CATHOLIC UNIVERSITY OF LOUVAIN-LA-NEUVE

Master thesis - 2018/2019

A decentralized social network based on event-sourcing principles

Lovisetto Gary - 36011600 - gary.lovisetto@student.uclouvain.be

March 15, 2019

1 Introduction on social networks

1.1 Open social network

A social network is a website that allows you to connect with different people all over the internet and share data such as photos, videos, music and other information. They can be used for entertainment purposes like Facebook, Twitter, etc... but also for enterprise social networking like LinkedIn, SocialCast and Yammer. The quick rise of all those social-networks led to a lot of problems that we are currently facing, the main one being the malicious usage of user data. As we've recently seen with the introduction of the GDPR, the way all the users' information were handled led to a lot practices flirting very closely with the edge of the law and sometimes even breaking it. This led to a new era of social networking where data protection and transparency must be explicit as to enforce the many new digital rights an individual has, such as the right to be informed and to erasure.

Since then, a lot of new data exchange platforms have been created but with the twist that they are open social networks. Open-sourcing is nothing new in the domain of computer science, it is a decentralized software development model that encourages open collaboration. A main principle of open-source software development is peer production, with products such as source code, blueprints, and documentation freely available to the public. By being completely open to the public, these new open-source social-networks (OSN) solve, for the most part, the data transparency issue. In addition to all the perks provided with open-sourcing such as customization, collaboration, etc..., all the users can directly see how their information is handled and propose their own modifications to the code base of the network.

The major issue with social networks like Twitter and Facebook is that they are monopolistic, the power of these networks allows them to dictate their own rules. With the lack of concurrency, we see a trend where the goal of the company shift much more to how to raise the profitability instead of improving the quality. This is the case here because the users have already invested themselves in the website for a long time and there are no real alternatives, so instead of trying to find the best products between all the possibilities, every individual flocks to the website with the monopoly.

To fight this tendency and have a much more healthy market, OSN have started working together to become decentralized networks. Instead of collecting and storing social users data (friends, posts, comments, likes, etc...) on a single website, these social networks operate as a federation of independent networks called the Fediverse. It is the ensemble of federated servers that are used for web publishing (social networking) and file hosting. On different servers, users can create so called identities. These identities are able to communicate over the boundaries of the servers because the softwares running on them support one or more communication protocols which follow the open standard. As an identity on the Fediverse, you are able to post text and other media, or to follow posts by other identities. In some cases, you 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 your data (calendar, address book, etc...). The softwares spanning the Fediverse cover a wide range of features like video hosting (PeerTube), microblogging (Mastodon) or even sound hosting (Funkwhale) [1].

This practice is completely adapted to the idea of a market where every individual is free to chose the solution he prefers without losing the time invested previously. With the Fediverse, we now have completely independent websites that are completely different from each other and yet are able to communicate between each others. This is a complete shift in the evolution of the social networks which was dictated for a long time by the aphorism *the rich get richer*, where individuals, almost exclusively, flocked to the biggest websites because that's where their relatives were. While that is part of the human nature and will still be relevant in the future, the Fediverse goal is to lead the market to be dominated by the quality of the product instead of the user base size. A good product arriving on the market will have a much higher chance of getting new users leading all the already established websites to seek to continually improve as to not get outperformed.

It is important to note that everything is not perfect, one of the main concern is that there is no real solutions in place to easily move the profile of an user between different networks. Instead, an individual as to manually recreate a new account each time he changes websites. This can easily be bothersome if there is a process in place such as Facebook's friend requests to connect with other people. There is also the issue of malicious websites being part the Fediverse and able to steal all the data that transit in the federation. There are currently different solutions being worked on such as having the profile being transferable through the block-chain to guarantee the protection

of data. This also creates new problems relative to the GDPR where some rights might hard to preserve (i.e. right to erasure) because of the decentralization of information. The following list is not exhaustive but summarize the disadvantages of decentralized networks :

- Latency issues
- Duplication of storage
- Deletion of content
- No central authority to handle malwares, spams, ...
- Distributed authentication and authorization

