CATHOLIC UNIVERSITY OF LOUVAIN-LA-NEUVE

Master thesis - 2018/2019

A decentralized social network based on event-sourcing principles

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

March 1, 2019

1 Introduction

Open and decentralized online social networks (OSN), such as Mastodon, Diaspora and GNUSocial are alternatives to monopolistic social networks such as Twitter or Facebook. Instead of collecting and storing social user data (friends, posts, comments, likes, etc.) in a single datacenter, these social networks operate as a federation of independent sites, called the Fediverse (universe of federations). Users connected to one website can interact with users from other members of the federation. With the recent scandal of Cambridge Analytica and Facebook, the interest of using decentralized OSN becomes very clear. Open and decentralized OSNs are all open-source and target interoperability. 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 members of the Fediverse.

All OSNs participating to the Fediverse are built as rigid monolithic applications often using a single relational database. This design leads to poor flexibility of purpose: it is difficult to adapt these OSNs, for instance to support new forms of social interactions. It is also difficult to isolate in these platforms the components that are critical for security, and in particular the ones that manipulate sensitive user data. A bug or exploit in any part of the monolith can allow a malicious attacker to read the entire database, and bugs are difficult to detect due to the size of the software. This work aims to evaluate the feasibility and performance of a modular and decentralized implementation of the ActivityPub protocol, to improve these networks.

2 The role of micro-services

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

2.1 Separation of concerns

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

2.2 Scalability

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

3 Architecture design

3.1 ActivityPub

Activity Pub is an open, decentralized social networking protocol providing a client/server API for creating, updating and deleting content, as well as a federated server-to-server API for delivering notifications and content. This protocol provides two layers:

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

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

In ActivityPub, a user is represented by "actors" via the user's accounts on servers. User's accounts on different servers correspond to different actors. Every actor has an inbox to receive messages and an outbox to send messages to others. An activity is delivered to the inboxes of all the actors mentionned in the to, bto, cc, bcc and audience fields of the activity. We must also always deliver an activity to the sender's follower collection (de-duplication needed). 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. An activity in an outbox is always delivered to the appropriate inboxes but also added in the sender's outbox collection (through the OutboxCollection API).

3.2 Patterns

3.2.1 CQRS

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

3.2.2 Event-sourcing

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

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

4 Implementation

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

4.2 ActivityStreams

4.3 Authentication

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

4.4 Event-sourcing - Wolkenkit

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

4.5 Testing

4.5.1 Unit testing

4.5.2 Integration testing

4.5.3 Load balance testing

5 Conclusion