POLITECNICO DI MILANO

Faculty of Industrial and Information Engineering

Master in Computer Science and Engineering

Pistis, a Credentials Management System based on Self-Sovereign Identity

Relatore: Prof. Francesco Bruschi

Master Thesis of:

Sinico Matteo Matr. 898773

Taglia Andrea Matr. 898733

Anno Accademico 2019 - 2020

Politecnico di Milano:
www.polimi.it

Scuola di Ingegneria Industriale e dell'Informazione:
www.ingindinf.polimi.it

Contents

List of Figures

List of Tables

Listings

Sommario

Viene definita "credenziale" uno o più attributi legati a un'entità (persona, organizzazione o cosa) che attestino una qualità, un aspetto, una qualifica o un risultato dell'entità in questione. In particolare una credenziale è composta da tre blocchi principali: l'attore che la rilascia, il soggetto della credenziale e infine il o gli attributi asseriti dalla credenziale stessa. Le credenziali nascono e tuttora vengono utilizzate principalmente nel loro formato cartaceo, non senza problemi di contraffazione, portabilità e interoperabilità. La loro versione digitale ha acquistato popolarità da quando l'informatica è diventata di uso comune e mira a migliorare gli aspetti negativi del formato cartaceo.

Tuttavia, vi è una forte necessità di una soluzione adeguata per gestire tutti i diversi formati di credenziali attualmente in uso. In particolare, sono state avviate numerose iniziative a livello europeo per trovare una soluzione inter-operabile, scalabile e decentralizzata per i sistemi basati sulle credenziali in ambito accademico. Nessuna di queste iniziative ha avuto l'impatto desiderato per standardizzare il modo in cui vengono gestite le credenziali accademiche.

La blockchain è caratterizzata da grandi promesse di decentralizzazione. Sono in discussione nuovi paradigmi per la gestione delle credenziali e, in termini più ampi, della gestione dell'identità digitale basata su questa nuova tecnologia, tra cui il più significativo è la cosiddetta "Self-Sovereign Identity". Lo scopo ultimo di questo nuovo paradigma è quello di dare la possibilità agli utenti di avere il pieno controllo sui propri dati, non solo autorizzando chi può vederli o trattarli ma soprattutto possedendoli fisicamente sul proprio device. È in questo contesto, e sulla base del framework SSI, che il W3C sta mettendo a punto un nuovo promettente standard basato sul concetto di Identificatori Decentralizzati e Credenziali Verificabili.

Il seguente lavoro di tesi si basa su questi standard e propone un nuovo sistema di gestione delle credenziali basato sulla blockchain Ethereum, chiamato "Pistis". L'obiettivo è quello di essere un sistema veramente usabile, scalabile, decentralizzato e inter-operabile.

"Pists DID Method" è stato ufficialmente revisionato e accettato dal gruppo W3C responsabile di questi nuovi standard.

Abstract

"Credentials" are defined as set of attributes linked to an entity (either person, organization or thing) to assert a quality, aspect, qualification or achievement related to the background of the subject entity. A credential is made of three main blocks: the attesting actor, the subject of the credential and finally the claim itself. Credentials have historically being used in paper format, not without issues of counterfeit, portability and interoperability. The digital version of their paper-based counterpart gained ground since IT became mainstream and aims at improving the downsides of the paper format.

Still, there is a strong need for a suitable solution to handle all those different credential formats currently in use. Specifically, there have been many initiatives run at European level to find an interoperable, scalable and decentralized solution for credential-based systems in the academic environment. None has had the desired impact to standardize the way academic credentials are handled.

The blockchain comes with great promises of decentralization. New paradigms for credentials handling and, in broader terms, of digital identity management based on this new technology are being discussed, the most notable being "Self-Sovereign Identity". It involves giving ultimate, sovereign control to users over their data. It is in this context, and based on the SSI framework, that the W3C is finalizing a new, promising, standard based on the concept of Decentralized Identifiers and Verifiable Credentials.

This thesis work builds on top of those standards and proposes a new Self-Sovereign Identity based credentials management systems running on the Ethereum blockchain: "Pistis". It aims at being a truly usable, scalable, decentralized and interoperable system.

"Pists DID Method" has been officially reviewed and accepted by the W3C group responsible for those new standards.

Introduction

0.1 Problem Framing

Mobility is increasing all over the world. The need for people to move around is rapidly growing and the issues that airplane companies are having in catching up with the increasing demand is just one of the many figures that can be explored to endorse this point[1]. In such a context the sharing and validation of different documentation is becoming more and more important.

The European Union has been trying hard to take down barriers and obstacles to the free movement across the Old Continent. However, a student/citizen who wishes to continue their studies at a foreign institution or to apply for an interesting (foreign) job is invariably faced with the challenge of **sharing** the obtained or required **diploma** and having accepted as **authentic**. This largely paper-based procedure results in a great deal of **additional work** for the student/citizen and inconvenience for the administration involved, without it strengthening confidence in the document delivered or limiting potential fraud. The CEF (Connecting Europe Facility) organism has recently introduced EBSI (European Blockchain Service Infrastructure), that is aiming at building a Proof of Concept about digital identity in the first quarter of 2020 [17]. These efforts have as well recently been undertaken by different countries oversees: Singapore has started issuing certifications to students [25], Canada is launching its own digital identity system [4]. Many are actively exploring the potential for Blockchain to provide a decentralized, transparent, validated and indelible management of student/citizen identity linked data (either diploma, or any other form of documents and qualifications).

Different options and certificate based models are currently in place, but all of them fail at addressing the main pain point, that is **absence of trust** in complex ecosystems such as the international Academic environment. **Lack of decentralization** is the most relevant factor to blame for.

We believe Self-Sovereign Identity model backed by the fast developing blockchain technology is the direction to take in order to find a suitable solution to a decentralized, reliable and inter-operable system.

0.2 The Need for Such a Solution

We cite some of the most recent councils and work group which, at European level, show the great interest by the European countries in this new paradigm of

[1] https://www.iata.org/pressroom/pr/Pages/2016-10-18-02.aspx

verifiable qualifications.

- Paris Communiqué (EHEA)[10]:

  > We also urge the adoption of transparent procedures for the recognition of qualifications, prior learning and study periods, supported by inter-operable digital solutions

- Council recommendation on the Automatic Mutual Recognition of Diplomas and learning periods abroad (European Council):[49]

  > Hereby welcomes the Commission's intention to: ... Explore, in cooperation with Member States, the potential of new technologies, such as blockchain technology, to facilitate automatic mutual recognition.

- (Draft) Global Convention on the Recognition of Qualifications concerning Higher Education (UNESCO):[48]

  > SECTION III. BASIC PRINCIPLES FOR THE RECOGNITION OF QUALIFICATIONS CONCERNING HIGHER EDUCATION Article III ... 8 . Parties commit to adopt measures to eradicate all forms of fraudulent practices regarding higher education qualifications by encouraging the use of contemporary technologies and networking activities among Parties.

There are few underlying open questions which have not been given a proper answer yet:

- How to overcome the lack of interoperability between systems to unlock much needed cost efficiencies?
- How to build a single source of truth to prove the veracity and provenance of people's qualifications?
- How to incorporate people consent to unlock the sharing of data?

Our Pistis Self-Sovereign Identity system, pairing with blockchain technology might have the answers to those, it is the thesis we are supporting in this document.

0.3 Thesis Contribution

We analyze existing systems whose aim is the handling of Credentials (also known as Attributes, Claims, Qualifications) and the way they tackle the verification of authenticity of those. Especially we deal with academic type of credentials, or courses attendance credentials, that is all those information whose aim is to prove one has attended some course/module and has achieved some results.

Firstly we cluster the possible approaches, as far as we could know of, into some specific Credentials Management Model. Each model is described giving a concrete

example of where it is used. We decided to focus on academic credentials to be able to concretely exemplify our methodology and to consider the details thoroughly.

We then explore the Self Sovereign Identity model (SSI for short), a user-centric approach to the credentials management problem which exploits the blockchain technology to ensure transparency and third party independence in the credentials handling processes.

We conduct a **survey of the models outlined and compare them** to the Self Sovereign identity model. We point out how it is arguably better from different points of view as we have analyzed in the Models Comparison section.

Comparison with alternative implementations of the SSI models is carried on.

We give a detailed specification for our **SSI model design Pistis**, built on top of the W3C Verifiable Credentials and the W3C Decentralized Identifiers standard. We detail the specification of Pistis DID Method adhering to the standard. We give a reference implementation built on top of some basic libraries from the SSI based project uPort, from Consensys team, creators of the popular Ethereum blockchain. and try out few complete use-cases to see the system working in practice. We comment on the results and criticalities of our solution.

0.4 Thesis Structure

Next chapters are organized as follows:

- In Chapter 1, we explore existing Credentials Management Systems clustering them for a fair comparison against the Self Sovereign Identity model. We show how the SSI model would outperform the others.
- In Chapter 2, we present our solution.
- In Chapter 3, we describe the building blocks which will be necessary to understand Pistis architecture in the following chapters.
- In Chapter 4, we expand the previous chapter giving specification of the Pistis system built on top of the standards explained.
- In Chapter 5,We present Pistis's architecture and the processes happening within the system.
- In Chapter 6, we report the details of Pistis components we have developed.
- In Chapter 7, we discuss Pistis limitations alongside compliance with current regulations.
- In Chapter 8 we look at other SSI solutions.
- In Chapter 9, we explore what yet has to be done as a future work and we conclude the document with a retrospective on the work done and on the vision we have of the near future.

Chapter 1

State of the art

We looked for different models to implement credentials handling, i.e. Credential Management Models. We explored some currently employed solutions and we clustered them under the same group when characteristics are similar.

Firstly the models are explained, bringing along a real world implementation and use case. Then the SSI approach is described, with a clear description and example of how it would work. A detailed comparison of the different Models has been carried on. Finally some SSI compliant certificate-based system, already existing are analyzed highlighting what we believe should be changed or improved.

1.1 Known Models

The models are described analyzing how each of them carry out the following processes, which we believe are the necessary processes to have a fully working Credential Management System.

- Request and Issue a Certificate
- Share and Verify a Certificate

1.1.1 Paper Model

Under the Paper Model we have clustered any model that involves the use of credentials in paper format. Paper certificates are indeed issued in a physical form. The delivery usually happens during a physical meeting between the credential issuer (or someone on behalf) and the recipient of the credential, or involves quite expensive cost of physical mail expedition. Scanning the document to gather a digital version of it is often not possible due to the loss of originality of the document. Authenticity of such credentials is granted by mean of handwritten signature or stamps. There is a rich literature about how a handwritten signature could be verified to prove its authenticity[12][50][39]. Sometimes just the handwritten signature or the stamp are not enough and the verification process could involves the verifier to get in contact with the issuer to double check the actual issuing of the credential and that its content has not been tampered with. This usually happens by an agreement via email, or through phone calls, etc...

Polimi Use Case

Politecnico di Milano allows students to require the issuing of different official documents, ranging from Transcript of Record to the official diploma. These are only available upon request and are all released in paper format. Either a signature or a stamp, or sometimes both, are applied to the documents to attest the validity of the issuer. The gathering of these documents implies either that the student shows up in person at the student desks or that Politecnico mailing them. The sharing of these credentials, as it could be a Transcript of Records to migrate and validate exams across different universities, involves the physical meeting of the student with the foreign university, or the mailing of the document. The verification process need the universities to be in contact or at least having established an agreement in order to validate the credential.

This whole process is extremely time consuming and cost inefficient as it will be better shown in the Models Comparison section.

1.1.2 Digital Signature Model

Under the Digital Signature model we have included any model which involves a digitally signed credential. A digital signature is a cryptographic scheme used to verify the authenticity, the integrity and to guarantee the non repudiation of digital messages or, more broadly, documents. A valid digital signature certifies that the message was created by a known sender (authentication), and that the message was not altered in transit (integrity).

It requires a Public Key Infrastructure (PKI) and Certification Authorities (CA) to be fully functioning. The former involves all the pieces needed to create, distribute, verify, store and revoke digital certificates. The infrastructure rotates around the concept of asymmetric key encryption, where a CA issues certificates attesting that a certain public key belongs to a certain individual or organization (i.e. what is later on referred to as "Entity").

The digitally signed credential, indeed, can be issued in digital format and redeemed by the recipient. From now on, the recipient is also the ultimate holder of the credential and he/she is able to freely share it using his preferred sharing tool (e-mail, social media, messaging apps, etc.). Whoever receives a digitally signed credential can verify it by means of a digital signature verification software.

UNIBO Use Case

The University of Bologna (Italy) issues certificates that are digitally signed and can be downloaded as PDFs. It also allows to print the document with a digital stamp on it. This digital stamp is simply a two-dimensional code (as a bar code or a QR code) that contains the digitally signed version of the original document. The result is a PDF file containing the certification in plain text, plus the two-dimensional code used to verify authenticity and integrity of the credential. This method allows to have a paper version of a digitally signed document. [2]

NTNU Use Case

To release the Transcript of Records to an Erasmus student, the Norwegian university NTNU uses a third party service called diploma registry[1]. This service allows the student to share their transcript of records or their certificate digitally signed by the NTNU university with anyone he wants to, for a limited period of time.[2], use the following code when prompted: 439367.

Given that the owner of the credential can redeem his own digitally signed TOR at any moment and share it in another way, not depending on the third party service, this is a digital signature use case.

1.1.3 Self Hosted

Any model where the credential is not digitally signed and is held by the issuer of the credential itself. The issuer is responsible for securely storing and making the credential available. This last part is crucial and divide this model into two different categories: private, where the credential is shared just with whom the owner of the credential has given the permission to; public, where the credential is accessible to whoever has access to the registry of the credentials.

Here all the processes are strictly dependant on the issuer itself. The issuing procedure is carried out by storing the credential in the issuer's storage. Is up to the issuer to ensure security, privacy, availability and persistence of the data stored. The actual owner of the credential doesn't really own the credential. Indeed, to share one of them, he relies again on the issuer, which has to make the credential available to whoever the owner wants to. Who needs to verify such a credential has to trust the issuer.

Oracle - private

Oracle allows anyone who has made a certification at Oracle to share his certification by email. In the email, like in the NTNU case, there is a link with an access code to an ORACLE web service which shows you the credential. The link has an expiry date past which it is no longer possible to access to the web service to verify the certificate.[3], use the following access code when prompted: Cr112719At3y

Differently from the NTNU case, here the certification is not digitally signed, hence the only way to verify it, is through the ORACLE web service itself.

PMI - public

PMI (Project Management Institute) certifications are publicly exposed at the following link https://certification.pmi.org/registry.aspx. Here anyone can see who has attained which certifications just by searching for the initials.

[1] Here there is the link to the diploma registry https://www.vitnemalsportalen.no/english/

[2] You can find an example of it at the following link https://app.vitnemalsportalen.no/vp/showResultFromLink/en/53F8FBB51EF7456D8A6FF3F1041B277A

[3] You can find an example at the following link https://catalog-education.oracle.com/pls/web_prod/ocp_interface.certification_authorization?arg_key=QGZFB112647eOMb404zQFJ&p_org_id=1001&p_lang=US

1.1.4 SaaS (Software as a Service)

Any model where the credentials are not available through the issuer itself, but through a third party service. Differently from the Self Hosted models, here there is another actor, the third party service, which needs to be trusted.

This model is quite similar to the previous one. The only difference rely in the presence of a third party which is responsible of handling the credentials in place of the issuer. However this single difference is quite important in therms of inter-operability, integrity, and issuer verification as is further discussed in the model comparison section.

AWS

AWS uses a third party service, Certmetrics in particular, as Software as a Service to handle their credentials. At the end of the course they release a link where you can freely access and verify the credential and who is the owner.[4] Certmetrics is a third party service, which offers a solution to securely host certification of any kind in their own infrastracture. A credential issuer, like AWS in this case, can use their infrastructure to abstract away all the issues related to the credential storing, by using Certmetrics services.

1.2 Shortcomings of previous models

Previously described models expose some shortcoming which fail at making any of those the ultimate Credential Management solution. Speicifically, we identitified the following pain points:

- Issuer Verification: how can one be sure who the issuer of the credential is? How trustworthy is that system? Does the verification process need to go through one or more third parties?
- Portability: How can a credential be transferred elsewhere? How cheap and time that process is?
- Interoperability: how compliant is the model with different standards? Does it allow for automatic credential verification? (i.e. verification of some basic properties of a credential without user interaction)
- Consent/Sharing Permission: Users must agree to the use of their credentials. How to make sure the credential holder is allowed to hold and share it?
- Data Minimization: Disclosure of claims must be minimized. Many of those models fail at implementing selective disclosure of information.
- Persistence: Credentials must be long-lived. They have to live at least after you.

[4] You can find an example at the following link https://www.certmetrics.com/amazon/public/transcript.aspx?transcript=TPYP1KG2CFRQ1JG5

- Control: Users must have access to their own data without relying on third parties. There should not be the possibility to enforce censorship or any other form of control to block availability.

In the model comparison section we have analyzed each model against different criteria. In general current solutions to deal with personal data, suffer from lack of decentralization and high cost required to maintain all the necessary infrastructure to store those data in a secure, privacy preserving, persistent and broadly available way.

High cost are due to the increasing effort done by organizations to improve their services, to be resilient to data breaches but at the same time offering an user friendly experience. Decentralizing is a possible cure. Decentralization, indeed, would help avoiding data to become more and more centralized in huge silos controlled by single organizations with full control over our personal data. Beside that, these data silos are at high risk, due to the daily data breaches we are experiencing nowadays.

The following words well describe these issues and gives an anticipation of the solution we described in the following chapters:

> "None of us actually owns a digital identity. We simply 'rent' identities from each of the websites or apps we use, resulting in an inefficient, fraud-riddled, privacy-invading mess. Additionally, each organization we interact with must store our personal information in massive databases. These 'silos' become gold mines to hackers and toxic liabilities for anyone obligated to store the data. A siloed approach to identity may have worked in the early days of the Internet, but with practically every business and billions of people now online, problems such as fraud are growing rapidly. The costs of these problems will soon balloon as billions more identities come online with the Internet of Things. Regulators try to police misbehavior by dishing out billions in fines each year, but they don't address the root cause. Data breaches continue to occur almost daily, often because siloed identity creates massive troves of data attractive to hackers.
>
> Solving the identity silo problem begins with a digital identity that you literally own, not just control — a "self-sovereign" identity. When combined with verifiable claims, it enables any person, organization, or thing to interact directly with any other person, organization or thing, with trust and privacy. If anyone other than you can "pull the plug" or change the rules for your identity, it isn't self-sovereign, it is siloed – even if it uses 'blockchain' technology. True, globally scalable self-sovereign identity requires an open source, decentralized network which no single entity owns or controls. Until the advent of distributed ledger technology (DLT) this was impossible.
>
> Like the Internet, it is not owned by anyone: everyone can use it and anyone can improve it.
>
> Any person, organization, or thing can actually own their digital identity – not just control it – independent from any silo. Any person, organization, or thing can instantly verify the authenticity of "claims," including

> who (or what) something claims to be. Complete control of how, what and when information is shared, without added risk of correlation and without creating troves of breachable data."[21]

The solution they are highlighting is called Self Sovereign Identity. In the following section we introduce this approach.

1.3 Self Sovereign Identity approach

1.3.1 Introduction

Modern year have seen a rapid change in digital identities models. New technologies offer different and better ways to handle how personal data is dealt with.

> "One of the first references to identity sovereignty occurred in February 2012, when developer Moxie Marlinspike wrote about "Sovereign Source Authority". He said that individuals "have an established Right to an 'identity'", but that national registration destroys that sovereignty. Some ideas are in the air, so it's no surprise that almost simultaneously, in March 2012, Patrick Deegan began work on Open Mustard Seed, an open-source framework that gives users control of their digital identity and their data in decentralized systems. It was one of several "personal cloud" initiatives that appeared around the same time. Since then, the idea of self-sovereign identity has proliferated."[20]

> "The models for online identity have advanced through four broad stages since the advent of the Internet: centralized identity, federated identity, user-centric identity, and self-sovereign identity".[3]

These are words by C. Allen that defines himself on his Linkedin profile: "an entrepreneur and technologist who specializes in collaboration, security, and trust. As a pioneer in internet cryptography, he's initiated cross-industry collaborations and created industry standards that influence the entire internet. Though he's worked within numerous privacy and security sectors, Christopher's recent emphasis has been on engines of trust such as blockchain, smart contracts, and smart signatures, in particular decentralized self-sovereign identity."

Indeed, he has recently been very active in the field of decentralized identity and Self Sovereign Identity specifically.

In the following sections a definition of SSI will be given, and a comparison between an SSI credential-based system and the currently used credentials management systems is carried on.

1.3.2 Our definition of Self Sovereign Identity

There is not globally agreed definition of SSI, as stated by Christopher Allen[3] as well as Metadium[5], a well known team in the SSI credential-based systems space.

[5]Metadium, Introduction to SSI: https://medium.com/metadium/introduction-to-self-sovereign-identity-and-its-10-guiding-principles-97c1ba603872

Many have had a go with their own thoughts about this new paradigm, but still it lacks a well defined wordings. In order to be clear on what we believe falls inside this category and what not, we give our own definition below:

Under self-sovereign identity model, individuals and organizations (users) who have one or more identifiers (something that enables a subject to be discovered and identified) can present digitally signed claims relating to those identifiers and issued by some other identifier. The trustworthiness verification process of those claims is automatic, unequivocal and is done without having to go through an intermediary.

There must not be any single authority which the system depends upon, in this way, the system must ensure decentralization property.

Every user is the only owner of his own claims, no third party service are needed to hold credentials. This way the user is responsible for safe-keeping at least his own claims, he decides who wants to trust and with whom wants to share with his credentials.

> SSI begins with a digital "wallet" that contains digital credentials. This wallet is similar to a physical wallet in which you carry credentials issued to you by others, such as a passport, bank account authorization, or graduation certificate, except these are digitally signed credentials that can cryptographically prove four things to any verifier:
>
> - Who is the issuer
> - To whom it was issued;
> - Whether it has been altered since it was issued;
> - Whether it has been revoked by the issuer.
>
> You can also carry self-signed credentials in your wallet, such as your preferences, opinions, legally binding consent, or other attestations you've made about anything.
>
> Verifiable credentials can be issued and digitally signed by any person, organization, or thing and used anywhere they are trusted. SSI is as strong as the credentials it contains, strong enough for even high-trust industries such as finance, healthcare, and government. Organizations can choose to trust only credentials they have issued, credentials issued by others, or some combination, according to their security and compliance needs. [[40]]

Every SSI system should include 4 main processes which we will explore further in the processes section:

- Request and Issue a Claim
- Share and Verify a Claim
- Revoke a Claim
- Entity Resolution

Thus a system willing to adhere to the Self Sovereign Identity model has to handle the above mentioned processes in order to have the system fully working.

In the Appendix A you can find an example of a SSI qualifications.

1.4 Models Comparison

We defined some criteria to base the comparison upon. We have taken inspiration from the 10 principles that Christopher Allen proposed:

- Existence: Users must have an independent existence.
- Control: Users must control their identities.
- Transparency: Systems and algorithms must be transparent.
- Persistence: Identities must be long-lived.
- Portability: Information and services about identity must be transportable
- Interoperability: Identities should be as widely usable as possible.
- Consent: Users must agree to the use of their identity.
- Minimization: Disclosure of claims must be minimized
- Protection. The rights of users must be protected

For each criterion a mark from 0 to 2 has been given, represented with the following colors: 0, 1, 2. The results are summurized in the table 1.1

The meaning of each criterion is further explained below, alongside the motivation for the marking assigned at each model.

1. **Ensure Integrity:** if someone tampers with a credential, would you notice it has been tampered with? How trustworthy is this system?

   Paper systems suffer from the forgery phenomena which in the years has been a great issue. However, it is not easy and there are ways to make this harder, thus a mid rating has been given.

   Digital Signature and SSI use the cryptographic digest to ensure integrity. It is provable that the best way to hack a system like this is brute forcing the hash, which takes too long with current supercomputers, thus it has been given the best rating.

   On the other side, SaaS Credentials and Self Hosted have a mid rating because of the lack of ability to self check the integrity, indeed it is all about trusting the services not to tamper with the credential. Also the storage systems may be subject of hackers, thus again we also need to trust their storage systems other than the Service Provider itself.

2. **Issuer verification:** how can one be sure who the issuer of the credential is? How trustworthy is that system? Does the verification process need to go through one or more third parties?

   Paper systems, again, suffer from the forgery phenomena.

Attribute / Model	Paper	Digital Signature	SaaS	Self Hosted	SSI
Ensure Integrity					
Issuer Verification					
Ensure Validity					
Portability					
Interoperability					
Consent/Sharing Permission					
Data Minimization					
Persistence					
Protection/ Privacy					
Data Sovereignty					

Table 1.1: Models comparison

On the top row the Models are listed, while on each row the criteria are expressed and marks are given at each model. For each criterion a mark from 0 to 2 has been given, represented with the following colors: 0, 1, 2

Digital Signature use the PKI system to verify who the issuer of a credential is. This system is largely used nowadays and can be proved to be trustworthy. But you have to verify a third party CA chain, where the CA are responsible of providing identity proof, thus a mid rating is given.

SSI check the authenticity of the document by decrypting it with the public key. Then use a list of trusted contacts/issuer, to verify the association of the public key of the issuer to a real identity. This list is managed by the user/verifier itself, so is up to him to decide who can and cannot trust. There is no third party, because the blockchain is not a real entity. There are various method to populate such a list, and they will be further explained in the Entity Resolution Process. This gives SSI a good score.

Self Hosted is equal to Digital signature if it is based on SSL, which is based on certificate issued through the PKI, thus a mid rating is given.

In the SaaS model you have to trust the third party service provider in order to know who is the credential issuer. The only thing you can do is to double check with the issuer. This gives the lowest rating.

3. **Ensure Validity:** how can I know if the credential has not been revoked or expired? Is it even possible to do?

   Paper simply has an expiration date which can be printed on top of the document. Revocation is only possible by physically invalidating the document, which however seems a very inconvenient way. However, this is not a huge issue when dealing with the sort of academic/courses attendance credentials we are dealing with. For this reason the rating is only mid range.

   Digital Signature can include an expiry date on the signed document, but like a pen and ink signature (a "wet signature", just like the Paper model), a digital signature cannot be remotely controlled. If a digital signature is valid at the time the document was signed, then it remains valid forever. This is a downside for this model (think of a Driving License which cannot be revoked). This is where it comes the same rating as the Paper model.

   SaaS model can show an expiry date and can also be responsible for revoking credentials, but this is all about trusting the Service Provider. This gives the model a mid rating.

   Self Hosted model deals with the issue in a better way as the Issuer itself is the one responsible for revoking and has the possibility to do it at anytime since the credential itself is stored on his servers. The solution looks good from this point of view.

   SSI model allows for both expiration and revocation by simply checking a revocation list, which could be decentralized using a smart contract. This gives the model a good score once more.

4. **Portability:** How can a credential be transferred elsewhere? How cheap and time consuming that system is?

   Paper can be transferred in only two ways if the original credential is requested, via normal mail or in person, both of this method are not cheap nor fast. If

the original credential is not requested, then paper credential can be sent via fax, email or any other digital way, like a URL where the credential is stored. This methods are cheaper and faster than the previous one, but now the credential has lost its original format. Thus, the lowest rating is given.

