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 16, 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 16, 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. [**?**]

1.2 Digital Identities

1.3 Identity Management

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? [**?**]

ActivityPub security concerns. Encryption, non-repudiation, confidentiality….. ? $\rightarrow$ No agreed-upon security mechanisms for ActivityPub. $\rightarrow$ No encryption in scope of ActivityPub. Research Questions What are the implications of introducing DIDs to Mastodon and ActivityPub in terms of usability, discovery 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 ActivityStreams 2.0

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[**?**].

To be able to describe what a social activity is, it is essential to be able to describe an Activity. One of the standards of the SocialWG is ActivityStreams 2.0[1]. This standard provides a model for representing *Activities* using a JSON-based syntax and a vocabulary that includes all the standard terms needed to represent social activities [**?**]. ActivityStreams describes an activity following a story of *an actor performing an action on an object* and specifies different types of actors, activities, and objects, as shown in **??**. Each of these objects can be represented as a JSON object, creating a solid foundation on which other protocols can build.

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

Table 2.1: ActivityStreams 2.0 vocabulary examples

ActivityStreams 2.0 builds on top of its predecessor, the 1.0 version, and it has improved it

[1] https://www.w3.org/TR/activitystreams-core/

in more than one perspective. One of these is the compatibility with JSON-LD[2], which is a JSON serialization for *Linked Data*[3]. The concept of Linked Data is based on interlinking data in such a way that it becomes more usable through associative and contextual queries [**?**].With JSON-LD, ActivityStreams 2.0 can define its own context and the terms that will be used inside this context. Figure **??** 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 activity [**?**]

2.2 ActivityPub

ActivityPub is another W3C Recommendation that originated from the SocialWG. It is a decentralized social networking protocol that uses the syntax and vocabulary of ActivityStreams 2.0. It provides a client-to-server API, which covers the requirements of a Social API[**?**], i.e., publishing, subscribing, reading content, and notifying when content gets created. In addition, it provides a server-to-server API that enables decentralized, specifically, federated communication. The way ActivityPub works is the following. A user of a social network that implements this standard has its JSON-LD-based profile **??**, which is the actor object specified before in **??**. ActivityPub added to this actor object two endpoints, namely, the inbox and the outbox, which represent how the actor gets and sends messages from other users. For example, when a user wants to share a post, 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. In the case of a private message, such as user A wanting to send a message to user B, the same workflow would occur. User A posts his message to his outbox, and then the ActivityPub server of A will go ahead and look for the inbox of B to make an HTTP POST request. Following this, B makes an HTTP GET request to his inbox to retrieve all the posts addressed to him [**?**]. Figure **??** illustrates the functioning of ActivityPub.

[2] https://www.w3.org/TR/json-ld/
[3] 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 [?]

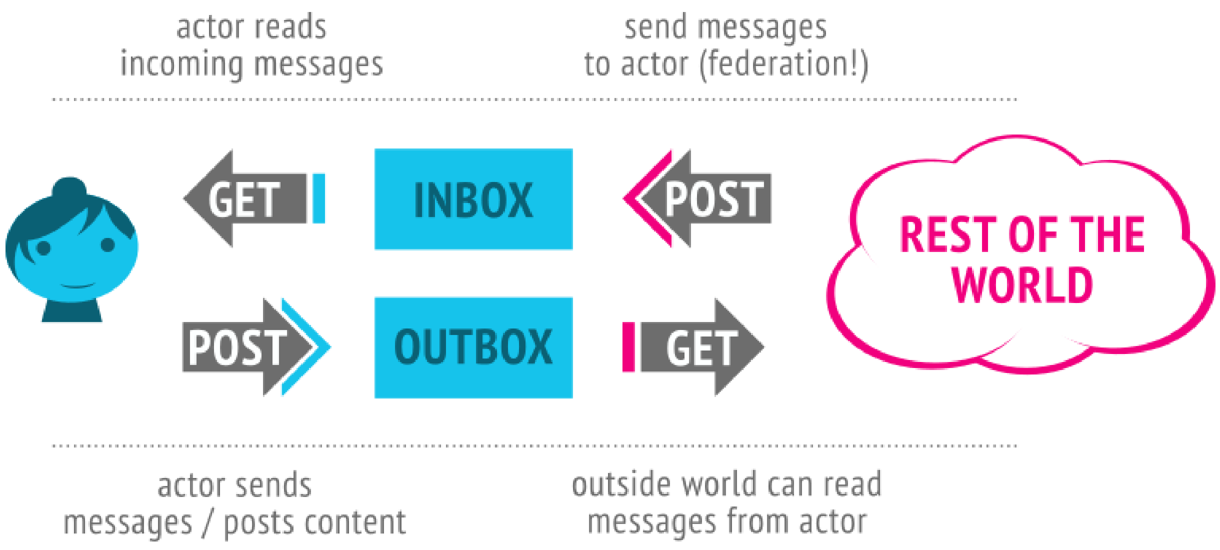
actor reads incoming messages
send messages to actor (federation!)
GET
INBOX
POST
REST OF THE WORLD
POST
OUTBOX
GET
actor sends messages / posts content
outside world can read messages from actor

Figure 2.1: ActivityPub overview [?]

Regarding security, ActivityPub's specification does not define any official security mechanisms to ensure confidentiality, non-repudiation, message integrity, authentication, or authorization [?]. It recommends using HTTPS for its HTTP-based communication, relying on third-party certificate authorities to provide transport-layer security. Nonetheless, other security aspects remain unaddressed. Although not included in ActivityPub, the SocialWG published a list of best security practices[4] which suggests using standards like OAuth 2.0[5] for client-to-server authentication, as well as HTTP Signatures[6] and Linked Data Signatures[7] for server-to-server authentication.

HTTP signatures extend the HTTP protocol by adding the possibility to cryptographically

[4] https://www.w3.org/wiki/SocialCG/ActivityPub/Authentication_Authorization
[5] https://oauth.net/2/
[6] https://tools.ietf.org/html/draft-cavage-http-signatures-08
[7] https://w3c-dvcg.github.io/ld-signatures/

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 [?]. 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 [?].

The reason why these security specifications were not included in ActivityPub is unclear. Nonetheless, some ActivityPub-based social networks have added HTTP and JSON-LD signatures to their implementation. Providing a new layer of security for its userbase. One of these social networks is Mastodon, which will be explained in the next section.

2.3 Mastodon

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, Mastodon[8] is a decentralized microblogging social network created 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[?]. *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* [?].

