Technische Universität Berlin

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 20, 2022

Supervised by:
Prof. Dr. Axel Küpper

Assistant supervisor:
Dr. Sebastian Göndör

SNET
SERVICE-CENTRIC NETWORKING

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 20, 2022 Chuck Norris' son

Abstract

In this thesis, we show that lorem ipsum dolor sit amet.

Zusammenfassung

Hier kommt das deutsche Abstract hin. Wie das geht, kann man wie immer auf Wikipedia nachlesen `http://de.wikipedia.org/wiki/Abstract`...

Contents

1 Introduction

This section gives an introduction into the general field in which you are writing your thesis. It further describes the situation today, and the problems that are solved and not.

One of the most accepted and used definitions of OSN services was given by Boyd and Ellison in, who defined a Social Network Site as a web based service that allows individuals to

1.1 Motivation

Why choose this topic? Because identities are important. Web3 brings decentralization, and identities should stay behind. W3C provides us with new standards for interoperability, and we should take this goals in mind. [1]

1.2 Digital Identities

1.3 Identifiers

Globally unique identifiers are used by individuals, organizations, abstract entities, and even internet of things devices for all kinds of different contexts. Nonetheless, the large majority of these globally unique identifiers are not under the entity's control. We rely on external authorities to issue them, allowing them to decide who or what they refer to and when they can be revoked. Their existence, validity scope and even the security mechanisms that protect them are all dependent on these external authorities. Leaving their actual owners helpless against any kind of threat or misuse[2]. In order to address this lack of control, the W3C DID Working Group conceptualized the Decentralized Identifiers or *DIDs*.

Identity management, sometimes called identity and access management, is composed of all the different ways to identify, authenticate and authorize someone to access systems or services within an organization or associated organizations.

There are several problems with our current identity management systems:

A paper-based identification such as a passport, birth certificate or driver's license is easy to lose, copy or be lost to theft. The bureaucracy behind this type of identity management is typically slow and hard to organize. The current identity and access management systems are storing your data on a centralized server along with everybody else's. This puts your digital property in danger as centralized systems are huge targets for hackers. Since 2019 alone, over 16 billion records have been leaked due to hacks and data breaches. This includes credit card numbers, addresses, phone numbers and other highly sensitive personal data. Current identities are not easily portable or verifiable. Blockchain identity management solves these

issues.

1.4 Missing factors

It was this gap in technology that prevented digital currencies from being created or adopted in the past. Blockchain fills the gap for currencies, but it also fills the gap for decentralized identity systems, opening up a whole new world of possibilities for data ownership.

Your Blockchain-Based Digital Identity At least 1 billion people worldwide are unable to claim ownership over their identities. This is one of the huge drawbacks of physical identity documentation. It's not widely available in every country. This leaves 1/7th of the entire planet unable to vote, open a bank account, or, in some cases, find a job.

Our current identity management systems are unfair and outdated, but there is a blockchain ID solution: a decentralized identity system that will revolutionize digital freedom. This is more critical now than ever before, with centralized companies left, right and center hoping to create the metaverse.

Your digital identity will be portable and verifiable all over the world, at any time of day. In addition, a blockchain-based decentralized identity is both private and secure. With verifiable credentials, your decentralized identity will empower you to interact with the SmartWeb without restrictions.

Your unique digital identity.

Imagine being able to verify your education qualifications or your date of birth without needing to actually show them. For example, a university degree could theoretically be on the blockchain and you could certify the credentials by checking the university or other issuing authority.

Similarly, you wouldn't have to show your physical ID to verify your date of birth. The authority that wants to know your age could instead use decentralized identifiers and verifiable credentials to find out if you're of the required age or not.

Authorization can be conducted in a trustless manner in which the digital identity in question is verified by an external source, and the person or organization checking can in turn verify the integrity of said source.

The verification of a proof is established by the verifier's judgment of the trustworthiness of the testimony.

This is known as a zero-knowledge proof.

Everyone in a distributed network has the same source of truth. This guarantees the authenticity of data without having to store it on the blockchain.

Blockchain technology has made it possible, for the first time ever, to have a trustlessly verifiable self-sovereign identity.

Blockchain Identity Management Blockchain identity solutions, such as Elastos DIDs, integrate state-of-the-art cryptography technology to ensure that your data is protected and private. By using decentralized identifiers, we can rebuild the structure of several flailing industries.

All of the following have poor identity management and could do with being brought up to date.

1.5 Problem statement

This section explains why this is important, why it is a problem, and why this hasn't been solved already yet. Centralized Identities, secure and open communication protocols. ActivityPub implementations at the present moment rely on HTTPS as their transport, which in turn relies on two centralized systems: DNS and SSL certificate authorities. Is there any way to bring self-sovereignty to the federated social web? [3]

ActivityPub security concerns. Encryption, non-repudiation, confidentiality..... ? → No agreed-upon security mechanisms for ActivityPub. → No encryption in scope of ActivityPub. Research Questions What are the implications of introducing DIDs to Mastodon and ActivityPub in terms of usability, and human readability? Can a DID-compliant ActivityPub protocol use DIDComm for its standard communication? Can DIDs allow ActivityPub to stop relying on the DNS for its server-to-server discovery? Can DIDComm allow ActivityPub to stop relying on transport-level security for its communication?

1.6 Expected Outcome

The goal of this thesis is to have a fully functioning Mastodon instance that is DID-compliant and that implements ActivityPub using DIDComm as its communication protocol. Replacing the existing centralized identity management with a Self-Sovereign Identity approach through DIDs, and enabling a communication protocol that allows confidentiality, non-repudiation, authenticity, and integrity without being bound to a platform-specific security mechanism.

