FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Decentralized Identity in DasContract Decentralized Applications
Student:	Bc. Tomáš Bydžovský
Supervisor:	Ing. Marek Skotnica
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2021/2022

Instructions

Living in this digital age allows us to access services worldwide and interact with thousands of other people. However, all the apps and services collect and store personal data that cause security and privacy concerns. Decentralized identity gives us the ability to own and control our digital identity and securely protect our personal information. This thesis's primary goal is to explore how the DiD can be used in the state of the art web applications and demonstrate it on simple proof of concept.

Steps to take:
- Review Dapps, DasContract, a DiD (including zero-knowledge proofs)
- Compare did vs. traditional approaches (OpenID, OAuth).
- How to use DiD it in the context of DasContract
- Propose an architecture on how to integrate the DiD standard compatible with OpenID.
- Proof of concept case study.

Electronically approved by Ing. Michal Valenta, Ph.D. on 9 December 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

Decentralized Identity in DasContract Decentralized Applications

Bc. Tomáš Bydžovský

Department of Software Engineering
Supervisor: Ing. Marek Skotnica

May 4, 2021

Acknowledgements

I would like to thank my supervisor Ing. Marek Skotnica for his guidance and valuable advice.

Declaration

In Prague on May 4, 2021 …………………

Citation of this thesis

Bydžovský, Tomáš. *Decentralized Identity in DasContract Decentralized Applications.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Technologie blockchainu slibuje, že od základu změní způsob, jakým žijeme naše životy. Blockchain umožňuje bezpečné, decentralizované ukládání dat, která pak následně nelze smazat nebo změnit. Společně s chytrými kontrakty (smart contracts), což je počítačový kód uložený v blockchainu, umožňuje tvořit decentralizované aplikace (DApps). Tento typ aplikací nepotřebuje žádný centrální server, protože se k jejich hostování používají decentralizované sítě. Decentralizované aplikace, a chytré kontrakty obecně, mají potenciál plně digitalizovat naše životy.

Nicméně, hlavní překážkou na cestě k bezpečným a zcela digitálním interakcím je prokazování identity a autentizace. Základní otázka je, jak prokázat něčí identitu nebo platnost dokladů online bez použití externí centrální autority. Podle mnohých by decentralizovaná identita mohla být řešením.

Tato práce se zabývá decentralizovanou identitou a celým jejím ekosystémem. Vysvětluje základní koncepty a motivaci stojící za jejím vznikem. Její součástí je i porovnání se současnými autentizačními metodami. Cílem práce je prozkoumat možnosti použití decentralizované identity při návrhu decentralizovaných aplikací, zejména těch, které byly navrženy pomocí systému DasContract. Mimoto práce obsahuje návrh architektury autentizačního systému kompatibilního s OpenID. Vlastnosti systému využívající takovou architekturu jsou demonstrovány na implementaci prototypu v případové studii voleb do Evropského parlamentu.

Klíčová slova Decentralizovaná Identita, DID, DasContract, Autentizace, Digitální Identita, Decentralizovaný Identifikátor, Ověřitelné Digitální Doklady, Blockchain, Chytrý Kontrakt, Ethereum

Abstract

Blockchain technology promises to revolutionize our lives. It provides a secure, decentralized, immutable way of storing data. Blockchain, together with smart contracts, which is executable code embedded into a blockchain, makes it possible to create decentralized applications (DApps). These applications do not require any central server as they are hosted on decentralized networks. DApps and smart contracts generally have the potential to digitize our lives fully.

Nevertheless, one major obstacle to fully-realized digital interactions is identity and authentication. The essential question is how to prove someone's identity or credentials online without the need for a central authority. Decentralized identity claims to be a viable solution.

This thesis deals with the decentralized identity ecosystem, as it explains its motivation and concepts. The thesis provides a comparison with current authentication methods. It aims to explore a possible application of decentralized identity in decentralized application design, especially those designed using the DasContract. Moreover, it proposes authentication architecture compatible with OpenID. Properties of a system using such architecture are shown in the proof of concept case study of the Elections to the European Parliament.

Keywords Decentralized Identity, DID, DasContract, Authentication, Digital Identity, Decentralized Identifiers, Verifiable Credentials, Blockchain, Smart Contracts, Ethereum

Contents

Introduction **1**
- Motivation . . . 1
- Goal and the Objectives . . . 2

1 State of the Art **3**
- 1.1 Blockchain . . . 3
  - 1.1.1 How Blockchain works . . . 4
  - 1.1.2 Blockchain Types . . . 6
  - 1.1.3 Blockchain Characteristics . . . 8
  - 1.1.4 Use-cases . . . 9
- 1.2 Smart Contracts and Decentralized Applications (DApps) . . . 9
  - 1.2.1 Properties . . . 10
  - 1.2.2 Ethereum Implementation . . . 10
- 1.3 DasContract . . . 12
  - 1.3.1 Specification . . . 12
  - 1.3.2 DasContract Models . . . 14
  - 1.3.3 Alternatives to DasContract . . . 14
- 1.4 Zero-Knowledge Proofs (ZKP) . . . 15
  - 1.4.1 Definition . . . 15
  - 1.4.2 Application . . . 16
- 1.5 Decentralized Identity (DID) . . . 16
  - 1.5.1 Decentralized Identity Ecosystem . . . 17
  - 1.5.2 Decentralised identifiers . . . 18
  - 1.5.3 Verifiable Credentials . . . 21
  - 1.5.4 DID Authentication . . . 24
  - 1.5.5 Decentralised Key Management System (DKMS) . . . 29
  - 1.5.6 Other DID Technologies . . . 31
  - 1.5.7 Decentralized Identity Community . . . 33
- 1.6 Chapter Summary . . . 34

2 Current Authentication Methods and Comparison with DID 35
2.1 Username and Password 35
2.2 OAuth 2.0 36
2.2.1 OAuth terminology 36
2.2.2 OAuth Flow 37
2.2.3 Pseudo-authentication using OAuth 38
2.3 OpenID Connect 39
2.3.1 Terminology 40
2.3.2 JWT and claims 40
2.3.3 OpenID Connect flow 41
2.4 SAML protocol 41
2.4.1 Terminology 42
2.4.2 SAML Authentication Flow 42
2.5 Comparison of Current Authentication Methods and Decentralized Identity 43
2.5.1 Username and Password 43
2.5.2 OAuth 43
2.5.3 OpenID Connect 44
2.5.4 SAML 45
2.5.5 Decentralized Identity 45
2.5.6 Authentication Methods Comparison 45
2.6 Chapter Summary 47

3 Using DID in Decentralised Applications (and DasContract) 49
3.1 Smart Contracts Access Control 49
3.1.1 What is Access Control 49
3.1.2 Current Access Control Solutions 50
3.1.3 Using Decentralized Identity For Access Control 50
3.2 Proposed Access Control Methods 51
3.2.1 Direct Access Control 51
3.2.2 Indirect Access Control 52
3.3 Using Decentralized Identity in DasContract 54
3.4 Chapter Summary 55

4 DID Authentication Architecture Compatible with OpenID 57
4.1 System Architecture 57
4.2 Capabilities 57
4.2.1 Granting Access (Authorization) 58
4.2.2 Issuing Credentials 58
4.3 Components 59
4.3.1 Access Control Service 59
4.3.2 Decentralized Identity Module 60
4.3.3 Smart Contracts 61
4.3.4 Client App 61

4.4 Authorization Flow 62
4.5 Credentials Issuance Flow 63
4.6 OpenID Connect Compatibility 65
4.7 Chapter Summary 66

5 Proof of Concept Case Study 69
5.1 Case Study 69
5.1.1 EU Elections Case Study 69
5.1.2 Case Study Analysis 69
5.2 Technology Stack 70
5.2.1 Solidity 70
5.2.2 Ganache 71
5.2.3 MetaMask 71
5.2.4 Remix IDE 71
5.2.5 uPort 71
5.2.6 Node.js 72
5.2.7 Express.js 72
5.2.8 ngrok 72
5.2.9 Bootstrap 72
5.2.10 OpenZeppelin 72
5.2.11 ethers.js 72
5.2.12 INFURA 72
5.3 Implementation 73
5.3.1 Smart Contract 73
5.3.2 Access Control Service 76
5.3.3 Client Application 79
5.4 Deployment and Testing 79
5.5 Discussion and Future Development 81
5.6 Chapter Summary 84

Conclusion 85

Bibliography 87

A Acronyms 97

B Contents of enclosed CD 99

List of Figures

1.1 Network Types . . . 4
1.2 Simplified Blockchain Structure . . . 5
1.3 Proof of Work Flow [6] . . . 7
1.4 DasContract Architecture [11] . . . 12
1.5 DasContract Maturity Model for Measuring Contracts Quality [11] 13
1.6 Example of DasContract Process Model [13] . . . 14
1.7 Evolution of Online Identity . . . 17
1.8 Self-Sovereign Identity Model . . . 17
1.9 URN Structure [34] . . . 18
1.10 DID stricture [34] . . . 19
1.11 Concept of DIDs and DID Documents . . . 19
1.12 Overview of DID Architecture [33] . . . 22
1.13 Overview of VC Roles and Information Flow [40] . . . 23
1.14 VC and VP Content . . . 24
1.15 Overview of the DID Auth [43] . . . 27
1.16 Example of Basic Web Page Login Using [43] . . . 29
1.17 SIOP Overview [50] . . . 30
1.18 Detailed SIOP Flow [48] . . . 31
1.19 DID Stack [51] . . . 32

2.1 Typical Password Authentication . . . 36
2.2 OAuth Authorization Code Flow . . . 38
2.3 OAuth Implicit Flow . . . 39
2.4 OpenID Connect Structure . . . 40
2.5 OpenID Connect Authorization Code Flow . . . 41
2.6 SAML Service Provider Initiated Authentication Flow . . . 43
2.7 SAML Identity Provider Initiated Authentication Flow . . . 44

3.1 Direct Access Control Flowchart . . . 51
3.2 On-chain Verification Flow . . . 52

3.3 Off-chain Verification Flow 52
3.4 Indirect Access Control Flowchart 52
3.5 On-chain RBAC Flow 53
3.6 Off-chain RBAC Flow 53
3.7 DasContract DSL Metamodel [17] 54

4.1 Conceptual UML Activity Diagram of Granting Access 58
4.2 Conceptual UML Activity Diagram of Issuing Credentials 59
4.3 Authorization Token Generation 60
4.4 Structure of an Authorization Token Data 60
4.5 Conceptual UML Class Diagram of the Smart Contract Component 61
4.6 Authorization Token Verification 61
4.7 UML Component Diagram 62
4.8 UML Communication Diagram Representing System's Data Flow During Authorization Flow 62
4.9 UML Sequence Diagram Showing Detailed Authorization Flow . . 63
4.10 UML Sequence Diagram Showing Detailed Credentials Issuance Flow 64
4.11 General OIDC Authentication Flow 66
4.12 Using DID Authentication and VCs for OIDC Authentication . . . 67

5.1 DasContract Process Model of the EU Elections [13] 70
5.2 Updated Process Model of the EU Elections 70
5.3 UML Use Case Diagram of the EU Elections Case Study 71
5.4 Simplified UML Class Diagram of the Smart Contract Component 73
5.5 UML Package Diagram of the Smart Contract Component 74
5.6 UML Package Diagram of the Access Control Service 76
5.7 Access Control Service Credentials Disclosure API Scheme 76
5.8 UML Package Diagram of the Client Application Component . . . 79
5.9 UML Deployment Diagram of the System 80
5.10 uPort Identity Wallet Application 81
5.11 The Client App with Displayed QR Code with Credentials Disclosure Request 82
5.12 The Client App with MetaMask Extension in Process of Calling the Vote Function 82
5.13 uPort Identity Wallet Application 83

List of Tables

2.1 Authentication Methods Comparison Table 46

Introduction

Motivation

Identity is a very complex concept. Essential questions are "Who am I?", and "How do I prove it?". In the physical world, we use identification documents issued by the government, e.g., ID card, driver's license, and passport.

In the modern world own identity is the most valuable asset. Fundamental rights and public services like voting, healthcare, and education require legal proof of identification. However, nearly one in six people worldwide do not have it [1]. These people essentially do not exist for the outside world. Without identification, a person can not prove to own a property, can not get a bank loan, and there is no simple way out.

However, having identity without having control over it can also have dire consequences. During the Second World War, in the Netherlands, people were issued identity cards. The cards were very hard to forge because a copy of personal data was kept in centralized civil archives. That, combined with data from the 1930's census, made it very easy for the Nazis to locate and deport Jews [2, 3]. We trust our governments to protect our personal information, but policies can change, and governments can become corrupt.

Apart from abuse, identity theft also has significant financial losses. The recent Equifax breach cost more than 1 billion US dollars [4].

The motion of digital identity is even more complex, how to prove someone's identity online. Living in this modern age allows us to access services worldwide and interact with thousands of other people.

We use username & password or social login, like "Continue with Facebook", "Google Sign-In", and "Sign in with Apple" to log in to our favorite websites and services. These companies, we share our credentials with, control our digital identity. Because ultimately, they have the authority to verify that a person is who he claims to be just because they provide authentication services.

Not only that, but they also collect and store personal data, sometimes even without the users' knowledge. That is severe security and privacy concern. We potentially expose ourselves to breaches of our social, professional, and financial data. Many individuals and organizations were affected by such breaches.

There is a need for some type of universal identification but without centralized control. Decentralized identity claims to be just it. Thanks to the decentralized identity, everyone can have a verifiable digital identity without relying on any authority. It can give people the ability to own and control their digital identity and securely protect their personal information. Furthermore, only they decide what information they share and with whom.

Goal and the Objectives

The goals of this thesis are to review the Decentralized identity (DID) in the context of decentralized applications, to compare DID with current authentication and authorization methods, to explore the possibility of using the DID in the context of DasContract, to propose DID-based authentication architecture compatible with existing authentication infrastructure and to demonstrate DID usage in a simple case study.

The first objective involves reviewing decentralized applications (DApps), DasContract, decentralized identity, and zero-knowledge proofs. That also includes technology end mechanisms they are built on top, like blockchain and smart contracts.

The objective of the second part is to compare DID authentication mechanism with current methods. These methods include traditional username & password and "social logins" based on OAuth, OpenID, or SAML. To provide information for comparison, an overview of said methods is also part of this objective.

The third objective is to explore an application of decentralized identity in the context of DasContract and decentralized applications in general.

The objective of the next part is to propose an architecture of authorization and authentication system based on decentralized identity and compatible with current authentication infrastructure, like OpenID.

The final objective involves creating a proof of concept case study of the simplified Elections to the European Parliament to demonstrate the advantages of using DID authentication and authorization.

Chapter 1

State of the Art

This chapter provides a theoretical foundation for this thesis. At first, it overviews the technology of blockchain and related topics, like Ethereum, smart contracts, and DApps. Next, it explains DasContract and zero-knowledge proofs. Finally, it deals with the main topic of decentralized identity review, including the evolution of digital identity.

1.1 Blockchain

Blockchain is a new emerging technology that promises to change computing. In simplest terms, blockchain is a type of database. However, it has unique properties [5]. It is a decentralized distributed immutable data storage technology for storing transaction records. [6]

In the past, transactions were recorded in centralized databases provided by government or financial institutions, and people relayed on these institutions to ensure that their assets were safe. Essentially people put trust in these centralized systems to keep accurate records and enforce the validity of transactions. An example of this is the current currency system. Central banks print banknotes. People trust that value of that banknote is what it claims to be. Without this trust, it is essentially just a piece of paper.

“Centralized databases and institutions work when there is trust in the system of law, regulations, government, finance, and people.” ([5])

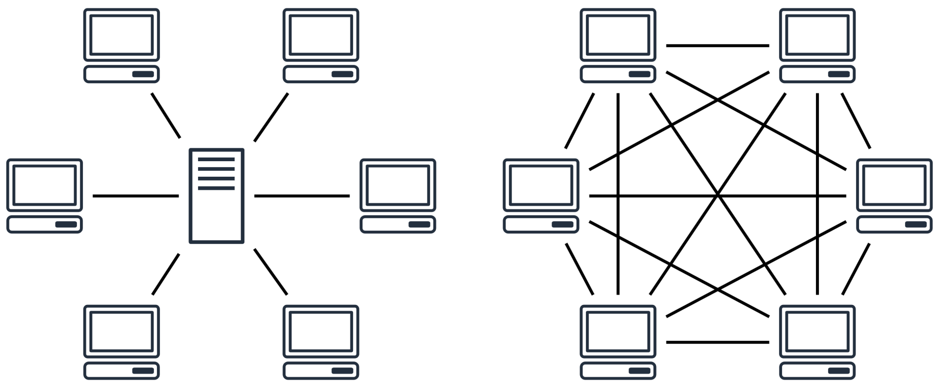
However, banks can go bankrupt, and governments can become corrupt or even oppressive. There are recorded instances of money tampering, fraud, government monitoring, etc.

The primary purpose of blockchain is to provide a system of trust between two parties without relying on a third-party institution (e.g., a bank) to facilitate the transaction.

