PRISM Slayer Protocol Specification

The Atala Team, IOHK

Revision: 6564b30
Revision Date: Thu, 7 Jul 2022 16:32:23 +0100

Contents

1 Intro **3**

2 Slayer v1: Bitcoin 2nd layer protocol **4**
2.1 Definitions . . . 4
2.2 Cryptography . . . 4
2.3 Differences from sidetree . . . 4
2.4 The path to publish an ATALA Operation . . . 4
2.5 Validation . . . 6
2.6 Updating ATALA Node state . . . 6
2.7 ATALA Operations . . . 7
2.7.1 CreateDID . . . 7
2.7.2 UpdateDIDOperation . . . 8
2.7.3 IssueCredential . . . 8
2.7.4 RevokeCredential . . . 9
2.8 Key management . . . 9
2.8.1 Issuer . . . 9
2.8.2 Holder . . . 10
2.8.3 Verifier . . . 10
2.9 More . . . 10
2.10 Attacks and countermeasures . . . 11
2.10.1 Late publish attack . . . 11
2.10.2 DoS attack . . . 11
2.10.3 Replay attack . . . 11
2.10.4 Hide-and-publish attack . . . 11

3 Slayer v2: Cardano 2nd layer protocol **12**
3.1 Definitions . . . 12
3.2 Protocol . . . 12
3.2.1 Objective . . . 12
3.2.2 Operation posting and validation . . . 13
3.2.3 Node state . . . 13
3.2.4 Operations . . . 14
3.2.5 Update DID . . . 15
3.2.6 IssueCredential . . . 17
3.2.7 Services . . . 18
3.3 Tentative schemas . . . 20
3.3.1 Verifiable Credential . . . 20
3.4 Notes . . . 20

4 Slayer v3: Scaling without a second layer **22**
4.1 Motivation . . . 22

4.2 Changes proposed . . . 23
4.2.1 Credential issuance . . . 23
4.2.2 Credential revocation . . . 24
4.2.3 DID creation . . . 25
4.2.4 DID updates . . . 26
4.3 Fee estimations . . . 26
4.3.1 Metadata usage concerns . . . 28
4.4 Formal changes . . . 28
4.5 Node State . . . 28
4.6 DID Creation . . . 29
4.7 DID batch Timestamping . . . 29
4.8 DID Update . . . 29
4.9 Credential Batch Issuance . . . 30
4.10 Batch Revocation . . . 31
4.11 Credentials Revocation . . . 32
4.12 Credential Verification . . . 33

5 Canonicalization (and comments on signing) 35
5.1 Our current state . . . 35
5.1.1 Signing atala operations . . . 35
5.1.2 Computing DID suffix . . . 36
5.1.3 Credential signing . . . 37
5.2 Comments on JSON . . . 37
5.2.1 JCS . . . 38
5.3 On JSON-LD, RDF and LD-PROOFS . . . 38
5.4 Comments on credentials' signing . . . 39
5.5 A note on future goals (selective disclosure) . . . 40

6 Key derivation and account recovery process 41
6.1 Context . . . 41
6.2 Definitions . . . 41
6.3 Generation process . . . 41
6.3.1 Root key generation . . . 41
6.3.2 Children key derivations . . . 42
6.3.3 The paths used in our protocol . . . 42
6.3.4 DID generation . . . 43
6.3.5 DID Document updates . . . 43
6.4 Account recovery . . . 44
6.4.1 DID recovery . . . 44
6.4.2 Key recovery . . . 44
6.4.3 Credentials recovery . . . 44
6.4.4 Test vectors . . . 45
6.5 Future challenges . . . 46

7 Unpublished DIDs 47
7.1 Objective . . . 47
7.2 Approaches reviewed . . . 47
7.2.1 Sidetree . . . 47
7.3 Proposal for first iteration . . . 48
7.3.1 Impact on recovery process . . . 48
7.3.2 DID length . . . 49

8 Late publication 51
8.1 The basic scenario . . . 51
8.2 Comparative - situation 2: Attack after key revocation . . . 52
8.3 Comparative - situation 3: Attack after and later credential and key revocation . . . 53

8.4 Some conclusions 55

9 Improvement proposals **56**
9.1 Make use of the underlying Cardano addresses 56
9.2 Light nodes and multi-chain identity 56
9.2.1 Bonus feature - multi chain DIDs 57
9.3 Layer 2 batching without late publication 57

1 Intro

This is a description of PRISM's 2nd layer protocol, internally called "Slayer". Historically, it has evolved in three variations (v1, v2, v3), which are provided here as the following three sections. Each variation has its own document, so we simply concatenate the three documents in order to create the current one. The original markdown source can be found in the repository.

In Slayer v1, the reference to Bitcoin is there because Bitcoin has been our initial target blockchain. Since then, we have transitioned to Cardano. Currently there is a tension between keeping Slayer's original blockchain-agnostic nature versus tying it to Cardano by leveraging its more flexible metadata feature.

The final sections of this document describe canonicalization, key derivation, unpublished DIDs, the *late publication* attack, and some more ideas on evolving the protocol.

2 Slayer v1: Bitcoin 2nd layer protocol

This document describes the protocol for building the credentials project as a 2nd layer on top of Bitcoin. The protocol is based on the sidetree protocol from Microsoft.

2.1 Definitions

The definitions from the high-level-design-document apply here, be sure to review those ones first.

- **DID**: A decentralized identifier, see the official spec.
- **DID Document**: The document representing a state of a DID, includes details like its public keys.
- **ATALA Operation**: An operation in the 2nd layer ledger (ATALA Node), like `Create DID`, `Issue Credential`, etc, look into the supported operations for more details. This is equivalent to a Bitcoin Transaction on the 2nd layer protocol.
- **ATALA Block**: A list of ATALA Operations, includes metadata about the block. This is equivalent to a Bitcoin Block on the 2nd layer protocol.
- **ATALA Object**: A file that has metadata about an ATALA Block and a reference to retrieve it. A reference to an ATALA Object is the only detail stored in Bitcoin operations.
- **Genesis Bitcoin Block**: The first Bitcoin Block aware of ATALA Objects, everything before this block can be discarded.
- **Content Addressable Storage**: System that allows storing files and querying them based on the hash of the file. We have yet to choose whether to use centralized one (e.g. S3), decentralized (IPFS) or some combination of these.

2.2 Cryptography

We use **hashing** to create short digests of potentially long data. For secure cryptographic hash algorithms there is an assumption that it is computationally intractable to find two values that have the same hash. This is why the hash is enough to uniquely identify the value. SHA256 algorithm is used, unless specified otherwise.

Cryptographic signatures are used to assure person who generated the message is who they claim they are. Keys are generated and managed by users - you can read more on that in Key management section. SHA256 with ECDSA is used unless specified otherwise.

2.3 Differences from sidetree

This section states the main known differences from the sidetree protocol:

- We support credentials while sidetree only supports DIDs.
- We are using Protobuf encoded data while sidetree uses JSON representation.
- We are using GRPC for API while sidetree uses REST.

2.4 The path to publish an ATALA Operation

There are some steps involved before publishing an ATALA Operation, the following steps show what the ATALA Node does to publish a credential proof, publishing any other ATALA Operation would follow the same process.

NOTE: We are using JSON protobuf encoding for readability here. Binary encoding is used for querying the node and storage.

1- Let's assume that issuer's DID is already registered on the ledger. In order to register an issued credential they need to send a signed ATALA operation containing its hash:

```
IssueCredentialOperation:
{
```

```
  "signedWith": "issuing",
  "signature": "MEUCIQDCntn4GKNBja9LYPHa5U7KSQPukQYwHD2FuxXmC2I2QQIgEdN3EtFZW+k/zOe2KQYjYZWPaV5SE
  "operation": {
    "issueCredential": {
      "credentialData": {
        "issuer": "7cd7b833ba072944ab6579da20706301ec6ab863992a41ae9d80d56d14559b39",
        "contentHash": "7XACtDnprIRfIjV9giusFERzD722AW0+yUMil7nsn3M="
      }
    }
  }
}
```


2- After some time the node creates an ATALA Block containing all operations it has received from clients. Here it contains just one, but generally nodes will batch operations in order to lower block publishing costs.


```
{
  "version": "0.1",
  "operations": [
    {
      "signedWith": "issuing",
      "signature": "MEUCIQDCntn4GKNBja9LYPHa5U7KSQPukQYwHD2FuxXmC2I2QQIgEdN3EtFZW+k/zOe2KQYjYZWPa
      "operation": {
        "issueCredential": {
          "credentialData": {
            "issuer": "7cd7b833ba072944ab6579da20706301ec6ab863992a41ae9d80d56d14559b39",
            "contentHash": "7XACtDnprIRfIjV9giusFERzD722AW0+yUMil7nsn3M="
          }
        }
      }
    }
  ]
}
```


3- An ATALA Object is created, it links the current ATALA Block. In the future we plan to add metadata which can help the ATALA Node to choose whether to retrieve the ATALA Block or not (think about a light client), such as list of ids of entities that the block affects. ATALA Block hash included is SHA256 hash of block file content - which consists of binary encoded `AtalaBlock` protobuf message.


```
{
  "blockHash": "s4Xy4+Cx4b1KaH1CHq4/kq9yzre3Uwk2A0SZmD1t7YQ=",
  "blockOperationCount": 1,
  "blockByteLength": 196
}
```


4- The ATALA Block and the ATALA Object are pushed to the Content Addressable Storage, each of these files are identified by the hash of its content, like `HASH(ATALA Object)` and `HASH(ATALA Block)` - the hash is computed from the file contents, binary encoded protobuf message.

5- A Bitcoin transaction is created and submitted to the Bitcoin network, it includes a reference to the ATALA Object on the special output (see the one with OP_RETURN), the `FFFF0000` is a magic value (to be defined) which tells ATALA Nodes that this transaction has an ATALA Object linked in it, after the magic value, you can see the `HASH(ATALA Object)` (SHA256 of the file).


```
{
  "txid": "0b6e7f92b27c70a6948dd144fe90387397c35a478f8b253ed9feef692677185e",
  "vin": [...],
  "vout": [
```

```
    {
      "value": 0,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_RETURN FFFF0000 015f510a36c137884c6f4527380a3fc57a32fdda79bfd18634ba9f793edb79
        "hex": "6a2258...",
        "type": "nulldata"
      }
    }
  ],
  "hex": "01000000...",
  "time": 1526723817
}
```


6- After the published Bitcoin transaction has N (to be defined) confirmations, we could consider the transaction as final (no rollback expected), which could mark all the underlying ATALA operations as final.

2.5 Validation

There are two notions related to operations: we say that an operation is **valid** if it is well-formed - has all required fields - and it is not larger than defined limits. Validity depends only on the operation itself and system parameters, not on its state. We say that operation is **correct** if it is valid, signed properly, and it is either creation operation or refers the previous operation correctly.

Operations that are not valid must be discarded by the node and not included in the block. Inclusion of even one invalid operation renders the whole block invalid and it must be ignored by the nodes. On the other hand operations correctness should not be checked before inclusion - the block can include incorrect operations. It is checked during state updates - incorrect operations are ignored, but one or more incorrect operations don't affect processing other operations in the block.

2.6 Updating ATALA Node state

The ATALA Node has an internal database where it indexes the content from the ATALA operations, this is useful to be able to query details efficiently, like retrieving a specific DID Document, or whether a credential was issued.

The ATALA Node needs to have a Genesis Bitcoin Block specified, everything before that block doesn't affect the ATALA Node State.

These are some details that the ATALA Node state holds:

- DIDs and their respective current document.
- Credentials (hashes only), who issued them and when they were revoked (if applies)

The component updating the ATALA Node state is called the Synchronizer.

The Synchronizer keeps listening for new blocks on the Bitcoin node, stores the Bitcoin block headers and applies rollbacks when necessary.

After block is considered finalized (getting N confirmations), we must look for ATALA Objects, retrieve the object from the storage, perform the validations (to be defined) on the ATALA Object, the ATALA Block, and the ATALA Operations, and apply the ATALA Operations to the current ATALA Node state. That means that the state is lagging N blocks behind what is present in the Bitcoin ledger - and all user queries are replied to basing on such lagging state.

2.7 ATALA Operations

Here you can find the possible ATALA Operations. The list will be updated when new operation get implemented.

Each operation sent via RPC to node needs to be signed by the client using relevant key (specified in the operation description) and wrapped into `SignedAtalaOperation` message. Signature is generated from byte sequence obtained by binary encoding of `AtalaOperation` message.

Operations must contain all fields, unless specified otherwise. If there is a value missing, the operation is considered invalid.

We can divide operations into two kinds: ones that create a new entity (e.g. CreateDID or IssueCredential) and ones that affect existing one (e.g. RevokeCredential). The latter always contain a field with previous operation hash (SHA256 of `AtalaOperation` binary encoding). If it doesn't match, operation is considered incorrect and it is ignored.

2.7.1 CreateDID

Registers DID into the ATALA ledger. DID structure here is very simple: it consists only of id and sequence of public keys with their ids. It must be signed by one of the master keys given in the document.


```
{
  "signedWith": "master",
  "signature": "MEQCIBZGvHHcSY7AVsds/HqfwPCiIqxHlsi1m59hsUWeNkh3AiAWvvAUeF8jFgKLyTt11RNOQmbR3SIPX.
  "operation": {
    "createDid": {
      "didData": {
        "publicKeys": [
          {
            "id": "master",
            "usage": "MASTER_KEY",
            "ecKeyData": {
              "curve": "P-256K",
              "x": "8GnNreb3fFyYYO+DdiYd2O9SKXXGHvy6Wt3z4IuRDTM=",
              "y": "O4uwqhI3JbY7W3+v+y3S8E2ydKSj9NXVOuS61MemOyO="
            }
          },
          {
            "id": "issuing",
            "usage": "ISSUING_KEY",
            "ecKeyData": {
              "curve": "P-256K",
              "x": "F8lkVEMP4pyXa+U/nE2Qp9iA/Z82Tq6WD2beuaMK2m4=",
              "y": "2hHElksDscwWYXZCx1pRyj9XaOHioYr48FPNRsUBAqY="
            }
          }
        ]
      }
    }
  }
}
```


RPC Response:


```
{
  "id": "7cd7b833ba072944ab6579da20706301ec6ab863992a41ae9d80d56d14559b39"
}
```

The returned identifier is hex encoding of the hash of the binary representation of `AtalaOperation` message. The DID can be obtained by prefixing the id with "did:atala:".

2.7.2 UpdateDIDOperation

Updates DID content by sequentially running update actions included. Actions available: **AddKeyAction** and **RemoveKeyAction**. Please note that key id must be unique - when a key is removed its id cannot be reused.

The operation must be signed by a master key. Actions cannot include removal of the key used to sign the operation - in such case the operation is considered invalid. That is to protect agains losing control of the DID - we assure that there is always one master key present that the user is able to sign data with. In order to replace the master key, the user has to issue first operation adding new master key and then another removing previous one. In order for the operation to be considered valid, all its actions need to be.


```
{
  "signedWith": "master",
  "signature": "MEQCIGtIUUVSsuRlRWwN6zMzaSi7FImvRRbjId7Fu/akOxFeAiAavOigmiJ5qQ2ORknhAEb207/2aNkQK
  "operation": {
    "updateDid": {
      "id": "7cd7b833ba072944ab6579da20706301ec6ab863992a41ae9d80d56d14559b39",
      "actions": [
        {
          "addKey": {
            "key": {
              "id": "issuing-new",
              "usage": "ISSUING_KEY",
              "ecKeyData": {
                "curve": "P-256K",
                "x": "Zk85VxZ1VTo2dxMeI9SCuqcNYHvW7mfyIPROD9PI9Ic=",
                "y": "QsI8QhEe4ZOYnG4kGZglvYfEPME5mjxmWIaaxsivz5g="
              }
            }
          }
        },
        {
          "removeKey": {
            "keyId": "issuing"
          }
        }
      ]
    }
  }
}
```


Response:

```
{}
```

2.7.3 IssueCredential

Register credential into the ATALA ledger, given hash of its contents. It must be signed by one of issuer's current issuing keys.

Example:


```
{
  "alg": "EC",
```

```
  "keyId": "issuing",
  "signature": "MEUCIQDCntn4GKNBja9LYPHa5U7KSQPukQYwHD2FuxXmC2I2QQIgEdN3EtFZW+k/zOe2KQYjYZWPaV5SE
  "operation": {
    "issueCredential": {
      "credentialData": {
        "issuer": "7cd7b833ba072944ab6579da20706301ec6ab863992a41ae9d80d56d14559b39",
        "contentHash": "7XACtDnprIRfIjV9giusFERzD722AW0+yUMil7nsn3M="
      }
    }
  }
}
```

RPC response:

```
{
  "id": "a3cacb2d9e51bdd40264b287db15b4121ddee84eafb8c3da545c88c1d99b94d4"
}
```

The returned identifier is hex encoding of the hash of the binary representation of `AtalaOperation` message. It is used to refer the credential in the revocation operation.