1.7 Outline

Overview of your thesis structure in this chapter.

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. In addition, 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 interaction on the Web called *Social Web Protocols*.[4]. 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

First, it is essential to be able to describe an Activity. 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 [5]. This standard describes an activity following a story of *an actor performing an action on an object*. For this, 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
etc...		etc...

Table 2.1: ActivityStreams 2.0 vocabulary examples

ActivityStreams 2.0 has improved its 1.0 version in more than one perspective. 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 [6].Ẇith JSON-LD, ActivityStreams 2.0 can define its own context and the terms that will be used inside this context. Figure 2.1 shows an example of a JSON-LD serialized ActivityStreams 2.0 activity.

```
1 {
2   "@context": "https://www.w3.org/ns/activitystreams",
3   "summary": "Alice created an image",
4   "type": "Create",
5   "actor": "http://www.test.example/Alice",
6   "object": "http://example.org/foo.jpg"
7 }
```

Listing 2.1: Example of activity [5]

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[7], 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 this actor object with several properties, as shown in 2.2. Optional properties include collections such as *following*, *followers*, *liked* and compulsory properties include an *inbox* and an *outbox*. These last two are URLs that represent how the actor gets and sends messages from other users.

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

```
{
  "@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 [8]

There are two workflows of communication for a user in ActivityPub:

- **Client-to-server communication:** A user wants to share a post so it makes an HTTP POST request to its outbox with the respective activity object. After this, other users interested in seeing this user's posts can make an HTTP GET request to the user's outbox and retrieve all his public posts.
- **Server-to-server communication (Federation)**: User *A* wants to send a post to user *B*, whose account is on a different server. 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 have to figure out how to do it independently.

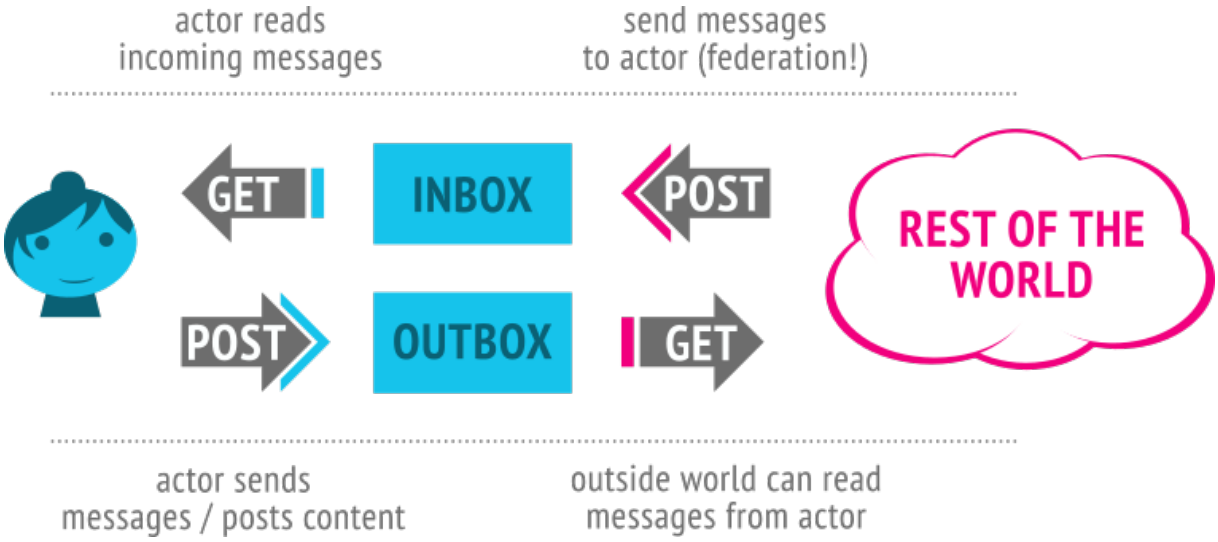

Figure 2.1: ActivityPub overview [8]

Regarding security, ActivityPub's specification does not define any official security mechanisms to ensure confidentiality, non-repudiation, message integrity, authentication, or authorization [8]. However, it mentions a list published by the SocialWG of best security practices[8] that may be used in ActivityPub. 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 at least transport-layer encryption.

2.2 ActivityPub-based Social Networks

2.2.1 The Fediverse

It is impossible not to refer to the *Fediverse* when we talk about ActivityPub. 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. In today's most popular social networks like Facebook, Twitter, or Youtube, a centralized architecture keeps millions of users on one platform. Control, decision-making, user data, and censorship depend on a single profit-driven organization. On the contrary, the *Fediverse* is developed by a not-profit-driven community of people around the globe independent of any corporation or official institution [9] [10]. 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 with each other. 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 in January 2018, a big number of these federated social networks upgraded to ActivityPub. Becoming rapidly the predominant protocol. 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. For example:

- **PeerTube[15]:** A decentralized alternative to video platforms, similar to Youtube.
- **Mastodon[16]:** A microblogging social network, similar to Twitter.
- **Pixelfed[17]:** An image-sharing platform, similar to Instagram.

[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://www.w3.org/community/ostatus/wiki/images/9/93/OStatus_1.0_Draft_2.pdf
[13] https://matrix.org
[14] https://diaspora.github.io/diaspora_federation
[15] https://joinpeertube.org
[16] https://joinmastodon.org
[17] https://pixelfed.org

Although it was not the first social network to implement ActivityPub, Mastodon is the one that pioneered its use on a large scale [11]. Moreover, it is still the social network in the *Fediverse* with the biggest user base and popularity. For this reason, Mastodon will be used in this thesis as the representative of 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 to bring 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 [12]: *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* [13]. Similar to 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*[18] 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 feeds of the users in other instances with similar interests [14]. 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.

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

2.2.2.1 Security

Mastodon has implemented the authentication and authorization mechanisms from the best practices list of the SocialWG, to address the security concerns in ActiviyPub. However, relevant for this thesis are the ones used in federated communication, namely HTTP and JSON-LD Signatures. HTTP signatures extend the HTTP protocol by adding the possibility to cryptographically sign the HTTP requests. 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 [15]. 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

[18] A server running Mastodon

the sender's public key. Finally, the *Signature* header indicates the key that was used to sign the document, the parameters that are inside the signature, and the algorithm used to hash it. An example in Mastodon is shown by figure 2.3. Following the same idea, Linked Data Signatures offer a way to create and attach signatures to JSON-LD documents, thus providing non-repudiation to e.g. an ActivityStreams Activity object even if the object has been shared, forwarded, or referenced at a future time [4]. However, this feature, although implemented, is not actively used in Mastodon.

