Integrating DIDComm Messaging to ActivityPub-based Social Networks

by

Adrián Isaías Sánchez Figueroa

Matriculation Number 397327

A thesis submitted to

Technische Universität Berlin
School IV - Electrical Engineering and Computer Science
Department of Telecommunication Systems
Service-centric Networking

Master's Thesis

August 26, 2022

Supervised by:
Prof. Dr. Axel Küpper

Assistant supervisor:
Dr. Sebastian Göndör

Eidestattliche Erklärung / Statutory Declaration

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed.

Berlin, August 26, 2022 Chuck Norris' son

Abstract

The centralized architecture that characterizes the current social Web led to the development of Decentralized Online Social Networks (DOSN). These DOSN rely on their communication protocols to interact with each other, allowing the creation of a boundaryless ecosystem where users can move seamlessly between applications without rebuilding their network of friends and profiles at each destination.

ActivityPub is a standardized decentralized social networking protocol implemented by a large number of DOSNs. However, this standard has failed to define a way for its communication to be secure, confidential, private, and non-repudiable. Furthermore, its implementations fall short of a decentralized way to manage identities.

Web 3.0 brings decentralization to the current mostly siloed Web architecture, and it does it through new technologies that enable new standards that have the potential to change the way the Web is used. The recently standardized Decentralized Identifiers (DIDs) from the W3C and the DID-based communication protocol DIDComm Messaging v2 are two specifications that promise to decentralize how identities are handled, and how communication security is accomplished.

This work proposes to address the limitations of ActivityPub-based social networks by integrating DIDs to allow decentralized identity management, and by enabling DIDComm to provide a decentralized way for secure communication.

Zusammenfassung

Die zentralisierte Architektur, die das derzeitige soziale Web kennzeichnet, führte zur Entwicklung dezentraler sozialer Online-Netzwerke (DOSN). Diese DOSN stützen sich auf ihre Kommunikationsprotokolle, um miteinander interagieren zu können, was die Schaffung eines grenzenlosen Ökosystems ermöglicht, in dem sich die Benutzer nahtlos zwischen Anwendungen bewegen können, ohne ihr Netzwerk von Freunden und Profilen an jedem Zielort neu aufbauen zu müssen.

ActivityPub ist ein standardisiertes dezentralisiertes Protokoll für soziale Netzwerke, das von einer großen Anzahl von DOSNs implementiert wird. Dieser Standard hat es jedoch versäumt, einen Weg zu definieren, wie die Kommunikation sicher, vertraulich, privat und nicht widerlegbar sein kann. Darüber hinaus bieten seine Implementierungen keine dezentralisierte Möglichkeit zur Verwaltung von Identitäten.

Das Web 3.0 bringt Dezentralisierung in die derzeitige, größtenteils siloartige Web-Architektur, und zwar durch neue Technologien, die neue Standards ermöglichen, die das Potenzial haben, die Art und Weise, wie das Web genutzt wird, zu verändern. Die kürzlich standardisierten *Decentralized Identifiers* (DIDs) des W3C und das DID-basierte Kommunikationsprotokoll *DIDComm Messaging v2* sind zwei Spezifikationen, die eine Dezentralisierung der Identitätsverwaltung und der Kommunikationssicherheit versprechen.

Diese Arbeit schlägt vor, die Einschränkungen von ActivityPub-basierten sozialen Netzwerken zu beheben, indem DIDs integriert werden, um ein dezentralisiertes Identitätsmanagement zu ermöglichen, und indem DIDComm einen dezentralen Weg für sichere Kommunikation bietet.

Contents

1 Introduction

1.1 Motivation

The Internet today is essential to our society. There is an online supply for almost every thinkable service. Activities like reading the news, booking tickets, doing sports, listening to music, calling, exploring, working, connecting, and shopping are now possible through online services. However, since the Internet was not conceptualized to identify people, all these services must find a way to do it [1]. This workaround can be referred to as *identity one-offs*. This means that users must create a different identity for each service they want to consume, resulting in hundreds of identities, each existing only in a single organizational context. Identity one-offs prove the lack of consolidated digital identities [2].

From the service provider's perspective, storing user data has found no better solution than an internal database, or put in other words, a silo of personal user information. For example, millions of users are kept on a centralized platform in today's most popular Online Social Networks (OSN) like Facebook, Twitter, or Youtube. Here, control, decision-making, user data, and censorship depend on a single profit-driven organization. As data became the *world's most valuable resource* [3], these silos became a target. Facebook's latest data breach made the private data of around 530 million users public [4]. Unfortunately, not even a *safe* password can protect a user's data because it is not under his control.

The stage that follows the centralized digital identity is the federated digital identity, where the *Sign in with*-pattern provided trusts relationships between service providers, allowing linking of identities among identity management services [5]. However, having a centrally federated identity represented even a more considerable risk, and users still had no control over their data. Finally, as a solution, the Self-Sovereign Identity (SSI) stage brings full user autonomy, putting him in the center of the identity administration [6].

SSI was before not possible because of a gap in technology, that could provide the infrastructure required for decentralized trust. Nonetheless, the emerging Distributed Ledger Technology (DLT) opened the door for a decentralized, verifiable identity foundation. The newly standardized Decentralized Identifiers (DIDs) from the W3C bring an approach to enable SSI. Furthermore, the DID-based communication protocol, DIDComm Messaging v2, promises high-trust, self-sovereign, and transport-agnostic interaction, following the same idea of further bringing decentralization to day-to-day Web usage.

Concerning OSNs, different approaches to bringing decentralization have been proposed and standardized. In 2018, the Decentralized Social Networking Protocol (DSNP) ActivityPub became a W3C recommendation [7], since then, many Decentralized Online Social Networks (DOSN) have implemented it. The largest ActivityPub implementer is the Twitter-like

microblogging DOSN Mastodon[1], which has around three million registered users scattered through different independent servers. Each server sets its own rules, policies, and topics like LGBT+, Art, or Music. In addition, ActivityPub allows Mastodon to interact and communicate with other DOSNs that implement ActivityPub. An unthinkable situation in a centralized architecture.

1.2 Problem statement

ActivityPub has proved to be a mature protocol that restores some decentralization in the current Social Web paradigm [8]. However, it still presents significant deficiencies. Firstly, it only brings a decentralization level similar to the email paradigm, where a few large providers control the space of a federated network [8]. Mastodon, for example, uses centralized identity management, where an account creation uses basic password authentication. A federated way to create an account using a third party like *Google* or *Facebook* is not in its scope. Further attempts to improve identity verification have been made, such as using *Keybase*[2] to prove ownership of accounts cryptographically. However, this functionality was removed soon after Zoom[3] bought Keybase in 2020 [9].

Secondly, ActivityPub has no security mechanisms defined. Crucial requirements like non-repudiation, message integrity, and confidentiality have not been included in the protocol specification [10]. So far, the only present security feature is the recommendation to use HTTPS, which relies on centralized certificate authorities. In Mastodon's implementation, the administrators of the servers have access to all the user's information, including private messages. Imposing a privacy risk for its users.

1.3 Research Questions

To address the deficiencies and improvement opportunities in the ActivityPub-based social network Mastodon, the following research questions have been identified:

- Can DIDs bring self-sovereignty to ActivityPub-based social networks?
- What are the implications of introducing DIDs to Mastodon and ActivityPub in terms of usability?
- Can a DID-based ActivityPub protocol use DIDComm for its communication?
- Can DIDComm provide a fully-decentralized and secure communication to Mastodon?

[1] https://joinmastodon.org
[2] https://keybase.io
[3] https://zoom.us

1.4 Contribution

- **SSI in Federation:** This work proposes the use of DIDs in ActivityPub to provide a framework to give users full control of their identities. Through this, all the DOSNs implementing ActivityPub can take advantage of it.
- **Decentralized Secure Communication:** This work proposes DIDComm Messaging v2 to address the shortcomings of ActivityPub in terms of communication security, in addition to bringing a fully decentralized architecture by removing the DNS.
- **Standard Adoption:** The biggest challenge for a new Web standard is that it requires adoption by a large number of people. Recently standardized DIDs are already being adopted by governments, retailers, institutions, and universities [11]. This work will extend this adoption into the social space. In addition, it intends to demonstrate the significance and potential of the DIDComm family in the decentralized communication field.

.

1.5 Outline

The thesis is structured as follows. First, chapter 2 gives a detailed overview of the core technologies and standards involved in the development of this thesis, as well as related approaches. Next, chapter 3 presents a state-of-the-art approach, followed by the modifications required to integrate the Decentralized Identifiers into ActivityPub and the requirements to enable DIDComm Messaging v2. The implementation of the proposed design is detailed in chapter 4, followed by its the evaluation in chapter 5. Finally, chapter 6 concludes the thesis.

2 Related Work

The following chapter covers the most significant concepts required to comprehend this thesis's approach. It includes a closer look at decentralized communication protocols, identifier standards, and social networks that implement them. The revision and structuring of these concepts allow us to understand, build upon, and apply them to address our identified research questions.

2.1 Social Web Protocols

Between 2014 and 2018, the Social Web Working Group (SocialWG) from the W3C embarked on the journey to bring social-networking standards to the Web. This journey included defining technical protocols, vocabularies, and APIs focusing on social interactions. Following the idea that systems implementing these features should be able to communicate with each other in a decentralized manner. These four years resulted in several W3C Recommendations, including a collection of standards that enable various aspects of decentralized social web interaction called *Social Web Protocols* [12]. Standards found in this collection are *WebSub*[1], *WebMention*[2], *Linked Data Notifications*[3], and the two most relevant for this thesis, *ActivityStreams 2.0*[4] and *ActivityPub*[5].

2.1.1 ActivityStreams 2.0

ActivityStreams 2.0 is a standard that provides a model for representing *Activities* using a JSON-based syntax. Additionally, it provides a vocabulary that includes all the standard terms needed to represent social activities [13]. This standard describes an activity following a story of an *actor* performing an *action* on an *object*. It specifies different types of actors, activities, and objects, as shown in Table 2.1. Each of these objects can be represented as a JSON object, creating a solid foundation upon which other protocols can build.

[1] https://www.w3.org/TR/websub/
[2] https://www.w3.org/TR/webmention/
[3] https://www.w3.org/TR/ldn/
[4] https://www.w3.org/TR/activitystreams-core/
[5] https://www.w3.org/TR/activitypub/

ActivityStreams Vocabulary		
Activity types	**Actor types**	**Object types**
Accept, Add	Application	Note
Announce, Arrive	Group	Document
Block, Create	Organization	Image
Delete, Dislike	Person	Article
Flag, Follow	Service	Profile
Ignore, Invite		Audio
Join, Leave		Event
Like, Listen		Tombstone

Table 2.1: ActivityStreams 2.0 vocabulary examples

ActivityStreams 2.0 has improved its 1.0 version in more than one aspect. One of these is the compatibility with JSON-LD[6], which is a JSON serialization for *Linked Data*[7]. The concept of Linked Data is based on interlinking data in such a way that it becomes more usable through associative and contextual queries [14].Ẇith JSON-LD, ActivityStreams 2.0 can define its own context and the terms that will be used inside this context. Listing 2.1 shows an example of a JSON-LD serialized ActivityStreams 2.0 activity.

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "summary": "Alice created an image",
  "type": "Create",
  "actor": "http://www.test.example/Alice",
  "object": "http://example.org/foo.jpg"
}
```

Listing 2.1: Example of a Create activity using JSON-LD [13]

2.1.2 ActivityPub

ActivityPub is another W3C Recommendation that originated from the SocialWG. It is a decentralized social networking protocol that is based on the syntax and vocabulary of ActivityStreams 2.0. It provides a client-to-server API, which covers the requirements of a Social API[15], i.e., publishing, subscribing, reading content, and notifying when content gets created. In addition, it provides a server-to-server API that enables federated communication. Furthermore, it provides users with a JSON-based *profile*, which is an ActivityStreams 2.0 actor object. This actor object includes standard properties such as *name*, *type*, and *summary*. ActivityPub extended

[6] https://www.w3.org/TR/json-ld/
[7] https://www.w3.org/DesignIssues/LinkedData.html

this actor object with several properties. Extended optional properties include collections such as *following*, *followers*, and *liked*. Compulsory properties include an *inbox* and an *outbox*. These last two URLs represent how the actor gets and sends messages from other users. Listing 2.2 shows an example of an ActivityPub object with the extended properties.

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Person",
  "id": "https://social.example/alice/",
  "name": "Alice P.",
  "preferredUsername": "alice",
  "summary": "TU Berlin student",
  "inbox": "https://social.example/alice/inbox/",
  "outbox": "https://social.example/alice/outbox/",
  "followers": "https://social.example/alice/followers/",
  "following": "https://social.example/alice/following/",
  "liked": "https://social.example/alice/liked/"
}
```

Listing 2.2: Actor object example in ActivityPub [7]

There are two workflows of communication for a user in ActivityPub, as shown in figure 2.1

- **Client-to-Server Communication:** A user wants to share a post publicly. This requires an HTTP POST request to his outbox with the respective activity object. After this, other users interested in seeing this user's post can make an HTTP GET request to the user's outbox and retrieve it.
- **Server-to-Server Communication (Federation)**: User *A* wants to send a post to user *B*, whose account is on a different server. For this scenario, the following steps are required. First, user *A* posts his message to his outbox. Consequently, his server looks for *B*'s inbox and performs an HTTP POST request. Finally, *B* makes an HTTP GET request to his inbox to retrieve all the posts addressed to him.

  A key thing to remember is that for this type of communication, *A*'s server has to retrieve somehow the *inbox* of user *B* based only on his username. This resolving process is not part of the ActivityPub specification. Therefore, implementers of this standard must find a way to achieve this independently.