2.7.4 RevokeCredential

It must be signed by one of issuer's current issuing keys.

Example:

```
{
  "alg": "EC",
  "keyId": "issuing",
  "signature": "MEUCIQCbX9aHbFGeeexwT7IOA/n93XZblxFMaJrBpsXK99I3NwIgQgkrkXPr6ExyflwPMIH4Yb3skqBhh
  "operation": {
    "revokeCredential": {
      "previousOperationHash": "o8rLLZ5RvdQCZLKH2xW0Eh3e6E6vuMPaVFyIwdmblNQ=",
      "credentialId": "a3cacb2d9e51bdd40264b287db15b4121ddee84eafb8c3da545c88c1d99b94d4"
    }
  }
}
```

RPC response:

```
{}
```

2.8 Key management

As the users of the systems are the owners of their data, **they are responsible for storing their private keys securely**.

There are several details to consider, which depend on the user role.

NOTE: This section details the ideal key management features which can't be supported by December.

NOTE: The Alpha proposal doesn't include any key management.

2.8.1 Issuer

The issuer must have several keys, which are used for different purposes:

1. A **Master Key** which can revoke or generate any of the issuer keys, this should be difficult to access, it's expected to be used only on extreme situations. A way to make it difficult to access could be to split the key into N pieces which are shared with trusted people, in order

to recover the master key, any K of these pieces will be required. Compromising this key could cause lots of damage.
2. A **Issuing Key** which is used for issuing credentials, ideally, it will be stored in a hardware wallet, the Shamir's Secret Sharing schema could be adequate if issuing certificates is only done a couple of times per year. Compromising this key could lead to fake certificates with the correct signatures, or even valid certificates getting revoked (unless there is a separate key for revocation).
3. A **Communication Key**, which is used for the end-to-end encrypted communication between the issuer and the student's wallet, this can easily be a hot key, Compromising this key could not cause much damage.
4. An **Authentication Key**, which is used for authenticating the issuer with IOHK. Compromising this key could lead the attacker to get anything IOHK has about the issuer, like the historic credentials, and getting new connection codes. This key will ideally be stored in a hardware wallet, or an encrypted pen-drive.

Key storage:

- The issuer private keys are the most powerful because they can issue and revoke certificates, hence, if these keys get compromised, lots of people could be affected.
- The issuer ideally will handle their private keys using a hardware wallet (like ledger/trezor), but **this won't be supported by December**.
- Another way is to use an encrypted pen-drive to store the keys, but that should be done outside of our system.
- Shamir's Secret Sharing is the most common approach to do social key recovery, and while Trezor already supports it, **it can't be done by December**.

2.8.2 Holder

- While the holder private keys are very important, they are less important than the issuer keys, compromising a user key only affects himself.
- The mobile wallet generates a mnemonic recovery seed which the holders should store securely, the keys on the wallet are encrypted by a password entered by the holder after running the wallet for the first time.

2.8.3 Verifier

- The verifier keys have the less powerful keys, as they are only used to communicate with the holder wallet, getting them compromised doesn't allow the attacker to do much with them, they should be threaded with care but it is simple to recover if they get compromised.
- The verifier will ideally use an encrypted pen-drive to store their private keys but even a non-encrypted pen-drive may be enough.

2.9 More

- The 2nd layer could work like a permissioned ledger on top of a public ledger, we could ask the issuers (trusted actors) to provide us their bitcoin addresses and process operations only from trusted addresses. This approach is similar to what OBFT does, it just accepts node holding keys on the genesis whitelist.

- Trezor or Ledger could be easily used for signing the issuer requests if we sign SHA256 hashes only, let's say, before signing any payload, its SHA256 is computed and that's signed. While this could be handy for testing but it breaks the purpose of hardware wallets, which is that they'll keep your keys safe even if your computer is compromised cause the SHA256 will be displayed before signing requests but the device owner doesn't have any warranty that the SHA256 was produced from its expected payload.

2.10 Attacks and countermeasures

2.10.1 Late publish attack

Malicious issuer, running its own node, can create an Atala Block including key change operation and reference it in Bitcoin blockchain without actually making it available via the Content Addressable Storage. In following blocks they can fully publish operations signed with such key - such as credential issuance or further key changes. In the future they can make the block available, invalidating all the operations made after that.

We don't have a solution for that problem - but the protocol assumes some level of trust towards the issuer anyways. If they want to make a credential invalid they don't need to launch any attacks - they can just revoke it.

The name *Late publish attack* comes from https://medium.com/transmute-techtalk/sidetree-and-the-late-publish-attack-72e8e4e6bf53 blogpost, where it has been first described (according to our knowledge).

2.10.2 DoS attack

Attacker might try to disrupt the service by creating blocks with large amount of operations. To counteract that limits are imposed on the number of operations in the block.

Another way of slowing down the network is creating many late publishes, forcing frequent state recomputation. In order to avoid such situation, in the future ATALA Blockchain can be made permissioned, so only approved nodes can publish blocks.

2.10.3 Replay attack

Adversary might try to fetch an existing operation from Atala Block and send it again, e.g. re-attaching keys that were previously compromised and removed from the DID by its owner. Such strategy won't work in ATALA Blockchain, as each modifying operation contains hash of previous operation affecting the same entity.

2.10.4 Hide-and-publish attack

In this attack malicious node receives an operation from an issuer, includes it into an ATALA Block and publishes a reference to the generated object in the Bitcoin ledger, but doesn't publish the ATALA Block or Object. The issuer is unable to know if their operation has been included into the block.

In such case they should send the operation again to another node - and it will be published. If the malicious node publishes their block, the new operation will become invalid (as it refers to the hash before first attempt to submit the operation, which is no longer the latest one), but as it is identical to the former, they have the same hash, so following operations won't be affected.

On the other hand when the client doesn't re-publish the exact same transaction, but publish different one instead, for example updating credential timestamp, the situation is much worse. When the malicious node publishes the old transaction, the following new one becomes invalid. Moreover, it has different hash, so whole chain of operations possibly attached to it becomes invalid. In case of very late publishes, days or even months of operations might be lost.

To avoid such situation implementation of issuer tool needs to make sure that if an operation has been sent to any node, it won't create any new operation affecting this entity until the original one is published.

3 Slayer v2: Cardano 2nd layer protocol

This document describes the protocol for building the credentials project on top of Cardano. The main differences with version v0.1 of the protocol are:

- We now post Atala operations one at a time instead of posting them in batches
- DID Documents' underlying data is posted along with DIDs on the blockchain during the creation event

Those decisions allow this protocol to work in the open setting while avoiding attacks described in version 0.1. Another observation is that the content addressable storage now has no need to store public data (DID Documents).

3.1 Definitions

- **DID**: A decentralized identifier, see the official spec.
- **DID Document**: The document representing a state of a DID, includes details like its public keys
- **Verifiable Credential**: A signed digital document. E.g. digital university degrees, digital passport, etc.
- **Atala Node**: An application that follows this protocol. It sends transactions with metadata to the Cardano blockchain, read transactions confirmed by Cardano nodes, and interprets the relevant information from metadata in compliance with protocol rules
- **Atala Operation**: Metadata attached to a Cardano transaction, it is used to codify protocol events like `Create DID`, `Issue Credential`, etc, look into the supported operations for more details
- **Atala Transaction**: A Cardano transaction with an Atala Operation in its metadata
- **Genesis Atala Block**: The first Cardano Block that contains an Atala transaction, everything before this block can be discarded by Atala Nodes

Throughout this document, we will refer to DIDs, DID Documents and verifiable credentials as *entities*.
We will also user the term *Atala Event* to refer to the event of detecting a new Atala operation in the stable part of Cardano's blockchain. In particular, we will refer to events named after the operations e.g. DID creation event, credential revocation event, etc. Finally, we will say that a node *processes* an operation when it reads it from the blockchain and that it *posts* an operation when a client sends a request to it to publish it on the blockchain.

3.2 Protocol

3.2.1 Objective

The goal of the protocol is to read Atala operations from the stable section of Cardano's blockchain, and keep an updated state of DIDs, DID Documents and verifiable credentials. The protocol operations modify the state of the said entities. The current list of operations is: - **Create DID**: Allows to create and publish a new DID and its corresponding DID Document - **Update DID**: Allows to either add or revoke public keys to/from a DID Document - **Issue Credential**: Allows an issuer to publish a proof of issuance in Cardano's blockchain - **Revoke Credential**: Allows to revoke an already published verifiable credential

Along the operations, the protocol also provides services that any node must implement that do not affect the state. The list of services is: - **(DID resolution)** Given a published DID, return its corresponding DID Document in its latest state - **(Pre validation)** Given a credential's hash, a signer's DID, a key identifier, and an expiration date, reply if the credential is invalid with respect to signing dates and keys correspondence. The operation returns a key if the validations pass. The client needs to validate cryptographic signatures with the provided key. See credential validation section for more details.

Possible future operations - **Revoke DID**: Allows to mark a DID as invalid. Rendering all its keys invalid.

3.2.2 Operation posting and validation

Atala operations are encoded messages that represent state transitions. The messages' structure and protocol state will be described later in this document. When a node posts an operation, a Cardano transaction is created and submitted to the Cardano network. The transaction includes the encoded operation on its metadata. In Bitcoin we used to prepend the metadata messages with the string `FFFF0000`, a magic value (to be defined) which told Atala Nodes that this transaction has an Atala operation linked in it. We will consider doing the same in Cardano blockchain once we get more details on the metadata field structure.

After the published Cardano transaction has N (to be defined) confirmations, we could consider the transaction as final (no rollback expected), which could mark the underlying operation as final.

There are two definitions related to operations: - We say that an operation is **valid** if it is well-formed - has all required fields - and it is not larger than defined limits. Validity depends only on the operation itself and system parameters, not on its state. - We say that operation is **correct** if it is valid, signed properly, and it is either creation operation or refers the previous operation that affects the same entity.

Operations that are not valid must be discarded by the node and not included in a transaction. On the other hand operations correctness should not be checked before inclusion. It is checked during state updates - incorrect operations are ignored.

3.2.3 Node state

In order to describe the protocol, we will consider the following state that each node will represent. We will define the protocol operations and services in terms of how they affect/interact with this abstract state.

```
// Abstract types
type Key
type Hash
type Date

Concrete types
type KeyUsage       = MasterKey | IssuingKey | AuthenticationKey | CommunicationKey
type KeyData        = {
  key: Key,
  usage: KeyUsage,
  keyAdditiontionEvent: Date,      // extracted from the blockchain
  keyRevocationEvent: Option[Date] // extracted from the blockchain
}
type DIDData        = {
  lastOperationReference: Hash,
  keys: Map[keyId: String, keyData: KeyData]
}
type CredentialData = {
  lastOperationReference: Hash,
  issuerDIDSuffix: Hash,
  credIssuingEvent: Date,            // extracted from the blockchain
  credRevocationEvent: Option[Date]  // extracted from the blockchain
}

// Node state
```

```
state = {
  dids       : Map[didSuffix: Hash, data: DIDDocument],
  credentials: Map[credentialId: Hash, data: CredentialData]
}
```

We want to remark that the fields - `keyAdditionEvent` - `keyRevocationEvent` - `credIssuingEvent` - `credRevocationEvent` represent **timestamps inferred from Cardano's blockchain and not data provided by users**

The representation is implementation independent and applications can decide to optimise it with databases, specialized data structures, or others. We aim to be abstract enough for any implementation to be able to map itself to the abstract description.

Let's now move into the operations descriptions.

3.2.4 Operations

Users send operations via RPC to nodes. Each operation request needs to be signed by the relevant key (specified for each operation) and wrapped into `SignedAtalaOperation` message. Signature is generated from byte sequence obtained by binary encoding of `AtalaOperation` message.

Operations must contain the exact fields defined by schemas, unless specified otherwise. If there is an extra or missing value, the operation is considered to be invalid.

We can divide operations into two kinds: ones that create a new entity (e.g. CreateDID or IssueCredential) and ones that affect existing one (e.g. UpdateDID, RevokeCredential). The latter always contain a field with previous operation hash (SHA256 of `AtalaOperation` binary encoding). If the hash doesn't match with the last performed operation on the entity, the operation is considered incorrect and it is ignored.

When describing checks or effects of an operation on the node state, we will use an implicit value `decoded` that represents the decoded operation extracted from the blockchain.

3.2.4.1 Create DID Registers DID into the ledger. The associated initial DID Document structure here is very simple: it consists only of a document id (the DID) and sequence of public keys with their respective key ids. It must be signed by one of the master keys given in the DID Document.


```
{
  "signedWith": "master",
  "signature": "MEQCIBZGvHHcSY7AVsds/HqfwPCiIqxHlsi1m59hsUWeNkh3AiAWvvAUeF8jFgKLyTt11RNOQmbR3SIPX.
  "operation": {
    "createDid": {
      "didData": {
        "publicKeys": [
          {
            "id": "master",
            "usage": "MASTER_KEY",
            "ecKeyData": {
              "curve": "P-256K",
              "x": "8GnNreb3fFyYYO+DdiYd2O9SKXXGHvy6Wt3z4IuRDTM=",
              "y": "O4uwqhI3JbY7W3+v+y3S8E2ydKSj9NXVOuS61MemOyO="
            }
          },
          {
            "id": "issuing",
            "usage": "ISSUING_KEY",
            "ecKeyData": {
              "curve": "P-256K",
```

```
              "x": "F8lkVEMP4pyXa+U/nE2Qp9iA/Z82Tq6WD2beuaMK2m4=",
              "y": "2hHElksDscwWYXZCx1pRyj9XaOHioYr48FPNRsUBAqY="
            }
          }
        ]
      }
    }
  }
}
```


RPC Response:


```
{
  "id": "7cd7b833ba072944ab6579da20706301ec6ab863992a41ae9d80d56d14559b39"
}
```


The returned identifier is hex encoding of the hash of the binary representation of `AtalaOperation` message. The DID can be obtained by prefixing the id with "did:atala:".

When the operation is observed in the stable part of the ledger, the node performs the following checks:

```
alias didSuffix = getDIDSuffix(decoded)
alias referredKey: Option[Key] = extractKey(decoded)
alias messageSigned   = decoded.operation

 referredKey.nonEmpty &&
 referredKey.get.usage == MasterKey &&
 isValid(decoded.signature, messageSigned, referredKey.get) &&
 ! state.dids.contains(didSuffix)
```

where `extractKey` searches the key mentioned in `signedWith` from the decoded operation and return the key if found or an empty option otherwise.

If the check passes, then we get the following state update:

```
state'.dids = state.dids + (didSuffix -> { lastOperationReference = hash(decoded),
                                           keys = createMap(decoded.operation.didData.publicKeys)
state'.credentials = state.credentials
```

where `createMap` maps the data from the request to the model

3.2.5 Update DID

Updates DID content by sequentially running update actions included. Actions available: **AddKeyAction** and **RevokeKeyAction**. Please note that: - The key id must be unique - when a key is revoked its id cannot be reused. - A revoked key can't be re-added.

The operation must be signed by a master key. Actions cannot include revocation of the key used to sign the operation. In such case the operation is considered invalid. This is to protect against losing control of the DID. We assure that there is always one master key present that the user is able to sign data with. In order to replace the last master key, the user first has to add a new master key and then revoke the previous one in a separate operation signed by the newly added key. In order for the operation to be considered valid, all its actions need to be.


```
{
  "signedWith": "master",
  "signature": "MEQCIGtIUUVSsuRlRWwN6zMzaSi7FImvRRbjId7Fu/akOxFeAiAavOigmiJ5qQ2ORknhAEb207/2aNkQK
  "operation": {
    "previousOperationHash": "o8rLLZ5RvdQCZLKH2xW0Eh3e6E6vuMPaVFyIwdmblNQ=",
    "updateDid": {
```

```
      "didSuffix": "o5fHLw4RvdQCZLKH2xW0Eh3e6E6vuMPaVeGDIwdmblaD=",
      "actions": [
        {
          "addKey": {
            "key": {
              "id": "issuing-new",
              "usage": "ISSUING_KEY",
              "ecKeyData": {
                "curve": "P-256K",
                "x": "Zk85VxZ1VTo2dxMeI9SCuqcNYHvW7mfyIPR0D9PI9Ic=",
                "y": "QsI8QhEe4Z0YnG4kGZglvYfEPME5mjxmWIaaxsivz5g="
              }
            }
          }
        },
        {
          "revokeKey": {
            "keyId": "issuing"
          }
        }
      ]
    }
  }
}
```

Response:

```
{}
```

When the operation is observed in the stable part of the ledger, the node performs the following checks:

```
alias didToUpdate = decoded.operation.UpdateDID.didSuffix
alias signingKeyId = decoded.signedWith
alias updateActions = decoded.operation.updateDID.actions
alias currentDidData = state.dids(didToUpdate).keys
alias messageSigned   = decoded.operation

state.dids.contains(didToUpdate) &&
state.dids(didToUpdate).keys.contains(signingKeyId) &&
state.dids(didToUpdate).keys(signingKeyId).usage == MasterKey &&
state.dids(didToUpdate).lastOperationReference == decoded.operation.previousOperationHash &&
isValid(decoded.signature, messageSigned, state.dids(didToUpdate).keys(signingKeyId).key) &&
updateMap(signingKeyId, currentDidData, updateActions).nonEmpty
```

where `updateMap` applies the updates sequentially over the initial state. It verifies that the operation signing key is not revoked by any action, that each action can be performed correctly and returns the updated DidData if everything is fine. If any check or action application fails, an empty option is returned. We will refine the specification of `updateMap` in a future iteration.

