University of Trento

Department of Information Engineering and Computer Science

Bachelor's Degree in Computer Engineering

~ · ~

Academic Year 2021–2022

SELF-SOVEREIGN IDENTITY
Custom DID method

Supervisors
Prof. Fabrizio Granelli
Dr. Mattia Zago

Graduate Student
Marco Xausa
211373

Final examination date: September 19, 2022

Dedico questa tesi ai miei genitori,
per avermi insegnato a non lasciare mai nulla di intentato

Abstract

The internet has always been divided into different versions with respect to the technologies it encompasses. The actual version, Web2.0, is based on centralized standards where an entity, usually a company, manages users' data for different purposes. This structure is reflected in many fields, including Identity and Access Management (IAM).

However, with the Web heading to its third version, new objectives have to be reached: avoid centralized services and leave to users as much control as possible over their data. Since typical IAM standards do not aim for such achievements, a completely redesigned model is necessary.

A solution to such a requirement is provided by Self-Sovereign Identity (SSI). It is composed of two main elements: Verifiable Credential (VC) and Decentralized Identifier (DID). VCs are lists of claims stored by the user which refer to an entity by means of DIDs. A DID is a unique string that can be used to retrieve information to communicate with a physical entity. No centralized entity should be involved in its usage, though this is not mandatory thus allowing different implementations.

This document's main focus is to study DIDs in order to define a new method called `did:monokee`. Its main purpose is to aggregate different DIDs from different environments to help the interpolation between already existing infrastructures. The design choices have been described alongside the operations and the reference implementation.

Acronyms

ABNF Augmented Backus-Naur Form. 4

AID Autonomous IDentifier. 32, 33

API Application Programming Interface. 22, 24, 26, 27, 31

CIE Carta di Indentità Elettronica. 2

CRUD Create, Read, Update, Delete. 6, 9, 10, 16, 24, 31

DI Decentralized Identity. 2, 3

DID Decentralized Identifier. ii, 3–11, 16–20, 22–24, 26, 27, 29–34

DIF Decentralized Identity Foundation. 2, 6, 32, 35

DNS Domain Name System. 32

IAM Identity and Access Management. ii, 1–3, 23

IDP Identity Provider. 1, 2

MFA Multi Factor Authentication. 2

SAML Secure Assertion Markup Language. 2

SCID Self-Certifying IDentifier. 32, 33

SP Service Provider. 1, 2

SPID Sistema Pubblico di Identità Digitale. 2

SSI Self-Sovereign Identity. ii, 1–3, 9, 23, 31, 32, 34

SSO Single Sign On. 1, 2

ToIP Trust over IP. 2

URI Universal Resource Identifier. 3

URL Universal Resource Locator. 5, 9, 21, 26–28

VC Verifiable Credential. ii, 3, 7, 9, 23, 34

VDR Verifiable Data Registry. 3, 6, 9, 16, 22–24, 31, 32

W3C World Wide Web Consortium. 2

Contents

Introduction

Nowadays, the internet allows its users to perform an enormous number of operations. They can be very different, ranging from reading books to bank transfers, and can involve data belonging to different people. For this reason, the access to some information and the ability to perform certain operations has to be restricted with some form of authentication and authorization.
The different web versions trusted different Identity and Access Management (IAM) standards. Web1.0 was based on the simple login and password paradigm, which still represents a basis for today's standards. The next version, Web2.0, introduced a new pattern, the Single Sign On (SSO) as well as Identity Provider (IDP) and Service Provider (SP), enabling the outsourcing of both authentication and authorization procedures. But recently, with the diffusion of the Web3.0 model, a new standard is needed. Thereby, a new identification system has been created, the Self-Sovereign Identity (SSI), which aims to bring decentralization to identity management.

SSO has been the first disrupting change to IAM, and is considered the state of the art in terms of security and usability. It brought a simplification to internet surfing since the authentication had to be performed only once for several different services. New entities were added in the authentication flow, for example, a SP exposes services that require authentication. If a user tries to access a service, he will be redirected to the login page of the IDP. The user will thus provide his username and password and will eventually be authenticated. The IDP informs the SP that the user has been recognized. The SP hence will provide a token to the user that can be used to prove the successful authentication.

A variety of identification types are available, based on:

- possession, something that the user owns. It can be lost and duplicated. Valid objects could be:
  - tokens (*e.g.*, RSA token, Yubikey, RFID, and acoustic/magnetic token)
  - device/location (*e.g.*, a known device)
  - document/smartcard (*e.g.*, ID card, x509 certificate)
- inherence, something that only the user is. In this case, it can not be changed, shared, or repudiated. Some examples are:

  - biometrical data (*e.g.*, facial recognition, iris/retina recognition, fingerprint)
  - behavioral data (*e.g.*, app usage)

- knowledge, something that the user knows. It can be shared, forgotten, and changed. Types of secrets are:
  - private secret (*e.g.*, password, pin or graphical password)
  - public secret (*e.g.*, access code)

These methods could be combined to create Multi Factor Authentication (MFA) systems. Normally, they leverage a group of two or more proofs from the above categories to provide a greater level of security and reliability to the authentication process.

In addition, authentication can be further subdivided in:

- legacy authentication *i.e.*, authentication protocols where the user sends credentials directly to the resource (*e.g.*, IMAP, SMTP, etc.)
- modern authentication *i.e.*, methods based on the SSO workflow, where IDP and SP are separated entities, providing a better and safer experience (*e.g.*, Secure Assertion Markup Language (SAML)).

During the last decade, these protocols have been leveraged by the government to create multiple identity standards as Sistema Pubblico di Identità Digitale (SPID) and Carta di Indentità Elettronica (CIE). Both provide SSO experiences, but the first uses SAML, whereas the second exploits X.509 certificates. However, these systems have the necessity to identify the user through a delicate enrollment process, which might be compromised or otherwise disabled. In addition, users' data has to be stored in a centralized infrastructure, that is affected by the single point of failure weakness. Besides that, the collected users' information and metadata can be used by the IDP for profit intents.

For these reasons along with Web3.0, a new digital identity, the Self-Sovereign Identity (SSI), is being developed. The focus is on decentralization and leaving to users more control of their information.

Decentralized Identity (DI), which is a form of SSI, represents a new paradigm leveraging distributed ledger technology. Data is no longer stored in centralized storage, but rather by the user. To ensure a chain of trustworthiness, public ledgers are exploited (*e.g.*, blockchains). This permits the user to own his information and provide it by himself without any request to a centralized entity, hence making decentralization the main strength of this system. Since DI represents an innovation to IAM, it is still being defined, and a lot of work is being put in place by different associations, international organizations, and government bodies - such as World Wide Web Consortium (W3C), Decentralized Identity Foundation (DIF) and Trust over IP (ToIP) - to define

the standards for this technology and to ensure the interoperability among the different platforms that will implement this methodology.
DI is structured as a layered system, where lower layers provide services to higher layers. The objective of this structure is to create a trust framework encompassing the layers and conforming to privacy laws.

At the top of this stack, there are applications and wallets. Any kind of application can be built at the top of this pile. The most used kind is wallets. Wallets store private keys and seed phrases to help the user manage his identity information, blockchain addresses, or multi-factor logins. Some of them, identity wallets, can store digital claims. They are slightly different from crypto wallets, which are meant to store only blockchain addresses. Moreover, another differentiation is made between custodial and non-custodial wallets. Custodial wallets are third-party applications that store the private keys on the tenant's behalf, while non-custodial wallets store the private keys locally.

Below the application and wallet layer, there is the agent layer. At this stage, agent frameworks allow lightweight apps (single-page apps or decentralized apps) to interact directly and discretely between each other or with a VDR, with a lower level of risk than traditional cloud and server infrastructure. Essentially, agents work as a bridge between applications and VDRs and between applications themselves. Agent frameworks lay upon Verifiable Credential (VC), which are the core component of this system. They are a set of claims that can be stored by the user on a digital wallet and disclosed at need. They are issued by a trusted authority to the subject they refer to, and can be presented to a verifier who can check their validity using cryptographic primitives.
A VC refers to a Subject by means of a Decentralized Identifier (DID).

