Centrifuge: Protocol for private by design transactions on a decentralized network

Lucas Vogelsang[*] Manuel Polzhofer[†] Philip Stehlik[‡] Vimukthi Wickramasinghe[§]

edb90b5 - 2019-05-17

Abstract

Ethereum has demonstrated the immense potential offered by trustless computing. However, organizational adoption of general purpose public blockchains such as Ethereum for transaction settlement has been hindered by issues of scalability and privacy. Centrifuge implements a hybrid on-chain/off-chain private by design protocol, with a focus on moving the financial supply chain towards a trustless and cryptographically secured protocol. We enable nodes to exchange documents privately in a p2p network and support on-chain tokenization and verification of documents.

1 Introduction

1.1 Prior work

Carbon dating, as described by Clark[2] was using bitcoin to commit to a message with a provable timestamp by embedding a commitment in a bitcoin block. Todd[8] introduced the first publicly available timestamp service with Open Timestamps in 2016. It enables committing to a state of a document by adding the hash of it to a Merkle tree that is then written into a Bitcoin transaction. By producing a proof of inclusion in the Merkle tree and a proof that the root hash is included in a given block in the Bitcoin blockchain any third party can verify that the document had the given state at that point in time. Centrifuge uses smart contracts on the Ethereum blockchain to timestamp the documents state in a very similar way. Coleman's work on state channels[3], the generalized version of a payment channel first introduced by Poon et al.[5] explores many of the challenges associated with settling peer to peer transactions without third party validators in form of a public blockchain. Channels must first be opened on-chain, but can then be modified very cheaply by sending signed messages back and forth. Expensive on-chain transactions are only used in case of disagreement, or to close a channel. Many of the attacks on state channel protocols were considerations in the design of the Centrifuge protocol. While Centrifuge does not strictly adhere to the definition of a state channel, it addresses many problems of maintaining state off chain similarly. Quorum's private transaction layer[6] describes a mechanism by which the state of a contract is encrypted and only updated and validated by the parties that have access to the smart contract. A transaction on-chain is used to update the hash of the off-chain state, but doesn't give any guarantees that the offchain transaction was actually valid.

1.2 Current State

This paper defines the current implementation of the Centrifuge protocol. It represents a work in progress and will be updated to reflect the newest version of the protocol deployed. Centrifuge launched in early 2019 with the version specified for the first time here. The current version should be viewed as a first minimum viable product. Refer to the section on notable revisions (ref. 9) for a list of prior versions of this document.

[*]Lucas Vogelsang, `l@lucasvo.com`
[†]Manuel Polzhofer, `manuel@centrifuge.io`
[‡]Philip Stehlik, `philip@centrifuge.io`
[§]Vimukthi Wickramasinghe, `vimukthi@centrifuge.io`

2 High level description of components

The components of Centrifuge are a collection of Ethereum Smart Contracts and a peer to peer network implemented on libp2p[1] (see Fig. 1). Ethereum smart contracts are used for maintaining identities in a similar format to the ERC725 standard[2]. There is a contract deployed to store state commitments and a standard way of minting NFTs[3] from off chain Centrifuge documents.

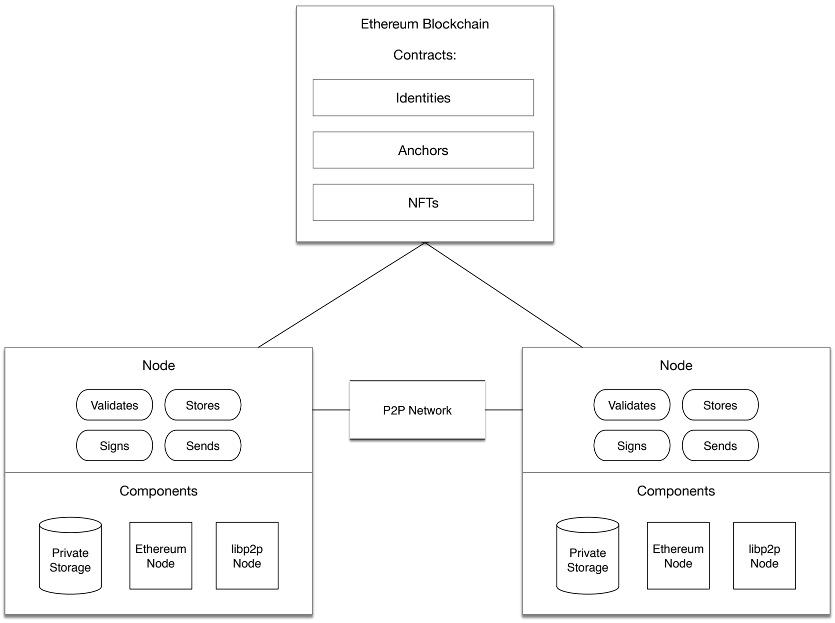


Figure 1: Overview of the components making up the Centrifuge protocol

2.1 Ethereum & P2P

Communication between nodes is done in two ways. To exchange information privately without a trusted third party or intermediary we are using libp2p' encrypted P2P communication stack. Ethereum is used as a blockchain to publish state commits to all participants, for executing smart contracts that manage identities, and allow the users to tokenize assets.

2.2 Identity

A node in the Centrifuge protocol uses different key pairs for signing documents and for authentication on the p2p layer. A user-deployed identity contract keeps the latest public keys. New keys can only be added by the user, and the contract can verify signatures on-chain. The identity contract thus represents the self sovereign identity of a user, as well as acting as a proxy contract for interaction with other on-chain decentralized applications. Centrifuge is adopting the DID-compatible ERC725 standard for identities.

2.3 Documents

A document is a well-known, structured set of fields with specific types. The protocol supports any document types as long as the formats are agreed upon and shared between the participants of a participant set– e.g. a document can be an invoice with agreed upon fields and line items, or a purchase order. The structure of the document becomes important for reaching consensus by attaching signatures to the document state, as well as creating specific attestations about a document at a later point in time. Documents are exchanged encrypted, and are only accessible for parties involved in this private data exchange. Collaborators can be added and removed from a document. Different collaborators can update a document and publish new versions within the set of nodes with access. A smart contract called *AnchorRepository* is used for carbon dating state updates and serves as a bulletin board[4] to

[1]`https://github.com/libp2p/specs`
[2]`https://github.com/ethereum/EIPs/blob/master/EIPS/eip-725.md`
[3]`https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md`

ensure that the update is made known to all collaborators. A document anchor is the root hash of the Merkle tree of the document. The tree is constructed by adding all fields of a document together with the collected digital signatures from all collaborators (an identity as defined above) as leaves in the tree. Publishing this anchor achieves that even if a party is censored on the P2P network, it can find out about the update by checking the Ethereum blockchain. A third party can easily verify the correctness of a received document on-chain and off-chain by reconstructing the Merkle root from the document based on the well-known document structure for the respective document type. Structuring the document as a Merkle tree allows creation of proofs only revealing individual fields of the document as opposed to revealing the entire document when making a statement about it.

2.4 NFT - Non Fungible Tokens

An integral part of Centrifuge is interaction with other applications built on Ethereum. One way to interact with the greater ecosystem is through NFTs. Centrifuge uses privacy-enabled NFTs[7] to allow for minting of NFTs that have verified attributes from private documents. The Merkle proofs that refer to a state commitment published in our *AnchorRepository* contract immutably link the NFT to an off-chain document.

3 Document and state commitments

The Centrifuge protocol does not have the concept of a global state for documents. Instead, the state is only shared across collaborators of a single *documents* which is the key component in the protocol. A document has a predefined schema to represent a specific document type, like a purchase order or an invoice. A Centrifuge node can create a document and share it with others. The transport of a document happens privately over secure channels in the P2P network. Every document collaborator keeps a local copy of a document in their storage. An update of a document can be triggered by multiple collaborators and is not restricted to the initial document creator. Whenever a change is made, a calculated merkle root of a document is committed on chain. A new version is only accepted by others if the document root hash exists in the *AnchorRepository* and the set of mandatory document fields in the new version satisfy protocol-specific requirements. A formal definition of the individual document components should expound the requirements of a valid new document state.

3.1 Conventions

We use the following conventions in the document. The lowercase letter d indicates a document in Centrifuge. Together with a subscript lowercase letter a field in the document or in a set is described. For example $d_{\texttt{id}}$ refers to the ID field in a document d. Lowercase letters with square brackets indicate an index. For example $d_{[2]}$ refers to the third document in a row. Arrows are used to describe communication with Ethereum and other participants in the P2P network. The arrow $\xrightarrow{\text{ETH}}$ is used to refer to Ethereum and $\xrightarrow{\text{P2P}}$ for the P2P network communication. For example $a \xrightarrow{\text{P2P}} b$, the right arrow describes a message send from a to b over the P2P network. A left arrow would indicate a P2P request or a call on an Ethereum smart contract (For example: $a \xleftarrow{\text{ETH}} b$).

3.2 Document

3.2.1 Fields

The smallest unit of a document is called *field*. A field F is a 3-tuple with a field name defined as $F_{\texttt{name}}$, a field value defined as $F_{\texttt{value}}$ and a salt defined as $F_{\texttt{salt}}$

$$F = (F_{\texttt{name}}, F_{\texttt{value}}, F_{\texttt{salt}}) \tag{1}$$

$$F_{\texttt{name}} \in \mathbb{B} \wedge F_{\texttt{value}} \in \mathbb{B} \wedge F_{\texttt{salt}} \in \mathbb{B}_{32} \tag{2}$$

Elements of $\mathbb{B}$ are an arbitrary-length byte array. Elements of $\mathbb{B}_{32}$ are a 32 byte array.