Mastodon strives to give people back control of content distribution channels by eliminating the presence of sponsored users or posts in feeds [?].

Mastodon differs from other commercial social networks in two aspects. First, it is oriented towards small communities and community-based services. In fact, each instance may support and favor specific topics. So prior to registration, a user is encouraged to choose the instance better suited to her own tastes. Second, the Mastodon platform does not provide any algorithm for recommending new friends or promoted contents. So, the only way to establish a connection or consume a content is by searching an already known account through the search functions or by exploring the feeds of the instances in search of users with similar interests or interesting posts.

Today, Mastodon is a network of thousands of communities. Each is operated by different individuals and organizations that implement their own policies, codes of conduct, and discussion topics. By means of this, the user has the opportunity to choose whichever instance he finds the most fitting. Mastodon takes a significant share in the Fediverse. A single interoperable ecosystem of different social networks that can communicate with each other. In other words, it is a collection of federated social networks running on free open software on thousands of servers across the world that implement the same protocol in order to be able to interact with each other. The Fediverse is developed by a community of people around the globe independent of any corporation or official institution. It is not profit-driven, and it gives users the freedom to choose which service and instance of the Fediverse to register. On top of that, you are free to run your own instance with your own policies and allow other users to join [?]. The range of services that can be found inside the Fediverse includes blogging, microblog-

[8] https://joinmastodon.org/

ging, video streaming, photo, music sharing as well as file hosting. Since its standardization, ActivityPub has been widely adopted in the Fediverse, being today the dominant protocol.

After ActivityPub was published as a Recommendation by the Social Web Working Group of the W3C in January 2018, Mastodon decided to switch from its prior and pioneered the use of ActivityPub on a large scale. Promoting this way, its adoption.

From an architectural viewpoint, the platform follows a federated architecture organized into two layers own rules, account privileges, and whether or not to share messages coming to and from other instances. Each server hosts individual user accounts, the content they produce, and the content they subscribe to. From a user experience viewpoint, Mastodon releases the major features of a microblogging platform: • users can follow one another, whether or not they are hosted on the same instance; • users can post short messages consisting of up to 500 text characters, called 'toots', for others to read. • toots are aggregated in local and federated timelines.

Mastodon differs from other commercial microblogging platforms w.r.t. two key points. First, it is oriented towards small communities and community-based services. In fact, each instance may support and favor specific topics. So prior to registration, a user is encouraged to choose the instance better suited to her own tastes. Second, the Mastodon platform does not provide any algorithm for recommending new friends or promoted contents. So, the only way to establish a connection or consume a content is by searching an already known account through the search functions or by exploring the feeds of the instances in search of users with similar interests or interesting posts. These two characteristics are fundamental for the study of the interplay between the physical layer of a decentralized social network and its overlaid social network, since: (i) the coupling server/topic affects how people are distributed on the servers set and how they interact among themselves, this is true because it is well-known that common interests shape the structure of social networks [7], [8]; (ii) the lack of recommendation systems removes different external factors – often hidden by recommendation algorithms – from the mechanisms driving the formation of new links [9], [10].

process and the decentralized organization of the instance might be one of the few factors affecting the choice of who to "follow".

2.3.1 Federation

[?] As per documentation, Mastodon implements the following 4 different well-known endpoints:

2.3.1.1 NodeInfo

NodeInfo is an initiative to standardize the presentation of metadata about a server operating one of the distributed social networks. The two main aims are to get greater insights into the distributed social networking user base and to provide tools that allow users to select the most suited software and server for their requirements. Mastodon is one of the implementers of this protocol, along with other federated social networks such as Diaspora, Peertube and Wordpress [http://nodeinfo.diaspora.software/]. NodeInfo specifies that servers must provide the well-known path /.well-known/nodeinfo and provide a JRD document referencing the supported documents via Link elements, as shown in . [http://nodeinfo.diaspora.software/protocol.html] Accessing the hypertext reference from the JRD response will give a schematized series of metadata of the instance running the endpoint, such as NodeInfo schema version, software, protocols supported by the server, statistics and even a list of third-party services that can interact with the server via an API. Figure shows the NodeInfo 2.0 schema of the Mastodon instance mastodon.social.

```
{
    "links": [
        {
          "Rel": "http://nodeinfo.diaspora.software/ns/schema/2.1",
            "href": "https://example.org/nodeinfo/2.1"
        }
    ]
}
```

```
    GET /wel-known/nodeinfo/2.0 HTTP/1.1
    Host: mastodon.social
```

Listing 2.4: NodeInfo request

```
{
   "version": "2.0",
   "software": {
       "name": "mastodon",
       "version": "3.5.3"
   },
   "protocols": [
       "activitypub"
   ],
   "services": {
       "outbound": [],
       "inbound": []
   },
   "usage": {
       "users": {
           "total": 718555,
           "activeMonth": 61717,
           "activeHalfyear": 178356
       },
       "localPosts": 36484017
   },
   "openRegistrations": true,
   "metadata": []
}
```

2.3.1.2 Web Host Metadata

Web host metadata is a lightweight metadata document format that allows for the identification of host policy or information, where *host* refers to the entity in charge of a collection of resources defined by URIs with a common URI host. It employs the XRD 1.0[9] document format, which offers a basic and flexible XML-based schema for resource description. Moreover, it provides two mechanisms for providing resource-specific information, specifically, *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 [**?**]. 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 **??** and **??**.

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

Listing 2.6: 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 2.7: Example Host metadata response from mastodon.social

2.3.1.3 Webfinger

Finally, Webfinger is the protocol on which Mastodon heavily relies for the resolving process and its federated functioning [**?**]. Webfinger allows for discovering information about persons or other entities on the Internet using HTTP such as a personal profile address, identity 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*[10] URI format, which aims to offer a scheme that generically identifies a user's account with a service provider without requiring a specific protocol. In the same way NodeInfo works, it returns a JRD Document describing the entity [**?**]. Fig. shows an example of the returned JRD that is being provided by the WebFinger endpoint in the mastodon social 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 2.8: HTTP request to Webfinger endpoint

