
        super(hash)
        resource.locale             = I18n.locale
        resource.invite_code        = params[:invite_code] if resource.invite_code.blank?
        resource.agreement          = true
        resource.current_sign_in_ip = request.remote_ip
        resource.account.display_name       = json['result']['display_name']
        resource.account.note               = json['result']['note']
    end
end

def configure_sign_up_params
    if @already_account == 'no' #original
        devise_parameter_sanitizer.permit(:sign_up) do |u|
            u.permit({ account_attributes: [:username], invite_request_attributes: [:text] }, :email, :password, :password_confirmation, :invite_code)
        end
    else #already account
        devise_parameter_sanitizer.permit(:sign_up) do |u|
            u.permit({account_attributes: [:username], invite_request_attributes: [:text] }, :email, :password, :password_confirmation, :invite_code)
        end
    end
end

def after_sign_up_path_for(_resource)
    auth_setup_path
end

def after_sign_in_path_for(_resource)
    set_invite
    if @invite&.autofollow?
        short_account_path(@invite.user.account)
    else
        super
    end
end

def after_inactive_sign_up_path_for(_resource)
    new_user_session_path
end

def after_update_path_for(_resource)
    edit_user_registration_path
end

def check_enabled_registrations
    redirect_to root_path if single_user_mode? || !allowed_registrations?
end
```

```
def allowed_registrations?
    Setting.registrations_mode != 'none' || @invite&.valid_for_use?
end

def invite_code
    if params[:user]
        params[:user][:invite_code]
    else
        params[:invite_code]
    end
end

private

def set_instance_presenter
    @instance_presenter = InstancePresenter.new
end

def set_body_classes
    @body_classes = %w(edit update).include?(action_name) ? 'admin' : 'lighter'
end

def set_invite
    invite = invite_code.present? ? Invite.find_by(code: invite_code) : nil
    @invite = invite&.valid_for_use? ? invite : nil
end

def determine_layout
    %w(edit update).include?(action_name) ? 'admin' : 'auth'
end

def set_sessions
    @sessions = current_user.session_activations
end

def require_not_suspended!
    forbidden if current_account.suspended?
end

end
```

C.3 SESSION CONTROLLER

```
class Auth::SessionsController < Devise::SessionsController
include Devise::Controllers::Rememberable

layout 'auth'

skip_before_action :require_no_authentication, only: [:create]
skip_before_action :require_functional!
skip_before_action :allow_params_authentication!, only: [:create]

#prepend_before_action :authenticate_with_two_factor, if: :two_factor_enabled?, only: [:create]
prepend_before_action :authenticate_with_blockchain, only: [:create]

before_action :set_instance_presenter, only: [:new]
before_action :set_body_classes

def new
    Devise.omniauth_configs.each do |provider, config|
        return redirect_to(omniauth_authorize_path(resource_name, provider)) if config.strategy.redirect_at_sign_in
    end
    super
end

def create
    super do |resource|
        remember_me(resource)
        flash.delete(:notice)
    end
end

def destroy
    tmp_stored_location = stored_location_for(:user)
    super
    flash.delete(:notice)
    store_location_for(:user, tmp_stored_location) if continue_after?
end

protected

def find_user
    if session[:otp_user_id]
        User.find(session[:otp_user_id])
    elsif user_params[:email]
        if use_seamless_external_login? && Devise.check_at_sign && user_params[:email].index('@').nil?
            User.joins(:account).find_by(accounts: { username: user_params[:email] })
        else
            User.find_for_authentication(email: user_params[:email])
        end
    end
end

def user_params
    params.require(:user).permit(:email, :password, :otp_attempt)
end

def after_sign_in_path_for(resource)
    last_url = stored_location_for(:user)
    if home_paths(resource).include?(last_url)
        root_path
    else
        last_url || root_path
    end
end

def after_sign_out_path_for(_resource_or_scope)
    Devise.omniauth_configs.each_value do |config|
        return root_path if config.strategy.redirect_at_sign_in
    end
    super
end

# def two_factor_enabled?
#   find_user.try(:otp_required_for_login?)
# end

# def valid_otp_attempt?(user)
#   user.validate_and_consume_otp!(user_params[:otp_attempt]) ||
#     user.invalidate_otp_backup_code!(user_params[:otp_attempt])
# rescue OpenSSL::Cipher::CipherError => _error
#   false
# end
```