3.2.2 Document

A document d can be formally expressed as the union of the schema field set S with a set of core fields called core document denoted as C.

$$d = S \cup C \tag{3}$$

For referring to all schema fields S of document or to the core document fields C we introduce the following notation

$$d_S = S \tag{4}$$
$$d_C = C \tag{5}$$

3.2.3 Schema Fields

A document d contains a set of fields S described by a document schema. The document schema defines the type of a document. The document schema for an invoice would define the required fields for an invoice. The set schema fields S can be defined as

$$S = (F_{[0]}, ..., F_{[n]}) \tag{6}$$

In the set of specific fields S for a document d, every field has a unique field name $F_{[i]\texttt{name}}$ to identify a member of the set.

$$F_{[i]} \in S \Rightarrow F_{[i]\texttt{name}} \neq F_{[j]\texttt{name}}\, ; \forall F_{[j]} \in S, i \neq j \tag{7}$$

All participants in the protocol agree in advance on the schema of documents the would like to exchange. In addition to the predefined schema fields a schema could contain a set of customized attributes fields for required additional information.

3.2.4 Core Document Fields

The core document C defines the required fields for every document d. The core document contains signatures, hashes, and other relevant information that allows users to exchange any document data.

We introduce the fields in C step by step in the following section and define the required constraints of the core document fields. Variables denoted with $R_{\texttt{x}}$ are derived from other fields but are themselves not persisted on the document. $R_{\texttt{x}}$ variables are needed for signing and for anchoring documents in the smart contract. A document field which is a core document field is denoted with $d_{\texttt{x}}$. The core document[4] includes other fields not mentioned in this section which are introduced in the following sections step by step.

Identifier a $\mathbb{B}_{32}$ value defines the unique identifier of a document, formally $d_{\texttt{id}}$

Current Version a $\mathbb{B}_{32}$ value defines a unique id for the current version of a document, formally as $d_{\texttt{current}}$

Current Version Pre Image is a $\mathbb{B}_{32}$ value needed to define the current version of a document, formally as $d_{\texttt{current-img}}$

Previous Version a $\mathbb{B}_{32}$ value needed to refer to the previous version of a document, formally as $d_{\texttt{prev}}$

Next Version Pre Image is a randomly chosen $\mathbb{B}_{32}$ value needed to define the next version of a document. This should be kept private to collaborators who can update the state. Formally as $d_{\texttt{next-img}}$

Next Version a $\mathbb{B}_{32}$ value defines a unique identifier for the next version of a document, formally $d_{\texttt{next}}$

Document Root a $\mathbb{B}_{32}$ value defines a root hash for document, formally $R_{\texttt{doc-root}}$

Signing Root a $\mathbb{B}_{32}$ value defines the root hash of the Merkle tree containing all fields except for the signatures, formally $R_{\texttt{signing}}$

Signatures a set of signatures of the signing root denoted as $d_{\texttt{signatures}}$

Previous Root a $\mathbb{B}_{32}$ value defines the document hash of the previous version of the document. The previous root denoted as $d_{\texttt{prev-root}}$

Nonce is a $\mathbb{B}_{32}$ value needed to indicate the version history of a document, formally as $d_{\texttt{nonce}}$

[4] `https://github.com/centrifuge/centrifuge-protobufs/blob/master/coredocument/coredocument.proto`

3.2.5 Version History

The version history H is a set of documents. An update of an existing document in the Centrifuge protocol happens by creating a new version and linking this new version to the previously existing version. We define the set of all documents as D:

$$d \in D \tag{8}$$

Every node keeps a history of all versions of a document in which they are listed as a collaborator.

$$H = (d_{[0]}, ..., d_{[n]}) \tag{9}$$

In the initial version of a document d_0, the current version and the identifier will be equal.

$$d_{[0]\texttt{id}} = d_{[0]\texttt{current}} \tag{10}$$

The fields for the previous document root hash $D_{\texttt{prev-root}}$ and for the $D_{\texttt{prev}}$ id will be empty in the initial version.

$$d_{[0]\texttt{prev-root}} = \emptyset \tag{11}$$
$$d_{[0]\texttt{prev}} = \emptyset \tag{12}$$

For all other documents D in a version history H, the following conditions must be true:

Every document in a version history H must have the same identifier.

$$\{d_{[i]} \in H \mid \forall d_{[j]} \in H : d_{[i]\texttt{id}} = d_{[j]\texttt{id}}\} \tag{13}$$

The current version field of a document must be equal to the next version field of the previous document.

$$d_{[i]\texttt{current}} = d_{[i-1]\texttt{next}} \tag{14}$$

The previous version field of a document must be equal to the current version field of the predecessor document.

$$d_{[i]\texttt{prev}} = d_{[i-1]\texttt{current}} \tag{15}$$

The previous root hash of a document must be equal to the document root hash of the predecessor document.

$$d_{[i]\texttt{prev-root}} = d_{[i-1]\texttt{doc-root}} \tag{16}$$

The current pre-image of a document must be equal to the next pre image of the predecessor document.

$$d_{[i]\texttt{current-img}} = d_{[i-1]\texttt{next-img}} \tag{17}$$

The history of the document can therefore be seen as a doubly linked list. Every document has the document hash $d_{\texttt{previous}}$ of the predecessor document and defines the $d_{\texttt{id}}$ of the next one.

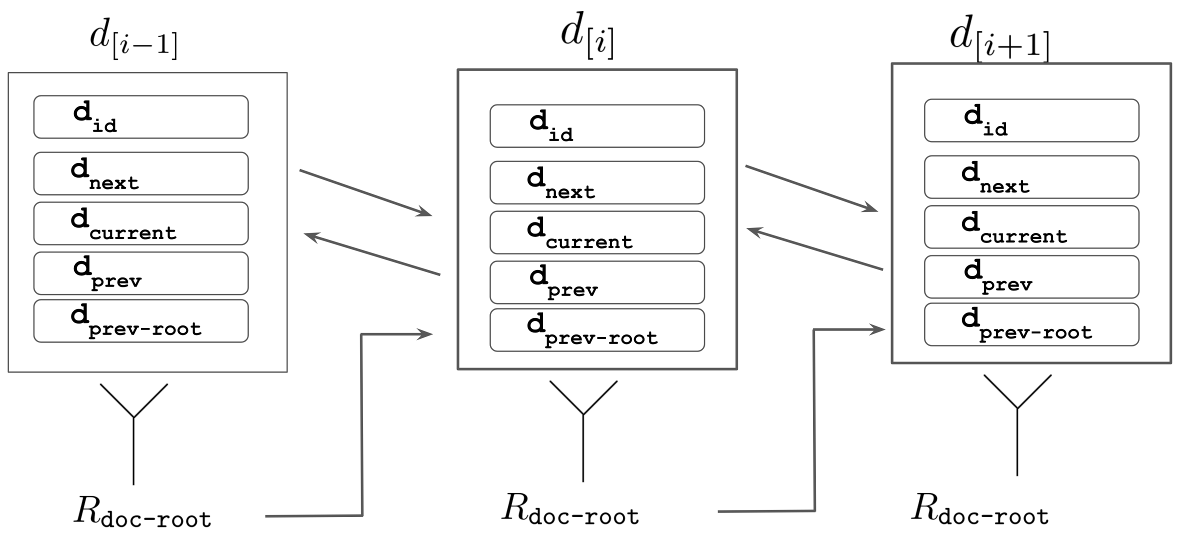
A node which performs a document update defines the the next version identifier $d_{\texttt{next}}$.

$$d_{[i]\texttt{next-img}} = \texttt{RAND(32)} \tag{18}$$

$$d_{[i]\texttt{next}} = \texttt{sha256}(d_{[i]\texttt{next-img}}) \tag{19}$$

The field $d_{\texttt{next-img}}$ is needed to prevent malicious anchoring of documents by non-collaborators. The value therefore must be kept secret from any non-collaborators. The function `RAND(32)` is a cryptographically secure random function and returns a byte 32 array. The function `sha256` denotes hash function SHA-2 256.

By referencing the previous root each state d, the history becomes an immutable attribute committed to in each version. To traverse the history H, one can follow the $d_{\texttt{prev}-\texttt{root}}$ and $d_{\texttt{prev}}$. To move forward in the list of history, H, one can go from $d_{\texttt{next}}$ to $d_{\texttt{current}}$.



$d_{[i-1]}$
$d_{[i]}$
$d_{[i+1]}$
$\mathbf{d}_{\texttt{id}}$
$\mathbf{d}_{\texttt{next}}$
$\mathbf{d}_{\texttt{current}}$
$\mathbf{d}_{\texttt{prev}}$
$\mathbf{d}_{\texttt{prev-root}}$
$R_{\texttt{doc-root}}$

Figure 2: Doubly linked list of different versions of a document. Indicating the relationship between different document index fields. The document root $R_{\texttt{doc-root}}$ is stored in the anchor registry and not in the document itself.

3.3 Calculation of the Root Hashes

Every root hash of a document $R_{\texttt{doc-root}}$ is stored in the *AnchorRepository* smart contract. The document root hash is calculated from three different Merkle trees (see Fig. 3).

3.3.1 Merkle Tree of Fields

A Merkle tree is a way to calculate a unique hash from a set of items. We formally define a Merkle tree function $\mathcal{M}$ to calculate a root hash $R \in \mathbb{B}_{32}$ of document fields F. The implementation of the Merkle tree function is introduced in Sec. 3.6.

$$R = \mathcal{M}(F_{[0]}, ..., F_{[n]}) \tag{20}$$
$$R \in \mathbb{B}_{32} \tag{21}$$

3.3.2 Schema Root and CoreDocument Root

As a first step, we calculate the Merkle root of the schema fields S and the core document fields C of a document d

Schema Root The Merkle root of the document schema data, formally $R_{\texttt{schema}}$

$$R_{\texttt{schema}} = \mathcal{M}(d_S) \tag{22}$$

CoreDocument Root the Merkle root of the core document data, formally $R_{\texttt{core}}$

$$R_{\texttt{core}} = \mathcal{M}(d_C) \tag{23}$$

3.3.3 Signing Root

The $R_{\texttt{signing}}$ is calculated from the SHA256 hash of the concatenated $R_{\texttt{core}}$ and $R_{\texttt{schema}}$

$$R_{\texttt{signing}} = \texttt{sha256}(R_{\texttt{core}} \| R_{\texttt{schema}}) \tag{24}$$

We formally define a helper function for calculating the $R_{\texttt{signing}}$

$$\begin{aligned} R_{\texttt{signing}} &= \mathsf{calculateSigningRoot}(d) && (25)\\ \mathsf{calculateSigningRoot}(d) &= \mathtt{sha256}(\mathcal{M}(d_C)\|\mathcal{M}(d_S)) && (26)\end{aligned}$$

3.3.4 Signatures Root

A Merkle root of the collaborator signature must be part of the document root hash $R_{\texttt{doc-root}}$. The signatures are relevant for the document consensus, explained in the following chapters.