DIDs are globally unique identifiers that can be used to retrieve information about a subject (a person, a company, or an object). Universal unique identifiers are used in a variety of circumstances. They can be used as ID numbers, product identifiers, and URIs. Typically they are not under the user's control, and they are released by external authorities who decide who or what they refer to and whether they can be revoked. DIDs can be resolved to obtain a DID document that reports a set of cryptographic keys and methods to communicate with the subject and to verify eventual information contained in a Verifiable Credential.

Monokee Srl, a company offering IAM services, decided to integrate a SSI products for its customers, helping enterprises in the transition to a decentralized pattern. During my internship, I was asked to study the Decentralized Identity and DIDs in order to develop a new DID method that could fulfill specific requirements such as working with the already existing Monokee IAM infrastructure. The project required the definition of a DID method that could encompass different DIDs from different contexts to help the interpolation between different SSI infrastructures. The final objective was to have a typescript implementation that could allow the full utilization of this DID.

Chapter 1

Context

This section examines DIDs, DID methods, and their structure as defined by the DID core documentation [9].

A DID is a unique string that functions as a decentralized address that refers to a DID subject. It is owned and controlled directly by one or more subjects (called controllers) and can be resolved to a DID document. It is called decentralized because it is registered and resolved on autonomous "namespaces" (not controlled in a centralized way).

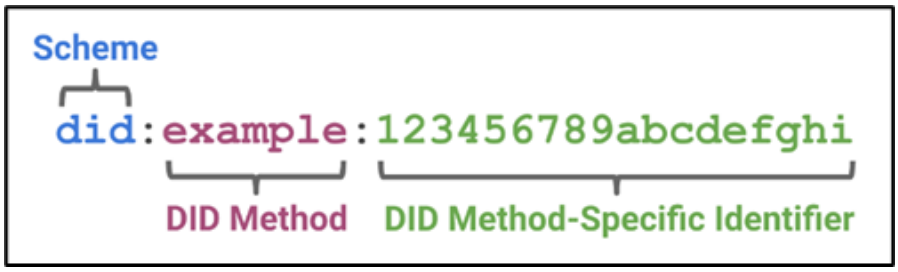

Figure 1.1: Structure of a DID

Figure 1.1 represents the structure of a DID. Every DID starts with the `did:` annotation, followed by the representation of the DID method. The last part is a unique identifier belonging to a specific namespace described by the DID method. The syntax is described by the following Augmented Backus-Naur Form (ABNF) [3]:

```
did                = "did:" method-name ":" method-specific-id
method-name        = 1*method-char
method-char        = %x61-7A / DIGIT
method-specific-id = *( *idchar ":" ) 1*idchar
idchar             = ALPHA / DIGIT / "." / "-" / "_" / pct-encoded
pct-encoded        = "%" HEXDIG HEXDIG
```

Since each DID must be unique, a specific mechanism that ensures uniqueness for each method-specific identifier must be chosen. Depending on the requirements and the design choices, the method-specific identifier could be set as a blockchain address (common option) or an encoded public key that is used to produce the document (for self-contained DID).
To mention some examples:

- `did:bnb` [1] leverages the blockchain address. A valid DID could be:
  `did:bnb:1f4B9d871fed2dEcb2670A80237F7253DB5766De`
- `did:key` [8] uses the encoded public key as method-specific identifier. This is used to expand the DID document. A valid DID could be:
  `did:key:z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK`

From a DID a DID URL can be created. It is a location identifier that can be used to retrieve specific information inside a DID document, for instance, a verification method, a service, a specific part of a document, or even an external resource. A DID URL is created following the below ABNF:

```
did-url = did path-abempty [ "?" query ] [ "#" fragment ]}
```

for example:
`did:example:123/path?service=agent&relativeRef=/credentials#degree`

As mentioned, a DID can be resolved to a DID document. A DID document is a set of data associated with a DID. It can contain public keys and algorithms to verify the identity and communicate with the DID subject. Moreover, it permits a DID controller to demonstrate ownership over a given DID.

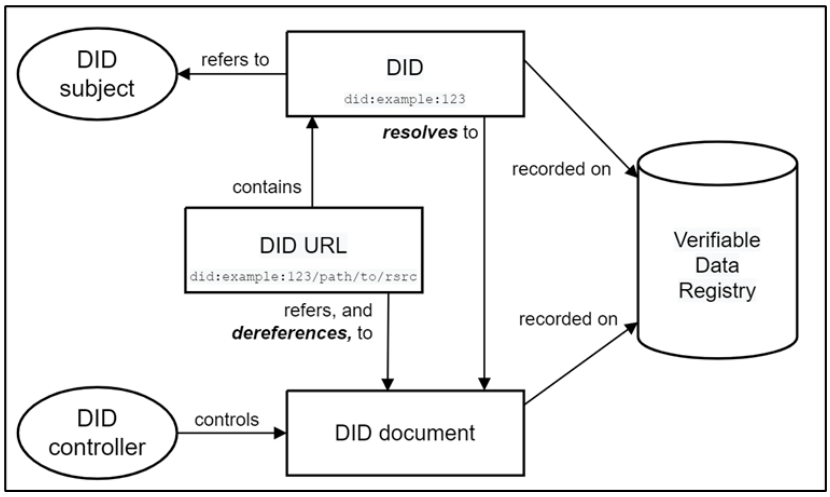

Figure 1.2: DID ecosystem

As reported on the Figure 1.2, each DID is registered in a Verifiable Data Registry (VDR). A VDR is a system that supports recording DIDs and returning data necessary to produce DID documents (*e.g.*, Distributed ledger, distributed database and so forth). Usually one DID method interacts with only one VDR, however, it could work anyway with more than one.
Despite that, some DIDs are "self-contained" and do not need an interaction with a VDR because of their nature, as examples: `did:key` and `did:jwk`.

No specific requirements are indicated for VDRs, however, there is a definition with functional requirements in the DID core specification [9]. As reported on DIF FAQ [2], the degree of decentralization of a DID depends on how it is managed. The CRUD operations of a DID method are applied to a VDR. The first example of VDRs are Distributed Ledgers. They are the most used technology among DID methods. They are distributed and publicly verifiable, and data stored there cannot be modified unless another entry is registered (they keep track of all the modifications that took place). Throughout this document, for all intended purposes, terms such as edit, modify, or other synonyms will be used to refer to the act of updating a record in a VDR accordingly. Although the above solution seems the most logical, DID methods might rely on centralized trusted service providers and merkle proofs [13] to provide tamper-evident logs without the costs associated with distributed consensus as contained in the DID implementation guide [12]. This is not a decentralized solution, but it is tamper-evident and therefore can be assumed as a valid VDR. As a result, it seems that the degree of decentralization of a DID can vary in the range of values between the solutions exposed above. However, in the DID core specification [9] it is reported that DIDs should avoid centralized authority and single points of failure, hence the second solution might not be the best choice, though it is valid. A good discussion has been made about that by DIF in its official blog [5]

Core properties and resolution

A DID document has a set of properties whose structure has been described in the DID core documentation [9]. These attributes are the basic set to consider when describing a document configuration in a DID method.

The core properties are:

- `id` - the only required field. It represents the DID which generates the document.
- `alsoKnownAs` - a collection of alternative identifiers that refer to the same subject as the document.
- `controller` - array of DIDs that expresses those subjects allowed to modify the document. If not present, the controller is the subject referred by the `id` property.