```
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

For Mastodon to be able to implement these signatures, it was necessary to generate keypairs for the users. For this, Mastodon added a new property *publicKey* to the actor object, which includes the pem-formatted public key of an RSA-2018 keypair, as shown in figure**??**.

2.2.2.2 Resolving accounts

As explained in 2.1.2, ActivityPub requires a resolving process when wanting to send a message to a user whose account resides on a different server. For this, Mastodon implemented *Well-Known URIs*[19], which enable the discovery of information about an origin in well-known endpoints[16]. The two most relevant endpoints are the following:

- **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 [17]. It employs the XRD 1.0[20] 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 [17]. 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.
- **Webfinger:** Mastodon relies on the Webfinger protocol for the resolving process and its federated functioning [18]. Webfinger allows for discovering information about persons or other entities on the Internet using HTTP such as a personal profile address, identity

[19] https://www.rfc-editor.org/rfc/rfc8615.html
[20] https://docs.oasis-open.org/xri/xrd/v1.0/os/xrd-1.0-os.html

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

Listing 2.4: Example Host Metadata request to mastodon.social

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

Listing 2.5: Example Host metadata response from mastodon.social

service, telephone number or email. 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*[21] 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 [19]. Fig. shows an example of the returned JRD document provided by the WebFinger endpoint of the *mastodon.social*[22] instance when querying the account *acct:bob@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

2.2.3 Extending ActivityPub

2.3 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 both individuals and organizations to take greater control of their online information and relationships while also providing greater security and privacy [20]. 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[21][2].

[21] https://datatracker.ietf.org/doc/html/rfc7565
[22] https://mastodon.social

```
{
    "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"
        },
        {
            "rel": "http://ostatus.org/schema/1.0/subscribe",
            "template": "https://mastodon.social/authorize_interaction?uri={
    uri}"
        }
    ]
}
```

Listing 2.7: Webfinger response

2.3.1 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 [22][2].

Figure 2.2: DID composition [2]

The DID Subject that can modify the DID Document is called the *DID controller*. Usually, the DID subject is the DID controller, however, 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, as well as services that are relevant to be able to interact with the DID Subject. An example of a DID Document can be seen in 2.8. 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