“A decentralized database built on the blockchain removes the need for centralized institutions and databases. Everyone on the blockchain can view and validate transactions creating transparency and trust.” ([5])

Blockchain technology was proposed in 2008 by Satoshi Nakamoto[1] in his paper “Bitcoin: A Peer-to-Peer Electronic Cash System” [7]. In 2009 Satoshi Nakamoto created the Bitcoin network using blockchain. [5, 6]

Blockchain is the basis of most decentralized systems. The most common use case is digital currencies like Bitcoin, Ethereum and others. [5, 6]



(a) Centralized Network (b) Distributed (P2P) Network

Figure 1.1: Network Types

1.1.1 How Blockchain works

Blockchain heavily relies on cryptography algorithms. It uses private-public key cryptography and hashing functions. To use blockchain user needs key pair. The public key sometimes called an address, is used to identify the user. The private key is used for signing, and it is necessary for the user to access his account. Without a private key user cannot send transactions. Therefore he cannot access transactions send to him. Currency or data send to him are essentially lost. [5]

Blockchain is essentially a database of records containing transactions. The transaction consists of a sender, recipient, and transaction data (e.g., digital currency). Sender and recipient are identified with their public key. Each transaction is signed with the private key of the sender. [5]

[1]It is believed that Satoshi Nakamoto is a pseudonym and his real identity is not known.

Transactions are assembled into blocks. Each block contains a certain amount of transactions and a header. The header consists of a timestamp, nonce (arbitrary random number), hash of the block, and hash of the previous block. By referencing the previous block, the whole system looks like a chain of blocks, hence the name blockchain. Blocks are appended chronologically, which ensures the chronological order of transactions [6]. The first block is known as the genesis block. Transactions are validated before they are added to a block [6].

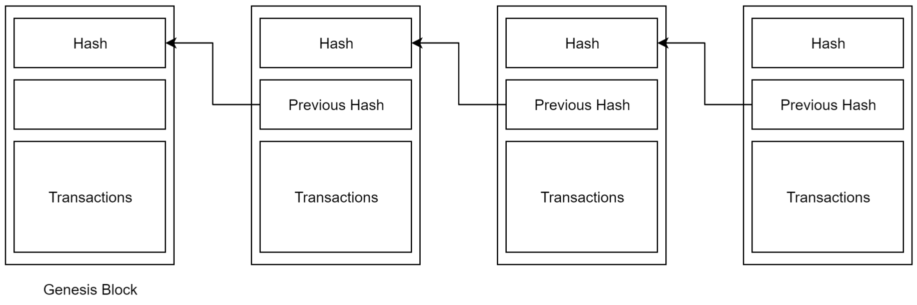


Figure 1.2: Simplified Blockchain Structure

Once added to a blockchain, a block can not be changed. Therefore transactions in the blockchain are immutable. If someone would attempt to change data in the blockchain, he would also have to change every block after the changed block because of the changed hash. It is reasonably impossible to do that, so it is believed that blocks deeper than six blocks from the top of the blockchain are safe.

Blockchain is based on a network of nodes. Node is a computer with running blockchain software. Each node contains its copy of the blockchain, and each node validates transactions [8]. Newly created transaction is in a pending state. It is sent to all nodes. Nodes take transactions, validate them ad form block. Adding a new block on top of the blockchain is a timely procedure (it can last several minutes), and all nodes race to be the first to add a new block.

Once some node adds a new block, it notifies others. Other nodes verify added transactions, and if a majority of nodes accept, this new block gets added to the blockchain in all nodes. This process is called distributed consensus [5]. This ensures that malicious nodes can not add invalid transactions to the blockchain and eliminates the need for centralized managing authority [6]. Transactions added to a blockchain change their state from *pending* to *confirmed* [6].

This system prevents the double spending problem (the double spending attack) when someone tries to spend the same token (digital currency, for example) in two separate transactions. Each transaction is added to a new block by two different nodes. However, only one of the nodes appends the new block. After that, all nodes can see that the second transaction is invalid because the token was already spent. Thus, that transaction is rejected.

The process of adding a new block is called mining, and nodes are therefore called miners. Miner creates a new block, called candidate block [6], but in order to be appended to the blockchain, the block needs to have special properties. It is called proof, and in most cases, it is done by Proof-of-Work (PoW). Proof-of-Work is simply a computational problem, which is difficult to solve but easy to verify. The difficulty of the problem is defined but can be adjusted to keep compute time consistent with increasing mining resources. Based on the difficulty target hash is chosen.

In order to solve the PoW, the value of a hash of the block has to be lower than the value of the target hash. Miners repeatedly try to guess a value of the Nonce to generate a valid hash. After the miner successfully appends a new block, it receives transaction fees and blocks reward. That is a reward to miners for validating transactions because without it blockchain network would collapse. [5, 6, 9]

Proof of work is necessary for blockchain to work. However, solving is just wasting computational resources. Therefore other methods of proof were proposed. The most prominent of them is Proof of Stake, where the probability of adding block is equal to the user's ownership stake in the system instead of the user's mining resources. Currently, PoW is still the most used method. [6]

There are many blockchains, most used for cryptocurrency. These include Bitcoin, Ethereum, Cardano, ZCash. Bitcoin was the first blockchain cryptocurrency. It was designed just as a shared public ledger based on blockchain. And as such, it is used almost solely as a cash system. Subsequent blockchains have brought something new. Smart contracts, better privacy, identity management, and others. [9, 5]

1.1.2 Blockchain Types

Two characteristics determine the type of blockchain. The first is whether the blockchain is public or private. The second is if the blockchain is permissionless or permissioned.

Public blockchain is a completely open network. Anybody can interact with blockchain or contribute to it. It does not have any central governing authority. An example of such a blockchain is Bitcoin.

Private blockchain uses the same technology as a public blockchain. However, the network is owned and controlled by some governing authority, who decides who can access the blockchain. The owner can be a single entity or a group. Private blockchains work similar to a centralized

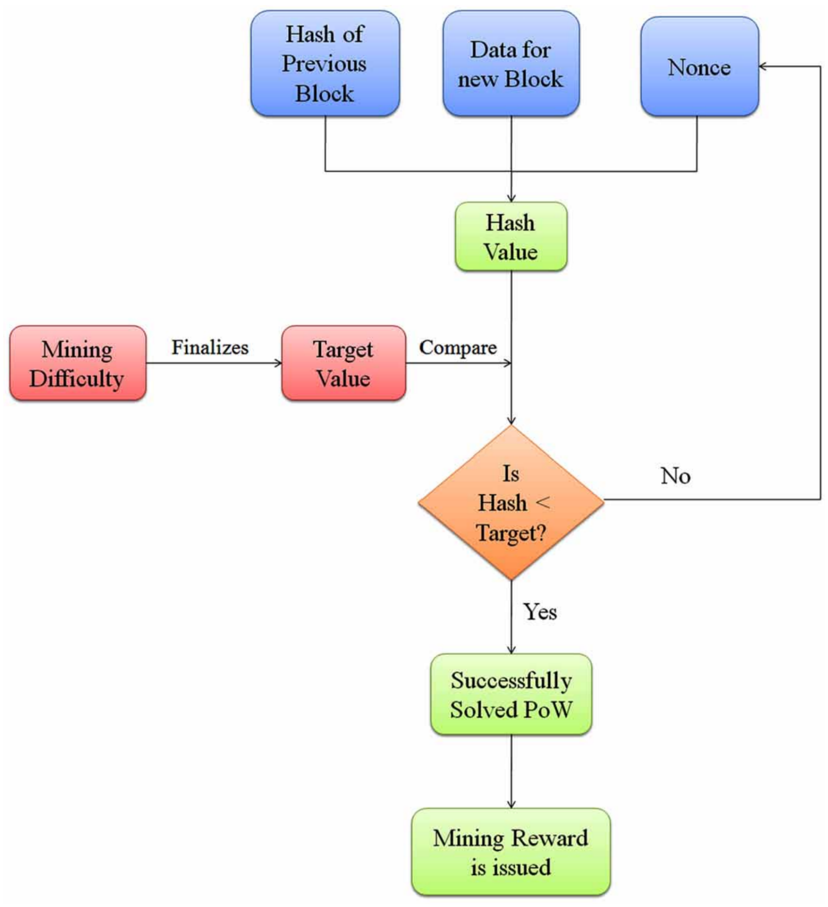


Figure 1.3: Proof of Work Flow [6]

system, just with some blockchain advantages. Private blockchains are usually used by organizations or groups of organizations in some industries, e.g., banks.

Permissionless blockchain is a blockchain where every user has the same rights. Nobody has special privileges. As with public blockchain, Bitcoin is a perfect example of a permissionless blockchain.

Permissioned blockchain differs by users having different rights. Usually, anyone can read data or mine, but adding new data is restricted. Generally, a permissioned blockchain is somewhere between public and private blockchain. It contains an access control layer that allows to make changes only by a user with verified identity and eventually also with the correct permission. Hyperledger Fabric is an example of a permissioned blockchain. [10, 6]

1.1.3 Blockchain Characteristics

1.1.3.1 Transparency and Trust

Blockchain offers much better transparency than current record-keeping systems. All transactions are visible, and altering existing records is almost impossible. There is a meager chance of someone altering records without being found.

There is no single entity that can control the blockchain. Any change has to be accepted by the majority of other stakeholders.

Blockchain enables secure and reliable transactions between two parties that do not need to trust each other without the need for an intermediary.

This transparency also has downsides. One of them is a lack of privacy. Because all transactions are traceable, anyone can see the content of any account. Also, after receiving payment from some person, it is easy to connect a real identity with the account.

1.1.3.2 Decentralization

Instead of storing records on multiple centralized ledgers, blockchain uses a shared ledger. A single entity can not control it, and it can not be blocked or disabled by attack due to its distributed nature. Also, sharing records in single shared ledgers can lower costs and speed up transactions between different parties (e.g., banks).

1.1.3.3 Security

Data stored in blockchain are immutable and can not be altered. All transactions are traceable and verifiable. Every transaction has to be validated by a majority in the network. Because of this, any fraud is much easier to trace than in current systems. Of course, there is always a possibility of fraud, but blockchain tries to minimize this possibility and makes it much harder to cover it up.

Because there is no central control to manage accounts, people can not reset their passwords like with current banking systems. The private key of the account is crucial to access it. If lost, the user can not access his money. If stolen, attackers transfer all money to his account, and there is no authority to revert that. Money sends to non-existing addresses is also lost.

Another security concern is 51% attack. The validity of transactions is determined by the consensus of the majority nodes in the network. However, if someone would gain control over more than 50% of nodes, he could maliciously allow double spending or other invalid transactions.

1.1.3.4 Removal of Intermediaries

In comparison with current systems, blockchain does not require any third party to provide trust and security. That can lead to the removal of the intermediaries from current processes. [8, 5, 6]

1.1.4 Use-cases

From the beginning, blockchain has been used as the basis for crypto-currencies or other financial use cases. However, recently, especially with blockchain 2.0, many non-financial use cases became possible. Most of these focus on digital asset handling, identity management, decentralized internet of things, and decentralized applications. That can have a massive impact on contract and document management and digitization, government digitization, electronic voting, digital assets protection and proof of ownership, as well as decentralized cloud storage.

1.2 Smart Contracts and Decentralized Applications (DApps)

In everyday life, contracts are a mechanism to uphold promises between individuals. By definition, a contract is an agreement between parties creating certain obligations enforceable by law. Smart contracts are digital contracts that leverage the power of blockchain to incorporate the power of contracts in a digital world. [6, 8]

"A smart contract is a set of commitments that are defined in digital form, including the agreement on how contract participants shall fulfill these commitments." ([11])

Smart contracts are contracts written in computer code operating on a blockchain or distributed ledger. They automatically verify and execute the contract and enforce rules and consequences based on conditions in the code. [5, 6]

A smart contract is basically a program that is stored in the blockchain and runs automatically on blockchain nodes. It is similar to Stored procedures in traditional SQL databases. Ethereum blockchain was designed primarily with smart contracts in mind. This evolution from simple distributed ledger to distributed execution platform is sometimes called Blockchain 2.0. [5]

Smart contracts use the processing power of computers connected to the network instead of just wasting it on mining. They are used to exchange payment or value based on conditions in a smart contract. Code is uploaded to the blockchain as part of a transaction to a special address. Once uploaded, the code runs automatically, and it can not be changed or removed. [5]

Smart contracts can take any form of data as input, perform computation and enforce decisions based on a result of computation and condition defined in the contract. That makes it possible to handle any valid and invalid transaction. It has enormous potential because it makes the whole network behave like a large computing mechanism. [6]

Decentralized Applications (dApps or DApps) are open source applications not controlled by a single entity and running on a blockchain network. Instead of a central server, they use blockchain and smart contracts as backend, and they use CDN or other distributed system to provide frontend components. DApps are build using smart contracts. [5]

DApps are similar to current three-tier web apps. They utilize blockchain as a data layer instead of a usual database. Smart contracts serve as an application layer. Clients interact with the application through accounts. [12]

1.2.1 Properties

Because smart contracts are automatic and self-executing, they can reduce delays and expenses normally caused by physical contracts. Contracts can not be changed, tampered with and they are precisely defined. This inflexibility and zero ambiguity mean that process can not be cheated. A smart contract will always behave the same way. [6]

Smart contracts also have a few drawbacks: interaction with external APIs. Because computation is done in a distributed manner, all nodes have to receive the same consistent data. Blockchain oracles are used for accessing external data. However, it brings another problem. Smart contracts are dependant on oracles, and therefore who controls oracle can manipulate the result of smart contracts. [13, 14]

Another drawback is that even though it is simple to prove ownership of digital token obtained via smart contract. It can be problematic to enforce ownership of physical property (represented by the token) in the real world. [6]

1.2.2 Ethereum Implementation

Because smart contracts were the primary design goal of Ethereum, its implementation of smart contracts became most common. Smart contracts are compiled to bytecode, which is in turn run by Ethereum Virtual Machine (EVM) on the network's nodes. EVM is essentially runtime for the Ethereum network [8]. Many Turing-complete high-level languages are used to program smart contracts, for example, Solidity, Serpent, Vyper, Mutan, and LLL. From these most common and most popular is Solidity. Its syntax is inspired by Java. [6, 8]

To prevent attacks, malicious code, or infinite loops creating and running smart contracts requires spending virtual currency, called gas or ether (hence the name Ethereum). Cost is determined by the compiler from the prices of

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.6;

contract Counter {
    uint public count;

    // Function to get the current count
    function get() public view returns (uint) {
        return count;
    }

    // Function to increment count by 1
    function inc() public {
        count += 1;
    }

