Self-Sovereign Distributed Identity Framework

@laugan

Abstract

This work presents a possible technical solution to Digital Identity problem for the Free TON ecosystem. This paper's goal is to describe technical and business aspects of a new Self-Sovereign Identity solution and present a motivation of made choices.

Introduction

It is known that Internet was made without any mechanism of identification in its core. The results of this flaw was a creation of lots of different ID systems: vendor-made (as Oracle, Apple, Google), government-made and many more. The one thing that unite them all - none of them became a really worldwide solution that can pretend to be your Web Identity. All of them are too specific or relying on some products and subscriptions.

More to say, the entire concept of each service each time asking for your data, storing all these data and eventually hacked or sold with all your personal data becoming public - seems very unintuitive.

Blockchain came with new solutions to this problem. A Self-Sovereign Identity concept where you can prove your existence and existence of your credentials by means of cryptographic proofs, without higher authority that should hold and manage all your data.

The overall idea of SSI is that user can not only prove his will to share or not share something, but also has a control on permissions to his credentials. This control is done by user deciding what credentials's issuer can show and what cannot.

When it comes to Free TON, it is obvious that this blockchain also can benefit from having its own SSI implementation because of its blockchain speed, low fees and overall decentralization of Governance that makes Free TON a more trusted network than more centralized ones. This is a crucial part of SSI and will be a good selling point for Free TON.

Also, from the technical side of view, the important thing is a Free TON's method of address calculation that was used in TIP-2,3,6,7,21,31 and other proposals. Its benefit is that if,for example, smart contract code and public key is known to verifier entity, the sole existence of that smart contract is a proof in itself. No other proofs needed. That makes some proofs in SSI completely off-chain (or at least free).

Use Cases List

Before talking about implementation, it is useful to imagine use cases that Self-Sovereign Identity solves. It comes with plenty of use cases, here are only a subset of all applications:

Identity Management	SSI can be used for storing reputation, data, some assets.
Disclosing Data	SSI system enable you to control when and what you permit to disclose to third parties.
Digital Signature	Your DID is a signature that can be used for a secure signing of different transactions, documents and so on.
On-chain Identity	You can sign in to different smart-contracts through using your DID contract if it's supported by smart-contracts.
Know Your Customer	You can use Identity to prove your off-chain credentials to third parties
Access Control	Identity can be used to implement some roles and permisssion.

To fulfill these use cases (or better say objectives) we should design a system where DIDs (digital identities) are not made only as a certain role but are role-agnostic and can take any role that they want currently.

System Overview

A proposed SSI system consists of a set of smart contracts in a Free TON blockchain. These core contracts manage all interactions between entities.

The overall process goes like this:

- Firstly, some request is made between parties in an off-chain environment (for example, I want new driver license from Department or some Bank wants to check my Digital ID).
- As request exists, Verifier party goes to Free TON blockchain and through address resolve via provided keys ensures that such DID exists (can be resolved with contract code and keys)
- After receiving request and getter checks (for example, request for issuing new credential), core contracts are making all the work of interacting between DIDs under the hood and then emit external outbound event to off-chain decision-maker (Issuer in this example).
- Event is listened by Event Consumer
- As the fact of the external outbound event is the proof in itself that all other checks are made, decision-maker can make request to issue Credential to some Holder (this is done by core contracts too) or Prover can give Permission to Verifier.
- If decision should be done in Client App, User should accept Sharing of Credential
- After decision is done, control backs to Free TON blockchain where info is written

- If some Credentials should be shown, they are presented through API on Event Consumer

That's the main sequence of actions. The only part that is out of scope of this document is a secure credential data storage that stores original credential that is shown on API of Event Consumer.

System Goals

This solution was made with the following goals:

Cheap Usage	To achieve this, "proof of address resolve" is used because it can be done with free getter methods.
Scalability	As all contracts will be made in distributed way and no large mappings or complex contract trees are proposed - we believe that the solution is verys scalable. Also, the concept of universal Identity contracts for all parties makes the solution ready for a more complex approaches.
Interoperability with other SSI solutions	We propose to implement a support of W3C Decentralized Identifier Specification v1.0 standard (as described here: https://w3c.github.io/did-core/#fragment). This will ensure a max interoperability with client apps that support this standard.
More On-Chain Steps	We believe that in the end, all steps of SSI should be made on-chain. Currently, we propose to make on-chain all interactions except Prover-Verifier initial connection and Issuer-Verifier exchange of permitted credentials (as credentials are stored off-chain)