Signature Root A signature root hash defines the Merkle root of all signatures $d_{\texttt{signatures}}$, formally $R_{\texttt{signatures}}$:

$$R_{\texttt{signatures}} = \mathcal{M}(d_{\texttt{signatures}}) \tag{27}$$

3.3.5 Document Root

The final document root hash $R_{\texttt{doc-root}}$ of the entire document d is defined as the SHA256 hash of the concatenated signing root hash $R_{\texttt{signing}}$ and signature root hash $R_{\texttt{signatures}}$

$$R_{\texttt{doc-root}} = \mathtt{sha256}(R_{\texttt{signing}}\|R_{\texttt{signatures}}) \tag{28}$$

A tree of the $R_{\texttt{doc-root}}$ calculation. (see Fig. 3)

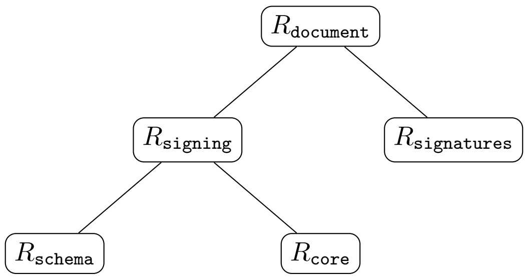


Figure 3: Calculation of the last layers of the document merkle tree. The leafs are the roots of merkle trees from the different document parts

We can define a second help function called $\mathsf{calculateDocumentRoot}(d)$ for calculating the entire $R_{\texttt{doc-root}}$.

$$\begin{aligned} R_{\texttt{doc-root}} &= \mathsf{calculateDocumentRoot}(d) && (29)\\ \mathsf{calculateDocumentRoot}(d) &= \mathtt{sha256}(\mathrm{R}_{\texttt{signing}}\|\mathcal{M}_{\texttt{tree}}(d_S)) && (30)\end{aligned}$$

3.4 State commitment

3.4.1 Anchors

An anchor is the root hash of a document $R_{\texttt{doc-root}}$ stored in an Ethereum smart contract called *Anchor-Repository*.

Every new document version d' must commit its document root hash $R_{\texttt{doc-root}}$ to the registry. A new document d' is only valid and accepted if the $R_{\texttt{doc-root}}$ exists as an anchor in the registry.

$$\begin{aligned} A &= (A_{\texttt{anchor-id}}, A_{\texttt{anchor-root}}, A_{\texttt{anchor-blockheight}}) && (31)\\ A_{\texttt{anchor-id}} &= d_{\texttt{current}} && (32)\\ A_{\texttt{anchor-root}} &= R_{\texttt{doc-root}} && (33)\\ A_{\texttt{anchor-blockheight}} &= \mathtt{block.number} && (34)\end{aligned}$$

The anchor id of a new document $d_{[i]\texttt{current}}$ is defined by the previous document $d_{[i-1]\texttt{next-img}}$. The same image is stored in the current document in $d_{[i]\texttt{current-img}}$. The hash of the $d_{[i]\texttt{current-img}}$ defines the current document identifier and the anchor id.

$$\begin{aligned} d_{[i]\texttt{current}} &= \texttt{sha256}(d_{[i-1]\texttt{next-img}}) \quad (35) \\ A_{\texttt{anchor-id}} &= d_{[i]\texttt{current}} \quad (36) \end{aligned}$$

The `block.number` contains the Ethereum block number the transaction got mined and acts as a timestamp for the anchor.

3.4.2 Verifying a document

Any user can verify the correctness of the document by comparing the root hash to the committed hash on chain. We formally define a method called getAnchor which returns an anchor from the contract.

$$\begin{aligned} A_{\texttt{anchor-current}} &\xleftarrow{\text{ETH}} \mathsf{getAnchor}(d_{[i]\texttt{current}}) \quad (37) \\ A_{\texttt{anchor-next}} &\xleftarrow{\text{ETH}} \mathsf{getAnchor}(d_{[i]\texttt{next}}) \quad (38) \\ & \quad (39) \end{aligned}$$

3.4.3 Latest Version

Because in each document (state) update the next anchor id is already defined, anyone can verify that they have the latest version by checking the anchor registry. A document $d_{[i]}$ is the latest version if the following condition is true:

$$A_{\texttt{anchor-current}} = R_{\texttt{doc-root}} \quad \wedge \quad A_{\texttt{anchor-next}} = 0 \quad (40)$$

If an anchor for $A_{\texttt{anchor-next}}$ exists, the current document $d_{[i]}$ is not the latest version. It is not possible to overwrite an existing anchor id or invalid it. For committing an anchor the $d_{[i]\texttt{current-img}}$ is needed therefore only document collaborators could propose a next version.

3.4.4 Pre-Commit

A document is committed only after reaching consensus with the other collaborators on the document. Consensus is reached by getting a cryptographic signature from other parties by sending them the signing root. To deny the counter-party the free option of publishing its own state commitment upon receiving a request for signature, the node can first publish a pre-commit. A pre-commit locks a commit to a specific document state for a given anchor id for a predetermined number of blocks. Only the `msg.sender` of the pre-commit is allowed to commit a corresponding anchor before the pre-commit has expired.

$$\mathsf{preCommit}(d_{[i]\texttt{current}}, R_{\texttt{signing}}) \xrightarrow{\text{ETH}} \mathsf{AnchorRepository} \quad (41)$$

The pre-commit stores the provided signing root $R_{\texttt{signing}}$ on the smart contract. Only a $R_{\texttt{doc-root}}$ in the commit which includes the same $R_{\texttt{signing}}$ is accepted as a valid commit. It is not possible to provide a new preCommit $R_{\texttt{signing}}$ within the next blocks for the given anchorId.
In the event that the user does not provide a commit within the lock time, the commit will be unlocked, and the `preCommit` method will be open to everyone again.

3.4.5 Commit

The anchor registry's `commit` method does not have any restrictions, except for the pre-commit flow as described above. If no valid pre-commit exists, anyone can commit an anchor to the registry. A commit of a new anchor is described by the following function.

$$\mathsf{commit}(d_{[i]\texttt{current-img}}, R_{\texttt{doc-root}}, P_{\texttt{sign-proof}}) \xrightarrow{\text{ETH}} \mathsf{AnchorRepository} \quad (42)$$

Next Pre-Image

The commit method hashes $d_{[i1]\texttt{next-img}}$ inside the smart contract to get the $A_{\texttt{anchor-id}}$. The $d_{[i1]\texttt{next-img}}$ of the previous version is needed to commit the next version. It is not possible to use $d_{[i1]\texttt{next}}$ (equal to $d_{[i]\texttt{current}}$) directly. In some cases, the field $d_{\texttt{next}}$ is revealed to the outside world to prove the latest version of a document. (See 8). The anchor contract challenges the submitter to reveal the pre-image to avoid an attacker committing an malicious anchor for $d_{\texttt{next}}$. The field $d_{\texttt{next-img}}$ is never revealed to the outside world.

Signing Root Proof
In case a pre-commit exists for the `anchorId`, calculated by hashing the given pre-image, the `msg.sender` must be the same as in the pre-commit. In the pre-commit, the signing-root $R_{\texttt{signing}}$ has been stored. A merkle proof verifies if the $R_{\texttt{signing}}$ is part of the $R_{\texttt{doc-root}}$ tree. $P_{\texttt{sign-proof}}$ is an array of $\mathbb{B}_{32}$ arrays which contains the needed sibling hashes for the merkle proof.

3.5 Protobuf for Schema Definitions

Every document has a predefined schema which is agreed upon by all network participants. The Protobuf[5] message format is used for defining the document schema. It was chosen as a schema and serialization standard for its efficient binary format, broad support in different programming languages and fixed schema. An example of such a message is shown below in Fig. 4.

Figure 4: An example protobuf definition

```
message ExampleDocument {
  string name = 1;
  repeated int numbers = 2;
}
```

All documents that are supported by Centrifuge protocol are hosted on GitHub in a repository at `https://github.com/centrifuge/centrifuge-protobufs`.

3.6 Precise proofs: Merkle tree format

Merkle trees are efficient ways to prove membership in a set. By encoding the leaves of the tree, a Merkle tree can be used to make a proof about the value of a field specific in the schema. The value of the field is prefixed with a byte encoded field path, P, and a salt, S.

3.6.1 Path encoding

The reference for the path encoding is the precise-proofs[6] library but is briefly summarized here. Protobuf uses 32bit unsigned integers to identify fields within a message uniquely. The property p is defined using these values. The encoding of property names also supports nested messages, mappings, as well as repeated fields (arrays).

$$p_F = \mathsf{FieldToProperty}(F_{\texttt{name}}) \tag{43}$$

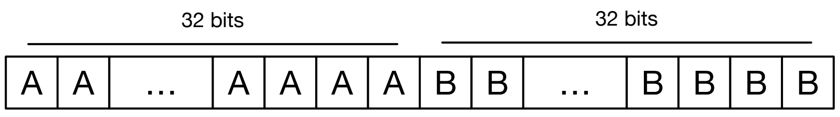


Figure 5: The field numbers for a path `"A.B"` are concatenated into a fixed length delimited byte value

In this paper, we refer to the fields by their human readable name using JavaScript-style dot notation for easier readability. However, the byte format will be used when encoding a leaves.