- `verificationMethod` - array of objects defined using the following schema:
  - `id` - unique identifier for the method
  - `controller` - method controller
  - `type` - type of method
  - a public key that can be represented by one of the following:
    * `publicKeyJwk`
    * `publicKeyMultibase`

  These methods can be used for different purposes which can be expressed in a DID document using verification relationships. They are different, and represent the boundaries in which a verification method can be used:

  - `authentication`: specifies how the DID subject is expected to be authenticated
  - `assertionMethod`: specifies how the DID subject is expected to express claims (*e.g.*, for issuing Verifiable Credentials)
  - `keyAgreement`: specifies how an entity can generate encryption material in order to transmit confidential information intended for the DID subject
  - `capabilityInvocation`: specifies a verification method that might be used by the DID subject to invoke a cryptographic capability
  - `capabilityDelegation`: specifies a mechanism that might be used by the DID subject to delegate a cryptographic capability to another party

- `service` - set of services that can be used to communicate with the DID subject. Each service contains the following set of properties:
  - `id`
  - `type`
  - `serviceEndpoint`

  furthermore, it can define new fields specific to that service.

The possible values for certain properties are described by the DID specification registries [14], such as the available types of verification methods. Alongside the core properties, there are core representation-specific entries, which are helpful only for representation purposes.

The `id` is the only required field. This might be misleading since the controller field might not be present, or could be removed after the DID creation using an updating capability, resulting in an ungovernable DID. It is necessary to point out that a DID method must describe how authentication has to be performed, and should not implement functions leading to loss of control over a DID. As a result, since authorization to modify a DID document is typically performed using a verification method, at least one

should be indicated in the document, and should not be removed by further actions.

A serialization of the previous properties is called representation. It is obtained through a process called production. The consumption process resolves a representation to a data model. Different representations are possible: JSON, JSON-LD, and CBOR.

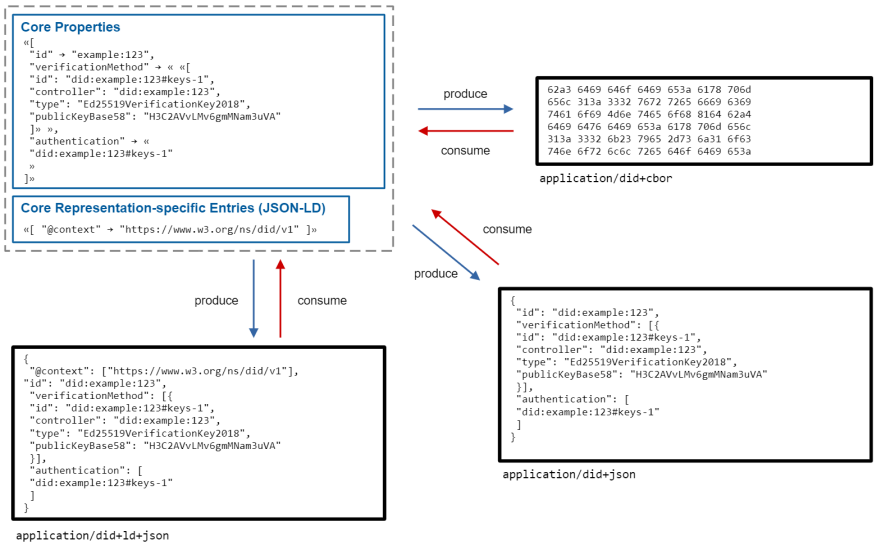

Figure 1.3: Illustration of the different representations and the production and consumption processes

When resolving a DID, either a data model or a representation is returned. Anyway, there is a certain amount of metadata coupled with the result. This metadata represents information about the document and the resolution process. It is divided in two different parts: `didDocumentMetadata` and `didResolutionMetadata`. The first contains information such as the creation date or the last update date, while the second usually contains information about which representation is being returned.

DID method

A DID method represents the governance of a DID. It is used to obtain the related DID document and to create a new DID.
Each DID method is a public micro-protocol containing:

- namespace rules
- crud and resolution mechanics
- references to all dependencies

Each DID method has its own characteristics and infrastructures, with specific strengths and weaknesses.

The DID core documentation [9] defines the list of requirements to implement a DID method. They determine how to establish: syntax, operations, security, and privacy. First of all, a DID method must define a `method-name` as expressed in the DID's ABNF rule. After that, the method has to describe a way to obtain the `method-specific-id` which must ensure local uniqueness for each identifier, since each DID must be unique within the method, and thus globally. Furthermore, the method must outline how the various paths, queries, and fragments are used while dereferencing a DID URL, and which of these are valid. The dereferencing process converts the DID URL to the respective resource and returns it to the requester. Once the syntax constraints have been defined, the method must describe the CRUD operations that can be performed on a DID and how they are authorized. Lastly, some considerations about security and privacy must be included in the documentation to describe the main threats to the system.

1.1 State of the art

To understand how the above requirements can be used to define a DID method, the following section describes two well-documented methods: `did:sov` and `did:key`. The first represents an example of a method interacting with a VDR. Indeed, the Sovrin VDR is a SSI-dedicated ledger specifically designed to store DIDs and VC schemas, and the method's documentation includes complete examples of ledger interactions for the different operations. On the other side, `did:key` provides the most straightforward implementation of a DID method that is able to achieve many, but not all, of the benefits of utilizing DIDs. These specifications are also guidelines to proceed with a code implementation of the methods, so they are general and not related to any programming language.

1.1.1 `did:sov`

`did:sov` is a DID based on the Sovrin public ledger. This ledger is specific for SSI, hence optimized for storing DIDs, DID documents and the necessary metadata. It is governed by the Sovrin Foundation [11].

Method syntax

As mentioned, every method definition starts with the syntax specification for both the method name and the method-specific identifier. The string that shall identify this DID method is: `sov`, while the namespace-specific identifier is identified using the following ABNF rule:

```
sovrin-did     = "did:sov:" idstring *(":" subnamespace)
idstring       = 21*22(base58char)
subnamespace   = ALPHA *(ALPHA / DIGIT / "_" / "-")
base58char = "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9" / "A"
/ "B" / "C" / "D" / "E" / "F" / "G" / "H" / "J" / "K" / "L" / "M" / "N"
```

```
    / "P" / "Q" / "R" / "S" / "T" / "U" / "V" / "W" / "X" / "Y" / "Z" / "a"
    / "b" / "c" / "d" / "e" / "f" / "g" / "h" / "i" / "j" / "k" / "m" / "n"
    / "o" / "p" / "q" / "r" / "s" / "t" / "u" / "v" / "w" / "x" / "y" / "z"
```

All Sovrin DIDs are base58 encoded of a 16-byte uuid. The encoding uses most of the characters with exception of "0OIl" to avoid readability problems. The result is a case-sensitive string of either 21 or 22 characters. The 16-byte uuid can be generated in various ways with standard uuid methods.
An example of a valid `did:sov` is: `did:sov:2wJPyULfLLnYTEFYzByfUR`.

CRUD operations

After the definition of the syntax, the method describes which CRUD operations are supported and how they can be performed on the target system.

