"A foundation for extensible and decentralized social networks"

Lovisetto, Gary

Abstract

The world ascended to a new era of communications in the last decade due to the release of centralized social networks such as Facebook and Twitter. As the popularity of these platforms soared, many changes were made, new features were introduced and websites were born. Among these new platforms, a new trend emerged towards decentralized and open social networks (DOSNs). Instead of a single administrative authority collecting and storing social user data (friends, posts, comments, likes, etc.), these social networks operate as a federation of independent websites. A federation is a set of platforms where users connected to one platform can follow and interact with users from other networks while disallowing the concentration of data in a single domain. These platforms all have in common that they are open and target interoperability. The problem with all DOSNs participating in the federation is that they are built as rigid monolithic applications using a single relational database. T...

Document type : *Mémoire (Thesis)*

Référence bibliographique

Lovisetto, Gary. *A foundation for extensible and decentralized social networks.* Ecole polytechnique de Louvain, Université catholique de Louvain, 2019. Prom. : Riviere, Etienne.

École polytechnique de Louvain

A foundation for extensible and decentralized social networks

Author: **Gary Lovisetto**
Supervisor: **Etienne Rivière**
Readers: **Peter Van Roy, Charles Pecheur, Raziel Carvajal Gomez**
Academic year 2018–2019
Master [120] in Computer Science

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Etienne Riviere, for the continuous support of my thesis. His guidance helped me throughout the research and writing of this work. I would also like to thank the rest of my thesis committee: Peter Van Roy, Charles Pecheur and Raziel Carvajal Gomez, for taking the time to read and evaluate this work.

Gary Lovisetto

Abstract

The world ascended to a new era of communications in the last decade due to the release of centralized social networks such as Facebook and Twitter. As the popularity of these platforms soared, many changes were made, new features were introduced and websites were born. Among these new platforms, a new trend emerged towards decentralized and open social networks (DOSNs). Instead of a single administrative authority collecting and storing social user data (friends, posts, comments, likes, etc.), these social networks operate as a federation of independent websites. A federation is a set of platforms where users connected to one platform can follow and interact with users from other networks while disallowing the concentration of data in a single domain. These platforms all have in common that they are open and target interoperability. The problem with all DOSNs participating in the federation is that they are built as rigid monolithic applications using a single relational database. This design leads to poor flexibility of purpose: it is difficult to adapt these DOSNs, 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. The objective of this project is to investigate the use of recent advances in the engineering of cloud back-ends for building DOSNs. This research will focus on decentralization, personalization, and interoperability through the microservices architecture and event-sourcing pattern.

Contents

Chapter 1

Introduction

Decentralized and open social networks (DOSN) are alternatives to monopolistic social networks such as Twitter[1] or Facebook[2]. Where these mega-platforms are single administrative authorities, DOSNs operate in federations of independent and open networks. The platforms in the same federation being interoperable, users are able to interact with individuals spread across all websites. This also means that centralization is avoided due to the information being distributed on different servers.

With the recent scandal of Cambridge Analytica[3] and Facebook, the interest of using DOSNs becomes very clear. W3C, the organism in charge of standards of the web such as HTTP, XML, etc. defines ActivityPub, a protocol for the exchange of social data between decentralized DOSNs: Mastodon[4] (Twitter-like), GNUSocial[5] (Facebook-like) and Pleroma[6] (Twitter-like) interact in what is called the "Fediverse" (universe of federations).

They are built as rigid monolithic applications, typically using Ruby on Rails or PHP and using a single relational database. This choice of implementation limits the extensibility, customizability, and scalability of these platforms. Adding new forms of social interactions is difficult and it is hard to isolate components either for security or personalization purposes. This project is part of a global effort to improve DOSNs, the aim being to analyze the issues/needs inherent in the monolithic architecture and research alternative adapted designs.

[1] www.twitter.com

[2] www.facebook.com

[3] https://en.wikipedia.org/wiki/Facebook-Cambridge_Analytica_data_scandal

[4] https://mastodon.social/about

[5] https://gnu.io/social/

[6] https://pleroma.social/

1.1 Objectives

The purpose of this thesis is to investigate the use of recent advances in the engineering of cloud back-ends for building DOSNs, with a focus on decentralization, personalization, and interoperability with the Fediverse. To achieve this, we seek to implement the configurable backbone of a decentralized and open social network with tools adapted to the core business of these platforms. The goal of this implementation is to find a possible solution to a complementary set of requirements: the platform must be easily extendable and customizable by having the different components as isolated as possible (black boxes), allowing efficient addition/removal of features (e.g. novel forms of interactions or media). This loose coupling is also required to prevent modules critical for safety from causing cascading failures. As the messages exchange between users is the main part of social media, the platform must provide an efficient communication scheme compatible with the Fediverse. It should be noted that the goal is not to build a brand new social network, this will be the topic of another thesis[7].

1.2 Solution

This work aims to be a foundation for extensible DOSNs through the combination of different software engineering techniques:

- The microservices architecture consists in isolating each component of the application in its own container. Having modules being independent or loosely coupled signifies that new ones can be added or removed easily and that failures/bugs only impact their own service.

- Event-sourcing is an event-driven communication pattern that allows services to exchange high-level events. These events reflect the state updates of domain-specific principals (posts, profiles, events, etc.). This is very similar to social media users posting messages (publishers) and notifying their friends/followers (subscribers). This also improves extensibility as event-sourcing can easily accommodate changes such as new forms of social interactions.

- Command query responsibility segregation (CQRS) is a pattern where each component is segregated into a command and query service to provide a separation of concerns between write and read operations.

Through the combinations of microservices, event-sourcing, and CQRS, we implement a solution for the different objectives, fully compatible with the ActivityPub protocol.

[7] Project implemented by Adrien Widart under the supervision of Etienne Riviere

1.3 Plan of the thesis

In the motivation and problem statement chapter, we will explain what monopolistic social networks and their shortcomings are. We will discuss the solutions brought by open, decentralized and vertical social networks as well as their requirements. We will talk about how these new platforms have put an emphasis on personalization and collaboration through new communication schemes. In the next chapter, we will look into the software engineering techniques used in existing implementations and discuss their efficiency. We will then discuss our approach and the tools we chose to implement an efficient interoperable vertical DOSN. We also cover the history of DOSNs. The following section will go in depth on the implemented interoperability protocol, ActivityPub, and detail its requirements. The following chapter is going to explain the complete internal workings of the application, from its architecture design to future improvements, while going through an explanation of each service. This thesis will end on a section discussing the future of decentralized and open social networks, and the challenges to come.

Chapter 2

Motivation and problem statement

This chapter describes the current and previous implementations of social networks since 2004. It will introduce the issues faced by large platforms and their consequences on the design of new social networks. We will see what openness and decentralization are and how they impact the user quality of experience.

2.1 Social networks

A social network is a website that allows users to connect with different people 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 as with LinkedIn[1] and Yammer[2].

2.1.1 Data privacy and monopoly

The quick rise of social media led to a lot of problems that we are currently facing, the main one being the malicious usage of users' data. This was proven with the recent scandal of Cambridge Analytica where Facebook's user data was used in campaigns to influence votes during the last presidential election in the USA. This and previous scandals are partly responsible for the data privacy concerns. These worries led to a new era of internet where data protection and transparency must be explicit so as to enforce new digital rights. The general data protection regulation (GDPR) was one of the first signs that these new privacy and ownership concerns are getting more attention. It was introduced in 2018 and is Europe's answer

[1] www.linkedin.com

[2] www.yammer.com

to the different user data privacy breaches that happened in the last decade. It brought with it new fundamental rights such as having the right to ask a company to remove personal data or what personal data is being processed and to what end.

In addition to data privacy concerns, social networks like Twitter and Facebook are monopolistic. Their reach is so large that they simply become the default choice, the alternative is not even considered. This provides them immense power they can leverage to influence the market and shut down the competition. The reason why these platforms can maintain such superiority despite the existence of alternatives can be explained by the investments made and the "rich get richer" aphorism. Investments refer to the time spent by a user on its profile and all the contributions made to it. It is hard for an individual to leave a website where she or he has spent a lot of time. There is often a strong feeling of attachment with the platform. Rich get richer means that the higher the number of users there is on a website, the more likely it is for a user to choose that platform. People want to go where their friends are and advertisers want to reach the largest audience possible. Data privacy breaches and monopoly-related issues have nonetheless led to the implementation of social platforms providing solutions to these problems.

2.1.2 Open social networks

As an answer to data privacy concerns, a lot of new social networks have been created but with the twist that they are open. Openness is nothing new in the domain of computer science, it is a software development model that encourages open collaboration. The principle of open software development is peer production, with products such as source code, blueprints and documentation freely available to the public. These new open social networks (OSN) solve, to some extent, the data transparency issue. In addition to all the perks provided with openness such as customization, collaboration, etc..., the users can see how their information is handled and propose their own modifications to the code base of the network.

It is important to note that open-source is required for openness but not insufficiently, as many things can still be hidden. Openness is achieved by a combination of open-source and other principles such as open-governance, allowing any individual to submit changes to the code through platforms such as Git. While very good for developers, the impact on the user is much smaller because of the rich get richer problem and investments made on other platforms. To combat this trend and strive towards a much healthier market, OSNs have started working together to become decentralized and act as a federation.

2.1.3 Decentralized social networks

Instead of collecting and storing social users' data (friends, posts, comments, likes, etc...) on a single datacenter, these platforms are hosted on many independent servers all connected to the same network. A user or company is even able to host its own users and the content that they produce, such that they can control how personal data is handled. For example, the Diaspora[3] social network is composed 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 node on their own server or create an account on any existing node, from which they can interact with other users on other servers. This means that Diaspora users retain ownership of their data.

Figure 2.1 illustrates the difference between the two types of networks. A single node is in charge of everything in a centralized network, where, in a decentralized network, different autonomous groups can be formed. Another feature brought by decentralization is the interoperability between different platforms. By setting the same core rules, for every network, on how the data must be handled, they allow users from different networks to exchange information regardless of the implementation of the service they use. For example, Mastodon and Friendica[4] are both microblogging websites (Twitter-like) that are completely independent, yet, their users are able to communicate seamlessly by using the same communication standard.

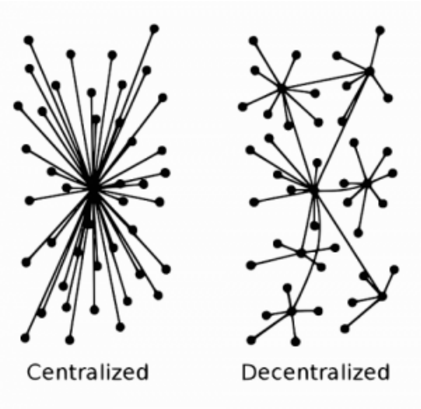


Figure 2.1: Topography of a centralized and decentralized network[5]

[3] https://diasporafoundation.org/

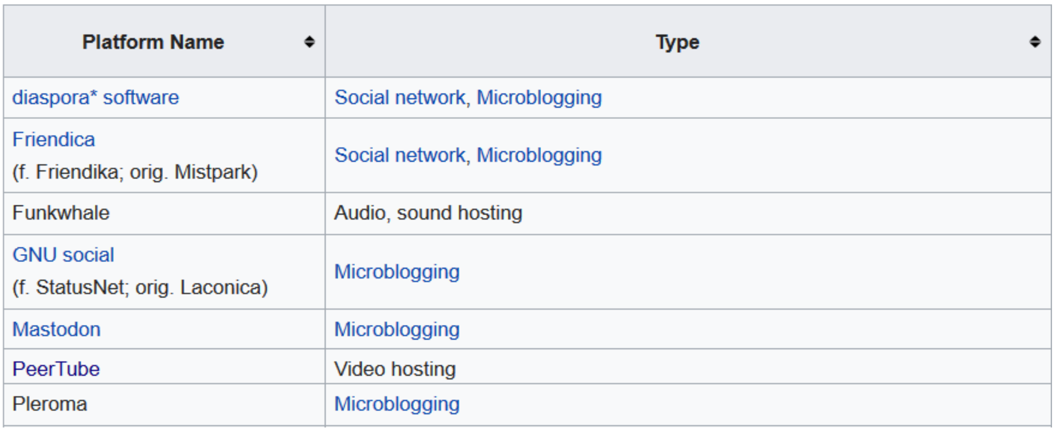
[4] https://friendi.ca/

[5] https://www.flickr.com/photos/jonphillips/33952492505, by Paul Baran. License CC BY 2.0.

2.1.4 Fediverse

Interoperable social networks behave as a federation of independent networks called the Fediverse. It is the set of federated servers that is 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 software running on them supports one or more communication protocols that follow open standards. As an identity on the Fediverse, users are able to post text and other media, or to follow posts from other identities. In some cases, users 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 their data (calendar, address book, etc...).

The software systems spanning the Fediverse, in Figure 2.2, cover a wide range of media and interactions like video hosting, microblogging or even sound hosting. Figure 2.3 is a snapshot of the connections between the `mastodon.host` node and all its neighbours. The full lines represent the communication between two nodes present in the Fediverse space. This represents only a restricted part of the federation, but we can already see hundreds of nodes and thousands of connections.