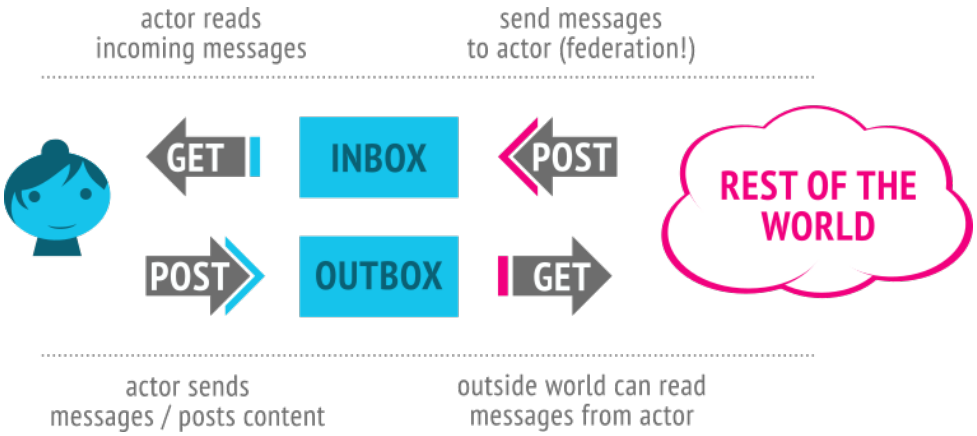

Figure 2.1: ActivityPub overview [7]

Regarding security, although ActivityPub's specification does not define any official security mechanisms, it mentions a list published by the SocialWG of best security practices[8] that may be used in an ActivityPub implementation. This list suggests using standards such as OAuth 2.0[9] for client-to-server authentication, as well as HTTP Signatures[10] and Linked Data Signatures[11] for server-to-server authentication. Furthermore, it recommends using HTTPS for its HTTP-based communication to provide transport-layer encryption.

2.2 ActivityPub-based Social Networks

2.2.1 The Fediverse

It is impossible not to refer to the *Fediverse* when ActivityPub is mentioned. The *Fediverse* is an interoperable collection of different federated social networks running on free open software on thousands of servers across the world that implement the same open-standard protocols to be able to interact with each other. It is developed by a not-profit-driven community of people around the globe independent of any corporation or official institution [16] [17]. The simplest way to explain how the federation works is the following example: Bob has a Twitter account, which he uses to follow all his friends that also have a Twitter account. Alice is a friend of Bob, but she only has an account on Youtube. In the real world, these two services are completely isolated and cannot communicate. However, if both had implemented the same social network protocol, such as ActivityPub, Bob would be able to find Alice by a normal search on Twitter and follow her. Allowing any new post of Alice on Youtube, to appear in Bob's Twitter timeline.

Before ActivityPub, the *Fediverse* implemented other protocols like *Ostatus*[12], *Matrix*[13], and *Diaspora*[14]. However, after ActivityPub was published as a Recommendation by the SocialWG

[8] https://www.w3.org/wiki/SocialCG/ActivityPub/Authentication_Authorization
[9] https://oauth.net/2/
[10] https://tools.ietf.org/html/draft-cavage-http-signatures-08
[11] https://w3c-dvcg.github.io/ld-signatures/
[12] https://ostatus.github.io/spec/OStatus%201.0%20Draft%202.html
[13] https://matrix.org
[14] https://diaspora.github.io/diaspora_federation

in January 2018, many of these federated social networks upgraded to ActivityPub, becoming the predominant protocol rapidly. Furthermore, the range of services that can be found inside the *Fediverse* includes blogging, microblogging, video streaming, photo, music sharing as well as file hosting. Examples are shown in table 2.2.

Name	Type	Equivalent to
[15]**PeerTube**	Video sharing	Youtube
[16]**Friendica**	Social networking	Facebook
[17]**Mastodon**	Microblogging	Twitter
[18]**Pixelfed**	Photo sharing	Instagram

Table 2.2: Examples of DOSN in the Fediverse

Although it was not the first social network to implement ActivityPub, Mastodon is the one that pioneered its use on a large scale [18]. In addition, it is the DOSN in the *Fediverse* with the most extensive user base and popularity. For this reason, Mastodon will be used in this thesis to represent the ActivityPub-based social networks and will be explained further in the next section.

2.2.2 Mastodon

Mastodon is a decentralized Twitter-like microblogging social network created with the idea of bringing social networking back into the hands of its users. The german creator of Mastodon, Eugen Rochko, shared the same opinion as what Fitzpatrick and Recordon said in 2007 [19]: *People are getting sick of registering and re-declaring their friends on every site*. For this reason, Eugen envisioned a social network that could end this, and *last forever* [20]. Like the *Fediverse*, Mastodon differs from other commercial social networks in two aspects. First, it is oriented towards small communities and community-based services. Each *instance*[19] is free to choose its topics; this way, users are encouraged to choose the instance better suited to their taste. Second, the Mastodon platform eliminates the presence of sponsored users or posts in feeds. This implies that the only way to connect or consume content is through a self-search to find an already known account or to explore the users' feeds in other instances with similar interests [21].

From a user experience perspective, Mastodon includes all the essential features of a microblogging platform, such as:

- Follow other users, even if they are not in the same instance.
- Post small status updates, or *toots*, up to 500 characters long.
- Access to a timeline of the local instance and federated statuses.
- Control over the visibility of their posts, with the option to set them as private, instance-level only, or federated.

[19] A server running Mastodon

Mastodon's implementation of ActivityPub follows the guidelines defined by the spec. However, as the protocol does not specify how to implement some key processes required for a fully-working social network, Mastodon extended the protocol with the following processes and features.

Security

Mastodon implemented the authentication and authorization mechanisms from the best practices list of the SocialWG to address the security concerns in ActiviyPub. The ones relevant to this thesis are the HTTP and the JSON-LD signatures. HTTP signatures extend the HTTP protocol by adding the possibility to sign the HTTP requests cryptographically. This signature gets added to the request within the *Signature* header, and it provides not only end-to-end message integrity but also proof of the authenticity of the sender without the need for multiple round-trips [22]. Creating an HTTP signature means signing the parameters of the request itself, i.e the *request-target*, the *host*, and the *date*. These parameters and their values are concatenated in a single string, hashed, and signed with the sender's public key. Finally, the *Signature* header indicates the key used to sign the document, the parameters inside the signature, and the algorithm used to hash it. An example in Mastodon is shown by listing 2.3. Following the same idea, Linked Data Signatures offer a way to create and attach signatures to JSON-LD documents. This kind of signature can provide non-repudiation to a JSON-LD object, like an ActivityStreams Activity object, even if the object has been shared, forwarded, or referenced at a future time [12]. However, this feature, although implemented, is not actively used in Mastodon. For Mastodon to be able to implement these signatures, it was necessary to generate keypairs for the users. For this reason, Mastodon added a new property *publicKey* to the actor object, which includes the pem-formatted public key of an RSA-2018 keypair. See appendix 1 for the complete actor object that includes the added properties in the Mastodon implementation.

```
1
2 GET /users/username/inbox HTTP/1.1
3 Host: mastodon.example
4 Date: 18 Dec 2019 10:08:46 GMT
5 Accept: application/activity+json
6 Signature: keyId="https://my-example.com/actor#main-key",headers="(request-
    target) host date",signature="Y2FiYW...IxNGRiZDk4ZA=="
```

Listing 2.3: Signed HTTP Request

Resolving accounts

As explained in 2.1.2, ActivityPub requires a resolving process when sending a message to a user whose account resides on a different server. For this, Mastodon implemented *Well-Known URIs*[20], which enable the discovery of information about an origin in well-known endpoints [23]. The two most relevant endpoints are the following:

[20] https://www.rfc-editor.org/rfc/rfc8615.html

Web Host Metadata

This endpoint allows the discovery of host information using a lightweight metadata document format. In this context, *host* refers to the entity in charge of a collection of resources defined by URIs with a common URI host [24]. It employs the XRD 1.0[21] document format, which offers a basic and flexible XML-based schema for resource description. Moreover, it provides two mechanisms for providing resource-specific information, *link templates* and *Link-based Resource Descriptor Documents* (LRDD). On the one hand, link templates require a URI to work, thus avoiding the use of fixed URIs. On the other hand, the LRDD relation type is used to relate *LRDD documents* to resources or host-meta documents [24]. In the specific case of the Mastodon implementation, requesting the host-meta endpoint will give us back the *lrdd* link to the Webfinger endpoint, where specific resource information can be found. This is illustrated by figures 2.5 and 2.4.

```
1    GET /.well-known/host-meta HTTP/1.1
2    Host: mastodon.social
3    Accept: application/xrd+xml
```

Listing 2.4: Web Host Metadata request to mastodon.social

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <XRD xmlns="http://docs.oasis-open.org/ns/xri/xrd-1.0">
3      <Link rel="lrdd" template="https://mastodon.social/.well-known/
     webfinger?resource={uri}"/>
4    </XRD>
```

Listing 2.5: Web Host metadata response from mastodon.social

Webfinger

Mastodon relies on the Webfinger protocol for the resolving process and its federated functioning [25]. It is an HTTP-based protocol that allows for discovering information about persons or other entities on the Internet. This information can be a personal profile, an address, an identity service, a telephone number or an email [26]. Performing a query to a WebFinger endpoint requires a query component with a resource parameter, which is the URI that identifies the identity that is being looked up. Mastodon employs the *acct*[22] URI format, which aims to offer a scheme that generically identifies a user's account with a service provider without requiring a specific protocol. Webfinger's response consists of a *JSON Resource Descriptor* (JRD) document describing the entity [26]. Fig. shows an example of the returned JRD document provided by the WebFinger endpoint of the *mastodon.social*[23] instance when querying the account *acct:bob@mastodon.social*.

[21] https://docs.oasis-open.org/xri/xrd/v1.0/os/xrd-1.0-os.html
[22] https://datatracker.ietf.org/doc/html/rfc7565
[23] https://mastodon.social

```
GET /.well-known/webfinger?resource=acct:bob@mastodon.social
Host: mastodon.social
Accept: application/xrd+xml
```

Listing 2.6: HTTP request to Webfinger endpoint

```
{
   "subject": "acct:bob@mastodon.social",
   "aliases": [
       "https://mastodon.social/@bob",
       "https://mastodon.social/users/bob"
   ],
   "links": [
       {
           "rel": "http://webfinger.net/rel/profile-page",
           "type": "text/html",
           "href": "https://mastodon.social/@bob"
       },
       {
           "rel": "self",
           "type": "application/activity+json",
           "href": "https://mastodon.social/users/bob"
       },
   ]
}
```

Listing 2.7: Webfinger response

2.3 Extending ActivityPub

This thesis is not the first attempt to address the deficiencies of the ActivityPub protocol. The co-author of ActivityPub, Christopher Webber, and the co-author of the DID specification Manu Sporny presented a different approach during the Rebooting Web of Trust[24] summit in 2017 [8]. This approach includes the following features:

End-To-End Encryption

To create a web of trust when using ActivityPub, the authors suggested using the IETF standard OpenPGP[25], which provides a format for encrypting data using public key cryptography. This way, only the user's device that contains the private key can decrypt and read the plain-text message. However, this encryption brings a series of consequences that affect the usability of a social network. Especially on the server side, some *activities* trigger callbacks that the server usually processes in the background. For example, if a server receives a *like* activity, it will increase a counter, add the liked post to the *liked* collection, and notify the post's creator. Nevertheless, all these features cannot be performed if the activity is encrypted. Other consequences

[24] https://www.weboftrust.info/
[25] https://www.openpgp.org/

include the key management for the users, which makes the user experience problematic, and in case of key loss, there would be no way to read old posts [8]. Listing 2.8 shows an example of an encrypted Activity.

```
{
  "@context": ["https://securityns.example/",
  "https://www.w3.org/ns/activitystreams"],
  "type": "EncryptedEnvelope",
  "encryptedMessage": "-----BEGIN PGP MESSAGE-----\r\n...",
  "mediaType": "application/ld+json; profile=\"https://www.w3.org/ns/
   activitystreams\"
}
```

Listing 2.8: Note object with encrypted content [8]

Signing Objects

They proposed using JSON-LD signatures to provide non-repudiation and to assert the integrity of messages sent over the network. They added a public key property to the actor object, just as Mastodon did to enable HTTP Signatures. The resulting signed activity is shown by listing 2.9.

```
  {
    "type": "Note",
    "id": "https://social.example/alyssa/posts/63cc87ec-416e-437d-...",
    "attributedTo": "https://social.example/alyssa/",
    "to": ["https://havoc.example~mallet/"],
    "content": "Tortellini are delicious",
    "signature": {
      "type": "RsaSignature2017",
      "creator": "https://social.example/alyssa/",
      "created": "2017-09-23T20:21:34Z",
      "nonce": "e3689a56da9b4bc",
      "signatureValue": "mJfe5OCb7J3WwI...8t5/m="
      }
  }
```

Listing 2.9: Note object with JSON-LD Signature. Adapted from [8]

Introducing DIDs

Their approach to implementing DIDs is using DIDs as actors. Thus, switching from a standard URL like *https://social.example/alyssa/* to a DID.

Integrating ActivityPub into the DID document

The authors opted to combine the actor object into the DID document, allowing all the necessary endpoints to be discoverable and stored in the VDR. In addition, to remove any dependency on a centralized DNS, the inbox, outbox, following, and other properties are now based on Tor Hidden Services[26]. The resulting DID document is illustrated by listing 2.10