[9] https://docs.oasis-open.org/xri/xrd/v1.0/os/xrd-1.0-os.html
[10] https://datatracker.ietf.org/doc/html/rfc7565

```
{
    "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 2.9: Webfinger response

2.4 Decentralized 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 [**?**]. In order to address this lack of control, the W3C DID Working Group conceptualized the Decentralized Identifiers or *DIDs*.

DIDs are a new type of globally unique identifier that enables individuals and organizations to create their own identifiers using trustworthy systems. By means of this, entities are able to prove control over them by authenticating using cryptographic proofs. Furthermore, given that the generation and assertion of DIDs are entity-controlled, an entity can create any number of DIDs that can be tailored and confined to specific contexts. This would enable interaction with other systems, institutions, or entities that require authentication while also limiting the amount of personal or private data to be revealed. All of this without the need to rely on a central authority [**?**]. To better illustrate how DIDs work, let's address the components which constitute DIDs. Figure **??** provides a basic overview of the major components of the Decentralized Identifier architecture.

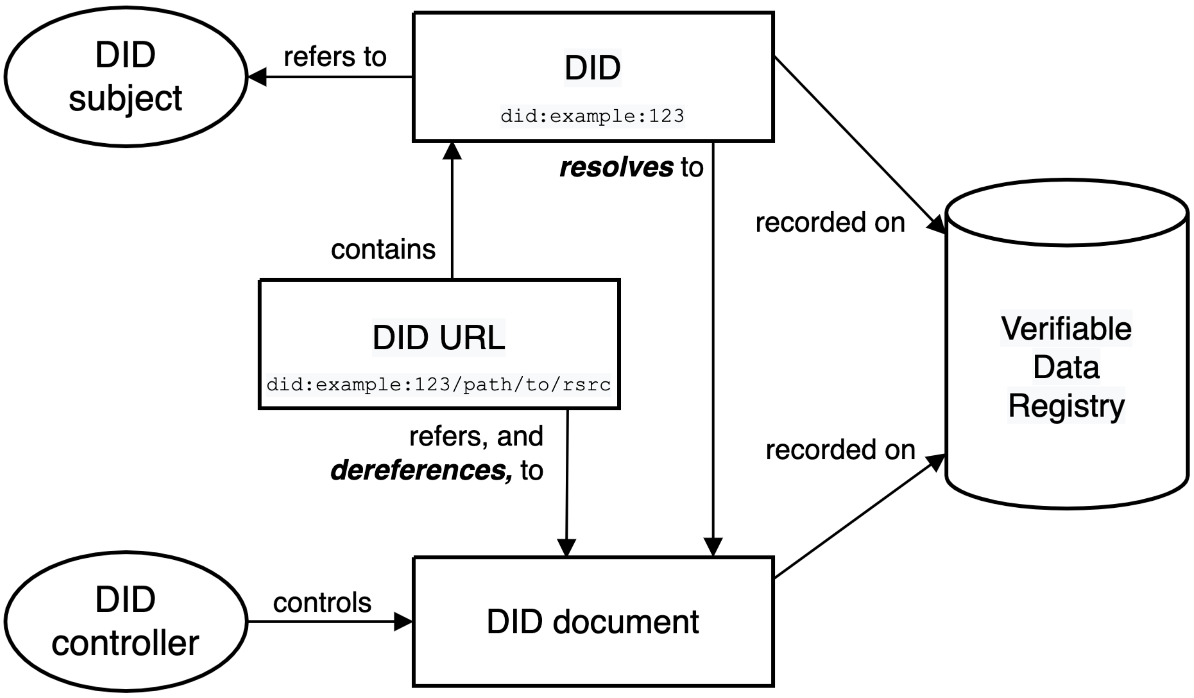

Figure 2.2: DID architecture overview [?]

2.4.1 DID

The DID itself is a URI[11] that consists of 3 different parts, namely the did URI scheme identifier, the method identifier and the DID method-specific identifier.

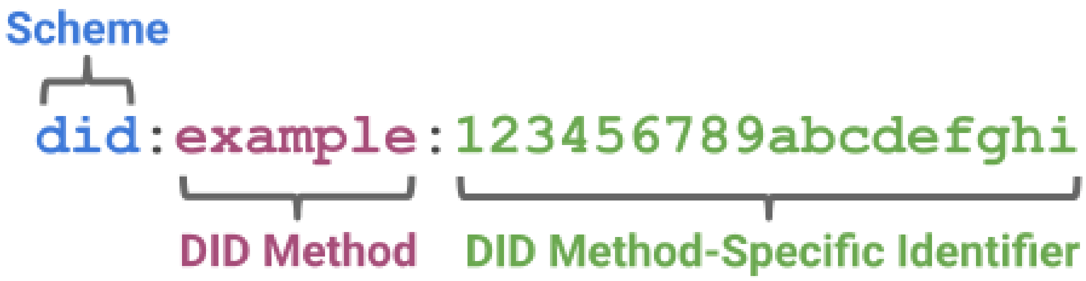
Scheme
did:example:123456789abcdefghi
DID Method
DID Method-Specific Identifier

Figure 2.3: DID composition [?]

[11] https://www.rfc-editor.org/rfc/rfc3986

2.4.2 DID URL

A DID can include a path, query and fragment to be able to locate a specific resource inside a DID document, as shown in **??**

2.4.3 DID Subject

Refers to the entity being identified by the DID. According to the specification [**?**], any person, group, organization, physical thing, digital thing or logical thing can be a DID Subject.

2.4.4 DID Controller

The DID controller is the entity that can make changes to the DID Document. This entity is not necessarily the DID Subject itself.

2.4.5 DID Document

They contain information associated with a DID. They usually describe verification methods, such as cryptographic public keys, as well as services that are relevant to interactions with the DID Subject. An example of a DID Document can be seen in **??**.

```
{
  "@context": "https://w3id.org/did/v1",
  "id": "did:example:123456789abcdefghi",
  "publicKey": [{
    "id": "did:example:123456789abcdefghi#keys-1",
    "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 2.10: Example DID Document

The only required attribute for a DID Document is the ID. Other optional attributes include a controller, which specifies the DID Controller; verification methods, used to authenticate or authorize interactions with the DID subject or associated parties; and services, which can be any type of service the DID subject wants to advertise, including decentralized identity management services for further discovery, authentication, authorization, or interaction.

2.4.6 DID methods

DID methods describe the processes for CRUD operations for DIDs and DID documents, based on a specific type of verifiable data registry. Each DID method describes and implements its own security and privacy considerations. According to the DID registry of the W3C[12], there are around 130 registered DID methods. Based on their characteristics and patterns, the following 4-way classification of DIDs has arisen.

- **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 correspondingly in the Bitcoin, Ethereum and Tron network [**?**].