[2].

```
{
  "@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 2.8: Example DID Document

DID documents are stored in a *Verifiable Data Registry* (VDR). This is essentially any system that enables capturing DIDs and returning required data to generate DID documents, such as distributed ledgers, decentralized file systems, any type of database, peer-to-peer networks, or other types of trustworthy data storage. Our 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[23] of the W3C, there are around 103 registered DID method specifications. More information about the different existing DID methods can be found in the next subsection 2.3.2.
The last major component in this architecture overview is the one in charge of resolving DIDs is called *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 [2]. The Identifiers & Discovery Working Group (ID WG) has implemented a prototype Universal Resolver[24], which allows the resolution of DIDs for numerous DID methods. In addition, this working group has also developed a Universal Registrar[25], which allows the creation, edition, and deactivation of the DIDs across different DID methods.

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

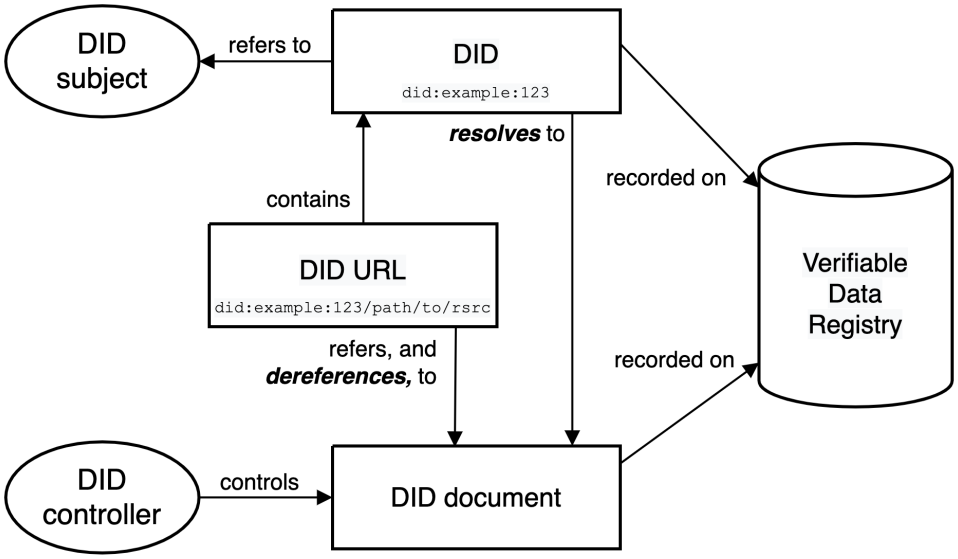

Figure 2.3: DID architecture overview [2]

2.3.2 DID methods

Based on their characteristics and patterns, most DIDs can be sorted into different categories [23], 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 correspondingly.
- **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 [24]. An example DID method in this category is the *did:ion*[26], 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. The DIDs in this category 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. [23].
- **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.[25].

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 make them, as

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

well as 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*.

2.4 DIDComm Messaging

The Hyperledger Foundation is an open-source collaborative effort intended to further develop blockchain technologies across industries [26]. 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 [27]. 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, HU is a shared cryptography library that helps to avoid duplicating cryptographic work across projects while also 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 [26]. Based on DIDComm v1, the Communication Working Group (CWG) of the DIF 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 would allow higher-order protocols to build upon its security, privacy, decentralization, and transport independence in the same way web services build upon HTTP. [28] [29]

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, routable, interoperability, and the ability to be transport-agnostic [29].
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 was sent. DIDComm's paradigm is asynchronous and one-directional, thus showing a considerable resemblance with 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 [29].
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 [30]

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 strived to define JSON-based object formats to represent integrity and confidentiality. As well as a JSON way to express keys. Relevant resulting standards from this WG, as shown in figure 2.4, are the following:

- **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)[31].
- **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. [32].
- **JSON Web Key (JWK):** It provides a JSON format to represent a cryptographic key [33].
- **JSON Web Algorithms (JWA):** It provides a collection of algorithms to be used by the previously mentioned standards [34].

DIDComm specifies using other proposed standards that were not part of the JOSE WG. For example, the **JSON Web Message (JWM)**[27], which is a basic, flexible JSON format to encode messages for a transport agnostic delivery; and the **ECDH-1PU**[28], a public key authenticated encryption algorithm designed for the JWE.

[27] https://datatracker.ietf.org/doc/pdf/draft-looker-jwm-01
[28] https://datatracker.ietf.org/doc/html/draft-madden-jose-ecdh-1pu-04

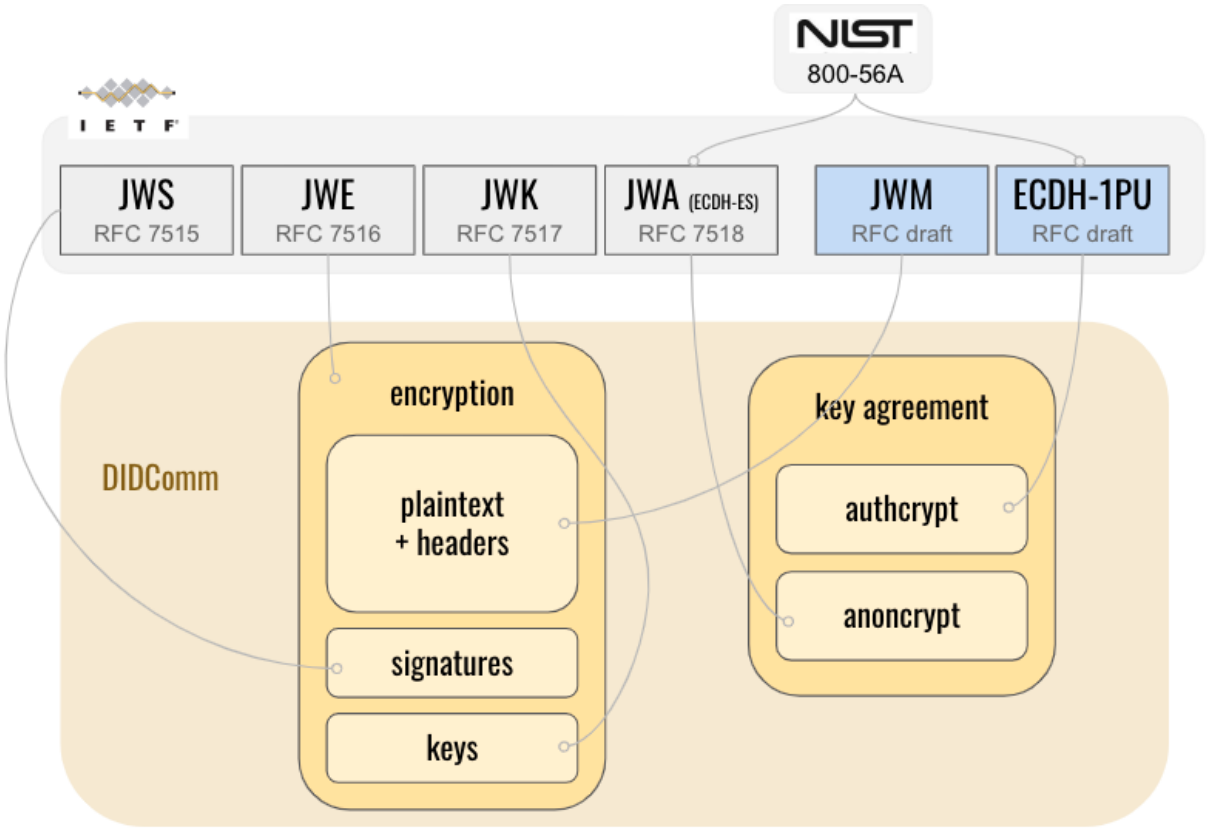

Figure 2.4: Standards used in DIDComm

As shown in figure 2.4, DIDComm offers different approaches for key agreement encryption. Authenticated Sender Encryption (*authcrypt*) and Anonymous Sender Encryption (*anoncrypt*) are both encrypted and delivered to the recipient's DID. Still, they differ because only *authcrypt* gives direct guarantees about the sender's identity. Sending anonymous messages in social networks is not usually the case, and removing the attribution of a post can lead to other problems [35]. However, social networks like Ask.fm[29] or NGL[30] 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 then the resulting JWS is used as the payload of a JWE. The algorithm 2 illustrates better the workings of this.

[29] https//ask.fm
[30] https://ask.fun

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

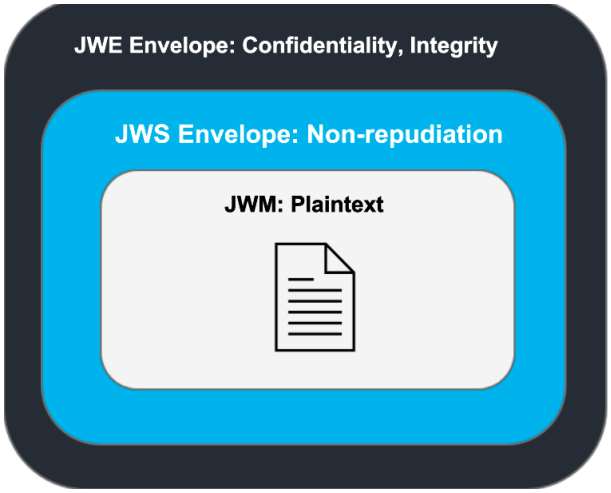

Figure 2.5: Nested JWT example

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 section, we are going to go through the different steps that are required firstly to integrate DIDs into Mastodon, and finally to enable DIDComm for its communication. As explained in chapter 2, Mastodon is the social network that pioneered the use of ActivityPub on a large scale, and it is also the social network with the most active users and presence inside the Fediverse. For this reason, the proposal presented in this section and the modus operandi of the ActivityPub server are 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, an example of a simple use case in MAstodon's implementation is going to be illustrated and analyzed in order to be able to compare it with section 3.4, which finally presents the same use case but 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. In order 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.
- **Toot**: In the user-facing part of Mastodon, a Toot is the äquivalent of a Tweet on Twitter. This is a small status update with a 500-character limit.
- **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

In order to explain the current ActivityPub flow in Mastodon, let's describe what happens in the simple use case:

Alice has an account in the Mastodon instance alice_server.com and 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

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"[8], 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, the Alices' server may take one of two options. Nonetheless, the use case explicitly dictates that Bob's account resides in a different Mastodon instance, namely *bob_server.com*.