Platform Name	Type	ActivityPub	DFRN	Diaspora Network	OStatus	Zot Zot/6
Aardwolf	Social network	Yes	No	No	No	No
Anfora	Image hosting	In progress [8]	No	No	No	No
CloutStream	Professional Social network	Yes	No	No	No	No
diaspora* software	Social network, Microblogging	Proposed [9] [10]	No	Yes	No	No
distbin	Pastebin	Yes	No	No	No	No
Friendica (f. Friendika; orig. Mistpark)	Social network, Microblogging	Yes	Yes	Yes	Yes	No
Funkwhale	Audio, sound hosting	Yes	No	No	No	No
GNU MediaGoblin	file, image, audio, video hosting	Proposed [11]	No	No	No	No
GNU social (f. StatusNet; orig. Laconica)	Microblogging	Proposed [12] [13]	No	No	Yes	No
Hubzilla (f. RedMatrix; orig. Friendica-Red)	CMS, Social network, Microblogging, Wiki, Blogging, Image gallery, File hosting	Yes	No	Yes	Yes [3]	Zot
Littr.me	Link-sharing	Yes	No	No	No	No
Mastodon	Microblogging	Yes [14]	No	No	Yes	No
microblog.pub	Microblogging	Yes	No	No	No	No
Misskey	Social network, Microblogging	Yes	No	No	No	No
Nextcloud	File hosting	Yes [15]	No	No	No	No
Numaverse	Microblogging, ETH blockchain	Yes	No	No	No	No
Osada	Social network, Microblogging	Yes	No	No	No	Zot/6
PeerPx	Image hosting	Yes	No	No	No	No
PeerTube	Video hosting	Yes	No	No	No	No
PixelFed	Image hosting	Yes [16]	No	No	No	No
Pleroma	Microblogging	Yes	No	No	Yes [4]	No
Plume	Blogging	Yes	No	No	No	No
postActiv	Microblogging	Proposed [17]	No	No	Yes	No
Prismo	Link-sharing	Yes	No	No	No	No
Pubcast (f. Metapods)	Netcasting	Yes	No	No	No	No
pump.io	Microblogging	Proposed [18]	No	No	Dropped [19]	No
Read.as	Feed reader	Yes [20]	No	No	No	No
Socialhome	Website, Social network, Microblogging	In progress [21]	No	Yes	No	Proposed [22]
Write.as / Write Freely	Blogging	Yes	No	No	No	No
Zap	Social network, Microblogging	No	No	No	No	Zot/6

Figure 1: Fediverse software platforms

1.2 Fediverse protocols

The capabilities of the Fediverse protocols all have some features in common that can be categorized as follows :

- A communications protocol and endpoint for introductions (e.g. the “friend request” of traditional social networks)
- A communications protocol and endpoint for responding to these requests and establishing permission to communicate
- A communications protocol and endpoint for notification of new or modified content with mutually authenticated source and destination

- A communications endpoint and protocol for polling available content with mutually authenticated source and destination
- A means to invisibly cross-authenticate within cooperating cells, so that communication is not impeded by requiring "in your face" authentication/authorization exchanges when crossing site boundaries

1.2.1 DFRN

The DFRN framework provides the communication basis for a decentralized social network where cooperating servers share information on your behalf while operating in a web of trust relationships you control. It can provide a "Facebook-like" experience without requiring a central company or server. The goal of DFRN is to provide an open and distributed social communication platform with server requirements comparable to that of a typical hosted blog. Instead of a central server, a collection of distributed 'cells' or 'nodes' are able to communicate with each other on your behalf. This goal of this protocol is to correct the common issues found in other distributed social technologies such as the weakness in the privacy model, the lack of specification of content deletion and re-distribution policies and poor ease of use.

1.2.2 Diaspora network

The Diaspora social network is constructed of a network of nodes hosted by many different individuals and institutions. Each node operates a copy of the Diaspora software acting as a personal web server. Users of the network can host a pod on their own server or create an account on any existing pod of their choice, and from that pod can interact with other users on all other pods. Diaspora users retain ownership of their data and do not assign ownership rights. The software is specifically designed to allow users to download all their images and text that have been uploaded at any time. It allows user posts to be designated as either "public" or "limited". In the latter case, posts may only be read by people assigned to one of the groups, termed aspects, which the user has approved to view the post. Each new account is assigned several default aspects (friends, family, work and acquaintances) and the user can add as many custom aspects as they like. It is possible to follow another user's public posts without the mutual friending required by other social networks.

1.2.3 OStatus

OStatus is an open standard for federated microblogging (Twitter style), 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, WebSub, 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.

1.2.4 Zot

Zot is a JSON-based web framework for implementing secure decentralised communications and services. In order to provide this functionality, Zot creates a decentralised globally unique identifier for each hub on the network. This global identifier is not linked inextricably to DNS, providing the requisite mobility. Many existing decentralised communications frameworks provide the communication aspect, but do not provide remote access control and authentication. Additionally most of these are based on 'webfinger', which still binds identity to domain names and cannot support nomadic identity. The primary issues Zot faces are :

- Decentralized communications
- Independence from DNS-based identity
- Node mobility
- Seamless remote authentication

1.2.5 ActivityPub

Activity Pub is an open, decentralized social networking protocol providing a client/server API for creating, updating and deleting content, as well as a federated server-to-server API for delivering notifications and content. It has been defined by the W3C (in charge of standard of the web) and it provides two layers:

- A server to server federation protocol (so decentralized websites can share information).