  Examples include: did:ion, did:elem.
- **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, their validity is not affected since they comply with the core properties and functionalities that a DID has to provide.
- **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 registry, 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 deterministically extracted from the DID itself [**?**].

did:ethr

This DID method was registered by Uport, an Ethereum-based system for self-sovereign identity [**?**]. It allows any Ethereum smart contract or key pair account, or any secp256k1 public key to becoming a valid identifier [**?**]. UPort is a smart-contract-based system that abstracts user accounts using proxy contracts that are held by the user but may be retrieved if keys are lost. To do this, controller contracts create trusted entities for asserting ownership [**?**]

did:ens:

2.4.7 Verifiable Data Registry

Essentially, any system that enables capturing DIDs and returning required data to generate DID documents. This can be distributed ledgers, decentralized file systems, any type of database, peer-to-peer networks, or other types of trustworthy data storage [**?**].

2.4.8 DID resolver

A DID resolver is able to implement the DID resolution, which consists of taking a DID as an input, and giving a DID Document as an output [**?**]. As of the writing of this thesis, the Identifiers & Discovery Working Group (ID WG) has implemented a prototype Universal Resolver[13],

[12] https://www.w3.org/TR/did-spec-registries/#did-methods
[13] https://github.com/decentralized-identity/universal-resolver

which allows the resolution of DIDs for numerous DID methods, including all the examples mentioned above. In addition, this working group has also developed a Universal Registrar[14], which allows the creation, edition and deactivation of the DIDs across different DID methods.

2.5 DIDComm Messaging

2.5.1 Definition

The Hyperledger Foundation is an open-source collaborative effort intended to further develop blockchain technologies across industries [?]. 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 [?]. 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 [?].

Based on Didcomm v1, the Communication Working Group (CWG) of the DIF has implemented DIDcomm v2. Among other key differences between versions, such as formalizations of Didcomm v1 methods, Didcomm v2 has removed the special handling of Peer-DIDs in order to make all DIDs equal from the perspective of the DIDComm spec. Furthermore, the CWG pursues the standardization of DIDcomm 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. [?] [?]

From this point on, we are going to refer to DIDComm Messaging v2 only as DIDComm. 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. Moreover, Didcomm is a very versatile protocol, as it supports a wide range of features, such as security, privacy, decentralization, routable, interoperability, and the ability to be transport-agnostic [?].

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:

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. This duplex request-response interaction is, however, not always possible as many agents may not have a constant network connection or may interact only in larger time frames, or may even not listen over the same channel where the original message was sent. DIDComm's paradigm is asynchronous and *simplex*. Thus showing a bigger resemblance with the email paradigm. Furthermore, the web paradigm goes under the assumption that traditional methods for processes like authentication, session management, and end-to-end encryption are being used. Didcomm does not require certificates from external parties to establish trust, nor

[14] https://uniregistrar.io/

does it require constant connections for end-to-end transport-level encryption (TLS). Taking the security and privacy responsibility away from institutions and placing it with the agents. All of this without limiting the communication possibilities because of its ability to function as a base layer where opposed capabilities like sessions and synchronous interactions can be built upon. [?]

To achieve the encryption and signing processes mentioned in algorithm **??**, Didcomm implements a family of the Internet Engineering Task Force (IETF) standards, collectively called JSON Object Signing and Encryption (JOSE), which will be further explained in the next section.

2.5.2 JOSE Family

This RFC family includes both the JSON Web Signature (JWS) and the JSON Web Encryption (JWE) standards that are subclasses of the JSON Web Token (JWT), and JSON Web Key (JWK).

The second part of the DIDComm enablement, namely the signing and encrypting, will depend on what kind of security guarantees we want to achieve. DIDComm offers different approaches based on signing and two types of message 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 [?]. However, social networks like Ask.fm[15] or NGL[16] 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 public key authenticated encryption algorithm *ECDH-1PU*[17]. This algorithm j

This is due to the use of the

Elliptic-curve DiffieHellman (ECDH) protocol, which allows two parties to build a secure and private channel across an insecure and observable network. This protocol is part of the foundation of popular messaging apps such as Facebook Messenger, Whatsapp, Signal, and Didcomm V1 [?]. This key agreement protocol works by having both parties interchanging the public keys of their EC keypairs and some other public information. Using this public data and their private keys both parties can calculate a shared secret value, which it's the same for both parties. Any other observer or third party is not able to compute this shared secret without the private data [?].

An alternative to *Authcrypt* that also complies with the required confidentiality and non-repudiation requirements is to have a nested JWT. To achieve this, the plaintext is first signed and then the resulting JWS is used as the payload of a JWE. The algorithm **??** illustrates better the workings of this.

[15] https//ask.fm
[16] https://ask.fun
[17] https://datatracker.ietf.org/doc/html/draft-madden-jose-ecdh-1pu-04