    // Function to decrement count by 1
    function dec() public {
        count -= 1;
    }
}
```

Listing 1: Simple Solidity Smart Contract [15]

individual instructions. Gas price, therefore, determines the price of smart contract execution. [6, 8]

There are two types of accounts on Ethereum blockchain. [6, 8]

Externally Owned Account (EOA) is controlled by private key of individual user.

Contract account is equivalent to the Client application in OAuth.is controlled by smart contract and can by activated by EOA. This account does not have private key.

Smart contracts get executed whenever their account receives a transaction. Contracts can not store data, but they can read and write to the blockchain. Contracts can communicate with each other by sending messages. These transactions do not spend gas. Deployment of a contract is done by sending the complied contract and the start price of gas in a transaction to a particular account.

JSON-RPC is used to interact with a smart contract on the blockchain. There are API libraries implemented in many languages for convenience. The most common used by DApps is the Web3.js javascript API library. [6, 8]

1.3 DasContract

DasContract is a domain-specific language (DSL) designed for specifying smart contracts. It has two main goals. The first is to be implementation independent, so it can create contracts for any blockchain that supports it. Another goal is to provide a higher level of abstraction than usual programming code. Programming code is harder to comprehend and prone to bugs, but DasContract models aim to be more understandable by using enterprise engineering concepts. DasContract model smart contract as a business process. [13, 16]

1.3.1 Specification

DasContract's specification is based on a subset of BPMN, UML class diagram, and concepts specific for blockchain technology. The first version of DasContract was also based on DEMO methodology, but it proved to have undesirable consequences due to the blockchain constraints. DasContract is also able to specify off-chain forms that are used for smart contract interaction. The language is still under development. All details concerning development can be found in the official GitHub repository at `https://github.com/CCMiResearch/DasContract`. Currently, its implementation is able to translate DasContract's models to solidity language. [11, 13, 17]

Part of the DasContract ecosystem is also a smart contract designer, also under development. [18]

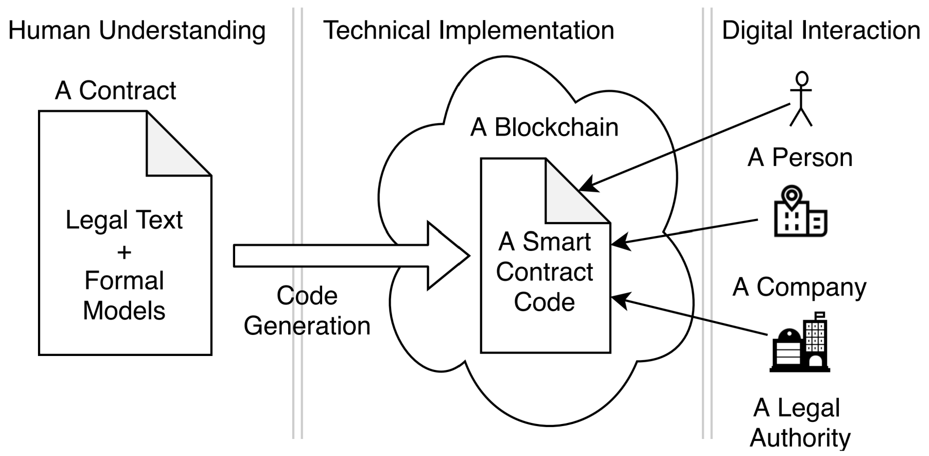


Figure 1.4: DasContract Architecture [11]

DasContract flow is defined in three parts [11, 13]:

Human Understanding part defies smart contract as the contract between multiple parties. It consists of formal ontological models and legal text.

The purpose of a legal text is to specify the legal validity of the formal model.

Technical Implementation part defies the process of transformation of the model from contract into executable code and subsequent deployment into blockchain in the form of the blockchain-specific smart contract.

Digital interaction part describes the interaction of people, companies, and legal authorities with contracts. All interactions are entirely digital and because blockchain properties, also legally binding.

Human understanding is the fundamental property of DasContract. Traditional text contracts and legal documents can be hard to understand truly or can be very ambiguous. Contracts specified only by computer code, on the other hand, can also be hard to comprehend. They can be prone to errors or bugs that are irreparable due to blockchain immutability. There is a third party: experts needed in both these cases for specifying or interpreting a contract. In the former case, a lawyer, a programmer in the latter. Ontological models of DasContract strife to be easier to understand in order not to require experts. [13, 19]

Maturity	**Name**	**Contract Form**	**Accuracy**
1	Verbal contract	A mutual understanding	No written record of a contract
2	Written informal contract	Informal text	Typically ambiguous interpretation, possible errors, no legal framework
3	Legally binding contract	Legal text	Risks of ambiguous interpretation, possible errors, legal framework contains ambiguities itself
4	Ontological contract	Ontological model	Ambiguity effectively controlled

Figure 1.5: DasContract Maturity Model for Measuring Contracts Quality [11]

Using models also has one significant benefit. A contract specified by models can be simulated in order to find ambiguities and errors before the contract is deployed to the blockchain. As already mentioned, implementation-independent. Therefore a contract can be translated to any supported blockchain smart contracts code. [13]

1.3.2 DasContract Models

DasContract model combines these interconnected models for specifying contracts. [13, 16, 17]

Process Model is used to specify the contract's process activities, control flow, property mappings, and user roles. That provides a way of modeling contract states and logic. This model is based on the extended subset of BPMN. The main addition is support for blockchain tokens. Contracts code generated from this model is mainly responsible for the control flow of the program.

Data Model specifies the domain data model of the process. It defines entities, their properties, and relations between them. It is based on a UML class diagram. It also provides blockchain-specific entities, like addresses, etcetera. Code generated from this model contains program data structures like domain classes, enums, and properties.

Forms Model is there to specify interfaces used for user activity input. Code generated from this model has two parts - on-chain and off-chain. Off-chain is similar to the client-side code of current applications. It is interpreted by blockchain wallet to create an UI that allows users to interact with the contract. On-chain code is like server-side code. It handles validations and authorization.

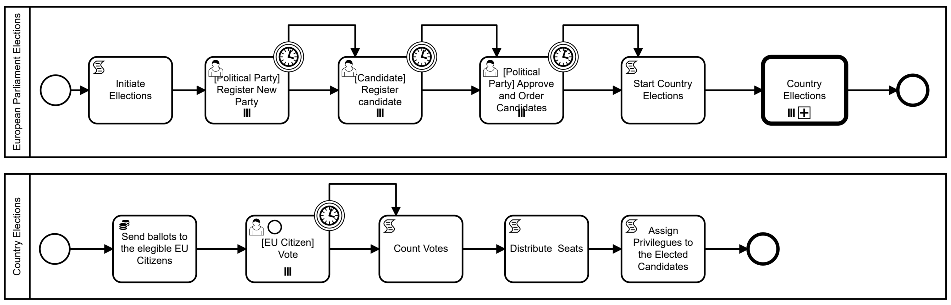

European Parliament Elections
Initiate Elections
[Political Party] Register New Party
[Candidate] Register candidate
[Political Party] Approve and Order Candidates
Start Country Elections
Country Elections
Country Elections
Send ballots to the elegible EU Citizens
[EU Citizen] Vote
Count Votes
Distribute Seats
Assign Privilegues to the Elected Candidates

Figure 1.6: Example of DasContract Process Model [13]

1.3.3 Alternatives to DasContract

There is a project called Caterpillar that is similar to DasContract. Caterpillar is an open-source blockchain-based BPMN execution engine. It supports creating instances of BPMN process models and execution performed by smart contracts thanks to BPMN-to-Solidity compiler. [20, 21]

As with DasContract, it takes the BPMN model as input and outputs smart contract in Solidity. Caterpillar uses REST API for interacting with smart contracts. It is implemented in Typescript, whereas DasContract is implemented in C♯. Caterpillar design is simpler because it supports only compilation of BPMN to Solidity. However, it is a more mature project than DasContract. It provides modeling tools as well as better documentation. [21, 16]

1.4 Zero-Knowledge Proofs (ZKP)

Zero-knowledge proof (ZKP) is a cryptographic method or cryptographic protocol. It has become widely spread in recent years. It is used when two parties exchange information. It allows one party (the prover) to prove having/knowing some information without revealing it to the other party (the verifier). For example, a prover can prove that he has a correct password without revealing the password to the verifier. [22, 23]

ZKP provides improved privacy and security in exchange for additional processing resources.

1.4.1 Definition

Two parties are involved in the ZKP protocol are a prover P and a verifier V.

At the end of the protocol flow, the verifier V is convinced that prover P knows the solution s to the instance I of a problem $\mathcal{P}$, without prover P revealing any information about s. [24]

Zero-knowledge proof has to satisfy these following properties:

Completeness If P knows s, then he is able to convince V. (If prover knows the information, he can convince the verifier.)

Extractability If P does not know s, then he is not able to convince V. (If prover does not know information, he is not able to convince verifier.) Except with some negligible probability. This probability is a function of required security as a parameter.

Zero-Knowledge V learns nothing about s except I. (Verifier learns nothing about information.) [25, 23]

There are two kinds of ZKPs: interactive and non-interactive. In interactive ZKP, the prover exchange messages with the verifier in order to convince him. In non-interactive, the prover creates proof that can be verified by the verifier. That allows broadcasting proof to multiple verifiers at once, who can verify it independently. Because of this property, it can be used in distributed systems. [25, 22]

1.4.2 Application

There are three major areas of application for ZKPs.

- Authentication
- Confidentiality
- Anonymity

ZKPs allow users to prove their identity without requiring their credentials or personal information. Another possible use case is when a user has to share information with some institution, he can limit the amount of that information or rather just provide proof instead. For example, when people provide their financial information in order to request a loan or mortgage. [26, 22]

Using ZKPs with blockchain also has significant potential. It provides confidentiality for transactions and also anonymity for accounts. It is possible to exchange digital currency without publicly declaring how much money the accounts have. However, for this purpose, ZKP has to be non-interactive and succinct, which means that verification has to be done fast, and the size of the proofs has to be small.

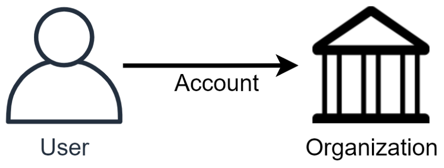
The most notable example of this is the Zcash blockchain. It is built on the same blockchain technology as Bitcoin, but it uses ZKPs for users' privacy. Its implementation uses zk-SNARK, which stands for "Zero-Knowledge Succinct Non-Interactive Argument of Knowledge". [22, 27]

1.5 Decentralized Identity (DID)

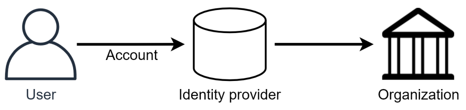
The motion of digital identity is evolving. Initially, online identity was based on ownership of an account. That is called centralized identity. People are identified by their accounts corresponding to a particular service or organization. These organizations were, in fact, in control of people's identities. This phase also includes government-issued ID documents, where identity is tied to the holder of the document and which can be stolen or revoked.

The second phase is called federated identity. It uses third-party identity providers to provide identity to services and organizations. Examples include OAuth, OpenID Connect, and SAML. People can now control access to their identities. However, an identity provider is still a sole centralized authority on identity.

The current evolution phase is called Self-sovereign identity (SSI). It requires that everyone has absolute control over their own identity without external central authority. Users control their profiles and what information services can access. It leverages the power of blockchain to provide proof of identity. Furthermore, Decentralized Identity is the technology that enables us to achieve SSI. [28, 29, 30]



(a) Centralized Identity Model



(b) Federated Identity Model

Figure 1.7: Evolution of Online Identity

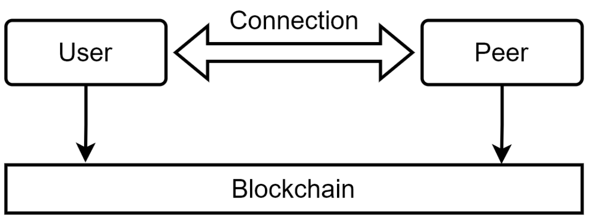


Figure 1.8: Self-Sovereign Identity Model

> "Self-sovereign identity is the next step beyond user-centric identity, and that means it begins at the same place – The user must be central to the administration of identity. That requires not just the interoperability of a user's identity across multiple locations, with the user's consent, but also true user control of that digital identity, creating user autonomy.
>
> To accomplish this, a self-sovereign identity must be transportable; it can't be locked down to one site or locale." ([28])

Essentially SSI is a lifetime portable identity for any person or organization that does not depend on any centralized authority, and it can never be taken away. In the case of a person, SSI has to emit directly from an individual human life and not some administrative mechanism. Every individual is the sole source of authority over their identity. [28, 30]

1.5.1 Decentralized Identity Ecosystem

Decentralized identity consists of a group of (proposed) standards and protocols. The most important of them is decentralized identifiers. The whole concept of decentralized identity revolves around it, and it serves as a basis for other standards. [31]

Creating a unified decentralized identity ecosystem requires addressing a set of fundamental user needs and technical challenges:

- enabling registration of self-sovereign identifiers that no provider owns or controls
- the ability to lookup and discover identifiers and data across decentralized systems
- providing a mechanism for users to securely store sensitive identity data and enabling them to precisely control what is shared with others

All parts of this ecosystem rely heavily on a public ledger (or other blockchain technology) and cryptography, especially on the public-key cryptography. [31]

1.5.2 Decentralised identifiers

Decentralized identifiers (DIDs) are permanent global identifiers. Their structure is based on URN. The scheme is defined as `did`. Instead of the namespace, there is the so-called DID Method. The last part is string specific for the specified DID method. [32, 33]

DID Definition [32] :

Permanent (persistent) - it does not need to change

Resolvable - metadata are easy to lookup

Cryptographically verifiable - ownership proof by public-key cryptography

Decentralized - no need for a centralized registration authority

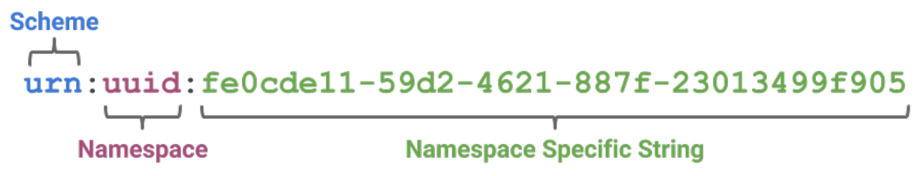

Scheme
urn:uuid:fe0cde11-59d2-4621-887f-23013499f905
Namespace
Namespace Specific String

Figure 1.9: URN Structure [34]

DID identifies DID document. DID document is a piece of data stored in the blockchain. The purpose of DID is to locate the corresponding DID document in the blockchain. This process is called DID resolving. DID and DID document are like key-value pair DID is the key and DID document is the value. [33, 32]

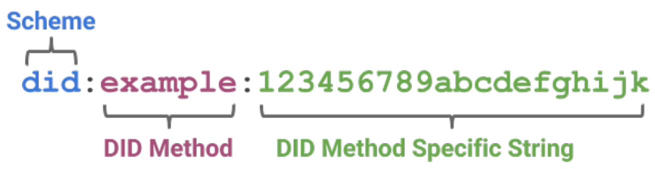


Figure 1.10: DID stricture [34]

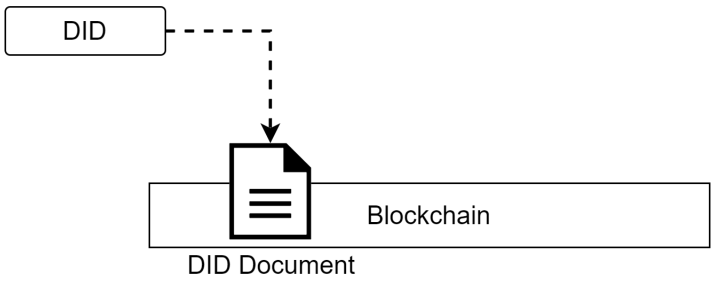


Figure 1.11: Concept of DIDs and DID Documents

Because there are several different blockchain implementations, each with its particular way of storing data, DIDs are specific for each. DID method specification defines used blockchain and storage operations (read/write). DID method-specific string is dependent on this method specification. Its form and value are determined by the underlying blockchain. It can be tied to address on the blockchain or to a specific block or a group of transactions. [35]

More than 30 DID method specifications exist at the time of the writing (2021). [36]

DID method specification defines:

1. syntax of method-specific identifier
2. method-specific elements of DID document
3. CRUD operations on DIDs and DID document for target blockchain

CRUD operations are Create, Read (Resolve), Update, Delete (Revoke). The operations are constrained by the blockchain implementation. For example, Revoke instead of Delete because blockchain is immutable. [32]

DID Examples:
btcr DID method (Bitcoin): did:btcr:x705-jznz-q3nl-srs
sov DID method (Sovereign): did:sov:danube:9iHzzjZVsh6qoWhBLEca78

Universal Resolver is a service for resolving DIDs without the need for resolvers for each DID method. It has a common API for resolving DIDs with a plug-in model for resolving individual methods, called drivers. [37]

DID document is a structured document containing metadata about DID. Thanks to the flexible specification, the format is not strictly defined, but the preferred format is JSON-LD or simple JSON.

Standard DID document content based on [33]:

DID - for self-description

Set of public keys - for verification

Set of authentication methods - for authentication

Set of service endpoints - for interaction between parties

Timestamp - for audit history

Signature - for integrity

Public keys are most important. The user holds his private keys, and his public key is stored in DID document. To verify that the user who provided DID is the owner, the other party uses public-key cryptography. The actual owner can prove that he holds a private key corresponding to the public key in DID document. It is used for this verification and can be as well for authentication.

Service endpoints are off-chain services or user agents that other users can interact with.

Like other identifiers (e.g., URI), DID also supports path, queries, and fragments. DID path identifies resource inside DID document. [33, 32]

DID specification defines two user roles: DID Subject and DID Controller. Whereas the Subject is an entity identified by DID, the Controller is an entity that capability to make changes to DID document. DID Subject and DID Controller can be a single entity, or they could be separate entities. [33]

Using DIDs has many advantages. A user can generate DID for every interaction with the other party, which basically creates a permanent secure channel with that party. It improves anonymity because there is no single DID that could be observed to gather personally identifiable information. Another advantage is that in contrast with the current public-key infrastructure, DID can stay permanent, but keys inside the DID document can be updated. That means that a user can rotate keys for security reasons, but DID stays the same.

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1"
  ],
  "id": "did:btcr:x705-jznz-q3nl-srs",
  "verificationMethod": [
    {
      "type": "EcdsaSecp256k1VerificationKey2019",
      "id": "did:btcr:x705-jznz-q3nl-srs#key-0",
      "publicKeyBase58": "02e0e01a8c302976e1556e95c54146e8464..."
    },
    {
      "type": "EcdsaSecp256k1VerificationKey2019",
      "id": "did:btcr:x705-jznz-q3nl-srs#key-1",
      "publicKeyBase58": "02e0e01a8c302976e1556e95c54146e8464..."
    },
    {
      "type": "EcdsaSecp256k1VerificationKey2019",
      "id": "did:btcr:x705-jznz-q3nl-srs#satoshi",
      "publicKeyBase58": "02e0e01a8c302976e1556e95c54146e8464..."
    }
  ],
  "authentication": [
    {
      "type": "EcdsaSecp256k1SignatureAuthentication2019",
      "verificationMethod": "#satoshi"
    }
  ]
}
```

Listing 2: DID Document Example (shortened)

DIDs create the base layer of decentralized identity infrastructure. Everything other is built on top of this. DIDs can serve as a form of decentralized public-key infrastructure (DPKI). They provide a trusted and verifiable public-key database in a decentralized manner. [38, 32]

1.5.3 Verifiable Credentials

Verifiable credentials sometimes referred to as Verifiable claims (VCs), are equivalent to current physical credentials like ID cards, passports, driving licenses, and qualification certificates. Most of these are still physical, and even though some are already digital, they are still mainly controlled by a

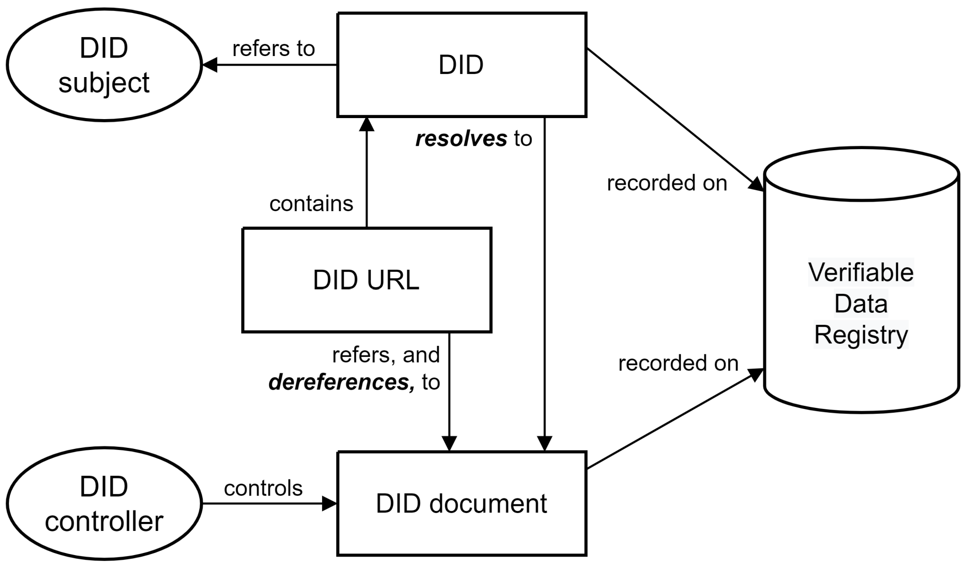


Figure 1.12: Overview of DID Architecture [33]

central authority.