[5]`https://developers.google.com/protocol-buffers/`
[6]`https://github.com/centrifuge/precise-proofs`

3.6.2 Salts

When proving, the hash of the sibling of the field is revealed. A salt is therefore added to the value of the leaves in order to avoid a rainbow table attack on sibling leaves revealed in a proof. The salt F_{salt} is stored in a special field on the *CoreDocument* protobuf message called `salt` in form of a map where a salt is stored for each field by the property p_F.

3.6.3 Construction of the tree

For each leaf, λ, the name P, the salt S and the value v are appended and then hashed with `sha256`.

$$\lambda = \texttt{sha256}(p_F \parallel F_{\texttt{value}} \parallel F_{\texttt{salt}}) \tag{44}$$

$$\mathrm{T(D)} = \mathcal{M}(\lambda_1 \parallel \lambda_2 \parallel ... \parallel \lambda_n) \tag{45}$$

3.6.4 Merkle Proof Validation

A Merkle proof can be seen as a function which proves if a provided field $P_{\texttt{field}}$ is part of document. By calling the NFT mint method the value of these fields is revealed to the public.

$$\begin{aligned} v = \mathcal{M}_{\texttt{proof}}(A_{\texttt{anchor-current}}, P_{\texttt{field}}, P_{\texttt{sibling-hashes}}) \\ v \in \{0;1\} \end{aligned} \tag{46}$$

The $P_{\texttt{field}}$ could be any field of the core document d_C or the schema field d_S. The required $P_{\texttt{sibling-hashes}}$ can be generated with the precise proofs library inside the node and contains the siblings of the field in each layer of the Merkle tree.

4 Participants

A user of Centrifuge is defined as an entity that owns a set of keypairs and is running a node to communicate with other nodes and the smart contracts. A user can be an individual or a different entity. Digital signatures are used to authenticate a user. A user controls a smart contract on Ethereum where they publish the public keys in use.

4.1 Self-Sovereign Identity

A W3C standard for self-sovereign identities called DID (Decentralized Identifiers)[10] is currently under development. Centrifuge identities are compatible with the ERC725 standard (see below 4.1.1) for identities which is self-sovereign (SSI), as well as DID compatible.

4.1.1 ERC725 Identity Standard

Many decentralized applications built on Ethereum require a form of identity to standardize interactions. A set of different standards for identities have been proposed to address these standardization issues[1][9]. In ERC725[7], a user's identity is represented as a smart contract. The user is the owner of the identity, and retains exclusive control over who is allowed to add new keys and interact with other contracts in Ethereum on behalf of the identity. Centrifuge is adopting the ERC725 standard but we are fully aware it might in change in the future and the current version is not final.

4.1.2 DID - Centrifuge ID

The address of the identity contract represents the unique ID of a user in the protocol. The ID can be extended with information about the Ethereum network to represent a valid DID. A sample DID representation of an identity address[11]:

```
did:erc725:mainnet:2F2B37C890824242Cb9B0FE5614fA2221B79901E
```

Within Centrifuge, only the address of the identity contract is used to describe a user ID as the only supported identities are Ethereum smart contracts.

$$\texttt{DID} \in \mathbb{B}_{20} \tag{47}$$

[7]A newer version of ERC734 dated February 2019 defines key management, while the newer ERC725 v2 defines only the proxy functionality. The Centrifuge protocol currently implements ERC725 v1.

4.1.3 Key Management

A user can add different public keys to their identity. A number of different signature and encryption standards are supported. A key can be seen as a 4-tuple.

$$K = (K_{\texttt{public}}, K_{\texttt{private}}, K_{\texttt{type}}, K_{\texttt{purpose}}) \tag{48}$$

An ERC725 key is a 3-tuple of three elements, not including the private key of the user. A user can use the same key for multiple purposes and needs only to add it once to the identity contract.

$$\begin{aligned} K_{\texttt{ERC725}} &= (K_{\texttt{public}}, K_{\texttt{type}}, K_{\texttt{purposes}}) && (49)\\ K_{\texttt{public}} &\in \mathbb{B}_{32} && (50)\\ K_{\texttt{type}} &\in \mathbb{B}_{32} && (51)\\ K_{\texttt{purposes}} &\in \mathbb{B}_{32}^{n} && (52)\\ |K_{\texttt{purposes}}| &> 0 && (53)\end{aligned}$$

The $K_{\texttt{ERC725}}$ key is the representation of a K in the identity contract.

4.1.4 Identity

Users have an unique id $I_{\texttt{DID}}$ which is the address of their identity contract. A user needs at least 4 different keys for purposes needed in the protocol.It is possible to add other keys to the identity contract or to add keys with same purpose.

We can formally define the identity of a user as:

$$\begin{aligned} I &= (\texttt{DID}, k_{\texttt{man}}, k_{\texttt{p2p}}, k_{\texttt{sign}}, k_{\texttt{action}}) && (54)\\ k &\in K && (55)\end{aligned}$$

4.1.5 Proxy Contract

A contract in Ethereum does not differentiate between calls from a user account or another contract. A proxy contract is a contract that can forward a call to another contract. From the perspective of the called contract, the `msg.sender` is the proxy contract. Identities act as a proxy contract. This allows an identity contract to interact with other contracts like any other account. The identity owner can add Ethereum addresses with the purpose `ACTION-KEY` to their identity that are able to invoke such proxy calls.

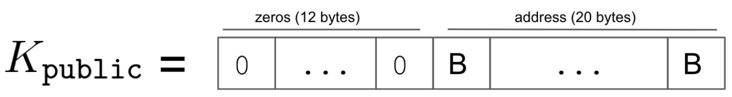


Figure 6: Storage of an Ethereum address in the identity contract.

These Ethereum addresses are allowed to use the proxy functionality of the contract. Later, the owner can revoke the permissions if needed. An Ethereum address has 20 bytes $\mathbb{B}_{20}$. The $K_{\texttt{public}}$ requires the public key to be 32 bytes. An Ethereum address needs to be converted into a 32 bytes array by adding 12 leading zeros to it. (see figure 6)

4.1.6 Key Types

In the protocol, four different key types are required. (see 4.1.6) The identity contract defines a set of methods for adding and revoking keys. Every public key must be added to the identity contract. The ERC725 standard defines a set of pre-defined key purposes. It is possible to add individual purposes for the Centrifuge protocol.

4.1.7 Management Key

The management key, $k_{\texttt{man}}$, is a reserved key type in the ERC725 standard and represents the key pair of the owner. This key has the right to add new keys, revoke existing ones, or allow other addresses to use the proxy functionality.

Purpose	Purpose ID	Key Type
Management Key	1	secp256k1
Action Key	2	secp256k1
Centrifuge Signing Key	`SIGNING = sha256("CENTRIFUGE@SIGNING")`	secp256k1
Centrifuge P2P Discovery	`P2P_DISCOVERY = sha256("CENTRIFUGE@P2P_DISCOVERY")`	ED25519

Table 1: Overview of the key types used in the Centrifuge protocol.

4.1.8 Action Key

The action key, $k_{\texttt{action}}$, is a reserved key type in the ERC725 standard. An action key can be added by a management key and allows the owner of the action key to use the proxy functionality. Only an Ethereum address stored as a key with the action purpose can trigger the proxy functionality.

4.1.9 Centrifuge Signing Key

The signing key, $k_{\texttt{sign}}$, is a Centrifuge specific key purpose. It is used to sign documents together with other collaborators. The key type needs to be secp256k1 because a signature can be part of a Merkle proof and needs to be verified on-chain to guarantee another identity has signed a document. The identity contract has a method which verifies signatures based on a key purposes. Secp256k1 signature verification has native support in the EVM.

4.1.10 Centrifuge P2P Discovery Key

See section 7.2.

4.1.11 Deployment of an Identity

An identity contract is deployed by a user through a well-known, Centrifuge-specific Factory contract, the `IdentityFactory` contract, that constructs the identity for the caller and keeps track of identities which are created within the Centrifuge network. This tracking allows for later verification of identity "validity" for users within the network.

4.1.12 Revoking Keys

The ERC725 identity standard allows key revocation. A revoked key is not removed from the identity contract. Currently other Centrifuge contracts like the NFT registry only supporting ERC725 identity contracts created by the Factory contract. In our ERC725 version the key is only marked as revoked. In the protocol, a message which is signed with a revoked key before the key was revoked is accepted. However, a message which is signed with a revoked key after the key was revoked is not accepted.

5 Document State Consensus

In 3, we introduced how state is committed to and in 4, we defined the attributes and behaviors of individual actors in the system. Building upon this, we now introduce our mechanism for enabling the reaching of consensus among different parties.

5.1 Consensus Trade-offs

For the initial protocol version we have decided to avoid a fully BFT consensus scheme. The protocol does not guarantee data availability for every participant. Instead, the focus is on preserving privacy in the optimistic case and requiring any consensus failures to be settled out of band.

5.2 Reaching consensus on the document state

When a collaborator wants to update the state and commit to a new version, the document changes are first sent to all collaborators on a document in order to collect the signatures by the collaborators. The collaborators sign the signing root, $R_{signing}$.

In the following section, the pre-commit and commit is described. A user does not need to use a pre-commit and can commit a document state immediately. However this has drawbacks if there are byzantine nodes.