If the check passes, then we get the following state update:

```
state'.dids = state.dids.update(didToUpdate, { lastOperationReference = hash(decoded),
                                               keys = updateMap(signingKeyId, currentDidData, upda
state'.credentials = state.credentials
```

3.2.6 IssueCredential

Publishes a proof of existence for a credential given hash of its contents. It must be signed by one of issuer's current issuing keys.

NOTES: - We are not enforcing that the signature of the RPC is the same as the credential's signature. Should we add this? It could be useful to keep flexibility and allow signing with different keys. For example, a university may have a DID and be the authority to issue credentials while the credential signers could be other institutions under the university control. The revocation control lives still under the sole control of the university. Note that depending on this decision we need to add or remove checks in the verification process. I am currently assuming that the keys could be different.

```
{
  "keyId": "issuing",
  "signature": "MEUCIQDCntn4GKNBja9LYPHa5U7KSQPukQYwHD2FuxXmC2I2QQIgEdN3EtFZW+k/zOe2KQYjYZWPaV5SE
  "operation": {
    "issueCredential": {
      "credentialData": {
        "issuerDIDSuffix": "7cd7b833ba072944ab6579da20706301ec6ab863992a41ae9d80d56d14559b39",
        "contentHash": "7XACtDnprIRfIjV9giusFERzD722AW0+yUMil7nsn3M="
      }
    }
  }
}
```

RPC response:

```
{
  "id": "a3cacb2d9e51bdd40264b287db15b4121ddee84eafb8c3da545c88c1d99b94d4"
}
```

The returned identifier is hex encoding of the hash of the binary representation of `AtalaOperation` message. It is used to refer the credential in the revocation operation.

When the operation is observed in the stable part of the ledger, the node performs the following checks:

```
alias signingKeyId    = decoded.keyId
alias issuerDIDSuffix = decoded.operation.issueCredential.credentialData.issuerDIDSuffix
alias signature       = decoded.signature
alias messageSigned   = decoded.operation

state.dids.contains(issuerDIDSuffix) &&
state.dids(issuerDIDSuffix).keys.contains(signingKeyId) &&
state.dids(issuerDIDSuffix).keys(signingKeyId).usage == IssuingKey &&
state.dids(issuerDIDSuffix).keys(signingKeyId).keyRevocationEvent.isEmpty &&
isValid(signature, messageSigned, state.dids(issuerDIDSuffix).keys(signingKeyId).key)
```

If the check passes, then we get the following state update:

```
alias credentialId = getCredId(decoded)

state'.dids = state.dids
state'.credentials = { state.credentials + (credentialId -> { lastOperationReference = hash(decode
                                                              issuerDIDSuffix        = issuerDIDSu
                                                              credIssuingEvent       = BLOCK_TIMES
                                                              credRevocationEvent    = None
                                                            })
}
```

3.2.6.1 RevokeCredential

It must be signed by one of issuer's current issuing keys.

Example:


```
{
  "keyId": "issuing",
  "signature": "MEUCIQCbX9aHbFGeeexwT7IOA/n93XZblxFMaJrBpsXK99I3NwIgQgkrkXPr6ExyflwPMIH4Yb3skqBhh
  "operation": {
    "revokeCredential": {
      "previousOperationHash": "o8rLLZ5RvdQCZLKH2xW0Eh3e6E6vuMPaVFyIwdmblNQ=",
      "credentialId": "a3cacb2d9e51bdd40264b287db15b4121ddee84eafb8c3da545c88c1d99b94d4"
    }
  }
}
```


When the operation is observed in the stable part of the ledger, the node performs the following checks:

```
alias signingKeyId    = decoded.keyId
alias credentialId    = decoded.operation.revokeCredential.credentialId
alias issuerDIDSuffix = state.credentials(credentialId).issuerDIDSuffix
alias signature       = decoded.signature
alias messageSigned   = decoded.operation
alias previousHash    = decoded.operation.revokeCredential.previousOperationHash

state.credentials.contains(credentialId) &&
state.credentials(credentialId).lastOperationReference == previousHash &&
state.dids.contains(issuerDIDSuffix) &&
state.dids(issuerDIDSuffix).keys.contains(signingKeyId) &&
state.dids(issuerDIDSuffix).keys(signingKeyId).usage == IssuingKey &&
state.dids(issuerDIDSuffix).keys(signingKeyId).keyRevocationEvent.isEmpty &&
isValid(signature, messageSigned, state.dids(issuerDIDSuffix).keys(signingKeyId).key)
```

If the check passes, then we get the following state update:

```
state'.dids = state.dids
state'.credentials = { state.credentials + (credentialId -> { lastOperationReference = hash(decode
                                                              issuerDIDSuffix        = state.crede
                                                              credIssuingEvent       = state.crede
                                                              credRevocationEvent    = Some(BLOCK
                                                            })
}
```

RPC response:

```
{}
```

3.2.7 Services

3.2.7.1 Verification In order to define credential verification we need to define what do we mean by a credential being valid. Here we have to distinguish between two types of validity. Given a credential, one notion of validity relates to the credential's specific semantic. E.g. a credential could represent a contract between two parties. In such case, our protocol has no say on telling if the contract signed is valid with respect to the legal system applied in a certain jurisdiction. However, the protocol probably should guarantee that the contract was published on a certain date, was signed by an specific issuer with a specific valid key, its expiration date (if any) hasn't occur, and guarantee that the credential's content was not altered.

We could define the former type of validity as *credential specific validity* and the later as *protocol validity*. Note that some credentials may not have any specific validations outside the protocol

ones. E.g. birth certificates, university degrees, national id documents, may only require the protocol validity. But other credentials may require additional ones on top of those.

For credential specific validations, we think that specific applications should be built on top of our protocol to fulfil specific use cases. The process described below will formalise the steps to verify protocol validity. Note that **all** credentials need to be valid according to the protocol independently to the existence of other credential specific validations. From now on, we will use the term valid referring to protocol valid.

Intuitively speaking, a credential is valid when an issuer signs it with a valid key and posts the Atala operation in the blockchain. The credential will remain valid as long as the credential's expiration date (if any) hasn't passed, and there must be no revocation operation posted in the blockchain referring to that credential. Otherwise the credential is considered invalid. The credential needs to have the following information to perform this checks:

```
- expirationDate: Option[Date]
- issuerDID: DID
- signature: Bytes
- signingKey: String // reference to a key in the issuers DID Document
```

The key is considered valid if it was added in the `issuerDID`'s DID Document before `credIssuingEvent` and, if the key is revoked in the said DID Document, the revocation event occurred after `credIssuingEvent` (i.e. after the credential was recorded as issued). As mentioned during the description of Issue Credential event, if we enforce that the key that signs the issuance request must be the same as the one that signs the credential, then we don't need to check that the credential was signed in proper dates because this will be guaranteed by the check performed in the IssueCredential operation. If we want to keep the flexibility of using different keys, we need to validate the relation between dates of key additions/revocations and credential issuance. The flexibility could be useful for an institution that allows other sub-institutions to sign credentials but only the main institution can publish the credentials to the blockchain. This same institution is the one used for revocation. An example of such relation could be faculties signing university degrees but only the university main administration is the one with the power to issue/revoke a credential. We should note that this could be simulated with a single DID by having different issuing keys for the faculties.

Expressed with respect to the node state, given a credential C:

```
// C has no expiration date or the expiration date hasn't occur yet, and
(C.expirationDate.isEmpty || C.expirationDate.get >= TODAY) &&
// the credential was posted in the chain, and
state.credentials.get(hash(C)).nonEmpty &&
// the credential was not revoked, and
state.credentials(hash(C)).data.credRevocationEvent.isEmpty &&
// the issuer DID that signed the credential is registered, and
state.dids.get(C.issuerDID.suffix).nonEmpty &&
// the key used to signed the credential is in the DID, and
state.dids(C.issuerDID.suffix).data.get(C.signingKey).nonEmpty
// the key was in the DID before the credential publication event, and
state.dids(C.issuerDID.suffix).data(C.signingKey).keyPublicationEvent < state.credentials(hash(C)
// the key was not revoked before credential publication event, and
(
  // either the key was never revoked
  state.dids(C.issuerDID.suffix).data(C.signingKey).keyRevocationEvent.isEmpty ||
  // or was revoked after signing the credential
  state.credentials(hash(C)).data.credPublicationEvent < state.dids(C.issuerDID.suffix).data(C.si
)
// the signature is valid
isValidSignature(
   C,
```

```
    state.dids(C.issuerDID.suffix).data(C.signingKey).key
)
```

3.3 Tentative schemas

3.3.1 Verifiable Credential

After reviewing the verification process we can propose this tentative generic credential schema. The schema could move around the fields, e.g. the signature field could be an object that contains the key reference.

```
{
  credentialName: String
  expirationDate: Option[Date]
  issuerDID: DID
  signature: Bytes
  signingKey: String // reference to a key in the issuers DID Document
  claim : Object // contains a mapping of key -> values that represent the credential specific da
}
```

We should note that this schema is not compliant with W3C standard drafts. It could be adapted on many parts.

However, the standard proposes a `credentialStatus` field for which documentation currently states:

```
credentialStatus
  The value of the credentialStatus property MUST include the:
  + id property, which MUST be a URL.
  + type property, which expresses the credential status type (also referred to as the credential
    expected that the value will provide enough information to determine the current status of the
    example, the object could contain a link to an external document noting whether or not the cre
    or revoked.
  The precise contents of the credential status information is determined by the specific credenti
  definition, and varies depending on factors such as whether it is simple to implement or if it
```

and later says

```
Defining the data model, formats, and protocols for status schemes are out of scope for this speci
Credential Extension Registry [VC-EXTENSION-REGISTRY] exists that contains available status scheme
want to implement verifiable credential status checking.
```

but the extension registry provides no relevant data.

It is unclear at the moment of this writing how to define this field. We may need to review more sections in the standard to clarify this point.

3.4 Notes

Below we raise points that may affect the protocol and schemas. We also note tasks related to the protocol that we should perform.

- Add a `RecoveryKey` value to the `KeyUsage` type. It could be useful to allow recovering the control over a DID even if the master key is lost/compromised. It should be non-revocable by any other key but itself (leaving an exception to our DIDUpdate operation).
- Allow to publish both batched operations or individual operations in transactions' metadata. Batching operations forces us the need to maintain a CAS which also brings possible problems related to files missing in a CAS. On the other hand, batching operations reduces fees. If IOHK restricts who can issue operations (in order to ease problems related to who add files into the CAS), then users could complain about centralization. We find then, as a reasonable approach, to allow any user to post an operation as long as it entire content is stored directly

in the transaction metadata. This allows for a batching service provided by trusted parties and also the option to maintain independence for those who prefer it.

- Allow issuers to not publish an issuance operation in the blockchain. During conversations we noted that the protocol always publishes a proof of existence on-chain along with the issuer DID. This allows for any actor to count how many credentials an issuer produces. This may be useful for some cases and we wonder if it could be a problem for others. If we remove the correlation of publication events with issuer's data, then the issuer looses the ability to detect if a credential is issued without authorization (e.g. due to a compromised key). If we want this correlation to be optional, we need to change the verification process and also define how issuers would specify which type of credentials they issue. Note that issuers need to specify somewhere that their credentials require a publication event to be valid, if not an attacker could simple issue credentials without this event and nodes wouldn't detect that the event is required. A possible place for such configuration is the issuer's DID itself, but we should analyse the limitations of such approach.
- Define an approach to privacy and share partial information from credentials. This is needed for compliance with W3C. This could also enable many interesting use cases.
- Define if we intend to be W3C standard, if so, update schemas and operations appropriately.
- Estimate bytes consumed by normal operation and estimate ADA fees.
- Define processes to establish trust on DIDs and how to map real world identities behind them.
- Define if we need credentials with multiple issuers.
- The W3C standard mentions `validFrom` and `validUntil` fields (where `validFrom` can be a date in the future). Decide if we want to update the verification process based on such fields and how to manage date verification (e.g. should we simply trust the issuer?).
- We can add other credential statuses. This is, instead of publishing a credential revocation event, we could post an update event that adds other statuses like `temporaly suspended` which can then transition back to `issued`. We should consider privacy implications of such update. E.g. a driver licence could keep forever a trace of times it has been suspended.
- We should define a way to inform verifiers more information about the credential status. This is akin to what the `credentialStatus` field attempts to do, which is basically to have a URL to check status and other information (like revocation reason). We should review if this field is mandatory or optional in the standard.
- If we decide to batch operations in files, we should consider how to order them. We could list first all DID related operations and then the credentials related ones. The motivation is that a user could send to a node multiple requests and expect the order to be preserved, e.g. first create a DID and then issue a sequence of credentials. If the order of events is inverted by the node, then we may end up rejecting the issuance of credentials and then creating a DID. An alternative to order events could be to provide a different endpoint for the node to receive batches of operations.

4 Slayer v3: Scaling without a second layer

4.1 Motivation

The goal of the protocol that we describe in this document, is to construct a scalable, decentralised and secure identity system. The system must provide the following features:

1. Allow the decentralised creation of self certifiable identifiers. This is, any person can create an identifier without the need of coordination with, or permission from any external authority. Only the creator (controller) of the identifier can prove his ownership through the use of cryptographic techniques.
2. Allow controllers to update the state of their identifiers.
3. Given an identifier, allow anyone to obtain its current state.
4. Allow asserting claims using the identifiers.

In particular, we will refer to these identifiers as **D**ecentralised **ID**entifiers (DIDs). There is a working group in W3C specifying the nature of these entities. For the purpose of this document, we will simplify their definition and declare that a DID is a string, which is associated to a document (DID Document). A DID Document declares the state of its associated DID. The first DID Document declares the *initial state* associated to a DID. This document contains cryptographic keys that allow controllers to prove their ownership over the DID, and to update the associated document. The DID Document can also contain other data, such as, URLs, referring to external information about the associated DID controller. The document can also be updated. In our construction, the identifier (DID) has the form `did:prism:hash_initial_DID_Document`, making our DIDs, self-certifiable. That is, given a DID and its initial DID Document, anyone could verify that the document is really associated to the identifier by comparing the identifier to the hash of the document.

In previous versions of our protocol, we followed the ideas behind Sidetree. Sidetree is a protocol developed by Microsoft, and is currently being specified in a Working Group inside the Decentralised Identity Foundation. In a simplified explanation, independent nodes run the Sidetree protocol using an underlying blockchain as base layer. Each node can post file references in the metadata (or equivalent) field the blockchain's transactions. The references point to files in a publicly accessible content addressable storage (CAS) service. The files contain the following events:

- Create DID: An event that declares the creation of an identifier, and declares its initial DID Document.
- Update DID: An event that allows adding/removing data to the associated DID Document.
- Deactivate DID: An event that declares that the DID is not usable anymore.

All events (except for DID creation ones), are signed by adequate keys to prove validity. The current state of a DID Document is computed by taking the initial state (provided on DID creation), and applying the events found in files referenced on the underlying blockchain following the order in which they appear.

With respect to claims, there is a variation of Sidetree that allows DID controllers to sign statements and post hashes of them as events in the protocol files (the ones referenced by blockchain transactions). This action, allows to timestamp the statement assertion while not revealing the statement content. The variation also allows the protocol to revoke an already asserted statement, also timestamping that event by adding it to a file.

The feature of `batching` events in files, and posting only references on-chain, allows a considerable scalability performance. However, we can observe some drawbacks related to this approach: 1. Data availability problems (referenced files may not be available). This leads to the inability to obtain consensus about the order in which events occurred in the past. Furthermore, it makes it difficult to know if certain event were ever valid at all. This allows a situation known as "late publication", that could affect certain use cases (e.g. DID transferability).

Late publication also represents a potential change to the "past" of the system state. 2. Even

though batching increases events throughput, it comes with the need of an anti spam strategy to avoid garbage events to saturate the system. This observation makes us think that the existence of batching does not necessarily imply lower operation costs or lower fees for users. 3. The problems related to data availability lead to not trivial implementation considerations.