Create The creation of a DID is made by means of an `NYM` transaction, which is a Sovrin specific message for the registration of a DID on the registry. The documentation [6] describes all the transactions precisely. The Sovrin specification adds a new entry under `publicKey` called `authorizations`. It is added to a public key's properties to establish what the key is allowed to do. The transaction should look as follows:

```
{
  'operation': {
      'type': <Transaction type -- NYM >,
      'dest': <new DID that is being registered>,
      'role': <Role given to the new DID -- based on network config>,
      'verkey': <Key material encoded>,
  },

  'identifier': 'L5AD5g65TDQr1PPHHRoiGf' <Trust Anchor DID>,
  'reqId': <A nonce for this transaction>,
  'protocolVersion': 2,
  'signature': <signature over this transaction from the Trust Anchor>
}
```

The transaction must specify a TrustAnchor, which is an entity allowed to write on the registry, its signature, and a new unregistered DID. The possible results for this transaction are:

- NACK - invalid transaction
- REJECT - invalid authorization rules
- SUCCESS - returns the transaction identifier and its merkle proof

By convention, the DID controller usually creates an `ATTRIB` transaction with a raw value containing the service endpoint. An `ATTRIB` object is a set of properties related to an `NYM`. The resulting transaction should contain something like:

```
    "raw": "{\"endpoint\":{\"endpoint\":\"https://example.com\"}"
```

An extended version of the endpoint could be included with more specific fields, as follows:

```
{
  "endpoint": "https://example.com",
  "types": [ "endpoint", "did-communication", "DIDComm" ],
  "routingKeys": [ "did:key12345", "did:key12345" ]
}
```

The result on the transaction looks like this:

```
"raw": "{\"endpoint\":{\"endpoint\":\"https://example.com\",\"types\":
        [\"endpoint\",\"did-communication\",\"DIDComm\" ],
        \"routingKeys\":[\"did:key12345\",\"did:key12345\"]}}"
```

The endpoint will be used to create a service entry in the DID document.

Read (resolve) A Sovrin DID can be resolved by sending a `GET_NYM` transaction. The response given by the ledger can be trusted either because different nodes return the same value or because along with the response the merkle proof is present. The query should look like this:

```
{
    "submitterId": <Optional; DID of the author of this query>,
    "reqId": <Optional; a nonce for this query>,
    "operation": {
        "did": <DID to be queried>,
        "type": "GET_NYM"
    }
}
```

The response will include most of the data necessary to build the DID document. However, to check whether there are services related to the document or not, the `ATTRIB` object must be retrieved from the ledger as well. The `GET_ATTRIB` object should look like this:

```
{
    "submitterId": <Optional; DID of the author of this query>,
    "reqId": <Optional; a nonce for this query>,
    "identifier": <Required; The DID being read/resolved>,
    "operation": {
```

```
      "type": "GET_ATTRIB",
      "raw": "endpoint" <Required; the value must be endpoint>
    }
}
```

If the NYM has no ATTRIB object related, the result could be a "Not found" error. While if the object is present, the response will be something like:

```
{
  ...
  'data': '{"endpoint":{"endpoint":"https://example.com/endpoint"}}',
  'dest': 'AH4RRiPR78DUrCWatnCW2w' < The DID being read/resolved>,
  'raw': 'endpoint',
  ...
}
```

If the endpoint is reported in a more specific way, it must be interpreted using the following method:

- If the types field is present it's entries must be "endpoint", "did-communication", and/or "DIDComm", generating entries in the DID document service block.
- If the endpoint is defined and the types item is not present, empty or contains any other values, default the types to "types": ["endpoint", "did-communication"].
- If the routingKeys item is present, use its value as the array for routingKeys in the did-communication and DIDComm type service blocks.
- If the routingKeys entry is not present, its value is set to an empty array (e.g. "routingKeys": [ ])

Once obtained both the GET_NYM and the eventual GET_ATTRIB responses, it is possible to proceed with the creation of the DID document JSON-LD representation using this schema:

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/suites/ed25519-2018/v1",
    "https://w3id.org/security/suites/x25519-2019/v1"
  ],
  "id": "did:sov:HR6vs6GEZ8rHaVgjg2WodM",
  "verificationMethod": [
    {
      "type": "Ed25519VerificationKey2018",
      "id": "did:sov:HR6vs6GEZ8rHaVgjg2WodM#key-1",
```

```
      "publicKeyBase58": "9wvq2i4xUa5umXoThe83CDgx1e5bsjZKJL4DEWvTP9qe"
    },
    {
      "type": "X25519KeyAgreementKey2019",
      "id": "did:sov:HR6vs6GEZ8rHaVgjg2WodM#key-agreement-1",
      "publicKeyBase58": "3mHtKcQFEzqeUcnce5BAuzAgLEbqKaV542pUf9xQ5Pf8"
    }
  ],
  "authentication": [
    "did:sov:HR6vs6GEZ8rHaVgjg2WodM#key-1"
  ],
  "assertionMethod": [
    "did:sov:HR6vs6GEZ8rHaVgjg2WodM#key-1"
  ],
  "keyAgreement": [
    "did:sov:HR6vs6GEZ8rHaVgjg2WodM#key-agreement-1"
  ],
  "service": [
    {
      "type": "endpoint",
      "serviceEndpoint": "https://example.com/endpoint"
    },
    {
      "id": "did:sov:HR6vs6GEZ8rHaVgjg2WodM#did-communication",
      "type": "did-communication",
      "priority": 0,
      "recipientKeys": [
        "did:sov:HR6vs6GEZ8rHaVgjg2WodM#key-agreement-1"
      ],
      "routingKeys": [],
      "accept": [
        "didcomm/aip2;env=rfc19"
      ],
      "serviceEndpoint": "https://example.com/endpoint"
    }
  ]
}
```

If the `NYM` has an `ATTRIB` related, but it does not have any `endpoint` then the service property will be empty. If the `NYM` has an `ATTRIB` and it contains an `endpoint` with the `types` property which contains a `DIDComm` entry, then the service will have the following shape:

```
{
```

```
  "id": "did:sov:HR6vs6GEZ8rHaVgjg2WodM#didcomm-1",
  "type": "DIDComm",
  "serviceEndpoint": "https://example.com/endpoint",
  "accept": [
    "didcomm/v2"
  ],
  "routingKeys": []
}
```

and the following entry must be added to the `@context` array:

```
    "https://didcomm.org/messaging/contexts/v2"
```

Update (Replace) To replace the DID document, the owner of the DID has to send the following transaction using the key referenced in the authentication property.

```
{
  "submitterId": <DID of the author of this query>,
  "signature": [<signature over this transaction from the author>,]
  "reqId": <a nonce for this query>,
  "operation": {
      "type": "NYM",
      "did": <DID whose record needs to be updated>,
      "document": {
          "publicKey": [{
              "id": <a valid unique identifier>
              "type": <ED25519>
              "publicKeyBase58": <Key material encoded in format>,
              "authorizations": ["all"]
          },{
              "id": <a valid unique identifier>
              "type": <ED25519>
              "publicKeyBase58": <Key material encoded in format>
          },{
              "id": <a valid unique identifier>
              "type": <ED25519>
              "publicKeyBase58": <Key material encoded in format>
          }],
          "authentication": [{
              "type": "ED25519SigningAuthenticationThreshold",
              "threshold": <an integer>,
              "publicKey": [<key references>]
          }],
          "service": [{
```

```
            "type": <agentService, emailService, apiService, dnsService,
                     authenticationService, etc>,
            "serviceEndpoint": <A URI for the endpoint>
          }]
      }
    }
}
```

Delete (Revoke) To revoke the DID, the owner has to send the following transaction:

```
{
    "submitterId": <DID of the owner>,
    "signature": <signature over this transaction from the
                  DID in identifier>,
    "reqId": <a nonce for this transaction>,
    "operation": {
        "type": "NYM",
        "did": <DID whose key is being revoked>,
        "document": null
    }
}
```

This action is irreversible since it sets the verification key to null, therefore the identity no longer operates on the network since it has no key to authenticate.

Security and privacy Communications to the Sovrin ledger use CurveZMQ [4], an authentication and encryption protocol designed for ZeroMQ which uses secure elliptic-curve cryptography. ZeroMQ is an open-source messaging library. This system mitigates the following attacks: eavesdropping, replay, message insertion, deletion, modification, impersonation, and man-in-the-middle. All the security issues on CurveZMQ will apply to this DID method.
Sovrin currently stores only the following data on the ledger:

- Public DIDs and DID documents which include: public keys and service endpoints
- Credential Schemas
- Credential Definitions - the use of a specific Schema, by a specific DID for the purpose of issuing credentials and a reference to a Revocation Registry
- Revocation Registries associated with Credential Definitions
- Policy Registries - used for authorizing agents and a reference to a Policy Accumulator.
- Policy Accumulators - proving to authorize agents in zero knowledge.

- All confidential information such as cryptographic private keys are stored with end consumers of Sovrin.

1.1.2 `did:key`

This DID method represents a well-known example of ledger-agnostic DID. In fact, it does not require any communication with a VDR, since all the necessary details can be retrieved from the DID itself. The method exploits cryptographic keys to generate DIDs and the relative documents. Because of their structure, these DIDs are not suggested for long usage.

Method syntax

For this method, every DID must be prefixed with `did:key` and completed with a base58 encoded value that is a concatenation of the Multicodec [10] identifier for the public key type and the public key's raw bytes. The ABNF format is the following:

```
did-key-format := did:key:<mb-value>
mb-value       := z[a-km-zA-HJ-NP-Z1-9]+
```

It can also be seen as a chain of functions over the key raws data:

```
did-key-format := did:key:MULTIBASE(base58-btc,
MULTICODEC(public-key-type, raw-public-key-bytes))
```

An example of ed25519 `did:key` is:

```
did:key:z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK
```

CRUD operations

Given the nature of this method, no updating or revoking functionalities are available for DIDs.

Create The creation of a `did:key` begins with the creation of a cryptographic key pair and encoding the public key as reported above. The DID document is therefore expanded from the public key. For instance, the previous DID example would be expanded to:

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/suites/ed25519-2020/v1",
    "https://w3id.org/security/suites/x25519-2020/v1"
  ],
  "id": "did:key:z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK",
  "verificationMethod": [{
    "id": "did:key:z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK
```

```
          #z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK",
    "type": "Ed25519VerificationKey2020",
    "controller": "did:key:z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK",
    "publicKeyMultibase": "z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK"
  }],
  "authentication": [
    "did:key:z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK
     #z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK"
  ],
  "assertionMethod": [
    "did:key:z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK
     #z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK"
  ],
  "capabilityDelegation": [
    "did:key:z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK
    #z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK"
  ],
  "capabilityInvocation": [
    "did:key:z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK
     #z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK"
  ],
  "keyAgreement": [{
    "id": "did:key:z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK
          #z6LSj72tK8brWgZja8NLRwPigth2T9QRiG1uH9oKZuKjdh9p",
    "type": "X25519KeyAgreementKey2020",
    "controller": "did:key:z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK",
    "publicKeyMultibase": "z6LSj72tK8brWgZja8NLRwPigth2T9QRiG1uH9oKZuKjdh9p"
  }]
}
```

Read (resolve) Resolving a `did:key` means expanding the encoded public key to a document. This can involve different phases, where attention has to be put not to forget some possible types of cryptographic key or algorithm since the public key used to create the DID can belong to many different standards (*e.g.*, ed25519, Secp256k1, RSA and so on). The procedure is detailed in a series of algorithms:

- Document creation algorithm
- Signature method creation algorithm
- Decode public key algorithm
- Encryption method creation algorithm
- Derive encryption key algorithm

- Context creation algorithm

The document creation algorithm describes the procedure for the creation of a DID document. The inputs are the DID and some options that are specified throughout the algorithms. The phases are:

1. Initialize `document` to an empty object.
2. Split the DID into its components using colon as delimeter, and check their validity.
3. Obtain the signature method running the signature method creation algorithm over the encoded component of the DID.
4. Set the document `id` properties equals to the DID. if the DID is not valid, an `invalidDid` error must be raised.
5. Initialize the `verificationMethod` property in the document to an array where the first value is the verification method previously obtained.
6. Initialize the `authentication`, `assertionMethod`, `capabilityInvocation`, and the `capabilityDelegation` properties in document to an array where the first item is the value of the `id` property in the verification method obtained.
7. If the options enable the encryption key derivation (is set to true):
   (a) Run the encryption method creation algorithm and add the returned value to the `verificationMethod` array in the document.
   (b) Add this verification method `id` property to the `keyAgreement` array in the document.
8. Run the Context creation algorithm to define which values are needed in the `@context` array
9. Return `document`.

The second algorithm is the signature method creation algorithm. It can be used for decoding a multibase-encoded multicodec value into a verification method that is suitable for verifying digital signatures. The required inputs are: the DID, a `multibaseValue` , and the options (same as above). the steps are:

1. Initialize `verificationMethod` to an empty object.
2. Set `multicodecValue` and `rawPublicKeyBytes` to the result of the decode public key algorithm over the multibase value.
3. Following the table in Figure 1.4, determine if `rawPublicKeyBytes` has a proper key length.

   If the length doesn't match the expected value, an `invalidPublicKeyLength` error must be raised.

Multicodec hexadecimal value	public key byte length	Description
0xe7	33 bytes	secp256k1-pub - Secp256k1 public key (compressed)
0xec	32 bytes	x25519-pub - Curve25519 public key
0xed	32 bytes	ed25519-pub - Ed25519 public key
0x1200	33 bytes	p256-pub - P-256 public key (compressed)
0x1201	49 bytes	p384-pub - P-384 public key (compressed)
0x1202	?? bytes	p521-pub - P-521 public key (compressed)
0x1205	?? bytes	rsa-pub - RSA public key. DER-encoded ASN.1 type RSAPublicKey according to IETF RFC 8017 (PKCS #1)

Figure 1.4: multicodec table

4. check whether `rawPublicKeyBytes` represents a valid key for the type specified in `multicodecValue`. If not, an `invalidPublicKey` error must be raised.
5. Set the `verificationMethod`'s `id` property to the concatenation of the DID, a '#' character and the `multicodecValue`.
6. Set `publicKeyFormat` to the public key format contained in the options. If it is unknown to the implementation, a `unsupportedPublicKeyType` must be raised
7. If the options for the experimental public key types is set to false, and `publicKeyFormat` is not `Multikey`, `JsonWebKey2020` or `Ed25519VerificationKey2020` an `invalidPublicKeyType` must be raised.
8. Set the `verificationMethod`'s `type` to the `publicKeyFormat` value and the `controller` value to the DID .
9. If `publicKeyFormat` is `Multikey` or `Ed25519VerificationKey2020`, set the verification method `publicKeyMultibase` value to `multibaseValue`.
   If `publicKeyFormat` is `JsonWebKey2020`, set the verification method `publicKeyMultibase` value to the result of the encode JWK algorithm. The latter can be used to encode a multicodec value to a JSON Web Key.
10. Return `verificationMethod`

The decode public key algorithm can be used for expanding a multibase-encoded multicodec value to decoded public key components. The inputs are `multibaseValue` and the options. The algorithm is the following:

1. Set `multicodecValue` and `rawPublicKeyBytes` to empty objects.
2. Set `multicodecValue` to the `multibaseValue` decoded result's header.
   Set `rawPublicKeyBytes` to the remaining decoded bytes.
3. Return `multicodecValue` and `rawPublicKeyBytes`.

The encryption method creation algorithm is similar to the signature method creation algorithm with some exceptions. The required inputs are: the DID, a `multibaseValue` and options. The steps are:

1. Initialize `verificationMethod` to an empty object.
2. Set `multicodecValue` and `rawPublicKeyBytes` to the result of the derivation key algorithm.
3. Ensure the proper key length of `rawPublicKeyBytes` based on the multicodecValue table provided in Figure 1.5.