Platform Name	Type
diaspora* software	Social network, Microblogging
Friendica (f. Friendika; orig. Mistpark)	Social network, Microblogging
Funkwhale	Audio, sound hosting
GNU social (f. StatusNet; orig. Laconica)	Microblogging
Mastodon	Microblogging
PeerTube	Video hosting
Pleroma	Microblogging

Figure 2.2: Some of the social networks present in the Fediverse[6]

As each node represents a possible platform choice, this practice is completely adapted to the idea of a market where every individual is free to choose his preferred solution without losing the time previously invested. With the Fediverse, we now have independent websites that are completely different from each other and yet are able to communicate. This is a complete shift in the evolution of the social

[6] https://en.wikipedia.org/wiki/Fediverse. License Creative Commons Attribution-ShareAlike 3.0

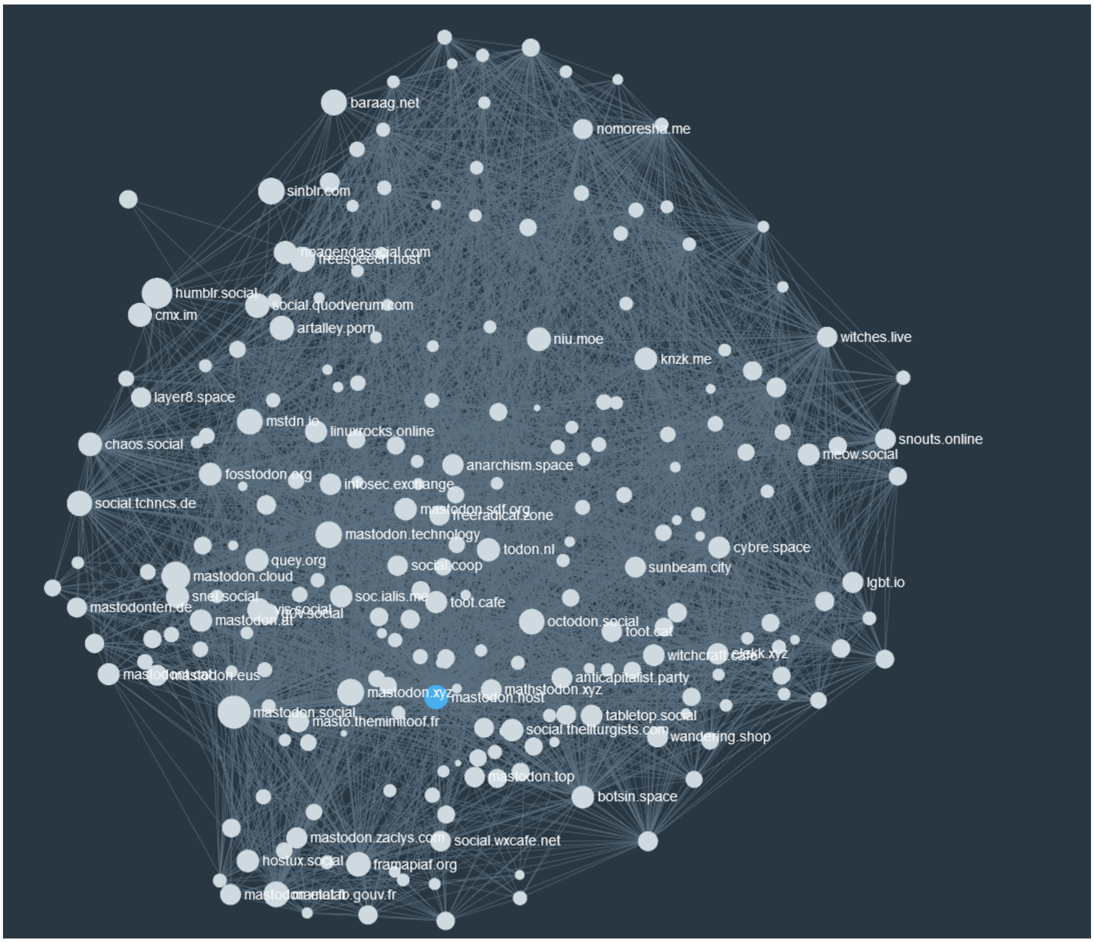


Figure 2.3: Screenshot of the connections in the Fediverse between mastodon.host and other nodes[7]

networks which was dictated for a long time by "the rich get richer". While this phenomenon is part of 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 the already established websites to seek continuous improvement as to not get outperformed.

[7] https://www.fediverse.space/, by Tao Bojlén. License AGPL.

2.1.5 Common issues with open and decentralized social networks

Everything is not perfect, one of the main concerns is that there are no real solutions in place to easily move the profile of a user between different networks or servers. Instead, an individual has to manually recreate a new account each time he or she 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 of the Fediverse and able to steal some of the data that transits in the federation. There are currently different solutions being worked on such as having the profile being transferable through blockchains to guarantee the protection of data[8]. The following list is not exhaustive but summarizes the disadvantages of decentralized networks implementation:

- The decentralization over multiple servers means that data is not quickly accessible, causing latency issues. It may require a search over the network to find its location, leading to longer response time.
- The ability of networks to be hosted in different locations means that to reduce latencies, or for other reasons, the same data might be present multiple times over the network. This creates a new requirement over the storage, its size must be greater than the total data size leading to an increase in cost. This issue is also common to mega-platforms duplicating their data in several data centers.
- A specific piece of data can be stored in many different locations and duplicated an unknown number of times such that removing it from the network becomes a very complex task. This also creates new problems related to the GDPR where some rights might be hard to enforce.
- Malicious attacks are handled independently by each network and, as they are interoperable, a single security breach, in a platform, could be used to contaminate other parts of the federation.

Part of the rise of decentralization and openness comes from the lack of customization in existing solutions.

[8] Evaluated through a master thesis under Prof. Etienne Rivière

2.2 The specialization of social networks

When new social networks are designed, the goal is often to provide different or better features. These new platforms are almost always implemented from the ground up despite sharing many similarities with existing solutions. This illustrates one of the issues with current social media: their components either cannot be exported or are very restrictive, limiting the customization possibilities.

2.2.1 Vertical networks

In recent years, there has been a new trend in social media called vertical social networks. These platforms contain a segmented user base specialized on specific topics and are often more private/gated. Specialization means that features must be adapted to the needs of each specific community. There are many examples of existing vertical networks:

- Clubface-Golf[9] is a social network solely dedicated to golf players where users only exchange information about that topic and can join different golf clubs to organize their schedule.
- The Edmodo[10] platform provides secure communication between people working in the field of education and allows them to share resources.
- Nextdoor[11] is a neighborhood social network allowing residents of the same town to communicate, find proximity services, and plan events.
- HumHub[12] is a free software and framework used to create extensible and customizable social networks. It is important to note that, while free, this application is not open and the created platforms are not interoperable.

Centralized social networks offered by large companies are ill-suited for such platforms because modifications cannot be made. The reliance on an outside solution implies that, should the access be lost, all the data and work done previously will no longer be accessible. In the case of Edmodo, using an external solution is dangerous if academic results of underage students can be collected and accessed by unauthorized individuals[13].

[9] https://clubface-golf.com/

[10] https://www.edmodo.com/

[11] https://nextdoor.com/

[12] https://humhub.org/en

[13] https://eduscol.education.fr/internet-responsable/ressources/legamedia/donnees-personnelles-et-monde-educatif.html

2.2.2 The importance of customization

When we look at Figure 2.2, we can see that many platforms have different types of content and as such, have completely different internal workings. Even if we compare two platforms with the same type of content, they work differently. For example, Mastodon is a clone of Twitter while Friendica is a clone of Facebook. Each of the platforms is specialized and most certainly uses adapted techniques to reach the best performance for its type of content. One of the goals of communication protocols is to make interoperability easily integrable, as to not require a complete shift of the code base in order to use them. This is done by providing a core of features that do not depend on the website and the type of content exchanged. However, the responsibility does not rely entirely on the interoperability protocol. Websites must also be designed such as to be as easily modifiable as possible.

The openness component of DOSNs means, for the most part, that they have a much higher degree of freedom when integrating new forms of social interactions. A few years ago, if a website wanted to add a communication feature where users could write their opinion or share knowledge, they had to either do it themselves or use one of the market leaders' solutions (facebook/google+ plugins) if they wanted more rich features. The issue stems, as always, from the dependency on another company if we want to implement already existing solutions. By being dependent, we lack the ability to modify features or control how the data is handled. Open social networks aim to fix this problem by providing completely independent and customizable components. No longer will these websites require to associate with Facebook to have a comments module, instead they will be able to pick the one that suits them the most among all the open-source solutions. They will also be able to pick a component and make changes that fit their requirements, be it modifying, deleting or adding features.

The House of European History[14] provides each visitor with a tablet to guide them through the exhibitions. An extension could be implemented where users would be able to leave comments and communicate with each other without having to authenticate through Facebook. Instead of starting from scratch and creating a brand-new module, they would be able to simply use the open-source component that suits them the most. We could also have an authentication component that requires an identity card, used by all governmental platforms, or a text chat adapted to communication during football matches to share feelings. If we take the example of the Clubface-Golf network, its features are close to Facebook with the addition of golf club management, yet, everything was built from the ground

[14] https://historia-europa.ep.eu/en/welcome-house-european-history

up. Instead of having to reinvent the wheel, we could simply take an existing microblogging component and add our own golf club management component on top of it. This concept of using components from different sources would naturally bring collaboration.

2.3 Interoperability protocols

Social networks have been operating for over a decade by relying on a centralized data store where all the information is gathered. This centralization is responsible for many issues such as the user's dependency on a specific company, leading to poor freedom of choice. To answer these issues, new protocols were designed with decentralization as the main focus, providing a new communication scheme between completely independent platforms such that users of different networks can communicate seamlessly. There have been many different approaches in the past ten years but all with the same core features that allow the following interactions:

- Asking for permission to establish a channel with another user: Bob sends a friend request to Alicia.

- Accepting or refusing the establishment of a channel with another user: Alicia accepts Bob request so they can now communicate through text, images, links or likes.

- Subscribing to another user to be notified of his actions: Bob follows Alicia, he will be notified of Alicia's publications.

- Polling available content from another user: Bob can see all of Alicia's publications.

- Cross-server authenticating so that communication is not impeded by requiring new authentication/authorization exchanges when crossing site boundaries: If Bob is authenticated on server A, he is able to send a message to Alicia even if she is authenticated on server B.

- Establishing group communications: Bob creates a group with all his friends where they can communicate.

2.3.1 Early approaches

The year 2010 saw the alpha release of Diaspora*[15], acting both as a DOSN and interoperability protocol (around 699.550 accounts). This platform was designed to address privacy concerns related to centralized social networks, by allowing users to set up their own server (or "pod") to host content. A key part of the original Diaspora software design concept was that it should act as a "social aggregate", allowing posts to be easily imported from Facebook and Twitter. The main problem with Diaspora* was that its code was never truly published and mainly built around the communication between Diaspora pods, leading to poor flexibility of purpose.

In the same year, a new microblogging platform was released, called Friendica (around 14.290 accounts). When creating the website, the goal of the developers was to provide a lightweight open and decentralized alternative to Facebook. Members of the Friendica work group started working on their own protocol for intercommunication called the Distributed Friends and Relations Network[16] (DFRN). The objective was to provide a solution for the lack of computer protocols for privacy-aware social networking on the web, much like Diaspora. The privacy capabilities having been over-engineered, no other services established secure links communications using the DFRN protocol. It was quickly dropped by the group in order to focus on their new protocol, much simpler and easier to use, Zot![17].

It is a streamlined privacy protocol for social networking, drawing on all the strengths of DFRN, but reduced down to the bare essentials. It had its brief popularity and was used by different platforms. Zot! was designed in parallel with their own social network and so presented the same issue as Diaspora*. OStatus[18] was a new protocol introduced in 2013 with the purpose of providing a common communication scheme between all networks. As it was the first iteration, many problems came to light and OStatus did not end up being the standard that it set out to be.

In 2014, the Social Web Working Group[19] was formed with the goal of designing the new interoperability protocol standard that came to be known later as ActivityPub[20]. Between the release of OStatus and ActivityPub, many new DOSNs were created such as Mastodon and GNU Social. GNU Social is a microblogging

[15] https://en.wikipedia.org/wiki/Diaspora_(software)

[16] https://web.archive.org/web/20120321204843/http://dfrn.org/dfrn2.pdf

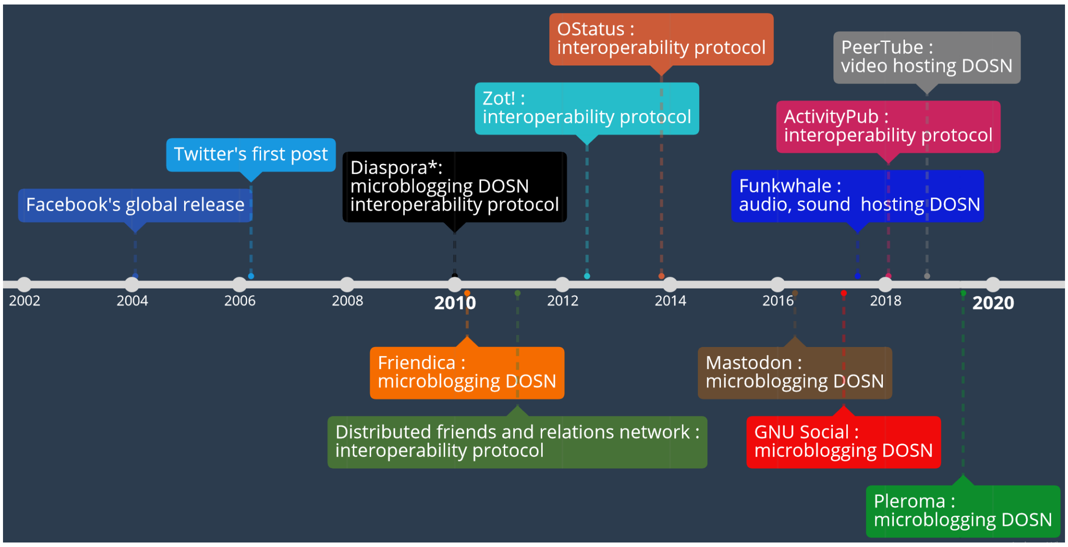
[17] https://zotlabs.org/help/fr/developer/zot_protocol

[18] https://en.wikipedia.org/wiki/OStatus

[19] https://www.w3.org/wiki/Socialwg

[20] https://www.w3.org/TR/activitypub/

network, implemented by the Free Software Foundation, working only with OStatus and hosting around 15.000 accounts. Just as Friendica provided an alternative to Facebook, Mastodon is a microblogging platform with the same features as Twitter with a total of approximately 2.163.859 accounts. The timeline observed in Figure 2.4 gives us a short summary of the chronology of social networks and interoperability protocols.