Algorithm 1 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 $\mathrm{pk_b}$ 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 $pk_a$
```

In both alternatives, the identity of the sender can be confirmed and the plain text message remains encrypted and hidden from any third party that might manage to read the payload. As mentioned in **??**, the JWE can be sent through any unprotected protocol and still keep all of its advantages.

Compare with other encryption methods. (TLS, Whatsapp end-to-end, Signal) (Use image below)

3 Concept and Design

The standards presented in **??** show the potential improvements that can be achieved in key components of ActiviyPubbased social networks such as identity management, discovery, and communication. In this section, we are going to go through the different steps that are required firstly to integrate DIDs into an ActivityPubbased social network, and finally to enable DIDComm Messaging v2 for its communication. As explained in **??**, 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 ideas 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 and definitions of the ActivityPub implementation are introduced and described. Then, an example of a simple use case in a contemporary ActivityPub server is going to be illustrated and analyzed in order to be able to compare it with **??**, which presents the same use case but with the proposed concept and design that includes DID integration and DIDComm enablement, including reasons behind every decision made along with comparisons with other nonformalized proposals.

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 userfacing part of Mastodon, a Toot is the äquivalent of a Tweet on Twitter. This is a small status update with a 500character limit.
- **Status**: In the backend of Mastodon, the descriptor used for a Toot is a Status. Moreover, an account in Mastodon has a 1:n relationship with status.

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 Today's implementation

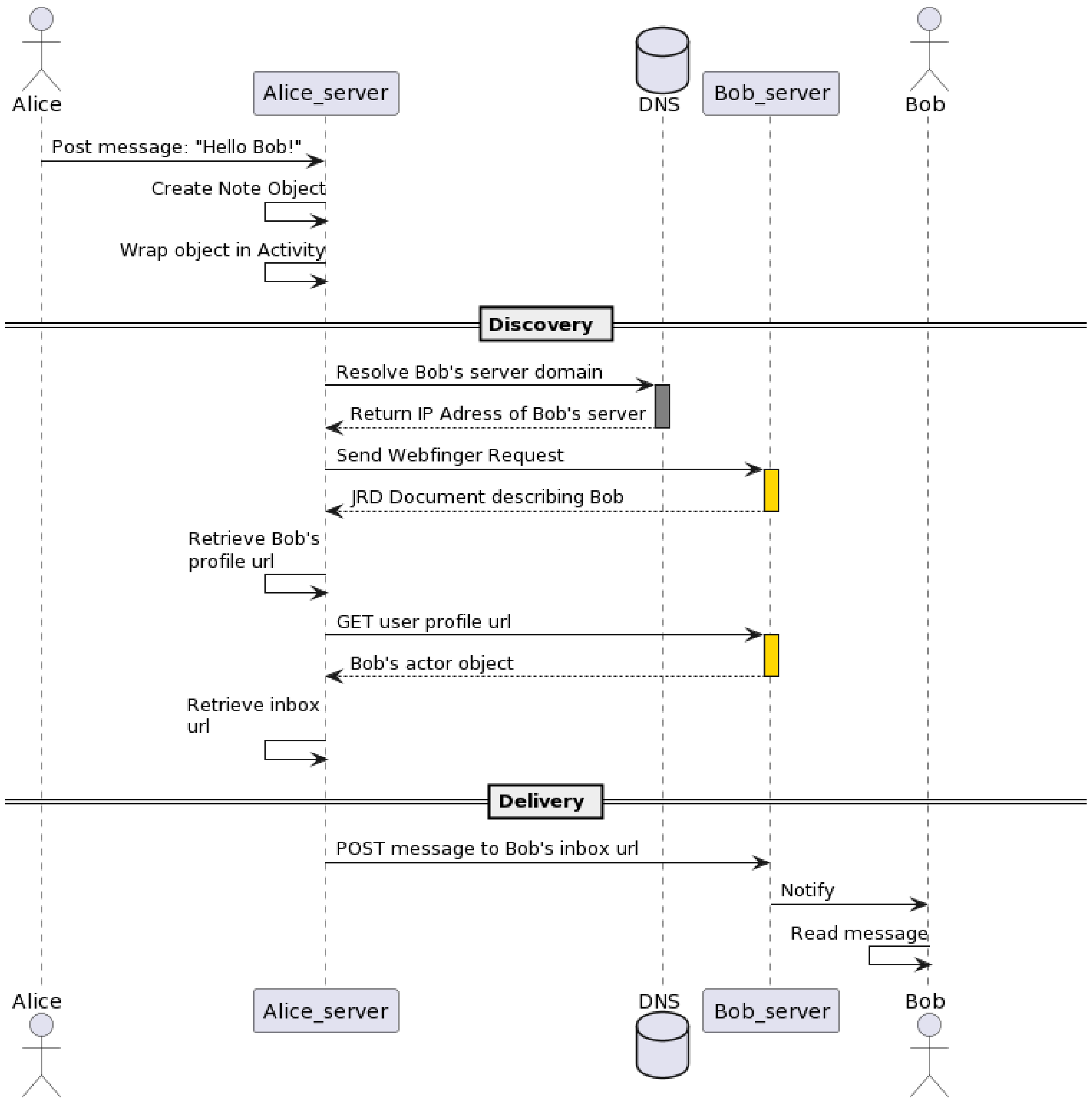

Figure 3.1: Current flow for sending message

3.3.1 Object 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 the ActivityPub server inside the Mastodon instance, as shown in **??**. Then, following the ActivityPub pattern of *some activity by some actor being taken on some object*, the server wraps it in an ActivityStreams *Create* activity, which contains Alice as the actor. This is illustrated by **??**. 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.

The ActivityPub server will now look at all the fields of the ActivityStreams *Audience*, which includes: to, bto, cc, bcc, and audience [?], where all the addresses of the recipients can be retrieved. Afterward, depending on where the recipient's account lives, the ActivityPub server may take one of two options. Even though the use case explicitly dictates that Bob's account resides in a different Mastodon instance, both cases will still be explained.

```
{
  "@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 Sameserver delivery

If the recipient's account is on the same server, there is then no explicit discovery process. A simple query in the ActivityPub's server would find the right account and save the status within the account's statuses.

3.3.3 Discovery

On the contrary, when the recipient's account is not on the same server, then a discovery process must be started. Discovery 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 discovery 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 **??**, Mastodon includes a series of wellknown endpoints that are used to retrieve information about the host itself, as well as resources or entities that are managed under the same host. By default, when the account is not found on the same server, a Webfinger query is performed.

In our case, Bob's account lives inside *bob_server.com*. The request shown in **??** will return a JRD Document, as shown in fig. **??**. Based on this document, the Mastodon instance retrieves the link with the *rel: 'self'* which includes the type and the href where Bob's actor object can be retrieved. If the Webfinger request returns a 404 code, it will then try, as a fallback, using the HostMeta endpoint. The request and response are displayed in **?? ??**. If successful, it will then proceed to take the link template provided and try the Webfinger request once more before throwing an error. After retrieving the needed URL, a subsequent HTTP GET request to this endpoint with the specific *application/activity+json* header will resolve Bob's actor object.

```
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: JRD Document

```
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

3.3.4 Delivery

Succeeding the retrieval of Bob's actor object and therefore the needed inbox URL, the delivery can now take place. In this case, an HTTP POST request with the previously generated activity object. To provide endtoend message integrity and to authenticate Alice in Bob's server, the request is signed by Alice's ActivityPub server using the HTTP Signature specification.

Finally, upon receiving the POST request to Bob's inbox URL, Bob's server verifies the validity of the signature using Alice's public key. After successful validation, it saves the Note object in Bob's statuses.

As indicated in chapter 2, HTTP signatures are not part of the ActivityPub protocol standard. These integrity and authentication features are within the Mastodon implementation of an ActivityPub server.

3.4 Proposed implementation

Proposed implementation

Having seen the current flow of our use case mentioned above in a working ActivityPub-based social network, this section will now address the first research question of this bachelor thesis. Namely, the implications of integrating DIDs to a Mastodon instance and therefore, to ActivityPub.

3.4.1 Implications of integrating DIDs

As mentioned in the definitions section, Mastodon's full username includes the domain of the server and a locallyunique username. This type of username accomplishes the goals of human readability and uniqueness. In addition, they are resolvable using DNS and the discovery methods previously mentioned in section **??**. An ActivityPub actor object using such a username is shown in **??**.