Multicodec hexadecimal value	public key byte length	Description
0xec	32 bytes	x25519-pub - Curve25519 public key

Figure 1.5: x25519 key length

   If the byte length of `rawPublicKeyBytes` does not match the expected public key length for the associated `multicodecValue`, an `invalidPublicKeyLength` error must be raised.
4. Create the `multibaseValue` by concatenating the letter 'z' and the base58 encoding of the concatenation of the `multicodecValue` and the `rawPublicKeyBytes`.
5. Set the `verificationMethod`'s `id` to the concatenation of the DID, the character '#' and the `multibaseValue`
6. Set `publicKeyFormat` to the public key format contained in the options. If it is unknown to the implementation, a `unsupportedPublicKeyType` must be raised
7. If the options for the experimental public key types is set to false, and `publicKeyFormat` is not `Multikey`, `JsonWebKey2020` or `X25519KeyAgreementKey2020` an `invalidPublicKeyType` must be raised.
8. Set the `verificationMethod`'s `type` to the `publicKeyFormat` value and the controller value to the DID .
9. If `publicKeyFormat` is `Multikey` or `Ed25519VerificationKey2020`, set the verification method `publicKeyMultibase` value to `multibaseValue`. If `publicKeyFormat` is `JsonWebKey2020`, set the verification method `publicKeyJwk` value to the result of the encode JWK algorithm. The latter can be used to encode a multicodec value to a JSON Web Key.
10. Return `verificationMethod`.

The derive encryption key algorithm can be used to transform a multibase-encoded multicodec value to public encryption key components that are suitable for encrypting messages to a receiver. The required inputs are `multibaseValue` and the options. The algoritm is structured as follows:

1. Set `publicEncryptionKey` to an empty object.
2. Decode `multibaseValue` using the base58-btc multibase alphabet and set `multicodecValue` to the multicodec header for the decoded value. Set the rawPublicKeyBytes to the remaining bytes.
3. If the `multicodecValue` is `0xed`, derive a public X25519 encryption key by using the `rawPublicKeyBytes` and set `generatedPublicEncryptionKeyBytes`
4. Set `multicodecValue` and `rawPublicKeyBytes` in `publicEncryptionKey` respectively to `0xec` and `generatedPublicEncryptionKeyBytes`
5. Return `publicEncryptionKey`.

The context creation algorithm is used to obtain the `@context` property for the representation. The context has a default value of `https://www.w3.org/ns/did/v1` that can be overridden by this algorithm. The necessary inputs are a `document` and the options. The functioning is simple: for each verification method, check the `type` property and add the respective URL to the `@context` array as reported on Figure 1.6. Once finished, return the array.

type value	Context URL
Ed25519VerificationKey2020	https://w3id.org/security/suites/ed25519-2020/v1
X25519KeyAgreementKey2020	https://w3id.org/security/suites/x25519-2020/v1
JsonWebKey2020	https://w3id.org/security/suites/jws-2020/v1

Figure 1.6: list of possible context URLs

Security and privacy considerations There are different aspects that should be considered while implementing or using this method. They derive from the method's intrinsic structure, therefore they cannot be overcome. The issues are:

1. Key rotation is not supported
2. Deactivation is not supported
3. Key derivation lacks proof
4. Long-term usage is discouraged

Other privacy or security threats are related to the specific method implementation that is being developed or used.

Chapter 2

Method specification - `did:monokee`

Every DID method needs a specification describing the DID and its characteristics. It has to conform with the DID core specifications [9] (as well as the implementation). The first questions that should be answered before implementing a DID method are:

- Why is a custom method needed?
- Is there a method that already satisfies the application's requirements?
- If not, what kind of DID is needed? public or private?
- What VDR should be used?

Given that this is an early implementation, every VDR interaction is replaced with API calls that are supposed to work as the respective operations on a VDR (for example: `create_did` means writing the DID document on the VDR) even though the final implementation might not use a VDR. These calls could be also replaced by methods to manage a self-contained DID (such as `did:key`). Details about an eventual VDR will be applied as soon as they are available. For this reason, some assumptions are made throughout the specification and some disclaimers are reported on blue boxes, while yellow boxes contain the initial definitions that have been further developed to reach the final result.

The APIs have been designed to interact with a MongoDB database, a simple NoSQL database commonly used for webservices. The APIs are meant to run in the localhost at the 8080 port. Both the API and the typescript reference implementation are available on GitHub [18] under MIT license.

2.1 Motivation

Monokee is a web-centric and cloud-based IAM. The product category refers to IDAAS (IDentity as a service).
The identity orchestration component has been extended to support the SSI in order to simplify the transaction to a decentralized model.

Therefore, the main purpose of this DID method is to group multiple representations of the same issuer within the various ledgers to facilitate the orchestrating of VCs related to different contexts. References to external DIDs are specified via the `service` property. The corresponding document is retrieved through the dereferencing process at 2.3.3.

2.2 DID schema

To begin developing a DID method, the structure of its schema must be defined according to DID syntax [9]. For this method, the syntax is the following:

```
did = "did:monokee:"method-specific-identifier
```

2.2.1 Method-specific identifier

For `did:monokee`, the `method-specific-identifier` is defined by the following ABNF:

```
method-specific-identifier = time-low "-" time-mid "-"
                             time-high-and-version "-"
                             clock-seq-and-reserved
                             clock-seq-low "-" node
time-low                   = 4hexOctet
time-mid                   = 2hexOctet
time-high-and-version      = 2hexOctet
clock-seq-and-reserved     = hexOctet
clock-seq-low              = hexOctet
node                       = 6hexOctet
hexOctet                   = hexDigit hexDigit
hexDigit = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9" /
           "a" / "b" / "c" / "d" / "e" / "f"
```

The `idstring` conforms to Universally unique identifier (UUID) v4[7].

> This specification could change depending on the VDR implementation decision since, as mentioned, the method-specific identifier could exploit the intrinsic uniqueness of a blockchain address.

2.3 CRUD Operations

Once the definition of the DID schema has been done, the specification must describe which operations are allowed over a DID method, how they have been realized and how they interact with the VDR. Not all the CRUD operations have to be realized, some DID methods do not make use of updating and deactivating capabilities, it all depends on the requirements and design choices.
The implementation has to be done accordingly to the DID method requirements [9].

For the CRUD implementation, there are two different possibilities:

1. API calls work on some kind of VDR (ledger or database. It doesn't matter whether it is decentralized or not)
2. The method is self-contained, no update nor delete operations are allowed.

This documentation refers to the first version since it is the one that has been realized.

2.3.1 Create (register)

Initial definition:
The creation function, which corresponds to the monokee `create_did` webservice, can only be performed by the owner of a monokee tenant that is identified according to an authentication flow defined for that specific domain.

In order to proceed with the creation of a DID, firstly the client has to generate the method-specific identifier and an ed25519/X25519 key pair.
The creation of the DID is hence immediate: concatenate the "`did:monokee:`" string with the previously created method-specific identifier.

As example: `did:monokee:bdc78193-5a5c-4584-83cd-a08f523ebdb7`

The DID document can be created following this schema using a JSON-LD representation:

```
{
  "@context": [
    "https://w3id.org/did/v1",
    "https://w3id.org/security/suites/ed25519-2018/v1",
    "https://w3id.org/security/suites/x25519-2019/v1"
  ]
  "id": "did:monokee:bdc78193-5a5c-4584-83cd-a08f523ebdb7",
  "verificationMethod": [
```

```
    {
      "type": "Ed25519VerificationKey2018",
      "id": "did:monokee:bdc78193-5a5c-4584-83cd-a08f523ebdb7#key-1",
      "controller": "did:monokee:bdc78193-5a5c-4584-83cd-a08f523ebdb7",
      "publicKeyBase58": "<encoded_public_key>"
    },
    {
      "type": "X25519KeyAgreementKey2019",
      "id": "did:monokee:bdc78193-5a5c-4584-83cd-a08f523ebdb7#key-agreement-1",
      "controller": "did:monokee:bdc78193-5a5c-4584-83cd-a08f523ebdb7",
      "publicKeyBase58": "<encoded_key>"
    }
  ],
  "authentication": [
    "did:monokee:bdc78193-5a5c-4584-83cd-a08f523ebdb7#key-1"
  ],
  "assertionMethod": [
    did:monokee:bdc78193-5a5c-4584-83cd-a08f523ebdb7#key-1
  ],
  "keyAgreement": [
    "did:monokee:bdc78193-5a5c-4584-83cd-a08f523ebdb7#key-agreement-1"
  ],
  "service": [
    {
      "id": "did:monokee:bdc78193-5a5c-4584-83cd-a08f523ebdb7#sov-did-1",
      "type": "SovrinIssuer",
      "serviceEndpoint": "did:sov:WRfXPg8danfKVubE3eX8pw"
    }
  ]
}
```

To prove the feasibility of the project, the `did:sov` service has been implemented as a technology demonstrator and, as such, it is always included in every `did:monokee`. Further study is required to refine the method to dynamically include other, potentially multiple, and cross-chain, services.