This document explores some changes to the previous version of our protocol (0.2), in order to balance better performance (in terms of event throughput) while avoiding the issues related to data availability.

4.2 Changes proposed

4.2.1 Credential issuance

In version 0.2, we describe an issuance operation that, in a simplified way, contains:

```
IssueCredential(
  issuerDIDSuffix: ...,
  keyId: ...,
  credentialHash: ...,
  signature: ...
)
```

Following Sidetree's approach, one would construct a file, `F1`, of the form

```
IssueCredential(issuerDIDSuffix1, keyId1, credentialHash1, signature1)
IssueCredential(issuerDIDSuffix2, keyId2, credentialHash2, signature2)
...
IssueCredential(issuerDIDSuffixN, keyIdN, credentialHashN, signatureN)
```

and we would post a transaction containing the hash of `F1` on chain while storing `F1` in a CAS.

However, it is reasonable to believe that, an issuer, will produce many credentials in batches. Meaning that each issuer could create a file where all the operations will be signed by the same key. This would lead to a file of the form:

```
IssueCredential(issuerDIDSuffix, keyId, credentialHash1, signature1)
IssueCredential(issuerDIDSuffix, keyId, credentialHash2, signature2)
...
IssueCredential(issuerDIDSuffix, keyId, credentialHashN, signatureN)
```

Given that all the operations would be signed by the same key, one could replace the `N` signatures and occurrences of the `issuerDIDSuffix` and `keyId` by one, leading to a file with a single operation of the form:

```
IssueCredentials(issuerDIDSuffix, keyId,
   credentialHash1,
   credentialHash2,
   ...
   credentialHashN,
signature)
```

Now, at this point, one could ask, could we replace the list of hashes for something shorter? The answer is, yes.

An issuer could take the list of credential hashes `credentialHash1, credentialHash2, ..., credentialHashN` and compute a Merkle tree with them. Obtaining one root hash and `N` proofs of inclusion.

```
MerkleRoot,
(credentialHash1, proofOfInclusion1),
(credentialHash2, proofOfInclusion2),
```

```
...
(credentialHashN, proofOfInclusionN),
```

Now, the issuer could simply post this operation on the metadata of a transaction:

```
IssueCredentials(issuerDIDSuffix, keyId, merkleRootHash, signature)
```

and share with each corresponding holder, the pair `(credential, p)` where:

- `credential` is the credential to share and,
- `p` is the proof of inclusion that corresponds to `credential`

By doing this, a holder could later share a credential to a verifier along with the proof of inclusion. The verifier would perform a modified version of the verification steps from version 0.2 of the protocol. See last section for the formal description.

A remaining question is, why would we want each credential operation separate in the first place? The answer is that, our intention is for the issuer to be able to detect if an unauthorized credential is issued. So, does this change affect that goal? Our argument is that, it does not. In order to detect that a credential was issued without authorization, the issuer needed to check each `IssueCredential` operation posted and check for those signed by his keys, then compare those to the ones in his database and make a decision. With the proposed change, the issuer would perform the same steps, he would check all the merkle tree hashes posted with his signature and compare if that hash is registered in his database.

In conclusion, this change could allow us to remove the need of external files for issuing credentials and allow credential issuance to be fully performed on-chain using Cardano's metadata.

4.2.2 Credential revocation

The proposed protocol changes that we will describe below, do not scale the throughput of credential revocations as much as the increase we added for issuance. However, we could argue that credential revocation should be a type of event with low throughput demand in comparison.

If we analyse revocation scenarios, we could see some special cases:

- The issuer detected a merkle root that is not authorised. In such case, he could revoke the full represented batch with a small operation of the form:

  ```
  RevokeCredentials(
    issuanceOperationHash,
    keyId,
    signature
  )
  ```

  The associated DID is the one from the referred `IssueCredential` operation.

- Another situation is when an issuer would like to revoke specific credentials *issued in the same batch*. For this case, we could see the use of an operation:

  ```
  RevokeCredentials(
     issuanceOperationHash,
     keyId,
     credentialHash1,
     credentialHash2,
      ...,
     credentialHashK,
     signature
  )
  ```

Each hash adds approximately 32 bytes of metadata, which adds very little fee. See fee for more details on fees.

Now, if the issuer needs to revoke credentials issued in different batches, he could post independent transactions per batch. Compared to version 0.2 of this protocol, we can revoke batches of credentials with a single operation in certain cases. Before, we needed one operation per credential to be revoked.

We explored the idea of having an RSA accumulator per credential type. Our preliminary research makes us believe that the approach may require the use of big prime numbers, and the accumulator may occupy a few kilobytes of space. We decided to leave out the idea for a future version and request the suggestions from a cryptographer.

4.2.3 DID creation

In order to create a DID, in version 0.2, we have to post a `CreateDID` operation on-chain. Sidetree allows to scale the amount of DIDs to create by the use of batches. However, they also provide a long-form that allows to use a DID before it is published. The long form also allows to never publish a DID if it is never updated.

We propose, for this protocol version, to leave DID creation operations as optional, just for the purpose of timestamping. This is:

- The initial DID could be posted on-chain as in version 0.2, or
- It could have a long format that describes its initial state and no event will be published. The format could, informally be thought as

  `did:prism:hash_initial_DID_Document?initialState=InitialDIDDocument`

  The initial DID document can be validated by the DID suffix.
- Another alternative is that many DIDs created by the same entity, could construct a Merkle tree (as we propose for credential hashes) and post the hash of the root on-chain for later timestamping proving.

In Sidetree's slack, we received the feedback that they see this as an evolution they plan for the protocol with some differences from our proposal.

- They do not care about batched creation timestamping as we propose.
- During the first update operation, they would like to post the initial DID state along with the update operation. They prefer to store the initial state in the CAS rather than sharing a longer identifier on every interaction. We could leave this as two variations of the first update operation in our protocol, but there are considerations to have.
  1. Not publishing the initial state leads to smaller metadata to add to the first update transaction. It also allows more privacy. One drawback is that invalid update operations would not be "prunable" until the associated DID is resolved for the first time. This could be mitigated by associating DIDs to addresses as mentioned in other ideas.
  2. In the variation where the initial state is posted along with the first update, we would have bigger metadata use. We should be careful to not exceed reasonable metadata size. See metadata considerations for more comments on metadata usage.

For simplicity, we incline for option 2. As it would require fewer changes in the way we process operations. Currently, when the node finds a new operation, it applies the state change right away. If we do not post the initial DID state, we would need to process operations differently, as DID update operations would have no initial state to be applied upon. Similarly, we need to have the DID state at the time of credential issuance and revocation. We should allow to publish a DID during the first issuance operation too. Note that we won't need to publish a DID during the first revocation, because the first revocation must occur after an issuance event. Hence, we will request the signing DID to already be published by this point.

In practice, we have implemented the above two cases through "on-chain batching", meaning that we allow users to publish a sequence of events in a single blockchain transaction. In order to publish a DID during its first update, the user can post the `CreateDID` event and the `UpdateDID`

events in the same underlying transaction. Similarly, the user can also publish a `CreateDID` event along with an event to issue a batch of credentials. Any other combination of events can be published too, the only restriction is for the entire sequence to be small enough too fit in a single transaction metadata field.

The use of a long format would enable us to create DIDs without the need of batching or any on-chain event whatsoever. This leads to unbound throughput for DID creation.

NOTE: Even though there does not seem to be a maximum value for DID length, we should be aware of such discussions in DID core or similar groups. We have been warned that QR codes would not be good for DIDs with "big" long form. we could evaluate compressing the data inside the QR and decompressing it at the receiving end.

4.2.4 DID updates

This is the only operation that cannot be scaled on-chain in an easy way. The main reason is that update operations are expected to be publicly available. This implies that a commitment (e.g. hash, merkle tree, RSA accumulator, etc.) is not enough for the users of the protocol. Nodes need the actual update data.

A question we may have is, how often do we expect an update to occur? If this is not a recurrent operation, we should consider leaving it as an on-chain operation.

Alternatively, we could consider adding a permissioned batching service. This is: - IOHK (and selected actors) could have a DIDs that can sign authorised batches. - Some drawbacks are the complexities added by a missing trust model on who can batch update events, and also complexities to handle data availability problems.

Originally, we were thinking about having permissioned batches for every type of operation. In the worst case scenario, it now seems that we would only require batches for DID updates. We should keep evaluating if this is needed for this version.

Another option could be to allow "on-chain batching". Note that we have a bit less than 16Kb for transaction metadata. Consider that updating a DID consists on:

- Adding a key: requires a keyId and a key.
- Removing a key: requires a keyId.
- Adding/removing a service endpoint: an id and the endpoint when adding it.

If we imagine small update operations we could allow users to cooperate and batch up updates in a single ADA transaction.

- Actors could take periodic turns to distribute fee costs.
- We could ask a cryptographer if there is any signature aggregation scheme that could help to reduce metadata size.
- We would like to refer the reader to the metadata considerations section to be aware of potential problems with "on-chain" batching.

4.3 Fee estimations

According to Shelley documentation, there is a maximum transaction size. The documentation also states the following about fees:

> The basic transaction fee covers the cost of processing and storage. The formula is
>
> `a + bx`
>
> With constants `a` and `b`, and `x` as the transaction size in bytes.

The constants, according to Shelley parameters are:

```
"maxTxSize": 16384,
"maxBlockBodySize": 65536,
```

```
"maxBlockHeaderSize": 1100,
"minFeeA": 44,
"minFeeB": 155381,
```

Interestingly enough, the values `a` and `b` are inverted (a typo in the documentation already reported and confirmed). From the above data, the maximum transaction fee that could be created is

```
maxFee = minFeeB + maxTxSize*minFeeA = 155381 + 16384*44 = 876277 lovelace = 0.876277 ADA
```

which represents a 16 kilobyte transaction.

Let us estimate fees per operation type. We will add extra bytes in our estimations due to the metadata scheme enforced by Cardano.

We will assume:

- A base transaction size (i.e. without the metadata) of 250 bytes.
- A signature size of 75 bytes
- A hash size of 32 bytes.
- A keyId size of 18 bytes.
- A key size of 32 bytes.
- A DID suffix of 32 bytes.

Given the above, we could estimate:

- DID creation without publishing would have cost of 0 ADA and uses no metadata nor transactions.
- The old DID creation (publishing the operation) with two keys (one Master and one for Issuing) would have:
  - At least two key ids (2 x 18)
  - At least two keys (2 x 32)
  - We currently also have a signature (75) Adding the base transaction, we could overestimate this with a 400 bytes transaction, leading to: `scala minFeeB + 400*minFeeA = 155381 + 400*44 = 172981 lovelace = 0.172981 ADA`
- DID "batched" timestamping would have an operation identifier and a Merkle root hash. We could overestimate this with a 300 bytes transaction.

  ```
  minFeeB + 300*minFeeA = 155381 + 300*44 = 168581 lovelace = 0.168581 ADA
  ```
- If we imagine a credential issuance operation (where the issuing DID was already published), we have:
  - an issuer DID suffix (32)
  - a signature (75)
  - a keyId (18)
  - a Merkle root hash (32) Overestimating, this leads to 450 bytes of metadata, leading to an issuance fee of:

  ```
  minFeeB + 450*minFeeA = 155381 + 450*44 = 175181 lovelace = 0.175181 ADA
  ```

  If we need to consider the data to publish the DID during the issuance operation, we could add:
  - 2 key ids (2 x 18)
  - 2 keys (2 x 32) Meaning that we add 100 bytes, leading to:

  ```
  minFeeB + 550*minFeeA = 155381 + 550*44 = 179581 lovelace = 0.179581 ADA
  ```
- For revocation, we could analyse the cost per type of revocation.
  - If the issuer revokes the full batch of credentials, it looks reasonable to estimate a similar cost that the issuance operation, i.e. ~0.175181 ADA.

  - If the issuer needs to revoke selective credentials from a single batch, we could expect a fee similar to the one before with an addition of 2000 lovelace (32 bytes * 44 lovelace/byte + encoding bytes) per credential hash. This represents 0.0002 extra ADA per revoked credential.

- DID updates would require a more variable estimation. Sidetree suggests a maximum of 1 kilobyte for the size of an update operation. In comparison, this would represent an approximate of:

  ```
  minFeeB + 1024*minFeeA = 155381 + 1024*44 = 200437 lovelace = 0.200437 ADA
  ```

We could apply some optimizations in exchange for making a slightly more complex protocol. See related work.

4.3.1 Metadata usage concerns

Even though ledger rules allow for transactions with big metadata, we should be aware that this is not a guarantee that the network will accept them. For example, Bitcoin transactions with more than one `OP_RETURN` output are valid according to consensus rules, however, they are not considered "standard" transactions and nodes do not tend to forward them 1. 2. We raise this observation to motivate the most efficient use of metadata bytes. It may be reasonable to adopt a conservative design principle of "the smaller, the better".

4.4 Formal changes

In this section we would like to summarise the protocol with respect to a more formal definition of the operations and node state.

4.5 Node State

Given that we will now batch credential issuance and also optionally timestamp DID creation batches, we will update our node state definition. Let us start with type definitions:

```
// Abstract types
type Key
type Hash
type Date

// Concrete types
type KeyUsage       = MasterKey | IssuingKey | AuthenticationKey | CommunicationKey
type KeyData        = {
  key: Key,
  usage: KeyUsage,
  keyAdditionEvent: Date,      // extracted from the blockchain
  keyRevocationEvent: Option[Date] // extracted from the blockchain
}

type DIDData        = {
  lastOperationReference: Hash,
  keys: Map[keyId: String, keyData: KeyData]
}

type CredentialBatch = {
  issuerDIDSuffix: Hash,
  merkleRoot: Hash,
  batchIssuingEvent: Date,            // extracted from the blockchain
  batchRevocationEvent: Option[Date]  // extracted from the blockchain
}
```

```
// Node state
type State = {
  didTimestamps     : Map[didBatchId: Hash, timestamp: Date]
  publishedDids     : Map[didSuffix: Hash, data: DIDDocument],
  credentialBatches : Map[credentialBatchId: Hash, data: CredentialBatch]
  revokedCredentials: Map[credentialBatchId: Hash, Map[credentialHash: Hash, credentialRevocationI
}
```

Given the above, we define the node initial state as:

```
state = {
  didTimestamps      = Map.empty,
  publishedDids      = Map.empty,
  credentialBatches  = Map.empty,
  revokedCredentials = Map.empty
}
```

We will add an additional value called `BeginingOfTime` which will be used to represent timestamps for keys that belong to unpublished DIDs.

4.6 DID Creation

The DID creation process from v0.2 remains supported and unchanged. It will run the same validations and only update the state of `publishedDids`.

4.7 DID batch Timestamping

A difference with other operations is that to timestamp a batch of DIDs, we will not have a signature involved. Recall that on DID creation we ask for a Master key signature. So, given an operation:

```
{
  "operation": {
    "timestampDIDBatch" : {
      "markleRoot" : ...
    }
  }
}
```

which posts a Merkle tree root hash, we update the state as follows:

```
state'.didTimestamps      = state.didTimestamps + { operation.timestampDIDBatch.markleRoot -> TX_T
state'.publishedDids      = state.publishedDids
state'.credentialBatches  = state.credentialBatches
state'.revokedCredentials = state.revokedCredentials
```

4.8 DID Update

Given that we implemented on-chain batching, we do not need to update this operation. For completeness, we will describe the formal specification of this event.

```
{
  "signedWith": KEY_ID,
  "signature": SIGNATURE,
  "operation": {
    "previousOperationHash": HASH,
    "updateDid": {
      "didSuffix": DID_SUFFIX,
      "actions": [
```

```
        ADD_KEY_ACTION | REMOVE_KEY_ACTION,
        ...
      ]
    }
  }
}
```

When the operation is observed in the stable part of the ledger, the node will act as before, this is:

```
  alias didToUpdate   = decoded.operation.updateDid.didSuffix
  alias signingKeyId  = decoded.signedWith
  alias updateActions = decoded.operation.updateDID.actions
  alias messageSigned = decoded.operation

  state.publishedDids.contains(didToUpdate) &&
  state.publishedDids(didToUpdate).keys.contains(signingKeyId) &&
  state.publishedDids(didToUpdate).keys(signingKeyId).usage == MasterKey &&
  state.publishedDids(didToUpdate).lastOperationReference == decoded.operation.previousOperationHa
  isValid(decoded.signature, messageSigned, state.publishedDids(didToUpdate).keys(signingKeyId).ke
  updateMap(signingKeyId, state.publishedDids(didToUpdate).keys, updateActions).nonEmpty
```

where `updateMap` applies the updates sequentially over the initial state. It verifies that the operation signing key is not revoked by any action, that each action can be performed correctly and returns the updated `DidData` if everything is fine. If any check or action application fails, an empty option is returned. We will refine the specification of `updateMap` in a future iteration.