System Parts

Logical Entities

We can divide our system into 5 logic entities:

- **Issuer** - entity that issues some Credential to Prover
- **Verifier** - entity that wants to check that Prover exists and have som Credential
- **Prover** - entity that has Credential and can give Permission to id to Verifier
- **Credential** - set of personal data or just a signature or Id of personal data that is issued by Issuer to Prover

- **Permission** - an explicit proof issued by Prover to Verifier, that Verifier can view some Credential in Issuer storage

First three are represented in the SSI as the same contract - Identity, because each Identity can take not only one role, but all of them or any two.
Credential is represented by Credential contract and an off-chain storage that is a property of the Issuer.
Permission is represented by Permission contract that can be manually or auto-revoked (with self-destruct of contract and return of TON funds).

Technical Components

From technical point of view, solution consists of three main layers:

- **Smart Contracts** - core part that makes all interactions and emits results as outbound events;
- **Event Consumers** - servers (or listeners) that handle incoming requests from Free TON. There requests contain info about credentials that need to be issued or permission that should be granted;
- **Client Apps** - end-user apps that are used to grant permissions (and in future to issue some custom credentials too). They receive info from Event Consumers to accept requests by calling Free TON smart contracts.

Limitations

We think that this system will be much more anonymous and private with two additional layers on top of it:

1. ZK-Proofs that will be used to hide interactions between entities.
2. Pseudonym-based DIDs instead of using public key/owner-based DIDs,for example through the use of Camenisch-Lysyanskaya signature scheme (https://www.iacr.org/archive/crypto2004/31520055/cl04.pdf)

Still, we don't see a cheap way to implement this currently as ZKP solution is not yet fully available in Free TON and Pseudonyms don't fit to the main benefit of Free TON - address resolve from initial contract data.
This work is a draft and we plan to work on this aspect more in the next version.

Smart Contracts

There is only three contracts in a proposed SSI system:

1. **Identity** - this is a contract of DID and there is no difference if it is a User, an Issuer or Verifier. This contract can be either of these or all-in-one - no difference.
2. **Credential** - deployed by any Identity by calling ***issue()*** method. This contract has static values of issuerDID, holderDID, attributeName. Also important that aside from holding attributeValue,
3. **Permission** - can be deployed only by credential holder Identity by calling permit() method. This contract has static values of ownerDID, viewerDID, credentialAddress. The important point is that ***permit()*** method accepts ***(viewerDID, issuerDID,***

attributeName) arguments and resolves Credentials' address internally, so there is no way to permit foreign credentials.

DID or Identity Contract

Identity Contract is a contract that is used by all system participants. There is no difference between identities of Credential issuer or Verifier. All of them are just Identity contracts in SSI system.
Because of this, there is no artificial separation of roles. Even if you're student - you can issue Credentials (perhaps you need them to approve some new guys to your Secret Illuminati Club in your University, who knows?).
If you're Issuer (for example, some Government Department) - you still can be Verifier and ask for credentials of another departments.
So, it's vital for all Identities to have same Identity contract for all SSI entities.

Static data (Used for address calculation)

- static owner(public key)

Issuance functions and events

- function requestIssue(externalRequestId) // client's app starts with requesting issuance of credentials
- function issueRequested(externalRequestId) // checks that sender address resolves with code+pubkey
- event eventIssueRequested() // makes external outbound message that Issuer's Event Consumer listens; uses externalRequestId as external address
- function issue(externalRequestId, DID, attributeName, attributeValue); // if Issuer decides to issue, he calls this

Permission functions and events

- function requestPermission(externalRequestId) // client's app starts with requesting permission for credentials
- function permissionRequested(externalRequestId) // checks that sender address resolves with code+pubkey
- event eventPermissionRequested() // makes external outbound notification that can be listened and read by User's app; uses externalRequestId as external address
- function permit(externalRequestId, DID, attributeName, attributeValue); // if User wants to permit, he calls this method

Revokation functions and events

- function revoke(externalRequestId) // if User wants to cancel permit, he calls this method

Credential/Claim Contract

Each Credential contract contains “attributeValue” field with a content of credential, but it is not static (not used for address calculation). Attribute can be modified later by issuer.

The storage and level of cryptography of this value is a wide discussion that we can’t fully describe in this document. So, most of the time it will store only hash or signature needed to find corresponding data in Issuers storage, not the credential itself.

Security of this contract made through checks on who can deploy this contract (msg.sender equals to issuerDid static var).