OStatus : interoperability protocol
PeerTube : video hosting DOSN
Zot! : interoperability protocol
ActivityPub : interoperability protocol
Twitter's first post
Diaspora*: microblogging DOSN interoperability protocol
Funkwhale : audio, sound hosting DOSN
Facebook's global release
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
Friendica : microblogging DOSN
Mastodon : microblogging DOSN
Distributed friends and relations network : interoperability protocol
GNU Social : microblogging DOSN
Pleroma : microblogging DOSN

Figure 2.4: Timeline of social networks and interoperability protocols

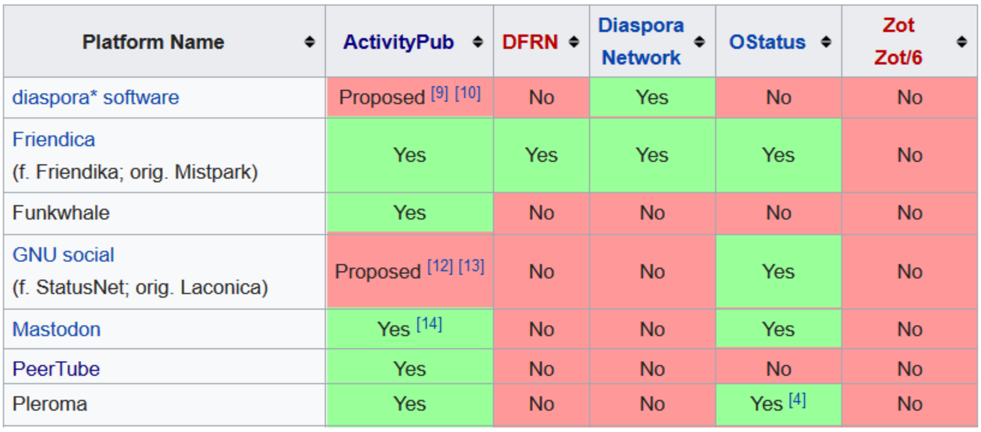
2.3.2 ActivityPub, the emerging standard

Activity Pub is an open and decentralized social networking protocol providing a client/server application programming interface (API) for creating, updating and deleting content. It also provides a federated server-to-server API for delivering this content. Its recommendation was endorsed by the W3C on the 23rd of January 2018. ActivityPub provides two layers:

- A server-to-server federation protocol for decentralized websites to share information.
- A client-to-server protocol for users to communicate with ActivityPub from any device or server.

ActivityPub implementations can integrate just one of these layers or the two of them. Once one is implemented, the requirements to implement the other are minimal. Using both has some benefits such as making the platform part of the decentralized social web and being able to use clients and client libraries that work across a wide variety of social websites. In ActivityPub, a user can possess multiple public identities on different servers called "actors". An actor identity can be a person, company, application, etc... and can be used by other authorized users. Every actor has an inbox to receive messages and an outbox to send messages.

The current state of implementation of the different interoperability protocols can be seen in Figure 2.5. Despite being much younger than the other protocols ActivityPub is already implemented by the majority of DOSNs and those who do not have it are thinking about integrating it[21] [22] [23]. If we look at all the decentralized and open social networks, ActivityPub is implemented by 22 out of the 30 most popular platforms[24]. In comparison, the second most used protocol is OStatus with only 6 networks using it.



Platform Name	ActivityPub	DFRN	Diaspora Network	OStatus	Zot Zot/6
diaspora* software	Proposed [9] [10]	No	Yes	No	No
Friendica (f. Friendika; orig. Mistpark)	Yes	Yes	Yes	Yes	No
Funkwhale	Yes	No	No	No	No
GNU social (f. StatusNet; orig. Laconica)	Proposed [12] [13]	No	No	Yes	No
Mastodon	Yes [14]	No	No	Yes	No
PeerTube	Yes	No	No	No	No
Pleroma	Yes	No	No	Yes [4]	No

Figure 2.5: Implementation state of interoperability protocols by platform[25]

[21] https://github.com/diaspora/diaspora/issues/7422

[22] https://socialhome.network/content/83203/federation-socialhome-federates-using-the-di/

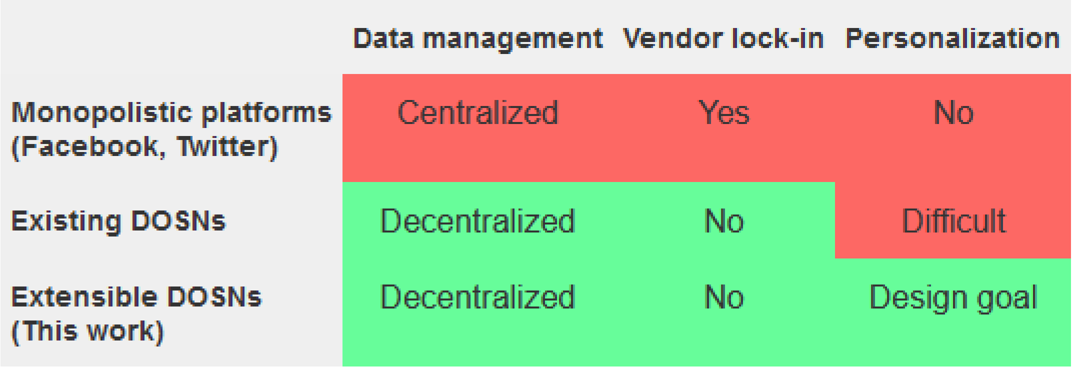
[23] https://git.gnu.io/gnu/gnu-social/issues/256

[24] https://en.wikipedia.org/wiki/Fediverse#cite_note-14

[25] https://en.wikipedia.org/wiki/Fediverse. License Creative Commons Attribution-ShareAlike 3.0

2.4 Summary of the problem statement

Mega-platforms like Facebook and Twitter introduced many new problems: breaches in data privacy, centralization of information, user dependency on the platform and lack of true customization. Decentralized and open social networks originated from the need to correct the first two issues. The combination of openness and decentralization provides the users with the ability to manage their data privacy and change platforms at will and minimum costs. The interoperability between websites of a federation leads individuals to make choices based on their preferences instead of being forced toward specific solutions. These new networks also guide developers towards building modular networks such that they become aggregations of independent blocks. These blocks can then be switched easily and at any time to fit the new requirements. Figure 2.6 illustrates the weaknesses and strengths of the different types of social networks. This works aims to provide a solid foundation, built from scratch, for vertical DOSNs to build upon. In the following chapter, we will discuss what software engineering techniques are good for implementing the foundation for an extensible DOSN with efficient separation of concerns, isolation, and customization.



	Data management	Vendor lock-in	Personalization
Monopolistic platforms (Facebook, Twitter)	Centralized	Yes	No
Existing DOSNs	Decentralized	No	Difficult
Extensible DOSNs (This work)	Decentralized	No	Design goal

Figure 2.6: Design choices between each kind of social network

Chapter 3

What software engineering for extensible DOSNs ?

In the motivation and problem statement, we talked about how decentralized and open social networks came to be and the flaws they were meant to correct. This chapter will discuss good approaches in software engineering to implement an efficient extensible DOSN. The goal is to determine if existing platforms use adapted tools to reach the objectives, namely if they can provide optimal components isolation and extension as well as an efficient communication scheme.

3.1 The rigidity of monoliths

All DOSNs participating in the Fediverse are built as rigid monolithic applications often using a single relational database. A monolithic application describes a software where every component is merged into a single program such that they all run on a single address space on the same server. This design leads to poor flexibility of purpose: it is difficult to adapt these DOSNs, for instance to support new forms of social interactions. It is also difficult to isolate the components that are critical for safety 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 engineers have been plagued with since the beginning of the field.

3.1.1 Maintainability and extensibility

Solutions to have systems modifiable and improvable as easily as possible have been the subject of many books and projects. Most software systems are often modified after they have been delivered either to add new features or fix bugs. As time is money, the efficiency with which issues can be resolved and enhancements made is therefore important. If an application is too large and complex to understand entirely, it is challenging to make quick and correct changes. Social networks are heavily impacted by their maintainability or lack thereof, as they last a long time and are in constant evolution. Efficient maintainability means that the application is easy to manipulate/fix and that these actions do not increase its complexity too much.

Extensibility is a design principle where we take the future of a system in consideration during its implementation. We can say that a system is extensible or not, based on the level of effort required to add features (extend the capabilities of our software). The goal is to integrate enhancements as easily as possible to minimize the impact on the project as a whole. In a monolithic architecture, the components are grouped together, leading to an exponential increase in the time required 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 effort. A system having poor maintainability will often have poor extensibility. If small amendments are tenuous to perform, the large changes brought by a new module or components will be heavily impacted. An extensible application must be built to be flexible around changes but also around demands.

3.1.2 Scalability

Scalability is the ability of softwares 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. In a social network, we can distinguish different components (authentication, notification, presentation, etc...), that are constantly being used at different times. When components are merged into a single application (monolith), the resources allocated to the platform must encompass all of them. For example, if we have a sudden spike in connection requests, the resources for the whole software must be increased and then decreased when no longer needed. The most common solution used in monoliths is to have a very high static number of resources allocated at all times, such that no matter the number of requests, the platform is able to handle them. For a large portion of the time, many resources are left unused, leading to a waste of money and energy.

If we take the example of a text chat application used during football matches we can quickly understand the need for efficient scalability. The enthusiasm of supporters during a match is, usually, related to the performance of the team they root for. While not much will be posted for the majority of the play time, the amount of information exchanged will spike during events such as goals, faults and such. As users will rather stay focused on the match than start a debate, the spike time will amount for a small portion of the match length. A system where all the resources are statically allocated at the start of the match, to handle spikes, will be wholly inefficient as the majority of resources will not be used for the biggest part of the match.

3.1.3 Cascading failures

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. One of the frequent sources of crashes in social networks is 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 does not natively provide a solution.

3.1.4 The need for appropriate tools

The design of monoliths itself goes against the principle of decentralization and customization. By being a single entity, the scalability, maintainability and extensibility efficiency is largely diminished. Customizability is also severely impacted because components cannot be isolated as they are not independent. Even in the best case, each modification makes the maintainability much harder to guarantee. Moderate changes in a monolith always require the engineer to be, at least, familiar with a large part of the code-base as to anticipate the consequences of his actions. All of these issues make monoliths inadequate for open, decentralized, and personalizable social networks, new techniques are needed.

3.2 The role of microservices

Social networks have special needs that the monolith architecture cannot provide. It was thus imperative to find an appropriate solution as the issues become more visible with the increasing number of users and the advent of vertical social networks. A new architecture, called microservices, consists of a collection of small and autonomous services communicating through lightweight protocols. Each service is self-contained and implements a single business capability. They all have their separate code base that can be deployed independently. As such, they are responsible for persisting their own data. To communicate, 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 streaming platforms (Netflix[1]), social networks (Uber[2]) and e-commerce websites (Amazon[3]).

3.2.1 Separation of concerns

To avoid cascading failure and enabling maintainability, services are independent. This way, if the authentication service is no longer available due to some failure, the rest of the application is not in jeopardy as that service was self-contained. This separation of concerns also improves extensibility by allowing to implement new features or modify existing ones without changing unrelated services. This way, small teams of developers can work in parallel on their own business case without interfering with the work of other groups. Services not being reliant on each other make them easily replaceable or usable elsewhere as independent components. As only the APIs of the services are exposed to the outside world, they behave like black-boxes (language agnostic). They can be plugged easily in other applications to allow for easy and efficient customization.

3.2.2 Scalability

The number of visits on a website is rarely constant, we may have more visitors in the evening than in the morning, leading to an increase in the required resources (RAM, storage, servers, etc...). With microservices, resource usage can be tailored to the actual need of the application. For example, if there is a spike in connection requests, we can simply allocate more resources to the authentication service instead of upgrading the entire application. In Figure 3.1, we can see that scaling

[1] https://www.youtube.com/watch?v=CZ3wIuvmHeM

[2] https://eng.uber.com/soa/

[3] https://aws.amazon.com/fr/microservices/

in monolithic application means replicating the monolith on multiple servers. With microservices applications, we can individually scale each part of the domain.

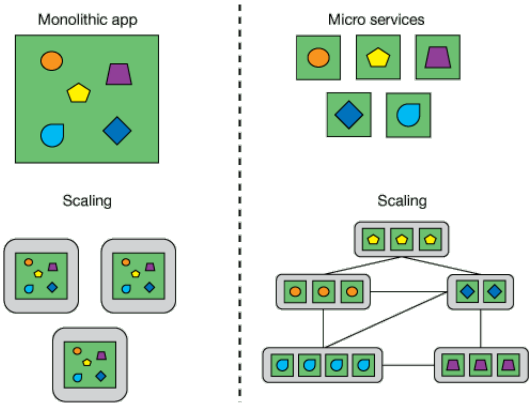


Figure 3.1: Monolith vs microservices scalability[4]

To illustrate better how microservices allow for component-oriented scalability, we will re-use the previous football application example. This text chat application is really simple, there is one service handling account creation/authentication and another in charge of communications between users. Our first service will not see much traffic except for a few minutes before a match, where everyone will connect to the application and authenticate. Shortly after the start, the authentication service will stop seeing any data. During the match, it is the communication service that will deal with the majority of the traffic.