```
{
  "@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

If the recipient's account is on the same server, there is then no explicit resolving process. A simple query in the ActivityPub's server would find the right account and save the status within the account's statuses. On the contrary, when the recipient's account is not on the same server, then a resolving process must be started. 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 the resolving in this specific use case is to find the inbox URL of Bob, 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 Webquery request to bob's server Webquery endpoint. The request shown in 3.3 will return a JRD Document, as shown in figure 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. If the Webfinger request 3.5 returns a 404 code, it will then try, as a fallback, using the Host-Meta endpoint. The response constains a link template3.6, that Alice's server can use to try the Webquery 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

```
{
  "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"
    },
    {
      "rel": "http://ostatus.org/schema/1.0/subscribe",
      "template": "https://bob_server.com/authorize_interaction?uri={uri}
  "
    }
  ]
}
```

Listing 3.4: Webfinger response

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

Listing 3.5: Hostmeta request

```
<?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: Hostmeta response

3.3.3 Delivery

Succeeding the retrieval of Bob's actor object and therefore the needed 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 feature are within the Mastodon implementation of ActivityPub.

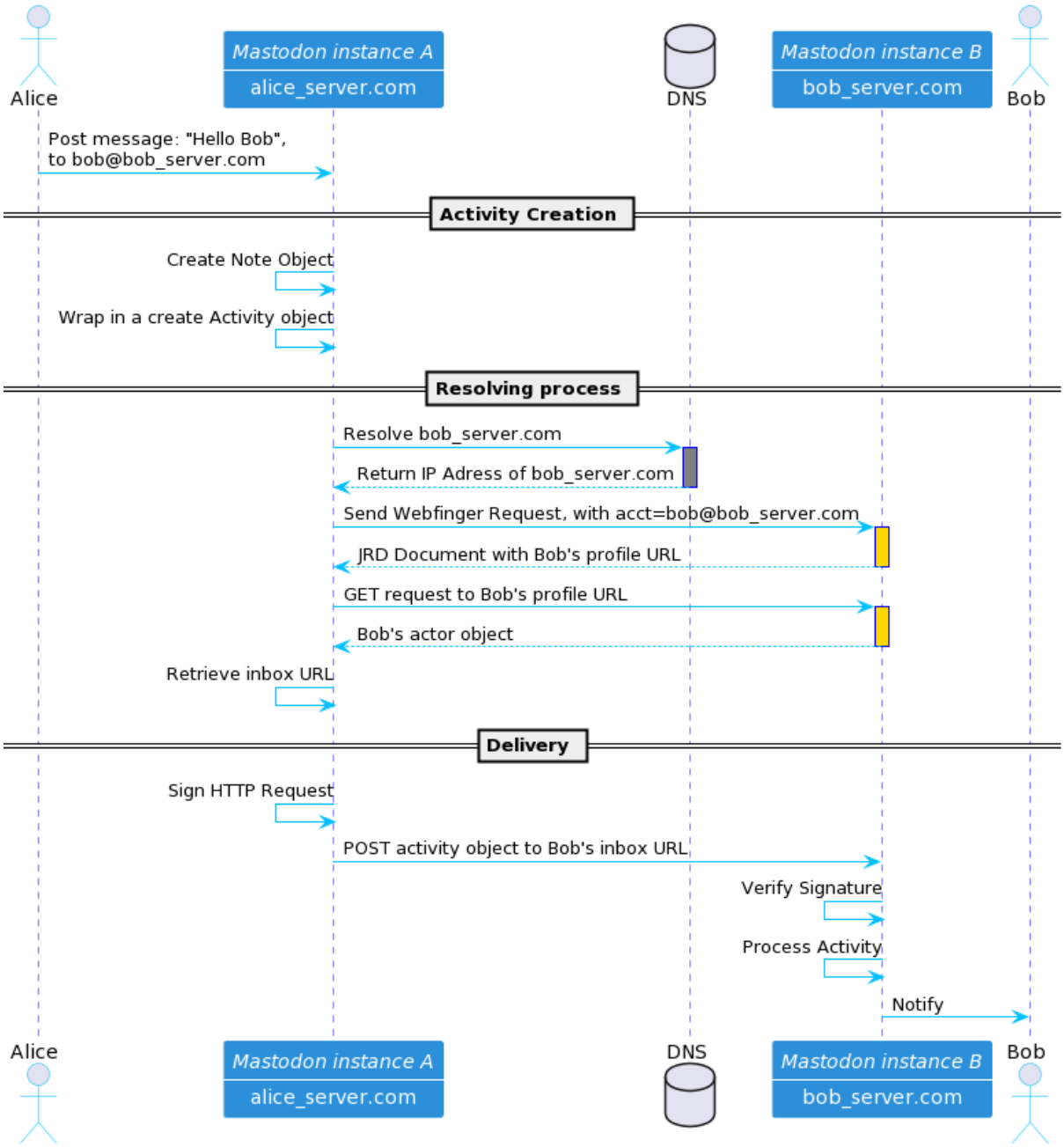

Figure 3.1: Current flow for sending message

3.4 Proposed implementation

Having seen the state-of-the-art flow of our use case mentioned above, it is now imperative to address the necessary steps to make ActivityPub DID-compliant, as well as enabling DIDComm as its communication protocol.

3.4.1 Implications of integrating DIDs

The first question that needs to be addressed when approaching the integration of DIDs is, what are 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 somehow. Currently, most of the interactions of Mastodon via ActivityPub require the ID property to resolve to the user's profile to his actor's object. Following a simple strategy, we could simply replace 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 [8]. As explained in section 2.3, 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. 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. Moreover, 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 [8]. Although plausible, for this thesis 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 idea was briefly suggested by [3] in a paper prepared for the 2017 Rebooting Web of Trust summit. Such an idea would look like fig. (DID Document with all ActivityPub endpoints). Furthermore, the authors went a step further and proposed cutting all dependency from the DNS by using onion websites. 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"* [2]. DID Documents are stored in a publicly available verifiable data registry, therefore any per-

sonal 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. Looking at figure **??**, the amount of personal information displayed in the DID Document, which would not be otherwise inferable, already poses a privacy issue for the DID Subject. For this reason, in this thesis we differ from removing the actor's object from the ActivityPub protocol itself. This would also allow us to use freely 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. Fig(final Actor'S object) illustrates the final design for the DID-compliant actor object.

Regarding Mastodon, replacing the standard username with a DID does not imply huge complications. The way Mastodon validates the username format is through the following regular expression:

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

Additionally, it has a length constraint of 30 characters. The following DID-syntax-compliant regular expression can be used instead, as well as an extended maximum length of 85 characters.

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

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 come into play. The proposed flow takes the following steps. Firstly, 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 [19]. 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 ActivityPub service section in the DID Document.