Digital Signature instead can be sent with the same methods used for digitized paper without losing its original status because of the signature attached which guarantees its originality. The only cons is in the user experience, which each time involves some steps, like research the credential and send it, which is not a real cons, that's the reason of the maximum score.

When it comes to SaaS or Self Hosted the credential can be shared just by connecting to the link shared by who is sharing the credential. This method is free and fast but you have to rely on third party/issuer infrastructure, which could be down and so your credentials may not be shareable. Thus a mid rating is given.

With SSI method you can share credentials, stored locally on your device, by scanning a QR code presented by the server of the entity who you are sharing your credentials with. This method is free, fast and you are not relying on any third party infrastructure. This gives the maximum score.

5. **Interoperability:** how standard is the model to be widely used and adopted by different system? Does it allow for automatic credential verification?

   Paper model involves some standard format for some categories of document, however this is not common practice to adhere to a specific format. This is especially true when dealing with our type of academic/courses attendance credentials. A mid rating has been given because of the few paper templates used, even though the standardization is not unknown to the paper world.

   Digital Signature is made of well defined algorithms and most software already know how to read them automatically. The issue comes with the content of the signed document, which doesn't adhere to any standard and thus it becomes hard to work automatically with bulk of similar files. This gives it a mid rating.

   Credentials hosted under a certain Service Provider (SaaS model) have some kind of standard format thus allowing for interoperability. The issue comes when multiple service providers have to deal with each other, just like it currently happens, and that is where interoperability is losing traction. This gives it a mid rating.

   Self Hosted falls behind other models here as there is no standardized API which one can expose to make the system interoperable across different services. Therefore the lowest rating has been given.

   SSI has the same benefits of Digital Signature in terms of standard algorithms for checking a credential. On top of it the claims follow some specific format which makes it a great option for broad standardization and interoperability. Better rating has been given.

6. **Consent/Sharing permission:** Users must agree to the use of their credentials. How to make sure the credential holder is allowed to hold and share it? Is it even possible to do?

   In all the models it is impossible to completely avoid that someone, who you have shared your credentials with, then reuse and share them with others.

   In the case of SaaS and Self Hosted you can do something to counter fight this possibility by sharing a link/credential with very short expiration date, so that just the first verifier can verify it, and then they do not longer have the access to that credential. Thus a mid rating is given.

   SSI doesn't allow to share credentials that are not owned by you or for whom you are not delegated to share them. But in both cases none of the methods can avoid the verifier to make a copy of the credential and share it with other methods. That's one the problems that SSI doesn't solve completely.

7. **Data Minimization:** Disclosure of claims must be minimized.

   All the models need some information to identify the subject of the claim, which in turn involves sharing those information for verification purposes.

   Digital Signature is the only one which allows for signing a credential with just the right information needed. However the process is not efficient as a credential should be split into multiple files each one of them with different information depending on the amount of information to be disclosed. A mid rating is given as it is possible to have a sort of data minimization even though it is extremely inefficient.

   SSI allows for selective disclosure of information thus sharing only the bare minimum required. This grants is the best score.

8. **Persistence:** Credentials must be long-lived. They have to live at least after you.

   Paper documents have a good persistence and it is the way we have always dealt with documents in general. It is also the simplest way of archiving information and thus the best rating has been given.

   Digital Signed credential is easy to store and with some back up can be considered to be persistent. This gives it a good score.

   SaaS credentials have a persistence which completely relies on those service providers, making it an extremely bad fit for credential. No backup is possible as the credential only lives as long as the service lives. Bad scoring has been given.

   Self Hosted certificates suffer from the same issue as the SaaS model. Same score.

   SSI inherits the benefits from Digital Signature, as it is the user's interest to securely store the credential in a digital format. Simply safekeeping a digital file is considered a quite persistent approach even though a digital file only store on a single device may be easier to get lost rather than the paper version.

9. **Protection/Privacy:** who can see my Credentials? Is it publicly visible? Is it restricted to certain people?

   Paper, Digital Signature and SSI grants a high protection and a high privacy level given the fact that you and only the people you have shared with have a copy of your credentials.

   SaaS method implicitly oblige you to share your credentials with a third party, so here privacy cannot be assigned with the highest rate.

   Both in Self Hosted and SaaS, if there is a breach all the credentials stored by these silos could be public available.

10. **Data Sovereignty (Access/Availability/Control):** Users must have access to their own data without relying on third parties. There should not be the possibility to enforce censorship or any other form of control to block availability.

    Paper grants an extremely good availability of the credential as the document is stored by the user himself. Also he has full control over it has there is no intermediaries involved in the storing nor there is possibility to block the user from using the paper documents. Good score for the paper model.

    Digital Signature also allows for great access capabilities as the user stores the file and thus he is able to access the data at anytime. This gives good score.

    SaaS Credentials on the other side may have downtimes, or even impeding user's access to his own data depending on service provider's behaviour. This is considered not be a good solution. Bad score.

    Self Hosted suffers the same issues as the SaaS Credential. Same score.

    SSI allows the user to access the data at anytime as the credentials are only locally stored. Same benefits as the digital documents. Best score has therefore been given.

1.5 Known Alternatives

There are many different projects out there, varying in maturity, blockchain usage extend, geographical location of the team and intended scope of the project. We valued many of them before starting our work, choosing the most mature and SSI compliant ones, and in the chapter Known Alternatives we carry out a brief comparison giving out a short description of the project highlighting the differences with Pistis.

Anlayzing them we find out some problems, which can be summarized into:

- **Usage of Blockchain:** Most of them use a DLT/Blockchain, like a notarization platform, where is possible to store a proof[6] that no one has tampered with a particular credential. We think this is not a correct use of the blockchain

[6] In all the cases analyzed, the proof is the credential hashed

since it forces the issuer to write on the DLT/Blockchain every time a credential needs to be issued. We prefer to limit to the bare minimum the use of blockchain given its inefficiency and cost if used frequently.

- **Type of Blockchain used:** Some solutions prefer to build their own DLT/Blockchain to manage identities and their credentials, instead of using an existing one.
- **Credential Standard Used:** Some solutions use standard that are not widely accepted or sometimes they create their own standard, rising walls for broad adoption and interoperability.

Our solution aims at solving those issues and to provide a truly usable Self Sovereign identity solution.

Chapter 2

Proposed Solution

In this chapter we describe our solution and what are our contributions with this thesis.

A major problem pointed out by the state of the art survey on credential management models was the lack of standardization. This problem is easily solved by the mass adoption of one unique standard. The two most popular standards for credentials are Open Badges[22] and Verifiable Credentials[53]; the former created by the Mozilla foundation and now maintained by IMS Global; the latter created and maintained by W3C.

Verifiable Credentials proposed by the W3C is a strong winning point against Open Badges. On top of that, Open Badges, differently from VC, aren't thought with Self Sovereign Identity in mind, indeed they lack the concept of data minimization and data sovereignty[1]. Finally, Verifiable Credentials are strictly related to the concept of identity thanks to their association with a DID (decentralized ID), which is another W3C standard, allowing VC to be easily extended to other use cases other than Academic Credential Management.

Surely we are not the first ones to have a go implementing an SSI system building on top of Verifiable Credentials and Decentralized Identifier standards, nonetheless the standards being only recently proposed. Analyzing the already existing SSI projects, we believe uPort is the most mature. It makes use of the public Ethereum blockchain, it is open source and it limits to the bare minimum the use of blockchain keeping off-chain[2] all the identity related information, making it GDPR friendly by design, as further explained in GDPR compliance section. But still it lacks some fundamental features which would make it the actual killer dApp (distributed applications) in the field of credentials management, and in broader terms of digital identity.

Indeed, uPort still lacks the following both from a specification and implementation points of view:

- Does not fully adhere to W3C standards as it as it uses a slightly different DID Document structure and does not take full advantage of the concept of Verifiable Presentation

[1] Section model comparison clarifies on minimization and sovereignty of data

[2] Dealing with blockchain technologies, therms like on-chain and off-chain are used in order to discriminate between what operation are done on the blockchain and what not. In this case is used to point out that no identity related information are stored on the blockchain

- Revocation of Verifiable Credential already issued
- Entity association between DID and the owner of that DID in the real world
- Selective disclosure schema which efficiently allows to share just portion of a Verifiable Credential
- Large files sharing management
- Standardization of the names used inside a verifiable credential
- Dashboard for issuer and verifier to interface with all the functionalities offered by an SSI system.
- Cost efficient way to handle delegates

With the goal of overcoming these issues, we designed a new solution: Pistis is a credential management system based on the Ethereum blockchain. It provides a set of novel smart contracts to handle efficient multi signature operations, delegates management, permissioned access to extensible services based upon the Decentralized IDentifier specification. It aims at being an actually usable system in a real world scenario rather than just being purely research and experimentational work.

We took inspiration from uPort, and we even worked alongside some of their developers for the first period of work. But once realized the just described drawbacks, with went on with our very own specifications and reference implementation we have written from scratch.

Hence, Pistis solves the previous issues by giving:

- Generalized and proposed MultiSig enforcing operations through a frontend smart contract, easily extensible in any SSI linked scenario. The contract is arguably more elegant and better than the current common usage of a MultiSig Contract in terms of gas cost usage.
- Reference implementation for Delegates Registry Smart Contract, which allows to efficiently manage delegates relative to a DID.
- Reference implementation for Revocation Smart Contract, which allows to check and change the status of a Verifiable Credential already issued.
- Trusted Contacts Management proposal and reference implementation to handle entity association in a real world scenario.
- An Extended VC to properly manage selective disclosure.
- An Extended VC to include any file other than just text.
- Proposal for a standard Naming Schema, to solve the lack of standardization of Verifiable Credential content.
- Reference implementation of a complete User Mobile Application, to safely store and share Verifiable Credential from your own device.
- Reference implementation for Issuer/Verifier dashboard to handle delegates, manage credential, read VCs, build and issue VCs

Chapter 3

Building Blocks

In order to ease the comprehension of this work, a step by step approach has been preferred in explaining how starting from the W3C standards we reached the full Pistis reference implementation and specification. We start with a brief overview of the building blocks and then we go deeper into the details.

Our implementation aims at being generic enough in order to have Verifiable Credentials to be extended for different use cases and go beyond the academic qualifications.

We build on top of the W3C Decentralized Identifier and W3C Verifiable Credentials standards. The Building Blocks of this chapter are largely taken from the specification published by W3C. It is extremely important to clearly state these basic concepts so that we can build upon in the next chapters.

3.1 DID & DDO

DID stands for Decentralized Identifier. Up until now most of the identifiers of things on the internet have been under the control of organizations. Your email address will likely be under Google's control. Your linkedin profile, indeed stays under Linkedin's control. Those are some kind of hierarchical namespaces which live under total ownership of some company.

W3C proposed a new standard for objects addressing which is independent from any centralized registry, identity provider, or certificate authority, leveraging distributed ledger technologies (discussed in the following section), explained in the following section.

Every DID has its own DID Method, that is a set of rules that govern how DIDs are anchored onto a ledger. Specifically DID methods are the mechanism by which a DID and its associated DID Document are created, read, updated, and deactivated on a specific distributed ledger or network. DID methods are defined using separate DID method specifications.[51]

Every DID points to a DID Document (DDO) which is the serialization of the data associated with that DID. The main data to be shown in a DID Document are the public keys that can act on behalf of the DID they are associated with, that we refer to also as delegates. In addition to the publication of authentication and authorization mechanisms, the other primary purpose of a DID Document is

to enable discovery of service endpoints for the subject. A service endpoint may represent any type of service the subject wishes to advertise, including decentralized identity management services for further discovery, authentication, authorization, or interaction.

3.2 Blockchain

We got to the point where we have a way to create globally unique identifiers, the DID and a relative DDO to provide some additional information about that DID. Where is all this information stored? In a decentralized identity system like Pistis, identifiers and public keys can be anchored to a variety of Distributed Ledger Technologies (DLT), such as Bitcoin, Ethereum, and a variety of other similar technologies.

These are networks of computers that keep in sync with each other, maintaining one global ledger or database that is mirrored and replicated across thousands of machines. Entries in the database are periodically (every 15 seconds in Ethereum, few minutes in Bitcoin) depending on the particulars) cryptographically "sealed" so that they are practically impossible to change.

Especially, the blockchain our Pistis project is built upon is the popular multi-purpose blockchain Ethereum. However, the modularity of our implementation allows it to use the blockchain as plug and play component, thus allowing Pistis to be used on different blockchain systems. This becomes interesting in those scenario where the State or Organization willing to use the system in production may want to use a permissioned blockchain of their choice for instance. The prerequisite for doing so is to have the Smart Contracts ported to the new Ethereum alternative blockchain, keeping the same interface.

3.2.1 Why we need the blockchain?

It could all be feasible by just generating a key pair and using the generated public key as the identifier to attach to my very person. Then I could ask the Identity Provider to issue a credential linked to my public key. The first issue comes when we need to make sure the credentials are valid at the moment we check its validity. Each issuer should have its own Revocation List and expose it on the internet and ensure availability and security, non trivial and non cheap requirements to satisfy to a certain degree. From a technical perspective that is where the blockchain comes in handy with its intrinsic availability and low cost of usage.

Therefore we have the need for a decentralized and trustworthy piece of code, i.e. a Smart Contract, to act as a global decentralized authority to allow revocations of Verifiable Credentials alongside revocation of public keys associated to a DID. Indeed the need for a SSI ecosystem is for a decentralized and distributed storage with authorized accesses, that is a Distributed Ledger Technology running the smart contract we want.

Having the blockchain as a decentralized, single source of truth, there is no need to interact with third parties for verification of credentials. Also, the only piece

that needs to be trusted is a smart contract, which is trustworthy by design. This gives a great transparency and decentralization, which is the only means by which we can achieve a fully interoperable system.

Another great feature of blockchain and the smart contracts which run on it, is the uptime granted. By design you will always be able to access the information needed for the SSI system to work as there will always be a server available to serve the request. This extremely high availability would be harder to achieve by a single party taking care of it. To have an idea of how much it is we can take an experimental value from the Bitcoin blockchain which still uses Proof Of Work as a consensus mechanism. According to http://bitcoinuptime.com/ it is over 99.983% at the time of writing. Ethereum hasn't official values, but it can be estimated to be pretty much the same, given the use of the same consensus mechanism along other different technical similarities in the blockchains. These are the main reasons which make the blockchain the perfect infrastructure to build the SSI system upon.

3.3 Smart Contracts

Years before Satoshi Nakamoto's revolutionary white paper introduced the idea of the blockchain, a Computer Science Researcher and Cryptographer named Nick Szabo introduced the concept of Smart Contract, and in 1997 wrote an entire research paper on it[47]. Nowadays, we can define a Smart Contract as "a computer protocol to digitally execute the terms of a contract". It needs to be:

- Trustless: no third party has to have control over it. Needs to be universally accessible
- Trackable: Transactions can be traced. Auditability is also an important feature
- Irreversible: Just like legal contracts are binding, smart contract transactions need to be final.

A smart contract is also self executing. This allows to reduce costs, increase speed, flexible for different use cases. It can be also defined, in a more technical way, as code that is stored on a blockchain and is self executing thanks to the trust and security of the blockchain network. In the case of Ethereum, the code is compiled into bytecode to be executed by the Ethereum Virtual Machine (EVM).

3.4 Verifiable Credential

In the physical world, a credential might consist of:

- Information related to the subject of the credential (for example, a photo, name, and identification number)
- Information related to the issuing authority (for example, a city government, national agency, or certification body)
- Information related to the specific attribute(s) or properties being asserted by issuing authority about the subject

- Evidence related to how the credential was derived
- Information related to expiration dates.

> A verifiable credential can represent all of the same information that a physical credential represents. The addition of technologies, such as digital signatures, makes verifiable credentials more tamper-evident and more trustworthy than their physical counterparts. [53]

Our implementation is built on top of the concept of Verifiable Credential (VC). What we generally called Claim, it now has been given the name of Verifiable Credential. This is because the Claim represent someone's piece of information, indeed a Credential; Verifiable as it is possible to verify the validity of that piece of information. This generic verification concept is further explained in the following lines.

Claims are simply <key, value> pairs of information. Single or multiple claims are grouped under a so called Verifiable Credential.

Upon reception of a Verifiable Credential the following information are guaranteed by cryptographics algorithms:

- It has not been tampered with
- Who the issuer is
- Whether the claim is being shared by an authorized entity
- That the claim has been accepted by the claim subject
- Verify that the VC has not been revoked or expired[53]

3.5 Verifiable Presentation

A Verifiable Presentation is a simple encapsulation of more than one Verifiable Credential. Any time one wants to move a VC, nevermind if it is composed by single or multiple claims, it has to be encapsulated inside a Verifiable Presentation. This is to allow the sharing of multiple verifiable credentials at the same time. Think about the case where a student needs to share with an employer his CV which is composed by a verifiable credential for each past work experience and for each diploma he has attained in the past. That is when all those information represented as Verifiable Credentials are packed inside a Verifiable Presentation for convenient sharing.

Holders of verifiable credentials can generate presentations and then share these presentations with verifiers to prove they possess verifiable credentials with certain characteristics. The rapidity and convenience of such a system makes it perfectly suited in when trying to establish trust at a distance.

3.6 Actors

Within the SSI system we propose there are some well defined and distinct requirements. A generic end user interacts with other DIDs here and there but he always has a direct and non-automatic interaction with them. This is what allows him to relax some entity resolution rules as it easy to check the identity of the one thing he is interacting with. On the other side a user which has the need to automatically verify credentials, let's say an institutional entity for instance, needs to have a way to resolve and trust/untrust entities associated to DIDs. This is a first distinction which brings up two different actors: User and Verifier.

Another simple yet important difference comes from the practical user experience one has with the system. The end user needs to hold the credentials on his mobile phone while a verifier or issuer entity needs to handle the operations from a backend application. This brings to another distinction among User and Issuer/Verifier.

This leaves us with three different actors in the SSI system:

- User: the owner of some Verifiable Credentials, a student.
- Issuer: the entity issuing one or more Verifiable Credentials, a university.
- Verifier: one who only reads a Verifiable Credential and makes sure he trusts the issuer, an employer.

User requirements:

- Share Verifiable Credentials
- Receive and Verify Verifiable Credentials
- Store and organize Verifiable Credentials
- Check the history of the previous engagement with others DID
- Manage a list of trusted contacts
- Key Management

Issuer requirements:

- Create and Issue Verifiable Credentials
- Keep track of issued credentials
- Revoke Verifiable Credentials
- Multi-Signature Operations
- Key Management

Verifier requirements:

- Request and Verify Verifiable Credentials
- Manage a list of trusted contacts

Chapter 4

Pistis Specification

In the previous chapter we only gave a brief overview of the very basic building blocks needed to understand our Self Sovereign Identity system. In this chapter we give more details on our specification which builds upon those basic concepts while being compliant with the draft proposed by the W3C Credentials Community Group.

4.1 DID & DDO

4.1.1 DDO Structure

The DDO data structure is made of the following fields:

- context: it specifies the specification being used;
- id: the DID subject of the DID Document;
- publicKeys: they are are used for digital signatures, encryption and other cryptographic operations, which in turn are the basis for purposes such as authentication or establishing secure communication with service endpoints. It basically contains the list of all public keys that have a certain influence on the relative DID. The fields below tell what permission those keys have by setting a reference to them;
- delegatesMgmt: array of references to public keys which have delegates management permissions granted.
- statusRegMgmt: embedded keys or references to publicKeys field above, which have permission to interact with the Credential Status Registry smart contract. Thus, if did:pistis:0xAB... has key 0xCD... within the statusRegMgmt field, ethereum address 0xCD... can interact with the smart contract to revoke/change the status of credentials issued by did:pistis:0xAB... ;
- service endpoints: array of services linked to the subject DID;
- potentially any type of permission[1] can be added.

[1]Please refer to Smart Contracts section for a complete explanation of the way it is handled by Pistis.

As extensively stressed out by the W3C Credentials Working Group, the DID Document does not contain any personal-identifiable information (PII).

A complete example of DDO can be found in Appendix A

4.1.2 DID Method

Our specification is fully compliant with the W3C Decentralized Identifier and W3C Decentralized Identifier Document. We define our DID method, did:pistis:, supporting the following operations:

Create: specifies how a client creates a DID and its associated DID Document on the Decentralized Identifier Registry, including all cryptographic operations necessary to establish proof of control.

In this we match the did:ethr: specification for which the DID is generated by appending a newly generated Ethereum address to 'did:pistis:'. The cryptographical suite to create a key pair comes from the same as bitcoin's core, that is libsecp256k1. Generating the private key and public key is the same for both Bitcoin and Ethereum, they both use libsecp256k1 elliptic curve cryptography. Deriving an account address from the public key differs slightly. Deriving an Ethereum address from a public key requires an additional hashing algorithm. Taking the keccak-256 hash of the public key will return 32 bytes which you need to trim down to the last 20 bytes (40 characters in hex) to get the address. [11]

An often discussed topic is about keys collision avoidance. 2^256 is the size of Ethereum's private key space, a huge number. It is approximately 10^77 in decimal; that is, a number with 77 digits. For comparison, the visible universe is estimated to contain 10^80 atoms. Thus, there are almost enough private keys to give every atom in the universe an Ethereum account. If you pick a private key randomly, there is no conceivable way anyone will ever guess it or pick it themselves.

The creation of a DID does not involve any on-chain registration. This is extremely relevant for a Self-Sovereign system as there is no intermediaries involved in the creation of a DID. Anyone can freely create his/her own identity and there is no entity able to censorship or impede that creation. Indeed, creating a DID is just a matter of creating an Ethereum key pair, let us call it 0xAB. The DID that address is controlling is the trivial did:pistis:0xAB. Now a Verifiable Credential can be issued by did:pistis:0xAB by signing the JWT with the private key relative to the ethereum address 0xAB. Also, that address will have any kind of permission on did:pistis:0xAB. Right after the creation of a DID it is important to set up some other DIDs to have the right permissions in order to enable key recovery. Please refer to the section Identity managment.

Read/Verify: how a client uses a DID to request a DID Document from the Decentralized Identifier Registry, including how the client can verify the authenticity of the response.

This is achieved through the DID Resolver (for which we provide a reference implementation as well), that is a software component with an API designed to accept requests for DID lookups and execute the corresponding DID method to retrieve the authoritative DID Document. As the W3C specification involves, the

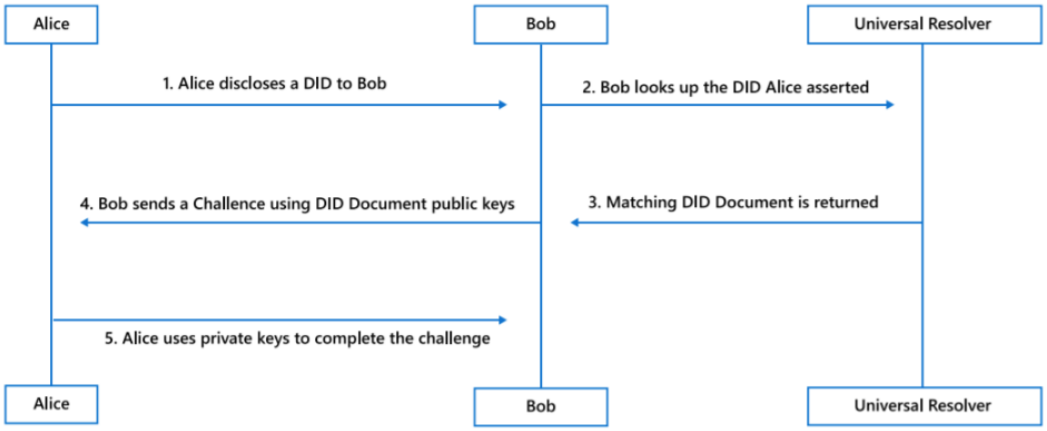


Figure 4.1: Handsaking DID

DDO contains a list of addresses which are authorized to sign on behalf of that DID, thus allowing a verifier to check authenticity by applying the asymmetric keys algorithm and check whether the signing key is amongst the ones who have authorization permissions. See the figure 4.1 to have an insight about the authentication procedure.

Update: how a client can update a DID Document on the Decentralized Identifier Registry, including all cryptographic operations necessary to establish proof of control.

It is done by updating our DID Registry Smart Contract that holds the mapping between DIDs and the relative addresses with their specific permission for that very DID.

As suggested in the spec from the W3C, Pistis supports a quorum of trusted parties to enable DID recovery. Once DID and its relative key pair is generated for a user, it is heavily recommended to the user to add two more delegates for the reasons which will be explained below. When only one delegate is owned by a user, regardless of the permissions, it is allowed for that address to add one delegate. Once two or more delegates are added for a certain service, a multi signature is needed to add another. Same rule applies for revocation of a delegate which simply consists in deleting the corresponding entry from the delegates array.

This delegation mechanism helps mitigating a scenario where the user loses access to the private key corresponding to a public key delegated of his DID or when that key is compromised. In either case, the presence of three or more delegates allows the user to ask the other delegates to revoke the compromised address. The legitimate user can then generate a new address and ask the delegates to associate it with the DID to get back control over it and fully restore its functionalities. The whole implementation has been done exploiting a fairly complex Smart Contracts inheritance structure to allow for extensibility. More on this in the Smart Contract section

Deactivate: how a client can deactivate a DID on the Decentralized Identifier Registry, including all cryptographic operations necessary to establish proof of

deactivation. Deactivation applies in a similar manner as the Update. Indeed, it is a matter of revoking all addresses for a certain DID. This makes the DID unusable and non retrievable from that point onward. In this case, there is no public key that can be used to authenticate the holder's identity.

More importantly, Deletion of a Pisits DID means that this DID cannot be reactivated again.

4.1.3 Service Endpoints

The main use Pistis makes of the service endpoints capability is to advertise for Trusted Contacts List. In this case, the serviceEndpoint points to where the tcl is hosted, that could be either a smart contract address, an https endpoint hosting the TCL or even an IPFS resource pointer.

Other services can be linked to a certain DID, it could either be Faucets, Name Services, or any other kind of service which has to deal with that DID and it makes sense for users to be aware of it. This is how they would discover those services.

4.2 Naming Schema

The above definition of Verifiable Credential and Verifiable Presentation can serve as a starting point, but actual widespread adoption still lacks some standardization. There are two main issues we identified coming from the flexibility of the W3C standards:

1. Naming System: Lack of a standard naming system for claims

2. Abstract Granularity: Lack of a fixed Abstraction Level in representing information between VC and VP

An ecosystem without such standardization would suffer interoperability as it already happens for some Credentials Management Models as already shown in the Model Comparison chapter. What we call Naming Schema proposal aims at solving both the issues mentioned above.