Content Data Model
To improve interoperability, we need to implement guidelines from:
https://www.w3.org/TR/vc-data-model/#core-data-model

Static data (Used for address calculation)

- static issuerDid // Identity address of Issuer, used to check who can deploy this contract
- static credentialId ; = hash(
- + static holderDid // Identity address of Holder
- + static credentialName // unique name of this credential in the Issuer’s. For example, if one issuer makes both passports and driver licenses, you should specify here, what credential it refers to)

Thus, each Credential contract is in fact a Key-Value store that doesn’t specify anything about real Users that have relation to this Credential. Its Key is hash derivation that can’t reveal anything if you don’t know who you are verifying in advance.

Issuance functions and events

- event eventIssued() // makes external outbound message that User’s Event Consumer listens; uses externalRequestId as external address

Permission functions and events

- function permitted()
- event showPermitted() ; uses externalRequestId as external address
-

Revokation functions and events

- function revoked()
- event hideRevoked() ; uses externalRequestId as external address

Permission Contract

Security of this contract made through checks on who can deploy this.

Static data (Used for address calculation)

- static permitterDid // Identity address of Holder, used to check who can deploy this contract

- static permissionId = hash(
- + static issuerDid // Identity address of Holder
- + static verifierDid // Identity address of Holder
- + static credentialName // unique name of this credential in the Issuer's. For example, if one issuer makes both passports and driver licenses, you should specify here, what credential it refers to)

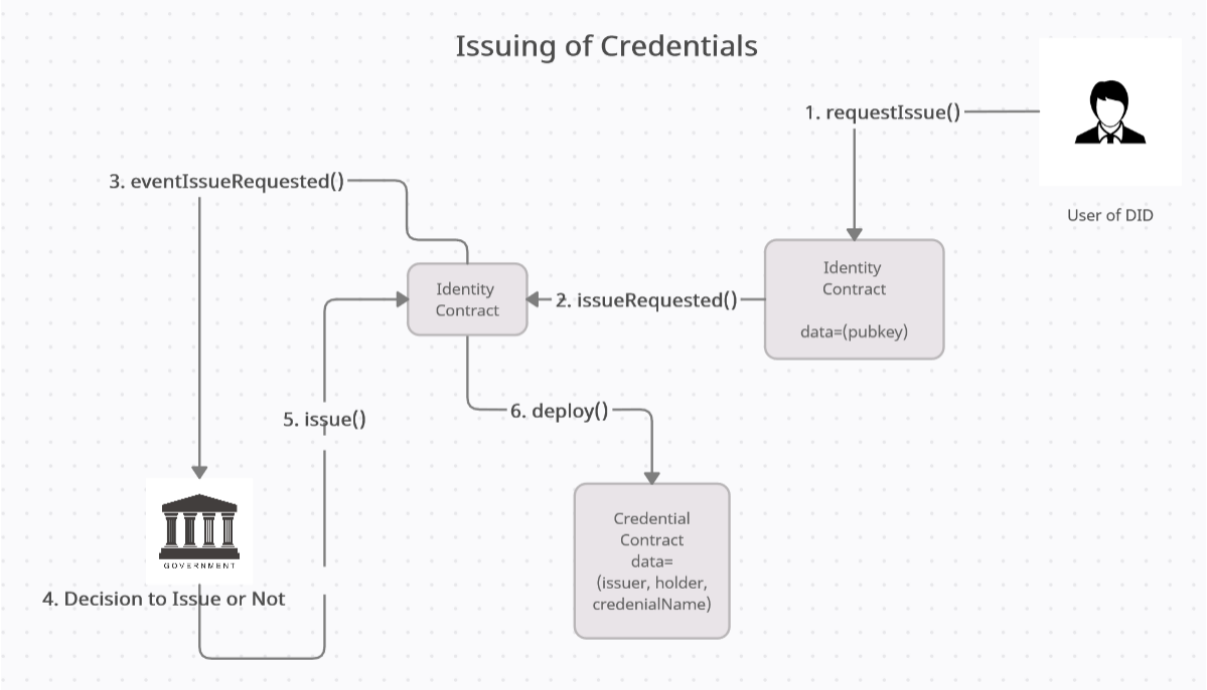
Permission functions and events

- function permitted()
- event showPermitted() ; uses externalRequestId as external address

Revokation functions and events

- function destruct() ; method to destroy this conract and return its balance to Permission issuer (Prover)