```
{
  "@context": ["https://example.org/did/v1",
               "https://www.w3.org/ns/activitystreams"],
  "id": "did:example:d20Hg0teN72oFeo0iNYrblwqt",
  "activityPubService": {
    "id": "did:example:d20Hg0teN72oFeo0iNYrblwqt#services/ActivityPub",
    // ActivityPub actor information
    "type": "Person",
    "name": "Alyssa P. Hacker",
    "preferredUsername": "alyssa",
    "summary": "Lisp enthusiast hailing from MIT",
    "inbox": "https://9GaksjPhy0mWToTV.onion/alyssa/inbox/",
    "outbox": "https://9GaksjPhy0mWToTV.onion/alyssa/outbox/",
    "followers": "https://9GaksjPhy0mWToTV.onion/alyssa/followers/",
    "following": "https://9GaksjPhy0mWToTV.onion/alyssa/following/",
    "liked": "https://9GaksjPhy0mWToTV.onion/alyssa/liked/"},
  // DDO information
  "owner": [{
    "id": "did:example:d20Hg0teN72oFeo0iNYrblwqt#key-1",
    "type": ["CryptographicKey", "EdDsaPublicKey"],
    "curve": "ed25519",
    "expires": "2017-02-08T16:02:20Z",
    "publicKeyBase64": "lji9qTtkCydxtez/bt1zdLxVMMbz4SzWvlqgOBmURoM="
  }],
  "control": [{
    "type": "OrControl",
    "signer": [
        "did:example:d20Hg0teN72oFeo0iNYrblwqt",
        "did:example:8uQhQMGzWxR8vw5P3UWH1j"
    ]
  }],
  "created": "2002-10-10T17:00:00Z",
  "updated": "2016-10-17T02:41:00Z"
}
```

Listing 2.10: DID document with inserted ActivityPub actor. Adapted from [8]

Mastodon implemented another approach to extending ActivityPub, which consists of an end-to-end encryption API[27]. This API was the result of a long-discussed feature request[28] for

[26] https://www.torproject.org
[27] https://github.com/mastodon/mastodon/pull/13820
[28] https://github.com/mastodon/mastodon/issues/1093

an Off-The-Record[29] (OTR) endpoint for direct messages. In this discussion, different alternatives were mentioned, such as using the PGP-based Mailvelope[30], which provides end-to-end mail encryption; or the Message Layer Security (MLS)[31] from the IETF. In the end, they added an API to send messages using the Matrix[32] implementation of the Double Ratchet Algorithm[33], which is part of the protocol used in the messaging app Signal[34]. With these endpoints, it is possible to register a device with a key generated by Mastodon and to send encrypted messages to specific devices of other people.

This implementation follows the same idea proposed by Webber and Sporny in 2.3. Nonetheless, it is still unclear why Mastodon has kept this feature *hidden*. There is no documentation, and designing a UI for it is not planned. Therefore, preventing people from knowing and taking advantage of it. For an example of an encrypted Activity using this endpoint see appendix 2.

2.4 Decentralized Identifiers

On July 19. 2022 the W3C announced that Decentralized Identifiers (DIDs) v1.0 is officially a Web standard. This new type of globally unique identifier brings a Self-Sovereign approach to digital identities, enabling individuals and organizations to take control of their online information and relationships while also providing greater security and privacy [11]. Self-Sovereign Identity *SSI* implies a sovereign, enduring, decentralized, and portable digital identity for any human or non-human entity that enables its owner to access services in the digital world in a secure, private, and trusted manner. DIDs are the key component of the SSI framework, as they allow identifiers to be created independently of any centralized registry, identity provider, or certificate authority with full control given to its owner[27][10].

2.4.1 Architecture

The DID itself is a URI that consists of 3 different parts. The DID URI scheme identifier, the method identifier, and the DID method-specific identifier as shown in figure 2.2. The entity being identified by the DID is called the *DID subject*. *Everything* can be identified by a DID, including any person, group, organization, as well as a physical, digital, or logical thing [28][10].

Figure 2.2: DID composition [10]

[29] https://otr.im/
[30] https://mailvelope.com/en
[31] https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/
[32] https://matrix.org/docs/projects/other/olm
[33] https://signal.org/docs/specifications/doubleratchet/
[34] https://signal.org

The DID Subject that can modify the DID Document is called the *DID controller*. Usually, the DID subject is the DID controller, but this is not compulsory. As shown in figure 2.3, a DID resolves to a *DID Document*, which is a JSON-based object that contains information associated with a DID. It includes verification methods, such as cryptographic public keys and relevant services to interact with the DID Subject. An example of a DID Document can be seen in 2.11. Furthermore, it is possible to retrieve a specific resource of the DID document by using a *DID URL*, which is a DID that includes a path, query, or fragment [10].

```

{
  "@context": "https://w3id.org/did/v1",
  "id": "did:example:123456789abcdefghi",
  "publicKey": [{
    "id": "did:example:123456789abcdefghi#keys-1",  // DID URL
    "type": "RsaVerificationKey2018",
    "owner": "did:example:123456789abcdefghi",
    "publicKeyPem": "..."
  }],
  "authentication": [{
    "type": "RsaSignatureAuthentication2018",
    "publicKey": "did:example:123456789abcdefghi#keys-1"
  }],
  "service": [{
    "type": "ExampleService",
    "serviceEndpoint": "https://example.com/endpoint/8377464"
  }]
}

%
```

Listing 2.11: Example DID Document

DID documents are stored in a *Verifiable Data Registry* (VDR). A VDR is essentially any system that enables capturing DIDs and returning required data to generate DID documents. For example, distributed ledgers, decentralized file systems, any decentralized database, peer-to-peer networks, or other types of trustworthy data storage. The next component is the *DID method*, which describes the processes for CRUD operations for DIDs and DID documents based on a specific type of VDR. According to the DID registry[35] of the W3C, there are around 103 registered DID method specifications. More information about the different existing DID methods can be found in subsection 2.4.2.

The last major component in this architecture overview is the one in charge of resolving DIDs, namely, the *DID resolver*. This component implements the *DID resolution*, which consists of taking a DID as an input, and giving a DID Document as an output [10]. The Identifiers & Discovery Working Group (ID WG) has implemented a prototype Universal Resolver[36], which allows the resolution of DIDs for numerous DID methods. In addition, this working group has also developed a Universal Registrar[37], which allows the creation, edition, and deactivation of the DIDs across different DID methods.

[35] https://www.w3.org/TR/did-spec-registries/#did-methods
[36] https://github.com/decentralized-identity/universal-resolver
[37] https://uniregistrar.io/

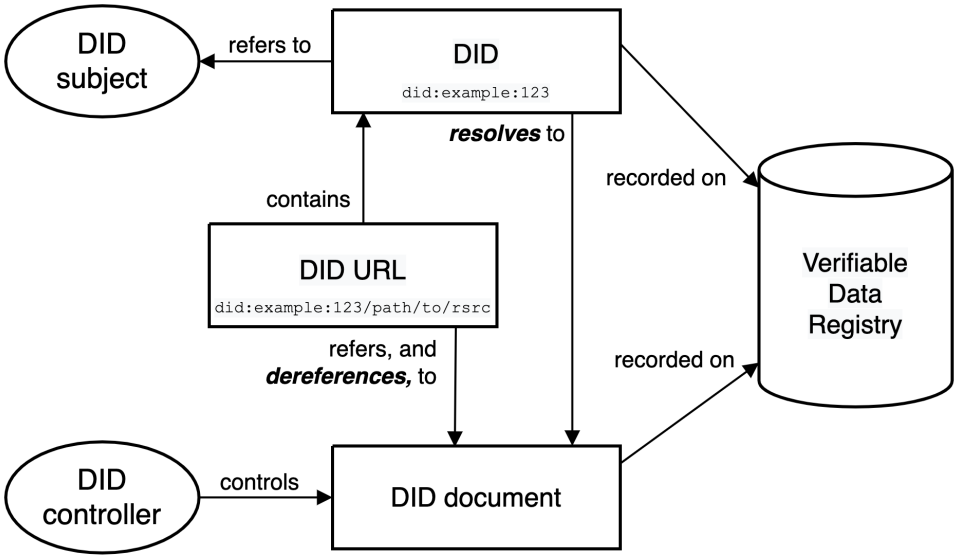

Figure 2.3: DID architecture overview [10]

2.4.2 DID methods

Based on their characteristics and patterns, most DIDs can be sorted into different categories [29], for example:

- **Ledger-based DIDs**: This includes all the DIDs that store DIDs in a blockchain or other Distributed Ledger Technologies (DLTs). Examples include *did:btcr*, *did:ethr* and *did:trx*, whose DIDs are stored in the Bitcoin, Ethereum, and Tron network respectively.
- **Ledger Middleware (*Layer* 2) DIDs**: Layer 2 refers to a framework or protocol that is built on top of an existing blockchain system that takes the transactional burden away from layer 1, making it more scalable [30]. An example DID method in this category is the *did:ion*[38], which runs in a layer on top of Bitcoin.
- **Peer DIDs**: DIDs have the required ability to be resolvable. However, not all of them have to be globally resolvable. Peer DIDs do not exist on a global source of truth but in the context of relationships between peers in a limited number of participants. Nonetheless, they are still valid DIDs as they comply with the core properties and functionalities that a DID has to provide. [29].
- **Static DIDs**: This type of DIDs are limited in the kind of operations that can be performed on them. These DIDs are not stored in any VDR. Consequently, it is not possible to update, deactivate or rotate them. Using the *did:key* method as an example, the DID-method-specific part of the DID is encoded in a way that the DID document can be extracted from it.[31].

Designing a DID method comprises different tasks such as defining if and how a DID will be anchored to a VDR, selecting if CRUD operations are possible and how to perform them, and

[38] https://identity.foundation/ion/

defining privacy and security mechanisms like verification method rotations or DID recovery. Some platforms behind these DID methods include the former Uport, with *did:ethr*; Sovrin Foundation, with *did:sov*; and the DIF with *did:ion*.

2.5 DIDComm Messaging

The Hyperledger Foundation is an open-source collaborative effort intended to develop blockchain technologies across industries further [32]. Started in 2016 by the Linux Foundation, it has given birth to numerous enterprise-grade software open-source projects that can be classified into DLTs, libraries, tools, and labs [33]. One of these graduate projects is Hyperledger Aries, which, together with Hyperledger Indy (HI) and Hyperledger Ursa (HU), makes up the Sovereign Identity Blockchain Solutions of Hyperledger. HI supplies a distributed ledger specifically built for decentralized identity, and HU is a shared cryptography library that helps avoid duplicating cryptographic work across projects while potentially increasing security. Finally, Aries provides solutions for SSI-based identity management, including key management, credential management, and an encrypted, peer-to-peer DID-based messaging system that is now labeled as DIDComm v1 [32]. Based on DIDComm v1, the DIF's Communication Working Group (CWG) has implemented DIDComm v2. The CWG pursues the standardization of DIDComm v2 not only to widen its implementation beyond Aries-based projects but to create an interoperable layer that allows higher-order protocols to build upon its security, privacy, decentralization, and transport independence in the same way web services build upon HTTP. [34] [35]

From this point on, the term *DIDComm* will refer exclusively to DIDComm Messaging v2. DIDComm can be described as a communication protocol that promises a secure and private methodology that builds on top of the decentralized design of DIDs. It is a versatile protocol that supports a wide range of features, such as security, privacy, decentralization, extensibility, interoperability, and the ability to be transport-agnostic [35].

DIDComm differs from the current dominant web paradigm, where something as simple as an API call requires an almost immediate response through the same channel from the receiving end. However, this duplex request-response interaction is not always possible for several reasons. For example, some agents may not have a constant network connection; others may interact only in larger time frames, and some may even not listen over the same channel where the original message came from. DIDComm's paradigm is asynchronous and one-directional, thus showing a considerable resemblance to the email paradigm.

Furthermore, the web paradigm assumes the use of traditional processes like authentication, session management, and end-to-end encryption. DIDComm does not require certificates from external parties to establish trust, nor does it require constant connections for end-to-end transport level encryption like TLS. This takes the security and privacy responsibility away from institutions and places it within the agents. All of this without limiting the communication possibilities due to its ability to function as a base layer, upon which capabilities like sessions and synchronous interactions can be built [35].
To better understand how it works, let's look at how it would work in a scenario where Alice wants to send a private message to Bob:

Algorithm 1 Example of DID communication using DIDComm [36]

1: Alice has a private key sk_a and a DID Document for Bob containing an endpoint $(endpoint_{bob})$ and a public key (pk_b).
2: Bob has a private key sk_b and a DID Document for Alice containing her public key (pk_a).
3: Alice encrypts plaintext message (m) using pk_b and creates an encrypted message (eb).
4: Alice signs eb using her private key sk_a and creates a signature (s).
5: Alice sends (eb, s) to $endpoint_{bob}$.
6: Bob receives the message from Alice at $endpoint_{bob}$.
7: Bob verifies (s) using Alice's public key pk_a
8: **if** Verify $(eb, s, pk_a) = 1$ **then**
9: Bob decrypts eb using sk_b.
10: Bob reads the plaintext message (m) sent by Alice
11: **end if**

To achieve the encryption and signing processes mentioned in algorithm 1, DIDComm implements a family of the Internet Engineering Task Force (IETF) standards, collectively called JSON Object Signing and Encryption (JOSE). The JOSE Working Group[39] strived to define JSON-based object formats to represent integrity, confidentiality, and cryptographic keys. The following are some of the resulting standards from this working group:

- **JSON Web Signature (JWS):** A JWS is a subclass of the **JSON Web Token (JWT)** standard, which provides a JSON format to represent claims. A JWT becomes a JWS when it is digitally signed or by adding Message Authentication Codes (MAC)[37].
- **JSON Web Encryption (JWE):** Similar to a JWS, a JWE is also a subclass of a JWT. However, instead of signing, it encrypts the claims to add confidentiality. [38].
- **JSON Web Key (JWK):** It provides a JSON format to represent a cryptographic key [39].
- **JSON Web Algorithms (JWA):** It provides a collection of algorithms to be used by the previously mentioned standards [40].

Furthermore, DIDComm specifies using other proposed standards that were not part of the JOSE WG. For example, the **JSON Web Message (JWM)**[40], which is a basic, flexible JSON format to encode messages for a transport agnostic delivery; and the **ECDH-1PU**[41], a public key authenticated encryption algorithm designed for the JWE.

[39] https://datatracker.ietf.org/group/jose/about/
[40] https://datatracker.ietf.org/doc/pdf/draft-looker-jwm-01
[41] https://datatracker.ietf.org/doc/html/draft-madden-jose-ecdh-1pu-04

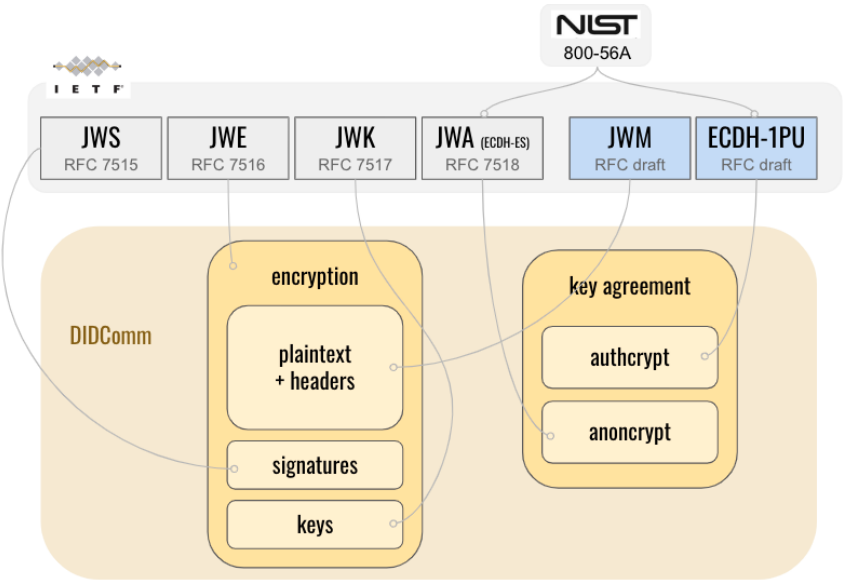

Figure 2.4: Standards used in DIDComm [41]

As shown in figure 2.4, DIDComm offers different approaches for key agreement encryption, namely, Authenticated Sender Encryption (*authcrypt*) and Anonymous Sender Encryption (*anoncrypt*). Both options are encrypted and delivered to the recipient's DID, but only *authcrypt* can authenticate the sender's identity. Sending anonymous messages in social networks is not usually the case, and removing the attribution can lead to other problems [42]. Nonetheless, social networks like Ask.fm[42] or NGL[43] that rely on anonymous posts could use the advantages of *anoncrypt*.

DIDComm recommends using *authcrypt* as the standard to provide confidentiality, message integrity, and authenticity of the sender. For this, *authcrypt* requires the *ECDH-1PU* proposed standard. An alternative to *Authcrypt* that also complies with the required confidentiality and non-repudiation requirements is to have a nested JWT, shown in figure 2.5. To achieve this, the plaintext is first signed, and the resulting JWS is used as the JWE's content. The algorithm 2 illustrates better the workings of this.

Algorithm 2 Communication example with nested JWT

1: Alice signs a plain text message using her private key sk_a and creates a *(JWS)*.
2: Alice encrypts the *(JWS)* using Bob's public key pk_b and creates a *(JWE)*.
3: Alice sends *JWE* to Bob.
4: Bob decrypts *(JWE)* using his private key sk_b and obtains the *JWS*
5: Bob verifies *(JWS)* using Alice's public key pk_a

[42] https//ask.fm
[43] https://ask.fun

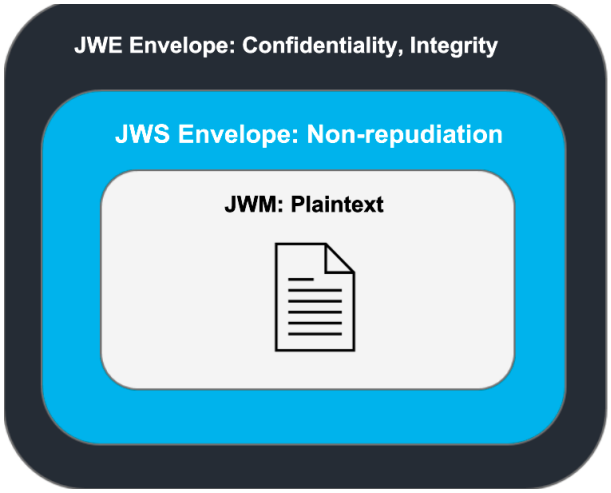

Figure 2.5: Nested JWT example

3 Concept and Design

The standards presented in chapter 2 show the potential improvements that can be achieved in key components of ActiviyPub-based social networks. In this chapter, the design that combines these standards with an actual ActivityPub implementation will be presented. As mentioned in chapter 2, the proposal presented in this chapter and the modus operandi of the ActivityPub server is going to be scoped to the actual implementation in Mastodon.

The outline for this chapter is the following. First, individual concepts of the ActivityPub implementation are explained. Then, a use case example is going to be illustrated and step-by-step explained to be able to compare it with section 3.4, which finally presents the same use case but this time with the proposed concept and design that includes DID integration and DIDComm enablement.

3.1 Definitions

Mastodon has implemented its own ActivityPub server, and with it also its own terms to express different social network vocabulary. To prevent confusion or ambiguities, the used terms in this chapter are explained here.

- **Username**: The username in Mastodon consists of a unique local username and the domain of the instance. Ex. alice@example.com
- **Actor object**: In this section, the term *Actor object* refers solely to the ActivityPub's actor object.
- **Status**: In the backend of Mastodon, the class used for a Toot is a Status. An account in Mastodon has a 1:n relationship with statuses.

3.2 Use Case

To explain the current ActivityPub flow in Mastodon, the following use case will be used:

Alice has an account in the Mastodon instance alice_server.com. Alice follows Bob, who has an account in the Mastodon instance bob_server.com. Alice sends a direct message to Bob with the text: "Hello Bob!"

3.3 Mastodon's Implementation

In this section, each step required for the use case to be successful will be explained. At the end of this section, there is a sequence diagram, figure 3.1, that provides an overview of the whole process, with a reference to every step.

3.3.1 Activity Creation

The first thing that happens when Alice presses the send button is the creation of an ActivityStreams object. In this case, the object is of type *Note* and will be created by Alice's server, as shown in 3.1. Then, following the ActivityPub pattern of "some activity by some actor being taken on some object"[7], the server wraps it in a *Create* activity. The activity includes Alice in the *attributeTo* field 3.2. Now that the actor, the activity, and the object are well defined and wrapped, it is time to shift our focus to the recipients of this note object. Alice's server will now look at the fields to, bto, cc, bcc, and audience to retrieve the recipients. Depending on where the recipient's account lives, Alices' server may take one of two options. If the recipient's account is on the same server, a simple query in the server would find the right account and save the status to the account's collection. Nonetheless, the use case explicitly dictates that Bob's account resides in a different Mastodon instance, namely *bob_server.com*. Therefore, a resolving process must be started.

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Note",
  "to": "http://bob_server.com/users/bob",
  "attributedTo": "http://alice_server.com/users/alice",
  "content": "Hello Bob!"
}
```