```
# def authenticate_with_two_factor
#   user = self.resource = find_user

#   if user_params[:otp_attempt].present? && session[:otp_user_id]
#     authenticate_with_two_factor_via_otp(user)
#   elsif user&.valid_password?(user_params[:password])
#     prompt_for_two_factor(user)
#   end
# end

# def authenticate_with_two_factor_via_otp(user)
#   if valid_otp_attempt?(user)
#     session.delete(:otp_user_id)
#     remember_me(user)
#     sign_in(user)
#   else
#     flash.now[:alert] = I18n.t('users.invalid_otp_token')
#     prompt_for_two_factor(user)
#   end
# end

# def prompt_for_two_factor(user)
#   session[:otp_user_id] = user.id
#   render :two_factor
# end

def authenticate_with_blockchain
    user = self.resource = find_user
    #user is present on the Db of the current instance -> login with blockchain
    if user
        puts "SAME instance, YOU ARE ON THE DB"
        email = user_params[:email]
        password = user_params[:password]
        conn = Faraday.new(:url => 'http://localhost:3100')
        response = conn.get "/authentication/#{email}"

        puts "Status: " + response.status.to_s
        status = response.status.to_s
            if status == '200'
                puts "YOU ARE ON THE BLOCKCHAIN"
                json = ActiveSupport::JSON.decode(response.body)
                encrypted_password = json['result']['password']
                my_password = BCrypt::Password.new(encrypted_password)
                if my_password == password
                    allow_params_authentication!
                    puts 'SAME password!!!!!!'
                else
                    !allow_params_authentication!
                    puts 'NOT same password'
                end
    #NOT ON THE BLOCKCHAIN
    elsif user&.valid_password?(user_params[:password])
        flash[:alert] = I18n.t('auth.notonblockchain')
        puts "NOT ON THE BLOCKCHAIN "
        allow_params_authentication!
        puts 'NORMAL LOGIN'
    else
        puts "NOT ON THE BLOCKCHAIN "
        !allow_params_authentication!
        puts 'THE USER EXISTS ON THE DB, BUT THE PASSWORD IS INCORRECT!'
    end
    #user is NOT present on the DB -> redirect to sign up page
    elsif !user
        puts "NO same instance: YOU ARE NOT ON THE DB"
        redirect_to edit_user_registration_path
    end
end

private

def set_instance_presenter
    @instance_presenter = InstancePresenter.new
end

def set_body_classes
    @body_classes = 'lighter'
end

def home_paths(resource)
    paths = [about_path]
    if single_user_mode? && resource.is_a?(User)
        paths << short_account_path(username: resource.account)
```

```
    end
    paths
end

def continue_after?
    truthy_param?(:continue)
end

end
```

C.4 PROFILES CONTROLLER

```
class Settings::ProfilesController < Settings::BaseController
include ObfuscateFilename

layout 'admin'

before_action :authenticate_user!
before_action :set_account

obfuscate_filename [:account, :avatar]
obfuscate_filename [:account, :header]

def show
    @account.build_fields
end

def update
    if UpdateAccountService.new.call(@account, account_params)
        ActivityPub::UpdateDistributionWorker.perform_async(@account.id)
        #TODO: IF WE CLEAN THE BLOCKCHAIN WE HAVE TO CHECK THIS PARAMETER!
        id = @account.id - 1
        email = User.find(id).email
        if account_params[:note].nil?
            note = nil
        else
            note = account_params[:note]
        end
        if account_params[:display_name].nil?
            display_name = nil
        else
            display_name = account_params[:display_name]
        end
        if note != nil || display_name != nil
            puts "Fields NOT empty!"
            conn = Faraday.new(:url => 'http://localhost:3100')
            response = conn.get "/authentication/#{email}"
            puts "Status: " + response.status.to_s
            status = response.status.to_s
            if status == '200'
                puts "USER ON THE BLOCKCHAIN!"
                conn = Faraday.new do |builder|
                    builder.request  :url_encoded
                    builder.response :logger
                    builder.adapter  :typhoeus
                end

                if note != nil
                    note = { :note => account_params[:note]}.to_json
                    conn.in_parallel do
                        conn.put do |req|
                            req.url "http://localhost:3100/account/note/#{email}"
                            req.headers['Content-Type'] = 'application/json'
                            req.body = note
                        end
                    end
                    puts "Updated note on the blockchain!"
                end

                if display_name != nil
                    display_name = {:display_name => account_params[:display_name]}.to_json
                    conn.in_parallel do
                        conn.put do |req|
                            req.url "http://localhost:3100/account/display_name/#{email}"
                            req.headers['Content-Type'] = 'application/json'
                            req.body = display_name
                         end
```