The goal of the verifiable credentials is to provide a standard way to express credentials online in a cryptographically secure, privacy-respecting, and machine-verifiable way. [39, 38]

Basic roles in VC ecosystem defined in [39] are:

Issuer - creates credentials and issues them to the holder

Holder - receives, holds, and shares credentials with verifiers

Verifier - receives and verifies proofs of credentials from the holder

A simplified example is a university (issuer) that issues student ID as VC to a student. The student (holder) wants to use a student discount in a bookstore. The bookstore (verifier) can verify proof of credentials provided by the student and then accepts student discount.

Additional entities in VC ecosystem defined in [40] are:

Verifiable Data Registry - It provides mediation of creation and verification of identifiers, keys, and other data. It can be distributed ledger, decentralized database, or other trusted database.

Subject - entity about which are claims made (holder and subject can be different entities: owner and pet, for example)

Claim - assertion/ statement about subject.

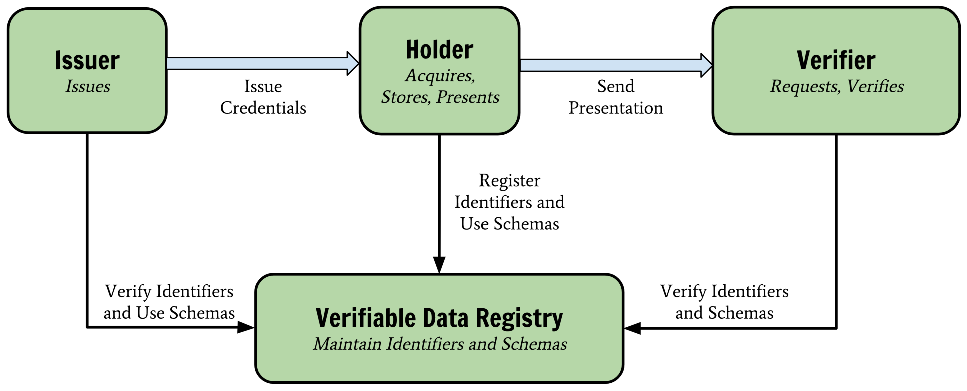

Issuer
Issues
Issue Credentials
Holder
Acquires, Stores, Presents
Send Presentation
Verifier
Requests, Verifies
Register Identifiers and Use Schemas
Verify Identifiers and Use Schemas
Verifiable Data Registry
Maintain Identifiers and Schemas
Verify Identifiers and Schemas

Figure 1.13: Overview of VC Roles and Information Flow [40]

Credential - one or more claims made by the issuer.

Verifiable Credential - a tamper-evident credential that can be cryptographically verified. It can be used to build a verifiable presentation.

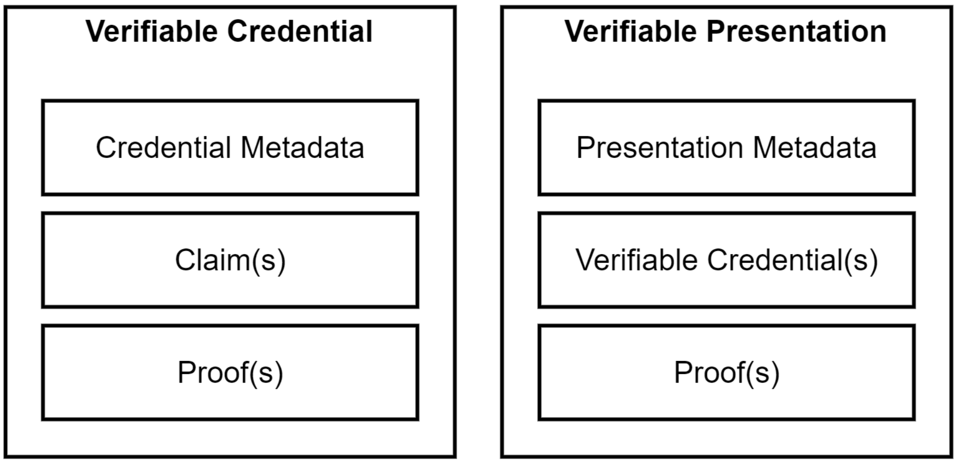
Presentation - data derived from one or more verifiable credentials that can be shared with the verifier.

Verifiable Presentation - a tamper-evident presentation that can be cryptographically verified and with trusted authorship. It may contain data synthesized from original VCs without including original VCs (zero-knowledge proofs).

The fundamental design principle is privacy. Verifiable presentations are used in order not to disclose private credentials. These can represent different subsets of user online identity, so-called personas—for example, professional, family, or incognito persona.

Verifiable presentation expresses data from one or more VCs so that authorship of the data is cryptographically verifiable. The verifiable presentation can contain VCs directly or data derived from them. Verifiable credentials and verifiable presentation supports zero-knowledge proofs to provide some credentials in the form of derived data without disclosing original private data from VCs. An example of this is when a person provides credentials of legal drinking age without disclosing birth date. [40, 39]

Verifiable credentials and verifiable presentations can use JSON-LD or plain JSON format. Proofs (sometimes called signatures) are cryptographic mechanisms used to detect tampering and verify authorship. Currently actively used are Linked Data Proofs (for JSON-LD) and JSON Web Tokens (JWT). [40]



(a) Verifiable Credential (b) Verifiable Presentation

Figure 1.14: VC and VP Content

Verifiable credentials have the potential to fully digitize many processes that were until now not able to because of security or privacy reasons. [41]

1.5.4 DID Authentication

Point of DID authentication for an individual to be its own identity authority, to be its own identity provider, without need for an external provider. It is the concept "I am me." or "I am, whom I claim to be." instead of "Tell me who I am.". [42]

DID Auth is still a concept, not a concrete standard or specification. Hence it is subject to change.

The core idea of DID Auth is proving control of a DID. DID Auth is concerned only with authentication. Authorization is not part of the scope. DID document contains metadata required for authentication: public keys, possible authentication methods, and service endpoints.

The proof of DID control is proving ownership of private key(s) corresponding to the public key(s) inside DID document. Some form of a public-key challenge-response protocol is used. [43]

There are two entities involved in the authentication flow [43] :

The identity owner who controls the DID and is identified by the DID

The relying party that requires proof of control of the DID

```
{

  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "id": "http://example.edu/credentials/1872",
  "type": ["VerifiableCredential", "AlumniCredential"],
  "issuer": "https://example.edu/issuers/565049",
  "issuanceDate": "2010-01-01T19:73:24Z",
  "credentialSubject": {
    "id": "did:example:ebfeb1f712ebc6f1c276e12ec21",
    "alumniOf": {
      "id": "did:example:c276e12ec21ebfeb1f712ebc6f1",
      "name": [{
        "value": "Example University",
        "lang": "en"
      }]
    }
  },

  "proof": {
    "type": "RsaSignature2018",
    "created": "2017-06-18T21:19:10Z",
    "proofPurpose": "assertionMethod",
    "verificationMethod": "https://example.edu/issuers/keys/1",
    "jws": "..."
  }
}
```

Listing 3: Verifiable Credential Example (shortened)

```json
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "type": "VerifiablePresentation",

  "verifiableCredential": [{
    "@context": [
      "https://www.w3.org/2018/credentials/v1",
      "https://www.w3.org/2018/credentials/examples/v1"
    ],
    "id": "http://example.edu/credentials/1872",
    "type": ["VerifiableCredential", "AlumniCredential"],
    "issuer": "https://example.edu/issuers/565049",
    "issuanceDate": "2010-01-01T19:73:24Z",
    "credentialSubject": {},
    "proof": {
      "type": "RsaSignature2018",
      "created": "2017-06-18T21:19:10Z",
      "proofPurpose": "assertionMethod",
      "verificationMethod": "https://example.edu/issuers/keys/1",
      "jws": "..."
    }
  }],

  "proof": {
    "type": "RsaSignature2018",
    "created": "2018-09-14T21:19:10Z",
    "proofPurpose": "authentication",
    "verificationMethod": "did:example:ebfeb1f712ebc6f1c276e12ec21#keys-1",

    "challenge": "1f44d55f-f161-4938-a659-f8026467f126",
    "domain": "4jt78h47fh47",
    "jws": "..."
  }
}
```

Listing 4: Verifiable Presentation Example (shortened)

Basic authentication flow steps are:

1. The relying party creates a cryptographic authentication challenge and sends it to the identity owner.
2. Identity owner constructs cryptographic authentication response to the challenge based on his private key and sends it back to the relying party.
3. Relying party verifies the response with a public key from the DID document. [43]

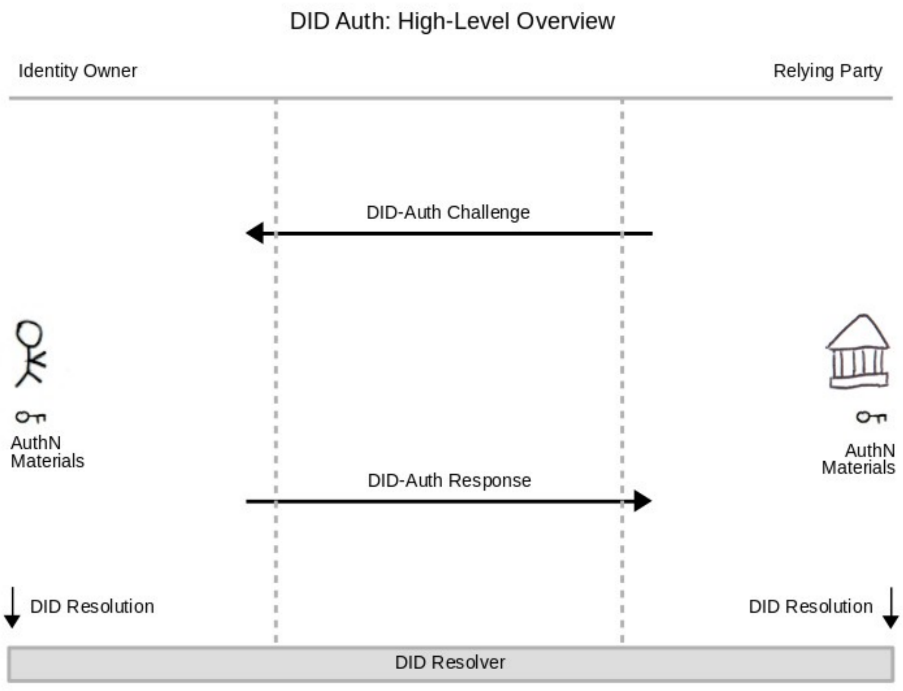


Figure 1.15: Overview of the DID Auth [43]

Challenge properties:

- It can contain the identity owner's DID (It may or may not be known at that time).
- It can contain proof of control of a DID of the relying party. (For mutual authentication). [43]

Response properties:

- It is linked to a challenge (e.g., with nonce).

- It contains proof of control of a DID of the identity owner. [43]

This protocol can be used with various transport mechanisms, such as HTTP POST method, QR code, web browser API, Bluetooth, NFC, etc. The underlying transport mechanism is called transport.

Transport-specific metadata, like endpoint URIs, can be included in the challenge or could be resolved from DID.

As with other DID concepts, a data format for authentication can be JSON Web Token (JWT) or JSON-LD. JWT format is inspired by the OpenID Connect JWT format. JSON-LD format is based on verifiable credentials format. [42]

Different DID authentication architectures can be distinguished by the used transports and challenge and response formats [43]. These architectures differ by:

- whether the DID is known by the relying party beforehand
- challenge and response transport mechanism
- purpose of authentication (user-to-service, user-to-service)
- agents or used for authentication (smartphone, browser, etc.)

DID Auth proposes using Credential Helper API (proposed standard for handling VCs in browser [44]) or Web Authentication (Webauthn API standard for public-key authentication in browsers [45]).

DID Auth also allows for OIDC integration. It can serve as a local authentication method (replacement for username and password). Furthermore, there is also a way for a user to provide identity themselves as Self-Issued OpenID Provider. This personal OIDC Provider then can be discovered from DID document. [42]

This is defined as Self-Issued OpenID Connect (SIOP) [46]. It is joined project of Decentralized Identity Foundation and OpenID Connect Foundation and it is proposed as a web standard. [47, 48]

The goal is to provide backward compatibility with OIDC while enhancing its security with DIDs. [49]

SIOP enables DID authentication and VCs for users instead of relying on companies like Facebook or Google without disrupting their authentication flow. [50]

Apart from authentication, DID Auth could also be used instead of or in addition to static public keys because public keys can be resolved from DIDs, for example, SSH or PGP. [42]

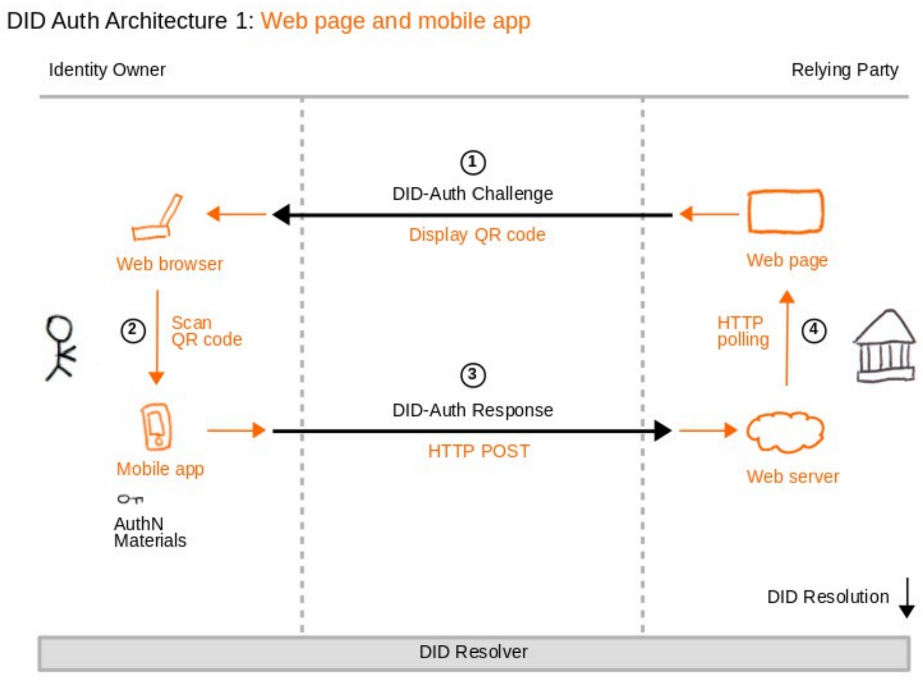


Figure 1.16: Example of Basic Web Page Login Using [43]

1.5.5 Decentralised Key Management System (DKMS)

This part of DID ecosystem is least defined, and there is no standard yet, so this section provides essential concepts of DKMS.

Individuals will use DIDs. It is possible to have thousands of DIDs (theoretically one for each interaction with someone), and each DID can have multiple key pairs. Therefore the essential question is how to manage DIDs and private keys and what to do if a user loses them. [51]

DKMS is an emerging open standard for DIDs and private key interoperable management. DKMS is applicable to identity wallets that store DIDs and private keys as well as to agents that use those wallets. [52]

The goal of DKMS is to standardize wallets to avoid vendor lock-in and ensure security and privacy. [51]

Decentralized identity stack consist of:

DID Layer is the basis for all decentralized identity systems.

Edge Layer are users' personal devices like computers or phones (edge of the network). They are the safest place to store DIDs and private keys, but they are not always online, have low computing power, and be lost.