User Journey I - Issuing Credentials

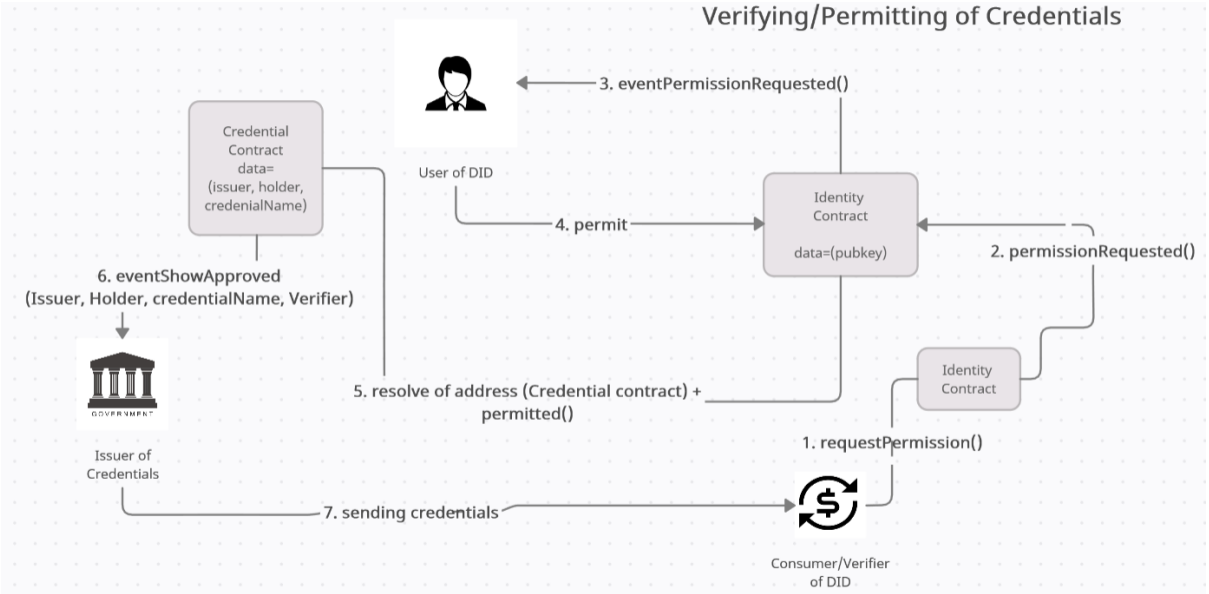


1. Bob (User) browses to some web banking service (Verifier) that requires the User to show its Dpassport credential, issued by Government.
2. Bob doesn't even have DID and no Dpassport by Government
3. Firstly, Bob deploys DID with his public key
4. Then, he requests Government to issue him a credentials, calling a method from his own Identity contract - ***requestIssue(issuer=GovernmentDID, credentialName=DPassport)***;
5. This method calls Government Identity contract ***issueRequested(holderPubkey=BobPubkey, holder=BobDID,***

credentialName=DPassport) method. It checks if msg.sender is BobDID and it can be resolved from pubkey.

6. Government Identity contract emits external event - ***eventIssueRequested(holder=BobDID, credentialName=DPassport)***. After Government service decides that it can issue credentials, service calls ***issue(holder=BobDID, credentialName=DPassport)*** method that deploys a Credential contract with (Issuer, Holder, credentialName) as data

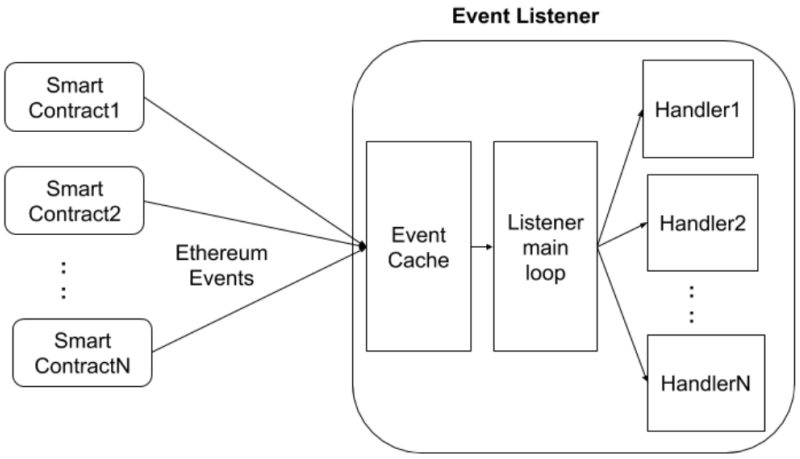
User Journey II - Veriying and Permitting