Listing 3.1: ActivityStreams note object

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Create",
  "id": "https://alice_server/users/alice/statuses/634367/activity",
  "to": "http://bob_server.com/users/bob",
  "actor": "http://alice_server.com/users/alice",
  "object": {
    "type": "Note",
    "to": "http://bob_server.com/users/bob",
    "attributedTo": "http://alice_server.com/users/alice",
    "content": "Hello Bob!"
  }
}
```

Listing 3.2: ActivityStreams create activity

3.3.2 DNS-based Resolving Process

Resolving is the fundamental part of the federated side of Mastodon. Without it, users within different instances would not be able to interact, as the instance itself does not know where to find the actor object with all required endpoints to send or receive activities from and to external accounts. For this reason, the current way to look up other accounts is through the DNS. In the same way Email works, the domain part of the username in Mastodon points to the domain of the instance where the account lives. The purpose of resolving is to find Bob's inbox URL, which can be found in Bob's actor object. As explained in chapter 2, Mastodon includes a series of well-known endpoints that are used to retrieve information about resources managed by the host. As Bob's account lives inside *bob_server.com*, Mastodon triggers a Webfinger request to Bob's server. This request returns a JRD Document, as shown in listing 3.3 and 3.4. Based on this document, Alice's server retrieves the link with the *rel: 'self'* which includes the type and the URL where Bob's actor object can be retrieved. A subsequent HTTP GET request to this URL with the specific *application/activity+json* header will return Bob's actor object 3.7. If the Webfinger request 3.5 returns a 404 code, it will then try, as a fallback, using the Host-Meta endpoint. The response contains a link template 3.6, that Alice's server can use to try the Webfinger request again.

```
GET /.well-known/webfinger?resource=acct:bob@bob_server.com HTTP/1.1
Host: bob_server.com
Accept: application/ld+json
```

Listing 3.3: Webfinger request to *bob_server.com*

```
{
  "subject": "acct:bob@bob_server.com",
  "aliases": [
    "https://bob_server.com/@bob",
    "https://bob_server.com/users/bob"
  ],
  "links": [{
      "rel": "http://webfinger.net/rel/profile-page",
      "type": "text/html",
      "href": "https://bob_server.com/@bob"
    },
    {
      "rel": "self",
      "type": "application/activity+json",
      "href": "https://bob_server.com/users/bob"
    },
  ]
}
```

Listing 3.4: Webfinger response from *bob_server.com*

```
GET /.well-known/host-meta HTTP/1.1
Host: bob_server.com
Accept: application/xrd+xml
```

Listing 3.5: Host-Meta request to *bob_server.com*

```
<?xml version="1.0" encoding="UTF-8"?>
<XRD xmlns="http://docs.oasis-open.org/ns/xri/xrd-1.0">
    <Link rel="lrdd" template="https://bob_server.com/.well-known/
 webfinger?resource={uri}"/>
</XRD>
```

Listing 3.6: Host-Meta response from *bob_server.com*

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Person",
  "id": "https://bob_server.com/bob/",
  "name": "Bob",
  "preferredUsername": "Bob",
  "summary": "Example Person",
  "inbox": "https://bob_server.com/users/bob/inbox/",// Needed inbox URL
  "outbox": "https://bob_server.com/users/bob/outbox/",
  "followers": "https://bob_server.com/users/bob/followers/",
  "following": "https://bob_server.com/users/bob/following/",
  "liked": "https://bob_server.com/users/bob/liked/"
}
```

Listing 3.7: Bob's actor object

3.3.3 Delivery

Succeeding the retrieval of Bob's inbox URL, the delivery can now take place. To provide end-to-end message integrity and to authenticate Alice in Bob's server, the request is signed by Alice's server using the HTTP Signature specification. Upon receiving the POST request to Bob's inbox URL, Bob's server has to verify the signature. For this, it starts the resolving process all over again to access the actor object of Alice, where Alice's public key can be found. After successful validation, Bob's server saves the Note object in Bob's statuses. As indicated in subsection 2.2.2, HTTP signatures are not part of the ActivityPub protocol standard. These security features are within the Mastodon implementation of ActivityPub.

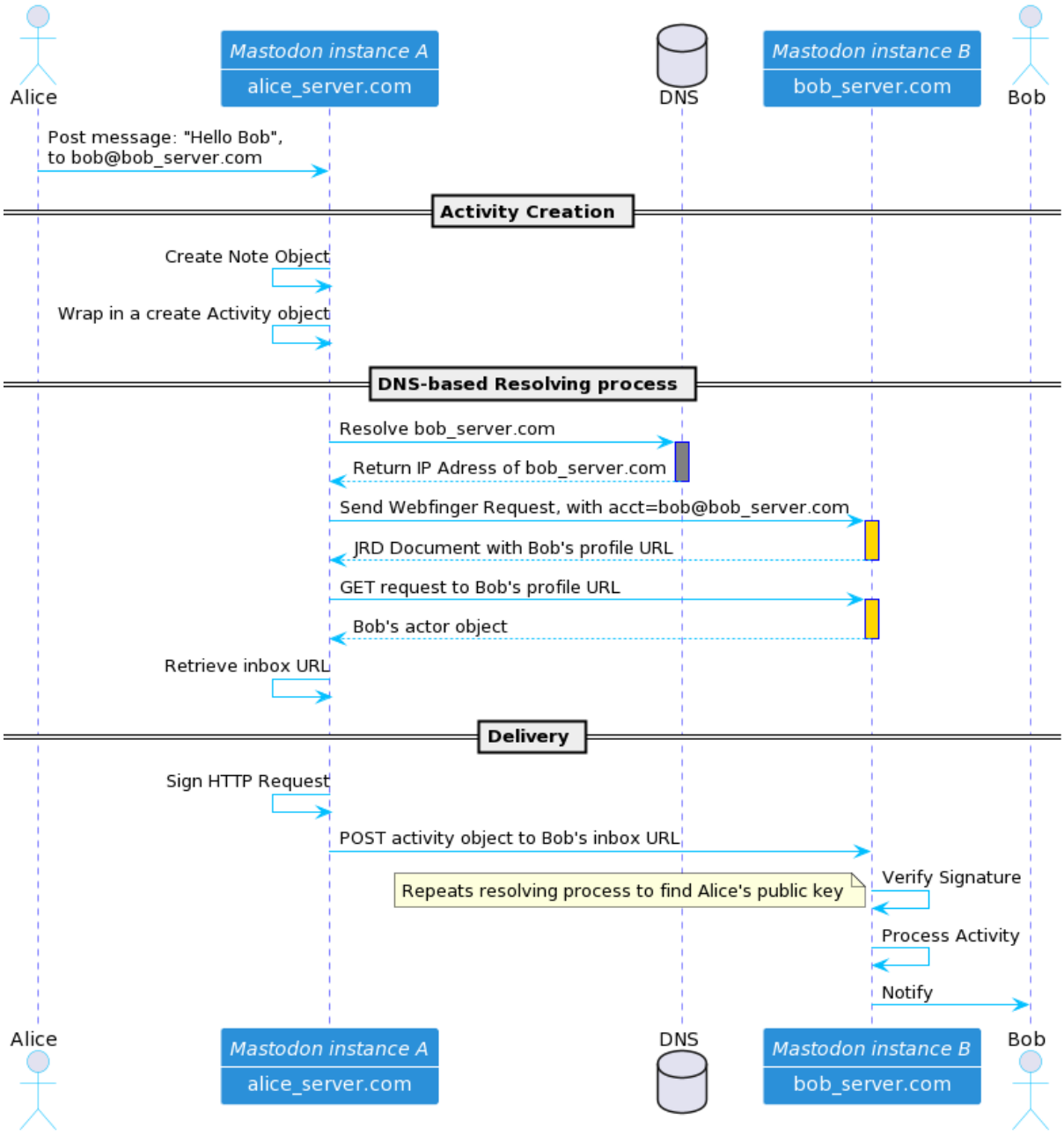

Figure 3.1: Mastodon's implementation for the use case mentioned in 3.2

3.4 Proposed Implementation

Having seen a state-of-the-art implementation of ActivityPub for our use case, it is now imperative to address the necessary steps to make ActivityPub DID-compliant, as well as enable DIDComm as its communication protocol.

3.4.1 Integrating DIDs

The first question that needs to be addressed when approaching the integration of DIDs is the implications of switching from standard mastodon usernames to DIDs. Integrating DID to ActivityPub points immediately to the actor's object, as the switch would mean that the DID has to be included. Currently, most of the interactions of Mastodon via ActivityPub require the ID property to resolve to the user's profile, and therefore, to his actor's object. The simplest strategy could be, replacing the username with the DID. Thus having an ID like `"www.alice_server.com/users/did:example:123456789abcdefghi"`.
However, there is another alternative that might work. Following the ActivityPub's specification, the ID must be a publicly dereferenceable URI, whose authority belongs to the originating server [7]. As explained in section 2.4, a DID is a URI and it is publicly dereferenceable by nature. This allows different possibilities, to which the DID can be added. For example, using a stand-alone DID as an ID to take advantage of the discoverability of DIDs, just as proposed by Webber and Sporney in section 2.3. This scenario would have the following implications. If another ActivityPub server wanted to get the user's profile URL, it would require resolving the DID to its respective DID document. This requires the DID document to include the actor's profile URL in the services section. Another option would be to add a DID URL with a query that points directly to the service endpoint that contains the actor's profile URL. Adding more precision, although still requiring a DID resolution as an intermediate step, as well as adding the service endpoint to the DID Document. Both cases are possible because ActivityPub has the *URL* property that requires the actor's profile URL in case it is not in the ID property [7]. Although plausible, to prevent any unnecessary overhead a simple replacing strategy will be implemented, keeping the ID property as the profile's URL with the username replaced with the DID.