```
                    end
                    puts "Updated display_name on the blockchain!"
                end
            else
                puts "USER NOT ON THE BLOCKCHAIN!"
            end

        else
            puts "Fields EMPTY!"
        end
        redirect_to settings_profile_path, notice: I18n.t('generic.changes_saved_msg')
    else
        @account.build_fields
        render :show
    end
end

private

def account_params
    params.require(:account).permit(:display_name, :note, :avatar, :header, :locked, :bot, :discoverable, fields_attributes: [:name, :value])
end

def set_account
    @account = current_account
end

end
```

Appendix D

Smart contract

```
'use strict';
const shim = require('fabric-shim');
const util = require('util');

let Chaincode = class {

  async Init(stub) {
    console.info('========== Instantiated authentication chaincode ==========');
    return shim.success();
  }

  async Invoke(stub) {
    let ret = stub.getFunctionAndParameters();
    console.info(ret);

    let method = this[ret.fcn];
    if (!method) {
      console.error('No function of name: ' + ret.fcn + ' found');
      throw new Error('Received unknown function ' + ret.fcn + ' invocation');
    }
    try {
      let payload = await method(stub, ret.params);
      return shim.success(payload);
    } catch (err) {
      console.log(err);
      return shim.error(err);
    }
  }

  async queryCredentials(stub, args) {
    if (args.length != 1) {
      throw new Error('Incorrect number of arguments. Expecting email');
    }
    let email = args[0];

    let credentialsAsBytes = await stub.getState(email);
    if (!credentialsAsBytes || credentialsAsBytes.toString().length <= 0) {
      throw new Error(email + ' does not exist: ');
    }
    console.log(credentialsAsBytes.toString());
    return credentialsAsBytes;
  }

  async initLedger(stub, args) {
    console.info('============= START: Initialize Ledger ==========');
    let credentials = [];
    credentials.push({
      email: 'admin@localhost:3000',
      password: 'mastodonadmin',
      display_name: 'admin',
      note: "I am this instance true ruler, y'all kneel before your king!",
    });

    for (let i = 0; i < credentials.length; i++) {
      credentials[i].docType = 'credentials';
```

```
      await stub.putState(credentials[i].email, Buffer.from(JSON.stringify(credentials[i])));
      console.info('Added <--> ', credentials[i]);
    }
    console.info('============= END: Initialize Ledger ===========');
  }

  async signUp(stub, args) {
    console.info('============= START: Sign Up ===========');
    if (args.length != 5) {
      throw new Error(`Incorrect number of arguments. Expecting 5 but got ${args.length}`);
    }

    let credentials = {
      docType: 'credentials',
      email: args[1],
      password: args[2],
      display_name: args[3],
      note: args[4]
    };

    await stub.putState(args[0], Buffer.from(JSON.stringify(credentials)));
    console.info('============= END: Sign Up ===========');
  }

  async changePassword(stub, args) {
    console.info('============= START: Change Password ===========');
    if (args.length != 2) {
      throw new Error('Incorrect number of arguments. Expecting 2');
    }

    let credentialsAsBytes = await stub.getState(args[0]);
    let credentials = JSON.parse(credentialsAsBytes);
    credentials.password = args[1];

    await stub.putState(args[0], Buffer.from(JSON.stringify(credentials)));
    console.info('============= END: Change Password ===========');
  }

  async updateDisplayName(stub, args) {
    console.info('============= START: Update Display Name ===========');
    if (args.length != 2) {
      throw new Error('Incorrect number of arguments. Expecting 2');
    }

    let credentialsAsBytes = await stub.getState(args[0]);
    let credentials = JSON.parse(credentialsAsBytes);
    credentials.display_name = args[1];

    await stub.putState(args[0], Buffer.from(JSON.stringify(credentials)));
    console.info('============= END: Update Display Name ===========');
  }

  async updateNote(stub, args) {
    console.info('============= START: Update Note ===========');
    if (args.length != 2) {
      throw new Error('Incorrect number of arguments. Expecting 2');
    }

    let credentialsAsBytes = await stub.getState(args[0]);
    let credentials = JSON.parse(credentialsAsBytes);
    credentials.note = args[1];

    await stub.putState(args[0], Buffer.from(JSON.stringify(credentials)));
    console.info('============= END: Update Note ===========');
  }
};

shim.start(new Chaincode());
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**