5.2.1 Consensus Protocol

Each collaborator, an identity I, is found in the P2P network and the new state is transmitted with a request to provide a signature. We define the user who initiates the state update as ι and collaborators c_n.

$$\iota, c \in I \tag{56}$$

$$d_{\texttt{collaborators}} = (\iota, c_{[0]}, ..., c_{[n]}) \tag{57}$$

We begin with a document d that will be updated from $d_{[n]} \rightarrow d_{[n+1]}$ or in shortened form from $d \rightarrow d'$.

$$d' \quad = \quad d_{[n+1]} \tag{58}$$

When proposing a new state d', the user ι will set the coredocument field author to its `DID`.

$$d'_{\texttt{author}} = \iota_{\texttt{DID}} \tag{59}$$

The user ι first calculates the signing root, $R_{\texttt{signing}}$ and signs it with its own key and adds it to the set of signatures on d'.

$$\begin{aligned}
R_{\texttt{signing}} &= \mathsf{calculateSigningRoot}(d') && (60)\\
\mathsf{preCommit}(d'_{\texttt{current}}, R_{\texttt{signing}}) &\xrightarrow{\text{ETH}} \mathsf{AnchorRepository} && (61)\\
s &= \mathsf{sign}(\iota_{\texttt{k.sign}}, R_{\texttt{signing}}) && (62)\\
d'_{\texttt{signatures}[\iota_{\texttt{DID}}]} &= s && (63)
\end{aligned}$$

The user then pre-commits to the state and gets signatures from all the collaborators. After a successful pre-commit, the node sends the new state d' to all collaborators and requests their signature on the new state.

$$\begin{aligned}
&s \xleftarrow{\text{P2P}} \mathsf{requestSignature}(d', c)\\
&d'_{\texttt{signatures}[c_{\texttt{DID}}]} = \begin{cases} s, & \text{if} \quad \mathsf{validateSignature}(c, R_{\texttt{signing}}, s) = 1 \\ \emptyset, & \text{else} \end{cases} \qquad \forall c \in d_{\texttt{collaborators}} \backslash \{\iota\}
\end{aligned} \tag{64}$$

If a collaborator refuses to sign, is unreachable or returns an invalid signature, it is not added to $d'_{\texttt{signatures}}$.

5.2.2 Receiving a signature request

The collaborator c that receives the signature request first validates the update by comparing the proposed updated document to the previous version of the document that it has in store.

$$\begin{aligned}
d' &\xleftarrow{\text{P2P}} \mathsf{recieve}(\iota) && (65)\\
d &= \mathsf{load}(c, d'_{\texttt{prev}}) && (66)\\
\mathsf{validate}(d, d') &= 1 && (67)
\end{aligned}$$

The function $\mathsf{validate}$ is explained in detail in the next section. The collaborator then verifies that there is a valid signature by the sender, ι on the document and signs it.

$$\begin{aligned}
A_{\texttt{signing-root}} &\xleftarrow{\text{ETH}} \mathsf{getPreCommit}(d'_{\texttt{current}}) && (68)\\
R_{\texttt{signing}} &= \mathsf{calculateSigningRoot}(d') && (69)\\
A_{\texttt{signing-root}} = R_{\texttt{signing}} &\vee A_{\texttt{signing-root}} = \emptyset && (70)\\
\mathsf{validateSignature}(\iota, R_{\texttt{signing}}, d'_{\texttt{signatures}[\iota]}) &= 1 && (71)\\
\mathsf{sign}(c_{\texttt{k.sign}}, R_{\texttt{signing}}) &= s && (72)\\
\mathsf{sendSignature}(s) &\xrightarrow{\text{P2P}} \iota && (73)
\end{aligned}$$

5.2.3 Publishing the commit

When all signatures have been collected, or before the timeout for the pre-commit expires, the node commits to the state.

After receiving all the signatures or the before the pre-commit expires the node finally calculates the document root and commits the anchor. In case of a timeout the node commmits only with the received signatures.

$$\begin{aligned} R_{\texttt{doc-root}} &= \mathsf{calculateDocumentRoot}(d') && (74)\\ \mathsf{commit}(d'_{\texttt{next-img}}, R_{\texttt{doc-root}}, M_{\texttt{proofs}}) &\xrightarrow{\text{ETH}} \mathsf{AnchorRepository} && (75)\\ & && (76) \end{aligned}$$

The state update is now considered complete and the node ι sends an update to all collaborators and readers.

$$\begin{aligned} \mathsf{sendDocument}(d') \xrightarrow{\text{P2P}} \; c \\ \forall c \in \{d'_{\texttt{collaborators}} \cup d'_{\texttt{readers}}\} \end{aligned} \qquad (77)$$

5.2.4 Recieving a commit document

After the user published the commit every collaborator should receive the final version of d' including the collaborators signatures.

$$\begin{aligned} d' &\xleftarrow{\text{P2P}} \mathsf{recieve}(\iota) && (78)\\ A_{\texttt{doc-root}} &\xleftarrow{\text{ETH}} \mathsf{getCommit}(d'_{\texttt{current}}) && (79)\\ R_{\texttt{doc-root}} &= \mathsf{calculateDocumentRoot}(d') && (80)\\ A_{\texttt{doc-root}} &= R_{\texttt{doc-root}} && (81)\\ \mathsf{validate}(d, d') &= 1 && (82)\\ \mathsf{validateSignatures}(d'_{\texttt{collaborators}}, R_{\texttt{signing-root}}, d'_{\texttt{signatures}}) &= 1 && (83) \end{aligned}$$

A collaborator can verify the received document d' if actually an anchor has been committed and which other collaborators signed the document.

6 Validators and Rules

A collaborator should only sign a state update, as described in 5.2, if it is considered to be valid. In the following paragraphs, we describe both the protocol-level validation rules as well as document-specific rules.

6.1 Protobuf Validation

A document consists of two parts. The schema fields (Sec. 3.2.3) depend on the document type (e.g. *Purchase Order*, *Invoice*). The core document fields (Sec. 3.2.4) are part of every document. Protobuf's schema definitions provides significant built-in validation by enforcing types and fields. When exchanging the document via P2P messages, they are marshalled into the protobuf byte format and thus type-enforced. We will skip over these validation rules and assume that a user would always reject a document that is an invalid protobuf message.

6.2 Schema and Base Validation

A validator defines a set of transition rules for a document field or a set of document fields. For example, it would be not allowed to change the order quantity field of a purchase order to a negative value. Formally, a validator V is a function which has as an input the current document d and the next document d'.

$$\begin{aligned} v &= \mathsf{V}(d, d') && (84)\\ v &\in \{0, 1\} && (85) \end{aligned}$$

The output of a validator function V is either 1 for accepted or 0 for denied. If the field transitions are described in the validator V the validator returns 1. The validator V must also define case $d = \emptyset$ for a given first version of a document.

6.2.1 Base Validation

Every document has the same set of base validators B. The base valdiators check the core document conditions introduced in section 3.2.5.

$$B = (V_0, ..., V_n) \tag{86}$$

6.2.2 Schema Validation

Together with the base validators every schema has S has a set of different document validators for describing a document schema like invoice or purchase order.

$$S = (V_0, ..., V_n) \tag{87}$$

We define a validation which evaluates all Validators V in a set Υ.

$$\Upsilon = B \cup S \tag{88}$$

$$\mathsf{validation}(d, d', \Upsilon) = \prod_{i=0}^{n} V_i(d, d') \qquad \forall V \in \Upsilon \tag{89}$$

All validators need to accept the proposed document d'. Otherwise, the function will return 0 for denied.

$$\mathsf{validation}(d, d', \Upsilon) = 1 \tag{90}$$

In the special case of the first document $d_{[0]}$, schema validation must occur.

$$\mathsf{validation}(\emptyset, d', \Upsilon) = 1 \quad \text{for} \quad d' = d_{[0]} \tag{91}$$

6.3 Validate Rules

A document has specific rules which define the read and write permissions field by field. A rule defines a set of roles which are allowed to write, read or should sign a document. A proposed new document d' is not allowed to violate the rules. Therefore for each changed field in the document d' the the rules need be evaluated.

$$\begin{aligned} r = \mathsf{validateRules}(d, d', d'_{\texttt{author}}, d'_{\texttt{rules}}) \\ r \in \{1; 0\} \end{aligned} \tag{92}$$

6.3.1 Roles

Every document can have a set of roles. Defined as $d_{\texttt{roles}}$

$$d_{\texttt{roles}} \quad = \quad (\mathbb{R}_{[0]}, ..., \mathbb{R}_{[n]}) \tag{93}$$

Roles are used to define different groups of document collaborators. Every document has at least one role with a single member.

$$\mathbb{R} \quad = \quad (\mathbb{R}_{\texttt{id}}, \mathbb{R}_{\texttt{members}}) \tag{94}$$

$$\mathbb{R}_{\texttt{id}} \quad \in \quad \mathbb{B}_{32} \tag{95}$$

$$\mathbb{R}_{\texttt{members}} \quad = \quad (M_{[0]}, ..., M_{[n]}) \tag{96}$$

A member of a role can be the `DID` of another Centrifuge user or an access token. Which would allow the holder of the token to be part of the role. The third possible member of a role is the owner of an NFT, which could represent the document on-chain.

$$M \in \{\texttt{DID}; \texttt{access\_token}; \texttt{NFT}\} \tag{97}$$

6.3.2 Rules

Rules give the linked roles certain capabilities on a document. Like read or write access to a document. The rules are denoted as $d_{\texttt{rules}}$

$$d_{\texttt{rules}} = (R_{[0]}, ..., R_{[n]}) \tag{98}$$

$$\begin{aligned} R &= (R_{\texttt{action}}, R_{\texttt{roles}}) && (99) \\ R_{\texttt{roles}} &= (\mathbb{R}_{[0]}, ..., \mathbb{R}_{[n]}) && (100) \\ R_{\texttt{action}} &\in \{\texttt{read}; \texttt{read\_sign}; \texttt{write}\} && (101) \\ & && (102) \end{aligned}$$

6.3.3 Collaborators and Readers

A document does not have global state and instead is only shared with a limited set of nodes. There are two groups of nodes that a document is shared with. The first group is comprised of nodes which are part of the consensus and will be asked to sign a document state. We refer to these nodes as collaborators, or $d_{\mathsf{collaborators}}$. The second group of nodes are ones that have the permission to read a document but not mutate it or sign any consensus update. We refer to the set of readers as d_{readers}.

$$\begin{aligned} d_{\texttt{readers}} &= \mathsf{getAllMembersWithAction}(d_{\texttt{rules}}, \texttt{read}) && (103) \\ d_{\texttt{collaborators}} &= \mathsf{getAllMembersWithAction}(d_{\texttt{rules}}, \texttt{read\_sign}, \texttt{write}) && (104) \end{aligned}$$

6.4 Validate Function

A user signs the signing root of a document only if all validators in the schema are accepted and if no document rule is violated.

$$\mathsf{validate}(d, d') = \mathsf{validation}(d, d', \Upsilon) * \mathsf{validateRules}(d, d', d'_{\texttt{author}}, d'_{\texttt{rules}}) \tag{105}$$

$$\mathsf{validate}(d, d') = 1 \tag{106}$$

6.4.1 Validation

If the user has both the previous and the current version of the document, it will sign off on the state update. A document signature of a collaborator on both d and $d\prime$ means the collaborator validated both the format of d and $d\prime$. It also indicates that the user has validated that no validation rules were violated transitioning from d to $d\prime$.