The microservices architecture is perfectly suited for this kind of application because it allows us to dynamically and temporarily allocate (or remove) a large number of resources to individual services. In that way, we can give the required power to the first service before the start and then downscale it after. The second service resource requirements can be handled completely independently and in parallel to handle sudden spikes (e.g. when a player scores). Unfortunately, this architectural model also has some downsides.

3.2.3 The disadvantages of microservices

There is no doubt that this architecture is perfectly suited for social networks with large user bases but they also have their share of drawbacks. A standard microservices application is exposed to different problems:

[4] https://apilama.com/2018/07/18/scalability-design-with-microservices/, by Michael Petychakis.

- As independent as services are, there still exists coupling between them as they need to interact with each other. Enhancing and refactoring services means that the adjustment of other interfaces is often required to avoid introducing new failures, making the platform harder to modify.
- Developers must be aware of how the services operate with each other, to make the appropriate modifications. These developers must often possess a large skill set as the services in the same application can use completely different languages, databases, and libraries according to their business logic.
- A single page/screen load might require several calls to multiple services. It can result in multiple network round trips between the client and the server, adding significant latency.
- Every service is exposed to the external world, making the attack surface larger. The smaller the attack surface is, the more secure the application can be. Exposing the whole application to the client is not optimal.
- Each publicly published service must handle its own cross-cutting concerns (e.g. authorization).

To fix these problems, an API gateway is required.

3.3 The need for an API Gateway

An API gateway is a door that sits in front of an API and acts as a single point of entry for a defined group of microservices. This type of front-end programming is especially useful when clients built with microservices 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. In addition to accommodating direct requests, gateways can be used to invoke multiple back-end services and aggregate the results. Most gateways features also include functions such as authentication, security policy enforcement, load balancing, and cache management.

If we take the example of the Amazon gateway in Figure 3.2, we see how it serves both as query optimizer and watchdog. Clients do not have direct access to the AWS services but must first pass through the gateway. Each request will be pre-emptively analyzed and, if the cache already contains the result to the user's query, the gateway may directly answer. The CloudWatch application is used in the gateway to monitor the application, optimize the resources by knowing in advance which services are going to be heavily solicited and aggregate operations to reduce the load on the network. To improve monolithic implementations further, and have a solid base for building vertical DOSNs, the event-sourcing pattern is required.

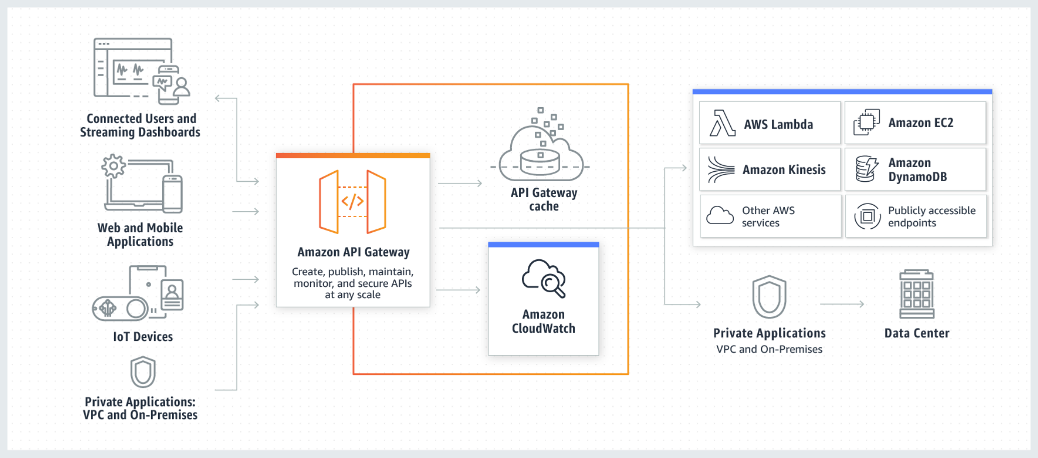

Connected Users and Streaming Dashboards
Web and Mobile Applications
IoT Devices
Private Applications: VPC and On-Premises
Amazon API Gateway
Create, publish, maintain, monitor, and secure APIs at any scale
API Gateway cache
Amazon CloudWatch
AWS Lambda
Amazon EC2
Amazon Kinesis
Amazon DynamoDB
Other AWS services
Publicly accessible endpoints
Private Applications
VPC and On-Premises
Data Center

Figure 3.2: Amazon API Gateway for the different AWS services[5]

3.4 Event-sourcing

As monoliths have poor flexibility of use, adding new forms of social interactions is complex and often increases the overall maintainability. While extensibility is one of the issues, it is not the only one. The use of a single relational database is as common as inadequate in the case of social networks. These kinds of applications are designed to deal with a large amount of communications happening in a short time and traditional databases, such as SQL, are not well suited for that kind of interaction. To handle data exchange and storage in distributed social networks as well as making extensibility easier, we use the event-sourcing technique.

Event-sourcing is an architectural pattern where a business object is persisted by storing a sequence of state changing events. Whenever the state of an object changes, a new event is appended to the sequence. An entity's current state is reconstructed by replaying its events, persisted in the 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. Event-sourcing also provides good consistency which is especially important when building a solid backbone. Event persistence and communication being a single atomic event, it guarantees failure resistance.

[5] https://aws.amazon.com/fr/api-gateway/

It is very natural to design social networks to be event-driven, where services publish and subscribe to events, followers (subscribers) are notified of every action done by the users they follow (publishers). This allows great improvements to extensibility because social features are producers and consumers of events: every social interaction will produce a specific event that will be consumed by the subscribers. This leads to efficient extensibility because adding a new feature will only require the specification of a new type of event to establish communication with the new service. This pattern also provides multiple additional benefits for the whole application:

- The application state can be completely rebuilt from scratch by replaying the events from the event-store.
- We have a complete trace of what happened previously so that data is never lost and previous states can be accessed at any time.
- If any event received in the past was incorrect, we can replay the events up to that point and remove the erroneous event. We can then replay the following events to obtain the correct current state.
- Data protection: The microservices architecture coupled with the event-sourcing patterns allow for data transparency, making it easier to comply with the law. In the case of an audit, for example, we can quickly access all the previous states of an object to see if there has been any data tampering or privacy violation. This transparency allows for complete visualization of how data is handled and what services are using it.

To illustrate better, consider the web shop application of Figure 3.3. When the user performs a specific action such as adding an item to its cart, removing items or paying, a corresponding event is created. Each new event is persisted in the event-store and then published on the event bus. Once an event is published, the view will be notified of events with the topic it has subscribed to and the same is also true for external services/applications. Should the client wish to make a purchase, we will query the event-store for all events relative to his cart. We will then replay them to obtain the list of desired items. We can also retrieve items he removed from his cart if he wants to add them back. Microservices and event-sourcing can be improved even further with the integration of a last pattern.

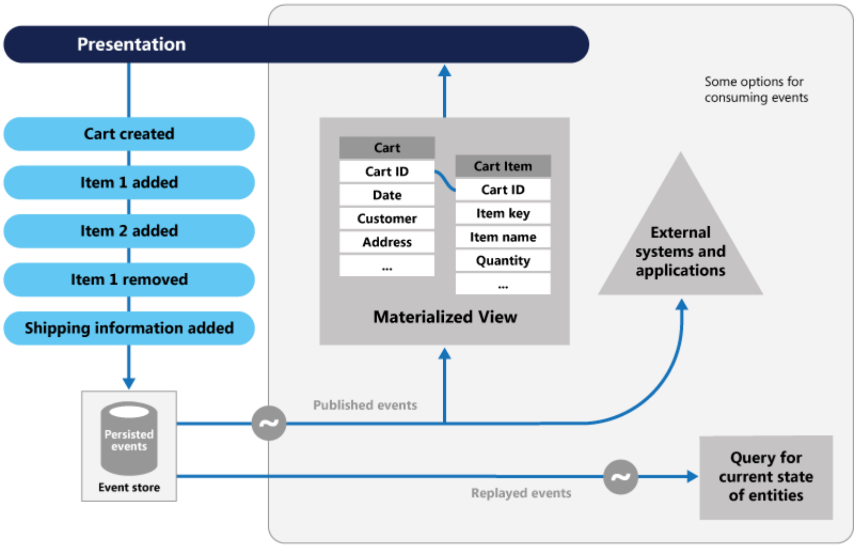


Figure 3.3: Event-sourcing in an online store[6]

3.5 Command Query Responsibility Segregation

With microservices, resources can be efficiently assigned where they are needed. However, scaling each service individually is not yet perfect. Based on the platform, the read and write requirements for the same service can be highly different. For the European House of History application described previously, we want to provide a solution for users to write and share comments. This is much more read oriented because users will spend time reading the comments and descriptions about the objects. However, our football text chat is more about sharing feelings on the moment and is thus heavily skewed towards writes. The solution is to use Command Query Responsibility Segregation (CQRS).

CQRS is an architectural pattern that separates the reading part (query) from the writing part (command), improving even further the separation of concerns. This is represented by separating common services into a command service handling CRUD requests (PUT, POST, ...) and a query service handling the GET requests. This pattern also improves extensibility as we can change one side without affecting the other. Data consistency is therefore not altered. Both of these benefits fully leverage the possibilities brought by the microservices architecture. One of the ideas behind CQRS is that a database is hardly as efficient to manage both reads and writes. It can depend on the choices made by the software vendor, the database used, etc...

[6] https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing

For most use cases, CQRS is required when we implement event-sourcing because we want to retrieve the current state without having to replay all the events. Applying CQRS on top of event-sourcing means persisting each event on the write part and deriving the read part from the sequence of events. Figure 3.4 illustrates how CQRS and event-sourcing work in pairs. When the user creates or modifies something, we call the command API to perform the write operation. The related event is created, stored in the event-store and published in the event bus to notify all subscribed services. When users wish to retrieve their data, we call the API of the query service to perform the read operation. According to the implementation, the querying service can either replay all the events to get the current state or get the data from a reporting view. This view would be updated each time there is a new event and would only store the current state.

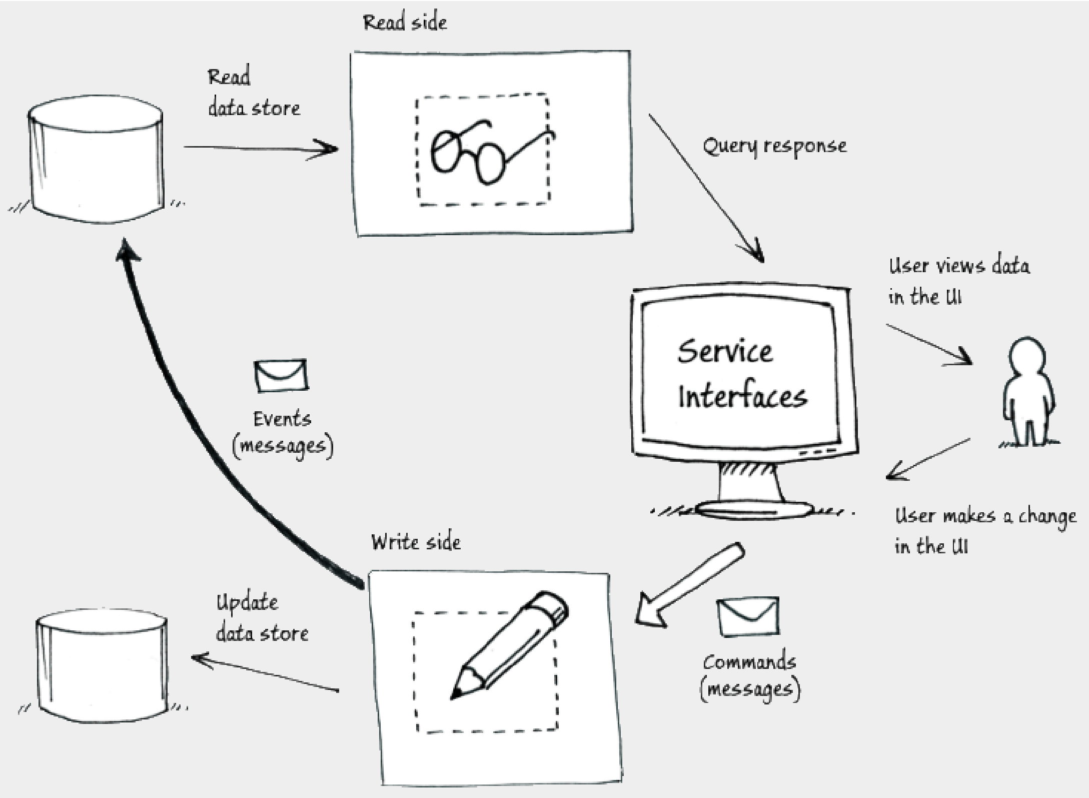


Figure 3.4: Path of the data when using CQRS and event sourcing[7]

[7] https://medium.com/eleven-labs/cqrs-pattern-c1d6f8517314, by Romain Pierlot.

3.6 Summary of the issues and their solutions

All DOSNs participating to the Fediverse are built as rigid monolithic applications, typically using Ruby on Rails or PHP and using a single relational database. This design leads to poor flexibility of purpose: it is difficult to adapt these DOSNs, for instance, to support new forms of social interactions. It is also difficult to isolate in these platforms the components that are critical for safety 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.