- A client to server protocol (so users, including real-world users, bots, and other automated processes, can communicate with ActivityPub using their accounts on servers, from a phone or desktop or web application or whatever).

In ActivityPub, a user is represented by "actors" via the user's accounts on servers. User's accounts on different servers correspond to different actors. Every actor has an inbox to receive messages and an outbox to send messages to others.

1.3 Platforms specialization

When we look at the figure 1, we can see that many platforms have different types of content and as such, have completely different workings internally. It is important to notice that the Fediverse protocols must be as agnostic as possible of each OSN architecture to not require a complete shift of the code base to be able to use the protocol. Each of the platforms is specialized and most certainly use adapted techniques to reach the best performance for its type of content. One of the goal of the communication protocols must be to make the interoperability between two completely different platforms as easy as possible and as such only provide a core of features that do not depend on the features of the website and the type of content exchanged.

1.4 The rigidity of monoliths

All OSNs participating to the Fediverse are built as rigid monolithic applications often using a single relational database. This design leads to poor flexibility of purpose: it is difficult to adapt these OSNs, for instance to support new forms of social interactions. It is also difficult to isolate in these platforms the components that are critical for security, and in particular the ones that manipulate sensitive user data. A bug or exploit in any part of the monolith can allow a malicious attacker to read the entire database, and bugs are difficult to detect due to the size of the software. These monoliths illustrate three common issues that computer scientist have been plagued with since the beginning of the field :

- Maintainability : solutions to have systems modifiable and improvable as easily as possible have been the subject of many books and projects. Most software systems are modified all the time after they have been delivered either to add new features or fix bugs. As time is money, the efficiency and effectiveness with which issues can be resolved and enhancements can be realized is therefore important, if an application is too large and complex to understand entirely, it is challenging to make changes fast and correctly. Social networks are heavily impacted by their maintainability or lack there of, as they last a long time and are in constant evolution. In a monolithic architecture, all the components are grouped together leading to an exponential increase in required time to perform maintenance (single-tiered software application). This also means that switching technologies during the development is an arduous task, costly both in time and efforts.

- Cascading failure : Cascading failures may occur when one part of the system fails. When this happens, other parts must then compensate for the failed component. This in turn overloads these nodes, causing them to fail as well, prompting additional nodes to fail one after another. A bug in any module (e.g. memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug impact the availability of the entire application. One of the frequent source of crashes in social networks is simply the platform not having enough resources to handle all the requests when there is a huge number of users trying to access the application. While there is a lot of effort done to prevent these problems from happening, the monolithic architecture doesn't natively provide a solution.

- Scalability : it is the ability of a software to grow and manage increased demand. For a system to be highly scalable, it must allow for quick and efficient allocation/removal of resources according to the current requirements of the application. In a social network we can distinguish different components (authentication, notification, presentation, etc...), that are constantly being used but not at the same time. This means that since the components are all merged into one in a monolith architecture, the resources allocated to the platform must account for all of them at the same time. For example, if we have a sudden spike in connection request, the resources for the whole software must be increased and then decreased when no longer needed. The most common solution used in monolith is to have a very high static number of resources allocated at all time such no matter the number of requests, the website is able to handle them. This means that a large portion of the time, many resources are left unused, leading to a waste of money.

1.5 Explain history of decentralized social networks

1.6 Contribution is the foundation of the new social network with activityPub, forward the universal side

2 Technical introduction on the tools used

2.1 The role of micro-services PROS CONS

A micro-services architecture consists of a collection of small and autonomous services. Each service is self-contained and should implement a single business capability. They all have their separate codebase that can be deployed independently at any time. As such, they are responsible for persisting their own data and/or external state. To communicate with them, services publish their self-defined API that can be accessed by clients or any other services. This kind of architecture is well suited for large and/or complex applications requiring high scalability and efficient maintainability such as social networks.

2.1.1 Separation of concerns

To avoid cascading failure and making the maintenance easier to perform, services are independent to each others. This way, if the authentication service is no longer available due to some failure, the rest of the application is not in jeopardy because the service is self-contained. This also means that we can set fail-safe mechanism to quickly restart crashed services. This separation of concern also improve the maintainability simply by allowing us to implement new features or modify existing ones without touching the rest. This way, small teams of developers can each work in parallel on their own business case.