The easiest way to set a DID resolver, is to use the one from the DIF. They provide Docker images for the service running the query endpoint, and for the drivers needed for each DID method. As Mastodon can be run using Docker Compose, the resolver can be added to the same docker network as the main backend application. This would allow accessing the resolver through the alias name *did-resolver*. The resolver validates the DID document by comparing the

searched DID against the ID attribute of the DID document and only returns the DID document when both match. To make the requests to the resolver the service class *DidResolverService* will be added to the backend. The only parameter it needs is the DID it needs to resolve, and it returns the DID Document as a JSON object. 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.

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 resolver, 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 also takes the username parameter in the form of *did:method:example* and then uses the resolver service to make a query to the universal resolver. A *DidDocument* class gets created with the JSON response and the actor URL its obtained 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.

3.4.3 Enabling DIDComm Messaging

Having introduced DIDs to Mastodon and ActivityPub, it is now possible to enable DIDComm. Taking into account the algorithm 1 shown in section 2.4, it is possible to derive some requirements that DIDComm imposes:

1. Access to private key: The ActivityPub server requires access to the private key of the selected verification method. This of course imposes risks, as the private key is no longer in the user's control. The administrator of the Mastodon instance would have access to the plaintext private key, and some security countermeasures like key rotation would not be able to counter this. Furthermore, the ActivityPub server must be able to support the keys and the cryptographic algorithms, which the JWA includes. This means having any kind of library that can parse them and perform encryption as well as decryption with them.

2. 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. The DID specification luckily provides us with 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 [2]. 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, we will stick to the recommendation using the *keyAgreement* key for this proposal.

3. Access to DID-Resolver: This is for cases where a signature needs to be verified using an external public key. However, this requirement has already been fulfilled.

Creating a DID is rather a simple task. However, finding a DID method that would allow CRUD operations to add the service endpoint and a *keyAgreement* key to its DID Document without needing to pay any GAS or any other kind of fees is not. The following options were tested. MATTR[1]offers creating DIDs using *did:key*, *did:web* and *did:ion* methods. However, they do not allow creating own keys or accessing private keys, which is necessary according to the first requirement. *did:ion* offers a set of tools[2] 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[3], but so far, it has not been addressed by the developers. An alternative to ledger-based DIDs developed for this thesis was using the *did:web* method. This method allows hosting the DID Document on any server, 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[4] and Veramo[5]. 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[6], where the only thing needed externally is an Infura[7] account to use as a Web3 Provider. Two DIDs were created in the Ropsten network for Alice and Bob, respectively.

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

[1] https://mattr.global
[2] https://github.com/decentralized-identity/ion-tools
[3] https://github.com/decentralized-identity/ion-tools/discussions/25
[4] https://serto.id
[5] https://veramo.io
[6] https://veramo.io/docs/node_tutorials/node_setup_identifiers
[7] https://infura.io

Having DIDs created, the next task is to add the ActivityPub service endpoint to the DID documents of Bob and Alice. The parameters required for Veramo's API to process the information correctly, as shown in figure 3.7, are the DID, the service, and options for the Web3 provider. The service includes 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://alice_server.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 3.7: Parameters to add a service in Veramo