The first question that needs to be addressed when approaching the integration of DIDs to Mastodon and therefore ActivityPub is, what are the implications of switching from stan-

```

{
  "@context": [
      "https://www.w3.org/ns/activitystreams",
      "https://w3id.org/security/v1",
  ],
  "id": "http://alice_server.com/users/alice",
  "type": "Person",
  "following": "http://alice_server.com/users/alice/following",
  "followers": "http://alice_server.com/users/alice/followers",
  "inbox": "http://alice_server.com/users/alice/inbox",
  "outbox": "http://alice_server.com/users/alice/outbox",
  "featured": "http://alice_server.com/users/alice/collections/featured",
  "featuredTags": "http://alice_server.com/users/alice/collections/tags",
  "preferredUsername": "alice",
  "name": "",
  "summary": "",
  "url": "http://alice_server.com/@alice",
  "manuallyApprovesFollowers": false,
  "discoverable": false,
  "published": "2022-06-14T00:00:00Z",
}
```

Listing 3.7: Alice's actor object

dard mastodon usernames to DIDs. Integrating DID to ActivityPub points immediately to the actor's object. Making the switch would mean that the DID has to be included. Currently, most of the interactions of the ActivityPub server inside Mastodon require the ID attribute to resolve to the Actor's profile and thus the Actor's object. Following a simple strategy, we could simply replace the username with the DID. Thus having an ID attribute like *www.alice_-server.comusersdid example 123456789abcdefghi*.

However, there is another alternative that might work. Following the ActivityPub's specification, the ID attribute must be a publicly dereferenceable URI, whose authority belongs to the originating server [**?**]. As explained in **??**, a DID is a URI and it is publicly dereferenceable by nature. This allows different possibilities, to which the DID can be added. Take, for example, using a standalone DID as an ID attribute, to take advantage of the discoverability of DIDs. This scenario would have the following implications. If another ActivityPub Server wanted to simply get the actor's profile URL, it would require resolving the DID to its respective DID Document, adding the need to parse it and to methodically do a search until the endpoint is found and retrieved. This additionally requires that the DID Document includes the actor's profile in the services section. This modification in the DID Document will be further discussed in the Discovery 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. Simplifying in this manner the work that the ActivityPub server must do, 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 attribute that requires the actor's profile URL in case it is not in the ID attribute [**?**]. Although plausible, for this Thesis we will keep a simple replacing strategy and keep the ID attribute as the profile's URL with the username replaced with the DID.

DID URLs provide a lot of freedom of usability for DIDs. In addition to the ID, the actor's object must provide a supplementary set of URLs that point to different collections related to the Actor. These include mandatory attributes, like the outbox and the inbox, and other optional attributes such as the followers and following attributes. What if, instead of using the actual URL of these collections, we specify DID URLs that then point to the correct endpoint inside the DID Document. This example is illustrated in fig. (Activitypub actor with all DID URLs).

This approach leads us to the question, isn't it simpler just to shift the whole actor's object endpoints directly to the DID Document? Therefore removing the need for an ActivityPub actor's object. This idea was briefly suggested by [**?**] 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. The authors however failed to offer an adequate explanation and consideration of the ActivityPub protocol in existing implementers. Moreover, there are some security privacy concerns regarding the use of service endpoints in the DID Document. The DID specification stipulates *"revealing public information through services, such as social media accounts, personal websites, and email addresses, is discouraged"* [**?**]. DID Documents are stored in a publicly available verifiable data registry, 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. Looking at Fig(fig. DID Document with all ActivityPub endpoints), 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. When creating a username there are some validations made to it, such as length, regex and uniqueness. All of this is contained inside the Account class, which is the object that abstracts all of a user's account. An example of the Mastodon instance that uses DIDs instead of standard usernames is illustrated in fig. (Image of Mastodon with DIDs).

3.4.2 New discovery process

As stated in the previous section, Mastodon starts the discovery process based on the username of the user. By replacing the standard username with a DID, the current discovery flow gets disrupted, as there is no domain and thus no well-known endpoints to send discovery requests to. Nonetheless, here is where the discoverability and always resolvability of the DIDs come into play. The proposed flow of discovery 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. An example can be the Universal Resolver from the DIF mentioned in **??**. The DID Document must now contain a service section with the type ActivityPub and a URL, 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 [**?**]. Webfinger's purpose shares a lot of ground with the discoverable design of DIDs, nevertheless, the DID design provides a less limited structure of discovery, 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 discovery protocol Webfinger used in Mastodon and include the URL of the actor's profile in the ActivityPub service section in the DID Document.

3.4.3 Enabling DIDComm Messaging

Because DIDs are now implemented in our ActivityPub prototype, it is now possible to enable the DIDComm messaging protocol. Taking into account the algorithm **??** shown in **??**, it is possible to derive some requirements that DIDComm imposes:

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 [**?**]. 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 our proposal.

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 plain-text 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.

Access to DID-Resolver: The ActivityPub server must have access to a DID-Resolver to be able to retrieve DID Documents. Especially in cases where an incoming message has been signed, and the signature needs to be verified using a specific public key. This requires extending the Mastodon server by adding possibly a service class that performs calls to a DID-resolver and parses the returned DID Document to process the information in it.

3.4.3.1 ActivityPub as DIDComm payload

As part of this proposal, we do not want to do further changes to the ActivityPub protocol itself. 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. DIDComm presented 3 different kinds of message structures, namely plain text (JWM), signed (JWS) and encrypted messages (JWE). The following sections present how the encapsulation, as well as signing and encryption processes, work in this proposal.

ActivityPub as JWM: Let's take our previous example of an ActivityStreams object where Alice sends a message to Bob. The simplest way to keep a modular approach is by using the ActivityStream object as a payload for our JWM, which would look like fig. (JWM with activity). As explained in chapter 2, the JWM specification defines a series of attributes that provide a starting point for the use of JWMs. Not all of these are mandatory and can thus be modified depending on the use being given to them. This gives us a lot of room to think about what is necessary and what is not. As seen in fig. (JWM with activity), redundancy can be found in each level of the JSON object. For example, the sender and recipient are defined in each layer. In the end, the Activity is the only object that will be processed, therefore, the JWM does not necessarily need any extra information, because the Activity has already all the necessary information. Furthermore, even if we wanted to use the routing mechanism of DIDComm, the attributes From and To in the JWM would only be necessary in cases where we want to send plain text messages. In addition, in almost every case the message is being sent to a specific URL, like the inbox of another user. So the receiver server will always know to whom the Activity was directed. This leaves us with a JWM structure with the attributes id and type. The requiredness of these two differs from the JWM specification and DIDComm. Nonetheless, we will stick to DIDComm and keep these two attributes compulsory. The type should be a message type URI, therefore we will use the ActivityStreams schema URI, and for the ID, we will reuse the status ID. The result is shown by figure **??**