6.4.2 Document Signature

As part of the consensus scheme other collaborators need to sign the signing root of a document. Adding a signature signals that the collaborator received the document and accepted the changes from d to d' according to the $\mathsf{validate}$ function. However it is needed to differentiate between new collaborators and collaborators with the full version history H.

6.4.3 New Collaborator Signature

During the life span of a document, new collaborators could be added or existing once removed. A newly added collaborator can only validate the received new document d' and not the transfer from d to d' which some validations V require. Therefore a signature contains the d_{nonce} of the earliest version a collaborator received. For example a nonce value of zero would indicate that a collaborators validated based on the entire document history. A new collaborator with access rights to previous versions would request older versions to completely validate the document history before adding the signature.

6.4.4 Document Acceptance

Adding a signature to a document doesn't mean a collaborator accepted the content of document from a business perspective. It only means the collaborator received the document and the proposed changes are not violating any document rules or validations. (See section 6.3.1).

A collaborator could for example receive a valid invoice document which would be signed during the document consensus but the invoice is not automatically accepted. The acceptance of a document requires an additional active document update step from a collaborator. For accepting a document an additional attribute field is used for document acceptance. The collaborator creates an additional signature of all the field which are accepted.

$$\begin{aligned} m &= (d_{\mathtt{id}}, d_{\mathtt{current}}, (F_{[0]}, ..., F_{[n]})) && (107) \\ s &= \mathtt{SIGN}(m, k_{\mathtt{sign}}) && (108) \\ d_{\mathtt{accepted}_{[\mathtt{did}]}} &= (d_{\mathtt{id}}, d_{\mathtt{current}}, m, s) && (109) \end{aligned}$$

Together with the $d_{\mathtt{id}}$ and $d_{\mathtt{current}}$ the signature s and the message m are added as value of the attribute field. The document identifiers are required otherwise the signature could be copied from other documents. A signature from a valid signing key of a collaborator in the $d_{\mathtt{accepted}}$ attribute field of an anchored document version indicates an acceptance of specific fields.

6.4.5 Consensus failure

The current design of the consensus protocol does not covers TODO Should one of the collaborators be unavailable or refuse to sign the proposed document update, the node will submit the new update on-chain without the signature of the offline node. There is no guarantee in the protocol that a node will collaborate at this point in time. If a signature is missing, it can mean one of following:

Offline The node was offline at the time of update

Denial of service The node refused to sign the update for selfish reasons

Consensus failure The node deemed the state transition to be invalid

Censorship The updating node didn't deliver the new state update to the counter-party

A third party can be used to act as a validator to mitigate some of these problems, but this is beyond the scope of the protocol in its current version. The primary objective of the protocol at this point is to reach consensus in an optimistic scenario. Should any of these happen, it will have to be dealt with out of band.

7 P2P Wire Protocol

Nodes on the Centrifuge peer to peer network communicate using the Centrifuge Wire Protocol, which utilizes libp2p[8] streams as the underlying network transport layer.

7.1 Usage of libp2p

libp2p provides building blocks for constructing peer to peer distributed systems and protocols. The Centrifuge network relies on the following components of libp2p for its functionality:

7.1.1 Node Identifier (PeerID)

A node in a libp2p network must generate a public-private key pair in order to participate in the network. The node's identifier is the multihash[9] or the sha2-256 hash of its public key, and is also known in libp2p terminology as the PeerID[10].

[8] `https://libp2p.io/`
[9] `https://github.com/multiformats/multihash`
[10] `https://github.com/libp2p/go-libp2p-peer`

7.1.2 Node Discovery

Discovery of a node in a libp2p network happens mainly through distributed hash table which works as key-value store to map a node's identifier to its physical location. Centrifuge network uses the Kademlia DHT implementation[11] provided by libp2p. The DHT based discovery works in two stages. The first stage of discovery is the connection to bootstrap nodes which are hard-coded into the node's configuration at the start. Bootstrap nodes allow newly joining or restarting nodes to query the updated DHT and refresh their own list of known peers(peer store) based on that. Centrifuge provides several sets of bootstrap nodes corresponding to different Ethereum networks such as Rinkeby or Kovan. Each set of bootstrap nodes together with the smart contracts which are deployed on the corresponding Ethereum network defines a Centrifuge network. For example, as of the time of writing, the `RussianHill` Centrifuge network corresponds to bootstrap nodes and smart contracts which are deployed on the Ethereum `Rinkeby` testnet.

Once a Centrifuge node is connected to the bootstrap nodes on Centrifuge network, and has updated its peerstore, the second stage of discovery starts. This is through receiving updates to the DHT from the nodes peers using the DHTs P2P protocol. All nodes in the Centrifuge network use the `centrifuge-dht` content topic to subscribe and publish their availability to other nodes in the network. When there is an update to the content of the topic all nodes in the network receive notifications.

7.1.3 Encrypted Transport

libp2p defines a common network security transport interface. Centrifuge uses the secio[12] implementation of that interface to create encrypted connections between the network nodes. The secio library executes the following steps, similarly to a TLS handshake, to establish a secure, encrypted channel to a remote node.

1. Propose and obtain agreement for supported exchanges, ciphers and hashes together with the public key of the proposer node for identification.
2. Perform `Elliptic-curve Diffie-Hellman` key exchange and generate a shared secret key. Here, the node's public key (node identifier) is used for initial message authentication.
3. Verify encryption using the new shared secret.

Once the shared secret key is established, it is used for encryption of all traffic on the channel. A brief description about libp2p encryption philosophy is available in the encryption spec[13].

7.1.4 Stream Multiplexing

The ability to multiplex protocols at multiple levels of the networking stack is a defining characteristic of libp2p with multi-multiplexing features[14]. Multiplexed protocol identifiers are strings in the form `/protocol-name/<optional string1>/../<optional string n>`. For example, `/http/v2`. A program can register handlers for each of these protocols in the libp2p host and it will take care of routing the messages of a protocol to the appropriate handler.

Centrifuge uses TCP level stream multiplexing with libp2p to support multiplexing of stream traffic intended for different Centrifuge accounts (`DIDs`) to a single TCP port in a node. Refer section 7.3 for more details.

7.2 Associating a Node(PeerID) with a DID

An identity on the Centrifuge network needs to be associated with a node in order for it to be able to perform Centrifuge-specific actions such as creating and signing documents. As introduced in section 7.1.1, a node's public key serves as its identity on the Centrifuge network. Therefore, to associate a node to a DID, the DID identity contract (appendix 4.1) on Ethereum stores its associated node's public key(ed25519) with the purpose ID - `P2P_DISCOVERY`. From the DID perspective, this key is known as the `P2P Discovery Key`, as any known Centrifuge DID can be looked up for its associated p2p node id using the method `getKeyByPurpose` with the purpose ID - `P2P_DISCOVERY` (refer section 4.1.6). TODO explain that its only one to one association right now

[11] `https://github.com/libp2p/go-libp2p-kad-dht`
[12] `https://github.com/libp2p/go-libp2p-secio`
[13] `https://github.com/libp2p/specs/blob/master/3-requirements.md#33-encryption`
[14] `https://github.com/libp2p/specs/blob/master/3-requirements.md#32-multi-multiplexing`

7.3 Multi-account(DID) handling

A node in the Centrifuge network can identify traffic associated with a single DID even at the wire protocol level. This is achieved through the use of libp2p stream multiplexing. A protocol identifier on Centrifuge networks is of the form `/centrifuge/<protocol-version>/<DID>`. Therefore, once a node has received a request and established a connection with a certain protocol identifier, for example `/centrifuge/v0.0.1/0x5571ecda2005a9B005014817607d02b4E1B7eAff`, it needs to:

1. Verify that it can handle version v0.0.1 of the Centrifuge Protocol or reject the request.
2. Verify that it has been associated to the account with DID `0x5571ecda2005a9B005014817607d02b4E1B7eAff` to handle or reject the request 7.2.
3. Handle the request.

The use of libp2p multiplexing for DID traffic segregation also offers the advantage of using separate ephemeral keys for encryption of DID-specific traffic.

7.4 Connection and Authentication Layers

There are two layers of authentication for establishing a connection between two nodes in the Centrifuge network. Following are the steps in the connection establishment flow.

1. Given a protocol action, look up relevant node ids to interact with. For example, given a document update, find the node ids of the collaborators using their DIDs (refer 7.2).
2. Run the libp2p handshake to establish an authenticated, encrypted channel between the two libp2p nodes (refer 7.1.3). This is authentication layer 1.
3. Now the receiving node needs to authenticate the senders DID (refer 3). This is the authentication layer 2.

7.5 Protocol Envelope

All requests and responses exchanged on the Centrifuge network must comply to the Google protobuf based `protocol envelope` format shown in figure 7. The `body` field of the envelope contains the serialized protocol message while the `header` field includes a set of header fields for the following purposes.