In addition to the ID, the actor's object must provide a supplementary set of URLs that point to different collections related to the user, as mentioned in subsection 2.2.2. Following the same approach as with the ID, what if instead of using the actual URL of these collections, DID URLs pointing to the correct endpoint inside the DID Document were specified? An example for the inbox could be: `"did:example:123456789abcdefghi#inbox"`.

This approach leads to the question, would it be simpler just to shift the whole actor object directly to the DID Document? This would imply that the actor object of ActivityPub is not necessary anymore and could be removed. This is also the approach Webber and Sporny proposed, mentioned in 2.3. However, there are some security privacy concerns regarding the use of service endpoints in the DID Document that would advise against this. The DID specification stipulates *"revealing public information through services, such as social media accounts, personal websites, and email addresses, is discouraged"* [10]. DID Documents are stored in a publicly available VDR, therefore any personal information revealed here is for everyone to see. The usage of URLs in service endpoints might lead to involuntary leakage of personal information or a

correlation between the URL and the DID subject. In the DID document proposed by these authors, listing 2.10, the amount of personal information displayed, which would not be otherwise inferable, already poses a privacy issue for the DID Subject. For this reason, the actor's object will remain a part of the ActivityPub protocol in this design. This would also allow the unrestricted use of all the other attributes in the actor's object to further describe its owner, such as *name*, *preferredUsername*, or *summary* without making this information forever public in an immutable ledger.

3.4.2 Decentralized Resolving Process

As stated in the previous section, Mastodon starts the resolving process based on the username of the user. By replacing the standard username with a DID, the current resolving flow gets disrupted, as there is no domain and thus no well-known endpoints to send requests to. Nonetheless, here is where the resolvability of the DIDs comes into play. The proposed flow takes the following steps. First, the username, which now is a DID, can be resolved to its DID Document using any kind of DID resolver. The DID Document must now contain a service with the type ActivityPub and an endpoint, where the actor's object can be retrieved. This gives us 2 possibilities. On the one hand, we could add in the well-known Webfinger endpoint, which then provides us with the profile URL from the user. On the other hand, we could skip the Webfinger request and provide the profile URL directly in the DID Document. The latter looks like the most meaningful path to take. Especially when we refer to the purpose of Webfinger, which was to enable discoverability of entities represented by URIs [26]. Webfinger's purpose shares a lot of ground with the design of DIDs, nevertheless, the DID design provides a less limited structure of resolving, as it does not rely on DNS and HTTP for its functioning. For this reason, the proposed workflow will completely remove the use of the Webfinger protocol used in Mastodon and include the URL of the user's profile in the DID Document.

To make the requests to the resolver a service class *DidResolverService* is needed. This class will only require one parameter, which is the DID it needs to resolve, and the response will consist of a DID document in JSON format. Furthermore, to facilitate the interaction with the properties of the DID document a class *DidDocument* with the methods listed in table 3.1 is also needed.

DID document Class	
Function	**Description**
initialize(attributes)	Stores the attributes of the DID document in accesible variables
serviceEndpoint	Returns the service endpoint URL of the first service
didcommKey	Looks for a specific key in the DID document and returns a parsed instance of it

Table 3.1: DID document instance methods

With DIDs, a DID resolver, a resolver service, and a class for DID documents set up, the

next task is to modify the Webfinger-based resolving process of Mastodon. Mastodon has a class called *ResolveAccountService*, which triggers the Webfinger requests and processes the respective responses. It takes a username in the form of *username@domain* as a parameter. If the username does not have an existing account in the local database, it makes the Webfinger request. The JRD response gets parsed to find the actor URL, and a subsequent request to the username domain gets triggered to get the actor object. Finally, it parses the actor object to create an account for this user in the local database. The new class handling the decentralized DID-based resolving process is not very different. It will be called *DID::ResolveAccountService* and will also take the username parameter in the form of *did:method:example*. To resolve a DID it will first make a query to the DID resolver calling the *DidResolverService*, this service instantiates a new *DidDocument* class with the JSON object of the response. The profile URL of the account is then retrieved using the *serviceEndpoint* method. Finally, as in the previous flow, a request is made to this URL to get the actor object for further processing, finalizing the resolving process.

3.4.3 Enabling DIDComm

Having introduced DIDs to Mastodon and ActivityPub, it is now possible to enable DIDComm. Taking into account the algorithm 1 shown in section 2.5, it is possible to derive some requirements that DIDComm imposes:

Access to DID-Resolver

This is for cases where a signature needs to be verified using an external public key published in a DID document. However, as the DID-based resolving process already requires and implements a DID resolver, this requirement can be considered fulfilled.

Key agreement

DID Documents may present more than one verification method specified in them. A specific standard verification method is required to maintain compatibility between the parties involved. This means that the sender and the recipient must use the same set of keys for encryption and/or signing purposes to have a successful message exchange through DIDComm. Nonetheless, addressing this requirement is unproblematic because the DID specification already provides a recommendation for this. The *keyAgreement* verification relationship is intended to provide the keys, which allows an entity to confidentially share information with the DID-subject using encryption [10]. Even though it is possible to add an extra verification relationship called DIDComm or ActivityPub that works in conjunction with our previously defined ActivityPub service, this work will comply with the recommendation using the *keyAgreement* key.

Access to private key

The ActivityPub server requires access to the private key of the selected verification method. The reason is, that creating a JWS, as well as decrypting a JWE requires the private key of the

user performing the action. To address this requirement, the private key of the user will be stored in the Mastodon server. The implications of this decision will be discussed in the 5.4

JWA Compatibility

The ActivityPub server must be able to support the keys and the cryptographic algorithms, which the JWA includes. This means having any library that can parse them and perform signing, signature verification, encryption, and decryption. This requirement raises the complexity level in implementing DIDComm. During the development of this thesis, there were no libraries that implemented the *ECDH-1PU* algorithm. For this reason, this design will implement the nested JWT alternative explained in section 2.5, given that various libraries support JWS and JWE in different programming languages. In addition, to prevent any further compatibility problems on the key generation side, the common widely-supported RSA key generation algorithm will be implemented, which is also supported for JWS and JWE creation [40]. This results in the following algorithm selection JWS tokens will use an RSA 2048 key and the *RS256* hash algorithm, and the JWE tokens the *RSA-OAEP*[1] algorithm for key management with the *A128CBC-HS256*[2] algorithm for encryption.

3.4.4 Payload Definition

Having addressed these requirements, it is now time to define how ActivityPub and DIDComm will work together. As part of this proposal, further changes to the ActivityPub protocol itself are not in scope. However, extending it and removing the dependency on the HTTP protocol for its communication is still intended. Therefore, encapsulation rather than modification of ActivityPub within DIDComm allows for a modular approach that keeps both protocols independent from each other. The best approach would be to follow DIDComm's schema of a nested JWT, where a plain-text message is encapsulated by a JWM, following a further encapsulation inside a JWS, and so on. The result consists of our *activity* object being used as the *body* of the JWM, as shown in figure 3.2.

Figure 3.2: Proposed JWT structure using DIDComm and ActivityPub

[1] https://www.rfc-editor.org/rfc/rfc7518#page-14
[2] https://www.rfc-editor.org/rfc/rfc7518#page-22

The final nested JWT consists only of a JWE token, which is the data that will be sent. A JWE offers two types of serializations. The first one is a compact one, where all elements are concatenated in a single string separated by dots [38], as shown in listing 3.8.

```
BASE64URL(UTF8(JWE Protected Header)) || '.' ||
BASE64URL(JWE Encrypted Key) || '.' ||
BASE64URL(JWE Initialization Vector) || '.' ||
BASE64URL(JWE Ciphertext) || '.' ||
BASE64URL(JWE Authentication Tag)
```

Listing 3.8: JWE compact serialization [38]

The compact serialization was conceptualized in a way, that it could be used in fields with a length limit, such as HTTP headers. On the contrary, the second one offers a JSON format, which is better suited to be sent as the body of an HTTP request rather than a plain-text token. For this reason, the payload should follow the JWE JSON serialization 3.9. Nevertheless, if the library used to create the JWE does not support the JSON format, or fails to provide access to the components of the JSON format, then the compact serialization can be implemented using the format specified in listing 3.10.

```
{
  "protected":"<integrity-protected shared header contents>",
  "unprotected":<non-integrity-protected shared header contents>,
  "recipients":[
    {
      "header":<per-recipient unprotected header 1 contents>,
      "encrypted_key":"<encrypted key 1 contents>"
    },
  "aad":"<additional authenticated data contents>",
  "iv":"<initialization vector contents>",
  "ciphertext":"<ciphertext contents>",
  "tag":"<authentication tag contents>"
 }
```

Listing 3.9: JWE JSON serialization [38]

```
{
  "payload": "JWE with compact serialization"
}
```

Listing 3.10: Alternativ payload structure

To make this architecture possible, Mastodon will require the following additions. First, a class *JWM*, with a setter to assign the activity object to its *body*. Second, a service class called *DIDCommService*, with the methods shown in table 3.2. This service will be responsible for signing, encrypting outgoing messages, and decrypting and verifying signatures of incoming messages.

DIDComm Service Class	
Function	**Description**
initialize	Takes the parameters *payload, from, to,* and *direction* and makes them instance variables. Parameter *direction* can be *incoming* or *outgoing*.
getKey	It fetches a private or public key based on an account
setFromAccount	If the message is *incoming*, sets the account based on the owner of the inbox URL
encrypt!	Encrypts the payload using a public key
encryptedPayload	Validates that all elements are present and then calls *encrypt!* to create the JWE
decrypt!	Decrypts a JWE using a private key
signedPayload	Creates a JWS using a private key
verifiedPayloadAndSigner	Verifies the JWS using a public key. Returns the decoded payload, and the account that signed it.

Table 3.2: Function list for a DIDComm Service class

Finally, a *DIDCommHelper* class will be implemented in the controller area, which is the one in charge of processing incoming requests to the server. This helper will check the incoming request for the *application/didcomm-encrypted+json* header. If present, it will send the payload to the *DIDCommService* to set the account of the receiver, decrypt the data, verify the signature, and then forward the Activity for its standard processing. Otherwise, it will allow the unmodified payload to be processed normally.

3.4.5 CRUD Operations on a DID

Creating a DID is rather a simple task. However, finding a DID method that allows CRUD operations to add the service endpoint and a *keyAgreement* key to the DID Document, without needing to pay any GAS or any other kind of fees, is not. The following options were tested. MATTR[3]offers creating DIDs using *did:key*, *did:web* and *did:ion* methods. However, they do not allow creating their keys or accessing private keys, which is necessary according to the first requirement. *did:ion* offers a set of tools[4] to perform CRUD operations in a self-created DID and DID document. These tools are bundled in a library called ION.js, which wraps the SDK and provides an interface to interact easily with the components of ION. However, even though the *update* operation is allowed, it was not possible to fetch a previously created DID and then update it, which was a necessary step. More users have encountered this issue[5], but so far, it

[3] https://mattr.global
[4] https://github.com/decentralized-identity/ion-tools
[5] https://github.com/decentralized-identity/ion-tools/discussions/25

has not been addressed by the developers.

An alternative to ledger-based DIDs is using the *did:web* method. This method allows hosting the DID Document on any server, thus giving the server-owner full control. Nonetheless, the discovery process of this type of DID relies heavily on DNS because the DID resolver makes a GET request to the *.well-known/did* endpoint of the domain in the DID to retrieve the DID document. This dependency on the domain would prevent achieving the goal of independence of centralized services.

Another DID method researched was the Uport-developed *did:ethr*. Uport is now divided into two projects, namely Serto[6] and Veramo[7]. Each one of them offers a decentralized identity solution. On the one hand, Serto provides a platform in the AWS Marketplace that can be easily deployed and would allow a user to create and manage DIDs from the *did:ethr* method. Unfortunately, after failing to deploy the EC2 instance and contacting Serto's developers, it turned out it was temporarily not working. On the other hand, Veramo's typescript-based API allows users to manage DIDs not only in the Ethereum main network but also in other test networks such as Ropsten and Rinkeby. This allows making CRUD operations to DIDs without incurring costs. Veramo provides a setup guide[8], where the only thing needed externally is an Infura[9] account to use as a Web3 Provider. Using Veramo's API will allow the creation of two different DIDs, and add the respective public keys and service endpoints.

Finally, to test this design the following elements are required. First, two DIDs for Alice and Bob respectively. Second, two servers running the Mastodon instance with the presented classes and methods implemented. One server will host the account for Alice, and the other for Bob with the respective DIDs as usernames. Third, both DID documents need to be updated with the service endpoint and the public key of the *keyAgreement* keypair they were provided by the Mastodon instance. With all these elements in place, Alice can send a private message to Bob's DID, which will trigger the whole flow this design presented. Figure 3.3 provides an updated diagram using the proposed DID and DIDComm-based flow.