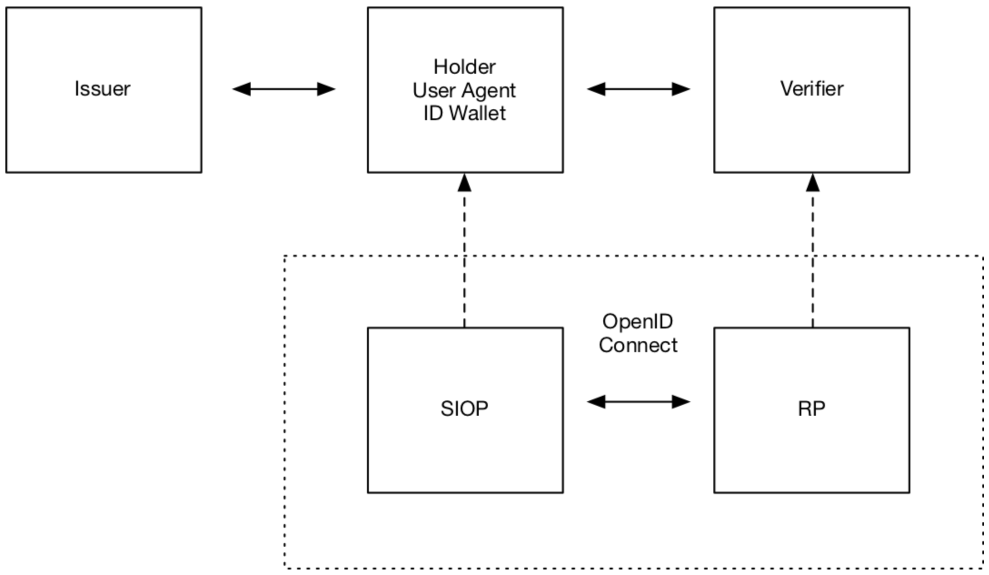


Figure 1.17: SIOP Overview [50]

Cloud Layer is the server layer. It negates the disadvantages of the edge devices in exchange for security and control. It is used primarily for synchronization and recovery. [51]

Wallet standard content:

1. DIDs
2. Key pairs
3. Verifiable Credentials
4. Endpoints
5. Link Secrets
6. Cryptographic Tokens [51]

An agent (user agent) is software that acts as a proxy to a user. It is used to automate establishing connections and cryptographic mechanisms (e.g., digital signing). Sometimes this functionality, together with the wallet, is also called user agent. [53]

Cloud agent's first purpose is DIDs synchronization across multiple edge devices. The second is wallet backup and recovery.

In the case of single device loss, the process of adding a new device and synchronizing is easy because it is the same as adding a new device. In case of loss of all edge devices, recovery is more complicated. [51]

There are two basis forms of DKMS key recovery:

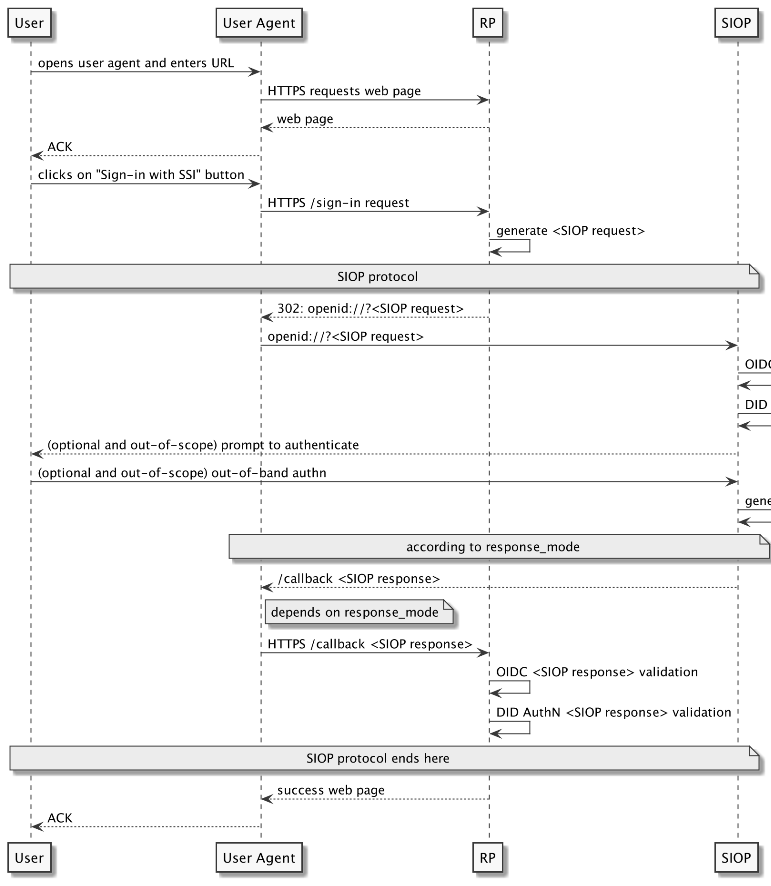

User
User Agent
RP
SIOP
opens user agent and enters URL
HTTPS requests web page
web page
ACK
clicks on "Sign-in with SSI" button
HTTPS /sign-in request
generate <SIOP request>
SIOP protocol
302: openid://?<SIOP request>
openid://?<SIOP request>
OIDC <SIOP request> validation
DID AuthN <SIOP request> validation
(optional and out-of-scope) prompt to authenticate
(optional and out-of-scope) out-of-band authn
generate <SIOP response>
according to response_mode
/callback <SIOP response>
depends on response_mode
HTTPS /callback <SIOP response>
OIDC <SIOP response> validation
DID AuthN <SIOP response> validation
SIOP protocol ends here
success web page
ACK

Figure 1.18: Detailed SIOP Flow [48]

1. Offline Recovery (“Paper Wallet”)

2. Social Recovery (“Trustee System”)

Both these methods use cloud agents. Cloud agent continuously creates and stores a backup copy of the wallet, encrypted with a special recovery key.

Offline recovery consists of physical backup of a recovery key in the form of code, phrases, or QR code stored on paper, another physical medium, or offline HW wallet.

Social recovery uses sharding to split recovery key into multiple parts that are shared with trusted peers, referred to as trustees. This process can be automated using cloud agents. [51]

1.5.6 Other DID Technologies

The decentralized identity ecosystem is still evolving, so here are some other emerging technologies and standards.

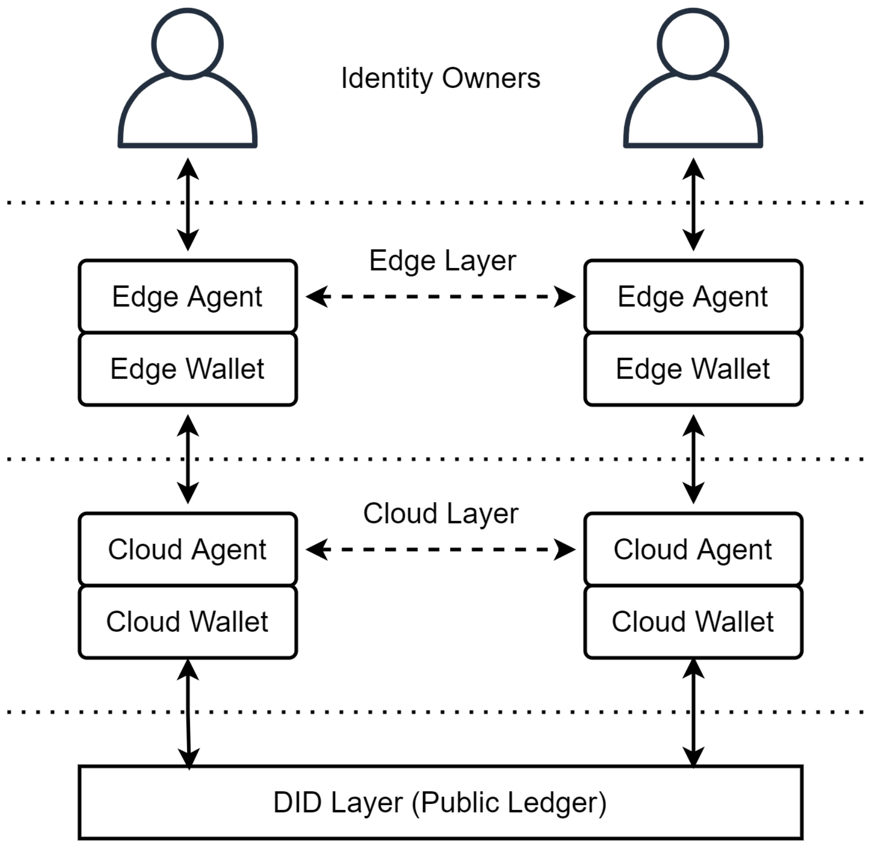


Figure 1.19: DID Stack [51]

1.5.6.1 DIDComm (DID Communication)

“The purpose of DIDComm is to provide a secure, private communication methodology built atop the decentralized design of DIDs.” ([54])

DIDComm is a cross-community proposed standard that proposes design patterns and libraries for direct peer-to-peer communication based on the DIDs. It creates a secure, authenticated channel between agents used by those peers, similar to current end-to-end encryption systems.

This standard then can serve as a base layer for more specialized protocols, called subprotocols. This relation is comparable to HTTP and REST APIs build on top of HTTP.

Apart from usual security and privacy, requirements also include being transport-agnostic, extensibility and interoperability. [55, 54]

1.5.6.2 Universal Resolver and Universal Registrar

Universal resolver and universal registrar are open-source projects that are crucial pieces of DID infrastructure.

Universal resolver enables universal DID resolution, similar to DNS. Universal Registrar can register new DIDs by specified DID method. They both provide DID method-agnostic API accessible by GUI or REST interface. Their modular design is easily extensible for new DID methods. [37, 36, 56]

1.5.6.3 Secure Data Storage (Confidential Storage)

Secure data storage, initially named identity hubs and currently referred to as confidential storage, is an ongoing effort to provide secure, encrypted, privacy-preserving, and provider-agnostic storage of personal data.

Current goals are to create a specification of data models, transport, syntax, data protection, CRUD API, access control, and synchronization compatible with DIDs and VCs standards.

This part of the ecosystem is still heavily underdeveloped and, therefore, very much subject to change. [57, 58]

1.5.6.4 Peer DIDs

Peer DIDs is a new, not yet proven, concept that allows individuals to establish a secure connection with DIDs without using blockchain or public ledger. It is the same concept of DIDs but DID documents are stored in a private ledger shared by two communicating parties.

Many interactions do not strictly require a public ledger, and this method is proposed for these cases.

In theory, every individual can generate new DID for every digital interaction, but it can be technically impractical due to cost and time. That is the main benefit of peer DIDs: scalability and no transaction cost. [59, 60, 61]

1.5.7 Decentralized Identity Community

There is a sizable community around self-sovereign identity and decentralized identity that includes communities like SSI Meetup or My Data Global and companies working on some form of a decentralized identity solution. Many of these companies started working together towards a common goal under the umbrella of the Decentralized Identity Foundation. [62]

The Decentralized Identity Foundation (DIF) is a collection of organizations and contributors working together to establish a decentralized identity ecosystem. The foundation has united a lot of companies and organizations, small and large. Members include Microsoft, Accenture, Hyperledger, The Sovrin Foundation, and many more. [63]

DIF works together with other organizations and foundations like W3C, IETF, OASIS, and OIDF to create open standards for DID ecosystem. [64]

1.6 Chapter Summary

The purpose of this chapter was to provide a theoretical background of blockchain and other related cryptographic technologies that will be needed for the next part of this thesis.

The first part was about blockchain - technology of immutable database in the form of a public ledger. It explained the principles of transaction blocks, consensus algorithm, and mining. It also provided an overview of blockchain characteristics like transparency, decentralization, and security.

The second part was dedicated to smart contracts and decentralized applications. Executable code in the form of smart contracts can be stored and run on a blockchain network, for example, Ethereum. Applications build with smart contracts are called decentralized applications (DApps).

The next part provided an overview of DasContract, a new tool for generating smart contracts from enterprise data models.

The next part explained how zero-knowledge proofs work.

The last part was dedicated to decentralized identity, a new way of proving someone's identity without a central authority. It also included the evolution of digital identity models from centralized to federated and now to self-sovereign, enabled by a decentralized identity ecosystem.

Chapter 2

Current Authentication Methods and Comparison with DID

This chapter deals with current methods of authentication. These include basic username & password, OAuth, OpenID Connect, and SAML. Each method has its dedicated section that explains how it works and what problems it solves. The last part provides a comparison of these methods together with DID. It contains an evaluation of strengths, weaknesses, and possible applications of all these authentication methods.

2.1 Username and Password

Username and password are the most basic form of verifying a user's credentials. A user usually provides a username and password using some type of form. These credentials are sent to a server with an authentication request. [65, 66]

The server then verifies if credentials are valid using some form of the database lookup. If credentials are invalid server responds with a reject response. Otherwise, the server sends an accepting response with a data object that the user uses as proof of authentication. That usually is a cookie (cookie-based authentication) or JWT token (JWT based authentication) in modern web applications. This method is most simple, but it has many drawbacks. [67]

Almost every system has its credential verification system, which means a user must remember large numbers of different passwords risking that he forgets some. Alternatively, users tend to reuse the same password again and again, which is a considerable security risk. [66]

Another possible problem may lay in implementation. All systems that use this method need to manage credentials, which includes transmission via

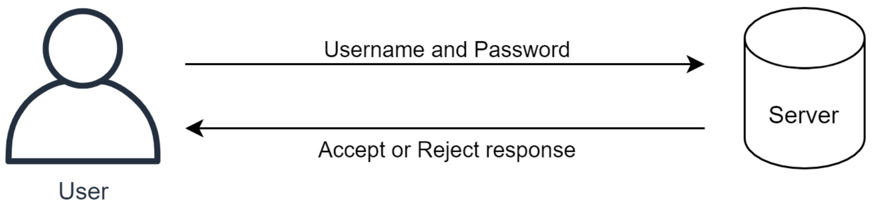


Figure 2.1: Typical Password Authentication

network and storing in database. Possible attackers could exploit these points. Current standards strongly encourage not to store and send plain-text passwords. Instead, a password should be hashed using cryptographically safe hash functions, and only resulting hashes should be transferred or stored. Verification is done by comparing the stored hash against the current one. [65]

Another significant drawback is the inability to integrate with other services for sharing data, for example, user's contacts. This is called delegated authorization. The only possible way for a user to authorize another service to use his data is to provide his username and password directly to that service. This is a huge security risk because that service has full access to his account without any limitation. [68, 66]

2.2 OAuth 2.0

OAuth framework is an open standard used for delegated authorization. OAuth was designed to solve security problems with delegated authorization as well as to allow authorization on non-web applications (mobile apps in most cases). [67, 69, 70]

Even though it was not designed initially with authentication in mind, it can be used that way. That is called pseudo-authentication using OAuth [71]. OAuth 1.0 is deprecated and no longer used, so when speaking about OAuth, it is usually meant OAuth 2.0.

2.2.1 OAuth terminology

Resource owner is a user. He owns his identity and the data associated with it. This data is stored at the resource server.

Client is an application that requests authorization to use the resource owner's data—for example, some e-shop site.

Authorization server is a system that provides authorization to a client. It manages resource owner's credentials. The resource owner authorizes the client by logging in and explicitly consenting to the specified usage of his data. This can be a Google account, for example.

Resource server is a system that stores resource owner's data. It can be accessed by public API, but requests have to be authorized by the authorization server. Resource server and authorization server can be part of the same system, or they could be separated entities.

Authorization grant is proof of authorization by the resource owner. It contains a list of permitted actions, which the client can use to interact with data.

Redirect URI (Callback URI) is Client URI to which the resource owner is redirected after he authorized the client.

Access token is a key that the client uses to interact with the resource server. It is used to verify that client is authorized to use resource owner data.

Scopes are operations used to interact with data. They are defined by the authorization server. That enables granular control over what the client can and can not do with data. The client has to specify which scopes it wants to use. The resource owner has to consent or deny every scope requested by the client. [69, 67, 72, 73, 74]

2.2.2 OAuth Flow

There are two possible ways to conduct authorization using OAuth, called flows. Either authorization code flow or implicit, sometimes called a token flow.

Authorization code flow starts with the client requesting an authorization grant from the authorization server. The client specifies response type as code (authorization code), callback URI and requested scopes.

The user then gets redirected to the login page of the authorization server. After successful login, the user is presented with a form generated from scopes listing all requested permissions. The user decides which permission to accept or decline.

The authorization server then invokes the Callback URI with a response. The response contains an access code. The client then exchanges the access code for the access token with the authorization server.

That is done via the client back end server to improve security because this exchange needs a shared secret between the client back end server and the authorization server. Also, the access token stays on the server, which is safer than the front end (web browser). Now the client has the access token, which is used to request data from the resource server. The resource server verifies the validity of the token and responds accordingly. [67, 72, 74]

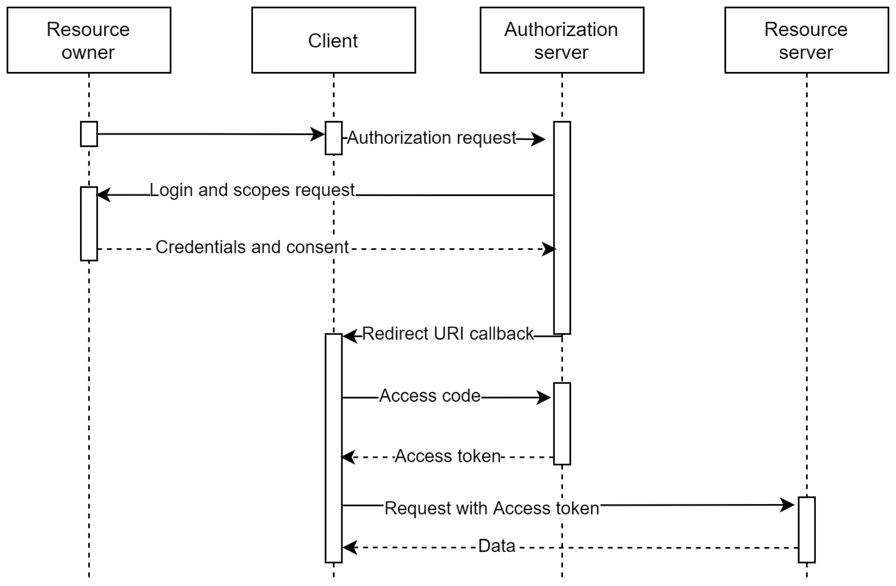


Figure 2.2: OAuth Authorization Code Flow

Implicit flow is similar to the authorization code flow. It is used when the client does not have a back end server to manage code for the token exchange, for example, JavaScript-only web applications or mobile apps. Therefore client specifies response type as a token instead of code. The authorization server then responds already with the access token.