1. `network_identifier` identifies the Centrifuge network the sender node belongs to. The receiving node would validate its belonging to the same network or else reject the message.
2. `node_version` is the version of the node software that the message originating node is running. Again, the receiving node would reject the message if the received version is incompatible with the version of software that it is running.
3. `sender_id` is the sender's DID (see 4.1.2). Because any DID on the Centrifuge network must be associated with a node on the network using the `P2P Discovery Key`, the receiving node validates that originating node is, in fact, associated with the sender's DID on Ethereum. Otherwise, it rejects the request.
4. `type` is included in the message header such that message body bytes could be decoded into the correct type to be processed by the receiving node.

7.6 Generic P2P Message Envelope

Wire level message encoding for the Centrifuge protocol is performed by the P2P messenger utility[15], hereinafter referred to as the messenger. A generic P2P message envelope shown in figure 8, implemented using Google Protobufs, is used by the messenger to offer a common interface for all Centrifuge p2p network traffic.

When a protocol envelope based(see 7.5) request or a response needs to be sent to a peer node, the initiating node creates an instance of the generic message envelope and sets the serialized bytes of the

[15] `https://github.com/centrifuge/go-centrifuge/tree/develop/p2p/messenger`

Figure 7: Protocol envelope

```
message Envelope {
  Header header = 1;
  bytes body = 2;
}

// Header above is defined as
message Header {
  uint32 network_identifier = 1;
  string node_version = 2;
  bytes sender_id = 3;
  // Body message type
  string type = 4;
}
```

Figure 8: Generic P2P message envelope

```
message P2PEnvelope {
  bytes body = 2;
}
```

request or the response in to the body field. Then, the created envelope is passed on to the messenger. The messenger in turn serializes it to the protobuf encoding format[16], and writes the serialized bytes to the wire using a length delimited format shown in figure 9. The message length header shown is encoded using varint format[17].

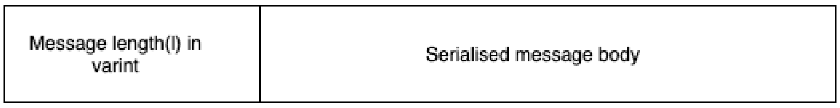


Figure 9: generic message envelope encoding

Maximum length(l) of a generic message envelope or the maximum size of a message in Centrifuge network is currently limited to 32MB.

8 User Mintable Non-Fungible Tokens

To allow for the integration of Centrifuge documents into other applications built on Ethereum, Centrifuge implements a smart contract library for minting non-fungible tokens (NFTs as specificed by ERC721). NFTs are a widely adopted standard to create objects that have trackable and unique ownership. Centrifuge extends this standard with methods that allow users to mint these NFTs based on the private off-chain data in a document. When invoking the mint method on an NFTRegistry, a set of fields of the document have to be revealed as Merkle proofs, using the precise-proofs library. This information can then be stored in the token registry. We call this data meta data. Semantically, an NFT can have any kind of meaning for the application using it. The Centrifuge protocol has some common NFT registries but also allows for deploying of custom registries. The contract library[18] provides the helpers to ensure that for one document only one token is minted in an NFT registry, that certain access rights are granted, and that the document is of a valid structure.

[16]https://developers.google.com/protocol-buffers/docs/encoding
[17]https://developers.google.com/protocol-buffers/docs/encoding#varints
[18]https://github.com/centrifuge/centrifuge-ethereum-contracts/blob/develop/contracts/erc721/UserMintableERC721.sol

8.1 Non-Fungible Token Contract

An NFT is a unique token that is always owned by one entity (an Ethereum account or contract). A token registry contract is a smart contract that manages a set of tokens.

Registry Contract is a smart contract implementing the logic to create and transfer tokens. We define it as $\mathsf{NFTRegistry}$

Registry Contract Address the Ethereum address of the $\mathsf{NFTRegistry}$

Token Identifier a $\mathbb{B}_{32}$ value defines the unique identifier of the token in the given registry contract, formally $T_{\texttt{id}}$

Owner a $\mathbb{B}_{20}$ value of the Ethereum account or contract that is the owner of the token, formally $T_{\texttt{owner}}$

Anchor a $\mathbb{B}_{32}$ value of the document identifier that the token is minted for, formally $T_{\texttt{anchor}}$

Metadata additional data that is supplied when the token is minted, formally $T_{\texttt{meta-data}}$

Constraints defines an array of constraints for specific document fields. It defines required values for the concrete document fields. For example, the an invoice status field to be *unpaid* formally $T_{\texttt{constraints}}$

8.2 Minting a Token

NFTs are created by calling the `mint` method of an NFT contract. The `mint` method requires the submission of a set of Merkle proofs for specific fields of the document for which the token has been minted. E.g. proofs for the invoice due date, amount, and currency in order to mint a token representing a payment obligation NFT. The Merkle proofs are validated by the NFT contract and checked against the document root provided by the Centrifuge. *AnchorRepository.* Any user who can produce valid proofs can mint the NFT. Together with the Merkle proof a second contract call is sometimes required. For example to proof a signature the Merkle proof verifies that a signature is part of a the document and a second call to an identity contract call would verify the signature itself.
The mint method can be defined as:

$$\mathsf{mint}(T_{\texttt{owner}}, T_{\texttt{id}}, T_{\texttt{meta-data}}, d_{\texttt{current}}, d_{\texttt{next}}, (P_0, \dots, P_n)) \xrightarrow{\text{ETH}} \mathsf{Identity} \xrightarrow{\text{ETH}} \mathsf{NFTRegistry} \tag{110}$$

A node doesn't call the mint method of the NFT Registry directly. The proxy functionality of the identity contract is used together with an action key $k_{\texttt{action}}$ mint stored inside the node.

The mint method requires a set of proofs $(P_{[0]}, ..., P_{[n]})$ to perform the Merkle proofs.

$$P = (P_{\texttt{field}}, P_{\texttt{sibling-hashes}}) \qquad P_{\texttt{field}} \in F \quad \wedge \quad P_{\texttt{sibling-hashes}} \in \mathbb{B}_{32}^n \tag{111}$$

8.2.1 Getting the Anchors

The mint method needs the anchor for $d_{\texttt{current}}$ to perform the Merkle proofs based on the document root.

$$\begin{aligned} \mathsf{NFTRegistry} &\xleftarrow{\text{ETH}} \mathsf{AnchorRepository} \\ A_{\texttt{anchor-current}} &\xleftarrow{\text{ETH}} \mathsf{getCommit}(d_{\texttt{current}}) \end{aligned} \tag{112}$$

$\mathsf{NFTRegistry}$ calls the $\mathsf{AnchorRepository}$ to receive the document root (anchor) of the document.

8.2.2 Individual Field Merkle Proof

Every NFT registry defines a set of mandatory fields on the document which need to be revealed. The mandatory fields depend on the purpose of the NFT and are defined by the initialization of the smart contract.

$$\mathcal{M}_{\texttt{proof}}(A_{\texttt{anchor-current}}, P_{[i]\texttt{field}}, P_{[i]\texttt{sibling-hashes}}) = 1 \tag{113}$$

Every mandatory field requires a proof to verify that it is part of the document. Some mandatory fields could require the $P_{[i]\texttt{field.value}}$ to have a concrete value.

$$P_{[i]\texttt{field.value}} = T_{\texttt{constraints[field.name]}} \tag{114}$$

The required values are defined in the contract itself and are formally noted as $T_{\texttt{constraints}}$. An example could be the status field of an invoice needs to be *unpaid* for a specific NFT registry. It is possible to implement more complex conditions in a registry contract. Like the value of a fields needs to equal one out of n pre defined values.

8.2.3 Latest Document

It should be only possible to mint an NFT out of the most recent document version. A proof can be supplied to the The $d_{\texttt{next}}$ defines the anchor id of the next version. The mint method receives $d_{\texttt{next}}$ as a parameter and can fetch the $A_{\texttt{anchor-next}}$ from the anchor registry.

$$\begin{aligned} \mathsf{NFTRegistry} &\xleftarrow{\text{ETH}} \mathsf{AnchorRepository} \\ A_{\texttt{anchor-next}} &\xleftarrow{\text{ETH}} \mathsf{getCommit}(d_{\texttt{next}}) \end{aligned} \tag{115}$$

The next anchor should not be set. This is enforced by checking that $d_{\texttt{next}}$ is part of document root and $A_{\texttt{anchor-next}}$ should not exist in the anchor registry.

$$\mathcal{M}_{\texttt{proof}}(A_{\texttt{anchor-current}}, P_{[\texttt{next}]\texttt{field}}, P_{[\texttt{next}]\texttt{sibling-hashes}}) = 1 \quad \wedge \quad A_{\texttt{anchor-next}} = \emptyset \tag{116}$$

The $A_{\texttt{anchor-current}}$ is the document root and $d_{\texttt{current}}$ is the id of the latest document version.

8.2.4 Verify an identity

A NFT registry should be called by an identity contract. It makes it easier to verify signatures later on. Only identity contracts created by the Centrifuge Identity factory are accepted. Every provided collaborator DID can be verified by checking if the factory contract created the collaborator's identity contract. First, it needs to be verified that the collaborators DID is part of the document.

$$\texttt{msg.sender} = I_{\texttt{DID}} \tag{117}$$

For the document collaborator who wants to mint the document the Ethereum `msg.sender` is equal to their DID. In a second step the identity factory is called to verify if it is an identity contract.

$$\begin{aligned} \mathcal{M}_{\texttt{proof}}(A_{\texttt{anchor-current}}, P_{[\texttt{collaborator.did}]\texttt{field}}, P_{[\texttt{collaborator.did}]\texttt{sibling-hashes}}) &= 1 \\ \mathsf{NFTRegistry} &\xleftarrow{\text{ETH}} \mathsf{IdentityFactory} \\ v &\xleftarrow{\text{ETH}} \mathsf{createdIdentity}(\mathsf{P}_{[\texttt{collaborator.did}]}) \\ v &= 1 \end{aligned} \tag{118}$$

8.2.5 Verify a Signature

Providing a signature of a collaborator as part of a NFT minting ensures that the collaborator signed the document state.