7. Bob returns to Bank and asks for passing the KYC, he presents his DID as a message signed by key pair.
8. Bank's app resolves DID contract by using public key. That is made off-chain with offline resolve from contract code and key, then free getter method is called to get DID info. That is called "proof-of-address-resolve" if such contract exists.
9. Bank then calls ***requestPermission(issuer=Government, credentialName=dPassport)*** on his DID address
10. This method calls Bob's Identity contract ***permissionRequested(credentialName=DPassport)*** method. It checks if msg.sender is BobDID and it can be resolved from pubkey.
11. Bob's Identity contract emits external event - ***eventPermissionRequested(issuer=Government, credentialName=DPassport, verifier=Bank)***. After User decides that it can permit viewing Credential, his app calls ***permit(issuer=Government, credentialName=DPassport, verifier=Bank)*** method that deploys a Permission contract with permissionId (hash of all involved) as data.
12. If Bank not only wants to check existence of contract (it can be done off-chain now), but also wants to see contents of Credential - it calls Permission contract, resolving it by hash and calls ***show()*** method.

Event Consumers

The goal of Event Consumers is to listen to happening events.



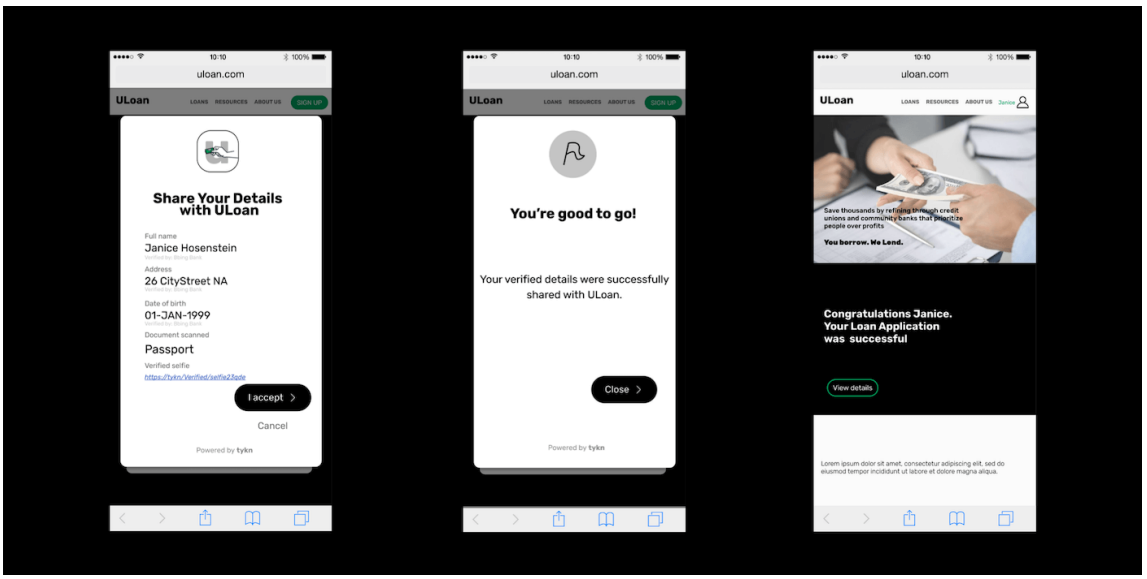
Event Consumers is the server application for Issuers and Verifiers. They can be written in a desired language.
List of mandatory functions:

- handler of eventIssueRequested event
- handler of showPermitted event
- handler of hideRevoked event
- API for credentials presentation with respect to W3C standards on Data Model
- API for sign-on requests with respect to W3C standards on DID requests

Handlers can be automatic or made with additional delegation to manual operators (for some sort of Issuers).
Also, Event Consumers can be split on Issuer and Verifier implementation to not implement all APIs in one project.

Client Apps

The draft client app can be made as a DeBot or as a simple web app that make requests to sign-on API.



GUI of Tykn SSI Mobile & Web App

Mobile App should support:

- Accepting notifications from Event Consumer that someone requested permission
- Opening W3C standard URI links to DID invites from Verifiers
- Opening QR-code images with the same links
- Showing current requests for credentials
- Showing “History of Permissions”
- Showing “Revoke” button

Technical Requirements

- JVM- or JS- based languages
- Support of Push notifications
- Support of TON SDK bindings

Contacts

- **Telegram ID:** @laugan
- **Wallet address:** 0:fd080fb5fc9266226ec59b062f0cdde85c818ef1d4ac4939804ee7616ec352f4