4.2.1 Naming System

The former becomes easy to spot once we tried to imagine of a system like this already in place among multiple independent entities, like it can be for different Universities in Italy or even Europe. Think of Politecnico di Milano releasing a Verifiable Credential with my University degree in Computer Science Engineering. How would they call the credential? "UniversityDegree", "University Degree", "University Diploma", "Uni Diploma", "Diploma" or what? Imagine now I want to spend the credential applying for a job in the UK. Would the system accept and automatically verify the credential look for all those names above and more? Or would they just ask me for any credential and then someone will manually look into it and understand that the verifiable credential is what they were expecting? The issue becomes even more relevant if issuer and verifier might refer to the very same

word giving different meaning as it is the case of a Master Degree. Each country in Europe has a slightly different way to value a Master Degree. There it comes the need to impose a structured yet flexible nomenclature for claims. On the other side, we cannot force a static set of properties. There are at least two downsides in doing so: we would have the power to decide what exists and what doesn't, creating centralization issues; the ecosystem would not be open to changes. An ecosystem like that would also not be open to changes.

The Naming System we propose involves the use of schema.org vocabulary and schemas as a community driven, decentralized and updated reference. Schema.org will then be used to give specific names and types to pieces of information.

The **data model** [41] used is very generic and derived from RDF Schema (which in turn was derived from CycL). We have a set of types, arranged in a multiple inheritance hierarchy where each type may be a subclass of multiple types. We have a set of properties:

- each property may have one or more types as its domains. The property may be used for instances of any of these types.
- each property may have one or more types as its ranges. The value(s) of the property should be instances of at least one of these types.

DataType types first level subclasses are primitive types. This means that a schema.org instance tree will always terminate each branch with a DataType type.

For example, if I wanted to make a university degree verifiable credential I would use the type: "EducationalOccupationalCredential". We may want to add some properties from the ones available under the very same type, or inherited from the upper classes. Below an example of standardized Verifiable Credential's claim for an "EducationalOccupationalCredential" with properties "credentialCategory" to explicitly say it is about a diploma and "educationalLevel" to point out that it is a master diploma (Note that not the grade nor the degree subject are being included. See the following subsection Abstraction Granularity, where the issue is dealt with):

```
[...]
"csu":
{
    "@context": "https://schema.org",
    "type": "EducationalOccupationalCredential",
    "name":"Master Diploma in Computer Science
       Engineering",
    "credentialCategory": "degree",
    "educationalLevel":
    {
        "@type": "DefinedTerm",
        "name": "Master",
        "termCode": 7,
        "inDefinedTermSet": "https://ncfhe.gov.mt/en/
           Pages/MQF.aspx"
    },
```

```
    "competencyRequired" : {
        "@type": "DefinedTerm",
        "termCode": "LM-32",
        "name": "Ingegneria Informatica",
        "inDefinedTermSet": "https://www.universitaly.it
            "
    }
}
```

Codice 4.1: Sample content of a *csu* field inside a credential representing a Master Degree in Computer Science

"EducationalOccupationalCredential" goes under the hierarchy Thing > CreativeWork. It is the Verifiable Credential's type. The *@context* establishes the special terms used, as proposed on the W3C standard. The *name* attribute is just for human readability. The *credentialSubject* attribute encapsulate the core information, i.e. the claim itself. Under *credentialSubject* we can only place properties allowed for the VC's type. In our example *credentialCategory*, *educationalLevel* and *competencyRequired* are three allowed properties. The former has a Text type as value, that is a data type (i.e. primitive type) and thus it goes as a leaf of the claim tree, while the latter has a non basic type that is DefinedTerm with other properties instancing primitive types to end the tree.

If needed, external extension mechanism through the so-called hosted extension is available and supported by schema.org through self hosted schemas. For example a certain university may want to add some very specific course information as credentials. All it would take is a definition of a schema as proposed by schema.org and they would have to expose an endpoint with that schema description. The credentials would then point to that url using the *@context* field.

4.2.2 Abstraction Granularity

At a higher level of abstraction a Presentation of Credentials may represent a School Diploma, including the following Credentials: School Name, Program Name, Graduation Year, Final Grades.

A great issue which comes with such flexibility is the need for a standardization of Presentation types and Credential types. Think of a new trustworthy marketplace website, where users would post some objects to be sold by some others. The buyer needs to go to the location where the object is placed in order to make a purchase. Since the exact location of a user selling the good doesn't have to be disclosed, only the postal code is necessary in order to give an idea of the non accurate location of the object. The website may then ask for an Address Verifiable Credential, when the user might have had released an AddressPostalCode VC from his municipality. An ecosystem without such standardization would suffer interoperability as it already happens for some Models of handling credentials as already shown in the Model Comparison section.

We face a trade-off between having atomic Credentials and giving the Presentation little abstraction means, in fact, it only represents an array of credentials, which also gives a clear and fixed structure alongside a lot of overhead to represent

multiple pieces of information on a single Credential. On the other side, we have a much more relaxed model with less overhead in packing up information, but giving up on fixed standards.

The Abstraction Granularity we propose enforces the use of atomic Credentials, that is Credentials only made by a single claim object, allowing for finer granularity of selective disclosure. As seen in the Naming System, the credential has a specific type taken from one proposed by schema.org tree. The credentialSubject field may then only contain properties of that type, which in turn may have non-primitive types having a non fixed depth of the credentialSubject field. Note that, as opposed to what the W3C specification defines, the properties fields inside credentialSubject are all strictly related to the same outer type, and are not separate, independent claims about the same subject DID only. This way the issuer can arbitrarily decide whether to include one or more claims inside a credential by choosing the right Type depth from the schema.org tree. For example, a "PostalAddess" credential may be created with "addressLocality" and "addressRegion" as a nested properties inside the credential, or may choose to have two simple credentials of type Text representing those two small pieces of information for greater granularity of information and thus allowing more privacy preserving information disclosure.

This standardization allows for a great communication interoperability. In fact, one may just ask for any type of credential by requiring something like Thing/*, or you may ask for any educational related information with the query: Thing/CreativeWork/EducationalOccupationalCredential/*

4.2.3 Extending the Naming Schema

In some specific contexts, there can be some well established schemas. Think about the healthcare field: any digital system already uses the HL7 standard. Pistis accounts for that allowing to specify the naming schema in the context field of the credentialSubject.

4.2.4 Choosing a Verifiable Credential naming

In a real world scenario where it comes for an Issuer to create a credential it becomes quite complex for the user to navigate through the schema.org website and try to find the naming which best suits the credential he is willing to issue.

For this reason we created a simple utility, the VC Builder, which offers a user friendly manner to create a Verifiable Credential. Writing a JSON may be quite a cumbersome task for those who are non technical, a UI to navigate the schema.org tree and input fields to fill in a credential is therefore a vital tool for real adoption. On the other side, a standard tool to build credential helps interoperability among different systems.

Appendix A.10 shows a screenshot of the utility.

4.3 Verifiable Credentials

Here we dig deeper into how we have implemented Verifiable Credentials.

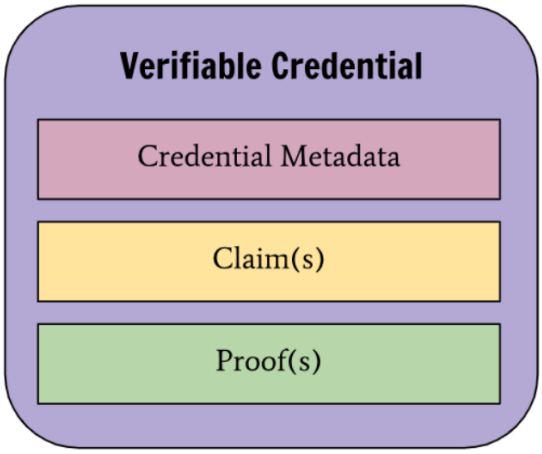


Figure 4.2: Verifiable Credentials fields

4.3.1 Simple VC

The W3C standards define the Verifiable Credential as subdivided into three logical sections as depicted in the figure 4.2 Credential Metadata are data which are used to contextualize the information included in the claims section. The following are the fields that we have chosen to include as metadata of a verifiable credentials, following the W3C standards.

- context: It's a URI, that once it is dereferenced, results in a document containing machine-readable information about how to properly read a verifiable credential. In our case is a document containing information about a Pistis verifiable credential.
- iat: issuance timestamp
- sub: the subject of the credential, to whom is entitled the credential
- exp: expiry timestamp
- iss: issuer's DID,
- csl: credential status list, which contains the information needed to verify the status of the credential itself.

The actual assertion done about the subject of the credential are called claims. All these data are contained in a field called Credential Subject (csu). The following is the structure we have chosen to organize the subject information into:

- context: It's a URI, that once it is dereferenced, results in a document containing machine-readable information about how to properly read the claim.
- type: the type of the claim. This field is used, to know which are the possible keys/properties that we could expect. Basically it is the path to be added to the context in order to retrieve the possible keys allowed

- name: a name given by the creator of the credential, just to enhance the user experience. So who reads this name can easily understand what this credential is about.

- <key, value> : a set of key value pairs, which are the effective attributes attested by the credential[2].

In order to transform a set of claims into a verifiable credential we need a way to verify that the data has not been tampered with and that they are issued by the actual issuer stated in the metadata. To allow for such a verification the claims need to be signed with one of the public keys contained into the DDO associated to the issuer's DID.

Our reference implementation uses the open standard Json Web Token (JWT)[3] that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA. In our implementation the signature is done using the elliptic curve signature algorithm with curve es256k1 (abbreviation: ES256K), the one used in both Bitcoin and Ethereum to sign transactions.

The following are the fields needed in the proof section of the Verifiable Credential:

- typ: it will always be JWT in our case

- alg: the signing algorithm used to sign the JWT

- signature: the signed JWT

This was the logical structure of a Verifiable Credential. Having chosen to use JWT, the verifiable credential, practically, will result subdivided into three main object following the JWT specification:

- header: typ, alg,

- payload: context, iat, sub, exp, iss, csu;

- signature: proof signature field (the signed JWT)

Here is an example of a verifiable credential attesting that the sub has attained a university degree:

```
"header":{
  "typ": "JWT",
  "alg": "ES256K-R"
},
"payload":{
  "context": "https://www.pistis.org/2019/credentials/v1
     ",
```

[2]Refer to the section Naming schema for clarification about keys and values data.

[3]refer to https://tools.ietf.org/html/rfc7519 for a complete explanation about what is a JWT

```
  "iat": 1554889743,
  "exp": 1554890343,
  "sub": "did:ethr:0xa0edad57408c00702a3f20476f687f3bf8b
     61ccf",
  "iss":"did:ethr:0x9fe146cd95b4ff6aa039bf075c889e6e47f8
     bd18",
  "csu": {
    "context": "https://schema.org",
    "@type": "EducationalOccupationalCredential",
    "name": "University Degree",
    "credentialCategory" : {
      "@type": "DefinedTerm",
      "name": "Computer Science Engineering",
      "termCode": "CSE"
    },
    "educationalLevel" : {
      "@type": "DefinedTerm",
      "name": "University Degree",
      "inDefinedTermSet": "https://www.eu-degrees.eu/
         degrees"
    },
    "aggregateRating" : {
      "@type": "AggregateRating",
      "ratingValue": "110"
    }
  },
  csl: {
    id: "<id>"
    type: "Pistis-CSL/v1.0"
  }
},
"signature": "eyshdkndkdlfj324ndsakjas..."
```

Codice 4.2: Verifiable Credential Sample

All the above data are enough if we want to read or write a simple Verifiable Credential, but when we need to share large files, or enable selective disclosure or create a one time use Verifiable Credential (aka a verifiable ticket) we need to enrich its structure with other properties and protocol for these particular purposes. In the next sections we will describe these new properties and protocols.

4.3.2 VC containing large files

It might often be useful to include a large file inside a Verifiable Credential, they may derive from legacy reasons. For instance, let's pretend we wanted to include a detailed scan of the paper version of a Diploma and include it in the relative Verifiable Credential. Being Pistis thought to be a versatile system, this aspect has been taken into account.

The main issues from including a large file as it is inside a VC are the following:

- The main transport used for Pistis is the QR. It only has a limited capacity of about 3,000 bytes if lowest correction level used.
- Mobile phones are becoming more and more powerful, but still it is not possible to ask for too much storage as they still are small devices. Especially, the target device we have in mind is a low range device as the system is to be used by anyone regardless of the device performance.
- Each time a VC of large size is shared, the whole VC has to be uploaded by the sender and downloaded by the receiver. This is totally feasible with small files (under 1MB), but it would become a real pain and extremely cumbersome to force the user to do so each time. It would be a non trivial limitation for the system usability.

What above led to a different approach which would ensure all those benefits Pistis already offers. Inside the credential only the hash of the intended file is kept. Thus the integrity of the file is ensured, and thus the whole VC keeps being the same piece of immutable information as before.

On the other side, the file itself is to be retrieved in a different way. We propose a few ways this can be accomplished, but the protocol is built in such a way that it is not tightly bound to any specific transport mechanism. Upon sharing of a VC, if the file type is recognized, the Verifiable Presentation will also include a files field which is an object containing the hashing function used and, for each Verifiable Credential present in the Verifiable Presentation, an array of objects (precisely a schema.org DownloadAction) to express the details of where to retrieve the files.

In a real life scenario, different download methods would be chosen depending mainly on privacy required for that file and bandwidth available.

- Low privacy: the example could be a simple logo which needs to be the content of a credential to be signed, and thus not just a simple logo linked to an association (as for that it would be just a matter of adding a "logo" property with an url as value). In this case the DownloadAction associated to the VC would include a simple remote and open a url where the file can be retrieved.
- Medium privacy: it could be a diploma, which doens't have to be fully available on the web, however it is still visible in some specific context, as it could be social media. In this case the storing of the file is delegated to a secure data hub as it could be the yet to be standardized Identity Hub [18]. It will then generate a one time link to be included in the Verifiable Presentation in order to let the receiver download the file from there. In this case the generation of the credential requires asking the Identity Hub for that one time link, and thus may suffer some delays, even though it would still be more than reasonable.
- High privacy: if the file has to be kept on the user's device no matter what. In such a scenario the Download Action would include the file content itself.

What is actually stored into a VC containing a large files, is a series of escaping character with the position in the VP files array where we can find the corresponding DownloadAction (<?f "position" ?>). After the escaping characters there is the hashed content of the file. Here there is an example of VC with large files:

```
{
  "iat": 1558515010,
  "sub": "did:ethr:0xa0edad57408c00702a3f20476f687f3bf8b
     61ccf",
  [...]
  "csu": {
    "@context": "https://schema.org",
    "@type": "EducationalOccupationalCredential",
    "name": "Diploma",
    [...]
    "image": "<?f1?>CF0BF0055AF44C1DFAC9FB48080DE93F6C1F
       54A220127C7EC37CA9E8898DB00A"
    }
  },
  [...]
}
```

Codice 4.3: Verifiable Credential with large files

As you can see the image property is the large file and in this case is stored at the second position of the files array contained in the verifiable presentation. In this case the files field will look like the following one:

```
{
  ...
  "files": [
       {
        encodingFormat: "SHA256",
        values: [
            {... download action referred to another
               file ...},
            {
              "location": "remote",
              "url": "https://www.qldxray.com.au/wp-
                 content/uploads/2018/03/imaging-
                 provider- mobile.jpg",
            }
          ]
       },
       {encodingFormat: "SHA256", values: [...]}
    ]
}
```

Codice 4.4: Verifiable Presentation with large files

So as you can see there is an array of download actions for each Verifiable Credential.

All these array are contained into another array for an easier implementation. The following are the steps performed by the Pistis system to verify a VP with one or more VCs containing large files:

- Parse the content of each Verifiable Credential searching for the escaping characters;
- Every time an escaping character is found look into the VP files property for the corresponding file. (e.g found the escaping characters <?f1?> in the second VC, you have to look in the second array for the second DownloadAction)
- Download the corresponding file
- Compute the hash of the file content
- Compare the computed hash with the one stored into the Verifiable Credential

When someone wants to share a Verifiable Presentation containing one or more VC with large files, he has just to include in the same order of the VCs the corresponding array of DownloadActions in the Verifiable Presentation.

We have chosen to implement this protocol in order to allow the sharing of multiple files without having to change the structure of the VC using the bare minimum additional field into the VP.

4.3.3 VC with selective disclosure

Selective disclosure is clearly an important feature of Verifiable Credentials, e.g. for driving licenses or passports we might only wish to reveal our name and nothing else. There are several potential ways of doing this:

- use of ZKPs - zero knowledge proof algorithms allow assertions to be made about the VC, without revealing the VC itself
- use of atomic credentials - each property of the credential is issued as a separate VC so that the holder can reveal individual properties
- use of hashes - The VC only contains hashes of each of the credential subject's properties, and the properties are separately held by the holder. The holder places the to-be-revealed claim in the Verifiable Presentation and the verifier computes its hash and compares it to the appropriate hash in the VC.

In our implementation we have chosen to use the last one, because it is the optimal compromise between being easy to implement and allows for a real selective disclosure.

The way it is implemented is very similar to the way we manage large files. The actual Verifiable Credential data will be contained into an external properties called data inside the Verifiable Presentation that wraps the VC. In the VC, instead, there will be just the hash of the content concatenated with a series of escape characters

like the ones used for the large files (<?d "position" ?>). Upon the sharing of the credential a user can choose which data wants to share by including them in clear into the data property of the Verifiable Presentation. This way who receives the Verifiable Credential can see just the data shared into the Verifiable Presentation, the other data in the Verifiable Credential are just the hashed one. This way we enable selective disclosure without the need to modify the VC or to ask the issuer to issue multiple credentials.

If the data that has been chosen to be hashed is an easily guessable data like my age, then an attacker could use bruteforce to retrieve my age even if I didn't want to share with him my age. To avoid this, these types of data are concatenated with a salt of 32 byte, which is just a random data to avoid brute forcing.

Here it is an example of a Verifiable Credential of a Diploma where the grade and the type of Diploma have been hashed.

```
"context": "https://www.pistis.org/2019/credentials/v1",
  "iat": 1554889743,
  "exp": 1554890343,
  "sub": "did:ethr:0xa0edad57408c00702a3f20476f687f3bf8b
     61ccf",
  "iss":"did:ethr:0x9fe146cd95b4ff6aa039bf075c889e6e47f8
     bd18",
  "csu": {
    "context": "https://schema.org",
    "@type": "EducationalOccupationalCredential",
    "name": "University Degree",
    "credentialCategory" : <?d0?>CF0BF0055AF44C1DFAC9FB4
       8080DE93F6C1F54A220127C7EC37CA9E8898DB00A
    "educationalLevel" : {
      "@type": "DefinedTerm",
      "name": "University Degree",
      "inDefinedTermSet": "https://www.eu-degrees.eu/
         degrees"
    },
    "aggregateRating" : "<?d1?>CF0BF0055AF44C1DFAC9FB480
       80DE93F6C1F54A220127C7EC37CA9E8898DB00A",
}
```

Codice 4.5: Verifiable Credential with selective disclosure

Then in the data property of the Verifiable Presentation in which the VC has been included there is an array for each VC presents in the VP, containing all the data in clear that the user has chosen to share.

```
{
  ...
  "data": {
        encodingFormat: "SHA256",
        values: [
          [
```

```
            {
              "@type": "DefinedTerm",
              "name": "Computer Science Engineering",
              "termCode": "CSE"
            },
            null
          ],
          [another array of data for the hashed one in
             the second Verifiable Credential],
        ]
}
```

Codice 4.6: Verifiable Presentation with selective disclosure

In this case the user has chosen to share just the type of diploma and not the grade, that's why the second entry of the array is a null object.

Actually not all the Verifiable Credential data need to be treated this way. Is up to the issuer who creates the Verifiable Credential to decide which data need to be hashed and which not. Obviously selective disclosure will be enabled only on the data that has been hashed.

The following are the steps performed by the Pistis system to verify a VP with one or more VCs containing selectively disclosed data :

- Parse the content of each Verifiable Credential searching for the escaping characters;
- Every time an escaping character is found look into the VP data property for the corresponding data. (e.g found the escaping characters <?d1?> in the second VC, you have to look in the second array for the second data)
- Check if the data has been disclosed or not
- If has been disclosed, compute the hash of the data
- Compare the computed hash with the one stored into the Verifiable Credential

When someone wants to share a Verifiable Presentation containing one or more VC with selectively disclosed data, he has just to include in the same order of the VCs the corresponding array of data in the Verifiable Presentation.

As for the large files, the encoding format is included inside the data property of the VP.

The use of a merkle tree has been considered, and discussed in the Future Work section.

4.3.4 VC as a Verifiable Ticket

Verifiable Credentials are issued to a certain DID and are bound to live with it. However, there may be the case where the VC might contain information which are not strictly linked to anybody and thus the credentials can be shared, used, and hold by anyone. A concrete use case can simply be a discount given at a

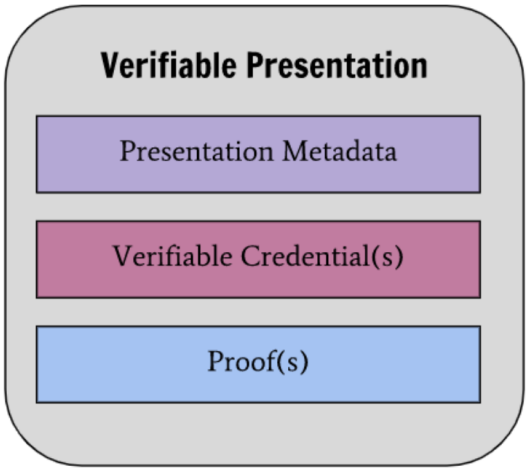


Figure 4.3: Verifiable Presentations fields

certain competition won by someone. In that case that discount can be issued as a Verifiable Credential, but it could also be fungible and spent by others.

Our implementation accounts for these types of situations by having a Verifiable Credential with the *sub* field equals to zero, that is the credential is not strictly bound to anyone, and can therefore be shared and exchanged among users. A common way to deal with this kind of Verifiable Tickets is to revoke them once used, as they are often meant to be used once, or the validity can be time bounded trivially using the expiration timestamp. Pistis does not enforce that the credential can only be shared once, but it is up to the issuer to deal with the situation in the most suitable way.

This approach is an alternative to the commonly used Non-Fungible Tokens (NFT)[42]. The major difference lies in that a Verifiable Ticket does not have any reference on chain (unless a revocation status is set for that credential), indeed it totally lives off-chain. The exchange of such a token/credential can therefore be private and free as it is not registered in an Ethereum transaction. Depending on the specific use case one approach outweighs the other in terms of benefits.

4.4 Verifiable Presentation

The W3C standards define the Verifiable Presentation as subdivided into three logical sections as depicted in the figure 4.3 Presentation Metadata are data used to contextualize the verifiable credentials shared. The following are the fields which we have chosen to include as metadata of a verifiable presentation, adapting the W3C standard.

- aud: The recipient's DID,
- iat: Issuance timestamp,
- exp: expiry timestamp,
- iss: The sender's DID

The actual Verifiable Credentials are inserted into an array already encoded as JWT, other than these we have included two other fields: file, used in order to manage large files which cannot be included directly into a verifiable credential; and data, used to enable selective disclosure. Here there is the structure of the Verifiable Credentials section inside a Verifiable Presentations.

- vcl: array of verifiable credential JWTs
- file: array of Download Actions
- data: unencrypted data to enable selective disclosure

As for the Verifiable Credential, a Presentation in order to be Verifiable needs to be signed with one of the public keys contained into the DDO associated to the issuer's DID. And also for Verifiable Presentation we have chosen to use Json Web Token (JWT). The following are the fields needed for the proof section of the Verifiable Presentation:

- typ: it will always be JWT in our case
- alg: the signing algorithm used to sign the JWT
- signature: the signed JWT

This was the logical structure of a Verifiable Presentation. Having chosen to use JWT as verifiable credential, practically, will result subdivided into three main object following the JWT specification:

- header: typ, alg,
- payload: iat, aud, exp, iss, vcl, file, data;
- signature: proof signature field (the signed JWT)

An example of Verifiable Presentation encapsulating my Career information as multiple Verifiable Credentials regarding courses would look like the following:

```
"header":{
 "typ": "JWT",
 "alg": "ES256K-R"
},
"payload":{
 "vcl": [
    eyJ0eXBaOwJSQ1....HdHrgh-EnAA ,
    eyJ0eX2dsisdc2....HoHthh-TmZZ ,
    eyJ0eXqiergGC3....HoHthh-TmZZ
 ],
 "file": [
     {encodingFormat: "SHA256", values: [...]}
     {encodingFormat: "SHA256", values: [...]}
 ],
```

```
15  "data": {encodingFormat: "SHA256",
16           values: [
17             null,
18             [data1, data2],
19             null
20           ]},
21  "aud": "did:ethr:0x7da253add95f4fe6gh269cf173c586s6g46d
      7va24",
22  "iat": 1554889743,
23  "exp": 1554890343,
24  "iss":"did:ethr:0x9fe146cd95b4ff6aa039bf075c889e6e47f8
      bd18"
25 },
26 signature:"eyshdkndkdlfj324ndsakjas..."
```

Codice 4.7: Verfiable Presentation Example

4.5 Smart Contracts

Pistis architecture needs to access the blockchain in different scenarios that can be divided into read and write operations as below:
Read Operations:

- DID-Resolver: Resolve DID into DID Document. This is needed on multiple cases, the most common of which is the verification of authorization to sign on behalf of a certain DID, using the concept of delegate
- VC Status Check: Verify the status of a verifiable credential by checking a revocation list.
- Entity-Resolver: Verify if the issuer of a verifiable credential is a trusted one or not by checking a Trusted Contacts list.

Write Operations:

- Delegates Management: Add or Revoke a delegate, update the permission associated with a delegate
- VC Status Set: Change the status of a verifiable credential updating a revocation list.
- Trusted Contact Management: Update the trusted contacts list adding or revoking entities.

Read operations doesn't need any kind of permission, anyone can perform them and more important they are completely free, you don't have to pay any Ether.

Write operations, instead, require permissions to avoid that anyone could freely edit the state of a smart contract. That's why we need a Permission Registry which

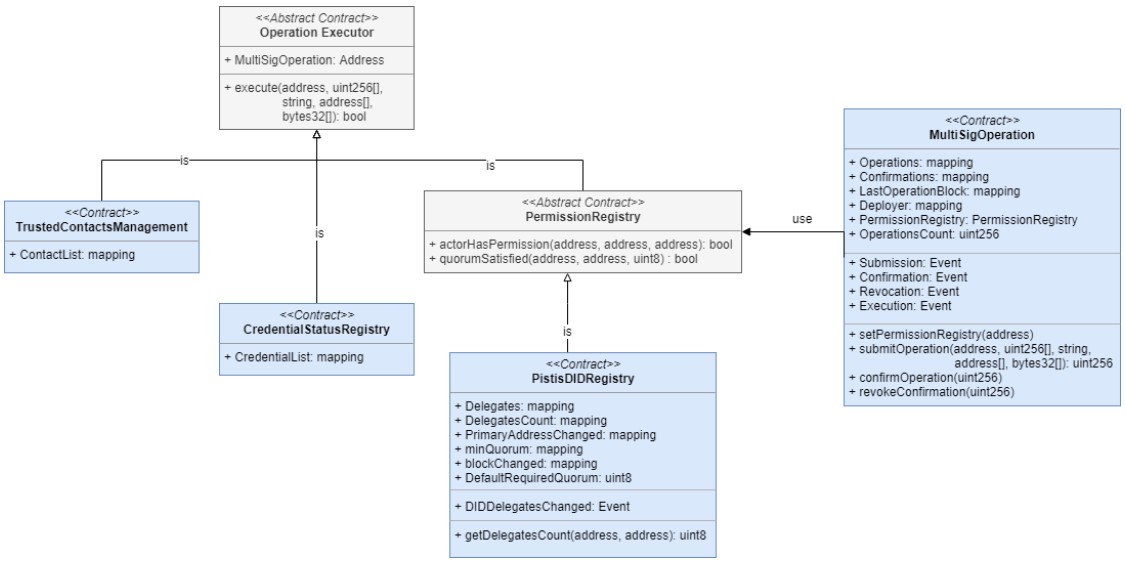


Figure 4.4: Smart Contract Class Diagram

holds these permissions and allow to check and update them. Obviously, write operations have a cost, because we are updating the state of a smart contract.