After the creation of the document, a signature has to be established utilizing the document and the previously created ed25519 private key. Encode the key using the base58 algorithm and insert it into a request body with the document as follows

```
{
  didDocument:<document>,
  signature:<base58_encoded_signature>
}
```

The body can then be sent through an HTTP POST request to the API endpoint, which should look as follows:

`http://localhost:8080/createdid`

2.3.2 Read (resolve)

Questions to be answered before the implementation:

- Is it better to implement a resolver or leverage the Universal Resolver [16] Or the typescript resolver implementation [15]?
- Which representation should be used for `resolveRepresentation()`? JSON, JSON-LD or JSON-CBOR?
- Which binding should be used (for the reference implementation)?

> Initial definition:
> DID resolution is done through the `resolve_did()` API call by adding the DID on the request URL.

For this method, an implementation of the resolution process has been developed from scratch. The resolution of a DID is done with an API GET request including the DID in the URL as follows:

`http://localhost:8080/resolvedid/<did>`

The response contains the DID document in JSON-LD format (see the example above) and all the metadata necessary in the response's body. A possible response body:

```
{
  "@context": "https://w3id.org/did-resolution/v1",
  "didDocument": [...],
  "didResolutionMetadata": {
    "contentType": "application/did+ld+json"
  },
  "didDocumentMetadata": {
    "created": "2019-03-23T06:35:22Z",
    "updated": "2023-08-10T13:40:06Z"
  }
}
```

The `didResolutionMetadata` and `didDocumentMetadata` conform to the DID core documentation [9] and the possible values that they can contain are reported on the DID specification registries [14]. The information contained in this metadata is technical and related only to the DID document, therefore it can be used only within the resolution

process.

The response can be used with no further modifications in a DID resolver reference implementation since the API response is already compliant with the resolver response requirements [14]

2.3.3 Dereference

Initial definition:
Via the function `dereference_did(URL)` and giving the id of the specific service as input, the monokee tenant will perform a resolution of the remote DID and return the relative DID document.

The DID URLs can be created by concatenating a fragment to a DID. These fragments can be used to retrieve:

- a verification method
- a service
- a DID document related to a `did:sov`

The dereferentiation of the URL starts with the DID resolution. Once the document has been obtained, simply search the fragment among the verification method and service `ids`. If it corresponds to a verification method, simply return the entire method. If it corresponds to service, check if it is a `did:sov` or not. If not, return the entire service, if yes resolve the DID and return the document.
For the resolution of `did:sov` services, the universal resolver driver [16] is leveraged. It can be run as a container whose port 8080 must be bounded with the client's port number 8888. Simply, after cloning the repository [16] and building the image, run:
`docker run -p 8888:8080 universalresolver/driver-did-sov`.

2.3.4 Update (replace)

Questions to be answered before the implementation:

- Are updating capabilities needed?
- Is a delegation process (adding/removing a controller) necessary?
- How to perform authorization?

Initial definition:
The update function, which corresponds to the monokee `update_did_document` web-service, can only be performed by the owner of a monokee tenant or optional controllers that are identified according to an authentication flow defined for that specific domain and that involve the private keys anchored to the DIDs that effectively control the DID document.

The updating capabilities for this method are as follows:

1. Replace the `#key-1` method's public key.
   The request body must contain the id of the method that is being updated and the new public key. If the `key-1` method key is changed, it will no longer be possible to use it to derive the x25519 `key-2` method key, and they will be "decoupled".

2. Add an ed25519 verification method.
   The enumeration of the method has to be compliant with the `key-<methodNumber>` notation, where `methodNumber` is the index of the last method inserted+1 starting from 1.

For each of these options, the utilization is described below:

1. To replace the public key in the `#key-1` verification method, the following request body has to be used:

   ```
   {
     methodUrl:<methodUrl>,
     newKey:<newKey>,
     signature:<encodedSignature>
   }
   ```

   where the `methodUrl` property is the DID URL of the method whose public key is being updated. The `newKey` field is the new base58 encoded public key. The `encodedSignature` in the request body is the base58-encoded signature created using the private key whose public key is being replaced and a message with the following structure (converted to string):

   ```
   {
     methodUrl:<methodUrl>,
     newKey:<newKey>
   }
   ```

   This signature will be checked to authorize the modification. The request body has to be sent through an HTTP PUT request to the following endpoint:

   ```
   http://localhost:8080/updatekey
   ```

2. To add a verification method, the following request body has to be used:

```
{
  "did":<did>,
  "verificationMethod":<verificationMethod>,
  "signature":<encodedSignature>
}
```

where the `did` property is the DID that is being updated. The `verificationMethod` must have the following structure:

```
{
  "id": <did>+"#key-"+<keyNumber>,
  "type": "Ed25519VerificationKey2018",
  "controller":<did>,
  "publicKeyBase58": <encodedPublicKey>
}
```

where `keyNumber` is the number of the next method and `encodedPublicKey` is the new ed25519 public key encoded using base58.
The `encodedSignature` in the request body is the base58-encoded signature created using the private key in the `#key-1` verification method and a message with the following structure (converted to string):

```
{
  "did":<did>,
  "verificationMethod":<verificationMethod>
}
```

This signature will be checked to authorize the modification.

The request body has to be sent through an HTTP PUT request to the following endpoint:

```
http://localhost:8080/addverificationmethod
```

2.3.5 Delete (revoke)

Questions to be answered before the implementation:

- Is revoking capability needed? How should it be implemented?
- DIDs are meant to be long-lasting (weeks or months) or not?

Initial definition:
The delete function, which corresponds to the monokee **delete_did** webservice, can only be performed by the owner of a monokee tenant (not other controllers) that is identified according to an authentication flow defined for that specific domain and that involves the private key anchored to the DID that is effectively the owner of the DID document.

To deactivate a DID, an HTTP DELETE request has to be sent to the following endpoint:

```
http://localhost:8080/deactivate/<did>
```

where `did` represents the DID that is being deactivated. The request body must contain a base58 encoded signature created using the private key related to the public key in the `#key-1` verification method and the message "deactivate" in lower-case. The body structure is the following:

```
{
    signature:<encodedSignature>
}
```

This operation sets the `deactivated` property in the metadata related to the DID to `true`.

Chapter 3

Results and discussion

Up to now, the `did:monokee` method is a technology demonstrator proving the feasibility of cross-chain DID method linking, a building block necessary to enable full interoperability within different competing standards in the DID ecosystem. All the CRUD functions respect the requirement given by the specification and work properly. All the core functionalities have been implemented, however, as previously mentioned, future works are required to refine the results demonstrated in this thesis. The APIs generalization permits a variety of possible future specializations, such as ledger-agnostic DIDs or direct peer-to-peer VDR interactions. Ideally, the final result should make use of these direct interfaces with a VDR (or potentially more than one) rather than a centralized database solution. A potential way to achieve so is to integrate Walt.id SSI kit [17] which groups different components to create an infrastructure that delivers SSI services. The objective is to use one of its components as a bridge between the API and different VDRs.