2.1.2 Scalability

The numbers of visits on a website is rarely linear, we may have more visitors in the evening than in the morning, leading to an increase in the required resources (RAM, storage, servers, etc...). On a monolithic application, this is often done by setting a static amount of resources, meaning that whenever the application is not at full capacity, we waste money. With micro-services, resources usage can be tailored to the actual need of the application. For example, if there is spike in connection requests, we can simply allocate more resources to the authentication service instead of upgrading the whole application and unrelated services.

2.2 Event-sourcing

IN DEPTH

Event-sourcing is an architectural pattern where a business object is persisted by storing a sequence of state changing events. Whenever an object's state changes, a new event is appended to the sequence. An entity's current state is reconstructed by replaying its events. Events are persisted in an event store, acting as a database of events and message broker at the same time. Services can either publish events or subscribe to them to be notified of the new ones. This is similar to the behaviour of users on social networks (post or follow). This pattern also provide two additional benefits for the whole application :

- History : As we do not store the current object state but each events starting at the creation of the objects, we have a complete trace of the previous events meaning that users' data is never lost and can be accessed at any time.
- Data protection : The micro-services architecture coupled with the event-sourcing patterns allows for data transparency, making it easier to comply with the law (GDPR, audit, etc...).

2.3 CQRS

CQRS is an architectural pattern that separates the data reading part (query) from the writing part (command), improving even further the separation of concerns. It makes it easier to scale read/write operations and to control security but introduce additional complexity to the system. In this architecture, this is represented by separating common services into a command service handling CRUD requests (PUT, POST, ...) and a query service handling the GET requests for the business case but both are connected to the same data storage. For example, in an

authentication service, creating an account calls the command service and retrieving an user's information calls the query service.

2.4 Cloud computing

Cloud computing makes computer system resources, especially storage and computing power, available on demand without direct active management by the user. The term is generally used to describe data centers available to many users over the Internet. Large clouds, predominant today, often have functions distributed over multiple locations from central servers. If the connection to the user is relatively close, it may be designated an Edge server. Clouds can be limited to a single organization, be available to many organizations or a combination of both. Cloud computing relies on sharing of resources to achieve coherence and economies of scale. It allows companies to avoid or minimize up-front IT infrastructure costs and allows enterprises to get their applications up and running faster, with improved manageability and less maintenance. It also enables IT teams to more rapidly adjust resources to meet fluctuating and unpredictable demand.

2.5 Gateway

An API gateway is programming that sits in front of an application programming interface (API) and acts as a single point of entry for a defined group of micro-services. Because a gateway handles protocol translations, this type of front-end programming is especially useful when clients built with micro-services make use of multiple, disparate APIs. A major benefit of using API gateways is that they allow developers to encapsulate the internal structure of an application in multiple ways, depending upon use case. This is because, in addition to accommodating direct requests, gateways can be used to invoke multiple back-end services and aggregate the results. Because developers must update the API gateway each time a new micro-service is added or removed, it is important that the process for updating the gateway be as lightweight as possible. In addition to exposing micro-services, popular API gateway features include functions such as :

- Authentication
- Security policy enforcement
- Load balancing
- Cache management
- Dependency resolution
- Contract and service level agreement management

2.6 Tools justification

2.6.1 Wolkenkit

The event-sourcing pattern is implemented through the Wolkenkit framework. It is a CQRS and event-sourcing framework for JavaScript and Node.js adapted to domain-driven design (DDD). It provides an event-store and a scalable real-time API. It is implemented as an additional service, handling activities events. According to the CQRS pattern, the query services subscribe each to the activities they are in charge of (posted, liked, etc...) and the command services all publish their related events (post, like, ...). Subscribed services will be directly notified of what is happening once an event they are looking for is published and they can then take the appropriate measures such as sending them to the front-end or modifying the targeted object.

2.6.2 Docker

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any platform. To handle the separations of concerns between services properly, Docker is used to isolate each of them into their own container making them only reachable through their exposed API. Docker-compose is used here to group all the services, starting them all in only one command.

2.7 Illustrate to the formal issue

3 Architecture design

3.1 ActivityPub

3.2 Patterns

3.2.1 CQRS

4 Implementation

4.1 ActivityStreams

4.2 Authentication

At the first, the authentication was supposed to be done using either the OpenID Connect or Passport.js framework. The issue was that the first one was clearly too much complex to implement for a simple authentication mechanism and the second was not really adapted to this micro-services application. As such, I implemented the authentication process using the JsonWebToken library as to have a strong token based authentication. Each user once connected or after creating an account, receive its unique token identifying the current user's session and store it in the cookies.

4.3 Testing

4.3.1 Unit testing

4.3.2 Integration testing

4.3.3 Load balance testing

5 Conclusion