If the check passes, then we get the following state update: `scala state'.publishedDids = state.publishedDids.update(didToUpdate, { lastOperationReference = hash(decoded), keys = updateMap(signingKeyId, state.publishedDids(didToUpdate).keys, updateActions).get } state'.didTimestamps = state.didTimestamps state'.credentialBatches = state.credentialBatches state'.revokedCredentials = state.revokedCredentials`

4.9 Credential Batch Issuance

Now, similar to the situation with the first update operation, we find ourselves with an operation that will be signed by a key referenced by a DID. We said that we implemented on-chain batching, so we do not need to embed the DID creation event in this operation.

The old credential issuance operation can be described in the following way:

```
{
  "keyId": KEY_ID,
  "signature": SIGNATURE,
  "operation": {
    "issueCredentialBatch": {
      "batchData": {
        "issuerDIDSuffix": DID_SUFFIX,
        "merkleRoot": MERKLE_TREE_ROOT
      }
    }
  }
}
```

The response is now a `batchId` that represents the batch analogous to the old `credentialId`:

```
{
  "batchId" : OPERATION_HASH
}
```

Its implementation will be the hash of the `batchData` field.

The node state is now updated as follows:

```scala
alias signingKeyId    = decoded.keyId
alias issuerDIDSuffix = decoded.operation.issueCredential.batchData.issuerDIDSuffix
alias signature       = decoded.signature
alias messageSigned   = decoded.operation

state.dids.contains(issuerDIDSuffix) &&
state.dids(issuerDIDSuffix).keys.contains(signingKeyId) &&
state.dids(issuerDIDSuffix).keys(signingKeyId).usage == IssuingKey &&
state.dids(issuerDIDSuffix).keys(signingKeyId).keyRevocationEvent.isEmpty &&
isValid(signature, messageSigned, state.dids(issuerDIDSuffix).keys(signingKeyId).key)
```

The state would be updated as follows:

```
 alias batchId = computeBatchId(decoded)
 alias merkleRoot = decoded.operation.issueCredential.batchData.merkleRoot
 alias issuerDIDSuffix = decoded.operation.issueCredential.batchData.issuerDIDSuffix

 state'.publishedDids = state.publishedDids
 state'.didTimestamps = state.didTimestamps
 state'.credentialBatches  = state.credentialBatches + { batchId -> {
                                                           merkleRoot = merkleRoot,
                                                           issuerDIDSuffix = issuerDIDS
                                                           batchIssuingEvent    = LEDGI
                                                           batchRevocationEvent = None
                                                         }
                                                       }
 state'.revokedCredentials = state.revokedCredentials
```

4.10 Batch Revocation

We now define the revocation for a batch of issued credentials.

```
{
  "keyId": KEY_ID,
  "signature":: SIGNATURE,
  "operation": {
    "revokeBatch": {
      "batchId": HASH
    }
  }
}
```

The preconditions to apply the operation are:

```
alias batchId          = decoded.operation.revokeBatch.batchId
alias signature        = decoded.signature
alias signingKeyId     = decoded.keyId
alias messageSigned    = decoded.operation
alias issuerDIDSuffix  = state.credentialBatches(batchId).issuerDIDSuffix

state.credentialBatches.contains(batchId) &&
state.credentialBatches(batchId).batchRevocationEvent.isEmpty &&
state.publishedDids.contains(issuerDIDSuffix) &&
```

```
state.publishedDids(issuerDIDSuffix).keys.contains(signingKeyId) &&
state.publishedDids(issuerDIDSuffix).keys(signingKeyId).usage == IssuingKey &&
state.publishedDids(issuerDIDSuffix).keys(signingKeyId).keyRevocationEvent.isEmpty &&
isValid(signature, messageSigned, state.publishedDids(issuerDIDSuffix).keys(signingKeyId).key)
```

If the precondition holds, we update the node state as follows:

```
alias batchId = decoded.operation.revokeBatch.batchId
alias initialBatchState = state.credentialBatches(batchId)

state'.publishedDids = state.publishedDids
state'.didTimestamps = state.didTimestamps
state'.revokedCredentials = state.revokedCredentials
state'.credentialBatches = state.credentialBatches + { batchId -> {
                                                          issuerDIDSuffix = initialBat
                                                          merkleRoot = initialBatchSta
                                                          batchIssuingEvent = initialBa
                                                          batchRevocationEvent = Some(
                                                      }
                                                  }
```

4.11 Credentials Revocation

We are finally on the credential revocation operation. We will update the operation from version 0.2 to now allow many credential hashes to revoke.

```
{
  "keyId": KEY_ID,
  "signature":: SIGNATURE,
  "operation": {
    "revokeCredentials": {
      "batchId": HASH,
      "credentialHashes": [ HASH, ...],
    }
  }
}
```

The preconditions to apply the operation are:

```
alias batchId          = decoded.operation.revokeCredentials.batchId
alias signature        = decoded.signature
alias signingKeyId     = decoded.keyId
alias messageSigned    = decoded.operation
alias issuerDIDSuffix  = state.credentialBatches(batchId).issuerDIDSuffix
alias credentialHashes = decoded.operation.revokeCredentials.credentialHashes

state.credentialBatches.contains(batchId) &&
// the batch was not already revoked
state.credentialBatches(batchId).batchRevocationEvent.isEmpty &&
state.publishedDids.contains(issuerDIDSuffix) &&
state.publishedDids(issuerDIDSuffix).keys.contains(signingKeyId) &&
state.publishedDids(issuerDIDSuffix).keys(signingKeyId).usage == IssuingKey &&
state.publishedDids(issuerDIDSuffix).keys(signingKeyId).keyRevocationEvent.isEmpty &&
isValid(signature, messageSigned, state.publishedDids(issuerDIDSuffix).keys(signingKeyId).key)
```

If the above holds, then:

```
alias batchId          = decoded.operation.revokeCredential.batchId
// we will only update the state for the credentials not already revoked
```

```
alias credentialHashes = filterNotAlreadyRevoked(decoded.operation.revokeCredential.credentialHash
alias issuerDIDSuffix  = state.credentialBatches(batchId).issuerDIDSuffix

state'.publishedDids = state.publishedDids
state'.didTimestamps = state.didTimestamps
state'.credentialBatches  = state.credentialBatches

state'.revokedCredentials =
    state.revokedCredentials + { batchId -> state.revokedCredentials(batchId) + buildMap(credentia
                               }
```

where `buildMap` takes the credential hashes and computes a map from the credential hashes to their revocation ledger time.

Note: The reader may realise that an issuer could mark as "revoked" credentials that are actually not part of the referred batch. This is because there is no check that the credential hashes shared are contained in the batch. The reason why we didn't add this check, is that merkle proof of inclusion for many credentials are big in size. We argue, however, that the verification process will take a credential and check for revocation *in its corresponding* batch, meaning that no issuer will be able to revoke other issuers credentials (or even a credential outside of its issuance batch).

4.12 Credential Verification

In order to describe credential verification, we assume the following data to be present in the credential.

```
{
  "issuerDID" : DiD,
  "signature" : SIGNATURE,
  "keyID" : KEY_ID,
  ...
}
```

We also assume the presence of a Merkle proof of inclusion, `merkleProof` that attests that the credential is contained in its corresponding batch.

Given a credential `c` and its Merkle proof of inclusion `mproof`, we define `c` to be valid if and only if the following holds:

```
alias computedMerkleRoot = computeRoot(c, mproof)
alias batchId = computeBatchId(c, computedMerkleRoot) // we combine the data to compute the batch
alias issuerDIDSuffix = c.issuerDID.suffix
alias keyId = c.keyId
alias signature = c.signature
alias credentialHash = hash(c)

// control key data
state.publishedDids.contains(issuerDIDSuffix) &&
state.publishedDids(issuerDIDSuffix).keys.contains(keyId) &&
state.publishedDids(issuerDIDSuffix).keys(keyId).usage == IssuingKey &&
// check that the credential batch matches the credential data
state.credentialBatches.contains(batchId) &&
state.credentialBatches.contains(batchId).issuerDIDSuffix == issuerDIDSuffix &&
state.credentialBatches.contains(batchId).merkleRoot == computedMerkleRoot &&
// check that the batch was not revoked
state.credentialBatches(batchId).batchRevocationEvent.isEmpty &&
// check that the specific credential was not revoked
(
  ! state.revokedCredentials.contains(batchId) ||
```

```
  ! state.revokedCredentials(batchId).contains(credentialHash)
) &&
// check the key timestamp compared to the credential issuance timestamp
(
  state.publishedDids(issuerDIDSuffix).keys(keyId).keyAdditiontionEvent < state.credentialBatches
  (
    state.publishedDids(issuerDIDSuffix).keys(keyId).keyRevocationEvent.isEmpty ||
    state.credentialBatches(batchId).batchIssuingEvent < state.publishedDids(issuerDIDSuffix).keys
  )
) &&
// check the credential signature
isValid(signature, c, state.publishedDids(issuerDIDSuffix).keys(keyId).key)
```

We want to remark that the key that signs a credential does not need to be the same key that signs the credential batch issuance event. This is why we add checks to control that the signing key in the credential was already present in the DID at the time the batch occurred and, if revoked, we request that the revocation occurred after the credential was issued.

We also want to remark that, currently, the key that signs the credential and the key that signs the issuance operation must belong to the same DID. The reason is that the DID extracted from the credential is the one used to compute the `batchId`. In order to allow one DID to sign the credential and another one to sign the issuance operation, we would need to add two DIDs to the credential and specify which DID should be used for each check.

5 Canonicalization (and comments on signing)

Our protocol uses cryptographic signatures and hash functions to guarantee its security (along with other properties). The use of these cryptographic primitives requires to translate programming languages data representations into sequences of bytes. This process receives the name of *data serialization*. Once data is serialized, we can hash and/or sign it.

In this document, we will describe what we are doing and we will explore some challenges related to signing data and exchanging it between many applications. At the of the document, we will describe a rudimentary signing technique that was implemented as a proof of concept for credentials. We will also comment about a more robust implementation and future challenges that may come.

5.1 Our current state

There are mainly three places where we are hashing/signing data.

- During Atala Operations construction
- When we compute a DID suffix, where we hash the initial DIDData
- When we need to sign credentials

Let's explore the approach we have on the three situations.

5.1.1 Signing atala operations

The way in which we are generating the bytes to sign from Atala Operations is through protobuf messages. We have:

```
message AtalaOperation {
    oneof operation {
        CreateDIDOperation createDid = 1;
        UpdateDIDOperation updateDid = 2;
        IssueCredentialOperation issueCredential = 3;
        RevokeCredentialOperation revokeCredential = 4;
    };
}

message SignedAtalaOperation {
    string signedWith = 1; // id of key used to sign
    bytes signature = 2; // signature of byte encoding of the operation
    AtalaOperation operation = 3;
}
```

In order to construct a `SignedAtalaOperation`:

1. We construct the needed `AtalaOperation` message using the corresponding protobuf model
2. We extract the bytes produced by protobuf based on the model
3. We sign those bytes and we build a `SignedAtalaOperation` message that is then serialized and posted as part of transaction metadata.

Nodes later validate the signature in the following way:

1. They see the message in transactions metadata
2. They decode the `SignedAtalaOperation` and extract the key, signature and the operation that was theoretically signed
3. They serialize again the `AtalaOperation` into a sequence of bytes and check the signature against those bytes

The process works independently of the programming language and platform used to generate the signature and the one used to verify it because protobuf is *currently* providing the same bytes

from our messages in all platforms. This means that protobuf is *currently* providing a *canonical* bytes representation of the serialized data.

However, we must remark that this is not a feature that protobuf guarantees nor provides in all situations. For example, if our models use maps then, the "canonical bytes" property we rely on would be lost, because different languages may encode maps in different ways. Furthermore, protobuf specification advise to not assume the byte output of a serialized message is stable.

If the application that creates the `AtalaOperation` would generate different bytes as serialization than the ones the node generates when serializing the operation, then the signature validation process would fail (because the bytes signed by the first application would not match the bytes used by the node during verification). This could for example happen if the verifying party is using old versions of the protobuf models.

In order to solve these issues, we should consider to attach the signed bytes "as is" in the encoded protobuf messages. For example, for `SignedAtalaOperation` we should refactor the message to:

```
message SignedAtalaOperation {
    string signedWith = 1; // id of key used to sign
    bytes signature = 2; // signature of byte encoding of the operation
    bytes operation = 3;
}
```

5.1.2 Computing DID suffix

For the computation of the DID suffix of a given initial state of a DID Document, we face a similar situation as the one before. Our protocol defines that the DID suffix associated to a DID Document is the hash of certain DID Data associated with the initial state of the document. An important property we have is that clients are able to compute a DID suffix without the need of publishing it.

NOTE: We are currently hashing the entire `AtalaOperation` (that contains a `CreateDIDOperation`) and not the `DIDData` part.

The way in which we achieve consistent hashes in the client and node side is that given the models associated to these protobuf messages:

```
enum KeyUsage {
    // UNKNOWN_KEY is an invalid value - Protobuf uses 0 if no value is provided and we want user
    UNKNOWN_KEY = 0;
    MASTER_KEY = 1;
    ISSUING_KEY = 2;
    COMMUNICATION_KEY = 3;
    AUTHENTICATION_KEY = 4;
}

message ECKeyData {
    string curve = 1;
    bytes x = 2;
    bytes y = 3;
}

message PublicKey {
    string id = 1;
    KeyUsage usage = 2;
    oneof keyData {
        ECKeyData ecKeyData = 8;
    };
}
```

```
message DIDData {
    string id = 1; // DID suffix, where DID is in form did:atala:[DID suffix]
    repeated PublicKey publicKeys = 2;
}

message CreateDIDOperation {
    DIDData didData = 1; // DIDData with did empty id field
}
```

both (client and node) can construct the DID suffix by hashing the bytes of the corresponding `AtalaOperation` message. Note again, that this still depends on the weak assumption that we can trust on the stability of the bytes obtained.

5.1.3 Credential signing

There is currently a PoC that needs to be reviewed on this topic. So we won't expand about any approach for now. We will document the final approach in this section. At the end of this document, there are some comments on a simple approach.

5.2 Comments on JSON

There has been conversation related to the use of JSON to model credentials. It is also the case that both DID Core and Verifiable Credentials Data Model drafts provide JSON and JSON-LD based descriptions of their data models.

According to ECMA-404, JSON is a text syntax that facilitates structured data interchange between all programming languages. However, on the same document it is stated that

> The JSON syntax is not a specification of a complete data interchange. Meaningful data interchange requires agreement between a producer and consumer on the semantics attached to a particular use of the JSON syntax. What JSON does provide is the syntactic framework to which such semantics can be attached.

JSON's simplicity favoured its wide adoption. However, this adoption came with some interoperability problems.

ECMA-404 says:

> The JSON syntax does not impose any restrictions on the strings used as names, does not require that name strings be unique, and does not assign any significance to the ordering of name/value pairs. These are all semantic considerations that may be defined by JSON processors or in specifications defining specific uses of JSON for data interchange.

On JSON RFC 8259 we see statements like:

> The names within an object SHOULD be unique.

Meaning that names could be repeated according to RFC 8174 definition of "SHOULD"

The RFC mentions differences on implementations based on repeated fields:

> An object whose names are all unique is interoperable in the sense that all software implementations receiving that object will agree on the name-value mappings. When the names within an object are not unique, the behavior of software that receives such an object is unpredictable. Many implementations report the last name/value pair only. Other implementations report an error or fail to parse the object, and some implementations report all of the name/value pairs, including duplicates.

On the topic of element ordering the same RFC says:

> JSON parsing libraries have been observed to differ as to whether or not they make the ordering of object members visible to calling software. Implementations whose behavior does not depend on member ordering will be interoperable in the sense that they will not be affected by these differences.

which is a relevant point to obtain canonicalization if one needs to hash/sign the same JSON in multiple applications.

The RFC and ECMA are consistent with respect to the definition of JSON texts. However, ECMA-404 allows several practices that the RFC specification recommends avoiding in the interests of maximal interoperability.

DID Core draft used to define DID Documents **as** JSONs. At the time of this writing the specification has no text in the Data model section and does not define a DID Document as a JSON anymore. It does talk about JSON, JSON-LD and CBOR core representations.

On section 8 (Core representations) it says:

> All concrete representations of a DID document MUST be serialized using a deterministic mapping that is able to be unambiguously parsed into the data model defined in this specification. All serialization methods MUST define rules for the bidirectional translation of a DID document both into and out of the representation in question. As a consequence, translation between any two representations MUST be done by parsing the source format into a DID document model (described in Sections § 6. Data Model and § 3.3 DID Documents) and then serializing the DID document model into the target representation. An implementation MUST NOT convert between representations without first parsing to a DID document model.