A simple Registry, as we have already described in the DID method section, cannot protect us from a lost private key, or a stolen private key used by a malicious attacker. That's why we need a way to execute these kind of operation with the consensus of a minimum quorum of parties. Here is where the multi signature wallet comes in handy. A multi signature wallet is a smart contract which allows multiple people to manage the same wallet, that is the contract itself. When a transaction needs to be executed one of the n owners of the wallet can submit a transaction, then this transaction in order to be executed needs to be confirmed by m over n wallet owners. That is exactly what we need in order to increase the security of our smart contracts.

However, using a normal multi signature wallet, every single DID who wants to have this kind of security needs to deploy on the net a multi signature wallet. This is something that we would like to avoid. Hence, we have reused the same idea of the multi signature wallet to create a smart contract which combined with a permission registry allows to have the same functionality of a multi signature wallet but deploying it just one time. The smart contracts as we have implemented them, are a way to handle any type of service and who has permission to work on it on behalf of a certain DID. This is done thanks to the generic OperationExecutor contract which can be seen a contract whose state has to be modified in a multi-sig manner by those who have permission for a certain DID. This ended up to be an interesting topic which could be treated per se on a different paper. The resulting Class Diagram of the Smart Contracts that we have implemented can be seen in the figure 4.4

4.5.1 Multi Signature Operation

This can be called as the frontend smart contract for the write operations. In fact is the only one which allow someone to modify contracts state. In this way we can control who has the right permission to complete a certain operation and at the same time if that operation has been approved.

Before going on, a clarification is needed. In the following paragraphs the term *function* refer to any of the smart contract function, like `SubmitOperation`, `ConfirmOperation` etc ... The term *operation*, instead, will be used to refer to the actual operation that who is using the multiSigOperation contract is trying to perform, like adding a new delegates or updating the contact list in the TCM smart contract, so basically an operation which modify the state of one of the smart contracts.

To complete any kind of operation, one has to first submit that operation using the `submitOperation` function. Then as soon as enough delegates have confirmed that operation, using the `confirmOperation` function, that operation will be executed. Executing an operation means to call another smart contract which inherits from Operation Executor smart contract. In this way the multiSigOperation contract is as general as possible, allowing anyone to create his own Operation Executor contract and to use the multiSigOperation as an alternative to the multi signature wallet.

Actually before executing any of these functions (`submitOperation, confirmOperation, revokeConfirmation...`), the contract checks if the caller has the right permission to complete that kind of operation, by calling a smart contract that inherits from Permission Registry. Again this is to allow to remain as general as possible and to allow anyone to implement his own Permission Registry.

The permission needed to complete an operation is the address of the Operation Executor responsible of executing the operation.

4.5.2 Operation Executor

The Operation executor smart contract is an interface smart contract. Inheriting from Operation Executor means being responsible of executing a particular operation. In order to inherit from it, any smart contract needs to implement the function `execute`, which is the function that actually execute an operation.

Every time an Operation Executor is deployed onto a net, the constructor required to set the address of the mutliSigOperation contract associated. This is to avoid that no one else but the multiSigOperation can call the `execute` function.

4.5.3 Permission Registry

The Permission Registry is an interface smart contract that inherits from the Operation Executor. Inheriting from Permission Registry means being responsible for tracking the association between addresses, their permissions and the minimum quorum needed to execute an operation . The inheritance from the Operation Executor is needed in order to update these associations.

In order to inherit from it, any smart contract needs to implements the functions :

- `actorHasPermission`, which returns true if the one who is trying to execute the operation has the right permission for it.
- `quorumSatisfied`, which returns true if the quorum to execute a particular operation has been reached.

4.5.4 Pistis DId Registry

The Pistis DID Registry Smart Contract is our implementation of the Permission Registry and is responsible for keeping the association between a DID and its DDO document. Based on the DDO structure the smart contract data structure is composed by 6 variables:

- delegates: the actual list of delegates, for each identity maps permissions (i.e. executors address) to those addresses who have it granted.
- delegatesCount: count delegates per identity. Needed to update the minimum quorum required when the number of delegates is less than the quorum set.
- primaryAddressChanged: needed to check whether the primary address associated to that identity is still the trivial delegate or not.
- minQuorum: minimum quorum required to perform a certain operation per identity and per permission
- blockChanged: Needed to retrieve the DID Document from the smart contract
- service endpoints: A list of DID address with the relative service endpoints publicly exposed by that DID.

The `execute` function inherited from Operation Executor is responsible of updating this data structure, in other words is responsible of the write operations. In particular the operations that can be performed are:

- Add a new delegate to an identity with a certain permission.
- Revoke a permission of an existing delegate from an identity
- Add or revoke a service endpoint associated to a certain identity

4.5.5 Credential Status Registry

The Credential Status Registry Smart Contract is an example of Operation Executor. In this case it is used to check and update the status of a credential. From a more generic view point it shows how a service can be added on top of the existing SSI registry and multi-sig operations with extreme simplicity.

The smart contract data structure is a nested mapping like the following:

```
mapping(address => mapping (uint256 => State) ) credentialLlist;
```

which is a list of credential status indexed by the credential id (uint256, unique per issuer), which are indexed by the issuer DID (address). More in details the credential status is a Struct composed by three fields:

- credential_status: the effective status of the credential, could be one of VALID, REVOKED or SUSPENDED;
- status_reason: the reason of the credential status, which is a string of 32 character, to briefly explain the reason of a status;
- time: the block timestamp in which the update of the status has taken place.

The storage of the smart contract is initialized by default with all zero. This means that every verifiable credential when is issued for the first time is valid, and we don't need to write on the smart contract to issue a new verifiable credential. The `execute` function inherited from Operation Executor, is the one responsible to update the credentialList mapping.

4.5.6 Trusted Contacts Management

The Trusted Contact Management compliant smart contract is an Operation Executor that holds a data structure like the following:

```
mapping(DID -> mapping(DID -> Entity))
```

to maintain the association between a DID and the entity which holds that DID, and it is indexed by the DID of whom has published and maintain that list.

The reasoning behind the full mapping is that TCL can be published by anyone, but it is of great importance to state the DID responsible for supporting that list, indeed being this a solution which fully supports and encourages decentralization, it is up to every single party deciding whoever they believe being trustworthy.

The `execute function` inherited from Operation Executor is the one responsible to update the data structure, adding, removing or updating a TCL.

4.5.7 Design Pattern Decisions

Here we describes the design patterns used and the rationale behind them

Circuit Breaker

Circuit breaker pattern is used when the contract may need to be paused from function, either totally or partially, due to a bug found in the code.

The implementation is done on the only contract that has a function which can be called by users, that is the MultiSigOperations contract. Especially, the confirmation of any operation is reverted in case the circuit breaker is active. This would impede any operation from being executed, which also means that no

OperationExecutor contract can have its state changed has no execution can take place.

A stopped boolean drives the pattern, which is modifiable only from the contract deployer.

Access Restriction

Access Restriction pattern is needed when a function should not be called by anyone. That's all the restriction you can give, as there is no way to restrict actual visibility of data or function as everything is clear on the blockchain. This is usually implemented by requiring that a certain pool of addresses are calling the function, reverting otherwise. About contract state the private keyword can be used to avoid other contracts to access it.

In this specific scenario the pattern is widely used. Two main uses are explained below:

1. Some functions can only be executed by those who have certain permissions. This is done by using modifiers such as `actorHasPermission` and `quorumSatisfied` in the MultiSigOperations contract. The former involves asking a PermissionRegistry contract to check if a certain address has permission for a certain identity. The logics by which that permission is given or not is decided by the PermissionRegistry contract. In this scenario there is the concept of delegates.

2. OperationExecutor contract has only one public function (other than the constructor) which can only be called by the MultiSigOperations contract. This ensures that the execution of operations in that OperationExecutor contract are only made in a multi signature manner. A concrete example is the CredentialStatusRegistry contract which can have credentials' status only changed after enough confirmations by those who have permissions.

4.6 Communication

The issue addressed in the following paragraphs concerns the communication standard to be used when two actors of the SSI system talk to each other. For instance, how would a Verifier require specific Verifiable Credential? We have built a simple communication protocol for that. Moreover, the transportation methods used to exchange information is also described in the Transport Methods paragraph below.

4.6.1 Protocol

Between the actors of our system there are 3 different types of communication which could happen based on the processes described below. These 3 types are:

- Attestation;
- ShareReq;

- ShareResp;

Appendix B shows the examples of the three different JWT used in the protocol: They all have 5 fields in common:

- The type property, which states that we are dealing either with an Attestation a Share Request or a Share Response.
- The aud property, which is the DID target to whom is directed the JWT.
- The iat property, which is the timestamp at which the JWT has been created.
- The exp property, which is the timestamp at which the JWT will expire.
- The iss property, which is the DID sender, who has created the JWT. Then the JWT could be signed by one of the delegates who has an authentication permission.

The other fields that are required only by some types are:

- The credentialSubject field, which is an array of signed Verifiable Credential, so they are basically other JWT. This field is required by the Attestation and the ShareResp types.
- The requested field, which is an array of paths to the types of Verifiable Credential requested during a Share Request.
- The callback field, which is the url where the requested Verifiable Credentials have to be sent to.

Each JWT is composed by a header a payload and a signature, in the header there is the algorithm used to encrypt it, in the payload there is the actual data exchanged and the signature is composed as follows:

```
<algorithm of encryption>
  (base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

where the secret is generated by the symmetric key pair composed by one of the public keys associated with the DID in the DDO and the associated private key.

4.6.2 Transport Methods

Taking inspiration from uPort project we have implemented a simple transport by means of QR codes for quick use. Quick Response codes are easy to generate and use. They are a convenient way to store all kinds of data in a small space. A single QR code can hold up to 4000 characters (if alphanumeric only).

It perfectly suites to pack a Verifiable Presentation inside a QR. Up to an average of 3 Verifiable Credentials can be easily held on a single QR. A communication between a user and a website using the SSI would imply the user to scan the QR for any kind of interaction, that is either:

- Download Verifiable Credential issued from a website
- Share a Verifiable Credential to the website willing to verify it

If the interaction was to come from a mobile device, a deep link would encapsulate the information and the user would be redirected to our reference application.

As a future work, we plan to look into push notifications and bluetooth communications for nearby devices.

Chapter 5

Pistis Architecture

This chapter aims at giving a comprehensive description of the architecture used for Pistis. Also some important processes are outlined in order to fully understand the system built.

Taking into account the different actors of the system (i.e. an issuer, a verifier and a user), the full architecture used for the reference implementation is shown in figure 5.1 on the following page

5.1 Mobile Application

The most straightforward component to satisfy the user's requirements is a mobile application, given his portability, user experience and his privacy preserving capabilities. The application communicates with the web server via HTTPS protocol in order to ensure a secure communication. Infura infrastructure has been used to easily connect to the blockchain, even though a physical Ethereum node would be definitely preferred if available. Indeed, in an ideal situation, a verifier would run his own node and use that as gateway to the blockchain instead of relying on a third party service like Infura. However, it still is the most used way to access the Ethereum blockchain without having to setup an entire node by yourself, thus it still makes a good choice where ease of implementation is preferred of very high level security. The infura infrastructure exposes a REST API to communicate with.

The application has been implemented using the react-native framework[16] and Redux[37] which is an implementation of the flux design pattern. It gives the great benefit to build a native app for both iOS and Android system while developing just one application using common web programming langagues and tools.

5.2 Issuer/Verifier System

An Issuer or a Verifier have different requirements[1] than an end user, and a mobile application does not fit them properly. It is often the case where the issuer/verifier needs to expose a web page a user may interact with. A common scenario is when authorization is given to certain users upon sharing and verification

[1] see actors section for a brief summary of issuer/verifier requirements

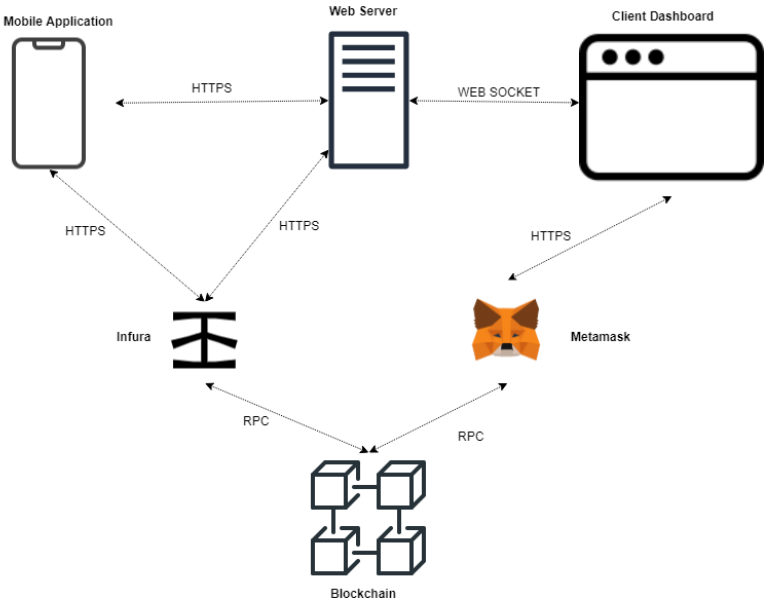


Figure 5.1: Pistis Architecture Overview

of a certain Verifiable Credential. Those checks surely have to be done on the back-end. A reference implementation of a web server and of a client dashboard have been furnished to satisfy these requirements.

The web server communicates with the mobile application through the communication protocol described in the communication section. The web server could communicate directly with the Ethereum blockchain, but again it should set up an entire Ethereum node to do so, which could be an adoption obstacle to anyone who does not already have one. That's why the web server access the Ethereum blockchain through the Infura infrastructure by default.

The client dashboard communicates with the Ethereum blockchain through MetaMask browser extension wallet to allow issuer and verifier to sign transactions with their preferred account, without having to run a full Ethereum node.

Finally the client dashboard and the web server communicate with each other through WebSocket protocol.

5.3 Blockchain Integration

5.3.1 Infura

In order to access the blockchain we would need to set up an entire node of the blockchain itself, which means to download more than 1 TB today. Then we could use a lightweight client which communicates over Https with the actual Ethereum node to make transactions on the blockchain. But again, somewhere you need to set up this node. Infura[24] infrastructure is one of the most used ways nowadays to access the blockchain, and is essentially a service which offers a REST full API to easily access Ethereum node which you don't need to worry about at all.

The only thing you have to do is to set up a project in infura in order to generate an API key to access the service and you are all done. This is how the reference

implementation has been built.

5.3.2 MetaMask

MetaMask[29] is a browser extension that injects a Web3 instance in the user's browser. It also works as a wallet to securely hold your accounts and tokens, allowing to interact with the blockchain without having to run a full node. Both issuer and verifier need such a interface to sign transaction in order to trigger a smart contract functions. Infura could have been used as well, exploiting their services to wrap a pre-signed transaction and send it to the network. However, the keys would have been handled on the backend and only one account would have been used. This way it allows for fast switching of accounts, so that multiple actors of the same organization can interact with the very same dashboard leveraging the user friendliness of MetaMask. In the section future work we explore the possibility of using the same mobile application, used by an end user, to replace MetaMask wallet.

5.3.3 Permissioned Faucet for On-Chain operations

We previously defined the smart contracts needed to run the Pistis system. On-chain write operations have been avoided as much as possible as it would require people to spend some wei[2] to pay for the gas to have the transaction processed by the Ethereum network. Of course, if the system was to be deployed on a permissioned blockchain with no digital currency, for instance,a blockchain system ran by an institution, that would not be the case. However, the Pistis is mainly thought to run on Ethereum and thus that is an issue to be overcome.

It is just not feasible to pretend that each single participant in the network manages his own ETHs. In today's society and in the next 5 years, this just can't be given for granted. On the other side, some operations requires on-chain operations. These are:

- Revoke a VC
- Revoke a DID's public key
- Add delegate
- Remove delegate
- Expose a TCL on a Smart Contract
- Any other possible service which can be added on a later time

In some cases, where institutions or national entities are responsible for the operations, as it can be the case for publishing a TCL or revoking a Verifiable Credential, having weis to spend on a transaction becomes more reasonable. When it comes to

[2]Wei is the smallest denomination of ether, the cryptocurrency coin used on the Ethereum network. 1 Ether = 1,000,000,000,000,000,000 Wei 10^18

some user adding a delegate, however, we needed a way for someone to pay for that user.

There it comes the idea of an ETH faucet which has limited access by a single DID and also requires some authentication (in the manner of sharing a Verifiable Credential) prior to the use of it: Pistis Faucet. It is a simple utility which allows some national entity to expose this service which lands the weis needed to make a transaction to those who need it. The flow is as follows:

1. The Faucet holder sets up the required criteria to allow the withdrawal of ETH from the faucet. That is a regex on the Verifiable Credential/s that the user has to own in order to be eligible to use the faucet. It can simply be a request for a Person VC with a valid National Identity Number in it and issued by one of the recognized service providers (this is automatically processed by the Trusted Contacts Management).

2. The user willing to make a transaction that requires spending some gas, for instance, the addition of a delegate, is prompted whether to use a faucet or to pay for the gas himself.

3. If a faucet is chosen (see below for faucets discovery), the user is simply redirected on the faucet utility page where it is required to share the VCs needed in order to prove his right to use that faucet.

4. Once his right is proven, a request is automatically made to the faucet service containing the operation to be performed (i.e. add delegate, revoke public key etc...), espliciting the smart contract function to be called and the user's DID.

5. The transaction is then automatically performed by the Application.

6. The faucet service will be listening for on-chain events and will look for that smart contract function call performed by that user's DID which previously requested a faucet withdrawal. This ensures the user actually used those weis for what meant. If no transaction appears in the next 24 hours, the user is marked as not eligible anymore for that faucet. Note that this could also be implemented by using a faucet eligibility VC which can be revoked upon unexpected user behaviour.

There will also be a limit on the number of withdrawals one can do per type of operation and per amount of type. That is totally dependant on the faucet holder. These parameters are easily settable in the utility we prepared.

5.3.4 Faucets discovery

The discovery of certain Faucets is done by means of the Service Endpoint spec of the DID Documents. A DID offering such a service can advertise the actual URL where the service is hosted by inserting it as a Service Endpoint with type "faucet".

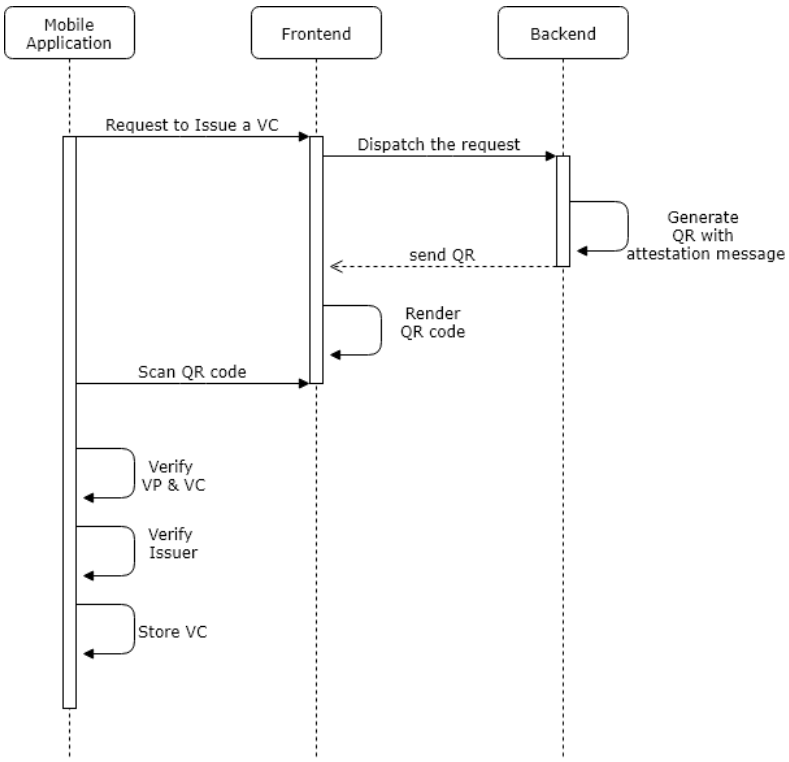


Figure 5.2: Sequence Diagram of the issuing process

5.4 Processes

In this section we will explain in detail what are the fundamental processes that we have implemented in our Pistis project. They can be classified into 6 main processes where there are included all the processes highlighted in the previous section:

- Issue Verifiable Credentials
- Share Verifiable Credentials
- Revoke Verifiable Credentials
- Entity Resolution

5.4.1 Issue Verifiable Credentials

The act of issuing a Verifiable Credential happens in those cases where a user requires some institution to issue a qualification which states some information about the requesting subject. This qualification (or Credential) becomes verifiable if the issuer signs it and releases it following the protocol highlighted in this section. Note that this can be done automatically: a Verifiable Credential can be released upon user request at any time and without operator interaction.

The action flow representing the request and subsequent issuing of a Verifiable Credential is as depitected in the figure 5.2 The back-end of the web app generates

the Verifiable Credential requested, formatted as explained in the verifiable credentials section, signs it with its own private key, wraps it into a verifiable presentation and finally produce a jwt of type "Attestation", and renders it as a QR code or in on of the other formats.

When the user has received the Verifiable Presentation the mobile application performs verifications first on the Verifiable Presentation and the on each Verifiable Credentials:

1. Is the Verifiable Credential/Presentation expired? Checks the exp field of the JWT.

2. Is the Verifiable Presentation sent to me? Checks that the aud field of the JWT is equal to my DID.

3. Is the Verifiable Credential/Presentation signed by the actual issuer of the credential? This is verified by resolving the issuer DID into the DID document through the Pistis DID Registry, and checking that the JWT is signed by one of the public keys with authentication permission inside the DID document.

4. Is the Verified Credential revoked? This is done by calling the credentialList method of the Verifiable Credential Status Smart Contract.

5. If large files are included into the VC, verify their content integrity, as explained in the VC section

6. Verify data selectively disclosed, by comparing their hashes, as explained in the VC section

If one of this steps is not fulfilled than the verifiable credential is rejected.

After having verified that the Verifiable Credentials is ok, the mobile application verifies if the issuer is a trusted one, which means that the mobile application checks if the issuer's DID is associated with a contact already stored in the list of trusted contacts of the application. If not, the application asks the user if he wants to add the new issuer to the trusted contacts list with the information attached in the vc field of the Verifiable Credential.

Finally the Verifiable Credential is stored into the application local storage, following the type path convention specified in the credential itself.

5.4.2 Share Verifiable Credentials

Sharing a VC happens when some service requires the user to be eligible for that service. In order to do so, the user must prove he matches the service requirements. This often happens with websites login where a user accessing the service has to prove to be someone. Nowadays the pair username and password is the most common way to do so, but a Verifiable Credential with the user's Fiscal Code could be enough to prove your identity and thus the user can get rid of the hassle of remembering loads of strings.

The flow representing a verifier requesting some Verifiable Credentials from the user is shown in the figure 5.3 on the facing page

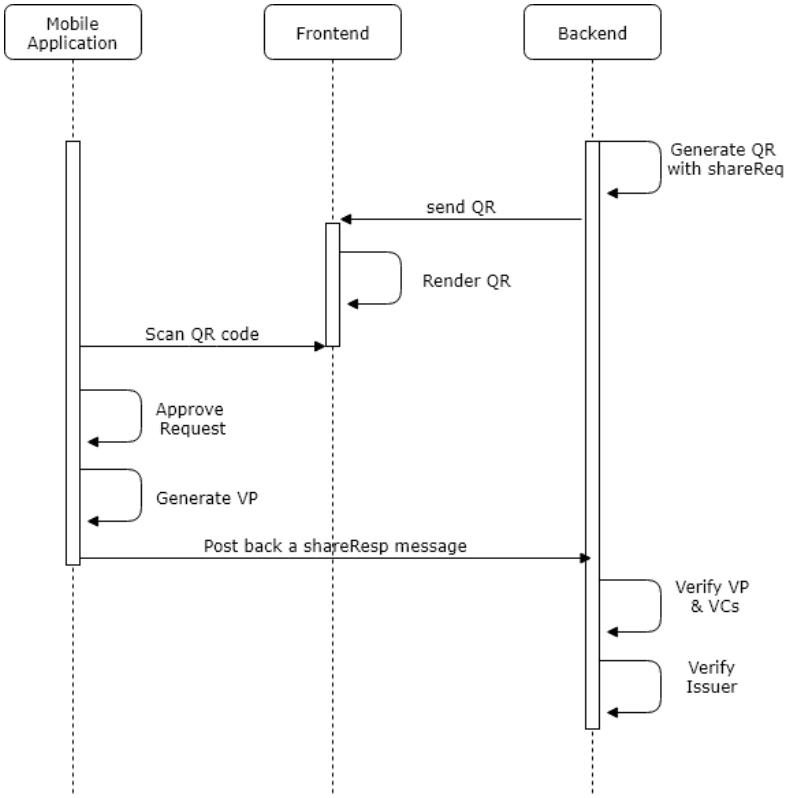


Figure 5.3: Sequence Diagram of the sharing process

Especially, a QR is shown on the screen for the user to scan using the mobile application. The QR carries a JWT which looks like the shareReq one discussed in the previous chapter.

The user is then prompted to accept the sharing of the Verifiable Credentials requested from the service requesting it. Upon acceptation from the user, the requested Verifiable Credentials are packed inside a Verifiable Presentation which is signed by the user and sent to the the callback url inside a JWT of type "shareResp".

The backend is waiting for a response on the callback url stated in the previous request sent to the user, and the authentication is done via the socket id which is used as a challenge text to secure the communication channel.

Then, as in the issue process, the one who receives the verifiable presentation needs to verify the jwt received following the same steps described in the issuing process, first for the verifiable presentation and then for each of the credential presents inside the presentation.

After that the verifier needs to perform verification about the issuer of the Verifiable Presentation and Credential. These are done using the Trusted Contact Management. Explained below.

5.4.3 Revoke Verifiable Credentials

It could happen that an issuer wants to revoke a verifiable credential for some reason. Think to a driver who is caught driving high on alcohol, one of the actions pursued by the law is to revoke his driving license. So it is important to have a

way to revoke, or to change the status of a verifiable credential.

In order to revoke a verifiable credential and to verify its status we have applied the W3C standard, which proposes to insert a pointer to a claim revocation list inside the VC itself. By default the pointer points to a smart contract which expose a getter and a setter function of the credential status.

Inside the verifiable credential there is an object called "csl" (credential status list) like the following one:

```
csl: {
  id: "<id>"
  type: "Pistis-CSL/v1.0"
}
```

Codice 5.1: Credential Status List Object

The type field indicates what version of revocation schema we are using, hence, how to resolve the id field. Currently the only revocation schema supported is a revocation list held by the Credential Status List Smart contract. This is to enhance the flexibility of the protocol to accept multiple revocation schema, for example it could be an URL which publicly expose a revocation list.

The id, indeed, is the identification number of the verifiable credential itself, which is unique per issuer.

The getter function accepts two parameters, the issuer DID and the id of the credential which you want to check.. The getter returns an object "credential status" like the following one:

```
credential status:{
    current status: "revoked"
    status reason: "misuse"
    iat: "23/04/2019"
}
```

Codice 5.2: Credential Status Object returned by Smart Contract

which describe the status of the credential (revoked, suspended, ...), the reason of this particular status and the timestamp of the last update.

The setter function accepts as parameter the credential issuer, the credential id, the status and the status reason. It then updates the smart contract storage.

The setter function, updating the smart contract storage, is a write operation. Hence the setter can not be called directly, but it has to be called by the multiSig-Operation smart contract, which is responsible to check if the caller has the right permission to execute such an operation and if the operation needs to be confirmed by other delegates before executing it

5.4.4 Entity Resolution (Trusted Contacts Management)

The system as described so far manages entities by assigning them an identifier, that is a DID. It can be thought of as a mobile phone which can't save numbers, still works and still makes sense to use it, however it lacks on user experience and

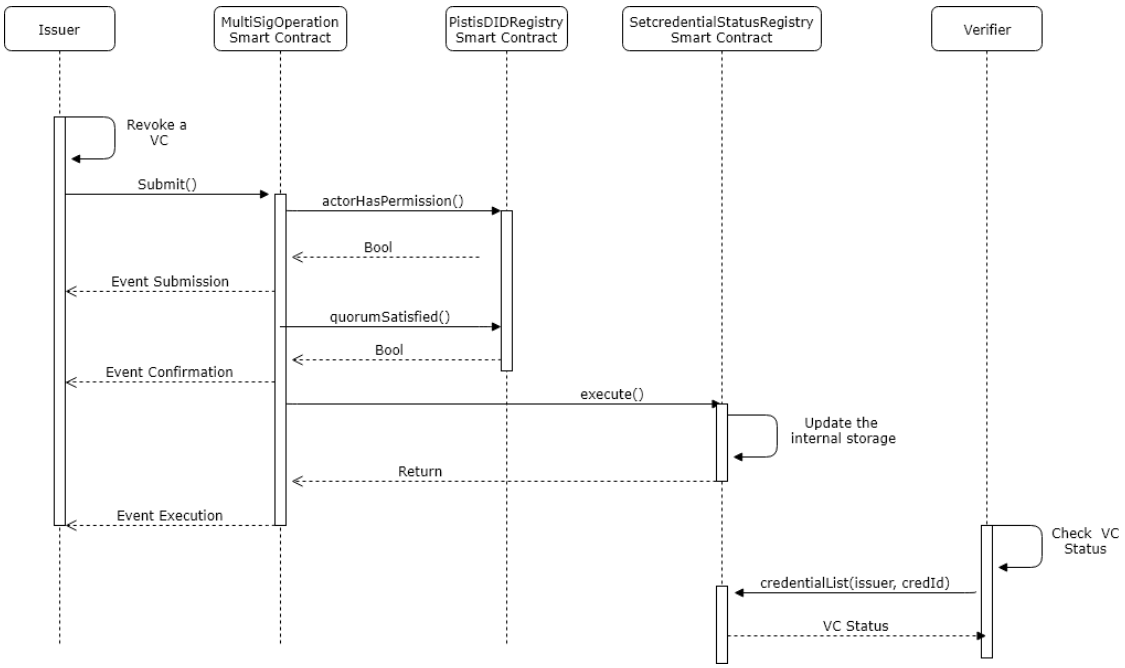


Figure 5.4: sequence diagram which visualize all the passages described in order to revoke and to verify a credential.

it may not be very useful with a large number of numbers to deal with. That is, Pistis as it is does not scale and is not usable.

There it comes the need to associate a real world Entity to those DIDs, for a truly usable system a trust model to scale is needed. Just the way it happens with mobile phones in which you choose whose number belongs to depending on your knowledge and on who you trust. In the same way, in our SSI system, the user has a Trusted Contacts List (TC List) which simply associates an Entity to a DID. However, just like it would happen when you boot up a brand new phone, you would like to recover your contacts, or importing the contacts of someone you trust and who knows around about the same group of people as you.

Before diving deeper into the Trusted Contacts Management we propose, we need to look into some basic data structures to fully understand it. We have previously defined what a DID is, but the Entity structure has not been discussed in greater details so far. An Entity data structure is a simple object which follows the schema.org naming schema. For example the university Politecnico di Milano could be described by the object below:

```
{
  "@context": "http://schema.org",
  "@type": "EducationalOrganization",
  "name": "Politecnico di Milano",
  "address": {
    "@type": "PostalAddress",
    "addressLocality": "Milano",
    "postalCode": "20133",
    "streetAddress": "Via Bonardi 1"
```

```
  },
  "logo": "https://upload.wikimedia.org/wikipedia/it/b/
     be/Logo_Politecnico_Milano.png"
}
```

Codice 5.3: Entity Object

For the reasons explained above, a list might also be populated with data coming from a third party TC List created by someone you consider to be trustworthy. We then come to define a Trust Contact that is a structure like the following:

```
{
  "src": [this, https://.../, 0xDCA7...],
  "did": [null, "did:ethr:0xbc3ae59bc76f894822622cdef7a2
     018dbe353840"],
  "entity": [null, {Entity}]
}
```

Codice 5.4: Trust Contact Object

While the TCL itself is defined as follows:

```
 "@context": "pistis-tcm/v1",
 "tcl" : [{TC#1}, {TC#2}, ..., {TC#N} ]
```

Codice 5.5: TCL list Object

The Trusted Contact List has a context field to account for the protocol version and possible future changes. The actual Trusted Contact objects are then held in the tcl array.

On the other side the Trusted Contact structure has the following fields:

- The src field states three possibilities, indeed a Trusted Contact might be a <DID, Entity> definition as it would be if 'this' is the value of *src* field, as it could also indicate an external source. If the external source is the case, we currently support either an https endpoint or a smart contract deployed on the Ethereum blockchain, that is Trusted Contact List can be downloaded from one of those endpoints. In greater detail, a REST endpoint /tcl would return a JSON object with a TCL as described in this paragraph. On the other side, the smart contract will need to expose a method getTCL(did) to retrieve the TCL associated with that DID passed as param as further described in the Smart Contracts section.

- the did field contains indeed the relative DID if the src has value 'this', null otherwise

- the entity field contains the Entity object relative to that DID if the src has value 'this', null otherwise

A TC List is therefore an ordered list of Trust Contact objects. The ascending order matters in case of multiple Trusted Contacts in the list referring to the same DID, the first one showing has precedence. This can happen when multiple sources

import a Trusted Contact referring to the same DID, and the definitions (i.e. the entity) might match or not.

The component whose job is to fetch the Entity object related to a DID, if known, is called the Entity Resolver. When called, a DID is given as input to the Entity Resolver, that is the DID whose we are trying to find the associated Entity, other than the TCL to look into. Also a localList boolean param is first given as true for what is explained below. The way the Entity Resolver works is by scanning the local TCL and performs the following simple operations shown in pseudo-code:

```
entity-resolver(did, tcl, localList){
    for (tContact in tcl){
        //if the DID is in the tcl passed as parameter return it
        if (tContact == 'this' && did == tContact.did){
          return tContact.entity
      }
      //else if we are parsing the local list and the DID is stored on a REST endpoint
      else if (localList && tContact.src starts with http){
          //first fetch the tcl list from the endpoint
          remoteTcl = fetch tContact.src
          //then call again the entity-resolver passing the tcl fetched
          resolvedEnt = entity-resolver(did, remoteTcl, false)
          // if something has been found return it
          if (resolvedEnt != null){
              return resolvedEnt
          }
      }
      // else if we are parsing the local list and the DID it is stored on a smart contract
      else if (localList && tContact.src starts with 0x){
          //first fetch the tcl list from the smart contract
          remoteTcl = fetch rpc tContact.src
          //then call again the entity-resolver passing the tcl fetched
          resolvedEnt = entity-resolver(did, remoteTcl, false)
          // if something has been found return it
          if (resolvedEnt != null){
              return resolvedEnt
          }
      }
```

```
29    }
30    return null
31 }
```

Codice 5.6: Entity Resolver pseudo code

When parsing a list, in case a Trusted Contact refers to an external source, it only looks into that source if the list is the local one (i.e. parameter localList in the function signature). This is to avoid chains of trust which go too far away from the user's contact, indeed our trust model aims at mitigating the security weaknesses introduced by broadly delegating trust.

On top of the operations above, our implementation allows to search the TCM smart contract for lists published by our trusted DIDs. This is an option which can either be turned on or off by the user. It gives more space for TCLs to be imported, while still keeping the non broad trust delegation as it is still considered a non local TCL and thus treated as discussed.

To summarize, there are few ways the Trusted Contacts list can be populated:

1. Manually adding/removing entries to the list. Note that a pair is simply <DID, Entity> both as defined earlier in the document. Also, every time we interact with some entity they will also claim some information about themselves (that is, they provide the Entity object related to themselves) which a user can decide to add to the trusted contact upon interaction with the Entity's DID. A practical scenario where this would happen is when I interact with some website. There they may assert their own identity providing the Entity object which I can then decide to add to my Trusted Contacts list.

2. Import TCL from an external REST endpoint. We picture a scenario where certain authorities publish a list (either on a smart contract or an https endpoint) with the association between Entity and DID which they believe should be trusted. Specifically each authority will publish the list of entities in its field of competence. In Italy it would mean that the MIUR (Ministero dell'istruzione, dell'università e della ricerca) keeps a list of the Italian Universities.

3. Import TCL exposed by some trusted DID in the smart contract. This can either be done explicitly as Trusted Contact, or implicitly as a back up method if no Entity information has been found in scanning the local TCL.

Then it is also important to keep it in order of precedence to arbitrate collisions.

The component responsible to handle all the mentioned operations it is called the Trusted Contacts Management (TCM). The system just explained is a hybrid between the current chain of trust up to the root Certification Authority and a Web of Trust. That's because we do have some sort of CAs which are entities supposed to be trustable from which we gather some Entity objects (even though nobody is forced to trust anyone), while on the other side each single interaction is subject to the user's intention to trust it or not, also the web of trust keeps building as the user interacts with more and more DIDs.

Taking inspiration from the current certificates chain which we can find in the SSL protocol for instance, there can be one or more Trusted Contacts Root (TC Root) which simply holds a list of pointers to trusted (by the TC Root) entities exposing a Trusted Contacts List (TC List) as described above.

At the very beginning of the set up of a new Issuer/Verifier system there it the possibility to trust some TC Root and import the relative pointed lists. This is a convenient way to kickoff a list of trusted contacts, while it also makes a lot of sense in some scenario like Academia, where the MIUR may publish its list of trusted contacts that universities are willing to import.

Chapter 6

Pistis Components Details

This chapter describes in greater detail the components part of the Pistis ecosystem. Both the mobile app and the issuer/verifier dashboard are explored in terms of functionalities the components need to offer and the way they implement the processes needed for the system to function as described.

Both the mobile application and the web dashboard can handle just an identity at a time. So every operation and information are related to the current identity handled. Whenever a delegate with the right permission[1] wants to perform an operation on behalf of another identity, he just have to switch from the current identity to the desired identity . Currently both the application and the web dashboard does not allow to switch between multiple identity, this will be part of future work.

6.1 User App

The first time a user open the mobile application, a new identity is generated without accessing the blockchain and without the need to spend any Ether to complete such an operation[2]. From now on this will be the identity associated with the mobile application.

The mobile application is composed by 5 components: [3] Credential, Contacts, History, Disclosure and Settings.

6.1.1 Identity Management

An end user needs to be able to manage its identity within the mobile app itself. As we have already seen in the previous sections, manage an identity means edit the DDO document associated with the current identity DID.

That's why the mobile application has a sub-component[4] into the setting component to add and revoke delegates and service endpoints, so far the only

[1] Every time that we refer to permission we are talking about the permissions explained in the Smart contract section.

[2] see the section DID & DDO to have a clarification on how an ethereum address is generated

[3] see the screenshot taken from the mobile app in the appendix A

[4] this component is still to be implemented in the mobile application, please refer to the corresponding identity management component implemented in the issuer/verifier dashboard

editable components of a DDO.

The component is composed by three interactive list:

- The list of the current delegates with the identityMgmt permission. Which is the permission needed to manage the identity and to sign on behalf of the DID associated with the DDO being edited.
- The list of the current service endpoints presents in the DDO.
- The list of the pending operations to be confirmed. This list is needed to allow delegates to confirm an operation that requires more than one confirmation to be approved.

Currently the identity management operations are the only one that delegates can complete on behalf of another DID using the mobile application.

6.1.2 Credential Storing

When a Verifiable Credential is received the mobile app is responsible of storing and visualizing it in a user friendly way while keeping all the characteristics that makes it a Pistis Verifiable Credential.

In order to guarantee all of the above, for each Verifiable Credential, the mobile app stores:

- the decrypted JWT payload [5], with all the hashed information (used for selective disclosure and large files management) replaced by the real information contained in the data and file fields of the Verifiable Presentation. This is useful to visualize all the information in a user friendly way in the Credential Component.
- the corresponding data fields of the Verifiable Presentation containing all the unencrypted data hashed in the verifiable credential.
- the corresponding file fields of the Verifiable Presentation containing the Download Actions describing where to retrieve the file.
- The JWT token itself, which is the one that will be included in the vcl fields of a verifiable presentation when might be shared with someone else.

The data and file field are both needed when the user has to share his verifiable credential with someone else. This is because the Verifiable Credential cannot be changed, otherwise it won't be valid anymore. Hence we need to remember the order in which data were hashed and the order in which large files were included in the verifiable credential at the time it was issued. Storing both of them upon the reception of the verifiable credential, is the easiest way to do it.

[5] see the Verifiable Credential section for more information about what is contained in a JWT payload

6.1.3 Data Sovereignty

In a SSI system the user should always be able to have control over his data, knowing what is sharing and with whom is sharing it, while is actually interacting with someone, but also having a clear view of the interaction completed in the past.

The latter is implemented through the history component which is a history of transactions showing the user all the transactions in the past, with all the details needed: when the transaction has taken place, the transaction type[6], who was the sender/receiver, the actual verifiable presentation transferred.

The former, instead, is implemented in the Disclosure component, by asking the user to confirm every transaction that take place while using the application, showing him the same information which will be then included in the transaction history. Other than this information, when the user share a Verifiable Credential, can select which information want to disclose and which it does not want to disclose (he can choose just the fields that were hashed by the issuer at issuing time).

6.1.4 Issuer Trust

The application will be used just by an end user and not by any issuer or verifier. A generic end user interacts with other DIDs in a direct and non-automatic way. This is what allows him not to have to rely on a Trusted Contacts Management. More easily, what it needs is just an address book to store all the DIDs he interacts with. That's what the Contacts component is, a simple address book that can be updated manually or automatically when the user accept to interact with a certain DID.

6.2 Issuer/Verifier Dashboard

The Issuer/Verifier dashboard is a web application which could be independently used by an issuer or a verifier, to interface in a user friendly way with the Pistis ecosystem to complete the processes described in the Processes section.

The dashboard is entitled to a unique identity and every operation that can be performed will be performed on the identity whom the dashboard is entitled to.

In order to access the dashboard you need to have MetaMask[7], which is a browser extension that works as a wallet to securely hold your accounts, and to interact with the blockchain without having to run a full node.

So whenever someone wants to access the dashboard entitled to a certain identity he has to login into MetaMask and import his DID address into MetaMask. Now if he has the right permission to, he can start performing operations on behalf of the current identity. The dashboard itself is subdivided into two main sections. One which is accessible by anyone without the need of any permission and contains the VC Reader utility and the Trusted Contacts list. The other section can be accessed just with the correct permissions and contains the Credential Management,Identity Management, Trusted Contacts Management and the VC Builder.

[6] refer to the section Communication to know what are the type of transaction.

[7] see section future work to see how and why the mobile application can replace MetaMask

All of the above will be further explained in the following sections.

6.2.1 Credential Management

This is the component where an issuer can see all the credentials that he has issued. From here he can check the status of a Verifiable Credential (it's expired? has been revoked? has been suspended?) and can possibly change the status of the verifiable credential. He can choose between VALID, SUSPENDED, REVOKED and he can add a brief motivation about that change.

Anytime an issuer changes a verifiable credential status, he has to perform a transaction to the MultiSigOperation smart contract, submitting an operation to be executed by the CredentialStatusregistry smart contract. This is done using MetaMask to sign each transaction to the blockchain with the private key held by the MetaMask account.

As we have already explained in the smart contract section, any operation which modify a smart contract storage, requires that a minimum quorum of delegates confirmations is reached in order to be executed. That's why other than the list of credentials there is the list of pending operations. A list which allows delegates to confirm an operation submitted by others delegates.

6.2.2 Identity Management

In this component any issuer/verifier can manage his identity. This means to manage the DID Document, by editing it adding or revoking delegates and service endpoints.

The delegates are subdivided into permission type. A delegate can be in more than one permission list.

Any update to one of these lists, is an update to the DID Document associated with the identity to which the dashboard is entitled. Again each one of this update is performed by submitting to the MultiSigOperation smart contract an operation to be executed by the PistisDIDRegistry smart contract.

As for the credential management, there is a list of pending operations, needed to confirm operations submitted by other delegates.

6.2.3 Trusted Contacts Management

The trusted contacts management component is the one responsible to actually implement the Entity Resolution process [8].

It is composed by a simple list of contacts that the issuer/verifier has decided to trust. It can be updated by adding manually a new contact inserting all the needed information (its DID and an Entity Object, which describes the contact information following the schema.org naming schema) or by importing a list of contacts from either a REST endpoint or from a smart contract like the Trusted Contacts Management smart contract described in the smart contract section.

[8] See the section Processes to have a full understanding of the entity resolution process

The trusted contacts management component allows also an issuer/verifier to publish is own trusted contact list to the Trusted Contact Management smart contract. The publishing of a TCL is done by submitting an operation to the multiSigOperation to be executed by the trustedContactMangement smart contract. As for the previous two components, there is a list of pending operations, needed to confirm operations submitted by other delegates.

6.2.4 Verifiable Credential Builder Utility

Actually creating a Verifiable Credential which follows the schema.org naming schema is not an easy task. You have to navigate to schema.org web site, then go through all the schema.org tree to find the correct type for the credential you are creating. When you have finally found the correct type, you need to choose the correct properties to populate your credential. At any time you have to be sure to respect the correct naming schema without forgetting any required fields. Then there is the struggle of managing large files and enable selective disclosure, populating the *data* and the *files* fields in the right way.

That' s why we have created a utility which guides you during the creation of a Verifiable Credential. The following are the main functionalities:

- A json editor with syntax highlighting and linting, to avoid syntax errors.
- A reset function, which reset the json editor to a Verifiable Credential sample which shows you what are the minimum required fields.
- An Object Viewer which reflects any edit in the json editor and visualize the Verifiable Credential in a tree view, to have a better understanding of the credential itself
- A form with all the minimum required fields necessary to compile the Verifiable Credential, to avoid that the issuer produce incomplete credentials.
- The full schema.org tree, which the issuer can navigate to search the correct main type or the desired property. Each of them has a brief description to help understand what is it about. If the description is not enough the link to the schema.org page is provided. Each type or property can be added to the credential from the schema.org tree. They are added already formatted in the right way, in order to reduce to the bare minimum the possibility of bad formatted credentials.
- An easy way to add a file inside the credential, without having to handle all manually.
- A selection functionality, which allow the issuer to select and hash the fields he want to make selectively disclosable. This functionality automatically add all the hashed fields into the *data* field.

Finally when the issuer is satisfied with the credential created, he can issue the credential by generating the verifiable presentation and the QR to be scanned by the future credential holder. The credential will end up in the Credential Management dashboard section.

6.2.5 Verifiable Credential Reader Utility

Pistis aims at being a truly usable system. The Verifiable Credential Reader Utility aims at being a tool to smoothen adoption and increase usability of Pistis. There will often be the need to simply check the content and the signer of a transaction, both in terms of DID and Entity resolution, from someone which is not part of the SSI system.

To let anyone read a VC even if it has not any software able to read it, there it comes the need for a website which simply shows a QR accepting the root level of types /* (any VC) so one can just scan it and the website can simply show the claim of the VCs received and who signed them. This utility can have a simple name like Credential Reader.

In the dashboard there is such a component which can be used by anyone who access the dashboard, no permission is required.

6.2.6 Trusted Contacts List

The Trusted Contacts List is just the list present in the trusted contact management component. The only difference it is that is publicly available and is not editable.

It is needed in order to allow anyone who use the VC reader to have a look at the contacts trusted by the identity who is currently hosting the VC reader, and by the VC reader itself to visualize more detailed data if a credential was issued by one of the trusted contacts.

Chapter 7

Results

In this chapter we comment on some of the results obtained. In particular, we ended up with a reference implementation as already described in the previous chapters.

We took part in the W3C Credential Community Group discussions and we had the pleasure to be part of some online meeting. It was definitely inspirational and a lot of good ideas come from those meetings.

A concrete result achieved by this thesis work has been the acceptance of the Pull Request into the W3C officially recognized DID Methods[1].

We built different Proof of Concepts alongside our work and had the pleasure to present them in different contexts. We presented the solution for Malta's Healthcare patients' record, as well as some Italian Banks for having some certified bank data. In Appendix C you can find an example of PoC for the Maltese government.

Also, we comment on a few key performance factors and some known system limitations, we deal with some security issues regarding our smart contracts, we tackle the sensitive topic of GDPR compliance.

7.1 System Performance

On such a broad system, performance can be measured from many different points of view and with a bunch of different metrics. We decided to take a few points which can be or already are system bottlenecks in some way, and see how Pistis performs in those.

7.1.1 How much space does a credential take up?

How much space a Credentials takes is quite an important metric to look at, since a user will likely end up with a handful of credentials to hold. Indeed, overhead coming from security and transport protocols have been kept as low as possible to save up some extra bytes. This also matters when thinking about the very nature of a credential meant to be shared and transmitted to other devices pretty often, thus bandwidth impact is also a huge point here.

[1] W3C DID Method Registry https://w3c-ccg.github.io/did-method-registry/

Text in a credential is Base64Url encoded as it is packed inside a Json Web Token, thus we will look at characters used for the different parts of a Verifiable Credential, and then we'll assume space taken up is: $characters * 4/3$ as per RFC-4648 [23].

Starting from the outer layer, we get the overhead given by the JWT. That is 30 chars for the header, variable number of chars for the payload which will be explored in the lines below, 64 chars for the signature.

The payload is dictated by the initial structure of a Verifiable Presentation which is as follows:

```
{
  "iat": 1562600828,
  "type": "attestation",
  "vcl": [
    ""
  ],
  "files": [
    []
  ],
  "data": [
    []
  ],
  "iss": "did:pistis:0xbc3ae59bc76f894822622cdef7a2018
    dbe353840"
}
```

Codice 7.1: JWT payload

That's around 180 chars as a baseline plus the JWT of the credentials which goes inside the vcl field.

What just said leads to a $30 + 64 + 180 = 274$ chars as overhead to transport any number of credentials.

Giving an average size of a VC with one level depth and enough information inside being 600 chars, neglecting the overhead given by hashed data or large files which most of the times only includes an external file, and considering the 210 bytes overhead from header plus payload we get around 800 chars per credential in plain text. Being it all Base64Url encoded a full average-sized credential is made of 1.1 KB.

Therefore, we end up with a single VC inside a VP to be around 1.4 KB which grows up to 1.9KB after turning it into a JWT.

The size of the actual credential text takes most of the space and there is a linear growing in size, which can be considered satisfactory. The main bottleneck will be the data transport which is being dealt with in System Limitations section.

7.1.2 Smart Contracts transaction cost

This is an extremely important measure we need to take into account. Being run on the blockchain, the main system architecture is free in a certain way. What needs to be paid is the transactions in order to change the state of the smart contracts.

Contract function	Gas Usage (gas)	Fiat cost ($)
$0.20		
deploy MultiSigOperations	2,753,261	$5.50
deploy PistisDIDRegistry	1,539,232	$3.07
deploy CredentialStatusRegistry	484,875	$0.97
add delegates (first time)	561,147	$1.12
submit add delegates	227,000	$0.45
confirm and execute	100,000	$0.20
confirm without execution	50,000	$0.10
deploy MultiSigWallet	2,578,288	$5.16
submit add delegate	230,000	$0.46
confirm without execution	78,000	$0.16

Table 7.1: Smart Contracts transaction cost

We ran experiments to see how much the main tasks use in terms of Ethereum gas and we compared against the alternative: our novel MultiSigOperation contract against a generic Multi Signature Wallet. The results are summarized in the table 7.1.

The table shows the most relevant costs to take into account and compares the two approaches: our SSI contracts vs a generic Multi Signature Wallet. Few notes before digging in the figures:

- the Multi Sig Wallet used is https://github.com/gnosis/MultiSigWallet. it is a popular implementation and used in production environments
- the gas price per Gwei is based on the average transfer time (2 blocks or 30 seconds confirmation) according to https://github.com/gnosis/MultiSigWallet
- the fiat cost per Gwei is estimated according to https://coinmarketcap.com/. The final figure is the 0.20 dollars per 100.000 gas used, that is

$$gas\_used * 100.000 * avg\_gas\_price * ether\_to\_dollar\_price$$

.

Once that is clear, we can appreciate how on average end-user usage of such a system, it is four times cheaper to use our contracts. An average scenario is considered one that sees an end user creating a DID and then changing the address owning the identity into a newly generated address associated with his DID to a safer multi signature wallet. This is the common way, the one that also uPort uses, which involves the deployment of a new Multi Sig Contract for that user so that he can handle operations in a multi signature manner. Indeed, the cost of deploying such a contract is around $5.2. By deploying the contract with the right params and the desired delegates the user reaches an initial state by which operations can be handled by having a quorum of trusted parties endorsing the operation before execution, that is the desired state.

On the other side, in Pisits it only takes to deploy the contracts once, and NOT per user such as the case of the MultiSigWallet. In Pistis scenario, the user would execute the addDelegate function with the desired amount of delegates. That cost is roughly $1.10 dollars, being the first time addDelegate called followed by a submitAddDelegate and a subsequent confirmation executed. That is around 5 times cheaper.

Bear in mind that user experience (which is arguably better in Pistis as it does not involve switching to a third party multi sig enabled wallet) is not taken into account in this comparison.

7.2 System Limitations

Just like any software, Pistis does not come without limitations. Below are described the ones addressed as most relevant:

7.2.1 Data Transports

QR transport with the lowest resolution cannot surpass 3KB on single request. This limitations impacts a practical use case if the Issuer is willing to share a lot of Verifiable Credentials at once. Giving an average size of a VP of 274 bytes of overhead plus 1.1KB for each credential, we can carry up to 2 average sized credentials at one, or 4 little credentials (roughly half of an average sized credential). In most common scenarios this is not a great issue, however it is important to state this limitation and to provide a potential workaround.

Indeed, a possible solution is the use of push notifications as a transport layer. This solution, although beating the maximum size wall, would imply to use a server to rely the notifications. These services create one more hop that becomes a single point of failure if not structured properly with some replication in order to increase availability. Another possibility comes from the splitting of the data into multiple QRs to be scanned one after the other. Usability would be a little penalized in such a scenario but still the user friendliness of QRs would be preserved.

7.2.2 Data Backup

Pistis does not rely on any third party service to function, therefore the user can freely store his data on his mobile phone. When sharing, direct upload of a Verifiable Credential to an https endpoint is used, thus the data only need to transact through the world wide web but without the need to ask permission or access some other service. On one side this is an extremely important feature and truly gives users control over their data. On the other side it could be risky and not very user friendly to only rely on the personal mobile phone to store all the data, a backup is somehow needed. What happens when the user loses or breaks his mobile phone? All the credentials are lost and the user would need to request back all the credentials, just like it would happen when dropping your physical wallet. Well, technology comes in handy and allows to overcome this current limitation.

The problem actually goes far beyond. How do we store application data, such as Verifiable Credentials, in a way that is controlled and administered by us, encrypted by default from parties that may not have our best interests in mind, and most importantly in a standards-compliant manner?

There is a handful of groups working on it like Hyperledger Aries[36], Decentralized Identity Foundation's Identity Hubs[18], at Solid/Inrupt[5], and elsewhere in the world. Each of them in their own way offer a what is being addressed as Secure Data Hubs. Taking from the Abstract of the Identity Hub, DIF's specification:

> "We store a significant amount of sensitive data online such as personally identifying information, trade secrets, family pictures, and customer information. The data that we store should be encrypted in transit and at rest but is often not protected in an appropriate manner. This specification describes a privacy-respecting mechanism for storing, indexing, and retrieving encrypted data at a storage provider. It is often useful when an individual or organization wants to protect data in a way that the storage provider cannot view, analyze, aggregate, or resell the data. This approach also ensures that application data is portable and protected from storage provider data breaches." [46]

A Secure Data Hub is what perfectly fits as a self-sovereign back up place for Credentials and other data currently only locally stored in Pistis. This would be a great leap forwards in terms of usability for end users.

7.2.3 Need to fund Ethereum transactions

Ethereum transactions need to be funded. The Ethereum clients work 24/7 for us but they want something in return. Pistis is based on Ethereum, and even though for many operations it is not needed to transact through the blockchain, there are cases where it is not avoidable. Adding delegates and revoking credentials are the most relevant for respectively end users and issuers. This implies all system users to hold some Ether to pay for these operations. There are a variety of concepts that users need to know in order to be able to safely use a crypto wallet and there are a lot of challenges still to be overtaken before every-day use can become a common thing [13].

We already provided the idea of a Permissioned Faucet to smooth the way for transparent transactions payment without the need of holding Ether.

Possible alternative to the faucet is to have anonymous transactions which only carry the signed payload and that can then be paid by a Transaction Relay service. This solution would speed up the time to run a transaction as there is no intermediate step of asking for ether, way for the ether and then perform the actual operation.

Still, someone needs to pay for those fees and in such a widely system a business model which allows for a Nation or any other Entity willing to carry those expenses need to be found. A big discussion may be opened up here and it would simply be off topic from the work of this thesis. A permissioned blockchain with consensus protocol that doesn't require paying transactions fee as an incentive for miners to keep the network up and running would surely solve the issue.

7.2.4 Offline support

Offline support is not a vital capability, even though it may turn out to be quite useful in some situations. Verifiable Credentials are locally stored, which scores a first point in favour of offline support. The process of verifying the signature and the content of a Credential, however requires to check whether the signing address has permission over a certain DID and that the credential is not being revoked or suspended. The former involves the resolution of a DID document which needs to be done throughout an Ethereum node. The latter involves checking the Credential Status Registry contract, and again an Ethereum node is needed. Therefore fully offline capabilities can't be achieved in lack of an internet connection. However, verifiable credentials are still fully visible even if no internet connection is present.

7.3 About Smart Contract security issues

Smart Contracts running on a public blockchain are quite a novel concepts. Unlike other blockchain systems[2], Ethereum smart contracts are not capable of being formally verified against security vulnerabilities. Indeed, in the past this has shown to be quite an issue. We tried our best in ensuring smart contracts security by looking at a list of common attacks relevant to our contracts and see how we can mitigate or just be safe from it:

7.3.1 Re-entrancy Attacks

A Re-entrancy Attack, according to Solidity documentation [15], could be present anytime there is an interaction from a contract (A) with another contract (B) and any transfer of Ether hands over control to that contract B. This makes it possible for B to call back into A before this interaction is completed. If contract A has not yet modified its internal storage, when contract B calls back A, then B could recursively call A an undefined number of times (until gas limit is not reached) drawing for example the contract A balance.

In our smart contracts we do make external calls to other contracts, but no one of those handle Ether transfers. Besides that, any external call which we make is towards a contract which is known at deploying time and can not be changed. Hence no unexpected behaviour could happen. The only call towards an unknown contract happens in the method `executeOperation` of MultiSigContract. This call doesn't handle any transfer of Ether and it is made at the end of the function, when all the internal storage update have been already made. Indeed an attacker could recursively call back our contract, but it will not be able to act maliciously.

7.3.2 Integer Overflow and Underflow

According to the solidity documentation [15], an overflow occurs when an operation is performed that requires a fixed size variable to store a number (or piece of data) that is outside the range of the variable's data type. An underflow is

[2] see Tezos blockchain for instance

the converse situation. These situations are problematics when an integer variable could be set by user inputs.

The only function which accepts integer variable as user input is the `confirmOperation` in MultiSigOperation contract. In this case the function accepts in input an uint256 as an identifier number for an operation to be confirmed, if this variable underflows or overflows there are no problems. This is because if the sender could not confirm an operation already confirmed or executed, and can not confirm an operation which is not already been submitted.

7.3.3 Denial of Service by Block Gas Limit (or startGas)

A Denial of Service by Block Gas Limit could happen when the execution of a function requires more gas than the Block Gas Limit. This could easily happen when contract functions works with unlimited size array or string.

In the contracts we do make use of un-sized arrays, primarily for future extendability to allow the execution of unknown function in the execute pattern. The important thing is that we do not loop over them, and we don't need to do that, because an Operation Executor knows always in advance which parameters is receiving and in which position they are. Basically the array is used just as a universal container for unknown parameters, which the executor knows.

7.4 GDPR compliance

Until May 2018 there was no proper law enforcing a certain usage with data at a European level, but just a directive giving out guidelines, that is the EU Data Protection Directive. With the General Data Protection Regulation (also known as GDPR) a new legal framework has taken over and it has shown to be very strict and penalties are quite a big deal.

GDPR aims at protecting personal data and how companies deal with data they collect, process and eventually destroy. The other target GDPR has been aiming at is the free data circulation among the European countries. Personal Data is defined in article 4 of the law:

> 'personal data' means any information relating to an identified or identifiable natural person ('data subject'); an identifiable natural person is one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person; [33]

Addressing what personal data Pistis deals with is the first milestone to be achieved. Trivally, all the information inside a Verifiable Credential are considered personal data. On the other side, DIDs and DID Documents being personal data is definitely a more intricate case which needs closer attention. The issue here comes when reading recital 26:

> "[...]The principles of data protection should therefore not apply to anonymous information, namely information which does not relate to an identified or identifiable natural person or to personal data rendered anonymous in such a manner that the data subject is not or no longer identifiable." [35]

That is, if DID were truly anonymous it would not be considered personal data, taking it out from GDPR appliance. Unfortunately we cannot ensure this is the case and a jurisdictional study should be carried out, going out of scope of this work. We'll keep the discussion with DIDs out of the way and only focusing on Verifiable Credentials that surely fall under the category of personal data.

Being our system Self-Sovereign, it naturally finds itself a good fit with GDPR, indeed many of the rights stated in the law are definitely shared by Pistis. The law wants to return more power to data owners, stating eight main specific rights [27]. However, it takes a risks-based approach to data protection, outlining certain data subject rights without dictating how these principles should be enforced.

We mainly look at the Articles 12-23 of the Regulation[34], that is the Rights of the data subject. We'll go through them one by one and see how Pistis deals with each:

Right to be informed:

This provides transparency over how your personal data is used.

Blockchain is all about transparency. Code enabling the core features of the system would surely be fully open-source. The Ethereum ecosystem includes a service, Etherscan, which allows to explore the transactions being executed in the Ethereum blockchain and to see the data created by the blocks. That includes Smart Contracts. Etherscan allows developers to upload the source code of a Smart Contract deployed at a certain address, and it will match the compiled code with that on the blockchain. This way, just like contracts, a "smart contract" provides end users with more information on what they are "digitally signing" for and give users an opportunity to audit the code to independently verify that it actually does what it is supposed to do.

Right to access:

Provides access to your data, how it is used, and any supplemental data that may be used alongside your data.

In some scenarios this is outsourced to those you share your credentials with. Let's say I share my diploma VC with an employee, it would be up to them to make sure they treat my data properly. Looking at Pistis as it is, it grants access simply because users hold their own credentials.

Right to rectification:

Your right to have your personal data rectified if it is incorrect or incomplete.

This is not granted by the system as it is. Indeed, a credential cannot be tampered with once created and thus rectification can be done by issuing another credential and revoking the previous one. This needs communication between the issuer and the user willing to rectify the data.

Right to erasure (or the right to be forgotten):

Your right to have personal data removed where there Is no compelling reason to store it.

In terms of Verifiable Credentials again that goes out of the system native capabilities. Who you share the credential with has to comply with GDPR in turn. Pistis alone does not store any Verifiable Credential on the blockchain, making this requirement achievable. One exception might be the revocation status written on the smart contract. While there should be no explicit reference to the user, it is up to the issuer to be compliant.

Right to restrict processing:

You can allow your data to be stored but not processed. This also involves having personal data being just what is needed and relevant to the processing being made.

This goes under the Data Minimization principle of SSI. We believe Pistis just exactly accomplished this by allowing selective disclosure.

Right to data portability:

You can request copies of information stored about you to use elsewhere, such as if applying for financial products across a number of vendors.

Having credentials aiming at being interoperabile, portability is a must in our system. Being fully W3C compliant is the first fundamental step we took on the road to interoperability. On the DID side there is a lot of work going on through the community for different DID methods interoperability, on top of the already standardized Universal DID Resolver we are compliant with.

Right to object:

You can object to your data being processed. One example may be in that you object to your data being used by direct marketing organizations. If you object, the regulation specifies they must comply.

Verifiable Credentials are held by users in the first place. Others that can access those data are the issuer of that credential itself and those who the users has shared that credential with. Objections have to be carried by other channels which again is delegated to those who process verifiable credentials information. Pistis itself does not go against the regulation from this viewpoint.

Rights to automated decision making and profiling:

You can object to automated decisions being made based on your personal data. Automated means without human intervention. An example may be online shopping habits being determined based on previous online behaviour.

Just like the previous point, Pistis does not involve automated decision making out of the box, even though automatic Verifiable Credential processing would likely being extensively used in a real world scenario. However, being compliant with the regulation would not be Pistis's concern but processors'.

Chapter 8

Known Alternatives