[6] https://serto.id
[7] https://veramo.io
[8] https://veramo.io/docs/node_tutorials/node_setup_identifiers
[9] https://infura.io

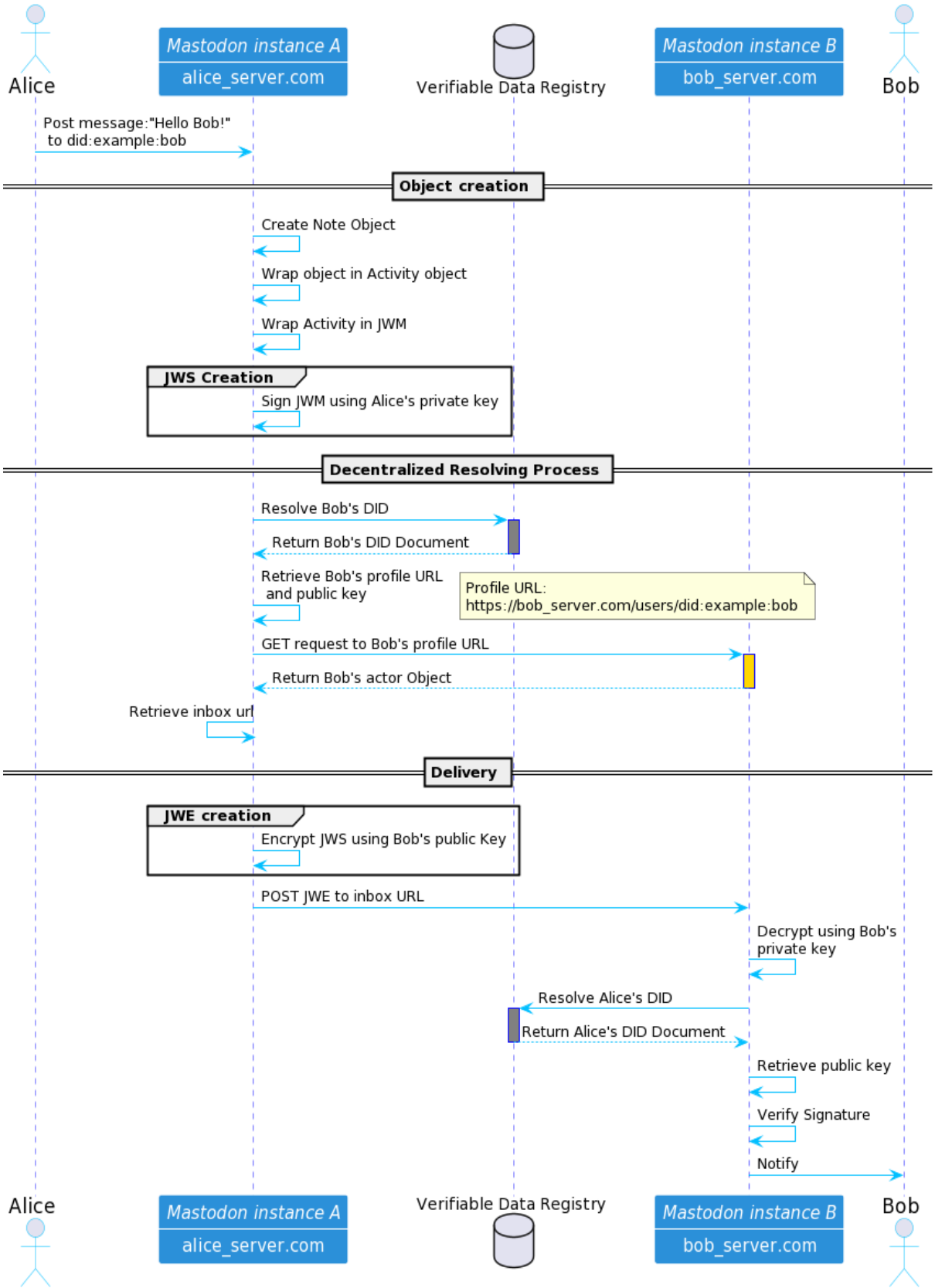

Figure 3.3: DID and DIDComm flow for use case mentioned in 3.2

4 Implementation

The design presented in 3 has been implemented to provide a proof of concept. This chapter discusses the implementation of the proposed design. Furthermore, it describes how to install, build and run the Mastodon prototype, as well as the steps needed to create the DIDs and DID documents.

The source code[1] of Mastodon is open source and accessible for everyone to download. The core backend was implemented using the Model-View-Controller framework Ruby on Rails[2], based on the Ruby programming language. The backend manages a Postgres[3] database, implements the ActivityPub server, provides a REST API for the frontend, and serves some web pages. The frontend of Mastodon was complemented using the React[4] framework, which manages most of the dynamic parts of the interface. Finally, Nginx serves as the reverse proxy for the Mastodon instance. Mastodon provides a docker-compose configuration file to facilitate the deployment process. However, it was extended with the DID resolver services for the prototype. The services inside the extended docker-compose file are described in table 4.1.

Service name	**Description**
Web	Mastodon Rails application
DB	Postgres database
Redis	In-memory data store for caching and streaming
ElasticSearch (es)	Search engine for accounts or tags
Streaming	Mastodon's React application
Sidekiq	Tool to perform asynchronous processing
*DID Resolver	Service to resolve DIDs
*Uni-resolver-driver-did-uport	Driver for the did:ethr method

Table 4.1: Mastodon services. *Only present in the Prototype

[1] https://github.com/mastodon/mastodon
[2] https://rubyonrails.org/
[3] https://www.postgresql.org/
[4] https://reactjs.org

4.1 Proof of concept

The source code of this thesis prototype is available on GitLab and can be accessed under the following link: `https://gitlab.com/Lisztos/mastodon`. It was implemented using two different Mastodon instances. The first one was a Linux server with Ubuntu 20.04, 2 CPU cores, 4GB RAM, and 80 GB storage capacity provided by the cloud provider Linode[5]. The TU Berlin provided the second one, which was mainly used for debugging requests and testing the resolving processes in federated communication. Both domains used for the servers are *lisztos.com* and *tawki.snet.tu-berlin.de*. Amazon Simple Email Service (SES) was selected as the email delivery service, and Amazon S3 was used for storing the profile images of the servers.

OpenSSL[6] is the library Mastodon uses for generating the RSA keys used for the HTTP and JSON-LD signatures. This library wraps the OpenSSL project toolkit[7] and provides a wide range of services such as key management, encryption, decryption, and certificate management. To create signed and encrypted messages, the JWT library[8] for Ruby on Rails was selected as it allows the spec-compliant creation of JWS and JWE tokens.

Replacing the standard username with a DID was unproblematic in Mastodon. The way Mastodon validates the username format is through the regular expression shown in listing 4.1. Additionally, it has a length constraint of 30 characters. For the prototype, the DID-syntax-compliant regular expression shown in listing 4.2 was used, as well as an extended maximum length of 85 characters.

```
1 USERNAME_REGEX = /[a-z0-9_]+([a-z0-9_\.-]+[a-z0-9_]+)?/i
```

Listing 4.1: Mastodon username regex

```
1 DID= /did+:+[a-z0-9_]+([a-z0-9_\.-]+[a-z0-9_]+)?:[A-Za-z0-9\.\-\:\_\#]+/i
```

Listing 4.2: Regex for DID used in the prototype

DID creation

Using Veramo's API mentioned in subsection 3.4.5, two DIDs were created in the Ropsten network for Alice and Bob, respectively. An account on each server was created with the DIDs. Figure 4.1 shows Alice's created account.

- **Alice:** did:ethr:ropsten:0x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d
- **Bob:** did:ethr:ropsten:0x03117951c6011b4a46f11a67fc7f67f746a7ad84daaae69623db833ddd56397c37

[5] https://linode.com
[6] https://github.com/ruby/openssl
[7] https://www.openssl.org/
[8] https://github.com/jwt

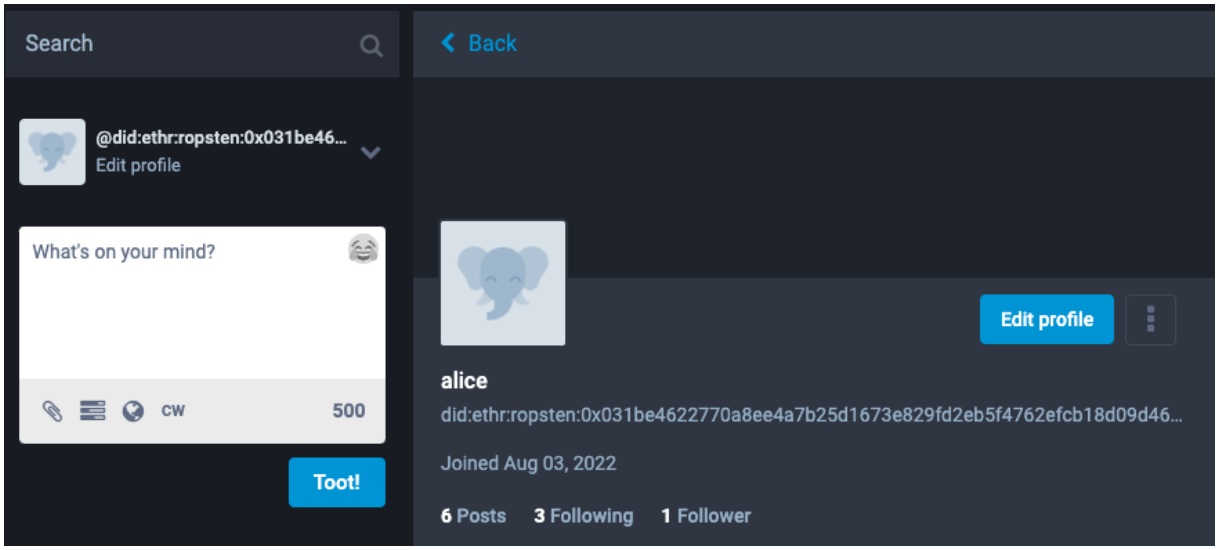

Figure 4.1: Alice's profile using DID as username in the Mastodon prototype.

DID Document modification

Adding the ActivityPub service endpoint to the DID documents of Bob and Alice Veramo requires the following parameters. DID, service, and the options for the Web3 provider, as shown in figure 4.3. The service must have a type, a service endpoint, and a description. A service's id is optional, as the Web3 Provider will overwrite it.

```
const service_args= {
  did: <alice DID>,
  service: {
    id: 'ActivityPub', // This field will be overwritten
    type: "ActivityPub",
    serviceEndpoint: "http://lisztos.com/users/" + <alice DID>,
    description: "DIDComm enabled ActivityPub Actor"
  },
  options: {
    gas: 100_000, // between 40-60000
    ttl: 60 * 60 * 24 * 365 * 10 // make the service valid for ~10 years
  }
}
```

Listing 4.3: Parameters to add a service in Veramo

The next step was adding the public key to the DID document. Mastodon generates an RSA keypair for an account when the account is created. After creating the accounts for Alice and Bob, it was possible to retrieve the public key, which needed to be added to the DID document. Veramo's API offers a way to add different kinds of cryptographic keys to a DID document, however, the only available key types that could be added were the *Ed25519*, *Secp256k1*, and the *X25519* from the *Elliptic Curve Cryptography* (ECC) family of cryptosystems. This represented a considerable complication because the already selected library for the JWT tokens did not have

support for these types of keys. Luckily, after researching possible workarounds, one of the main developers of the Veramo project assisted and developed a way to *force* an RSA key into the DID document. The parameters required to add a key are a type, the *kid*, Key ID; and the public key of the RSA key in Base16 encoding, as shown in listing 4.4. Furthermore, the final DID document for Alice can be seen in appendix 3.

```
1
2  const rsaKeyPair = {
3    privateKey: "-----BEGIN PRIVATE KEY----.....",
4    publicKey: "-----BEGIN PUBLIC KEY-----\nMII...."
5  }
6
7
8  const key_args{
9    did: <Alice DID>,
10   key: {
11     type: "RSA" as TKeyType,  // Veramo doesn't really know about RSA
     keys, but it doesn't matter in this case if we coerce it.
12     kms: 'local',
13     kid: 'my RSA key',
14     publicKeyHex: u8a.toString(u8a.fromString(rsaKeyPair.publicKey, 'utf
     -8'), 'base16')
15   },
16   options: {
17     encoding: 'pem',
18     ttl: 60*60*24*365*10, // the key will be valid for 10 years
19     gas: 100000 // use 100,000 gas at most for this transaction
20   }
21 }
```

Listing 4.4: Adding an RSA key using Veramo

Finally, an example of a successful direct message sent with DIDComm can be seen in fig. 4.2.

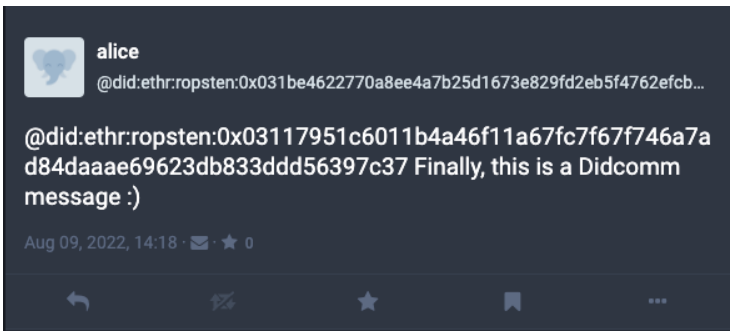

Figure 4.2: Alice's private message sent to Bob's server using proposed design

4.2 Mastodon

4.2.1 Requirements

The requirements to run a mastodon server are:

- Ubuntu 20.04 or Debian 11 for the operating system
- Domain name
- Email delivery service
- Object storage service (optionally)

Software requirements include:

- Nginx
- Docker
- Docker-Compose

4.2.2 Configuration

For external configuration, it is required that the domain name points to the IP address of the server. Furthermore, the accounts and providers for the Email delivery service or the object storage service must be set up in advance. Additionally, Nginx[9] is used to serve the Mastodon instance. Unfortunately, Nginx was not containerized and it must be manually configured. For complete instructions to configure Nginx see appendix 6.1

4.2.3 Build

To build the source code of the prototype it is first required to add an *env* file with the necessary environment variables for development to the root folder. An example file can be found in appendix 6.1, however, it is necessary to fill it up with the required credentials. Next, the image from the web service needs to be manually built with the following command. The *-f* flag specifies to use the custom *Dockerfile* and the *-t* flag adds a name and tag to the image.

```
1    docker build -f Dockerfile.dev -t mastodon:dev .
2
```

Listing 4.5: Building the development Dockerfile

4.2.4 Run

After building the development image for the rails application, it is now possible to pull the images for the other services from the docker registry and get every container running using the commands shown in listing 4.6.

[9] https://nginx.com

```
// Create and start containers
docker-compose up -d

// Restart NGINX
sudo systemctl restart nginx

// When everything is running, we need to run the migrations
docker-compose exec web rails db:migrate
```

Listing 4.6: Starting all services of Mastodon

Visiting the provided domain using HTTPS will show the welcome page of Mastodon. In order to test the federated communication, this process should now be repeated using a different server. Having two different domain names was required for the prototype to test the whole DNS-based resolving process. However, the setup might also work using the raw IP address as the default domain.

4.3 DIDs

In order to create a DID, the source code of the prototype includes a */veramo_agent* folder with all the needed methods to perform CRUD operations to DIDs from the *did:ethr* method. The setup was taken from Veramo's guides[10], and the methods for the CRUD operations were written based on the API reference [11].

4.3.1 Requirements

Software requirements to build the Veramo agent include:

- Node v14+
- Yarn

4.3.2 Configuration

First, an *.env* file that contains the variable: *INFURA_PROJECT_ID="example-infura-project-id"* from an Infura[12] account is required. This project ID has to match the Ethereum network desired. The default selected in the setup is *Ropsten*.

4.3.3 Build

To build the project, the only command needed is *yarn install* in the root of the */veramo_agent* folder.