The lack of a canonical binary representation of JSON texts makes them not ideal for cryptographic treatment. There are different proposals to get canonical JSON serialization, none of which seems to be considered a formal standard.

5.2.1 JCS

JCS (IETF Draft 17) is a canonization proposal for JSON texts. The JCS specification defines how to create a canonical representation of JSON data by building on the strict serialization methods for JSON primitives defined by ECMAScript, constraining JSON data to the I-JSON RFC7493 subset, and by using deterministic property sorting.

We found implementations in different languages, including Java. There is also an available JWS-JCS PoC.

5.3 On JSON-LD, RDF and LD-PROOFS

Another alternative proposed so far is the use of Linked Data structures and LD-PROOFS.

LD-PROOFS is an experimental specification on how to sign Linked Data. At the time of this writing, the work published doesn't seem robust. The draft is from March 2020 and says:

> This specification was published by the W3C Digital Verification Community Group. It is not a W3C Standard nor is it on the W3C Standards Track. Please note that under the W3C Community Contributor License Agreement (CLA) there is a limited opt-out and other conditions apply. Learn more about W3C Community and Business Groups.
>
> This is an experimental specification and is undergoing regular revisions. It is not fit for production deployment.

The document doesn't clearly define any proof type or explain how to use them. Furthermore, on the section titled Creating New Proof Types, we infer that the specification plans to describe with Linked Data representation (illustrated with JSON-LD properties like `canonicalizationAlgorithm`, which would refer to the canonicalization algorithm used to

produce the proof. Hence, this does not seem to focus on defining a canonical representation. It instead attempts to give a way to inform all needed data to verify a generated proof.

JSON-LD is a specific JSON based format to serialise Linked Data. At the time of this writing, its status is a Candidate Recommendation and is believed to soon become an endorsed recommendation from W3C. In particular,

> JSON-LD is a concrete RDF syntax as described in RDF11-CONCEPTS. Hence, a JSON-LD document is both an RDF document and a JSON document and correspondingly represents an instance of an RDF data model. However, JSON-LD also extends the RDF data model…
>
> Summarized, these differences mean that JSON-LD is capable of serializing any RDF graph or dataset and most, but not all, JSON-LD documents can be directly interpreted as RDF as described in RDF 1.1 Concepts.

RDF is another syntax used to describe resources and linked data. In particular, RDF is an official W3C recommendation.

RDF-JSON was an initiative to represent JSON documents with RDF. However, we can see the following message in the draft page:

> The RDF Working Group has decided not to push this document through the W3C Recommendation Track. You should therefore not expect to see this document eventually become a W3C Recommendation. This document was published as a Working Group Note to provide those who are using it and/or have an interest in it with a stable reference. The RDF Working Group decided to put JSON-LD on the Recommendation track. Therefore, unless you have a specific reason to use the syntax defined in this document instead of JSON-LD, you are encouraged to use JSON-LD.

We didn't invest further time researching on Linked Data signatures after exploring LD-PROOFs. If needed, we could ask for an update in DIF slack.

5.4 Comments on credentials' signing

Research aside, a simple process we could follow to sign credentials is:

1. Model the credential data as a JSON due to its flexibility
2. Serialize the JSON to an array of bytes
3. Sign the bytes
4. Define our `GenericCredential` as based64url of the serialized bytes of the following JSON

```
{
  signature:  base64url(signature),
  credential: base64url(bytes(credential_data))
}
```

The recipient will have to store the bytes "as is" to preserve a canonical hashable representation.

Alternatively, instead of using JSON we could create the following protobuf message

```
message GenericCredential {
  bytes signature = 1;
  bytes credential = 2;
}
```

In both cases, the signing key reference can live inside the credential. The use of protobuf, as an alternative to JSON, allows to obtain the bytes of the `GenericCredential` and hash them to post them in the blockchain. Later, if the credential is shared to a client using another implementation language, we wouldn't need to worry about canonicalization to compute the hash as long as the protobuf message is exchanged. This is the same trick we are using for operation signing and DID suffix computation.

An even simpler approach was implemented as a proof of concept. In that PoC, the generic credential was represented as a pair of dot (.) separated strings, where the first string represented the encoded bytes of the credential signed and the second string represented the encoded bytes of the signature.

```
base64url(bytes(credential)).base64url(signature)
```

This model resembles to JWS. The main difference is that JWS contains a header with key reference, signing algorithm information and other data. We are planning to implement a JWS based version of verifiable credentials for MVP scope.

5.5 A note on future goals (selective disclosure)

We want to remark that, in order to implement selective disclosure, we may probably need something different to the approach described in the previous section. For example, if we use a Merkle Tree based approach, the bytes to sign are the ones corresponding to the merkle root of the eventual generic credential tree and not the credential bytes themselves. This implies that the credential to share could be of the form:

```
{
  proof: base64url( bytes(  {
    signature:  base64url(signature),
    root: base64url(merkle_root)
  } ) ),
  credential: Actual_credential_data
}
```

and we would need a canonical transformation from the credential data to the Merkle Tree used to compute the root. The hash of the `proof` property would be posted as a witness on-chain. Later, on selective disclosure, a user would send the following data:

```
{
  witness: base64url( hash(the `proof` property defined above) ),
  data_revealed: [
    {
      value: value_revealed,
      nonce: a nonce associated to the revealed value,
      path: [ ... ], // basicaly a list of left-right indicators
      hashes: [ ...] // list of base64url hashes that need to be appended following `path` instruc
                     // merkle root
    },
    ...
  ]
}
```

This may also be changed if we move toward ZK proofs.

6 Key derivation and account recovery process

6.1 Context

This document describes the process used to derive keys from a given seed and recover account information. A user account is composed of mainly three parts:

- Cryptographic keys: Keys used by a user to control his communications and identity
- DIDs: Decentralised identifiers created and owned by the user
- Credentials: Credentials issued, sent and/or received by the user

The three parts of an account are stored in different places. - First, cryptographic keys are stored in user wallets. During registration, each user generates a master mnemonic seed based on BIP39. The seed is known only by the user and it is the user's responsibility to store this seed securely. We currently have no mechanism to recover an account if a user loses his seed. From that seed, we generate keys based on BIP 32. - Second, based on the cryptographic keys, users create their DIDs. DIDs are posted on atala files which are anchored in the underlying blockchain by the node. Associated DID Documents and events that update them are also stored off-chain and anchored on the ledger by atala operations. - Third, credentials are stored by different components, specifically: - The credentials manager for the case of issuers and verifiers, and - The mobile wallet in the case of other users.

For all users, shared credentials are also stored (encrypted) in the connector.

The account recovery process that we are describing, applies to solve extreme cases, such as hardware damage on clients' side. For example, when a user's computer/phone is lost, stolen or breaks down.

In the following sections, we will first describe the process of keys and DIDs generation. Later on, we will explain how to recover the different parts of an account.

6.2 Definitions

Throughout this text, we will use definitions extracted from BIP 32 conventions

We assume the use of public key cryptography used in Bitcoin, namely elliptic curve cryptography using the field and curve parameters defined by secp256k1.

Given a initial (public, private)-key pair `m`, that we will call key master seed, we will define a function that derives a number of child keys from `m`. In order to prevent all keys depending solely on the key pair itself, the BIP extends both private and public keys first with an extra 256 bits of entropy. This extension, called the **chain code**. The chain code is identical for corresponding private and public keys, and consists of 32 bytes. We refer to these (key, chain code) pairs as *extended keys*. - We represent an extended private key as `(k, c)`, with `k` the normal private key, and `c` the chain code. - An extended public key is represented as `(K, c)`, with `K = point(k)` and `c` the chain code. Where `point(p)` returns the coordinate pair resulting from EC point multiplication (repeated application of the EC group operation) of the secp256k1 base point with the integer `p`.

Each extended key has `2^{31}` normal child keys, and `2^{31}` **hardened child** keys. Each of these child keys has an index. - The normal child keys use indices `0` through `2^{31}-1` - The hardened child keys use indices `2^{31}` through `2^{32}-1`. To ease notation for hardened key indices, a number `i'` represents `i+2^{31}`

Hardened keys present different security properties that non-hardened keys.

6.3 Generation process

6.3.1 Root key generation

From BIP39 spec, the user generates a mnemonic seed that can be translated into a 64 bytes seed.

To create a binary seed from the mnemonic, we use the PBKDF2 function with a mnemonic sentence (in UTF-8 NFKD) used as the password and the string "mnemonic" + passphrase (again in UTF-8 NFKD) used as the salt. The iteration count is set to 2048 and HMAC-SHA512 is used as the pseudo-random function. The length of the derived key is 512 bits (= 64 bytes).

From that 64 bytes seed, we will derive an extended private key that we will note with m. Given the initial seed, libraries will provide a `generate` method to obtain m. We will refer to m as the root of our keys.

6.3.2 Children key derivations

BIP32 spec presents three functions for extended key derivation: - CKDpriv((k, c), index): Given an extended private key and an index (>= 0), returns a new "child" extended private key. - N((k, c)): Given an extended private key, it returns the corresponding extended public key (point(k), c) In order to derive the ith child extended public key from an extended private key (k, c), we perform N(CKDpriv((k, c), i)) - CKDpub((K, c), index): Given an extended public key and an index, returns a new "child" extended public key. We won't use this last function

Comments - Note that it is not possible to derive a private child key from a public parent key. - We refer to these child keys as *nodes* because the entire derivation schema can be seen as a tree with root m. - Each node can be used to derive further keys. This allows to split the different branches of the tree and assign different purpose for each one of them. - **Notation:** We will use path notation, meaning that instead of `CKDpriv(m, i)` we will write `m / i`. For example, the expression `CKDpriv(CKDpriv(CKDpriv(m, 0'), 2'), 5')` translates to `m / 0' / 2' / 5'`. Recall that `i'` means `i+2^{31}`. - Libraries typically use path notation.

For security reasons we will only use hardened child keys. For more details on this decision, read this link .

6.3.3 The paths used in our protocol

Given our root key m, generated from a mnemonic seed, we will structure our derivation tree according to the following path.

`m / DID_NUMBER / KEY_TYPE / '`

where

- `DID_NUMBER` is a hardened node i' and represents the (i+1)th DID of a user.
- `KEY_TYPE` is one of:
  - 0': Representing master keys. The keys derived from this node are the only keys that can be used as MASTER_KEYs in users' DIDs.
  - 1': Representing issuing keys.The keys derived from this node are the only keys used for credentials and operations signing.
  - 2': Representing communication keys. The keys derived from this node are keys used to exchange establish connections and exchange messages with other users.
  - 3': Representing authentication keys. The keys derived from this node are keys used to authenticate with websites and servers.
- The final `'` means that all derived keys must be hardened keys

NOTE: The `KEY_TYPE` list may be updated as we progress with the implementation.

Examples

- m / 0' / 0' / 0' is the first master key for the first DID of a user derived from m
- m / 0' / 0' / 1' is the second master key derived for the first DID for a user derived from m.
- m / 1' / 0' / 0' is the first master key for the second DID derived from m
- m / 3' / 0' / 5' is the sixth master key derived for the fourth DID derived from m.
- m / 1' / 1' / 0' is the first issuing key for the second DID derived from m
- m / 3' / 1' / 1' is the second issuing key for the fourth DID derived from m

Note that all nodes and final keys are hardened ones.

Terminology and notation: - Given a `DID_NUMBER` `n`, we refer to the master key `m / n' / 0' / 0'` as the **canonical master key of the DID**. Every DID that we will generate has a unique canonical master key. - We will use the term *fresh key* to refer to a key that has not been marked as used already. - Given a path `p` we say that a key `k` is *derived* from `p` if there exist `i` such that `k = p / i`. - We say that a key `k` is derived from a master seed `m` if there exists a path `p` with root `m` and index `i` such that `k = p / i`.

Conventions Whenever we generate a fresh key: - We will use hardened keys. - we will use keys in order. This is, we always use the key derived from the smallest non-used hardened index. For example, with respect generating a fresh issuing key for seed `m` and DID `d`, we will always pick the key generated by the smallest `i` where `m / d' / 1' / i'` hasn't been used.

6.3.4 DID generation

We would like to obtain a DID based from the mnemonic seed too. For this, we will fix a format for our DID Documents. When a user wants to create his first DID Document, the following process will be followed:

- Take `cmk = m / 0' / 0' / 0'`. This is, the canonical master key for the first DID derived from m.
- Mark this key as used for future reference.
- Create a DID Document that only contains `cmk` as a master key as its only field. The `keyId` field of the key object will be `master-0`. We call this document the *canonical document associated to* `cmk`
- Send a CreateDID request to the node.

The request will return the DID suffix of the canonical document. We refer to `did:atala:[returned suffix]` as the *canonical DID* associated to `cmk`.

To create the second DID, we now follow these steps: - Take `cmk = m / 1' / 0' / 0'`. This is, the canonical master key for the second DID derived from m. - Mark this key as used for future reference. - Create a DID Document that only contains `cmk` as a master key as its only field. Again, with `keyId master-0`. We call this document the *canonical document associated to* `cmk` - Send a CreateDID request to the node.

And, in the general case. If a user wants to create his (n+1)th DID: - Take `cmk = m / n' / 0' / 0'`. This is, the canonical master key for the (n+1)th DID derived from m. - Mark this key as used for future reference. - Create a DID Document that only contains `cmk` as a master key as its only field, with `keyId master-0`.
We call this document the *canonical document associated to* `cmk` - Send a CreateDID request to the node.

DIDs are generated in order and no keys can be shared between DIDs. This is, in order to create a new DID, we always pick the minimal N such that `m / N' / 0' / 0'` hasn't been used. No DID can use a key from a branch of the tree that does not derived from its own `DID_NUMBER`.

6.3.5 DID Document updates

In order to update the DID Document and add more keys. The keys should also be generated following similar processes as the one for generation. As examples: 1. Imagine a user owns 2 DID Documents D1 and D2 with corresponding canonical master keys k1 and k2 and DID numbers 0 and 1. Assume that D1 also has a non-canonical master key NK1. Graphically `D1 = { keys : [ { keyId: "master-0", type: "Master", id: "k1", key: ... }, { keyid: "master-1", type: "Master", id: "NK1", key: ...}]  D2 = { keys : [ { keyId: "master-0", type: "Master", id: "k2", key: ... }]` To add a new master key to D2 we need to derive the next fresh key from `m / 1' / 0'` which is `m / 1' / 0' / 1'` and add it to D2 through an UpdateDID operation, the `keyId` used will be `master-1`. 2. To add the first issuance key to D1,

we should derive the next issuing key from `m / 0' / 1'` which is `m / 0' / 1' / 0'` and add it to D1 through an UpdateDID operation, the `keyId` used will be `issuance-1`.

By convention, we will always have that the Nth key added of a given `KEY_TYPE` to a DID Document will be `[KEY_TYPE]-[N-1]`. Applications can present uses with local aliases in the front-end. We won't be concerned about recovering such aliases in this document.

6.4 Account recovery

Given the generation and derivation rules. The process to recover an account can be model as follows.

6.4.1 DID recovery

Given that DIDs are generated in order based on canonical master keys. We can use the following process. Given a master seed m

1. Set `i = 0`
2. Compute `cmk = m / i' / 0' / 0'`
3. Compute the canonical DID associated to `cmk`
4. Resolve the canonical DID
    1. If the DID can't be resolved: STOP
    2. If the DID is resolved, then:
        - Store tuples (i, KEY_TYPE, key) for each key present in the resolved DID Document for future steps (do not store cmk)
        - Store cmk as a recovered master key
        - Store the canonical DID as a recovered DID
        - Increase i by 1 and go to step 1

This process will end up with all generated DIDs and their canonical master keys stored. It will also compute the set of keys present in the DID Documents that we will attempt to recover in the next step of account recovery.

6.4.2 Key recovery

Note: This process assumes that all keys in the DID Documents owned by the user, were derived by the initial seed m. While recovering all DIDs, we computed a set of (DID_NUMBER, key, KEY_TYPE) tuples that need to be recovered. Let's call this set `KTR`. Given that we derive keys in order, we can recover all listed key in the following way.

1. Group tuples in `KTR` and group them by `(DID_NUMBER, KEY_TYPE)`. This will give us a set of keys that belong to the same DID and have the same KEY_TYPE. We will note each one of this set `KTR(DID_NUMBER, KEY_TYPE)`
2. For each set `KTR(i, kt)` from step 1. Derive the first `size(KTR(d,kt))` hardened keys from `m / i' / kt'`. This is the set `{ m / i' / kt' / 0',..., m / i' / kt' / (size(KTR(i, kt)) - 1)' }`
3. The keys from step 2 MUST provide us matching private keys corresponding to the ones in the `KTR(i, kt)` sets.
4. Mark the recovered keys as used for future reference.