With the architecture shift from monolithic to microservices, we have a strong way to ensure that DOSNs will be able to efficiently scale their resources according to the load of the network as each service is independent. This absence of dependence upon other services also means that the components critical for safety can be isolated to better handle sensitive information. Improving previous features and supporting new forms of social interactions is also done easily thanks to the segregation of services, allowing teams to work in parallel. CQRS goes a step further in isolating services as to better distribute resources by separating the read and write part. Both the independence and the more efficient extensibility allow for a wide range of customization possibilities on the components of each platform. Event-sourcing further enhances the overall solution by also improving the extensibility and consistency. We posit that implementing an interoperable vertical DOSN requires the combinations of these technologies.

Chapter 4

ActivityPub as a backbone

In the last decade, several communication schemes have been designed with the goal of providing interoperability between open and decentralized social networks. However, just as many implementations ended being inadequate due to their focus on specific platforms and features. In 2018, after 4 years of work, the Social Web Networking Group of the World Wide Web Consortium (W3C) published the ActivityPub protocol as a W3C Recommendation. The goal was to introduce a standard for all communications between DOSNs while addressing the problems of previous protocols.

4.1 Overview

ActivityPub is an open and decentralized social networking protocol providing an API for creating, updating, deleting content, and delivering notifications. It provides a server-to-server federation protocol for decentralized websites to share information and a client to server protocol for users to communicate with, from any device or server. In this protocol, a user is represented by one or multiples identities called "actors" that are unique for each platform. Every actor has an inbox to receive messages and an outbox to send messages to other actors. Figure 4.1 illustrates communications using the protocol.

A user is able to retrieve all his messages by doing a GET on his inbox or retrieve all the messages from others by doing a GET on their outbox. In the same fashion, a user can POST to someone's inbox to send them a message or POST to his outbox to publish his message to the world. A user posting a message to his own outbox is similar to a user posting a message on Twitter (tweet): every follower will be notified and anyone, who is not blocked, will see the new tweet when visiting the user's page.

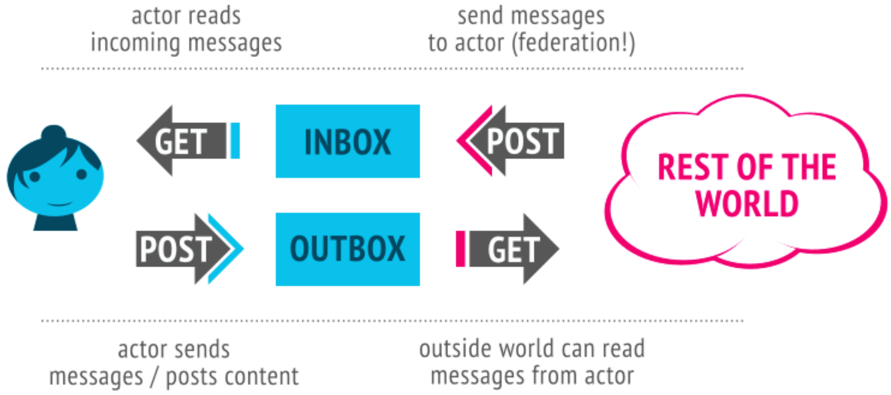


Figure 4.1: Message passing in ActivityPub[1]

For example, if Alyssa wants to send a message to Ben, Alyssa POST it to her outbox and then the server will POST the message to Ben's inbox. This also shows us that the protocol is very much event-based, actors publish messages to the servers and all the followers (subscribers) are notified. The messages transferred, using ActivityPub, are called activities and their vocabulary is defined by a W3C Recommendation called ActivityStreams[2].

4.2 ActivityStreams

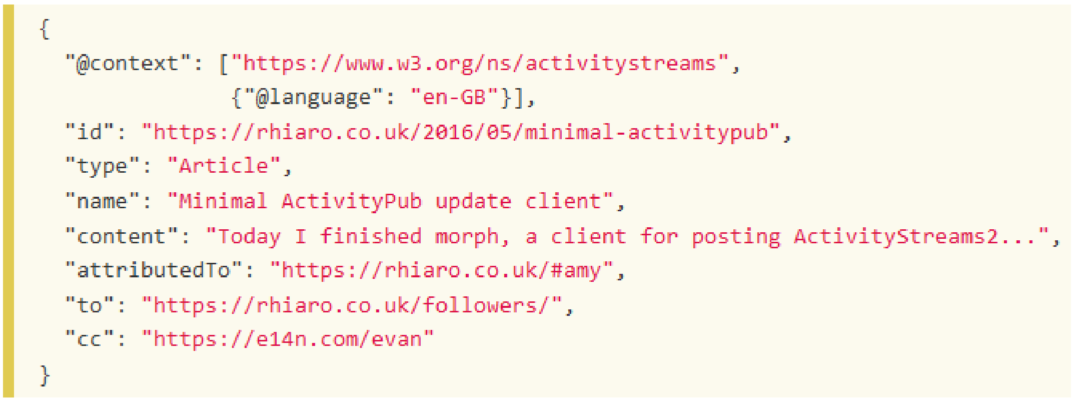
This specification details a model for representing objects as well as potential and completed activities using the JSON format. It is intended to be used with vocabularies that detail the structure of activities and define specific types of activities. The goal of this specification is to provide a JSON-based syntax that is sufficient to express metadata about activities in a rich, human-friendly but machine-processable and extensible manner.

4.2.1 Object, activities and properties

Objects are the primary base type in ActivityStreams. They are used to represent all the entities in social networks such as actors, documents, notes, relationships, etc.... They must all have a unique global identifier and a type in addition to all the properties required for their type. Messages sent by the client to the server can

[1] https://www.w3.org/TR/activitypub/, by Social Web Working Group. License W3C.
[2] https://www.w3.org/TR/activitystreams-vocabulary/

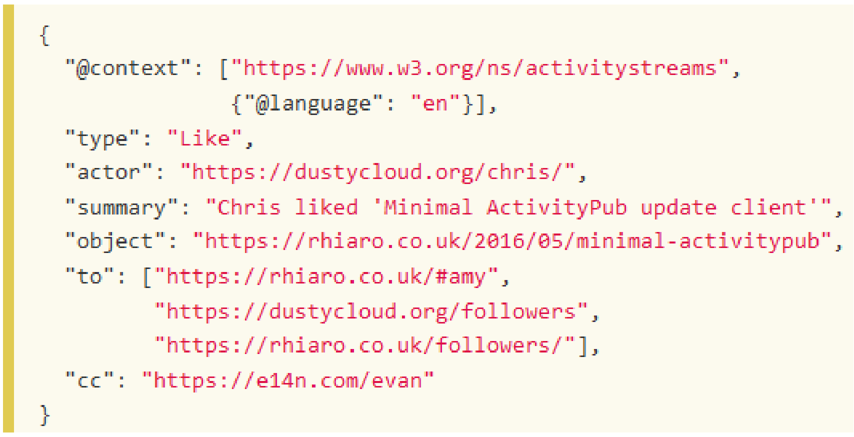
be either newly created objects or activities. An activity is a specialization of the base 'Object' type, providing information on the type of action performed. Figure 4.2 represents an object of type 'Article' with all its required fields detailing who it belongs to (Amy), its content, name and who can see it (Amy's followers and Evan).



```
{
  "@context": ["https://www.w3.org/ns/activitystreams",
               {"@language": "en-GB"}],
  "id": "https://rhiaro.co.uk/2016/05/minimal-activitypub",
  "type": "Article",
  "name": "Minimal ActivityPub update client",
  "content": "Today I finished morph, a client for posting ActivityStreams2...",
  "attributedTo": "https://rhiaro.co.uk/#amy",
  "to": "https://rhiaro.co.uk/followers/",
  "cc": "https://e14n.com/evan"
}
```

Figure 4.2: 'Article' object

When Chris likes this article, a new activity is created and sent to all relevant users, as seen in Figure 4.3. Each object and each combination of objects and activities has its own set of mandatory properties. When dealing with properties specifying an id or collection of objects (e.g. list of followers), the value will be the location where the data can be retrieved.

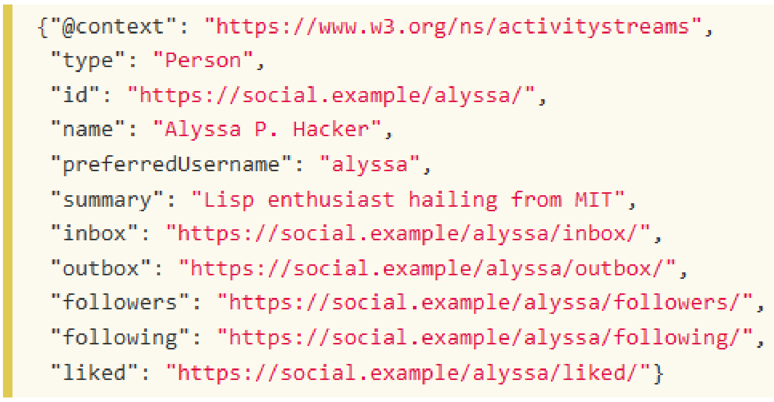


```
{
  "@context": ["https://www.w3.org/ns/activitystreams",
               {"@language": "en"}],
  "type": "Like",
  "actor": "https://dustycloud.org/chris/",
  "summary": "Chris liked 'Minimal ActivityPub update client'",
  "object": "https://rhiaro.co.uk/2016/05/minimal-activitypub",
  "to": ["https://rhiaro.co.uk/#amy",
         "https://dustycloud.org/followers",
         "https://rhiaro.co.uk/followers/"],
  "cc": "https://e14n.com/evan"
}
```

Figure 4.3: 'Like' activity with an article as the object

4.3 Retrieving objects

In ActivityPub, the identifier of an object is always a publicly dereferenceable URI (e.g. https URI) with its authority belonging to that of its originating server. These URIs can be used to retrieve the content of other properties like the one required in 'Actor' objects. Figure 4.4 shows the different properties present in an 'Actor' object. The 'inbox', 'outbox', 'followers', 'following' and 'liked' properties all contain the URI to call for retrieving the relevant information as JSON objects. For instance, sending an HTTP GET request to `https://social.example/alyssa/inbox/` will return all the messages destined to Alyssa directly or indirectly (i.e. follower of a follower). The first step of communication is always to retrieve the address of the recipient's inbox or outbox through his id URI.



```
{"@context": "https://www.w3.org/ns/activitystreams",
 "type": "Person",
 "id": "https://social.example/alyssa/",
 "name": "Alyssa P. Hacker",
 "preferredUsername": "alyssa",
 "summary": "Lisp enthusiast hailing from MIT",
 "inbox": "https://social.example/alyssa/inbox/",
 "outbox": "https://social.example/alyssa/outbox/",
 "followers": "https://social.example/alyssa/followers/",
 "following": "https://social.example/alyssa/following/",
 "liked": "https://social.example/alyssa/liked/"}
```

Figure 4.4: Alyssa's information represented as an 'Actor' object

4.4 Client-to-server interactions

Client-to-server interactions take place through clients posting activities to an actor's outbox. To do this, clients can retrieve the location of the outbox from their profile and then send it a POST request. For client-to-server interactions, a request may contain either an activity or an object. In the second case, the server must wrap the object into a 'Create' activity. There are 9 possible interactions through activities :

- The 'Create' activity is used when posting a new object. The object embedded in the activity is created.
- The 'Update' activity is used when updating an existing object. The information must be modified with the new values present in the activity.

- The ’Delete’ activity is used when deleting an existing object. The object is then deleted or replaced with a tombstone (reference to a deleted object).
- The ’Follow’ activity is used to subscribe to the activities of another actor.
- The ’Add’ activity is used to add an object into a specific collection (e.g. adding an actor in the followers’ collection).
- The ’Remove’ activity is used to remove an activity from a specific collection (e.g. remove an actor from the user’s followers collection).
- The ’Like’ activity is used to indicate that an actor likes an object.
- The ’Block’ activity is used to prevent the actor defined in the object from interacting with activities posted by a specific actor.
- The ’Undo’ activity is used to undo a previous activity.

4.5 Server-to-server interactions

Once an activity has been posted to an actor’s outbox, the outbox server must forward it to all the recipients’ inboxes across the different servers and interoperable platforms. The location of the receivers’ inboxes is obtained by looking up the actor’s profile through the ’to’, ’bto’, ’cc’, ’bcc’, and ’audience’ properties. During the delivery phase, the recipients’ inbox servers will perform any actions related to the activities received, such as incrementing the like count of an object when receiving a ’Like’ activity and delivering the activities to the corresponding recipients. If an activity is sent to the public collection, it is not delivered to any actors but viewable by all in the actor’s outbox. Server-to-server interactions are the same as client-to-server interactions with a few new additions:

- The ’Follow’ activity is used to request the subscribe to the activities of another actor. It must be followed by an ’Accept’ or ’Reject’ activity (reject by default).
- The ’Accept’ activity is used to accept an activity.
- The ’Reject’ activity is used to reject an activity (e.g. reject a follow activity if the actor is blocked).
- The ’Announce’ activity is used when sharing/reposting an object.

Chapter 5

Implementation

We present an implementation of the ActivityPub protocol using a microservices architecture, event-sourcing, and CQRS.

5.1 Architecture

The first step before designing the architecture is to define the different services present in the application. From the W3C Recommendation, five features of social networks can be identified: posting some information (note), liking an object (like), sharing an object (share), following an actor (follow), and blocking an actor (block). These fives components represent the base of the microservices architecture, to which we add the authentication service (user) and the service handling the different identities (actor). The next step is to duplicate these components into two groups, the outbox, and inbox set of services for the client-to-server and server-to-server interactions. We must then further divide our services into two parts, read (query) and write (command), according to the CQRS pattern.