The biggest challenge in implementing DIDComm is the compatibility between the algorithms specified by the JWA spec, the key types, and the libraries used to generate keys, sign, and encrypt. In Mastodon, OpenSSL[8] is the library used for generating the RSA keys used for the HTTP and JSON-LD signatures. This library wraps the OpenSSL project toolkit[9] and provides a wide range of key management, encryption, decryption, and certificates management. To create signed and encrypted messages the JWT library[10] for ruby on rails was selected as it allows the spec-compliant creation of JWS and JWE tokens.
For this proposal, the objective is to add confidentiality, integrity, and non-repudiation to every ActivityPub object sent in Mastodon. As explained in section 2.4, the recommended way to achieve this is by using *authcrypt*. However, during the development of this thesis, there were no libraries for Ruby and Rails that supported the *ECDH-1PU* algorithm. Furthermore, the JWT library had a limited number of available algorithms. For this reason, it was decided to implement a nested JWT, following the same procedure as algorithm 2. To prevent any further compatibility problems on the key generation side, it was decided to use the common widely-supported RSA key generation algorithm. This means, that both Alice and Bob will have an extra RSA keypair, and the private keys will be stored in their respective Mastodon instance. The JWS tokens were set to use an RSA 2048 key and the *RS256* hash algorithm, and the JWE tokens to use the *RSA-OAEP*[11] algorithm for key management with the *A128CBC-HS256*[12] algorithm for encryption. With existing RSA keys for both our actors, it is now necessary to add the respective public keys to their DID documents. Veramo's API offers a way to add different kinds of cryptographic keys to a DID document, however, the only available key types

[8] https://github.com/ruby/openssl
[9] https://www.openssl.org/
[10] https://github.com/jwt
[11] https://www.rfc-editor.org/rfc/rfc7518#page-14
[12] https://www.rfc-editor.org/rfc/rfc7518#page-22

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, as the already mentioned 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* adding an RSA key to the DID document. The final DID document for Alice's DID can be seen in figure **??**.

At this point, all the elements needed to enable DIDComm are present, i.e. two Mastodon instances for federated communication with DID integration, a DID resolver and a service class that can resolve DIDs, as well as DID documents for both actors with the ActivityPub service endpoint and the public key for the verification method. The next step is to put them together. As part of this proposal, further changes to the ActivityPub protocol itself are not in the 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 simplest way to keep a modular approach is by using the ActivityStream object as a payload for our JWM, which would look like figure 3.8. The final payload that will be sent in server-to-server communication will consist of a JWE token, whose content is a signed JWM that includes the Activity in its body, following the structure of figure 3.2.

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

Listing 3.8: JWM example

The final flow for our use case mention in3.2 is illustrated in figure 3.3

Figure 3.2: Proposed Payload structure using DIDComm and ActivityPub

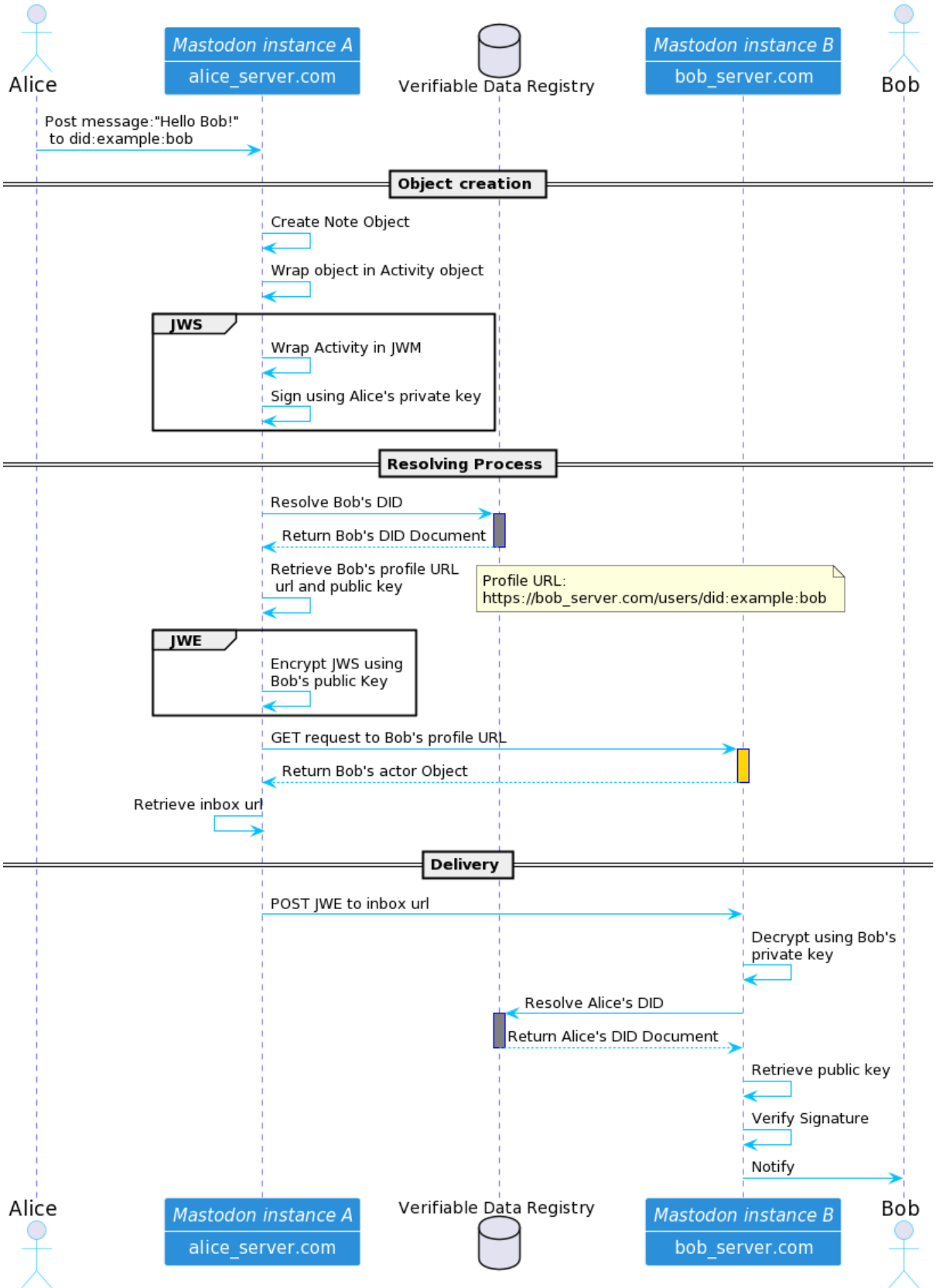

Figure 3.3: DID and DIDComm flow for use case

4 Implementation

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[1]. 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.

4.1 Mastodon

The source code[2] 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[3], based on the Ruby programming language. The backend manages a Postgres[4] 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[5] 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.

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

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

4.1.1 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

4.1.2 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[6] 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 1

4.1.3 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 2, 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.

[6] https://nginx.com

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