We would like to remark that the process could be adapted to allow external keys in our DID Documents. The process would follow the same steps with the modification that, in step 2, we will generate keys until we find the first one that does not match a public key in the set `KTR(d,kt))`. Also, in step 4, only keys with matching public key in a DID Document will be marked as used.

6.4.3 Credentials recovery

Now that we have explored how to recover all DIDs and keys present in them. The remaining part of the account that we need are the credentials relevant to the user. Issuers and verifiers

currently have their credentials in the credentials manager. Holders can find them in the connector. The recovery process will consist of generating keys used to communicate with services (e.g. authorization keys) from the master seed and call adequate APIs to get credentials stored by these components.

Note that the above process should be updated according to implementation updates.

6.4.4 Test vectors

Test vectors are available as JSON file.

6.4.4.1 Vector 1 Seed phrase: abandon amount liar amount expire adjust cage candy arch gather drum buyer * [DID: 1] * DID: did:prism:6fe5591aabaf1e41744f074336001f37be74534c00a99c3874c3a4690981dced * [Key master 0] * Key ID: master-0 * BIP32 Path: m/1'/0'/0' * Secret key * (hex): dd7115d710d2eaa591f241145ebead016f7a2b90f2b86f5f743731fe37aa3ac5 * Public key * (hex): 03cba11a413c631c853685bfd852b3163ffb124c03712f4a81cd115f72d6ced9f9 * (x-hex): cba11a413c631c853685bfd852b31 * (y-hex): 69278cf55b5d72ea6ad01f1a14787c2ee316cbe8f96897c6f8e9b23b13efe565 * [Key master 1] * Key ID: master-1 * BIP32 Path: m/1'/0'/1' * Secret key * (hex): 3c10eeba06cd6efefe017146fdf400d7e78b9fa461a6c7cc * Public key * (hex): 03cdd203ac26fbc3282abd9a422558a3185371a27406164e2433a155a7bf901fa8 * (x-hex): cdd203ac26fbc3282abd9a422558a3185371a27406164e2433a155a7bf901fa8 * (y-hex): e9b72fe04894b19a8931d483a6b0979e95e3bbb34f786b6ac8512199989d2703 * [Key issuing 5] * Key ID: issuing-5 * BIP32 Path: m/1'/1'/5' * Secret key * (hex): f3222b9f1eeea1c11ceaaf39ff428c140c0f03e323d82d13ba94c * Public key * (hex): 0208e12a3029d8e6635ed40788250014831d47f58ed9e9ff5ddb217ab9fe931c09 * (x-hex): 08e12a3029d8e6635ed40788250014831d47f58ed9e9ff5ddb217ab9fe931c09 * (y-hex): 2eb3147e1c0d93908189f708657ac35b4e06a563b98bf09731d997be719a0d5e * [Key communication 20] * Key ID: communication-20 * BIP32 Path: m/1'/2'/20' * Secret key * (hex): f8047dd871d8e79f54d9f202910eba779e67761d60b1b09ad1b8e72f2acbab2d * Public key * (hex): 02c10a63a773514bebfc8c9425737963ed135936172cec6aa13c9777201ffac50c * (x-hex): c10a63a773514bebfc8c9425737963ed135936172cec6aa13c9777201ffac50c * (y-hex): 58eaf1c1ed59a0eb56525498aabf8256e93953baca0c320c24827ea203b33ec4 * [Key authentication 27] * Key ID: authentication-27 * BIP32 Path: m/1'/3'/27' * Secret key * (hex): 0251a9f0c65de414c9522834afe62806d8c1a538c5282abbebf51f9f92f1eab5 * Public key * (hex): 02ae26207160333e91fad88529b784bdbd9cd1a95e5ab6bbdc93ef289fa617c72e * (x-hex): ae26207160333e91fad88529b784bdbd9cd1a95e5ab6bbdc93ef289fa617c72e * (y-hex): c48c13b927248aab32874c0506dd26b6a90ef8452ab86147a4317045aed8f958 * [DID: 17] * DID: did:prism:60e5c0b68701bac49873bc273017ad199a063e1b614444312dd2e97e1e9fb164 * [Key master 0] * Key ID: master-0 * BIP32 Path: m/17'/0'/0' * Secret key * (hex): 038ff9d6e6830ca7f5e875d4400c2a9f973551cafc7b077e5dfb23e49eb13e3f * Public key * (hex): 023d372976f436182400d21c07404b6d42fb87f84e59b5a7c715025cf85a5a3362 * (x-hex): 3d372976f436182400d21c07404b * (y-hex): dd0be8438eacd02d82e2e0c3069b20f71bda8e9f57a49183f73f4f4c127435d4 * [Key master 17] * Key ID: master-17 * BIP32 Path: m/17'/0'/17' * Secret key * (hex): 415590b46f2e4b0453da62328b8701becdacffd582fcd1008973774589199457 * Public key * (hex): 02855112018b81d80d0187480fee241d0abf38e2f80dcab0039344f1b4a97cb7ab * (x-hex): 855112018b81d80d0187480fee241d0abf38e2f80dcab0039344f1b4a97cb7ab * (y-hex): 5dcc4c3877189fceaf1809aed0fd449 * [Key authentication 0] * Key ID: authentication-0 * BIP32 Path: m/17'/3'/0' * Secret key * (hex): 7da8202ee5c58aebe3e3b1003c5217c8e9a841ff5da9e6358010635ed126383c * Public key * (hex): 03f6ede796792f949807db272c40f451811faf0f7eecb7fb2c6c0fd15d1c62b778 * (x-hex): f6ede796792f949807db272c40f451811faf0f7eecb7fb2c6c0fd15d1c62b778 * (y-hex): 217cb6a49fce5b20e646015422d7ad7c9c70ff7e6f2d02a1228eaff0ada9d66d * [Key authentication 17] * Key ID: authentication-17 * BIP32 Path: m/17'/3'/17' * Secret key * (hex): 9f60801c2bf70b18071e8dd2d01d851a5e15602a407a79ff110e6dfb8371e6ce * Public key * (hex): 021b22881120925cf10381f5a247634665a677f98505bf183726a9b90f245a8a95 * (x-hex): 1b22881120925cf10381f5a247634665a677f98505bf183726a9b90f245a8a95 * (y-hex): 5a1b2feb44de35e5071810f931b8b4cf93bb6103eb86665563c65e120f6d9bc8

6.5 Future challenges

A problem for a near future is that we would like to enable users to create DID Documents that contain more than just the canonical master key. This can be achieved today by creating a canonical DID and then performing an update operation. Given the metadata space we have, we could batch the two operations in a single Cardano transaction. If we opt for this approach, the recovery process would work without the need of changes.

As an alternative, we could generate initial DID documents with more keys and data than the canonical master key and back up them in some external storage. The recovery process would then iterate upon the documents and re-generate the keys associated to them.

Another alternative could be to extract from an initial state of DID Document its *canonical* part, this is the initial master key associated to the DID. The DID suffix could be dependent only on the canonical part. For example, an initial DID Document could be:

```
{
  "keys": [
    {
      "id": "master0",
      "type": "MASTER_KEY",
      "key": ...
    },
    {
      "id": "issuance0",
      "type": "ISSUING_KEY",
      "key": ...
    }
  ]
}
```

where the canonical part is the DID Document

```
{
    "keys": [
        {
          "id": "master0",
          "type": "MASTER_KEY",
          "key": ...
        }
    ]
}
```

The DID suffix could be computed from the canonical part (its hash). This could allow for published DIDs to have a generic initial state without breaking our recovery process. However, the non-canonical part does not become self certifiable anymore. Meaning that for an unpublished DID, a user could have the DID Documents:

```
{
  "keys": [
    {
      "id": "master0",
      "type": "MASTER_KEY",
      "key": ...
    },
    {
      "id": "issuance0",
      "type": "ISSUING_KEY",
      "key": ...
    }
```

```
  ]
}
```

and

```
{
  "keys": [
    {
      "id": "master0",
      "type": "MASTER_KEY",
      "key": ...
    },
    {
      "id": "master2",
      "type": "MASTER_KEY",
      "key": ...
    }
  ]
}
```

and both would have the same short form (canonical) DID. The approach still would not solve the recovery of unpublished DIDs.

We have also mentioned that the node could expose an endpoint to obtain DIDs based on a key. This could also allow a recovery process that remains compatible with a generic initial DID Document.

The above non-extensive analysis suggests the need of external storage for DID recovery or to split the creation of complex DID Documents into a Create and an Update part.

7 Unpublished DIDs

7.1 Objective

In order to increase the scalability and reduce operational costs for our protocol, we have proposed the use of unpublished DIDs. In the current implementation (at the time of this writing), every DID is published on-chain as soon as it is created. After publication, a node can resolve a DID and obtain the current state of the DID document. Publishing a DID currently allows to:

- Resolve the DID
- Send DID update operations
- Issue credentials in our setting where proofs of existence are posted on-chain

The main drawback we face is that every DID publication requires an underlying ledger transaction, leading to a delay between the moment the DID is generated, and the time it becomes resolvable by nodes. The process also adds a fee cost. We find it reasonable to believe that a substantial number of users won't use every DID for all the above features. Hence, we propose to mitigate the stated drawbacks by allowing a DID to fulfil a subset of functionalities *before* being published. In particular, we would like to create a concept of **unpublished DID** that:

- Allows a DID to Be resolvable before on-chain publication
- Allows the DID to be published if needed

In the rest of the document, we present an approach we have evaluated from Sidetree and comment on the potential next steps we could follow.

7.2 Approaches reviewed

7.2.1 Sidetree

In Sidetree we can see the definition of Long form DID URI designed to

- Resolving the DID Documents of unpublished DIDs.
- Authenticating with unpublished DIDs.
- Signing and verifying credentials signed against unpublished DIDs.
- After publication and propagation are complete, authenticating with either the Short-Form DID URI or Long-Form DID URI.
- After publication and propagation are complete, signing and verifying credentials signed against either the Short-Form DID URI or Long-Form DID URI.

In a simplified description, the specification uses URI parameters (the `?initial-state=` parameter) to attach the initial state of the DID to the DID URI. The intended use is that, a resolver would take the DID and attempt to resolve it, if the associated DID Document is not found, then the resolver returns the decoded initial state document attached in the `initial-state` URI parameter. If, however, the DID Document is resolved, then the `initial-state` value is ignored and resolution works as if the URI parameter was never attached.

The approach described faced the following problem As discussed in issues #782 and #777 on Sidetree's repository and on this issue in DID Core, the use of URI parameters may lead to some inconsistencies in the resolved DID document. For example, DID Documents MUST have an `id` field which MUST be a DID. This means that if we have a long form DID:

```
did:<method>:<suffix>?initial-state=<encoded-state>
```

which hasn't been published, and we resolve it, the DID Document obtained should look like:

```
{
  id: "did:<method>:<suffix>",
  ...
}
```

leading to an `id` that could be accidentally used and share while being it not resolvable (because the DID could remain unpublished). At the same time, the `id "did:<method>:<suffix>?initial-state=<encoded-stat` is not a valid DID (it is a DID URL though), leading to an invalid DID Document.

To mitigate the issue, Sidetree's working group added to their reference implementation a different format,

> // Long-form can be in the form of: // 'did::?–initial-state=.' or // 'did:::.'

During the WG call on Tuesday 29th 2020, it was confirmed that the syntax with `-initial-state` will be deprecated. According to W3C spec, the second syntax leads to a valid DID. The long form still maintains a short form counterpart, i.e. for a long form DID:

```
'did:<methodName>:<unique-portion>:<create-operation-suffix-data>.<create-operation-delta>'
```

we have a short form (a.k.a canonical form)

```
'did:<methodName>:<unique-portion>
```

The WG didn't define yet (by the time of this writing) how to treat the returned `id` in the resolved DID Document *after* the DID is published. This is, after publication, if a user attempts to resolve the long form DID, should the `id` field of the resolved document contain the long form or short form of the DID? Different proposals are under discussion.

7.3 Proposal for first iteration

In order to provide our equivalent to long form DIDs, we will adopt the form:

```
did:prism:hash(<initial-state>):base64URL(<initial-DID-Document>)
```

7.3.1 Impact on recovery process

We must consider the impact of these new DIDs in our recovery process. The problem we have with unpublished DIDs, is that they won't be found on-chain during the recovery process iteration.

this will lead the algorithm to stop in step 4.1. Given that mobile apps will be the first users of these DIDs, it looks reasonable to adapt the DID recovery process to ask the connector if a DID was used for a connection instead of asking or DID resolution. In the future, we could store all generated DID Documents in a data vault like service.

7.3.2 DID length

We decided to make some tests to measure how long our long form DIDs would look like. We generated 100000 DIDs under 3 settings:

1. With 1 master key
2. With 2 master keys
3. With 3 master keys

The code used can be found here. Example results are:

For 1 key

```
Generating 100000 dids

printing 3 shortest DIDs
(did:prism:7e5b03ee503c6d63ca814c64b1e6affd3e16d5eafb086ac667ae63b44ea23f2b:CmAKXhJcCgdtYXN0ZXIwEA
(did:prism:8f4ecd2b7123b0ddfb7c4a443b3d25823041d470b388e6bb0577cd5d49f9b8a9:CmEKXxJdCgdtYXN0ZXIwEA
(did:prism:0ad343d0c4b9a69014cde239589df57fb3a0adbe0fb3f5c8c4a294bbfd9827cb:CmEKXxJdCgdtYXN0ZXIwEA

printing 3 longest DIDs
(did:prism:0dcbbc7ba0e4194bc43fee20cb9ecf156d9c191b2eb88e75a45decf68d56665c:CmIKYBJeCgdtYXN0ZXIwEA
(did:prism:bbf07987129f43ea213d7c1a1222d6a242efe76c9c853c27d7c90560f287c10d:CmIKYBJeCgdtYXN0ZXIwEA
(did:prism:8b5094925066ade140f99c9bba00c0a8f139882138b5939180af4d827388c1f8:CmIKYBJeCgdtYXN0ZXIwEA

Average DID length 207.99972 bytes
```

For 2 keys

```
Generating 100000 dids

printing 3 shortest DIDs
(did:prism:8159135db4b3cac4c09667087ea7c46b1019d85885f72af2bab083ccd092c5d4:Cr0BCroBEloKB21hc3Rlc
(did:prism:471d8c0b8b813de92b4a70604a56fc0ea68d84ccd9c1a4de9ebaeb1afb906235:CsABCr0BElwKB21hc3Rlc
(did:prism:ef51a3fa3e5e222c819575e3f137ad45c1c1db7e3c06b21a590332e73d20953b:CsABCr0BEl0KB21hc3Rlc

printing 3 longest DIDs
(did:prism:6c5ca0c67ea138eb41ee1a8060527b8d3bbf921e103e977b77786f6b71567000:CsIBCr8BEl0KB21hc3Rlc
(did:prism:0a3d98977ebeda7d11bddcc98d3b3e7072886e3e91ddf2fa7f99bdb1c110b00a:CsIBCr8BEl0KB21hc3Rlc
(did:prism:ca6b2058009b455dca1c3b491fb06e4d37713eda32099b2cf91e31e06befb569:CsIBCr8BEl4KB21hc3Rlc

Average DID length 337.737 bytes
```

For 3 keys

```
Generating 100000 dids

printing 3 shortest DIDs
(did:prism:e2f9e03bf49c5644066dd6efb09a130576c46e4c687861c18685ffc0c63f0bce:Cp0CCpoCEl0KB21hc3Rlc
(did:prism:ad7a81274c0a0295dcf4417ccc501ccfbca1165bf2bcad94430aa42171ed173d:Cp0CCpoCElwKB21hc3Rlc
(did:prism:b7800057dd8d48c86709db5f9191105666301449fa481f0bd4e2e0f244af2745:Cp0CCpoCElwKB21hc3Rlc

printing 3 longest DIDs
(did:prism:4bb94e65c2bc37fc4e52c4f2471418302d26e43cf126677f407a0451be9a7c22:CqECCp4CEl4KB21hc3Rlc
(did:prism:ca7fbf765201acafc4e20ce38ef4dc3d159b6067b5b13ee99ffbf6b7e3f6facc:CqECCp4CEl0KB21hc3Rlc
```

```
(did:prism:33afe0e357fa60eaf205ec9f1fd3e579c3a656c27b5143fbbdfa35c2c2b5916d:CqECCp4CEl0KB21hc3Rlc
```

```
Average DID length 464.28944 bytes
```

We see an increase of ~130 bytes per key. We should point out that we are representing keys as elliptic curve points. We could reduce space by representing keys in compressed representation (i.e. a coordinate and a byte that indicates the sign of the other coordinate).

By comparison, we can see this Sidetree test vector that represents a long for DID of them. The DID has 883 bytes.