Surely Pistis is not the first of its kind and we are not the first ones to give a reference implementation of an SSI based system. There are many different projects under development, varying in maturity, blockchain usage extent, geographical location of the team and intended scope of the project. We valued many of them before starting our work, choosing the most mature and SSI compliant ones, and in the following lines we carry out a brief comparison giving out a short description of the project highlighting the differences against Pistis. It is worth noting that some of them are not focused on credentials management only, but aim at being broader systems as they intend to handle digital identity as a whole.

8.1 Ethense

Ethense [26] [14] is the most similar to our solution. Indeed, it builds on top of uPort and is backed by Consensys. The main goal of the project is the issuing of diplomas in the form of Verifiable Credentials[53] and/or Open Badges[22]. The issued credentials are sent to users (i.e. students) via emails. The email could contain both the QR with the Verifiable Credential or an Open Badge. The student can then scan the QR with the uPort app in order to import the verifiable credential in his uPort wallet, or share the Badge compliant devices.

A great benefit of such a solution is the use of both Verifiable Credential and Open Badges standards as it would greatly benefit an initial adoption.

However, the project is no longer actively supported by Consensy Team and the development has been at a standstill for more than a year. On top of this it is completely based on uPort, leading to a rejection of Ethense a good solution for the same reasons we drove off uPort project, as explained in Chapter 2.

8.2 OpenCerts

The project is actively being tested in Singapore for the issuing and verification of academic credentials.

> "OpenCerts [31] is the umbrella trademark under which we have released a few key components:

1. An open source schema for publishing educational credentials
2. A set of tools for generating cryptographic protections for educational credentials
3. An online website for verifying the authenticity of OpenCerts files." [30]

Under the hood, it creates a digital certificate, then writing its hash on-chain. This allows an easy verification process as it only involves hashing the plain text credential and compare it against the hash stored on the blockchain. That proves the certificate has not been tampered with and also gives a guarantee on the time of issuing by exploiting the timestamped version of the certificate on the blockchain.

They do use DID to associate a credential to an identity, but their identity registry, at the time of writing, is a json file hosted directly by OpenCerts, which means that is not decentralized at all. Currently they are working on decentralization adopting the same concept of DNS (Domanin Name Server) as an identity registry.

When it comes to the standard used for the credentials, it relies on Open Attestation based on Json schema [32]. The huge downside is that it is not supported by W3C, nor any other standardization group.

Hence we rejected OpenCerts because they don't follow a standard for the Credential, but they propose one. Also, the solution adopted for the identity management is far from the concept of Self sovereign Identity, indeed it even lacks decentralization at all. Another huge downside is the extensive use of the blockchain forcing issuers to write on blockchain each time a credential is issued. Our approach, instead, limits writing on blockchain in very few cases, in order to minimize costs of transaction fees and not to saturate a system which is not capable of high Transactions Per Second (TPS).

8.3 Blockcerts

Blockcerts [8] is an open standard for creating, issuing, viewing, and verifying blockchain-based certificates. It currently uses both Bitcoin and Ethereum blockchains.

Just like OpenCerts, the main difference against Pistis lies in the fact that it involves writing the certificate hash on the blockchain, in order to guarantee that they are tamper proof.

Also, it does not address the problem of associating blockchain addresses to a real entity as we have done using the TCM. Citing their FAQ page:

> "Blockcerts has a claims-orientation to identity. This means that identity is always self-curated by the individual through the claims about themselves that they disclose. All claims have to be assessed in some manner. Those claims that are blockchain verifiable are guaranteed to represent what was originally issued. So, Blockcerts is not attempting to prove identity directly. In other words, this solution does not certify the mapping of public keys to individuals or organizations. Further, there is no registration process in this system, so any issuer may issue

> certificates and recipients may provide any Bitcoin address. However, it is in the issuer's and recipient's interest to provide public addresses they own, because this is the only way either can demonstrate ownership of or revoke certificates." [7]

8.4 Accredible

Accredible [1] is a platform quite similar to the previous two solutions, which aims to adopt more than one standard for the credential structure in order to be more flexible and easily adoptable.

Like OpenCerts and Blockcerts, it writes on the Bitcoin blockchain the certificate hash to guarantee that they are tamper proof.

Differently from the previous two solutions they use both Open Badges and an internal standard called Digital Certificate as the standards of their credential. As explained in Chapter 2, Pistis aims at broad adoption and interoperability, thus adhering to a global standard is a key point. Another hint that led to rejection is the frequent writing on the blokchain. The cost of doing so would badly impact system scalability.

8.5 Metadium

Metadium [28] is not a simple platform or tool like the ones presented just above, it is a whole new blockchain ecosystem.

It makes use of DID specifications[51] provided by W3C standard. They also have their own DID formally registered at the W3C-CCG DID Method Registry[52]. Creating a DID involves deploying two smart contracts to the Metadium blockchain.

In Metadium there is no such a concept as credential, it is a matter of associating attributes to the DIDs. These are not publicly visible by everyone, the user needs to give permission in order to share his personal attributes, and select what others can see, enabling selective disclosure.

Overall, Metadium is a quite different approach from the one we have adopted. The main difference relies on the need of deploying a brand new blockchain. We totally disagree with such a system as setting up a global chain is quite a big deal, and, failing at doing so it would end up to be a totally centralized system under the control of the Metadium team. Also adoption of a system with this prerequisite surely has a slower adoption rate. Instead, our solution is based on the more tested and widely adopted Ethereum blockchain.

8.6 Civic

Civic [9] is a platform that aims at fully digitalizing the identity using the Bitcoin blockchain.

Currently, the team behind the project is running a platform called SIP (Secure Identity Platform) where the user can create his own identity using a mobile application, the Civic app. Before creating a new identity the user needs to go

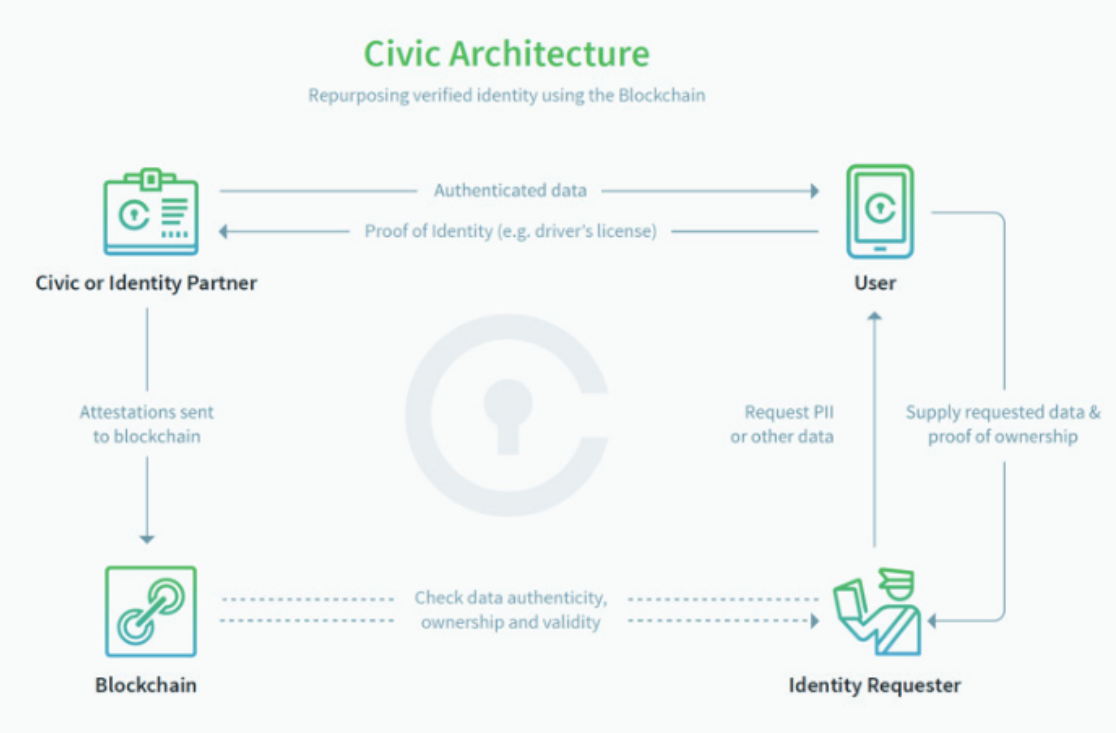


Figure 8.1: Current Civic Architecture

through a KYC (Know Your Customer) process, providing personal data to the application, which is in turn reviewed by Civic or one of its partner. If the data are successfully validated, they are hashed and stored on the blockchain. This is achieved by embedding these data in a new Bitcoin address, created by arithmetically adding the hashed data to the private key of the user. This way they ensure that:

- the data are tamper proof
- they are associated to a certain user,
- only the user can access them, or give access to who he wants.

On top of that, a new ecosystem is under development at Civic. It involves the use of a token (the CVC) in order to incentive all the parties in the ecosystem. It runs on top of Rootstock (RSK)[38], a layer on top of the Bitcoin blockchain to bring smart contracts support to it. The token can then be spent inside the ecosystem to buy services from Civic itself or from other service providers inside the ecosystem.

Again the main difference lies in the need to write on the blockchain to enrich the attributes associated with a user. That has huge costs as already stated in the previous sections. Another key difference against Pistis is in the use of the token as an incentive. We believe the main driver for such a choice is business factor rather than true technological benefits, that is not something that will benefit the ecosystem. Also, Civic forces users to perform a KYC to enter the ecosystem which is currently audited by Civic itself. This is a clear point of centralization, which turns Civic away from a truly SSI solution, giving the auditors the power of censorship.

8.7 ION

ION is a solution developed by Microsoft together with DIF (Decentralize Identity Foundation).

> "ION is a public, permissionless, Decentralized Identifier (DID) network that implements the blockchain-agnostic Sidetree protocol on top of Bitcoin (as a 'Layer 2' overlay) to support DIDs/DPKI (Decentralized Public Key Infrastructure) at scale". [19]

Their solution has two main differences compared with Pistis:

1. It doesn't store anything on the user devices, instead, they use Identity Hubs[46] as a secure place to store users' data following a semantic data model, where data are represented as JSON schema.

   From the Azure Blog:

   > "Identity Hubs are decentralized, off-chain, personal datastores that put control over personal data in the hands of users. They allow users to store their sensitive data—identity information, official documents, app data, etc.—in a way that prevents anyone from using their data without their explicit permission. Users can use their Identity Hubs to securely share their data with other people, apps, and businesses, providing access to the minimum amount of data necessary, while retaining a record of its use."[43]

2. Secondly, they use the sidetree protocol to anchor tens of thousands of DID/DPKI operations on a target chain (in ION's case, Bitcoin) using a single on-chain transaction. This allows to build a solution that scales.

   This is a valuable solution, however there are some criticalities in the use of Secure Data Hubs to fully store users' data. We believe that giving full control of data means storing it locally, and have a system such that just for backups. Also, the Sidetree protocol is quite an overhead to carry along, which is needed in ION's scenario as the blockchain usage is quite extensive. As already states we do not agree with extensive use of blockchain, indeed Pistis limits the use to the bare minimum.

8.8 Sovrin

Sovrin [44] is the major competitor of uPort in terms of maturity and adoption.

Differently from uPort and Pistis, Sovrin runs its own DLT based on Hyperledger. Reading the blockchain happens in a permissionless manner, however the writing is limited to some.

Like Civic they have their own token to create trust and incentivize the good behaviour between parties. From the Sovrin white paper:

> "The result is a marketplace where any source of trust—from a government to a family—now has an incentive to realize value from helping

> build trust with others. And those who have earned that trust—the identity owners—can now benefit from the ability to transfer that trust to other relationships. Verifiers, for their part, can now take advantage of a vastly expanded marketplace for trust information—a marketplace in which issuers are constantly competing to fill any "trust gaps."" [45]

When it comes to the standard used, they completely adhere to the W3C verifiable credentials, like Pistis does.

The main pain points identified are the use of a brand new blockchain and the use of a token to incentivize the creation of an ecosystem. They are what made us reject Sovrin at last. It has already been expressed our negative judgment about those points in the previous sections.

Chapter 9

Conclusion & Future Work

9.1 Future Work

What we see as the immediate upgrade of such a system is the extension from Credentials Management only to a full fledged digital identity system. The gap to be filled in order to take this leap forward is not really that big as can be seen from the comparison with other SSI alternatives being Pistis similar in terms of features needed to be able to fully handle a digital identity, also thanks to the multi purpose smart contract and the versatility of the system as a whole.

Below we discuss what we see as possible improvements for Pistis.

9.1.1 Explore pairwise DIDs

Conceptually, DIDs can fall into two classes: public DIDs and pairwise DIDs. Public DIDs are IDs that users choose to knowingly link themselves with data intended for the public—for example, a small bio that includes a photograph and a brief description. Public DIDs are suitable if you intend an activity or interaction to be linked to yourself in a way that can be verified by others. But having everything you do tied to a single DID and traceable across the web poses serious privacy and safety risks. This is why pairwise DIDs are useful. Pairwise DIDs are generated whenever users want to isolate their interactions and prevent correlation. For many users, pairwise DIDs might be the primary mechanism they use to conduct identity interactions. It would be reasonable to explore the pairwise DIDs to be used by end users in Pistis.

9.1.2 Implement Full Key Management

Being a well known theme we decided not to provide our implementation, but just to reuse one of the battle tested solutions. But we didn't implement it yet.

Both the mobile application and the web dashboard should implement the well known key management used by every crypto wallet. That is based on BIP39 [6], the mnemonic phrase generator used for the first time for bitcoin wallet and then used for all the other wallet.

9.1.3 Improve Selective Disclosure

A possible improvement to selective disclosure could be obtained exploiting the Merkle Tree.

Every time a VC is created, alongside it also the merkle tree of its csu field is created following a parent first order. This means that the csu field of the VC get parsed and every key/value pair is hashed and inserted as a leaf of the Merkle tree. If the value is an object, then first is inserted the complete object and then its children. Finally, following the merkle tree strategy, the leaves are hashed in pairs and the resulting hashes are hashed again in pairs until just one hash remain which is the merkle root. The final result is a VC with the merkle root as value of the csu field.

Every time the subject of the VC wants to share such a VC he has to select which field wants to selectively disclose, creating a subset of the original VC, then in the data field of the Verifiable Presentation will be inserted just this subset with the key concatenated with the index of the leaf that they occupy in the merkle tree, plus an array of the needed hash to compute the merkle root hash.

Upon the receiving of a VC, it has to be parsed and if a leaf is missing (the difference between two consecutive keys is more than one), then the first $(currentKeyNumber - previousKeyNumber - 1)/2$ hashes are retrieved from the array of hashes and used to compute the parent hashes until the merkle root is retrieved, which is finally compared with the one inserted in the csu field.

9.1.4 Sign Any Transactions with Pistis Mobile App

In an ideal scenario, Pistis mobile app can be used to sign any kind of transactions. It is quite relevant in some scenario where Issuer/Verifier dashboard overlaps on an end user needs, and the user may end up needing a duplication of wallet in both Pistis mobile app and Metamask to interact with the dashboard. Having the mobile app sign any Ethereum transaction would allow to get rid of Metmask so that the user can have full control over his keys within the Pistis mobile app.

While this may not be a great deal in many cases, duplicating wallets reduces security and usability, two key factors we have had in mind during the whole work.

9.1.5 On-Chain Automatic VC Verification

Solidity has a built it function, `ecrevocer`, which returns the public key that signed the input data using elliptic curve cryptography. If the input data to that function was a Verifiable Credential, the public key that signed the relative JWT can be recovered, a DDO can be wrapped around the VC signer, and any other operation might be performed on that credential.

A smart contract capable of doing this kind of automated VC verification (or any other sort of operation on that credential) would enable use cases of extreme interest such as the ability to add a certain criteria to the transfer of fungible or non-fungible tokens. Those criteria can be certified by the sender/receiver by proving the ownership of a certain verifiable credential.

One major limitation to this may be the gas limit per block imposed by Ethereum. It currently runs at 8000000 gas[1].

9.1.6 Smart Contracts Improvements

This version 1.0 that we currently have in our reference implementation has shown to be working fine and it scores great in terms of costs as well. However, improvements are always possible and we do have some in mind:

- Operations need an expiry time to avoid unexpected results. A maximum of 24 hours should be allowed to confirm an operation otherwise it must be voided.
- Settable minimum quorum. Currently the MultiSigOperation smart contract can be deployed with a default_quorum parameter. However, it may be the case that each DID needs to set a different quorum based on its specific scenario.
- MultiSigOperation contract contains a generic Operation struct whose task is to pack any kind of parameters a function may need. Currently they are packed into dynamic arrays, however there are potentially more efficient ways to explore. One of them could be packing the params into a single bytes argument and then use low level EVM opcodes to fetch back the needed params after casting. It would surely be a neater solution, whereas performance needs to be tested.

9.2 Conclusion

Looking at the broad picture all over the world we found out that there is a great barrier which stops any kind of new digital system of this kind, and especially SSI based systems, that is due to the legacy systems being used and the previous economic effort carried on in order to get to this point. Especially architectural costs incurred (see Italian SPID server to carry the load), or current jobs created (see CAs created in Dubai and that are not willing to go away) are there and are not happy to be discarded and be replaced by a blockchain and newer systems of that kind.

We really believe it works, however there is a non-trivial barrier to take down before having an SSI system to gain traction.

We believe Pistis can give a nice guideline on the progress of the Self Sovereign Identity approach, at the very least just by spreading the word about the new extremely promising paradigms to invert the centralized and dangerous trend of centralized identity management approaches.

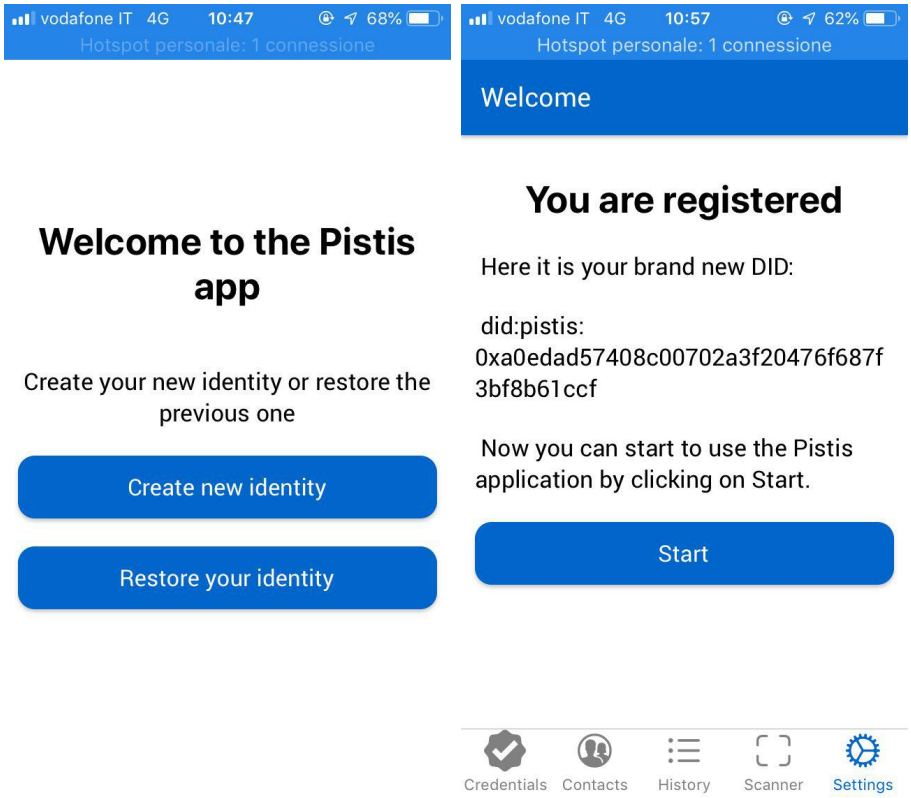
At the end of the day we believe this work achieved, even though just at a Proof of Concept level of maturity, a system protecting data privacy of identity owners and enabling seamless, efficient credentials exchange.

[1] According to https://ethstats.net/

Appendix A

Screenshots

A.1 Mobile Screenshots



(a) OnBoarding Screen **(b)** Welcome Screen

Figure A.1: First time using mobile application

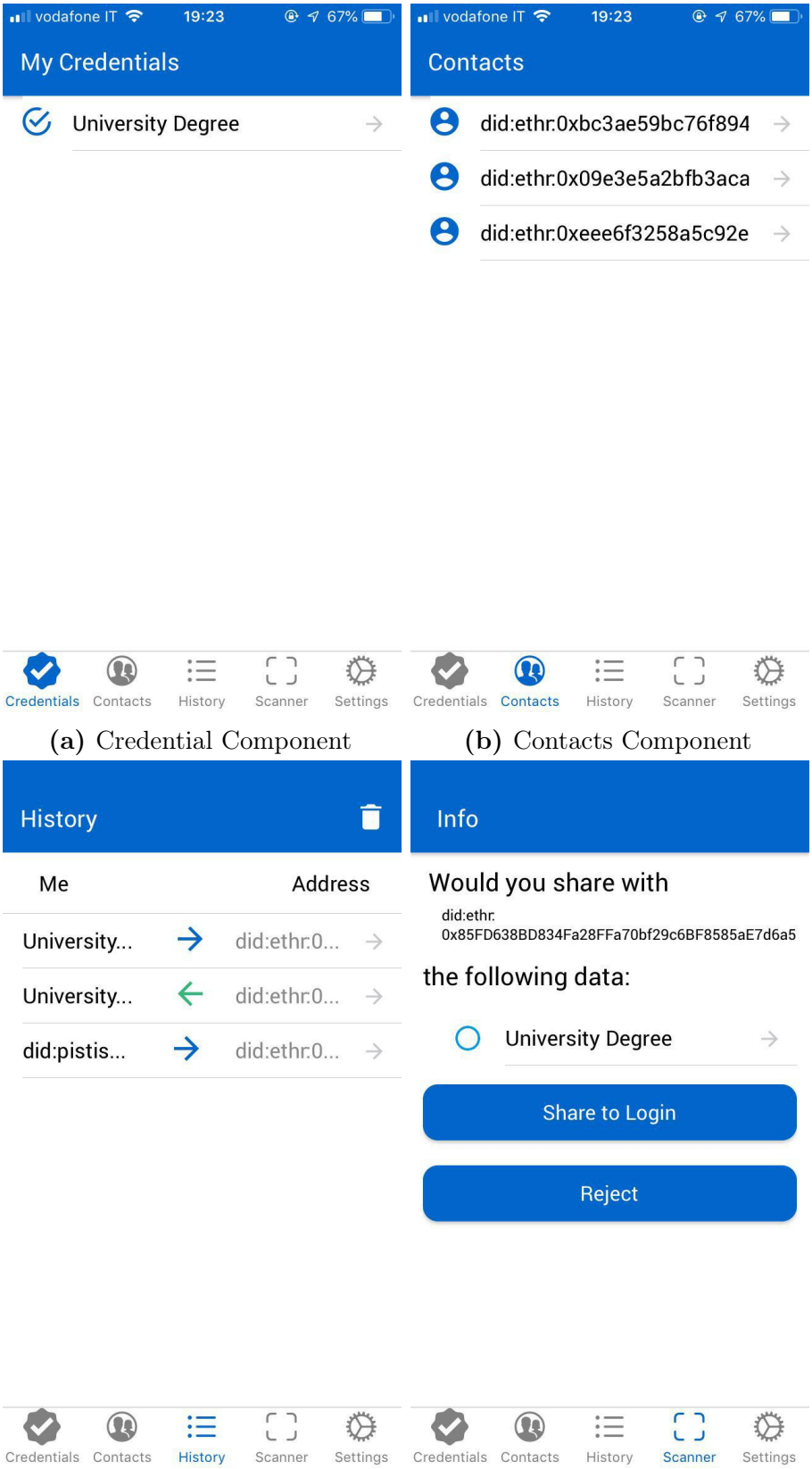
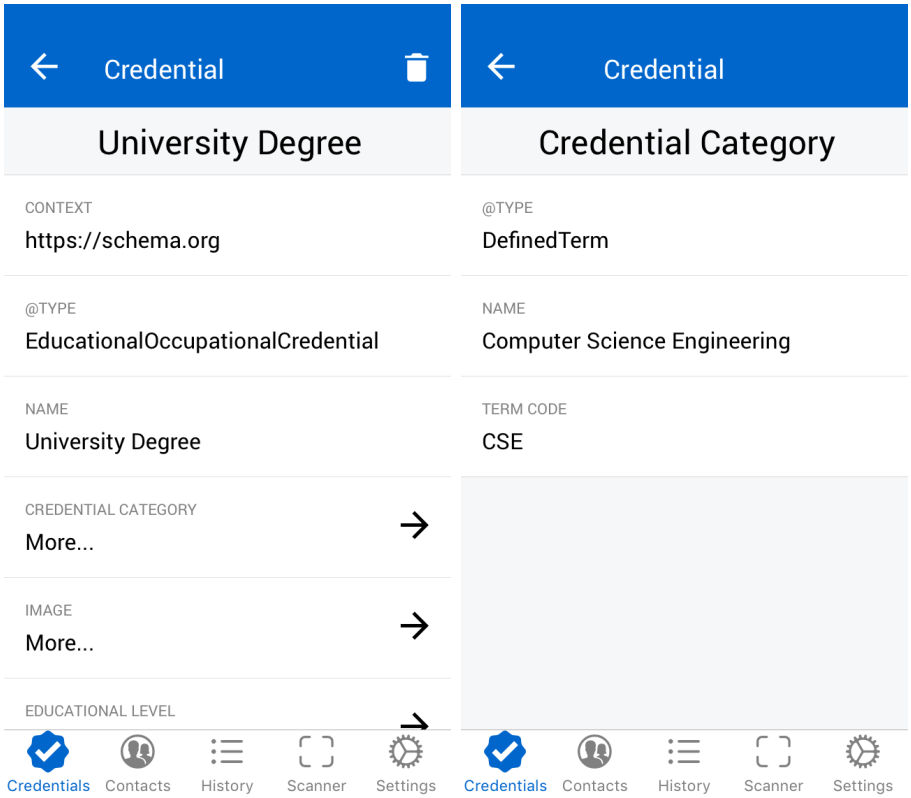


Figure A.2: Mobile Application Components



(a) Credential Subject sample **(b)** Sub field of a credential subject

Figure A.3: SSI Verifiable Credential example

A.2 Dashboard Screenshots

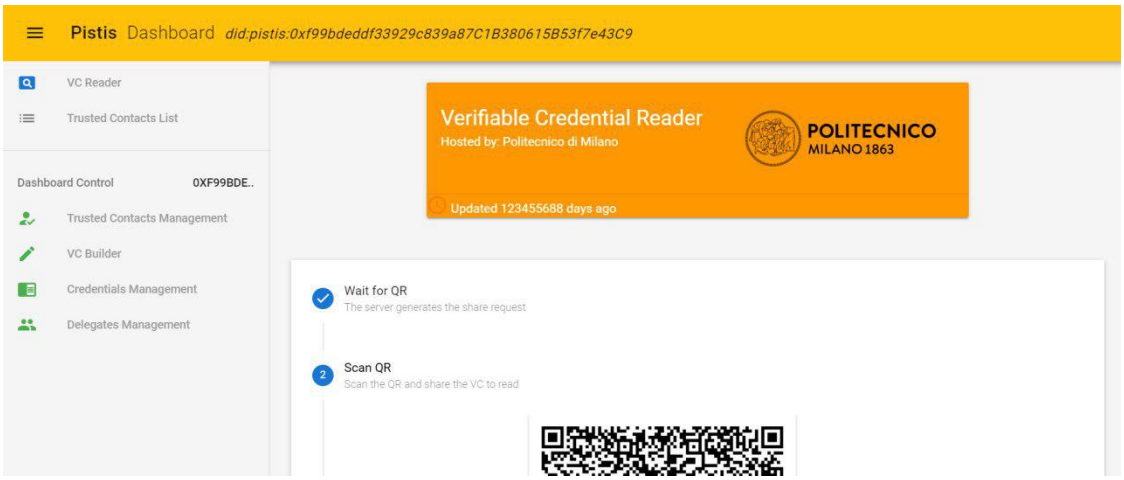


Figure A.4: VC Reader and dashboard overview

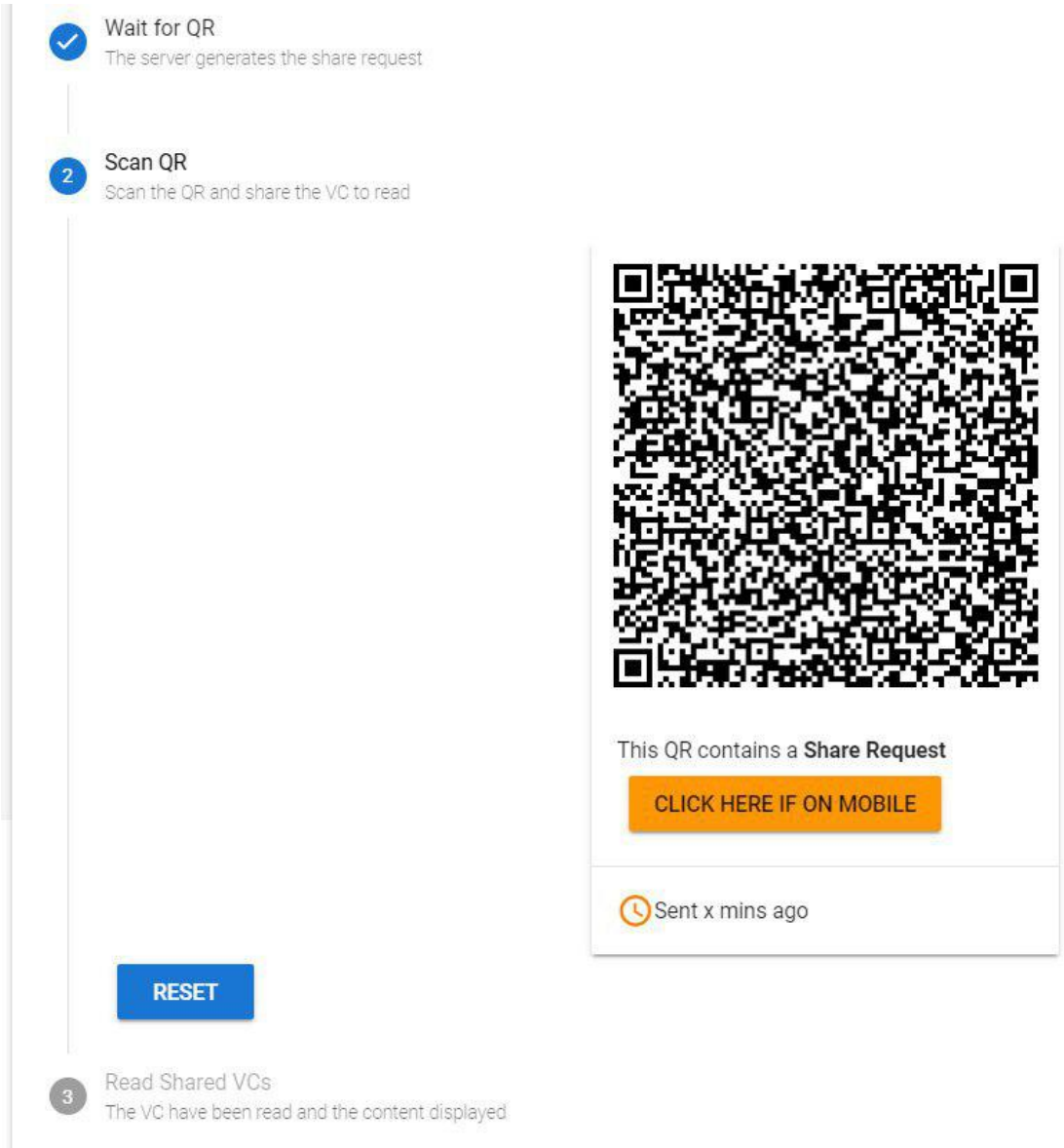


Figure A.5: VC Reader details

Figure A.6: Delegates Management

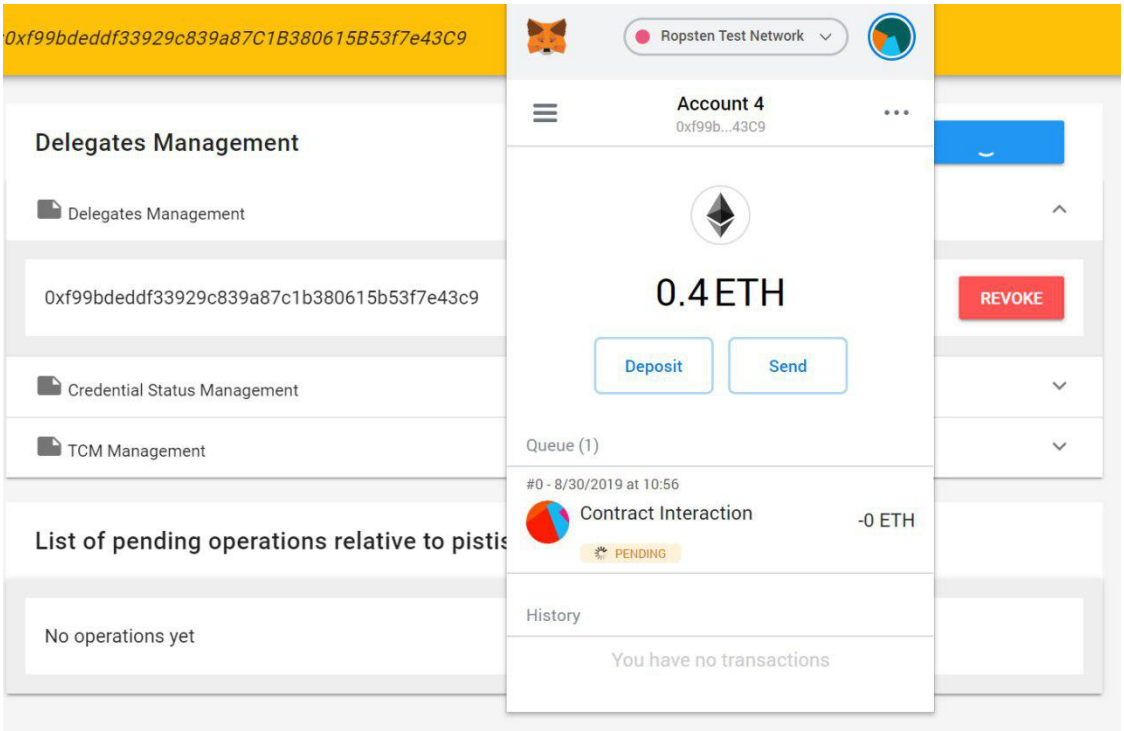


Figure A.7: Example of an on going transaction

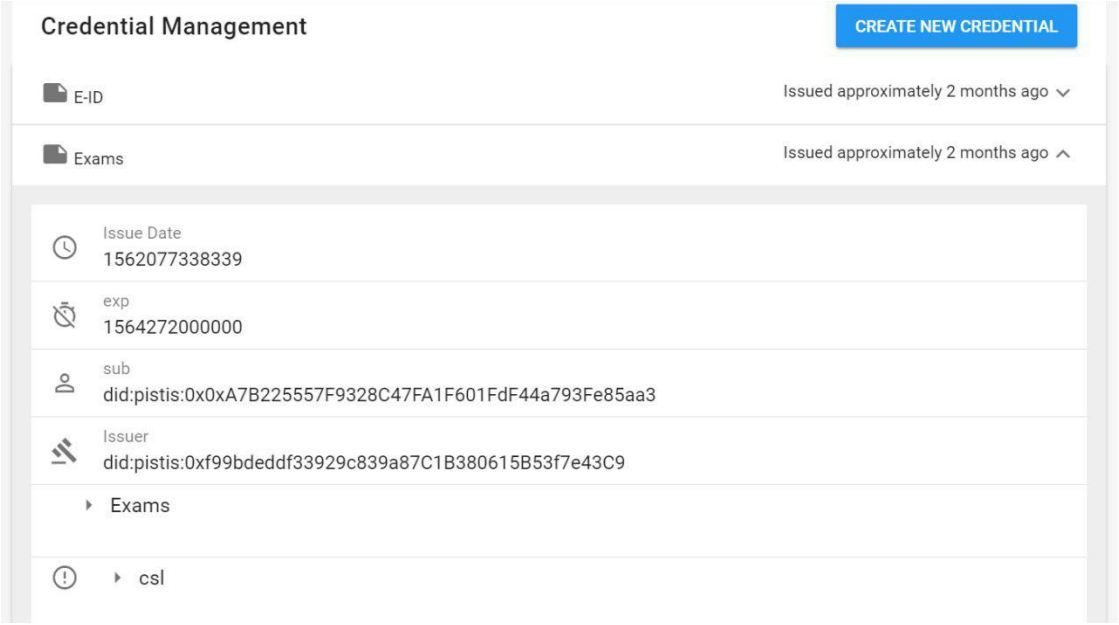


Figure A.8: Credential Management

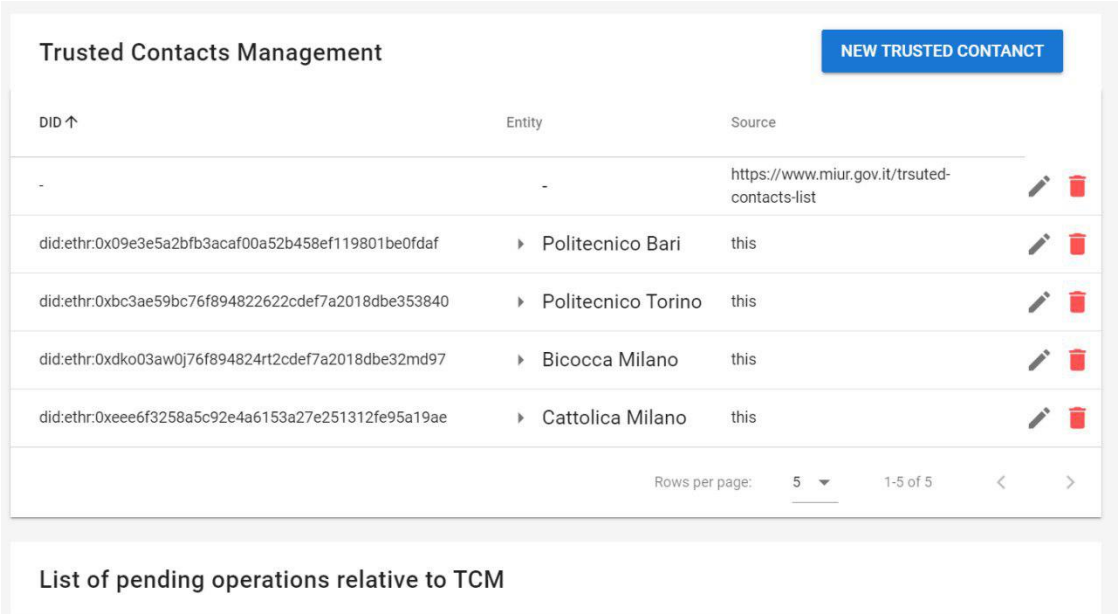


Figure A.9: Trusted Contact Management

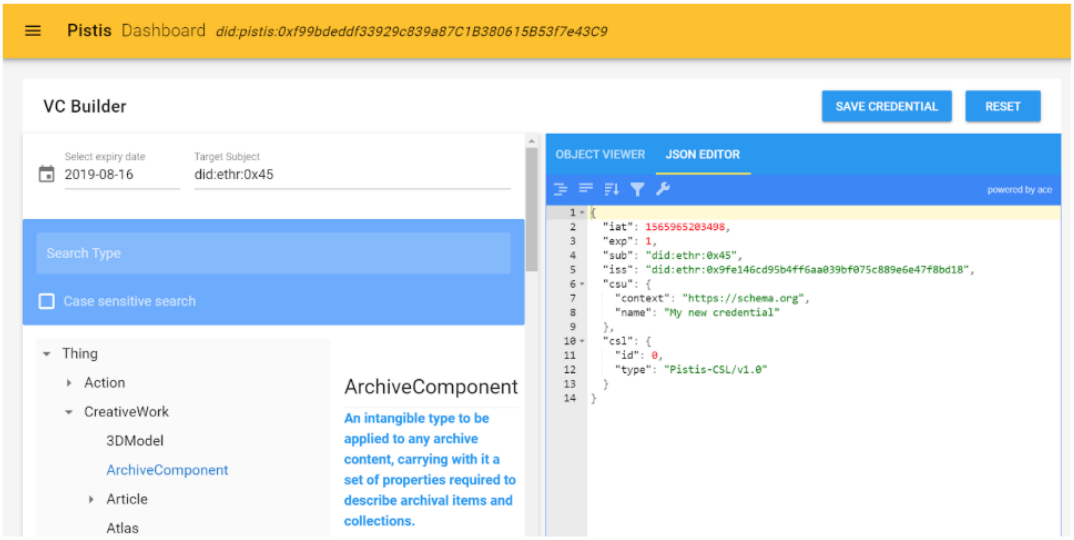


Figure A.10: VC Builder utility

Figure A.11: Object Viewer

A.3 DDO example

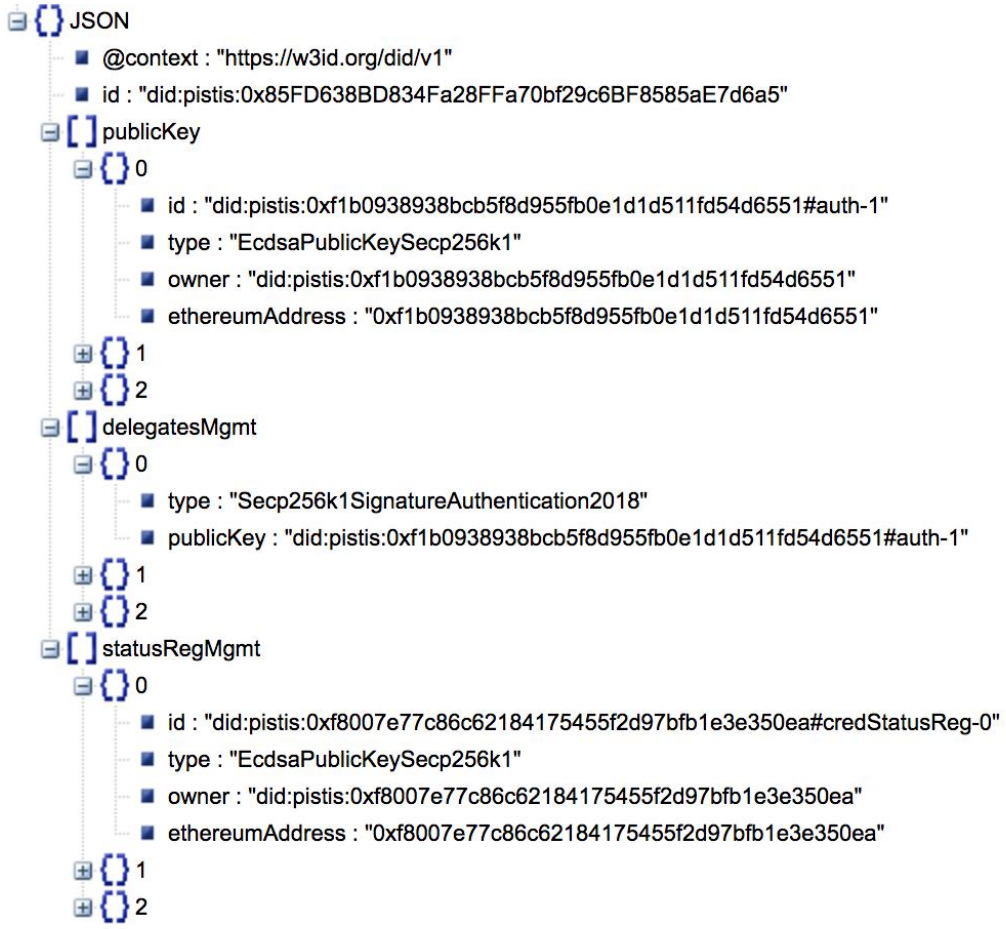


Figure A.12: DDO example

Appendix B

Communication protocol objects