This discussion highlights how diverse design choices and VDRs characterize DID methods and make them unique. Depending on the guidelines and the use context, DIDs can be developed for specific environments. The documentation leaves some questions open, such as how to verify the binding between the physical entities and the DID, and whether the physical identity needs to be aware of being identified by a DID or not. These aspects are to be considered in the specific application field since different needs require different management of these elements. Generally, the subject does not need to be aware of being referenced by a DID, and it should not be possible to determine the DID subject from a DID document given that this would lead to privacy and security problems unless the DID subject decides to publicly state this relationship. Moreover, services in the DID documents should not report any personal information about the subject to avoid such troubles.

Besides that, as mentioned, no requirements have been defined for the VDR, hence developers adopted a wide variety of different solutions, creating many DID methods with different decentralization levels. As reported in the specification registry [14], where

more than 130 methods are registered, many different standards can refer to the same VDR yet using different ways to operate. Furthermore, many DIDs are ledger agnostic (*e.g.*, `did:key` and `did:pkh`), or leverage the DNS system (*e.g.*, `did:web`). There can be also ephemeral DIDs, such as `did:peer`, which can be used only once. These examples point out the heterogeneity among DIDs and DID methods. This freedom could negatively impact the decentralization objective of DIDs. As reported on the DIF blog [5], it is necessary to find a balance: the focus has to be kept on independence from central authorities but remembering the interest to apply this technology in several contexts including non-decentralized environments. For this purpose, DIF helps users and developers evaluate DID methods to guarantee a conscious choice.

Another critical aspect to discuss is DID controllers. As mentioned, it is essential to express how authentication is performed to authorize operations on a DID. This is typically done by leveraging controllers indeed. Attention should be paid when modifying this property since they should not be removed or added lightly. The loss of control over a DID and the respective DID document must be avoided, hence the method has to be designed accordingly. Moreover, it must be considered that there can be multiple controllers. There are mainly two ways to manage such a situation as reported on Figure 3.1:

1. Each controller can alter the DID document autonomously
2. Controllers refer to a group controller that applies the modifications only after a threshold of signatures is reached

The developers must understand which one suits their method the best, and then they have to implement it.

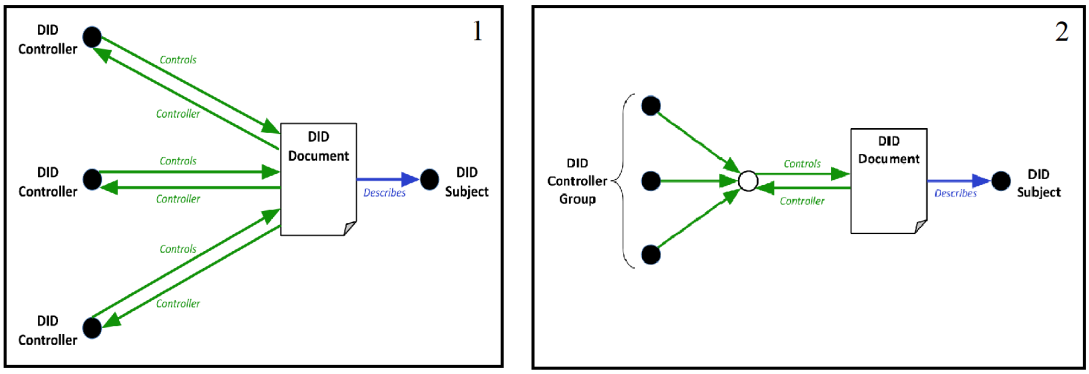

Figure 3.1: The two methods to deal with multiple controllers

Beyond DIDs, there are other forms of identifiers which have similar ambitions and that are being considered in the SSI field: Self-Certifying IDentifier (SCID) and Autonomous IDentifier (AID). SCIDs are deterministically derived from a public key. They are similar to ledger agnostic DIDs, where the identifier ownership can be proved using the

related private key. Some SCIDs could be DIDs and some DIDs could be SCIDs. AID are similar to SCID, but they must be utilizable with no third-party interaction. These solutions, however, are representing different approaches and are to be considered out of the scope of this thesis.

Conclusion

From my point of view, DIDs represent a new standard whose possibilities are still being explored. Though decentralization might not be achieved by all DID methods, it will likely become basic as Web3.0 spreads around. However, users will not deal directly with DIDs, but rather with Verifiable Credentials and digital wallets, with DIDs working behind the scenes to ensure a non-centralized digital identity.

The possibilities that can be explored by the community and the consequent variety of different DIDs and DID methods might be discouraging, leading to a progressive consolidation of multiple competing standards with derived interoperability challenges. Nonetheless, public registries exist where information can be found, and associations are putting a lot of effort into making the DID field easy to be understood and used.

To conclude, DIDs and VCs represent a great leap towards decentralization and are one of the most important milestones for Web3.0. They are subjected to an enormous amount of work, and both of them are still in their early stage, therefore, small choices taken today might change completely the future of the internet and of the people who use it. In any case, the freedom left to developers to create and adapt the specifications [9] to different environments and to use different technologies open the SSI paradigm to a wide number of implementations with custom services that satisfy different needs.

Bibliography

[1] *Binance DID Method Specification*. URL: https://github.com/ontology-tech/DID-method-specs/blob/master/did-bnb/DID-Method-bnb.md.

[2] Juan Caballero. *Decentralized Identity FAQ*. Jan. 5, 2021. URL: https://identity.foundation/faq/.

[3] Dave Crocker and Paul Overell. *Augmented BNF for Syntax Specifications: ABNF*. Jan. 2008. URL: https://www.rfc-editor.org/rfc/rfc5234.

[4] *CurveZMQ - Read The Docs*. URL: http://curvezmq.wikidot.com/page:read-the-docs.

[5] Decentralized Identity Foundation. *The DID Dilemma: When is an identifier 'decentralized'?* URL: https://blog.identity.foundation/the-did-dilemma--when-is-an-identifier--decentralized--/.

[6] *Hyperledger Indy Transactions*. URL: https://hyperledger-indy.readthedocs.io/projects/node/en/latest/transactions.html.

[7] P. Leach et al. *A Universally Unique IDentifier (UUID) URN Namespace*. June 2005. URL: https://www.ietf.org/rfc/rfc4122.txt.

[8] Dave Longley, Dmitri Zagidulin, and Manu Sporny. *The did:key Method v0.7*. July 29, 2022. URL: https://w3c-ccg.github.io/did-method-key/.

[9] Dave Longley et al. *Decentralized Identifiers (DIDs) v1.0*. July 19, 2022. URL: https://www.w3.org/TR/did-core/.

[10] *Multicodec*. URL: https://github.com/multiformats/multicodec/blob/master/README.md.

[11] *Sovrin foundation*. URL: https://sovrin.org/.

[12] Orie Steele and Michael Prorock. *DID Implementation Guide v1.0*. Apr. 11, 2022. URL: https://www.w3.org/TR/did-spec-registries/.

[13] Orie Steele and Michael Prorock. *Merkle Disclosure Proof 2021*. URL: https://w3c-ccg.github.io/Merkle-Disclosure-2021/#merkle-proofs.

[14] Orie Steele, Manu Sporny, and Michael Prorock. *DID Specification Registries*. Sept. 8, 2022. URL: https://www.w3.org/TR/did-spec-registries/.

[15] *Typescript DID Resolver*. URL: https://github.com/decentralized-identity/did-resolver.

[16] *Universal Resolver - Driver Development*. URL: https://github.com/decentralized-identity/universal-resolver/blob/main/docs/driver-development.md.

[17] *Walt.id SSI Kit*. URL: https://walt.id/ssi-kit.

[18] Marco Xausa and Mattia Zago. URL: https://github.com/monokee-dev/stage-unitn-xausa.