All of this is illustrated in Figure 5.1, representing the architecture of the application. On the right side, we have the user and actor services, each with their own databases. The outbox command services are called when an actor posts an activity to its outbox. From there, the activity will be delivered to every recipient's server. Our recipient's servers, represented by the inbox command services, will start the delivery phase once it has received an activity. It will make updates related to the side effects of the activity received and then publish it to the event-store. Whenever a new activity is published, the query services subscribed to that type of activity (note, like, follow, share or block) will be notified and able to deliver it to all recipients' inboxes. It must be noted that the front-end is not an implementation goal, it only serves as an illustration for messages passing.

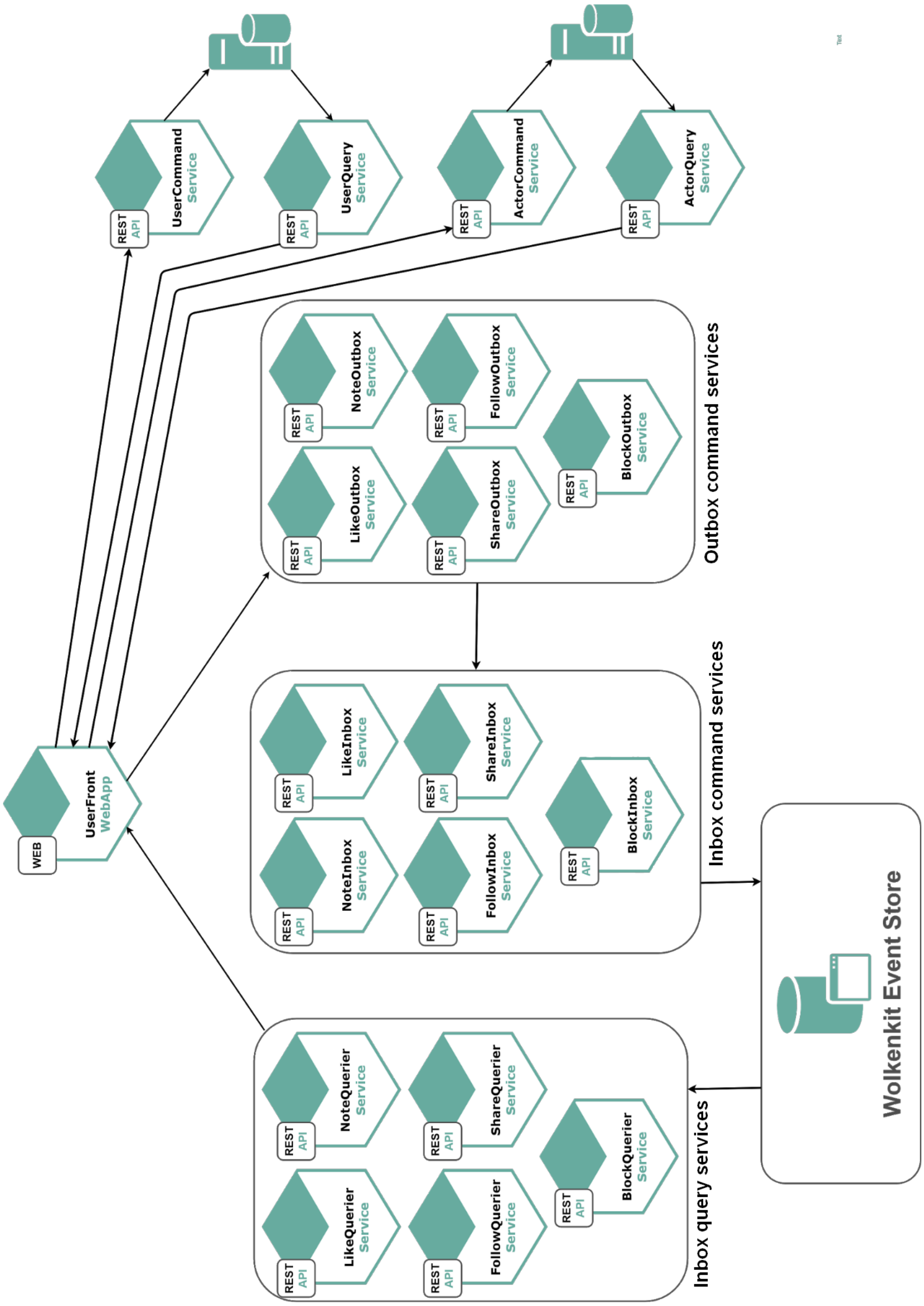


Figure 5.1: Architecture of the ActivityPub protocol implementation

Here is a summary on how communications, seen in Figure 4.1, work in this architecture if we take the example of Jean writing a note:

- The client (`UserFront`) makes a POST request, containing a note activity of type 'Create', to the `NoteOutbox` service.
- The `NoteOutbox` service forwards the request to the `NoteInbox` service that publishes the activity to the event-store.
- The `NoteQuerier` service, being subscribed to the note topic, is notified of the new 'Create' activity. On reception of the notification, it delivers the activity to each recipient's inbox (Jean's followers).
- To obtain the notes written by Jean, a GET request can be sent to the `NoteQuerier` service. This service will replay all the events related to Jean's notes. The set of inbox query services acts as a database querier module and can be used to retrieve the activities received by an actor (inbox) and the activities sent as well (outbox).

This design provides efficient extensibility and customizability by allowing for easy addition and removal of components. If we wish to add a new feature for uploading videos, we would simply have to design three new services: `VideoOutbox`, `VideoInbox`, `VideoQuerier` and a database for the videos. The only other service that would be modified is `UserFront` as it must know where to send and retrieve videos (front-end design must also be adapted). For the event-sourcing part, we simply have to specify a new type of event for services to subscribe to.

5.2 The APIs of services

This section details the APIs of all the services of the application. The ActivityStreams type means that the parameter must be a JSON object with properties corresponding to a valid ActivityStreams activity. In accordance with ActivityPub, all requests output the status code 201 on success and status code 500 on error.

5.2.1 User

UserCommand

Method	URN	Parameters	Description
POST	`/user/create`	username=string, password=string	Register a new user

UserQuery

Method	URN	Output	Description
GET	`/user/authenticate /:username/:password`	token=JsonWebToken	Log in an user
GET	`/user/authorization /:username`	-	Validate a JSON web token

5.2.2 Actor

ActorCommand

Method	URN	Parameters	Description
POST	`/actor/create`	user=string, type=string, id=string	Create an actor for user
POST	`/actor/update`	activity=ActivityStreams	Update an existing actor
POST	`/actor/delete`	activity=ActivityStreams	Delete an existing actor

ActorQuery

Method	URN	Output	Description
GET	`/actor/getAll /:user`	actors=Collection<ActivityStreams>	Get all of user's actors
GET	`/actor/get /:actor`	actor=ActivityStreams	Get the information of actor

5.2.3 Outbox

NoteOutbox

Method	URN	Parameters	Description
POST	`/note/create`	activity=ActivityStreams	Creation of a note
POST	`/note/update`	activity=ActivityStreams	Update of an existing note
POST	`/note/remove`	activity=ActivityStreams	Removal of an existing note

LikeOutbox

Method	URN	Parameters	Description
POST	`/like/create`	activity=ActivityStreams	Like of an object
POST	`/like/undo`	activity=ActivityStreams	Undo a previous like

FollowOutbox

Method	URN	Parameters	Description
POST	`/follow/create`	activity=ActivityStreams	Subscribe to the activities of another actor
POST	`/follow/undo`	activity=ActivityStreams	Undo a previous follow

ShareOutbox

Method	URN	Parameters	Description
POST	`/share/create`	activity=ActivityStreams	Share of an object of another actor
POST	`/share/undo`	activity=ActivityStreams	Undo a previous share

BlockOutbox

Method	URN	Parameters	Description
POST	`/block/create`	activity=ActivityStreams	Block an actor from some content
POST	`/block/undo`	activity=ActivityStreams	Undo a previous block

5.2.4 Inbox

NoteInbox

Method	URN	Parameters	Description
POST	`/noteIB/create`	activity=ActivityStreams	Creation of a note
POST	`/noteIB/update`	activity=ActivityStreams	Update of an existing note
POST	`/noteIB/remove`	activity=ActivityStreams	Removal of an existing note

LikeInbox

Method	URN	Parameters	Description
POST	`/likeIB/create`	activity=ActivityStreams	Like of an object
POST	`/likeIB/undo`	activity=ActivityStreams	Undo a previous like

FollowInbox

Method	URN	Parameters	Description
POST	`/followIB/create`	activity=ActivityStreams	Subscribe to the activities of another actor
POST	`/followIB/undo`	activity=ActivityStreams	Undo a previous follow
POST	`/followIB/accept`	activity=ActivityStreams	Accept a follow activity sent previously
POST	`/followIB/reject`	activity=ActivityStreams	Reject a follow activity sent previously

ShareInbox

Method	URN	Parameters	Description
POST	`/shareIB/create`	activity=ActivityStreams	Share of an object of another actor
POST	`/shareIB/undo`	activity=ActivityStreams	Undo a previous share

BlockInbox

Method	URN	Parameters	Description
POST	`/blockIB/create`	activity=ActivityStreams	Block an actor from some content
POST	`/blockIB/undo`	activity=ActivityStreams	Undo a previous block

5.2.5 DatabaseQuerier

NoteQuerier

Method	URN	Output	Description
GET	`/note/from /:actor`	notes=Collection<ActivityStreams>	Get all the notes posted by the actor
GET	`/note/to /:actor`	notes=Collection<ActivityStreams>	Get all the notes with the actor as receiver
GET	`/note/get /:object`	note=ActivityStreams	Get the current state of a note

LikeQuerier

Method	URN	Output	Description
GET	`/like/likedBy /:actor`	likes=Collection<ActivityStreams>	Get the list of objects liked by actor
GET	`/like/likes /:object`	actors=Collection<ActivityStreams>	Get the list of actors that have liked object
GET	`/like/get /:object`	like=ActivityStreams	Get the current state of a like

FollowQuerier

Method	URN	Output	Description
GET	`/follow/followed /:actor`	followers=Collection<ActivityStreams>	Get the list of followers of actor
GET	`/follow/following /:actor`	following=Collection<ActivityStreams>	Get the list of actors that actor follows
GET	`/follow/get /:object`	follow=ActivityStreams	Get the current state of a follow

ShareQuerier

Method	URN	Output	Description
GET	`/share/sharedBy /:actor`	shared=Collection<ActivityStreams>	Get the list of objects shared by actor
GET	`/share/sharedWith /:object`	actors=Collection<ActivityStreams>	Get the list of actors sharing object
GET	`/share/get /:object`	share=ActivityStreams	Get the current state of a share

BlockQuerier

Method	URN	Output	Description
GET	`/block/blocked /:actor`	blocked=Collection<ActivityStreams>	Get the actors blocked by actor
GET	`/block/blocking /:actor`	blocking=Collection<ActivityStreams>	Get the actors blocking actor
GET	`/block/get /:object`	block=ActivityStreams	Get the current state of a block

5.3 Internal workings

In this section, we will discuss the way services interact which each other and how their APIs are used. The API implementation of each services can be found under `serviceName/src/routes/topic.js` (e.g. `userQuery/src/routes/user.js`). The `couchdb_api` files contain the communication with the database, `inbox_api` files contain the communication with other services, and `event_handler` files contain the communication with the event-store.

5.3.1 Authentication

Authentication is handled by the `UserCommand` and `UserQuery` services. Whenever an individual wishes to use the application, he must first create an account by making a POST request to `/user` (`UserCommand`) with a name and password hash in parameters. The service will then insert it in the couchDB[1] database handling all the users' information. If the insertion is successful, a json web token (jsonwebtoken library) will be generated based on a secret key and added to the body of the response. When receiving the reply, the user will be able to store the bearer token and put it in the header of the following requests to prove its identity.

The `UserQuery` service provides a login feature (`/user/authenticate/:username /:password`), in charge of verifying the credentials of an individual and generate a token if they are valid. This service is also in charge of verifying the identity of users through their token (`/user/authorization/:username` with token in header). Authentication through this token is required for all the other POST requests of this application. Once a user is connected, he must choose an identity among the ones he has or create a new one through the `ActorCommand` service.

This service handles the creation, update and removal of actors in a couchDB database. The `ActorQuery` service is used to retrieve information on a specific actor (`/actor/get/:actorId`) or all the actors belonging to a user (`/actor/getAll/:user`). Figure 5.2 illustrates this process. A new user calls `UserCommand` to create an account, he receives his credentials in the response. Then he creates a new actor profile by calling `ActorCommand`, which verifies the identity of the user by calling `UserQuery` . Now that the user is connected and has created an actor profile, he is able to send and receive messages.