```
"header":{
 "typ": "JWT",
 "alg": "ES256K-R"
}
"payload":{
 "type": "attestation",
 "credentialSubject": [
    eyJ0eXBaOwJSQ1....HdHrgh-EnAA ,
    eyJ0eX2dsisdc2....HoHthh-TmZZ ,
    eyJ0eXqiergGC3....HoHthh-TmZZ
 ],
 "aud": "did:ethr:0x7da253add95f4fe6gh269cf173c586s6g46d
    7va24",
 "iat": 1554889743,
 "exp": 1554890343,
 "iss":"did:ethr:0x9fe146cd95b4ff6aa039bf075c889e6e47f8
    bd18",
 "file": {encodingFormat: "SHA256", value: [...],
 "data": {econdingFormat: "SHa256", value: [...]}
}
signature:{...}
```

Codice B.1: Attestation sample

```
"header":{
 "typ": "JWT",
 "alg": "ES256K-R"
}
"payload":{
 "type": "shareReq",
 "requested": [
   "Thing/CreativeWork/EducationOccupationalCredential
      /*",
 ],
```

```
 "callback": "https://verifierWebSite/verifyDegree?
    socketId=...",
 "aud": did:ethr:0x7da253add95f4fe6gh269cf173c586s6g46d7
    va24
 "iat": 1554889743,
 "exp": 1554890343,
 "iss": "did:ethr:0x9fe146cd95b4ff6aa039bf075c889e6e47f8
    bd18"
}
signature:{...}
```

Codice B.2: ShareReq sample

```
"header":{
 "typ": "JWT",
 "alg": "ES256K-R"
}
"payload":{
 "type": "shareResp",
 "req": [eyJ0eXAiOiJKV1....HoHthh-TmZZ],
 "credentialSubject": [
    eyJ0eXBaOwJSQ1....HdHrgh-EnAA ,
    eyJ0eX2dsisdc2....HoHthh-TmZZ ,
    eyJ0eXqiergGC3....HoHthh-TmZZ
 ],
 "aud": "did:ethr:0x9fe146cd95b4ff6aa039bf075c889e6e47f8
    bd18",
 "iat": 1554889743,
 "exp": 1554890343,
 "iss":"did:ethr:0x7da253add95f4fe6gh269cf173c586s6g46d7
    va24",
 "file": {encodingFormat: "SHA256", value: [...],
 "data": {econdingFormat: "SHa256", value: [...]}
}
signature:{...}
```

Codice B.3: ShareResp sample

Appendix C

PoC for the Maltese Government

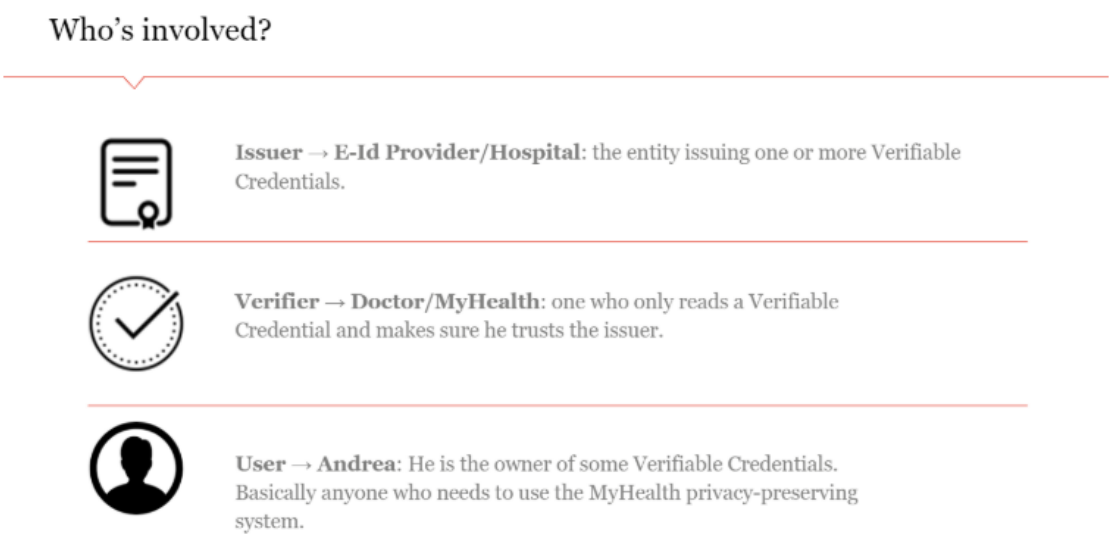


Figure C.1: Actors involved

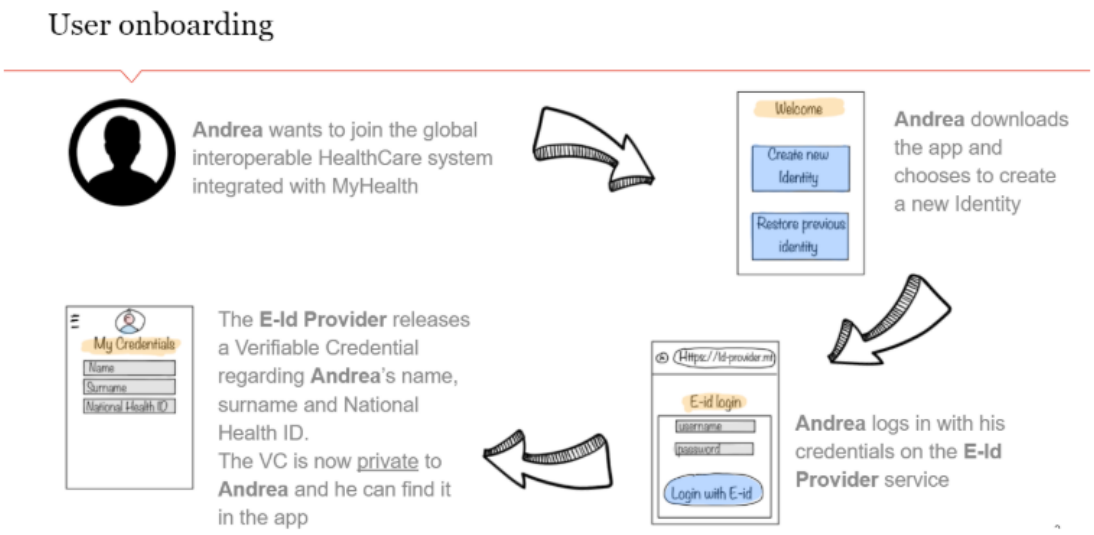


Figure C.2: User On boarding

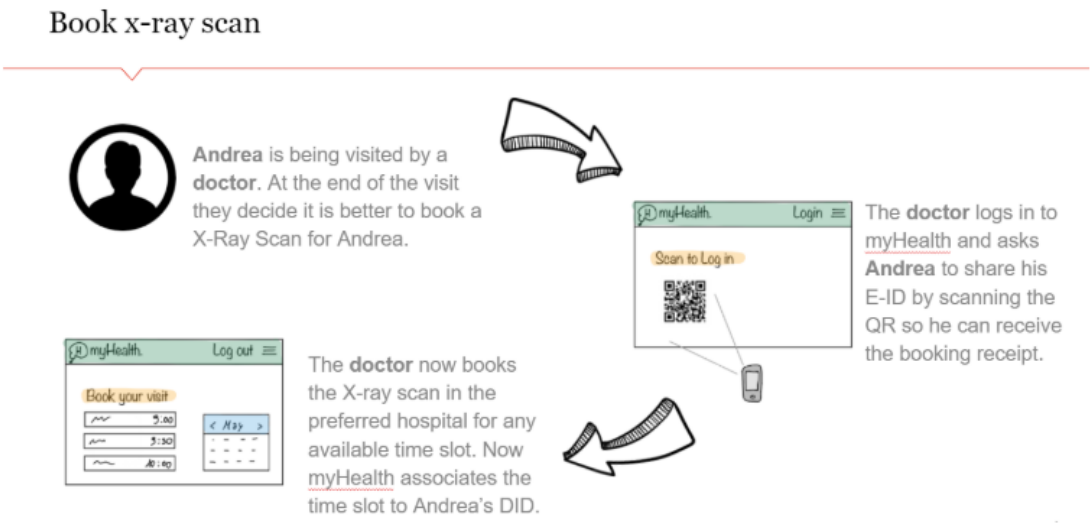


Figure C.3: Book an X-Ray Scan

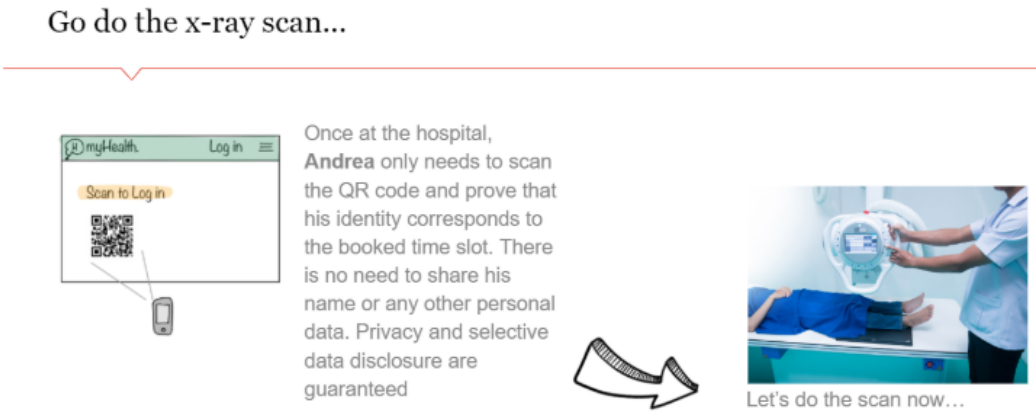


Figure C.4: Do the X-Ray Scan

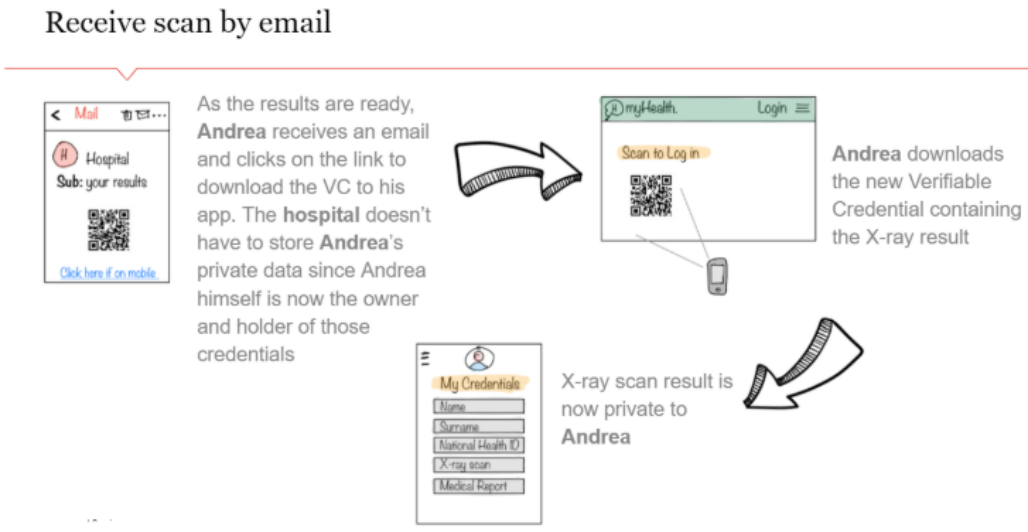


Figure C.5: Receive the scan via e-mail

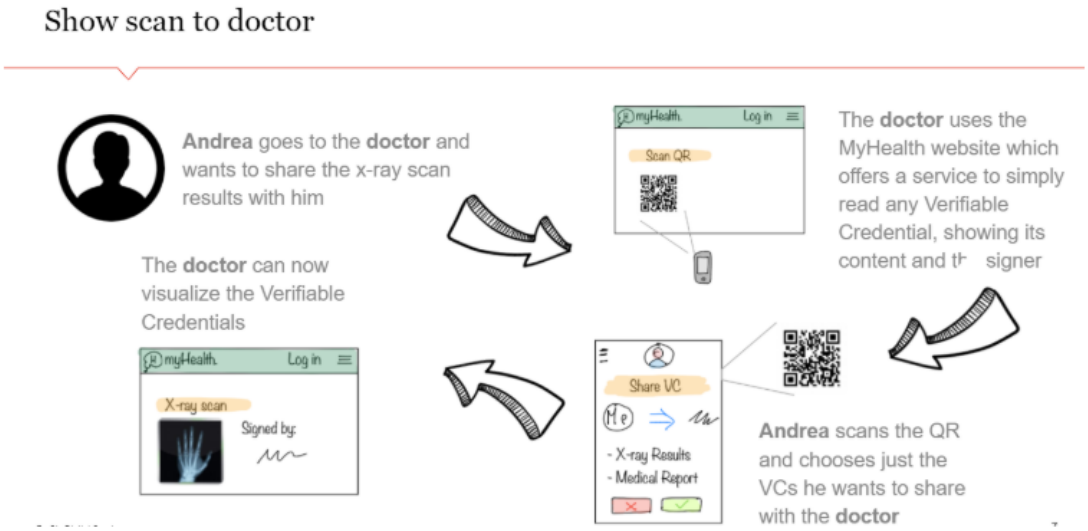


Figure C.6: Show the scan to the doctor

Appendix D

Smart Contracts Documentation

D.1 MultiSigOperations

MultiSigOperations - Generalized multi signature operations contract for a Self Sovereign Identity approach

constructor

sets the deployer of the contract equals to message sender

setPermissionRegistry

Development notice: *need to set PermissionRegistry contract address before the contract can start operating*

Parameters:

- registryAddress `address`: *(address) any contract address extending PermissionRegistry interface*

enableCircuitBreaker

in case of emergency it can be called by the deployer to block the contract confirming any operation

disableCircuitBreaker

used to go back functioning after an emergency stop

submitOperation

entry point to submit a generic operation

Development notice: *operation's parameters are packed into generic arrays. Execute methods of each executor contract expects its own params*

Parameters:

- identity `address`: *(address) identity subject of the operation*

- executor `address`: *(address) contract address extending OperationExecutor interface on which the operation will be executed*
- intParams `uint256[]`: *() generic integer parameters for the operation*
- stringParams `string`: *(string) generic string parameter for the operation*
- addressParams `address[]`: *() generic address parameters for the operation*
- bytesParams `bytes32[]`: *() generic bytes parameters for the operation*

Return Parameters:

- `uint256`

confirmOperation

entry point to confirm an operation. Only authorized addresses for that executor and for that identity are authorized

Parameters:

- opId `uint256`: *(uint256) operation identifier to be confirm*

confirm

Parameters:

- opId `uint256`: *(uint256) operation identifier to be confirmed*

revokeConfirmation

allows to revoke a confirmation already placed for a certain operation, as long as that operation has not been executed yet

Parameters:

- opId `uint256`: *(uint256) operation identifier for which to revoke confirmation*

executeOperation

Development notice: *generic operation executor. it checks the operation has enough confirmation, then calls the execute method on the intended OperationExecutor contract*

Parameters:

- opId `uint256`: *(uint256) operation identitifier to execute*

Return Parameters:

- `bool`

D.2 CredentialStatusRegistry

CredentialStatusRegistry

constructor

Parameters:

- multiSigContract `address`: *(address) address of the multiSigOperations contract to handle multi sig updates on this contract structure*

execute

Development notice: *change the status of a credential*
Parameters:

- identity `address`: *(address) identity relative to changes to make*
- intParams `uint256[]`: *(uint256[]) index 0 has to carry the id of the credential to be changed. index 1 has to carry the credential Status code. other indexes not used*
- stringParams `string`: *(string) not used*
- addressParams `address[]`: *(address[]) not used*
- bytesParams `bytes32[]`: *(bytes32[]) index 0 has to carry the status reason. other indexes not used*

Return Parameters:

- `bool`

D.3 PermissionRegistry

PermissionRegistry abstract contract

constructor

Parameters:

- multiSigContract `address`: *(address) address of the multiSigOperations contract to handle multi sig updates on this contract structure*

actorHasPermission

Parameters:

- identity `address`: *(address) identity to check permissions for*
- executor `address`: *(address) executor to check permissions for*
- actor `address`: *(address) acot to check permissions for*

Return Parameters:

- `bool`

quorumSatisfied

Parameters:

- identity `address`: *(address) identity to check quorum for*
- executor `address`: *(address) executor address to check quorum for*
- confirmationCount `uint8`: *(uint8) how many confirmations*

Return Parameters:

- `bool`

execute

Development notice: *this is the only function that will be called from the MultiSigOperations contract. If multiple operations have to be executed use one param to select one or another*

Parameters:

- identity `address`: *(address) identity subject of the operation to be executed*
- intParams `uint256[]`: *() generic integer parameters for the operation*
- stringParams `string`: *(string) generic string parameter for the operation*
- addressParams `address[]`: *() generic address parameters for the operation*
- bytesParams `bytes32[]`: *() generic bytes parameters for the operation*

Return Parameters:

- `bool`

1

Acronyms

DLT	Distributed Ledger Technology
VC	Verifiable Credential
VP	Verifiable Presentation
DID	Decentralized Identifier
DDO	Decentralized Document
TCM	Trusted Contacts Management
TCL	Trusted Contacts List
SSI	Self Sovereign Identity
DIF	Decentralized Identity Foundation
W3C	World Wide Web Consortium
EBSI	European Blockcahin Service Infrastructure
CEF	Connecting Europe Facility
EHEA	European Higher Education Area
MIUR	Ministero dell'istruzione, dell'università e della ricerca
CBA	Canadian Association Bank
CVC	Civic Token
RSK	RootStock

Bibliography

[1] Accredible. *Accredible main page*. URL: https://accredible.com (cit. on p. 85).

[2] Agid. *Linee guida per il contrassegno generato elettronicamente*. 2013. URL: https://www.agid.gov.it/sites/default/files/repository_files/linee_guida/circolare_n._62_recante_linee_guida_contrassegno_elettronico_art_23_ter_cad_0.pdf (cit. on p. 6).

[3] Christopher Allen. *The Path to Self-Sovereign Identity*. URL: http://www.lifewithalacrity.com/2016/04/the-path-to-self-soverereign-identity.html (cit. on p. 10).

[4] Canadian Banks Association. *White Paper: Canada's Digital ID Future - A Federated Approach*. 2018. URL: https://cba.ca/embracing-digital-id-in-canada (cit. on p. 1).

[5] Tim Berners-Lee. *Solid Inrupt*. URL: https://solid.inrupt.com/ (cit. on p. 77).

[6] Bitcoin. *Mnemonic code for generating deterministic keys*. URL: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki (cit. on p. 89).

[7] Blockcerts. *Blockcerts faq page*. URL: https://www.blockcerts.org/guide/faq.html (cit. on p. 85).

[8] Blockcerts. *Blockcerts main page*. URL: https://www.blockcerts.org/ (cit. on p. 84).

[9] Civic. *Civic main page*. URL: https://civic.com (cit. on p. 85).

[10] Paris Communiquè. *Final Draft*. URL: http://www.ehea.info/media.ehea.info/file/BFUG_Meeting/48/8/BFUG_BG_SR_61_4_FinalDraftCommunique_947488.pdf (cit. on p. 2).

[11] Consensys. *How to generate an ethereum address*. URL: https://github.com/ConsenSys-Academy/ethereum-address-generator-js (cit. on p. 28).

[12] Ashish Dhawan and Aditi R. Ganesan. *Handwritten Signature Verification*. URL: https://pdfs.semanticscholar.org/7244/79ac337d5c1262fb151281b900a4f4210426.pdf (cit. on p. 5).

[13] Shayan Eskandari et al. *A first look at the usability of bitcoin key management*. URL: https://arxiv.org/abs/1802.04351 (cit. on p. 77).

[14] Ethense. *github repository of ethense project*. URL: https://github.com/ethense (cit. on p. 83).

[15] Ethereum. *Security consideration*. URL: https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html (cit. on p. 78).

[16] Facebook. *React native framework*. URL: https://facebook.github.io/react-native/ (cit. on p. 53).

[17] Connecting Europe Facility. *Introducing the European Blockchain Service Infrastructure (EBSI)*. URL: https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/ebsi (cit. on p. 1).

[18] Decentralized Identity Foundation. *DIF Identity Hubs*. URL: https://github.com/decentralized-identity/identity-hub/blob/master/explainer.md (cit. on pp. 37, 77).

[19] Decentralized Identity Foundation. *ION github page*. URL: https://github.com/decentralized-identity/ion (cit. on p. 87).

[20] P2P Foundation. *Evolution of Online Identity*. URL: https://wiki.p2pfoundation.net/Evolution_of_Online_Identity (cit. on p. 10).

[21] P2P Foundation. *Self-Sovereign Identity*. URL: https://wiki.p2pfoundation.net/Self-Sovereign_Identity (cit. on p. 10).

[22] IMS GLOBAL. *Open Badges v2.0*. URL: https://www.imsglobal.org/sites/default/files/Badges/OBv2p0Final/index.html (cit. on pp. 19, 83).

[23] IETF. *The Base16, Base32, and Base64 Data Encodings*. URL: https://tools.ietf.org/html/rfc4648#page-5 (cit. on p. 74).

[24] Infura. *Infura*. URL: https://infura.io/ (cit. on p. 54).

[25] Terence Lee. *Singapore to roll out blockchain-based education certification system nationwide*. 2019. URL: https://www.techinasia.com/singapore-rolls-blockchainbased-education-certification-system-nationwide (cit. on p. 1).

[26] Ania Lipińska. *Ethense and uPort bring educational certificates to Self-Sovereign Identity*. URL: https://medium.com/uport/ethense-and-uport-bring-educational-certificates-to-self-sovereign-identity-7a6b6c2f41c0 (cit. on p. 83).

[27] Donnie MacColl. *What Is GDPR?* URL: https://www.srcsecuresolutions.eu/news-media/news/what-is-gdpr (cit. on p. 80).

[28] Metadium. *Metadium main page*. URL: https://metadium.com (cit. on p. 85).

[29] Metamask. *Metmask Browser Extension*. URL: https://metamask.io/ (cit. on p. 55).

[30] OpenCerts. *OpenCerts faq page*. URL: https://opencerts.io/faq (cit. on p. 84).

[31] OpenCerts. *OpenCerts main page*. URL: https://opencerts.io/ (cit. on p. 83).

[32] Json Schema Organization. *json schema*. URL: https://json-schema.org/ (cit. on p. 84).

[33] European Parliament and Council of the European Union. *GDPR, Article 4, Definitions*. URL: https://gdpr-info.eu/art-4-gdpr/ (cit. on p. 79).

[34] European Parliament and Council of the European Union. *GDPR, Chapter 3, Rights of the data subject*. URL: https://gdpr-info.eu/chapter-3/ (cit. on p. 80).

[35] European Parliament and Council of the European Union. *Recital 26 EU GDPR*. URL: http://www.privacy-regulation.eu/en/recital-26-GDPR.html (cit. on p. 80).

[36] The Linux Foundation Projects. *Hypeledeger Aries*. URL: https://infura.io/ (cit. on p. 77).

[37] Redux.org. *Redux design pattern*. URL: https://redux.js.org/ (cit. on p. 53).

[38] RootStock. *RSK*. URL: https://www.rsk.co/ (cit. on p. 86).

[39] Osvaldo A. Rosso, Raydonal Ospina, and Alejandro C. Frery. *Classification and Verification of Handwritten Signatures with Time Causal Information Theory Quantifiers*. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5131934/ (cit. on p. 5).

[40] Timothy Ruff. *Linee guida per il contrassegno generato elettronicamente*. 2018. URL: https://medium.com/evernym/the-three-models-of-digital-identity-relationships-ca0727cb5186 (cit. on p. 11).

[41] Schema.org. *Data Model*. URL: https://schema.org/docs/datamodel.html (cit. on p. 31).

[42] Dieter Shirley. *ERC 721 standard*. URL: http://erc721.org/ (cit. on p. 42).

[43] Alex Simons. *Identity Hubs as personal datastores*. URL: https://techcommunity.microsoft.com/t5/Azure-Active-Directory-Identity/Identity-Hubs-as-personal-datastores/ba-p/389577 (cit. on p. 87).

[44] Sovrin. *Sovrin main page*. URL: https://sovrin.org/ (cit. on p. 87).

[45] Sovrin. *Sovrin whitepaper*. URL: https://sovrin.org/wp-content/uploads/2018/03/Sovrin-Protocol-and-Token-White-Paper.pdf (cit. on p. 88).

[46] Manu Sporny et al. *Secure Data Hubs*. URL: https://github.com/WebOfTrustInfo/rwot9-prague/blob/master/topics-and-advance-readings/secure-data-hubs.md (cit. on pp. 77, 87).

[47] Nick Szabo. *Formalizing and Securing Relationships on Public Networks*. URL: https://nakamotoinstitute.org/formalizing-securing-relationships/ (cit. on p. 23).

[48] UNESCO. *Global Convention on the Recognition of Qualifications concerning Higher Education*. URL: https://www.chea.org/sites/default/files/2019-06/Global-Convention-on-the-Recognition-of-Qualifications-concerning-Higher-Education.pdf (cit. on p. 2).

[49] Council of theEuropean Union. *Council Recommendation on promoting automatic mutual recognition of higher education and upper secondary education and training qualifications and the outcomes of learning periods abroad*. URL: http://data.consilium.europa.eu/doc/document/ST-14081-2018-INIT/en/pdf (cit. on p. 2).

[50] Everis US. *Handwritten Signature Verification with Neural Network*. URL: https://medium.com/@everisUS/handwritten-signature-verification-with-neural-networks-94da5cbe0c5f (cit. on p. 5).

[51] W3C. *Decentralized Identifiers*. URL: https://w3c-ccg.github.io/did-spec/ (cit. on pp. 21, 85).

[52] W3C. *DID Method Registry*. URL: https://w3c-ccg.github.io/did-method-registry/ (cit. on p. 85).

[53] W3C. *Verifiable Credentials Data Model 1.0*. 2019. URL: https://www.w3.org/TR/vc-data-model/ (cit. on pp. 19, 24, 83).