The final flow for our use case mention in**??** is illustrated in figure **??**

```
{
  "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

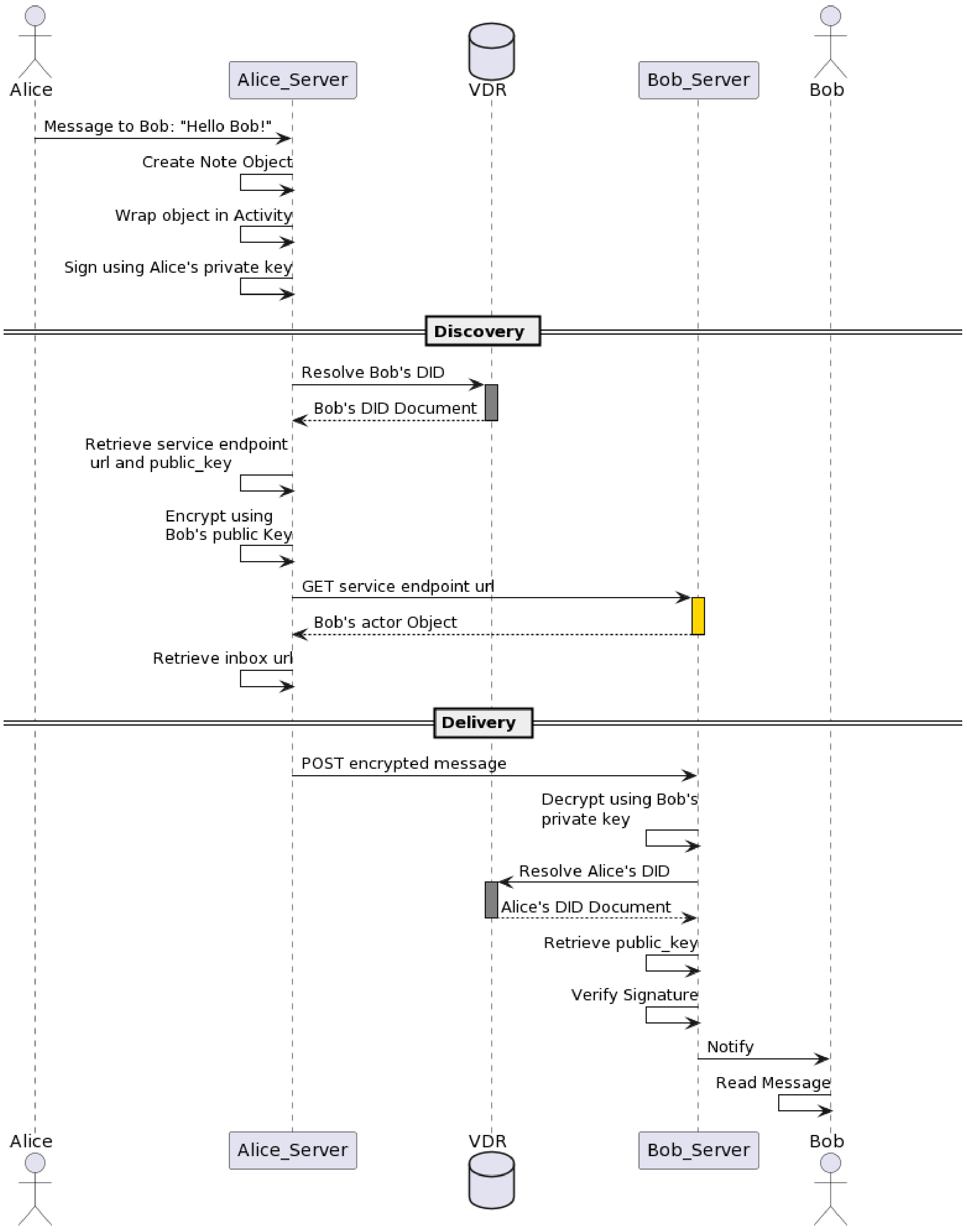

Figure 3.2: DID and DIDComm flow for use case

4 Implementation

4.1 Mastodon

The source code[1] of Mastodon is open source and accessible for everyone to download. The requirements to run a mastodon server include Ubuntu 20.04 or Debian 11 for the operating system, a domain name, an email delivery service, and an object storage service optionally. The prototype was implemented using two different servers. The first was a Linux server with Ubuntu 20.04, 2 CPU cores, 4GB RAM, and 80 GB storage capacity provided by the cloud provider Linode[2]. The TU Berlin provided the second one. However, the latter was mainly used for debugging requests and testing the new discovery process. 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. The whole project is containerized using docker, and a docker-compose.yml file for production is given. A Vagrant file is also provided for development, but for this Thesis, a custom dockerfile was written to run Mastodon in development mode. The core backend was implemented using the Model-View-Controller (MVC) server-side framework Ruby on Rails (RoR), which, as the name says, is based on the programming language ruby. The backend manages a Postgres relational database, implements the ActivityPub server, provides a REST API for the frontend, and serves web pages in the about-endpoint. In addition, Mastodon offers a node.js-based streaming API that sends real-time updates through long-lived HTTP connections or WebSockets [**?**]. The frontend of Mastodon was developed using the React framework, which manages most of the dynamic parts of the social network.

4.1.1 DIDs

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. A DID-syntax-compliant regular expression was used for the prototype, and the maximum length was incremented to 85 characters.

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

The universal resolver of the DIF was added to the docker-compose file to allow resolving DIDs. It consists of the main service, the query endpoint, and a driver for each DID method.

[1] https://github.com/mastodon/mastodon
[2] https://linode.com

The resolver was added to the same docker network as the main backend application, which meant the access through the alias name *did-resolver* was possible. Figure **??** shows a sample request to the 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. The service class *DidResolverService* was added to the backend to make the requests to the resolver. The only parameter it needs is the DID it needs to resolve, and it returns the DID Document as a JSON object. Furthermore, a class *DidDocument* was added with the methods listed in table **??**, to facilitate the interaction with the properties of the DID document