This method is considered less secure because client-side code is responsible for token management. It could potentially be exploited using DevTools in a browser, for example. [67, 74]

2.2.3 Pseudo-authentication using OAuth

OAuth is an authorization framework. Authentication is not part of the standard, but it can serve as a basis for an authentication protocol [75]. This led to several nonstandard solutions by major web companies like Facebook and Google. These implementations were not interoperable and not entirely safe. Using OAuth to implement authentication is called pseudo-authentication because it is indirect authentication based on authorization token. [70, 66]

In order for the client to authenticate a user, it needs information about the user (e.g., email), but OAuth does not specify identity information exchange in the flow. Therefore possession of an access token can be interpreted as proof of authentication. However, the access token is designed to be opaque to the client, and in order to authenticate the user, the client needs to derive

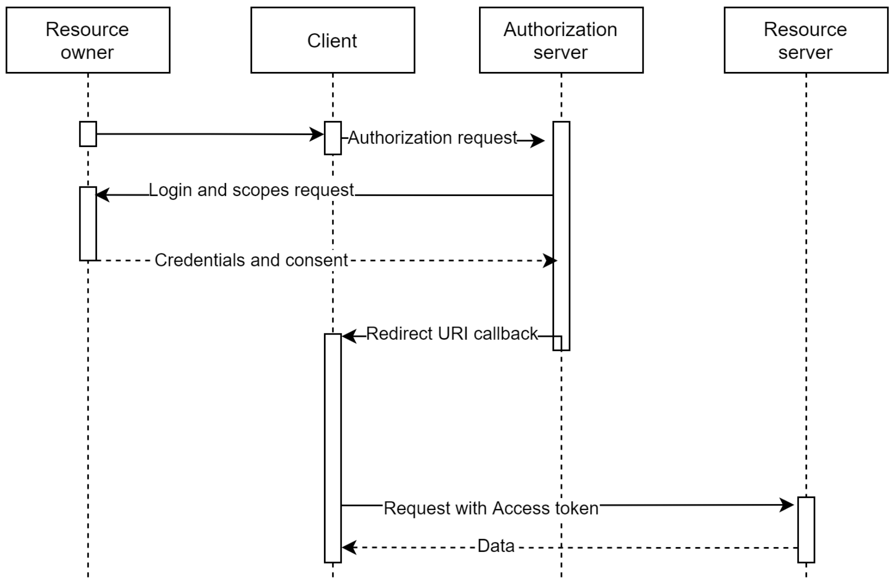


Figure 2.3: OAuth Implicit Flow

this information from the token. In that case, the token has to have some type of format to carry that information.

Another possibility of authentication is to use an access token to access protected identity API, which provides identity information for the valid access token.

Both variants are based on the assumption that possession of a valid access token is enough proof that the user is authenticated. This is true in most cases. However, tokens have different lifetimes and can be recycled. This means that tokens can be used long after being issued, even without the user's knowledge. Another vulnerability is the security of the tokens. Tokens can be leaked from an application and used to get identity information, or other tokens can be injected into the application to impersonate other users. [76, 73, 71, 70]

2.3 OpenID Connect

OpenID Connect (OIDC) is an identity layer on top of OAuth 2.0. It extends OAuth to standardize a way for authentication [75, 66]. OpenID Connect adds an ID token, a standard set of scopes, and a standardized UserInfo endpoint for additional user information. OpenID Connect was designed to solve problems of pseudo-authentication with OAuth. OpenID Connect is most current version of OpenID, previous versions called OpenID 1.0 and

OpenID 2.0 are deprecated. [76, 73, 77]

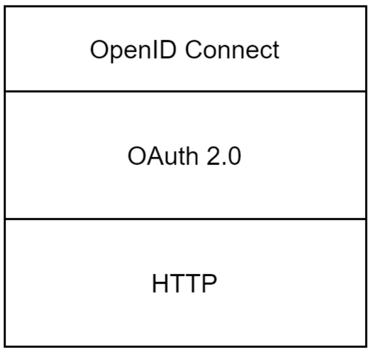


Figure 2.4: OpenID Connect Structure

2.3.1 Terminology

In OpenID Connect most roles are similar to that of the OAuth but have different name.

End-User is a human participant. In OAuth, it refers to the resource owner.

Relying Party is equivalent to the Client application in OAuth.

Identity provider is an OAuth authorization server implementing OpenID Connect. It authenticates End-Users and provides information about the user and authentication event to Relying Party.

Identity token (ID token) is JSON Web Token (JWT) containing information about an authentication event. It can also contain other data as well. [67, 72, 73, 77]

2.3.2 JWT and claims

JWT is an internet standard for creating data with optional signature and optional encryption. JWT has three parts: header, payload, and signature. A payload is a JSON object containing a set of claims. [78, 79]

A claim is a statement about a user or authentication event. Some standard claims are `iss` - token issuer, `email` - users email, or `exp` - token expiration date. Claims are implemented as key-value pairs. To request specific claims corresponding scopes have to be used. JWT is designed as a method for representing claims securely between two parties. [67, 72, 80]

2.3.3 OpenID Connect flow

OpenID Connect flow is based on OAuth flow. There are also two flows: authorization code flow and implicit flows. Relying party initiates authentication request by OAuth authorization request with mandatory scope `openid`.

Then the flow is the same as with OAuth. However, when exchanging access code for access token Identity token is also received. In the case of implicit flow, the Identity token is also received directly by URI callback with the access token. [67, 80, 73, 74]

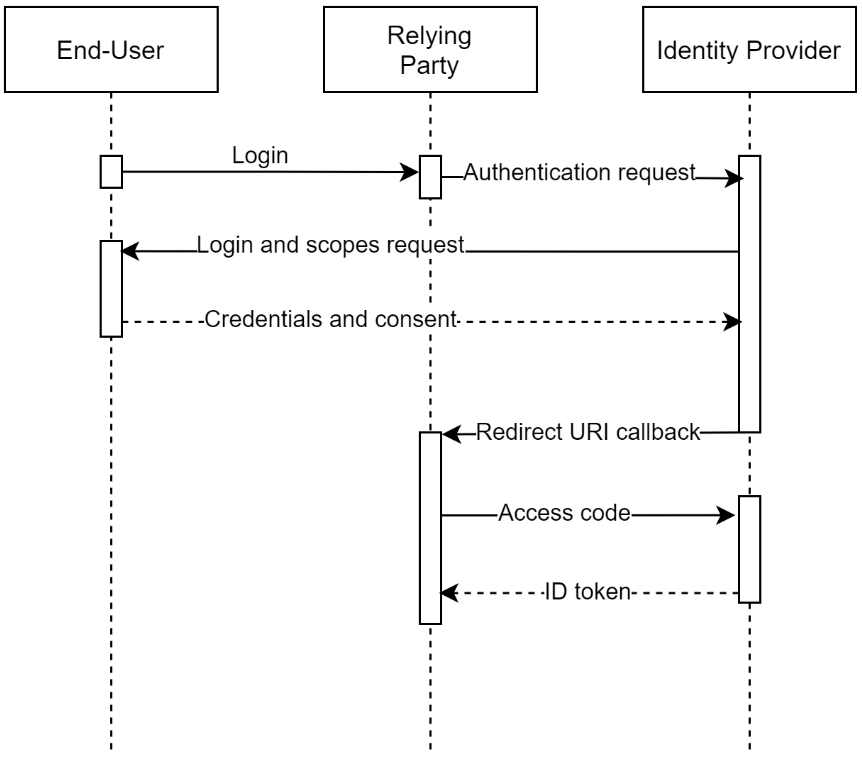


Figure 2.5: OpenID Connect Authorization Code Flow

2.4 SAML protocol

SAML is an open standard protocol for authentication and authorization. SAML stands for Security Assertion Markup Language. It is designed to communicate identities between organizations, and it is based on an exchange of XML messages. SAML is much older than OAuth and OpenID Connect. The current version is version 2.0 from 2005. [81, 82, 83]

The primary use case of SAML is Single Sign-On (SSO). That allows users to access multiple related systems using only one set of credentials without being asked to re-authenticate when already logged-in. An example of SSO is multiple CTU systems using Shibboleth [84] to authenticate students. [82, 66]

2.4.1 Terminology

User Agent (sometimes called just user) is a client that the user uses to access services. In most cases, it is a web browser.

Service Provider is an application that the user wants to access and requires user authentication.

Identity provider is a service that provides a directory of users and an authentication mechanism.

Assertion is XML message containing authentication data. It includes the method of authentication, user attributes, issuer, and other metadata for security. [81, 82, 66]

2.4.2 SAML Authentication Flow

For SAML authentication to work, Identity Provider and Service Provider have to establish trust using certificates and set matching configuration beforehand.

A user can initiate authentication flow either from an Identity provider or from Service Provider. In the case of Service Provider initiated authentication flow, it starts with the user accessing Service Provider. If the user is not yet authenticated, he is redirected to the identity provider's login screen. After successful login, the User-Agent receives SAML Assertion and is redirected back to Service Provider. Service Provider verifies SAML Assertion and uses it to authenticate the user. The user then can use services or applications provided by Service Provider. [82, 66, 83]

Identity initiated flow is almost the same. User starts by logging-in to Identity Provider, and only after that when he is already authenticated, he accesses Service Provider. [83]

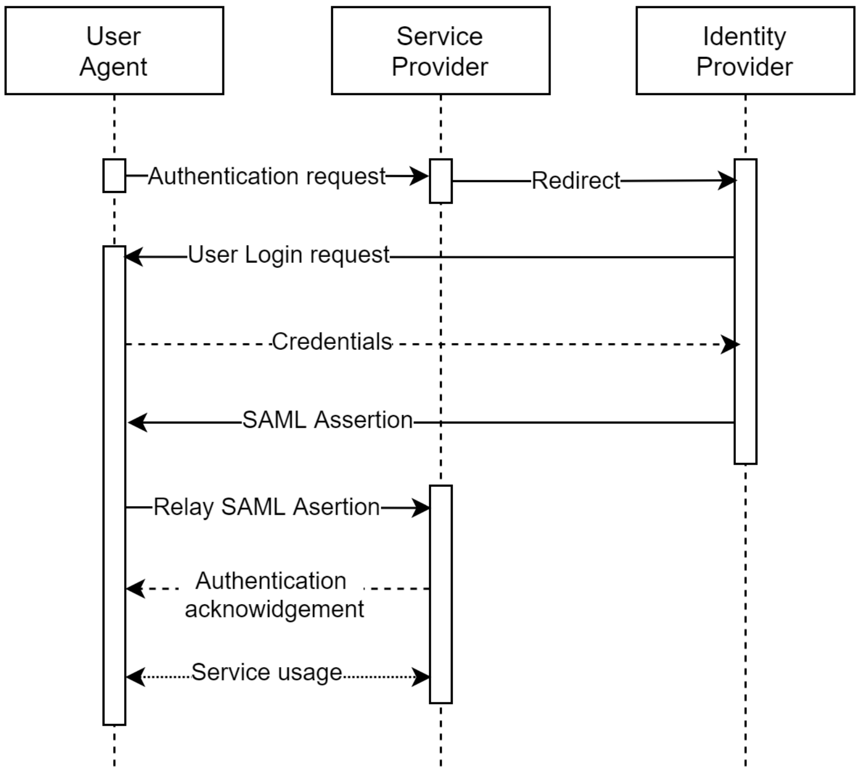


Figure 2.6: SAML Service Provider Initiated Authentication Flow

2.5 Comparison of Current Authentication Methods and Decentralized Identity

2.5.1 Username and Password

Username and password is the most straightforward method. Because of that, it is easy to implement. It is also the most common, so it is comprehensible to almost every user.

However, its security depends heavily on implementation. Having users remember all the passwords is a potential security risk also. It is impossible to reliably and securely integrate this method with other systems. [68, 65]

2.5.2 OAuth

Authentication using OAuth is possible (Pseudo-authentication), but it is not advised. Every such solution is non-standard proprietary and prone to security

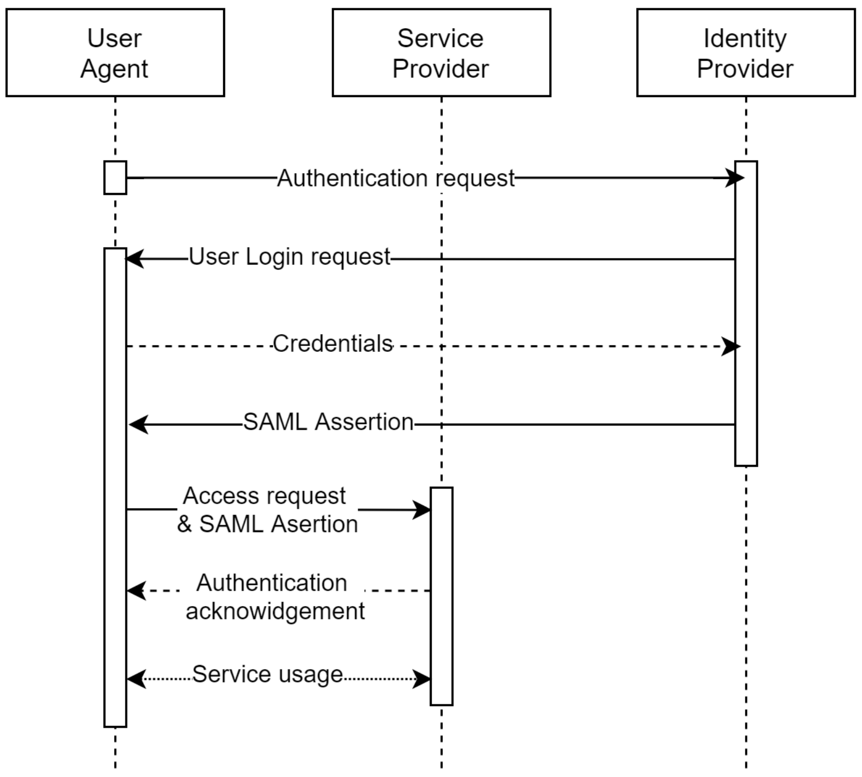


Figure 2.7: SAML Identity Provider Initiated Authentication Flow

issues. That is why OpenID Connect exists and why all major social sites login moved away from this. [76, 66, 70]

2.5.3 OpenID Connect

OpenID Connect solves issues of OAuth authentication. It is a safe, open standard, and it is widely used by many. It has one major issue, and that is control of the user's identity. The user has single credentials setup with the Identity provider and uses it login to his services.

Nevertheless, this single identity provider is a single point of failure. If its service becomes unavailable for whatever reason, users can not access their data or use services. Furthermore, even though OpenID Connect authentication is safe, an identity provider that stores personal data can be attacked by malicious parties. Another problem is trust. User entrusts control of his identity to the identity provider. There is no practical way to ensure the misuse of personal data. [66, 80]

2.5.4 SAML

SAML is also a safe, open standard proven by years of use. However, due to this design's age, it can be challenging to integrate with the modern system. It uses XML instead of more lightweight JSON. Integrating RESTful APIs and XML messaging is not that simple. Moreover, due to more complex flow, implementing client is much more complicated than with OpenID Connect. Mobile devices were not part of the design. [66, 74]

Apart from these technical disadvantages, it also shares the same problems of having a single identity provider as OpenID Connect.

2.5.5 Decentralized Identity

Decentralized identity is a new identity and authentication ecosystem. It is a way to reach a self-sovereign identity. It builds on solid foundations of proven cryptographic mechanisms and mature blockchain technology. It provides trust, security, and privacy while being flexible to support any public ledger or blockchain, multiple data formats, and transport layer technologies. Even though much work has been put into it, it is still not fully defined and standardized[2]. That is the reason why it is not yet fully adopted[2].

The goal of the decentralized identity is for individuals to regain control over their digital identities without depending on external authorities, specific vendors, or single technology stack. From a technological point of view, being a decentralized system avoids a single point of failure and other client-server disadvantages. Personal data and credentials are entirely under users' control and protected by encryption. However, this can be problematic in the case of a private key loss.

Decentralized identity solves problems of federated identity, like OIDC and SAML. It is a pretty complex concept and can be hard to implement in current applications due to incomplete standards and missing SDKs and libraries[2].

While it provides many benefits for individuals and organizations alike, some users may not like the idea of using cryptographic wallets or other user agents just to log in to a web page.

2.5.6 Authentication Methods Comparison

This table provides overview of the authentication methods by these categories:

Identity Model

Decentralized

Authorization Support - whether it natively supports authorization.

[2]At the time of the writing (2021).

Claims Support - whether that method can be used to provide personal data or credentials.

Standardized

Security Risk

Integration Difficulty - How difficult it to implement and configure authentication for application.

SSO Support - whether it can be used to provide SSO capability.

Platform Constraints

Popularity - how much it is being used in current applications.

User Familiarity - how familiar are the users with that method.

	Password	OAuth	OIDC	SAML	DID
Identity Model	Centralized	Federated	Federated	Federated	Self-Sovereign
Decentralized	no	no	no	no	yes
Authorization Support	no	yes	yes	no	no*
Claims Support	no	no	yes	no	yes
Standardized	no	no	yes	yes	yes**
Security Risk	high	medium	low	low	low
Integration Difficulty	low	medium	medium	high	high***
SSO Support	no	yes	yes	yes	yes
Platform Constraints	none	none	none	only web apps	none
Popularity	high	medium	high	medium	low
User Familiarity	high	medium	medium	medium	low

Table 2.1: Authentication Methods Comparison Table

* Not part of the standard, but it may change
** Mostly
*** Libraries and SDKs mostly not yet implemented

Username & Password Applicable to a smaller or private project due to the easy implementation.