8 Late publication

So far, we have explained how the protocol can achieve scalability though a batching technique. However, it is important to also consider other implications of this approach. We will describe a potential "attack" that is possible due to having off-chain data that is referenced from on-chain transactions. In short, the absence of control mechanisms to manage off-chain data availability may allow an attacker to produce undesired situations as we will explain below.

8.1 The basic scenario

At this point, the implementation is not whitelisting who can publish Atala objects and blocks. Let's consider then, the following scenario, Alice decides to create a new DID `D1` with DID Document with key `k1`. She creates an Atala Object, `A1`, and a single operation Atala block, `B1`. She then sends a transaction, `Tx1`, to the underlying ledger with a reference to `A1` in its metadata and publishes the generated files in the CAS. All nodes will see `D1` with `k1` as valid since `Tx1.time`.

Now, imagine that Alice creates another Atala object, `A2`, and corresponding single operation Atala block, B2, that updates `D1` by revoking `k1`, she sends a valid transaction, `Tx2` (with a reference to `A2`), to the underlying ledger, but this time she does not post the files to the CAS. This should lead other nodes that are exploring ledger transactions, to attempt to resolve the reference in `Tx2` and fail. In order to avoid nodes to be blocked by this situations, we could mark `Tx2` with an `UNRESOLVED` tag and continue processing subsequent transactions. Nodes would keep a list of `UNRESOLVED` transactions and would keep attempting to retrieve missing files periodically until they succeed. Once a missing file is found, the node state should be properly updated (where properly means, reinterpret the entire ledger history and applying updates to the state incorporating the missing file operations). Note that we say that this *should* happen, however, the current implementation assumes that the files are always published.

The outcome of the above process should lead all nodes to have the following state:

- A transaction `Tx1` which publishes `D1` with a DID Document that contains `k1` as valid key
- An `UNRESOLVED` transaction `Tx2` (which updates `D1` to revoke `k1`)

```
Ledger
   ----+-----+---------+-----+-----
  ···  | Tx1 |   ···   | Tx2 |   ···
   ----+-----+---------+-----+-----

CAS
[
 hash(A1) --> A1,
 hash(B1) --> B1
]

Node State
unresolved = [hash(A2)]
dids = [ D1 --> [k1 valid since Tx1.time ] ]
```

Alice could now issue a credential `C1` signing the `IssueCredential` operation with `k1`, post it in files `A3`, `B3` with a transaction `Tx3` and publishing `A3`, `B3` in the CAS. Leading to:

```
Ledger
   ----+-----+---------+-----+---------+-----+-----
  ···  | Tx1 |   ···   | Tx2 |   ···   | Tx3 |   ···
   ----+-----+---------+-----+---------+-----+-----

CAS
[
 hash(A1) --> A1,
```

```
 hash(B1) --> B1,
 hash(A3) --> A3,
 hash(B3) --> B3
]

Node State
unresolved = [hash(A2)]
dids = [ D1 --> [k1 valid since Tx1.time ] ]
creds = [ hash(C1) --> Published on Tx3.time ]
```

If Bob receives `C1` and validates it, the node will state that the credential is correct.

Later, Alice can post `A2` and `B2` to the CAS. This *should* lead to a history reinterpretation (note that this is not currently implemented) producing the state in all nodes as follows:

```
Ledger
   ----+-----+---------+-----+---------+-----+-----
  ···  | Tx1 |   ···   | Tx2 |   ···   | Tx3 |   ···
   ----+-----+---------+-----+---------+-----+-----

CAS
[
 hash(A1) --> A1,
 hash(B1) --> B1,
 hash(A2) --> A2,
 hash(B2) --> B2,
 hash(A3) --> A3,
 hash(B3) --> B3
]

Node State
unresolved = []
dids = [ D1 --> [k1 valid since Tx1.time, revoked in Tx2.time ] ]
creds = [ ] -- as the IssueCredential event was signed with an invalid key, C1 may not even be ad
            -- map
```

If Bobs tries to validate `C1` again, he won't only find that `C1` is invalid now. According to ledger history, `C1` was *never* valid because it was signed with a key that was revoked *before* `C1` was published. Note that, the only ways to notify Bob the change in state of `C1` are 1. Bob periodically queries the state of `C1`. 2. The node updates the history rewrite, replays all queries done by Bob and notifies him if any response would be different than the one he received before the history change.

If there is no history rewrites, Bob could simply subscribe to a notification of a `C1` revocation event. As `k1`, in this example, is used to sign the operation, Bob could notify the node to inform him if the credential becomes invalid after history rewrite. Furthermore, note that if `C1` did not support revocation, then this history rewrite could "in the real world" imply its revocation after a proper validation.

8.2 Comparative - situation 2: Attack after key revocation

Let us now imagine the scenario where Alice is not hiding files. She starts by publishing `Tx1`, `A1` and `B1` as before. Imagine that now Alice suspects that her key was compromised and decides to revoke it by publishing `Tx2`, `A2` and `B2`. The state reflected in the system from the perspective of a node would be:

```
Ledger
   ----+-----+---------+-----+-----
  ···  | Tx1 |   ···   | Tx2 |   ···
```

```
   ----+-----+---------+-----+-----

CAS
[
 hash(A1) --> A1,
 hash(B1) --> B1
 hash(A2) --> A2,
 hash(B2) --> B2
]

Node State
unresolved = [ ]
dids = [ D1 --> [k1 valid since Tx1.time, revoked at Tx2.time ] ]
creds = [ ]
```

If an attacker Carlos, who actually got control of `k1`, creates a credential `C1` and sends an `IssueCredential` event for `C1` in `Tx3`, `A3`, `B3`. We would get the state:

```
Ledger
   ----+-----+---------+-----+---------+-----+-----
  ···  | Tx1 |   ···   | Tx2 |   ···   | Tx3 |   ···
   ----+-----+---------+-----+---------+-----+-----

CAS
[
 hash(A1) --> A1,
 hash(B1) --> B1,
 hash(A2) --> A2,
 hash(B2) --> B2,
 hash(A3) --> A3,
 hash(B3) --> B3
]

Node State
unresolved = []
dids = [ D1 --> [k1 valid since Tx1.time, revoked in Tx2.time ] ]
creds = [ ] -- C1 is not even added to creds map because the key to sign the IssueCredential opera
```

Note that in this situation there is no point in "real world time" where `C1` could have been valid, as Carlos was not able to publish `C1` before `k1` is revoked. The point that we want to remark is: ***this ledger, CAS and node states are identical to the ones where the late publish occurred***

8.3 Comparative - situation 3: Attack after and later credential and key revocation

Let us now analyse a third situation. In this case, there is again no late publication. As in the last two scenarios, we start with Alice publishing `D1` with valid `k1` through `Tx1`, `A1` and `B1`. Now imagine that Carlos gains access to `k1` and issues `C1` though `Tx2`, `A2`, `B2` before Alice manages to revoke `k1`. The system state would look like this:

```
Ledger
   ----+-----+---------+-----+-----
  ···  | Tx1 |   ···   | Tx2 |   ···
   ----+-----+---------+-----+-----

CAS
[
```

```
 hash(A1) --> A1,
 hash(B1) --> B1
 hash(A2) --> A2,
 hash(B2) --> B2
]

Node State
unresolved = [ ]
dids = [ D1 --> [k1 valid since Tx1.time ] ]
creds = [ hash(C1) -> Published on Tx2.time ]
```

If now Bob receives `C1` and verifies it, it will receive a response stating that the credential is valid. Now, if Alice notices that a credential was issued without her knowledge, she could revoke `C1` and `k1` through a transaction `Tx3` with files `A3` and `B3` containing an `UpdateDID` and a `RevokeCredential` operations. Leading to the state:

```
Ledger
   ----+-----+---------+-----+---------+-----+-----
  ···  | Tx1 |   ···   | Tx2 |   ···   | Tx3 |   ···
   ----+-----+---------+-----+---------+-----+-----

CAS
[
 hash(A1) --> A1,
 hash(B1) --> B1,
 hash(A2) --> A2,
 hash(B2) --> B2,
 hash(A3) --> A3,
 hash(B3) --> B3
]

Node State
unresolved = []
dids = [ D1 --> [k1 valid since Tx1.time, revoked in Tx4.time ] ]
creds = [ hash(C1) -> Published on Tx2.time, revoked on Tx3.time ]
```

Note that both the blockchain and the node state reflect the period of time in which verifying `C1` could have resulted in an is valid conclusion. We could even extend the operations we support to add a `RevokeSince` to reflect that the intention is to revoke a credential since a block number/time previous to the revocation event and still keep the record of precisely what happened on chain.

Comment

We would like to remind that, we wait for transactions to be in the stable part of the ledger before applying them to the node state. This means that there is a time period a user has to detect that an event triggered on his behalf is going to be applied. We could allow issuers to specify waiting times in some way to facilitate key compromised situations. E.g. a credential schema could be posted on chain specifying how many **stable** blocks the protocol needs to wait to consider a credential valid (to facilitate the issuer with time to detect compromised keys). If a credential is issued but a revocation event is found (in a stable block) before the end of this waiting period (after the issuance event), the protocol can then ignore the credential, never adding it to its state and, hence, improving the chance for unauthorised credentials to never be valid.

Another small improvement we could remark is, if a client queries data to a node, and the node can see in the **unstable** part of the blockchain that an event will affect the queried data, the node could reply requests with an extra flag pointing out the situation. It will be then up to the client to wait for a few minutes to see if data changes are confirmed, or to simply ignore the incoming information.

8.4 Some conclusions

A system that allows late publishing brings some complexities:

- Rollbacks and history rewrites should be handled properly
- Clients should have richer notification methods to understand what happened
- A priori, credentials that do not have a revocations semantic, could be revoked through late publication
- Auditability becomes a bit blurry
- It makes reasoning more complex if we expand the protocol with more operations
- As illustrated here, it also adds complexities to DIDs or credentials transferability.

Now, not everything is negative. Late publishing possibilities are a consequence of decentralization.

If one decides not to allow late publishing, the selected approach to do so may bring disadvantages:

- Whitelisting who can issue operations with batches
  - This leads to some centralization. It could be mitigated by allowing on-chain operation publishing (which is our current plan). Whoever does not want to relay on IOHK, can issue their operations on-chain without batching. They would need to trust that IOHK won't create a late publish scenario (which everyone would be able to audit).
  - We could also consider to give two references to our protocol files, one to S3 and one to IPFS. Batching transactions would still be done only by IOHK. The node would try to get the file from IPFS first, if it fails it would try IOHK CAS (currently S3), if both fail, then the node should stop and report the error. In this way, the system would be less dependent on IOHK's existence because anyone could first post the file in IPFS and then send it to us. In this way, even if IOHK disappears, the files could be supported online by other entities making the DIDs generated truly persistent. The system would still remain functional without IOHK. This may also require less resources from IOHK as IPFS could reply to queries that would go to S3.
  - Unfortunately, the above idea still does not remove the possibility of IOHK performing a late publish attacks because IOHK could still post references to self generated blocks or by modifying Atala block sent by a user and posting a reference on chain of a subset (or superset) of the block provided and not posting the file itself.
    - We may be able to mitigate the subset case by requesting users to sign the list of all operations they intend to batch in a single block (assuming all operations batched by a user are signed by single DID, which is the case of issuing a barch of credentials). Then we could request that signature in the batch as an integrity requirement. This would not allow IOHK to publish a subset of the batch.
    - Note (as a consequence of the above point) that if we restrict batches to only contain events created by a single authority (i.e. all events in a batch are signed by the same key), we could request that the batch file must have a signature of the same key. This removes any possibility for IOHK to perform late publish attacks on other users data.
- A BFT CAS system that provides proofs of publication could be a great solution for us, but may be complex (if at all possible) to implement.
- Allow legally bounded entities to also publish files could aid towards a semi-decentralized system.
- Once we implement rollbacks/history rewrites we could consider further decentralization if we see the need, but it would bring the possible auditability details or the issue with non-revocable credentials mentioned in the document.

Long story short. If we handle history rewrites, we can guarantee consensus of all nodes about the *current* state of the system. However, if we implement a way to remove late publication possibilities, we would also get consensus about the *past history* of the system. Different use cases may require different approaches.

9 Improvement proposals

This document contains ideas we would like to evaluate for future releases.

9.1 Make use of the underlying Cardano addresses

Our operations require signatures that arise from DID related keys. Apart from this signature, an operation is attached to a transaction that will also contain a signature related to its spending script. We *may* be able to reduce the number of signatures to one.

Sidetree has update commitments (idea that probably comes from KERI1) 2. The idea is that, during DID creation, the controller commits to the hash of a public key. Later, during the first update, the operation must be signed by the key whose hash matches the initial commitment. The update operation defines a commitment for the next operation. These keys are part of the protocol and are optionally part of the DID Document.

Now, here is an improvement we could use:

- The initial DID could set a commitment to a public key hash PKH1 (as in KERI and Sidetree). We call this, the genesis commitment.
- Now, in order to perform the first update, the controller needs a UTxO to spend from. Let the user receive a funding transaction to an output locked by PKH1. Let us call this UTxO, U1.
- The update operation could now add metadata in a transaction that *spends* U1. The transaction signature will be required by the underlying nodes, meaning that we could get signature validation for free. This transaction could also create a new single UTxO, that we could name U2. This output, could again be a P2PKH script representing the next commitment.

We could define similar "chains" of transactions for DID revocation, credential issuance (declared in an issuer DID), and possible credential revocation registry.

Some observations of this approach are:

- Positive: We get smaller metadata.
- Positive: Light node ideas described in the section below.
- Positive: We may be able to get "smarter" locking scripts for operations relaying on the underlying chain scripts.
- Negative: It becomes more complex to do "on-chain batching" for DID updates. We may be able to use a multi-sig script for this. Credential batching remains unaffected. Recall that on-chain batching may be considered bad behaviour, see metadata usage concerns
- Negative: One would need to do the key/transaction management a bit more carefully. The key sequence could be derived from the same initial seed we use.

9.2 Light nodes and multi-chain identity

We are having conversations with the research team and Cardano teams to suggest a light node setting for PRISM. Assume we could: 1. Validate chain headers without downloading full blocks. 2. Find in each header an structure like a bloom filter with low (around 1%) false positive response that tells if a UTxO script has been used in the associated block. Note: our proposal is to add a hash of a bloom-like structure and not the filter itself.

With those two assumptions, if a light node has the full chain of headers, then given a DID with its initial state, the node could:

1. Check all the bloom filters for the one that spends the initial publish key hash we described in the previous section. If no bloom filter is positive, then the current DID state is the one provided. If K filters return a positive response, we could provide the script to a server which should provide either:
    - A transaction (with its metadata) with its merkle proof of inclusion in one of the positive blocks, or

- The actual blocks matching to those filters so that the node can check that all were false positives.

The addition of the bloom filter is important because it mitigates the typical SPV problem of a node hiding data.

We may be able to follow a similar process for credential issuance and revocation events. We could also check on real time for updates based on the headers for those DIDs for which we already know about. For this process we just need to know a genesis commitment.

We should check for possible spam attacks. E.g. providing an address with many matches. We have been referenced to section `TARGET-SET COVERAGE ATTACKS` and this attack on Ethereum filters. Given the small size of Cardano blocks, we should evaluate the number of UTxOs and transactions an attacker should create to saturate the filter. It should also be compared the optimal relations of filter size w.r.t. block size.

9.2.1 Bonus feature - multi chain DIDs

With a "light node" approach, we may be able to move from one chain to another while only downloading headers. This may be achievable by posting an update operation that declares a change of blockchain and then search for the desired script in the other chain.

We are not aware of blockchains that implement bloom filter like structures (or their hashes) in headers for UTxO scripts at the time of this writing. We haven't explored yet how to translate this approach to account based systems like the Ethereum case.

9.3 Layer 2 batching without late publication

To be expanded - The basis is to register *publishers* that could batch operations. The publishers could be whitelisted by IOHK. - DID controllers could send an on-chain event registering to a publisher (useful if they plan multiple updates to the same DID). The registration to a publisher could be part of the initial DID state too. - From that point on, the associated updates for that DID could only be posted by the publisher. - The publisher must publish file references signing the publication with his DID. - If a publisher does not reveal a file `F1`, and publishes a file `F2`, then no node should process `F2` until `F1` is published. This prevents late publication because no DID is attached to many publishers. - The controller can decide to post a *de-registration* event at any point. It will be interpreted only if all previous files are not missing. - If there is a missing file and the controller is still registered to the publisher, then its DID gets stuck. This seems to be, so far, the only way to avoid late publication. - We have evaluated other ways to handle the situation where the publisher does not reveal the file. For example, to allow controllers to post on-chain the operation they tried to batch. The complexity arises because we should assume that the controller and publisher could collude (they could even be the same person). - For example, we thought about having a merkle tree hash published with the file reference, to allow controllers to move on in case a file is not revealed. However, the publisher can simply decide to post a different merkle root hash or even to never provide the inclusion proof to the controller.