```
GET /1.0/identifiers/<did>
Host: did-resolver:8080
Accept: application/json
```

Listing 4.1: Example request to the DID resolver

heightFunction	Description
heightheightinitialize(attributes)	Stores the attributes of the DID document in instance variables
heightserviceEndpoint	Returns the service endpoint URL of the first service
heightrsaDidcommKey	Looks for a specific key in the DID document and returns an RSA instance of it
height	

Table 4.1: DID document instance methods

With DIDs, a resolver, and a class for DID documents set up, it is time to modify the discovery process of Mastodon, which relies on Webfinger. 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. It is important to clarify that even if the account is saved locally, it will be flagged as *not local*.

The new class handling the decentralized DID-based discovery 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 running locally. 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.

4.1.2 DIDComm

Now that finding accounts from other servers using DIDs is possible, it is time to enable DIDComm. The biggest challenge here was the compatibility between the algorithms specified by the JWA spec, the key types, and the libraries used to generate keys, sign, and encrypt. OpenSSL[3] for Ruby is the library used for generating the RSA keys. This library wraps the

[3] https://github.com/ruby/openssl

OpenSSL project toolkit[4] and provides a wide range of key management, encryption, decryption, and certificates management. To create signed and encrypted messages, the JWT library[5] for ruby on rails was added to the backend. This library allows the spec-compliant creation of JWS and JWE tokens. However, one of the drawbacks was the limited number of algorithms for JWE encryption. As the ECDH algorithm was not available during the development of this prototype, RSA key types and

The JWS tokens in the prototype were created using the *RS256* algorithm and the digital signature algorithm *RSASSA-PKCS1-v1_5 using SHA-256*[6].

4.2 Creating and editing a DID document

Creating a DID is rather a simple task. This prototype's main challenge was 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. My first option was MATTR[7], which offers creating DIDs using *did:key*, *did:web* and *did:ion* methods. However, they would not allow creating own keys or accessing private keys, which is necessary according to **??**. Furthermore, any editing of the DID document was also restricted. Independent from MATTR, *did:ion* offers a set of tools[8] 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[9], 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 owner full control. This way, any new field or attribute could be added without the burden of making a third-party library work. The DID is *did:web:lisztos.com* and an example of my self-hosted DID Document can be seen in figure **??**.

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 our 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[10] and Veramo[11]. 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 customer support, it turned out that they were having problems with the IAM permissions, and it was temporar-

[4] https://www.openssl.org/
[5] https://github.com/jwt
[6] https://www.rfc-editor.org/rfc/rfc3447
[7] https://mattr.global
[8] https://github.com/decentralized-identity/ion-tools
[9] https://github.com/decentralized-identity/ion-tools/discussions/25
[10] https://serto.id
[11] https://veramo.io

ily not available. On the other hand, Veramo offers a typescript-based API that allows users to manage DIDs not only in the Ethereum leading 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[12], where the only thing needed externally is an Infura[13] account to use it as a Web3 Provider. Two DIDs were created in the Ropsten network for Alice and Bob, respectively. Figure **??** shows the default DID document that gets created when creating a new DID.

- **Alice:**

  did:ethr:ropsten:0x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468
  e6a00cc6c4d.

- **Bob:**

  did:ethr:ropsten:0x03117951c6011b4a46f11a67fc7f67f746a7ad84daaae69623db833d
  dd56397c37

4.2.1 Adding a service

For DIDComm to work in our proposed implementation, a service endpoint with the profile URL must be added to the DID document of Bob and Alice, respectively. The parameters required for Veramo's API to process the information correctly, as shown in figure **??**, are the DID, the service object, and options for the Web3 provider. The service object includes a type, the service endpoint, and a description. The service's id is optional, as the Web3 Provider will overwrite it.

4.2.2 Adding a key

Finally, the last required editing task in the DID document is to add a public key. As explained in

[12] https://veramo.io/docs/node_tutorials/node_setup_identifiers
[13] https://infura.io

```
 {
   "context": "https://www.w3.org/ns/did/v1",
   "id": "did:web:lisztos.com",
   "publicKey": {
       "id": "did:web:lisztos.com#main-key",
       "controller": "did:web:lisztos.com",
       "type": "RSA",
       "publicJwk": {...}
   },
   "authentication": [
       "did:web:lisztos.com#main-key"
   ],
   "assertionMethod": [
       "did:web:lisztos.com#main-key"
   ],
   "capabilityDelegation": [
       "did:web:lisztos.com#main-key"
   ],
   "capabilityInvocation": [
       "did:web:lisztos.com#main-key"
   ],
   "keyAgreement": {
       "id": "did:web:lisztos.com#main-key",
       "controller": "did:web:lisztos.com",
       "type": "RSA",
       "publicJwk": {...}
   },
   "service": [
       {
           "id": "did:web:lisztos.com#ActivityPub",
           "type": "ActivityPub",
           "serviceEndpoint": "https://lisztos.com/users/@did:web:lisztos.
  com"
       }
   ]
}
```

Listing 4.2: did:web DID Document

```
{
"@context": [
  "https://www.w3.org/ns/did/v1",
  "https://w3id.org/security/suites/secp256k1recovery-2020/v2"
],
"id": "did:ethr:ropsten:0x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d",
"verificationMethod": [
  {
    "id": "did:ethr:ropsten:0x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d#controller",
    "type": "EcdsaSecp256k1RecoveryMethod2020",
    "controller": "did:ethr:ropsten:0x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d",
    "blockchainAccountId": "eip155:3:0xcDC3B55934073f7BCA8a34d0561006CC1f26E9Fa"
  },
  {
    "id": "did:ethr:ropsten:0x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d#controllerKey",
    "type": "EcdsaSecp256k1VerificationKey2019",
    "controller": "did:ethr:ropsten:0x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d",
    "publicKeyHex": "023ea34111106fc45001a76aed5681f8187c56d6eaf0a55fc9af92d11c2732c2c8"
  },
],
"authentication": [
  "did:ethr:ropsten:0x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d#controller",
  "did:ethr:ropsten:0x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d#controllerKey"
],
"assertionMethod": [
  "did:ethr:ropsten:0x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d#controller",
  "did:ethr:ropsten:0x031be4622770a8ee4a7b25d1673e829fd2eb5f4762efcb18d09d468e6a00cc6c4d#controllerKey",
],
"service": []
}
```

Listing 4.3: Initial DID document for Alice

```
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 4.4: Parameters to add a service in Veramo

5 Evaluation

The evaluation of the thesis should be described in this chapter

6 Conclusion

Describe what you did here

List of Tables

List of Figures

Appendices

Appendix 1

```
for($i=1; $i<123; $i++)
{
    echo "work harder! ;)";
}
```

Listing 1: NodeInfo response for mastodon.social