OAuth Do not use for authentication, prefer OIDC.

OIDC OIDC is useful for larger projects, social applications or when SSO is required.

SAML SAML can be required by the project constraints, but rather use OIDC.

DID When security, privacy or anonymity is primary concern, for DApps ecosystem or as bleeding edge technology for long term project.

2.6 Chapter Summary

This chapter consisted of an analysis of the current authentication methods.

The first part reviewed classic username & password authentication and its security weaknesses.

The second part explained how the authorization framework OAuth works and how it can be (ab)used to be used for authentication.

The following two parts were dedicated to OIDC and SAML. Both OIDC and SAML are mature authentication methods for federated identity.

The last part provided an analysis of mentioned methods and a method based on decentralized identity. It discussed the strengths and weaknesses of these methods, their differences, and their suitable use cases. And while most current methods are sound and usable, the DID has vast potential for the future.

Chapter 3

Using DID in Decentralised Applications (and DasContract)

This chapter aims to investigate the possibilities of using decentralized identity and its ecosystem in DasContract and all DApps generally. It will focus on DID authentication and verifiable claims especially, as they provide a convenient way of access control for smart contracts.

3.1 Smart Contracts Access Control

Smart contracts are similar to the current backend servers that are usually used when developing applications. They provide API for interaction with the system. API access is often restricted for security and system integrity reasons, and developers allow access only to authorized entities. An entity that wants to use the API has to be authorized first, and after that, when making an API request, it has to provide proof of authorization. That is the core idea behind access control.

3.1.1 What is Access Control

Access control is a security mechanism used to protect computer systems and APIs. Essentially it controls incoming requests and rejects those from unauthorized entities. The essential components of access control are a resource, subject, and action. The resource is an asset that the subject wants to access. The subject is an entity requesting access to the resource. It can have various attributes, like identifiers, roles, and credentials. The action is what the subject wants to do with the resource.

There are multiple access control strategies, but role-based access control (RBAC) and attribute-based access control (ABAC) are widespread. With RBAC, a system defines roles, and each user (subject) in the system is assigned a role or multiple roles. A role contains a set of permissions and a set of resources allowed to access. In an ABAC system, there is a set of rules and policies for each resource that defines which combination of the subject's attributes allows access to the resource. [85, 86, 87]

3.1.2 Current Access Control Solutions

In permissioned blockchain networks, an access control layer can be based on blockchain permissions [10]. However, public permissionless blockchain, by definition, can not use that. Smart contract published in public permissionless blockchain can be accessed by anybody. Only the contract's address and its interface are needed to access the published contract. Furthermore, even though neither of these is explicitly public knowledge, both can be found out. Therefore relying on not sharing these is not adequate security.

In current systems, authorization proof is done via a token, like a cookie or JWT, appended to a request. However, using these standard tokens can be extremely difficult due to inefficient string parsing and manipulation in some blockchains and smart contract programming languages, like Ethereum and Solidity. So different access control method is required. [88, 89]

Rudimentary role-based access control can be implemented by using account addresses as identifiers and specifying the access control list of addresses for each role inside the contract. Contract functions are called by sending signed transactions. Before executing a function, the contract checks whether the sender account has been assigned the required role by checking the address list for that particular role. For example, only the owner (an account that published the contract) can use all contract functions, and a few selected accounts (accounts assigned a specified role) can use a limited subset of the contract's functions. [90]

This access control method has several drawbacks. A contract has to include all the access control implementation. It also has to store all verified addresses and associated data inside the blockchain. That means that it can be very costly, depending on the number of addresses. Access rules stored inside a blockchain or even hardcoded into the contract. Because of that, it can be hard to change these rules. Also, using only addresses does not provide granular control over access rules. [91]

3.1.3 Using Decentralized Identity For Access Control

To summarise, a decentralized identifier (DID) can be used to identify and authenticate an entity (individual or organization). Verifiable credentials (VCs) contain claims about an entity identified by DID. Entities can create and sign credentials using their DIDs, and issue them to other entities. The validity of any verifiable credentials could be cryptographically verified. For example, a government could issue digital ID cards in the form of verifiable credentials. Citizens then could provide said credentials to any state institution they interact with to prove their citizenship, instead of the usual physical ID card.

That leads to the idea of using decentralized identity (DIDs and VCs) to provide an access control mechanism for smart contracts, especially on public permissionless blockchains. A system can specify access rules and permissions based on credentials (DID with VCs). That means that access rules are no longer tied to some arbitrary address but to an entity identified by a global universal identifier in the form of the DID. Also, the system can use provided verifiable credentials as reliable and verified user attributes.

Decentralized identity is flexible enough to be used in RBAC or ABAC system. A system could assign roles according to provided credentials, or it could control access by using credentials as attributes.

3.2 Proposed Access Control Methods

An access control system based on the decentralized identity can use one of four possible methods. The four methods can be divided into two types: direct and indirect.

3.2.1 Direct Access Control

Direct access control accepts DIDs and VCs as part of the contract function call. After verifying DID or VCs with access rules, it executes called function.

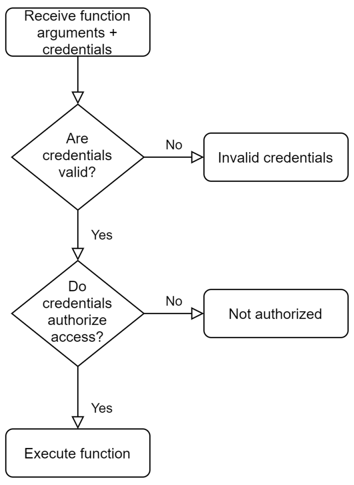


Figure 3.1: Direct Access Control Flowchart

An entity provides DID or VCs together with function arguments. Smart contract authenticates DID and verifies VCs. After that, it checks whether provided DID and VCs meet access rules requirements for this function, essentially whether this entity can be authorized on the basis of provided credentials. If yes, then the function executes.

The first direct method is called on-chain verification because DID authentication and authorization are done in the smart contract. This method is theoretically possible but currently extremely hard to implement. DID standards rely on JSON-LD and JWT, and so parsing would be very difficult [88]. There is a proposed Ethereum standard for Ethereum VCs compliant implementation [88]. However, even if usable, it would mean that every blockchain would have its implementation, and an identity wallet used for authorization would need to support them.

The second direct method of this type is called off-chain verification. It uses the same process as before, but the verification process is outsourced to a separate service accessed via an oracle. That resolves the problem with parsing and verifying credentials, but it also brings problems with a contract to oracle communication.

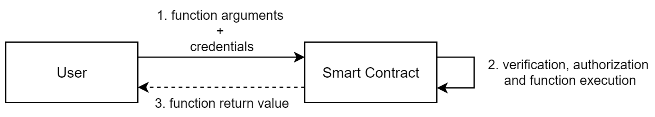


Figure 3.2: On-chain Verification Flow

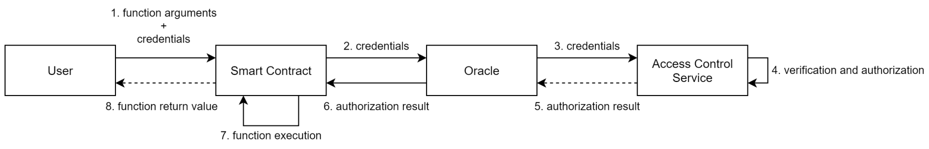


Figure 3.3: Off-chain Verification Flow

3.2.2 Indirect Access Control

Indirect access control differs by having separate off-chain authentication and authorization service to verify credentials and grant access accordingly.

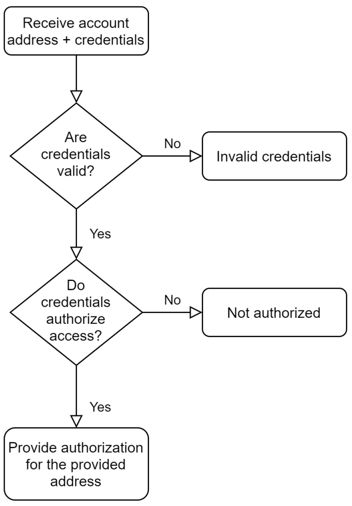

Receive account address + credentials
Are credentials valid?
No
Invalid credentials
Yes
Do credentials authorize access?
No
Not authorized
Yes
Provide authorization for the provided address

Figure 3.4: Indirect Access Control Flowchart

An entity provides DID and VCs together with an account address to the access control service. The service authenticates DID and verifies VCs. Then it assigns a role to the entity according to the provided credentials. The authorized entity can then call a smart contract that just verifies the validity of the assigned role before function execution.

The first indirect access control method is called on-chain RBAC. This method is an extension of RBAC based solely on the accounts' addresses. The access control service verifies user credentials and assigns a role to the user by adding provided account address to the access control list for that role in a smart contract. After that, the user's account is authorized to call functions, which require that role. The advantage of this improvement is that users can be authorized automatically by providing valid credentials. However, it also shares many drawbacks of purely address-based on-chain RBAC, like storing access control lists in smart contracts.

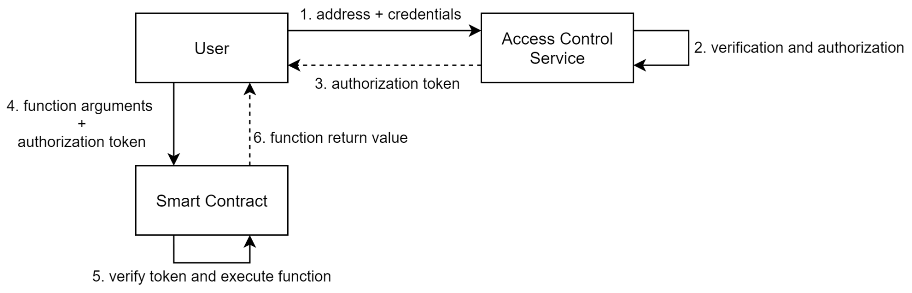

User
1. address + credentials
Access Control Service
2. verification and authorization
3. authorization token
4. function arguments + authorization token
6. function return value
Smart Contract
5. verify token and execute function

Figure 3.5: On-chain RBAC Flow

The other indirect access control method is called off-chain RBAC. With this method, the access control service creates and issues authorization tokens to verified users. The token can contain access rules, like roles or other attributes assigned to the user. The token is signed by the access control service to ensure its security. The format of the token has to be chosen with easy parsing in mind. The user then provides the token when calling a smart contract function. The contract verifies authorization of the user from the token before function execution.

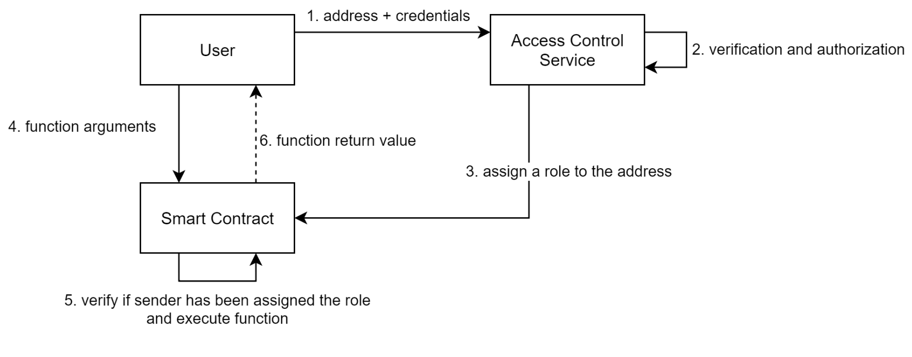

User
1. address + credentials
Access Control Service
2. verification and authorization
4. function arguments
6. function return value
3. assign a role to the address
Smart Contract
5. verify if sender has been assigned the role and execute function

Figure 3.6: Off-chain RBAC Flow

3.3 Using Decentralized Identity in DasContract

DasContract is used to create smart contracts in the form of business process modeling. It heavily relies on the BPMN to model contracts. BPMN uses user tasks (UserTask) to represent user interaction. DasContract metamodel defines ProcessUser and ProcessRole. ProcessRole describes a contract actor role, for example, an accountant. ProcessUser is an individual actor. ProcessUser can have multiple ProccessRoles, and multiple ProcessUsers can have the same ProccessRole. ProcessUser and ProcessRole are used for specifying UserTask. [17]

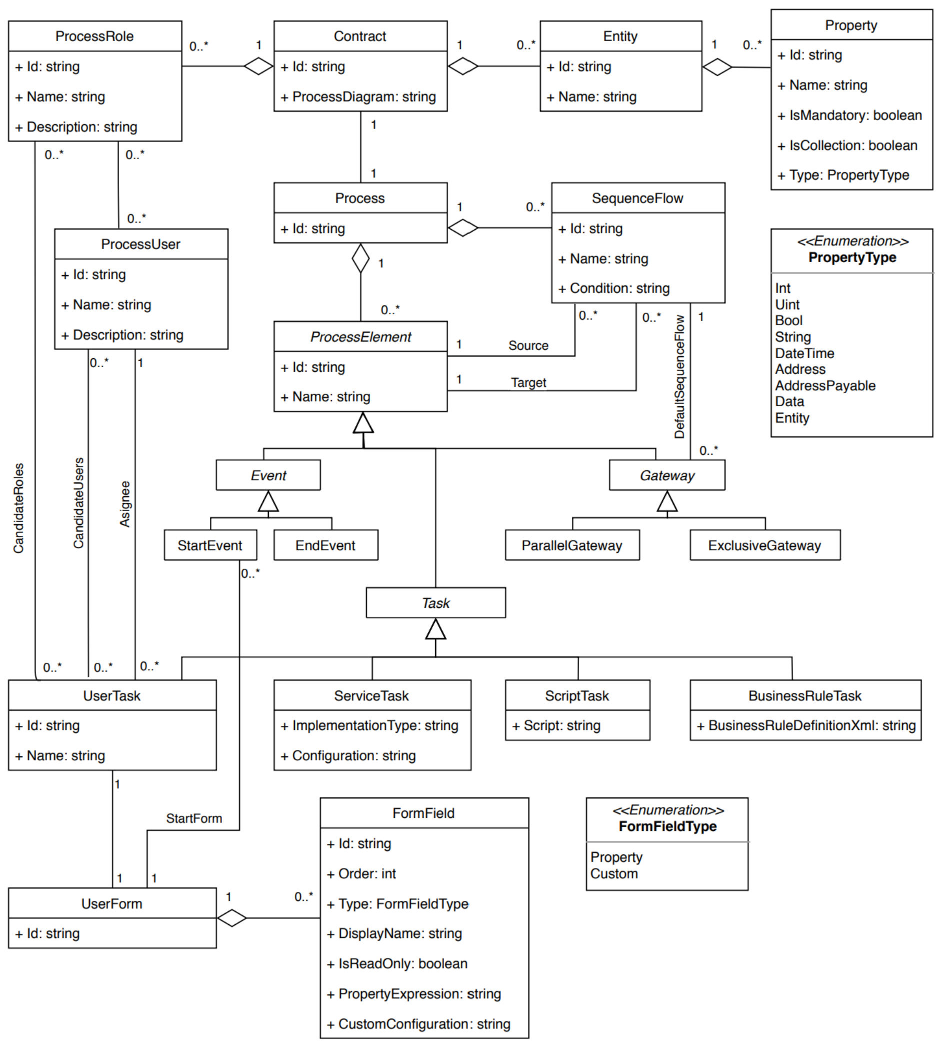


Figure 3.7: DasContract DSL Metamodel [17]

Therefore when designing a contract, specific requirements for ProcessUser and ProcessRole could be specified. For a user, it could be specific DID (DID of an actual individual) or verifiable credentials required from that user, for example, government-issued ID. For a role that requires a specific qualification, that qualification could be specified via required verifiable credentials.

This naturally correlates with the role-based access control mentioned earlier. DasContract could generate access control rules from specified requirements. In the case of on-chain credentials verification, the rules would be generated as smart contract code. In the case of off-chain credentials verification, the rule could be generated as a configuration file for access control service.

Another possible application could be issuing credentials or certificates based on the conditions in a contract. If the user fulfills conditions specified in the contract, he can be issued an official confirmation document in the form of verifiable credentials that can entitle him to additional actions. For example, after repaying the loan, the borrower can be issued confirmation of repaying the loan.

From a contract design point of view, DIDs and VCs could provide a way of specifying process user and process role requirements when designing a user task. Additionally, DasContract could also specify credentials issued according to the contract to users in the form of the VCs. All this is possible in a standardized universal way.

3.4 Chapter Summary

This chapter dealt with using decentralized identity in DApps as well as in DasContract.

Unlike current centralized systems, DApps based on a public permissionless blockchain do not have a clear way of implementing access control.

The first part briefly explained access control strategies, like RBAC and ABAC, and how decentralized identity could be used to implement access control.

The following part outlined four different methods of implementing access control based on the decentralized identity.

The last part was about using decentralized identity to specify required credentials for users and roles in DasContract. And how VCs can be used to express credentials gained from contracts.

Chapter 4

DID Authentication Architecture Compatible with OpenID

The goal of this chapter is to propose an architecture of the DApp system. The system has to have access control based on the decentralized identity. The access control mechanism should be compatible with DasContract, and it also has to be compatible with OpenID Connect.

4.1 System Architecture

The proposed DApp architecture is based on the indirect off-chain RBAC method defined in the previous chapter. It is inspired by the Smart Contract Access Control Service (SMACS) [91].