[10] https://veramo.io/docs/node_tutorials/node_setup_identifiers/
[11] https://veramo.io/docs/api/core
[12] https://infura.io/

4.3.4 Run

A set of *scripts* are provided to facilitate the use of the CRUD methods. See table 4.2 for a detailed view. However, to make any changes to a DID, it is first necessary to fund the Ethereum account to which the DID is anchored. This account can be found in the DID document, with the key *blockchainAccountId* of the first verification method, as shown in listing 4.7. It follows the format *eip155:<network id>:<0x ethereum address>*. If the DID was created using a test network, the address can be funded using any Faucet[13]. Otherwise, Ether has to be transferred from a real account. Finally, the prototype only supports parsing RSA keys from the DID document. To further understand how adding an RSA key works in Veramo, please refer to:
`https://gist.github.com/Lisztos/eade1f9199d46392f7c7d1f9bbfe7a3b`

Command	Parameters	Description
yarn id:create		Creates a new DID
yarn id:list		List the created identifiers by this agent
yarn id:resolve_did	DID	Resolved the DID to its the DID document
yarn id:add_service	DID, Service 4.8	Adds a new service to the DID document
yarn id:remove_service	DID, Service ID	Removes the specified service
yarn id:add_key	DID, Key	Adds a new key to the DID document.
yarn id:remove_key	DID, Key ID	Removes the specified key

Table 4.2: CRUD operations for *did:ethr* DIDs

[13] https://faucet.egorfine.com/

```
{
"@context": [
  "https://www.w3.org/ns/did/v1",
  "https://w3id.org/security/suites/secp256k1recovery-2020/v2"
],
"id": "did:ethr:ropsten:0x031be462277...",
"verificationMethod": [
  {
    "id": "did:ethr:ropsten:0x031be462277...#controller",
    "type": "EcdsaSecp256k1RecoveryMethod2020",
    "controller": "did:ethr:ropsten:0x031be462277...",
    "blockchainAccountId": "eip155:3:0
 x577361D41748c83ab328E90a51054712Fe49e211"  // Ethereum Address
  },
],
{...}
}
```

Listing 4.7: Controller address inside the DID document

```
const service= {
  did: <DID>,
  service: {
    id: 'ActivityPub', // This field will be overwritten
    type: "ActivityPub",
    serviceEndpoint: "http://your-domain.com/users/" + <DID>,
    description: "DIDComm enabled ActivityPub Actor"
  },
  options: {
    gas: 100_000, // between 40-60000
    ttl: 60 * 60 * 24 * 365 * 10 // make the service valid for ~10 years
  }
}
```

Listing 4.8: Parameters to add a service in Veramo

5 Evaluation

The design for a DIDComm-enabled ActivityPub protocol for federated social networks strives to take advantage of the features that both DIDs and DIDComm provide. The following chapter evaluates the proposed design to assess the achievement level of these features. In addition, the evaluation compares the state-of-the-art ActivityPub implementation, the extended versions mentioned in 2.3 and the proposed design.

5.1 Decentralization

5.1.1 Trust Encryption

ActivityPub relies on HTTPS to provide confidentiality and data integrity. This comprises a dependence on the authority issuing the certificates that make the TLS/SSL encryption possible. By implementing DIDComm, the encryption trust gets transferred to the decentralized identifiers. Thus removing dependence from any third parties. By using a nested JWT in the proposed design, a signature and encryption layer were provided. The trust in these extra layers relied solely on the DIDs communicating, and no other party was needed to ensure it.

However, the deeply nested use of HTTPS in Mastodon's codebase did not allow testing communication using only HTTP. In addition, taking advantage of DIDComm's transport independence was not possible, due to the complexity of Mastodon. Therefore, the foundation to use any communication method with DIDComm was set, but no actual implementation could be reached.

5.1.2 Resolving Process

The current implementation of Mastodon relies on the Webfinger protocol, which itself relies on the DNS, to resolve *username@domain* usernames to an actor object. The DNS builds the foundation of daily Internet activity. As a centralized service, being so involved in the critical part of the Internet makes it a valuable target for attacks [43]. A single institution controls it, the ICANN[1], that until 2016 was under the control of one single country [44]. By integrating ledger-based DIDs, this dependency on Webfinger and the DNS was only partially removed. Resolving a DID does not require the DNS to be successful. However, the service endpoint in the DID document still includes the domain of the Mastodon instance where it resides. To retrieve the actor object of the user being looked up, an HTTP GET request to the service endpoint is necessary. This means that a DNS resolution will still take place. On the contrary, the approach of the coauthors of ActivityPub and the DIDs mentioned in 2.3 completely removes

[1] https://www.icann.org/

the DNS dependency by using The Router Onion *(Tor)* to host the ActivityPub server [8]. This indicates that if Mastodon were running in a DNS-free space, the design would achieve a fully-decentralized resolving process.

5.1.3 Identity Management

As mentioned in 1, Mastodon uses basic password authentication to create accounts. With this, the account is restricted to a specific server. The user has no other way to register to a different server, and the identity lives as long as the instance keeps running. On the contrary, in the proposed design, the use of DIDs brings an SSI approach where the user creates his identity independently from Mastodon. In the design, the DIDs are anchored to a test network of Ethereum. Consequently, their existence and validity will persist even if the Mastodon instance ceases to exist, eliminating the *single-point-of-failure* of the identity itself. On the downside, as proved in this thesis, modifying a DID document can present complications depending on the DID method. Modifications could also imply costs when not using test blockchain networks. All of this presents extra overhead for the average user, reducing the usability of such a design. Nonetheless, independent of the overhead, with the proposed design Mastodon was interoperate with an SSI ecosystem.

5.2 Security

Mastodon implemented HTTP Signatures to add non-repudiation and preserve integrity against tampering with the messages sent within the federation. This approach fulfills the same goals as the JWS token used in the proposed design. Both methods follow the same idea and use public key cryptography to perform the signature. Furthermore, the same key generation algorithm is being used by both. Nonetheless, the downside of HTTP signatures is that they are limited to the HTTP protocol, whereas the JWS has no constraints. Moreover, the JSON-LD signatures provide an HTTP-independent manner to provide non-repudiation. However, the superiority of one over the other is still an open discussion [2]. JSON-LD signatures offer more flexibility and thus scalability for global decentralized networks due to their compatibility with JSON-LD. In contrast, JWTs provide a straightforward way to express data with low overhead [45]. As Mastodon only implements them in particular cases, it is not possible to reach a verdict. For this thesis, the conclusion is that both the Mastodon and the proposed implementation offer the same level of non-repudiation when using HTTP as the transport protocol.

5.3 Privacy

The user has no control over his data by using centralized identity management. DIDs, on the other hand, implement natively the 7 Foundational Principles of *Privacy by Design* [46]. This gives consumers control over their personal information, including choosing what to disclose and what not. [10]. The proposed design requires the profile URL of the user to be included in the DID document to allow the decentralized discovery of his actor object. Although the parameter inside the profile URL does not include the user's personal information, like a human-

[2] https://w3c.github.io/vc-imp-guide/#benefits-of-json-ld-and-ld-proofs

friendly username, anyone can still access this endpoint and get all the information from the actor object. This endpoint raises a significant privacy concern and opens the question of how to be discoverable while keeping privacy. A possibility could be adding the inbox URL instead of the profile URL in the DID document. This possibility would provide a direct endpoint to interact via ActivityPub with the user. This approach might prove to be better than the proposed design, not only by improving privacy but also by shortening the number of requests needed to resolve an account and find the inbox URL for the use case defined in section 3.2. Nonetheless, there are many other use cases where having just the inbox URL might be limiting. For example, when *Alice* wants to follow *Bob*. An extra service endpoint with the *follow* URL would be necessary to achieve this with the same approach, and so on with other endpoints until we get to the same approach of Webber and Sporny, mentioned in section 2.3.

5.4 Confidentiality

As explained before, the only confidentiality provided by Mastodon's implementation is the use of HTTPS. By enabling DIDComm, it is possible to encrypt the payload before sending it and decrypt it after it has arrived at its target server, extending the scope of confidentiality beyond the transport layer. Theoretically, only the DIDs at the respective endpoints could see the plain text message. Nonetheless, the confidentiality level reached by DIDComm and the JWE standard in the proposed design has not yet achieved complete end-to-end encryption. The reason is that the Mastodon instance must have access to the private key of the DID subject to decrypt the payload and display the message to the user, as mentioned in the DIDComm requirements in subsection 3.4.3. This imposes a significant risk because the private key is no longer under the user's control. The administrator of the Mastodon instance would have access to the plain-text private key, and some security countermeasures like key rotation would not be able to counter this.

5.5 Usability

The standard username format in Mastodon is compact, human-readable, and suitable to use in a social network context. As a consequence of Zooko's triangle, DIDs are not human-meaningful by design [47] [10]. Due to the long format of a DID, it proved to be difficult to read interactions with other users when a DID is present, and mentioning a user with the DID would take a significant amount of the 500-character limit. A workaround was to use the *displayName* property of the actor object, allowing a human-meaningful descriptor for a user, as shown in figure 5.1.

Furthermore, the overhead mentioned in 5.1.3 to create and update a DID contributes to a less user-friendly experience, that could keep users in easy-to-use centralized services.

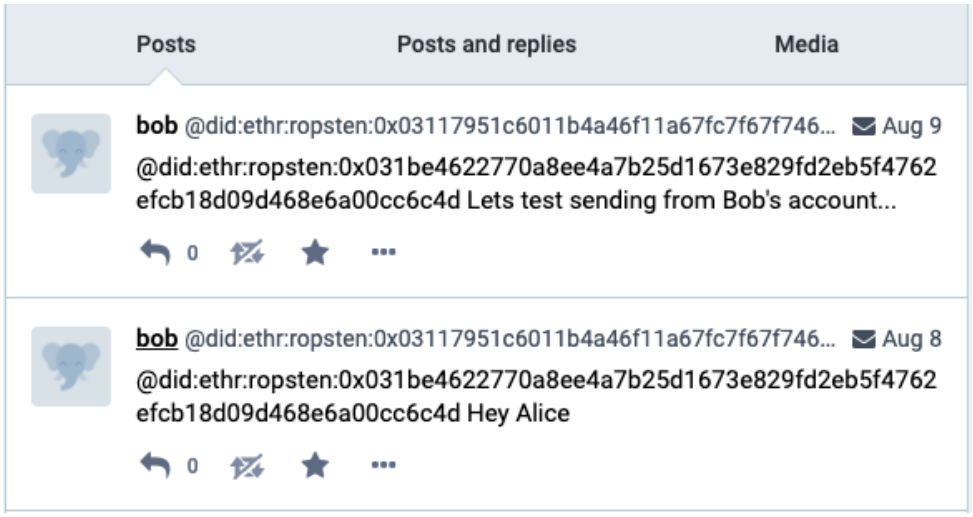

Figure 5.1: Example of mentioning another user in a *toot* using DIDs.

6 Conclusion

This thesis presented a design to integrate DIDs and enable DIDComm Messaging v2 into an ActivityPub-based Online Social Network (OSN). The OSN Mastodon was chosen as the representative ActivityPub implementer, where DIDs could be introduced and used to enable DIDComm. The integration of DIDs into ActivityPub was discussed, concluding with the substitution of Mastodon's standard usernames with DIDs. Consequently, a way to preserve the ActivityPub protocol while using DIDComm was explained, resulting in the encapsulation of ActivityPub as the content of a JWM.

Different DID methods were researched to find a way to create DIDs and modify the DID documents to include a new verification method and a service endpoint. Tested methods included *did:ion, did:web* and *did:key*. As the selected DID method for the prototype, *did:ethr*, did not support adding RSA keys to a DID document, a custom way was developed with the help of one of the lead developers of the SSI platform provider *Veramo*. Using the RSA keys and the service endpoints in two DID documents, Mastodon's process to resolve an account in federated communication was modified to partially remove any dependency on centralized third parties and allow a decentralized communication flow when sending a private message. This decentralization process included the following features. First is the ability to send and receive encrypted payload without needing HTTPS for encryption. The second is finding other users based on their DIDs by retrieving their DID documents stored in Ethereum's test network Ropsten instead of using Mastodon's default Webfinger. The DID resolver from the DIF was deployed to achieve the latter. Furthermore, nested JWTs were implemented to provide non-repudiation, message integrity, and encryption using JWM and the JWS and JWE standards.

The proposed design took advantage of most of the features of DIDs and DIDComm. However, it is still not entirely independent of the centralized DNS. In addition, risks and deficiencies were found. For instance, the private key being stored in Mastodons' database and the still present administrators' access to plain-text private messages. Nonetheless, it proved to be a successful implementation of DIDs and DIDComm in a real-world social network.

6.1 Future Work

The evaluation has shown that although achieving this thesis's goal, some issues still need to be addressed. For example, the risks imposed by sharing a private key with an ActivityPub server, or Mastodon instance, could be solved using the self-hosted approach of Göndör et al. called *Blade* [48]. If every user had its own Mastodon server, storing private data would not represent a problem. Nevertheless, this implies a considerable overhead for an average user and goes against the lightweight and easy-to-deploy idea that *Blade* proposes. Alternatively, taking advantage of the marketplace implemented in *Blade*, a module for ActivityPub that can interact with a Mastodon instance while keeping private data under the user's control, could

prove to be a better option.

Furthermore, a registration process that takes advantage of the *authentication* property of a DID document has yet to be implemented. Spruce[1] is working on a similar idea, where a user can *sign in* to a service using their Ethereum-based *ENS* domain.

Finally, further development for the DID-based Mastodon prototype is intended. The goal is to be able to participate in the federation while still supporting DIDs by adding backward compatibility. In this manner, DIDs can be introduced to Mastodon and promote their further adoption. Following this idea, a DIDComm endpoint will be adapted so that users can send private messages, following the design proposed in this thesis.

[1] https://www.spruceid.com/

List of Tables

List of Figures

Bibliography

[1] A. Tobin, D. Reed, F. P. J. Windley, and S. Foundation, "The inevitable rise of self-sovereign identity," p. 24, 2017.

[2] J. MacInnis, "A brief history of the password and why it matters," Aug 2019. [Online]. Available: https://blog.hidglobal.com/2019/08/brief-history-password-why-it-matters

[3] D. Parkins, "The world's most valuable resource is no longer oil, but data," May 2017. [Online]. Available: https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data

[4] A. Holmes, "533 million facebook users' phone numbers and personal data have been leaked online," Apr 2021. [Online]. Available: https://www.businessinsider.com/stolen-data-of-533-million-facebook-users-leaked-online-2021-4

[5] S. Shim, G. Bhalla, and V. Pendyala, "Federated identity management," *Computer*, vol. 38, no. 12, pp. 120–122, 2005.