Ensure that the signature part of the document
First a Merkle proof verifies that a provided signature is part of the document.

$$\mathcal{M}_{\texttt{proof}}(A_{\texttt{anchor-current}}, P_{[\texttt{signature}]\texttt{field}}, P_{[\texttt{signature}]\texttt{sibling-hashes}}) = 1 \tag{119}$$

Collaborator part of the document
In a second Merkle proof the DID of the collaborator who signed the document needs to be part of the document collaborators.

$$\mathcal{M}_{\texttt{proof}}(A_{\texttt{anchor-current}}, P_{[\texttt{collaborator.did}]\texttt{field}}, P_{[\texttt{collaborator.did}]\texttt{sibling-hashes}}) = 1 \tag{120}$$

Signing root part of the document
A third Merkle proof verifies that the signing root $R_{\texttt{signing}}$ is part of the document.

$$\mathcal{M}_{\texttt{proof}}(A_{\texttt{anchor-current}}, P_{\texttt{[signing-root]field}}, P_{\texttt{[signing-root]sibling-hashes}}) = 1 \tag{121}$$

Collaborator identity verification
The DID of the collaborator needs to be an ERC725 identity contract created by the Identity factory.

$$\begin{aligned} \mathsf{NFTRegistry} &\xleftarrow{\text{ETH}} \mathsf{IdentityFactory} \\ v &\xleftarrow{\text{ETH}} \mathsf{createdIdentity}(\texttt{msg.sender}) \\ v &= 1 \end{aligned} \tag{122}$$

Verify the signature of the Collaborator
The NFTRegistry can recover the used address from the signature and the signed message. Afterwards the identity contract of the `msg.sender` is called to verify if the recovered address exists as a key and is not revoked.

$$\begin{aligned} k &= \texttt{recover}(P_{\texttt{[signing-root]field}}, P_{\texttt{[signature]field.value}}) \\ \mathsf{NFTRegistry} &\xleftarrow{\text{ETH}} \mathsf{Identity} \\ v &\xleftarrow{\text{ETH}} \mathsf{keyHasPurpose}(k, \texttt{sha256}(\text{"}\texttt{CENTRIFUGE@SIGNING}\text{"}))) \\ v &= 1 \\ r &\xleftarrow{\text{ETH}} \mathsf{keyIsRevoked}(k) \\ r &= 0 \end{aligned} \tag{123}$$

If the signature was signed with the purpose key for signing (recovered address) the signature is accepted as valid by the mint method.

8.2.6 Uniqueness Constraint

A document can have different NFTs minted. However an NFT registry can enforce that for a document d it only has one token minted. This is done by adding the token to $d_{\texttt{nfts}}$.

$$\begin{aligned} N &= (N_{\texttt{registry-id}}, N_{\texttt{token-id}}) \\ N_{\texttt{registry-id}} \in \mathbb{B}_{32} \quad &\wedge \quad N_{\texttt{token-id}} \in \mathbb{B}_{32} \\ d_{\texttt{nfts}} &= (N_{[0]}, ..., N_{[n]}) \end{aligned} \tag{124}$$

A collaborator who wants to mint an NFT first needs to create a new document version d' that contains the token id, $N_{\texttt{token-id}}$, and registry address, $N_{\texttt{registry-id}}$.

$$\begin{aligned} N_{\texttt{token-id}} &= \texttt{RAND(32)} \\ N_{\texttt{registry-id}} &= T_{\texttt{registry-address}} \quad || \quad \mathbb{B}_{12} \\ N &= (N_{\texttt{registry-id}}, N_{\texttt{token-id}}) \\ d' &= d \\ d'_{\texttt{nfts}} &= \mathsf{append}(d_{\texttt{nfts}}, N) \end{aligned} \tag{125}$$

The new document version d' needs to include the token id $T_{\texttt{id}}$ which is randomly chosen and committed to by the user. In the Protobuf definition of the document $d_{\texttt{nfts}}$ is implemented as a map where the key is 32 bytes including the 20 bytes of the Ethereum registry address and 12 bytes of zeros in the end. The user needs to request the signatures of the collaborators and commit the new document version as described in a previous chapter. (See chapter 5)

$$\begin{aligned} \mathcal{M}_{\texttt{proof}}(A_{\texttt{anchor-current}}, P_{\texttt{[nfts][registry-address]field}}, P_{\texttt{[nfts][registry-address]sibling-hashes}}) &= 1 \\ P_{\texttt{[nfts][registry-address]field.value}} &= T_{\texttt{id}} \end{aligned} \tag{126}$$

Inside the mint method a Merkle proofs verifies that the provided T_{id} is part of the document. A collaborator would not sign a new document version d' if the current one already includes a token id for the same NFT registry. Without the signatures of the collaborators it is not possible to mint a token for the same registry twice.

8.3 Token Methods

The ERC721 standard defines the interface for a NFT registry.

8.3.1 Transfer

The token smart contract has a function to get the owner of a token by the token identifier. This method is compliant with the ERC721 standard

$$T_{\texttt{owner}} \xleftarrow{\text{ETH}} \mathsf{ownerOf}(T_{\texttt{id}}) \tag{127}$$

A token can be transferred by the current owner to a different owner:

$$\mathsf{transferTo}(T_{\texttt{id}}, T_{\texttt{owner}}, T_{\texttt{new-owner}}) \xrightarrow{\text{ETH}} \mathsf{NFTRegistry} \tag{128}$$

8.3.2 Metadata

The ERC721 standards defines a specific field called metadata which should contain a url for further information about the NFT itself.

8.4 Document Access

Over the life time of an NFT it can be transferred to different users and companies. A person or a user who owns an NFT has the option to request read access to the document which belongs to the NFT. The token owner needs to have a node and a DID. It is not required that the owner is in the list of document collaborators $d_{\texttt{collaborators}}$. At the present time, there is no way of forcing the a node that has the document to make it available to an NFT owner.

$$\begin{gathered}(\texttt{DID}_{\texttt{sender}}, T_{\texttt{id}}, T_{\texttt{registry-address}}, d_{\texttt{id}}) \xleftarrow{\text{P2P}} \mathsf{receiveDocumentRequest}(\texttt{DID}_{\texttt{sender}}) \\ T_{\texttt{owner}} \xleftarrow{\text{ETH}} \mathsf{ownerOf}(T_{\texttt{id}}) \\ d = \mathsf{loadLatestVersion}(d_{\texttt{id}})\end{gathered} \tag{129}$$

If the following condition is true the $\texttt{DID}_{\texttt{sender}}$ is a NFT owner of the document.

$$d_{\texttt{nfts}[\texttt{T}_{\texttt{registry-address}}]} = T_{\texttt{id}} \quad \wedge \quad T_{\texttt{owner}} = \texttt{DID}_{\texttt{sender}} \tag{130}$$

The provided token id $T_{\texttt{id}}$ needs to be part of the document and the NFT owner in the registry needs to be the $\texttt{DID}_{\texttt{sender}}$. If both requirements are fulfilled the document will be send.

$$\mathsf{send}(d) \xrightarrow{\text{P2P}} \texttt{DID}_{\texttt{sender}} \tag{131}$$

9 Revisions

This document is a working document that will be updated as the Centrifuge Protocol evolves. It is hosted at `https://github.com/centrifuge/protocol`. Additions and changes are welcome.

9.1 Notable Revisions

1. 17.05.2019 ([caf6e80]): First public release of paper, covers Centrifuge Beta launch
2. 17.03.2019 (8c4357b): First draft version

References

[1] Eips/eip-725.md at master ethereum/eips. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-725.md. (Accessed on 02/27/2019).

[2] Jeremy Clark and Aleksander Essex. Commitcoin: Carbon dating commitments with bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 390–398. Springer, 2012. https://eprint.iacr.org/2011/677.pdf.

[3] Jeff Coleman, Liam Horne, and Li Xuanji. Counterfactual: Generalized state channels. https://l4.ventures/papers/statechannels.pdf, 2018. https://l4.ventures/papers/statechannels.pdf.

[4] Sven Heiberg, Ivo Kubjas, Janno Siim, and Jan Willemson. On trade-offs of applying block chains for electronic voting bulletin boards. *E-Vote-ID 2018*, page 259, 2018. https://eprint.iacr.org/2018/685.pdf.

[5] Joseph Poon. The bitcoin lightning network: Scalable off-chain instant payments. 2016. http://lightning.network/lightning-network-paper.pdf.

[6] Transaction processing - quorum wiki. https://github.com/jpmorganchase/quorum/wiki/Transaction-Processing/9637df4934995acbd81948e28501fcd5f9b7df83.

[7] Philip Stehlik et al. Privacy-enabled nfts: Self-mintable non-fungible tokens with private off-chain data. 2018. https://github.com/centrifuge/paper-privacy-enabled-nfts/releases/download/v1.01/paper-privacy-enabled-nfts.pdf.

[8] Peter Todd. Opentimestamps: Scalable, trust-minimized, distributed timestamping with bitcoin. 2016. https://petertodd.org/2016/opentimestamps-announcement.

[9] Fabian Vogelsteller. Erc: Lightweight identity issue #1056 ethereum/eips. https://github.com/ethereum/EIPs/issues/1056. (Accessed on 02/27/2019).

[10] Decentralized identifiers (dids) v0.11. https://w3c-ccg.github.io/did-spec/. (Accessed on 02/27/2019).

[11] rwot6-santabarbara/did-method-erc725.md at master weboftrustinfo/rwot6-santabarbara. https://github.com/WebOfTrustInfo/rwot6-santabarbara/blob/master/topics-and-advance-readings/DID-Method-erc725.md. (Accessed on 02/28/2019).