[1] https://github.com/apache/couchdb-nano

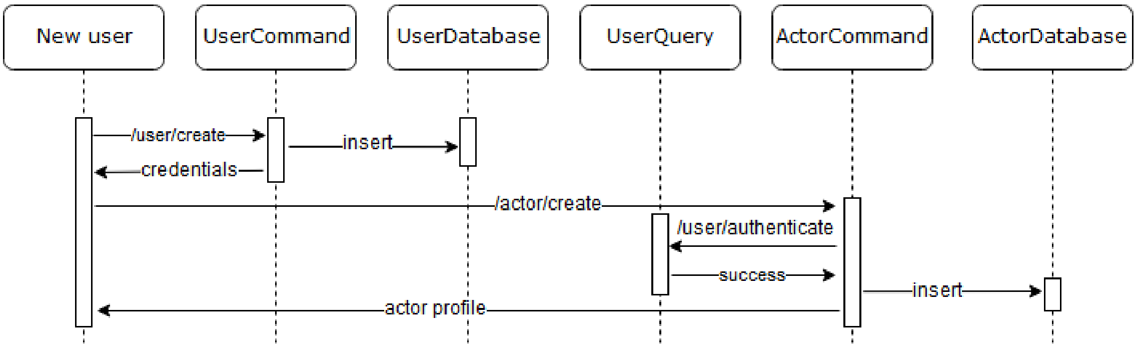


Figure 5.2: Diagram of the services call for the authentication

5.3.2 Communication

The first stage of sending messages is to put it in the sender's outbox. It is represented by the outbox command services seen in Figure 5.1. Instead of sending everything to the same URL, an activity is sent to the service in charge of the related interaction. For instance, if a user writes a note, the 'Create' activity must be sent to `NoteOutbox`. If someone else likes that note, the 'Like' activity must be sent to `LikeOutbox` as showed in Figure 5.3. The role of these services is only to forward the data to all subscribed servers and make appropriate changes such as embedding new objects into 'Create' activities.

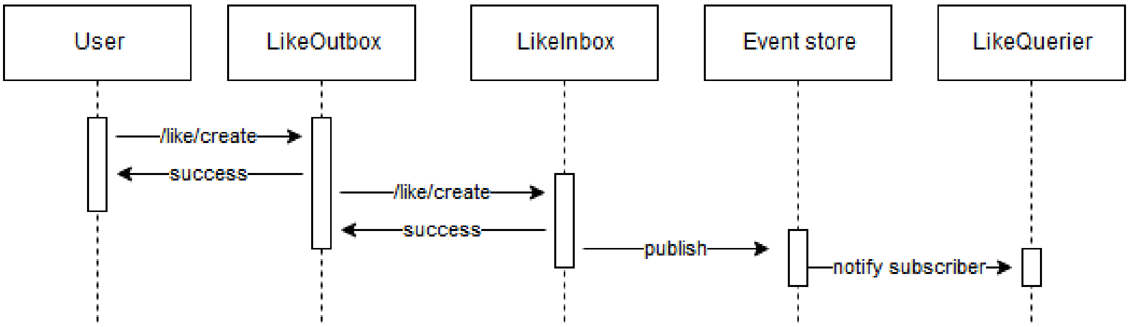


Figure 5.3: Diagram of the services call for the communication of a 'Like' activity

The inbox command services are in charge of receiving messages from the outbox services, as well as other interoperable platforms, and delivering them. The delivery to the actors' inbox is performed by publishing activities to the event-store. When launching the inbox query services, they subscribe to the topic related of they are interested in such that they can be directly notified with new incoming events. For instance, when the `LikeOutbox` service sends a 'Like' activity to its inbox counterpart, `LikeInbox` will publish it to the event-store.

`LikeQuerier` will be directly notified of the new activity and can then forward it to the appropriate service. In a fully-fledged social network, `LikeQuerier` could forward the 'LikeActivity' to the front-end as to increment the likes count. These read-only services can also be queried to retrieve the outbox of an actor (the gateway handles the aggregation of all services data) or the content of an actor's inbox (e.g. a GET request to `/note/from/:actor` will reply with all the messages written by `actor`). The communication was implemented by using different tools.

5.4 Technologies used

In the previous section, we explained how users can connect to the application and how they can exchange messages through their public identities. All of these interactions rely on external technologies.

5.4.1 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 an application with all of the parts it needs, such as libraries and other dependencies, and ship it as one package. The developer can ensure that the application will run on any platform. To handle the separations of concerns between services properly, Docker isolates them into their own address space, making them only reachable through their exposed API. This suits well microservices-based applications. Docker-compose is used here to group all the services, starting them all in only one command (`docker-compose up`). To reach multiple containers in one request, we use an API gateway.

5.4.2 Express Gateway

Express Gateway is an API gateway built on Express.js. Its configuration can be found in `services/gateway/config/gateway.config.yml`. It comes with a component that registers consumers (users and apps) of the hosted APIs. This component is a highly extensible identity and authorization system out of the box for APIs, used to validate the bearer token in each request before forwarding it to the services. Queries requiring data from multiple services are handled by the gateway as well. For instance, when sending an HTTP GET request to `/outbox/:actor`, the gateway will send requests to all inbox querying services (`/note/from/:actor`, `/like/liked/:actor`, etc...) and answer with the complete data set. After discussing how the authentication token is validated for each request and how multiple requests can be hidden behind a single URL, we will see how event-sourcing is implemented. It can be started through the following command:

```
cd MasterThesis/services/gateway && sudo npm start
```

5.4.3 Event-sourcing with Wolkenkit

The event-sourcing part of the application is used between the command and query inbox services, and leverages the Wolkenkit library. It is a CQRS and event-sourcing framework that includes an event-store and a scalable real-time API. The configuration of the event-store and events pub-sub can be found under `services/inbox/eventStore/server`. This folder contains two components that are used to handle events, the read and write models. In the write model, we define 'commands' functions used to publish each type of activity (post, like, share, follow and block).

The role of these 'commands' is to publish activities under specific topics (posted, liked, shared, followed and blocked). To publish a 'Like' activity, for instance, we call `eventStore.activityPub.activity().like(likeActivity)` where `eventStore` is the Wolkenkit client, `activityPub` the name of our application, `activity` the write model file and `like` the 'command' name. In the read model, we define 'projections', which are functions to execute when being notified of new events. Query services can subscribe to events by observing incoming activities and filtering by event type. Now that we know how services interact with each other we can install the application.

5.5 Installation guide

The first step is to retrieve the repository hosted on git:

```
$ git clone https://github.com/glovise15/MasterThesis.git
```

Once the project has been cloned, the following modules must be installed: `npm`[2], `node`[3] (10.x), `docker`[4], and `docker-compose`[5]. The next step is to install the different libraries through this command:

[2] https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-16-04

[3] https://joshtronic.com/2018/05/08/how-to-install-nodejs-10-on-ubuntu-1804-lts/

[4] https://docs.docker.com/install/linux/docker-ce/ubuntu/

[5] https://www.digitalocean.com/community/tutorials/how-to-install-docker-compose-on-ubuntu-16-04

```
$ cd MasterThesis/services && sudo ./npm-install.sh
```

The script will take some time to complete as it will install every *npm* dependency (e.g. artillery for load testing and jwt for authentication) for each service. Once the installation is done, the last step is to move to the `services` folder and start the application:

```
$ cd services && sudo docker-compose up
```

This will also take a long time as each docker image must be built. Once everything is running, the last step is to verify that the application behaves as expected.

5.6 Testing

The behavior of each service is verified using unit tests (`mocha`[6] test framework with `chai`[7] assertion library) to see if some problems were overlooked. We check the interactions between services by running scenarios spanning multiple services (`Artillery`[8] toolkit with `expect` plugin[9]). We then verify that our implementation can handle a large amount of requests by performing load testing (`Artillery`).

5.6.1 Scenario-based testing

Our application is composed of scenarios whose role is to check if services, combined as a group, are working properly. We start by testing the authentication sequence of requests seen previously in Figure 5.2. With this first scenario, we verify that individuals are able to log-in and create new accounts. We then perform a test scenario for each set of services. For example, to test all the services related to notes, we send different 'note' activities to the `NoteOutbox` and verify the output by calling the `NoteQuerier` service. If the output is not what we expect, we know that there has been an issue with at least one of the components. Scenarios can be run by moving to the `MasterThesis/scenarios` folder and executing the following command:

```
$ artillery run fileName.yml
```

Identifying issues for each service specifically is done through unit testing.

[6] https://mochajs.org/

[7] https://www.chaijs.com/

[8] https://artillery.io/

[9] https://www.npmjs.com/package/artillery-plugin-expect

5.6.2 Unit testing

Unit testing is done in the same fashion for each service. The only difference lies in whether we are testing a command or query service. In the first case, we send POST requests with incorrect activities (empty payload, missing fields, incorrect properties, wrong values). In doing so, we can verify that incorrect requests are rejected by the service and potential problems are minimized. When dealing with query services, APIs only allow for one or two parameters and as such, there is not much to test. The GET requests are verified through scenarios testing as they require for the database to be populated (command services). Unit tests can be run by moving to the `tests` folder of any command service and running the following line:

```
$ mocha tests.js
```

Once we know that everything is working as it should, we can test how the services handle the load.

5.6.3 Load testing

The scalability of services, and how they handle large amounts of data, plays an important part in this project. We performed several load tests to identify the threshold at which requests can no longer be processed on time. Each test consists in sending *n* requests for two minutes (*n* starts at 10 and is increased until the threshold has been found). We record the average delay of every test and compare it to the numbers of requests served (throughput). Load tests can be run by moving to the `MasterThesis/loadTests` folder and using the following command:

```
$ artillery run fileName.yml
```

Account creation and authentication

The authentication scenario calls the `UserCommand` (`/user/create`), `UserQuery` (`/user/authenticate` and `/user/authorize`), `ActorCommand` (`/actor/create`), and `ActorQuery` (`/actor/get`) services. The plot in Figure 5.4 shows how the average delay increases according to the throughput. As these requests must be done sequentially (previous requests information required), they must wait some time before being sent. This set of services starts being throttled when sending 70 requests per second (RPS). Above that amount, some requests are not being served as the delay is too important. With sending 80 RPS, only 76 requests are served, and when sending 90 RPS, only 73 requests are served. This means that above 80 RPS, the number of requests served will keep diminishing. We consider this result good, as mega-platforms such as Facebook only register in average 6 new accounts every second[10] (stable up to 15 here). The next test will analyze the average delay when sending multiple asynchronous requests.

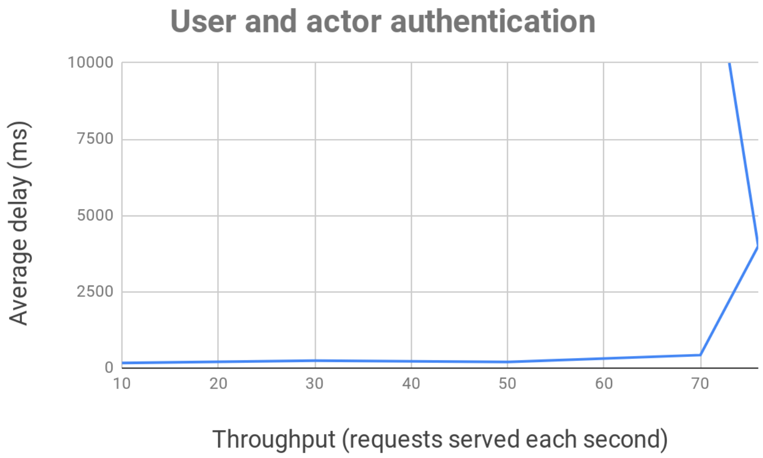


Figure 5.4: Average delay compared to the throughput when users create an account and authenticate

Creation and update of activities

In this test, the scenario consists of sending requests for creating a new note (`/note/create`) and updating it 4 times (`/note/update`). Update requests being asynchronous, we must not wait for their result. The requests are first sent to the `NoteOutbox` service which forwards them to `NoteInbox` service, responsible for storing them in the event-store. The `NoteQuerier` service will be notified each time a new note event is published. We can observe in Figure 5.5, the average delay for 125 requests sent each second being 671 ms, indicating that the threshold has been met. If we test further, we start dropping messages at 130 RP, the delay being around 2100 ms and quickly increasing. The spike can be explained by the event-store implementation not able to process events as quickly and thus creating a bottleneck where requests take several minutes to be processed. The results are satisfying if we take into account that one million users are equal to 5 posts each second (Facebook registers 8500 posts/second for a total of 2.3 billion users [11]). In the following test, we will check the result when only performing read operations.

[10] https://www.socialmediatoday.com/social-networks/kadie-regan/2015-08-10/10-amazing-social-media-growth-stats-2015

[11] https://zephoria.com/top-15-valuable-facebook-statistics/

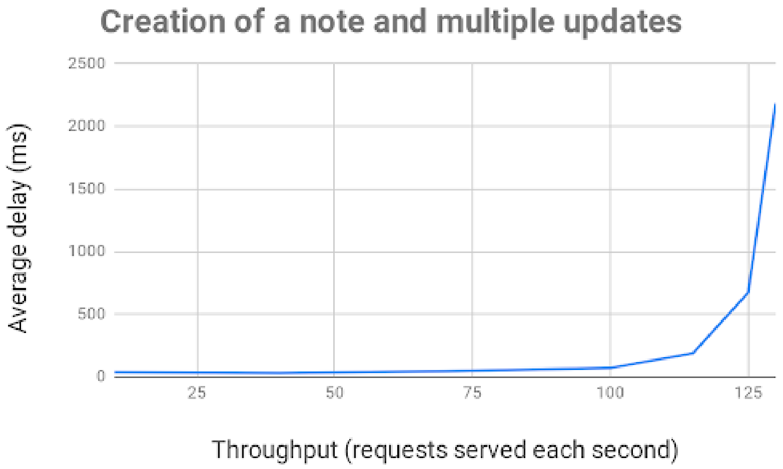


Figure 5.5: Average delay compared to the throughput when users create and update notes

Retrieval of activities

While in previous scenarios we tested multiple services, here we will only send a read request to a unique service. We will call the `NoteQuerier` service (`/note/from`) to fetch all the notes previously created and updated (total of 44 create and 176 update activities). What we can see in Figure 5.6 is that we encounter the threshold at 190 RPS (755 ms), which is much later than in the previous test. This can be explained by the lack of writes requests and event processing through the event-store (only reading and filtering). Reads being much more prevalent that writes operations in social networks, this result is satisfying. It is important to note that through the replaying of events, the processing delay will only increase as new events are stored such that with very large data sets, the threshold will be met sooner. We must also take into account the filtering policy of Wolkenkit, forcing the `NoteQuerier` service to iterate over every event to verify its activity type. The last step is to test what happens when different services are queried simultaneously.