[6] C. Allen, Apr 2016. [Online]. Available: http://www.lifewithalacrity.com/2016/04/the-path-to-self-soverereign-identity.html

[7] C. Lemmer-Webber, J. Tallon, A. Guy, and E. Prodromou, "1.1 social web working group," Jan 2018. [Online]. Available: https://www.w3.org/TR/activitypub

[8] C. Webber and M. Sporny, "Activitypub: From decentralized to distributed social networks," Dec 2017. [Online]. Available: https://github.com/WebOfTrustInfo/rwot5-boston/blob/fcbc33835c1b76b7526a8a82cd9cac9c23828711/final-documents/activitypub-decentralized-distributed.pdf

[9] E. Rochko, "Remove keybase integration by gargron · pull request 17045 · mastodon/-mastodon," Nov 2021. [Online]. Available: https://github.com/mastodon/mastodon/pull/17045

[10] M. Sporny, D. Longley, M. Sabadello, D. Reed, and O. Steele, "Decentralized identifiers (dids) v1.0," Aug 2021. [Online]. Available: https://www.w3.org/TR/did-core/

[11] "Decentralized identifiers (dids) v1.0 becomes a w3c recommendation," *W3C*, Jul 2022. [Online]. Available: https://www.w3.org/2022/07/pressrelease-did-rec.html.en

[12] T. Celik, E. Prodromou, and A. LeHors, "Social web working group chart," Jul 2014. [Online]. Available: https://www.w3.org/wiki/Socialwg

[13] "Activity streams 2.0," May 2017. [Online]. Available: https://www.w3.org/TR/activitystreams-core/

[14] T. Berners-Lee, "Linked data," Jul 2006. [Online]. Available: https://www.w3.org/DesignIssues/LinkedData.html

[15] "https://www.w3.org/tr/social-web-protocols/," Dec 2017. [Online]. Available: https://www.w3.org/TR/social-web-protocols/

[16] J. Holloway, "What on earth is the fediverse and why does it matter?" Oct 2018. [Online]. Available: https://newatlas.com/what-is-the-fediverse/56385/

[17] A. Raman, S. Joglekar, E. De Cristofaro, N. Sastry, and G. Tyson, "Challenges in the decentralised web: The mastodon case," 2019. [Online]. Available: https://arxiv.org/abs/1909.05801

[18] C. Lemmer-Webber, "Mastodon launches their activitypub support, and a new cr!" Sep 2017. [Online]. Available: https://activitypub.rocks/news/mastodon-ap-and-new-cr.html

[19] B. Fitzpatrick and D. Recordon, "Thoughts on the social graph," Aug 2007. [Online]. Available: http://bradfitz.com/social-graph-problem/

[20] S. Tilley, "One mammoth of a job: An interview with eugen rochko of mastodon," Jul 2018. [Online]. Available: https://medium.com/we-distribute/one-mammoth-of-a-job-an-interview-with-eugen-rochko-of-mastodon-23b159d6796a

[21] M. Zignani, C. Quadri, S. Gaito, H. Cherifi, and G. P. Rossi, "The footprints of a "mastodon": How a decentralized architecture influences online social relationships," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2019, pp. 472–477.

[22] M. Cavage and M. Sporny, "Signing http messages," Oct 2019. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-cavage-http-signatures-12

[23] M. Nottingham, "Rfc 8615 - well-known uniform resource identifiers (uris)," May 2019. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8615

[24] B. Cook, "Rfc 6415 - web host metadata," Oct 2011. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6415

[25] E. Rochko, "Webfinger," Jan 2020. [Online]. Available: https://docs.joinmastodon.org/

[26] P. Jones, G. Salgueiro, M. Jones, and J. Smarr, "Rfc 7033 - webfinger," Sep 2013. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7033

[27] N. Naik and P. Jenkins, "Sovrin network for decentralized digital identity: Analysing a self-sovereign identity system based on distributed ledger technology," in *2021 IEEE International Symposium on Systems Engineering (ISSE)*, Sep 2021, p. 1–7.

[28] S. Conway, A. Hughes, M. Ma, J. Poole, and M. Riedel, "A did for everything," p. 24, 2019.

[29] A. Preukschat and D. Reed, *Self-sovereign identity decentralized digital identity and verifiable credentials*. O'REILLY MEDIA, 2021.

[30] G. Weston, "What is a layer 2 protocol in blockchain?" Aug 2022. [Online]. Available: https://101blockchains.com/layer-2-protocols-blockchain/

[31] D. Longley, D. Zagidulin, and M. Sporny, "The did:key method v0.7," May 2022. [Online]. Available: https://w3c-ccg.github.io/did-method-key/

[32] R. Jones and D. Boswell, "Hyperledger - hyperledger," Jul 2022. [Online]. Available: https://wiki.hyperledger.org/

[33] A. Lusard, A. Le Hors, B. Muscara, D. Boswell, and C. Zsigri, "An overview of hyperledger foundation," Oct 2021. [Online]. Available: https://www.hyperledger.org/wp-

content/uploads/2021/11/HL_Paper_HyperledgerOverview_102721.pdf

[34] K. Young, "Understanding didcomm," Sep 2020. [Online]. Available: https://medium.com/decentralized-identity/understanding-didcomm-14da547ca36b

[35] "Didcomm messaging," May 2020. [Online]. Available: https://identity.foundation/didcomm-messaging/spec

[36] W. Abramson, A. J. Hall, P. Papadopoulos, N. Pitropakis, and W. J. Buchanan, "A distributed trust framework for privacy-preserving machine learning," in *Trust, Privacy and Security in Digital Business*. Springer International Publishing, 2020, pp. 205–220. [Online]. Available: https://doi.org/10.1007%2F978-3-030-58986-8_14

[37] M. Jones, J. Bradley, and N. Sakimura, "Rfc 7515: Json web signature (jws)," May 2015. [Online]. Available: https://datatracker.ietf.org/doc/rfc7515/

[38] M. Jones and J. Hildebrand, "Rfc 7516: Json web encryption (jwe)," May 2015. [Online]. Available: https://datatracker.ietf.org/doc/rfc7516/

[39] M. Jones, "Rfc 7517: Json web key (jwk)," May 2015. [Online]. Available: https://datatracker.ietf.org/doc/rfc7517/

[40] ——, "Rfc 7518: Json web algorithms (jwa)," May 2015. [Online]. Available: https://datatracker.ietf.org/doc/rfc7518/

[41] D. Hardman, "1e/ didcomm mythconceptions: Understanding the most misunderstood opportunity in ssi," Oct 2021. [Online]. Available: https://iiw.idcommons.net/1E/_DIDComm_Mythconceptions:_understanding_the_most_misunderstood_opportunity_in_SSI

[42] I. Martin, "Hugely popular ngl app offers teenagers anonymity in comments about each other," Jul 2022. [Online]. Available: https://www.forbes.com/sites/iainmartin/2022/06/29/hugely-popular-ngl-app-offers-teenagers-anonymity-in-comments-about-each-other/?sh=38a95a5153d1

[43] F. Carli, "Security issues with dns," *Retrieved October*, vol. 3, p. 2005, 2003.

[44] D. Lee, "Has the us just given away the internet?" Oct 2016. [Online]. Available: https://www.bbc.com/news/technology-37527719

[45] D. Chadwick, D. Longley, M. Sporny, O. Terbu, D. Zagidulin, and B. Zundel, "Verifiable credentials implementation guidelines 1.0," Aug 2022. [Online]. Available: https://w3c.github.io/vc-imp-guide/

[46] A. Cavoukian, "The 7 foundational principles implementation and mapping of fair information practices," Nov 2006. [Online]. Available: https://sites.psu.edu/digitalshred/2020/11/13/privacy-by-design-pbd-the-7-foundational-principles-cavoukian/

[47] Z. Wilcox-O'Hearn, "Zooko.com," Oct 2001. [Online]. Available: https://web.archive.org/web/20011020191610/http://zooko.com/distnames.html

[48] S. Göndör, H. Yildiz, M. Westerkamp, and A. Küpper, "Blade: A blockchain-supported architecture for decentralized services," 08 2022.

Appendices

Appendix: Listings

```
{
    "@context": [
        "https://www.w3.org/ns/activitystreams",
    ],
    "id": "https://mastodon.social/users/bob",
    "type": "Person",
    "following": "https://mastodon.social/users/bob/following",
    "followers": "https://mastodon.social/users/bob/followers",
    "inbox": "https://mastodon.social/users/bob/inbox",
    "outbox": "https://mastodon.social/users/bob/outbox",
    "featured": "https://mastodon.social/users/bob/collections/featured",
    "featuredTags": "https://mastodon.social/users/bob/collections/tags",
    "preferredUsername": "bob",
    "name": "Bob",
    "summary": "",
    "url": "https://mastodon.social/@bob",
    "manuallyApprovesFollowers": false,
    "discoverable": false,
    "published": "2022-01-18T00:00:00Z",
    "devices": "https://mastodon.social/users/bob/collections/devices",
    "publicKey": {
        "id": "https://mastodon.social/users/bob#main-key",
        "owner": "https://mastodon.social/users/bob",
        "publicKeyPem": "-----BEGIN PUBLIC KEY-----\n..."
    },
    "tag": [],
    "attachment": [],
    "endpoints": {
        "sharedInbox": "https://mastodon.social/inbox"
    },
    "icon": {
        "type": "Image",
        "mediaType": "image/jpeg",
        "url": "https://files.mastodon.social/accounts/avatars/107.jpg"
    }
}
```

Listing 1: Mastodon extended version of ActivityPub's actor object of user bob@mastodon.social

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    {
      "toot": "http://joinmastodon.org/ns#",
      "Device": "toot:Device",
      "Ed25519Signature": "toot:Ed25519Signature",
      "Ed25519Key": "toot:Ed25519Key",
      "Curve25519Key": "toot:Curve25519Key",
      "EncryptedMessage": "toot:EncryptedMessage",
      "publicKeyBase64": "toot:publicKeyBase64",
      "deviceId": "toot:deviceId",
      "claim": {
        "@type": "@id",
        "@id": "toot:claim"
      },
      "fingerprintKey": {
        "@type": "@id",
        "@id": "toot:fingerprintKey"
      },
      "identityKey": {...},
      "devices": {...},
      "messageFranking": "toot:messageFranking",
      "messageType": "toot:messageType",
      "cipherText": "toot:cipherText"
    }
  ],
  "id": "http://localhost:3000/f41c9aae-92fc-48bf-8d87-0606197f340a",
  "type": "Create",
  "actor": "http://localhost:3000/users/admin",
  "to": "http://localhost:3000/users/velda_moriette0",
  "object": {
    "type": "EncryptedMessage",
    "messageType": 0,
    "cipherText": "AwogNcm0y1HR42K07iyNs37uTlQ7jrX3n7...",
    "digest": {
      "type": "Digest",
      "digestAlgorithm": "http://www.w3.org/2000/09/xmldsig#hmac-sha256",
      "digestValue": "5f6ad31acd64995483d75..."
    },
    "messageFranking": "...",
    "attributedTo": {
      "type": "Device",
      "deviceId": "11119"
    },
    "to": {
      "type": "Device",
      "deviceId": "11876"
    }
  }
}
```

Listing 2: Example of encrypted message using Mastodon's E2EE API and extended JSON-LD vocabulary

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/suites/secp256k1recovery-2020/v2"
  ],
  "id": "did:ethr:ropsten:0
  x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d",
  "verificationMethod": [
    {
      "id": "did:ethr:ropsten:0
  x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d#
  controller",
      "type": "EcdsaSecp256k1RecoveryMethod2020",
      "controller": "did:ethr:ropsten:0
  x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d",
      "blockchainAccountId": "eip155:3:0
  x577361D41748c83ab328E90a51054712Fe49e211"
    },
    {
      "id": "did:ethr:ropsten:0
  x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d#delegate
  -1",
      "type": "RSAVerificationKey2018",
      "controller": "did:ethr:ropsten:0
  x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d",
      "publicKeyPem": "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFA
  ...npUZV+Q1FdWLiuFlKAwDt3DiD3\nNSz2b27Iga7wTSCoQ/
  DAhFAZ8KH8uCMrioIWFSKuYRgFOysbDfTT9Kpha8ST8K/H\nTQIDAQAB\n-----END PUBLIC KEY
  -----\n"
    }
  ],
  "authentication": [...],
  "assertionMethod": [...],
  "service": [
    {
      "id": "did:ethr:ropsten:0
  x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d#service-1
  ",
      "type": "ActivityPub",
      "serviceEndpoint": "http://lisztos.com/users/did:ethr:ropsten:0
  x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d"
    }
  ]
}
```

Listing 3: DID Document for Alicecreated with the *did:ethr* DID method. It includes an RSA public key and a service that points to her profile URL in the Mastodon instance *lisztos.com*

Appendix: Nginx Configuration

```
// First, clone the project
git clone https://gitlab.com/Lisztos/mastodon

// Quick install nginx
sudo apt update && sudo apt install nginx

// Copy configuration file from the repository sample to the Nginx folder
cp ~/root/mastodon/nginx.conf /etc/nginx/sites-available/

// Rename with your domain name
mv /etc/nginx/sites-available/nginx.conf /etc/nginx/sites-available/your-domain.com.conf

// Update the configuration file with your domain information
nano /etc/nginx/sites-available/your-domain.com.conf

// Activate the Nginx configuration
cd /etc/nginx/sites-enabled
ln -s ../sites-available/your-domain.com.conf
sudo systemctl stop nginx

// Open ports 80 and 443
sudo apt install ufw
sudo ufw allow Nginx Full

// Get SSL certificate using Let's encrypt
sudo apt install certbot
certbot certonly --standalone -d your-domain.com

// Start Nginx
sudo systemctl start nginx
```

Listing 1: Adding Nginx configuration file.

Appendix: Env File

```
# Federation
 This identifies your server and cannot be changed safely later
# ----------
LOCAL_DOMAIN=

# DEV ENV
# ---------
RAILS_ENV=development
NODE_ENV=development

# Redis
# -----
REDIS_HOST=redis
REDIS_PORT=6379

# PostgreSQL
# ----------
DB_HOST=db
DB_USER=mastodon
DB_NAME=mastodon_dev
DB_PASS=
DB_PORT=5432

# Sending mail
# ------------
SMTP_SERVER=
SMTP_PORT=
SMTP_LOGIN=
SMTP_PASSWORD=
SMTP_FROM_ADDRESS=
SMTP_REPLY_TO=

# File storage (optional)
# -----------------------
S3_ENABLED=true
S3_BUCKET=
AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=
```