```

Listing 4.1: Building the development Dockerfile

4.1.4 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 4.2.

```
// 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 4.2: 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.

4.2 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[7], and the methods for the CRUD operations were written based on the API reference [8].

4.2.1 Requirements

Software requirements to build the Veramo agent include:

- Node 10.8.2+
- Yarn

[7] https://veramo.io/docs/node_tutorials/node_setup_identifiers/
[8] https://veramo.io/docs/api/core

4.2.2 Configuration

First, an *.env* file that contains the variable: *INFURA_PROJECT_ID="example-infura-project-id"* from an Infura[9] account is required. This project ID has to match the Ethereum network desired. The default selected in the setup is *Ropsten*.

4.2.3 Build

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

4.2.4 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 4.3. 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[10]. 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 4.4	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

[9] https://infura.io/
[10] 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 4.3: 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 4.4: Parameters to add a service in Veramo

5 Evaluation

As mentioned in the definitions section, Mastodon's full username includes the domain of the server and a locally-unique username. This type of username accomplishes the goals of human readability and uniqueness. In addition, they are resolvable using DNS and the well-known URIs.

6 Conclusion

Describe what you did here

List of Tables

List of Figures

Bibliography

[1] Guttenberg, "Double Brackets WiLL preserve cApItAlIzAtIoN," *Guttenberg Press*, 2011. [Online]. Available: http://www.examplelink.de

[2] 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/

[3] 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

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

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

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

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

[8] 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

[9] 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/

[10] 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

[11] 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

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

[13] 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

[14] 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.

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

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

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

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

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

[20] "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

[21] 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.

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

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

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

[25] 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/

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

[27] 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

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

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

[30] 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

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

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

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

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

[35] 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

Appendices

Appendix 1

Appendix: Installation Guide

1 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.

2 Environment 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=
```

Listing 2: Environment file template