Mix of services

Requests have been sent to different services as to simulate how the application would operate in practice: `UserCommand` (`/user/create`), `UserQuery` (`/user/ /authenticate`, `/user/authorization`), `ActorCommand` (`/actor/create`), `Actor Query` (`/actor/get`), `NoteOutbox` (`/note/create`, `/note/update`), `LikeOutbox` (`/like/create`), `NoteQuerier` (`/note/from`, `/note/to`, `/note/get`). The plot present in Figure 5.7 shows that the threshold is reached at 110 RPS, for an average

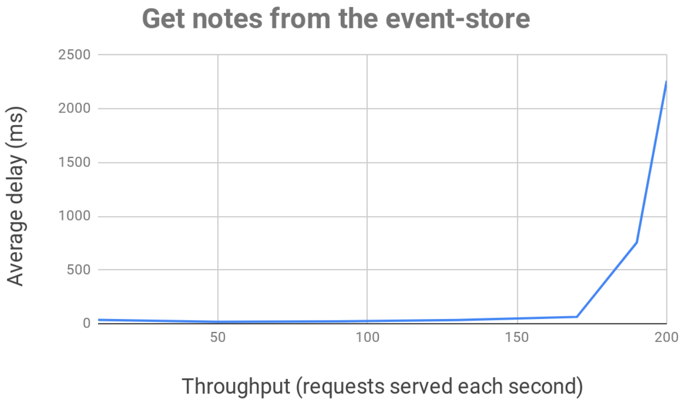


Figure 5.6: Average delay compared to the throughput when retrieving the user's notes

delay of 439 ms. Trying to sustain a higher number of RPS will lead to inefficient delay and packet loss. We have an even mix between read (60%) and write (40%) requests, yet, the threshold is reached quicker than doing only read or only write operations. This can be explained by the sequential order needed between the creation of a user, actor, and note (the rest is asynchronous). This sequence is required as they all need information from the previous response. By being able to handle around 100 RPS consistently, this application has some margin before having to consider increasing its resources and storage. The implementation of this project and its tests illustrated some obstacles.

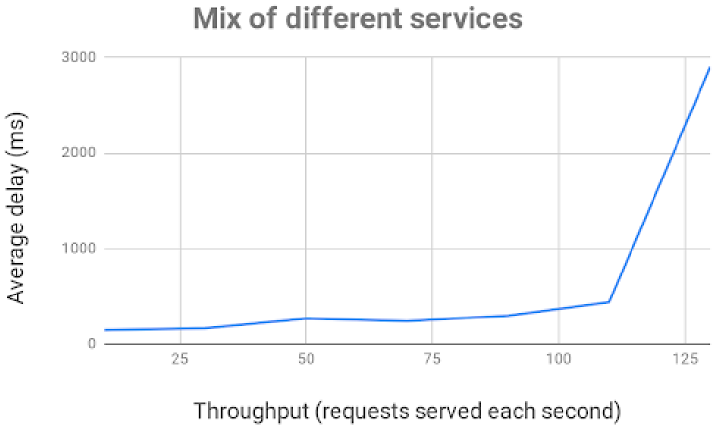


Figure 5.7: Average delay compared to the throughput when calling different services

5.7 Limitations

The implementation of the ActivityPub protocol through microservices, event-sourcing, and CQRS, faces some limitations:

1. CQRS is problematic when performing unit tests. Services being divided into read and write components means that query services cannot be tested properly (the data retrieved must first be inserted by the command service). With microservices, each component must be independent from others and as such we should not have to call a second API to test a service. To test the `NoteQuerier` service, the event-store must contain Create/Update/Remove note activities. This would require calling the `NoteOutbox` or `NoteInbox` service to populate the event-store. Our solution is to validate query services through scenario based testing.

2. Some properties of activities (id, outbox, inbox, etc...) have for value a dereferenceable URI that allows actors and servers to retrieve various information. For instance, a server will send a POST request to an actor's inbox URI to deliver a message and the actor can send a GET request to that same URI to retrieve all of its messages. Once again, the CQRS pattern means that we have two different services for read (GET) and write (POST) operations, meaning that we have two different addresses. Microservices make the problem even worse as there is a service for each interaction. The outbox and inbox are distributed over several URIs yet there is only one inbox/outbox field in an actor's profile. The solution implemented is to transfer requests to hard-coded destinations based on the type of activity and to handle global outbox and inbox requests through the gateway. This solution is not optimal as it may cause some interoperability problems (e.g. a POST request sent to `/inbox/:actor` will be rejected) and impact the extensibility (the gateway must be adapted to each change).

3. ActivityPub states that if an activity is submitted with a value in the id property, servers must ignore this and generate a new id for the Activity. In short, this tells the server to ignore client-generated IDs which function as the locator for resources. Idempotence means that operations will produce the same results if produced once or multiple times. Through idempotence, the server can establish whether the operation previously succeeded, or if there is no trace of it, in which case it must be carried out from the start. Breaking this rule introduces duplication of data as activity coming from the client cannot be accurately identified (the client and server use different unique identifiers for the same activity). We chose to ignore this part of the W3C Recommendation.

4. One of the critical parts of event-sourcing, time-wise, is the replaying of events required to obtain the current state of objects. While not important at first, the time required to build the proper version of an object will increase with each modification. If we wish to retrieve the current state of some note that has been modified 10 times, liked 45 times and shared 24 times, there are at least 79 events to replay. When applied to the full outbox of a user, there might be hundreds of objects with as many events for each of them.

These limitations are problematic but future work can be done to fix them.

5.8 Future work

The purpose of a foundation is to be built upon and so the implementation of this framework for vertical DOSNs was designed with future work in mind. Now that the application has been detailed, we can discuss the possible improvements:

- With interoperability comes communication with many other platforms bringing their own specificities with them. At this point in time, our application does not test activity exchange with other networks, leading to potential issues (activities can have many different formats that could generate failures). Integration testing, being really complex and requiring multiple teams to perform properly, is also not done. The robustness of this application should be improved through additional real-world testing.

- Security was not a goal of this project and as such, it is very light. The only mechanism in place is the authentication through a json web token and its validation during each POST request. Social media are known for containing sensitive data. Developers building their social networks on this foundation should enhance the security to fit their requirements.

- Sharding is a data management technique consisting of taking advantage of decentralization by dividing a data store into smaller sets and hosting them in different locations. Sharding is crucial in social networks where the growth of data is known to be very large and should be considered when deploying real-world solutions.

- The replaying of events could be optimized by having a script running in the background. It would collapse all the events, for each object, into only a few and store them in another database that could be queried to get the last state. This would improve the response time while still having all the events available in the event-store. We could also have an event-store for each service.

Chapter 6

The future of DOSNs

In the span of a decade, we have seen the birth and rise of social networks as well as their declination into decentralized and open platforms. In the rapidly growing domain that is the web, new solutions, tools, and technologies are constantly released. There are many reasons for designing new tools and platforms and, often, it is to improve already existing solutions. In this section, we will discuss the shortcomings of openness/decentralization and discuss emerging technologies and how they may impact the design and implementation of social networks.

6.1 The challenges to come

As experience shows us, there is a clear difference between theory and practice. No matter how promising DOSNs are on paper, beating the monopolistic networks, represented by mega-platforms such as Facebook and Twitter, is a colossal endeavor in practice. There are several key reasons as to why openness and decentralization might not be enough to address the threats to free expression[1]:

1. Attracting the attention of the public and developers is a complex task, individuals seldom join platforms because of ideological reasons. The willingness to join a social network is much more based on the presence of friends than the quality of the platform. Even though DOSNs in the same federation are interoperable, users must still join one of the networks of that federation. This represents a big step for many individuals as it means losing the content posted previously as well as leaving some relationships behind because the network may be the only communication link between two people. Joining something new does not always mean leaving something else but if both

[1] https://www.wired.com/story/decentralized-social-networks-sound-great-too-bad-theyll-never-work/

products have the same features, there are not many reasons to keep using both solutions.

2. Monopolistic platforms act as publishers as well as curators. For instance, Facebook controls not only what is acceptable to publish but what messages we see, bringing the most interesting posts to one's attention and removing the undesired content[2]. Designing robust reward mechanisms to curate content, that keeps people informed rather than entertained, remains a problem. DOSNs much like mega-platform still have issues such as echo chambers and filter bubbles. Even if some platforms chose to not curate content, the problems are still there. Furthermore, advertisers rely on content curation to reach as many users as possible. Choosing to do without advertising requires an alternative revenue model to maintain the platform.

3. Social networks are known for being resource-intensive, thus they benefit a lot from economies of scale. Resources like storage and bandwidth are much cheaper when bought in bulk. This leads to a natural consolidation toward super-participants represented by platforms with large user bases. DOSNs are impacted as much by economies of scale as any other social network. They are driven to centralize their data and become larger, which goes against the principle of decentralization.

An optimal strategy would be to pursue policies that strengthen the environment for decentralized platforms such as data portability and interoperability. If users have more control of their data, like the ability to export and reuse content they have created, they'll be more willing to experiment with new platforms. These issues highlight the work that still needs to be done to make open and decentralized social networks the default solution.

6.2 Openness and decentralization with blockchains

With data privacy scandals like Cambridge Analytica, users have become wary of their information being mishandled and sold to the highest bidder. One of the possible responses to that concern is the blockchain. It is a safe, transparent and distributed database, storing contracts between users (e.g. monetary transaction). While the implementation is not exempt from challenges, its use is starting to spread to many different sectors including social networks. By using blockchains, developers can ensure that interactions and transactions are completely transparent (public) and decentralized (peer-to-peer) while still being trustworthy (encryption

[2] https://www.getresponse.com/blog/what-is-content-curation-and-why-do-you-need-it

and peer-to-peer validation). The most well-known example is Streemit, a microblogging social network similar to Reddit and Facebook. Instead of a centralized curation of content to please advertisers, users become curators by getting rewarded for their contributions (e.g. posting an article) and activities (e.g. liking an article) on the platform. The curation reward system is fully integrated in their blockchain. Every transaction and all communications go through it, making them completely transparent. This also allows users to retain full ownership of their content forever and no one is able to delete it. Furthermore, the technology is also suited to resolve the data portability problem where users cannot easily migrate their profile between two platforms. By storing their information in a platform-agnostic blockchain, users would be able to retrieve their data anywhere. This solution is currently being explored as a master thesis under the supervision of Prof. Etienne Riviere at UCLouvain.

Chapter 7

Conclusion

This thesis is part of a global movement to improve open and decentralized social networks, by identifying the shortcomings of existing monolithic platforms and providing the base framework for better extensible networks. The objective was to investigate the use of recent advances in software engineering for building a framework for extensible DOSNs, with a focus on decentralization and interoperability with the Fediverse. The ActivityPub implementation through the combinations of microservices, event-sourcing, and CQRS, improves existing solutions. Core issues around inefficient maintainability, scalability, extensibility, and customization are corrected by these new tools. While microservices and the CQRS pattern allow for optimal components isolation, event-sourcing is highly suited for social networks where interactions are similar to publish/subscribe messaging.

These technologies, put together, allow for efficient communication, wide-scale personalization and easy extensibility. However, from this set of technology, limitations emerge such as the presence of inboxes for each kind of interaction, not being compatible with the ActivityStreams format, as well as the time required to replay every event. Some work is still required for DOSNs to become tomorrow's trending social media because of the complex tasks ahead such as attracting the attention of the public and finding an appropriate revenue model. The real-world must also be taken into account, economies of scales trend toward centralization to reduce the cost of resources and data transparency, while being needed for democracy, present a treat to privacy. Future avenues of improvement could leverage blockchains to provide safe data portability and to allow users to keep ownership of their information.

References

Chris, R. (2018). *Pattern: Microservice architecture.* Retrieved 14-10-2019, from https://microservices.io/patterns/microservices.html

Chris, R., & Floyd, S. (2016). *Microservices: From design to deployment* (1st ed.). NGINX.

Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., ... Jackson, B. (2019). An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems* (pp. 3–18).

Hugo, R. (2018). *What they don't tell you about event sourcing.* Retrieved 07-03-2019, from https://medium.com/@hugo.oliveira.rocha/what-they-dont-tell-you-about-event-sourcing-6afc23c69e9a

Martin, F. (2005). *Event sourcing.* Retrieved 24-02-2019, from https://martinfowler.com/eaaDev/EventSourcing.html

Martin, F. (2011). *Cqrs.* Retrieved 24-02-2019, from https://martinfowler.com/bliki/CQRS.html

Teiva, H. (2018). *1 year of event sourcing and cqrs.* Retrieved 22-02-2019, from https://hackernoon.com/1-year-of-event-sourcing-and-cqrs-fb9033ccd1c6

Vernon, V. (2013). *Implementing domain-driven design* (1st ed.). Addison-Wesley Professional.

Wasson, Bennage, Buck, Wood, Hughes, Palamakumbura, ... Wilson (2019). *Command and query responsibility segregation (cqrs) pattern.* Retrieved 23-05-2019, from https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs

Wenzel, Nish, Schonning, & Lee. (2018). *Designing a microservice-oriented application.* Retrieved 14-10-2019, from https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/multi-container-microservice-net-applications/microservice-application-design

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