The architecture consists of three main components: Access Control Service (ACS), Client Application (Client App), and Smart Contracts Component (Smart Contract).

Client Application is the front-end of the DApp. Its purpose is to interact with a user and communicate with Smart Contract Component.

Smart Contract Component consists of smart contracts collection in blockchain serving as the back-end for the DApp.

Access Control Service grants Smart Contract access to the Client App based on provided credentials. It also issues VCs to a user through the Client App.

4.2 Capabilities

The Access Control Service design has two main capabilities: granting access by verifying credentials (DID and verifiable credentials) and issuing credentials in the form of verifiable credentials.

4.2.1 Granting Access (Authorization)

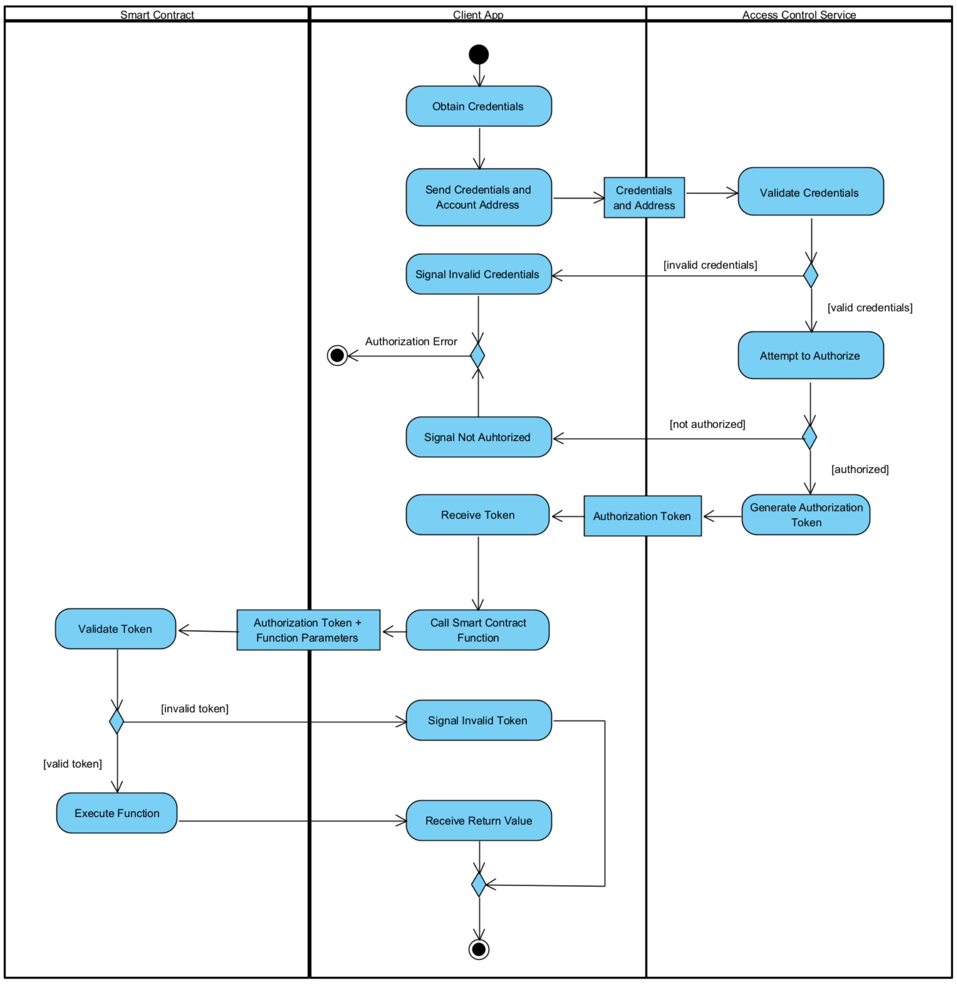


Figure 4.1: Conceptual UML Activity Diagram of Granting Access

Conceptual access granting flow steps are following:

1. The Client App sends credentials with user blockchain account address to the ACS.
2. Access Control Service verifies credentials and authorizes a user by generating the Authorization Token (Access Token).
3. With the Authorization Token, the Client App can interact with the Smart Contract.

4.2.2 Issuing Credentials

Conceptual credentials issuing flow has the following steps:

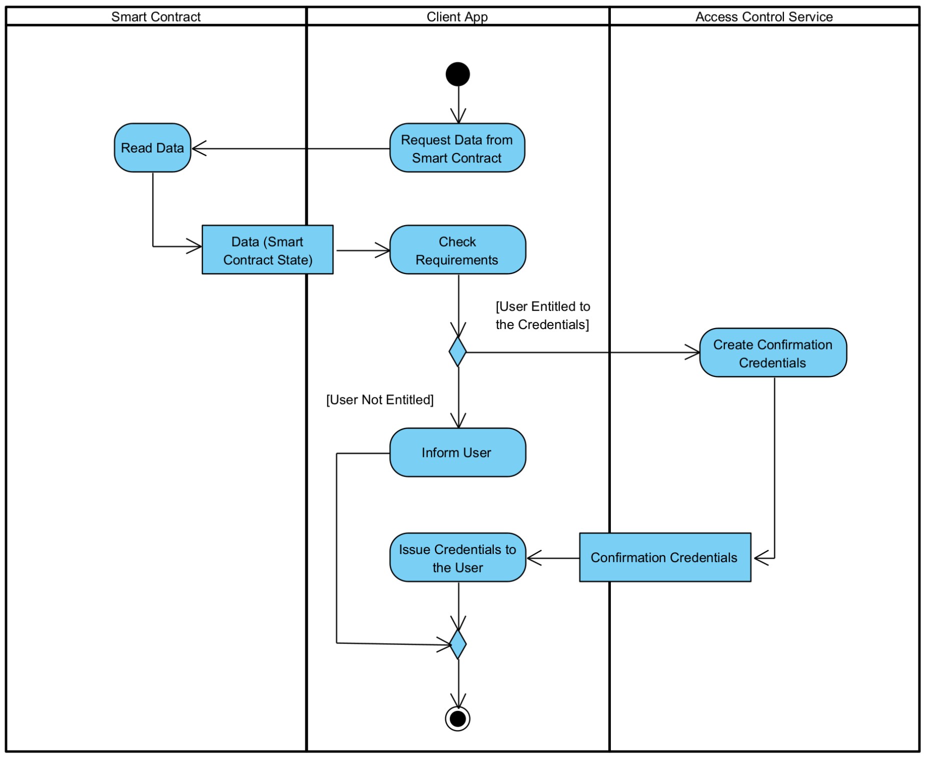

Smart Contract
Client App
Access Control Service
Read Data
Request Data from Smart Contract
Data (Smart Contract State)
Check Requirements
[User Entitled to the Credentials]
Create Confirmation Credentials
[User Not Entitled]
Inform User
Issue Credentials to the User
Confirmation Credentials

Figure 4.2: Conceptual UML Activity Diagram of Issuing Credentials

1. The Client App gets the Smart Contract state.
2. If the state entitles the user to some credentials (like proof of payment), the Client App will request credentials issuance from the ACS.
3. The ACS then generates credentials and issues them to the user via the Client App.

4.3 Components

4.3.1 Access Control Service

Access Control Service is, as the name suggests, responsible for access control. It uses the Decentralized Identity Module for authentication, verified credentials verification, and verified credentials issuing.

For granting access to the Smart Contracts, it uses authorization tokens. The ACS has key pair (public key pk_{ACS} and private key sk_{ACS}) generated.

It creates the token data by concatenating a string containing a role assigned to a user with the user's account address. Then it hashes the token data. The token is created by signing this hash.

$$token = Sign_{sk_{ACS}}(Hash(role \cdot address))$$

Figure 4.3: Authorization Token Generation

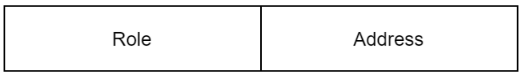


Figure 4.4: Structure of an Authorization Token Data

4.3.2 Decentralized Identity Module

The Decentralized Identity Module is an external component. It deals with DIDs and verifiable credentials. It mainly consists of Identity Wallet, libraries, or SDK to interact with DID ecosystem and transport mechanism for a Client App to Identity Wallet communication.

Identity Wallet is a DID identity user agent. It stores user DIDs and corresponding VCs. Typically, the mobile application would implement the role of the Identity Wallet. Identity Wallet requirements:

- authenticate user
- provide requested verified credentials (disclose verified credentials)
- receive new issued verified credentials

The libraries provide a mechanism for creating VCs and implementing the Credentials Disclosure Flow, which is used for user authentication and requesting and verifying credentials.

Credentials Disclosure Flow:

1. The Access Control Service creates the Credentials Disclosure Request and sends it to the Client App.
2. The Client App uses a specified transport mechanism to transfer the request to the Identity Wallet. The proposed system uses QR codes containing encoded Credentials Disclosure Requests that the wallet can scan.
3. The user then accepts the disclosure request. After that, the wallet generates the Credentials Disclosure Response and sends it to the ACS.
4. The ACS uses provided library to verify the response.

Credentials Disclosure Request contains a list of required VCs and DID authentication challenge. It also contains disclosure response callback URI.

Credentials Disclosure Response contains a list of disclosed VCs and signed DID authentication challenge.

The Decentralized Identity Module could also use zero-knowledge proofs to disclose credentials synthesized from existing credentials without disclosing existing credentials directly. For example, a user could disclose credentials proving he is at least 18 years old without disclosing his date of birth.

4.3.3 Smart Contracts

The Smart Contract Component is a collection of smart contracts serving as the back-end of the system. It contains at least two contracts: a contract that requires access control and a contract that implements the access control mechanism. The AccessControlContract is an abstract contract (abstract class), and any contract that requires access control can use it by inheriting from it.

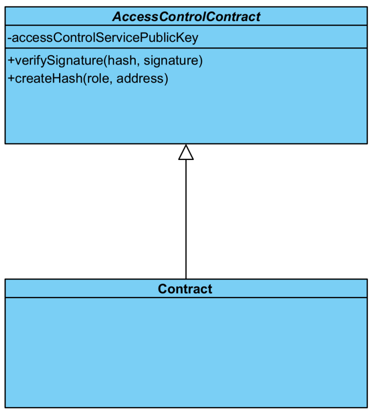

AccessControlContract
-accessControlServicePublicKey
+verifySignature(hash, signature)
+createHash(role, address)
Contract

Figure 4.5: Conceptual UML Class Diagram of the Smart Contract Component

A function that is protected by the access control requires the authorization token. The token validation has two steps.

The first step is similar to creating the token in the ACS. The system creates token data from a string containing the function's required role and account address of the function caller (sender of the transaction). Then the Token Data are hashed the same as when creating the token.

In the second step, the system verifies the token's validity. The token contains signed hashed Token Data. The signature is cryptographically verified using reference hash from the first step and the public key of the ACS (ps_{ACS}).

If the token is valid, the function execution can continue.

4.3.4 Client App

The Client Application is a user-facing part of the system. Together with the Access Control Service and the Identity Wallet, it participates in granting access and issuing

$$grantAccess = SigVerify_{pk_{ACS}}(Hash(role \cdot address_{sender}), token)$$

Figure 4.6: Authorization Token Verification

credentials. During the Credentials Disclosure Flow, it is responsible for displaying disclosure requests embedded in QR codes that the wallet can scan.

After successful authorization, the Client App can use an access token to invoke functions protected by access control in the Smart Contracts.

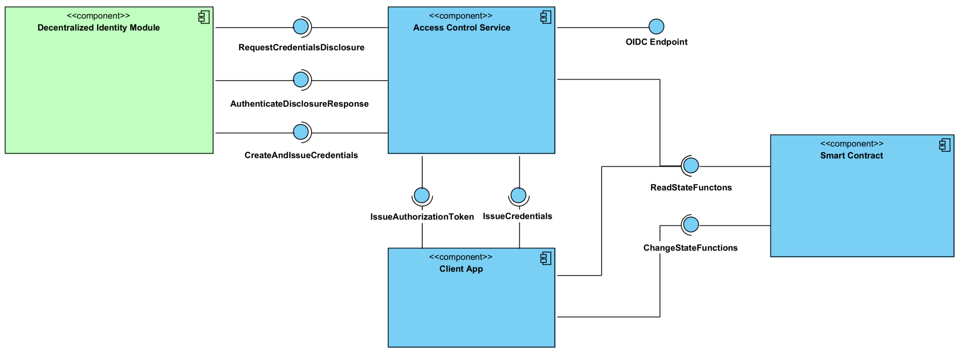


Figure 4.7: UML Component Diagram

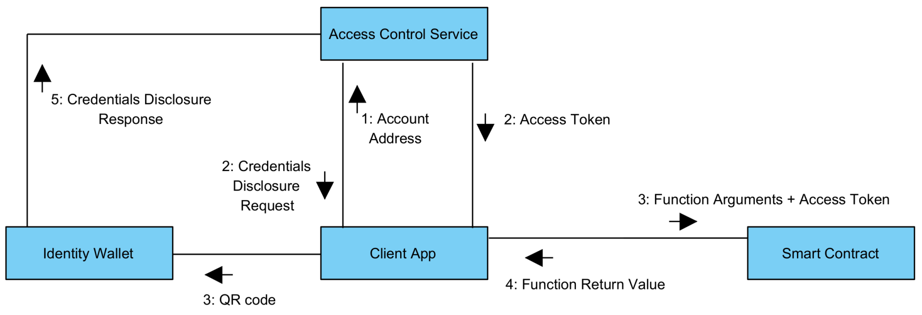


Figure 4.8: UML Communication Diagram Representing System's Data Flow During Authorization Flow

4.4 Authorization Flow

1. The Client App requests authorization from the Access Control Service.
2. The ACS starts the credentials disclosure flow by creating and sending a Credentials Disclosure Request in the form of a QR code.
3. The Client App displays the QR code so it could be scanned.
4. The user scans the QR code using the Identity Wallet.
5. The Identity Wallet prompts the user to authenticate and disclose requested credentials.

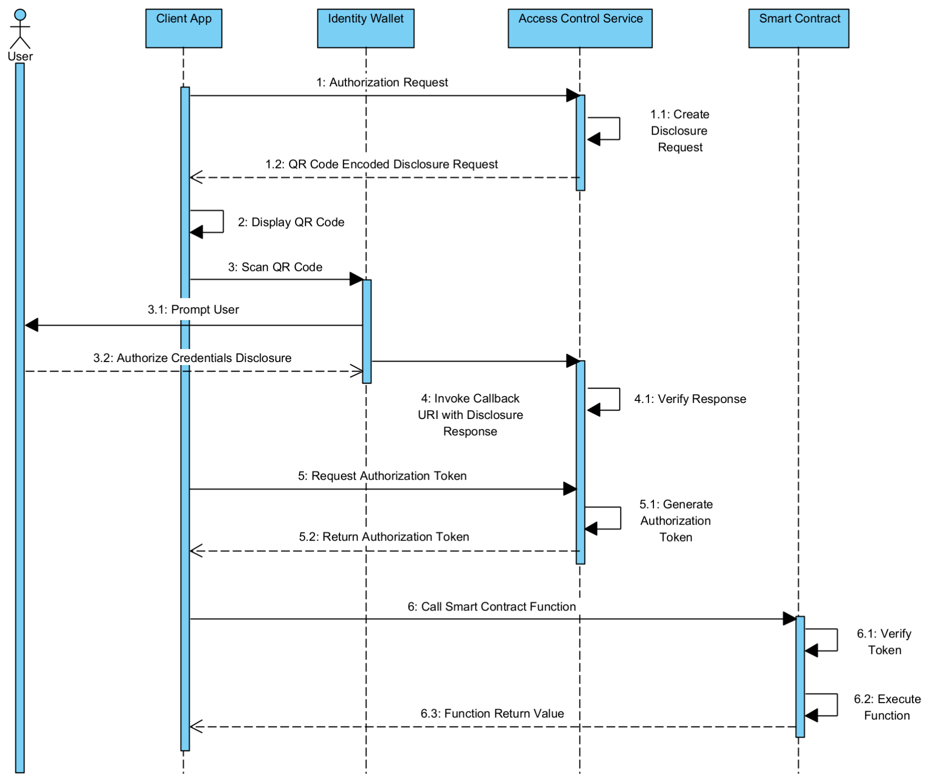

User
Client App
Identity Wallet
Access Control Service
Smart Contract
1: Authorization Request
1.1: Create Disclosure Request
1.2: QR Code Encoded Disclosure Request
2: Display QR Code
3: Scan QR Code
3.1: Prompt User
3.2: Authorize Credentials Disclosure
4: Invoke Callback URI with Disclosure Response
4.1: Verify Response
5: Request Authorization Token
5.1: Generate Authorization Token
5.2: Return Authorization Token
6: Call Smart Contract Function
6.1: Verify Token
6.2: Execute Function
6.3: Function Return Value

Figure 4.9: UML Sequence Diagram Showing Detailed Authorization Flow

6. The user accepts the credentials disclosure.
7. The Identity Wallet sends Credentials Disclosure Response to the ACS by invoking callback URI.
8. The ACS verifies the response, and if everything is valid, it will authorize the user.
9. Now the Client App can request the Authorization (Access) Token.
10. The ACS generates the token if the user is authorized and sends it back.
11. The Client App can now call the Smart Contract's functions protected by access control by providing the Access Token.

4.5 Credentials Issuance Flow

1. The user requests the credential to be issued.
2. Optionally the Client App can verify if the user is entitled to the credentials by validating the Smart Contract state.
3. The Client App requests authorization from the Access Control Service.

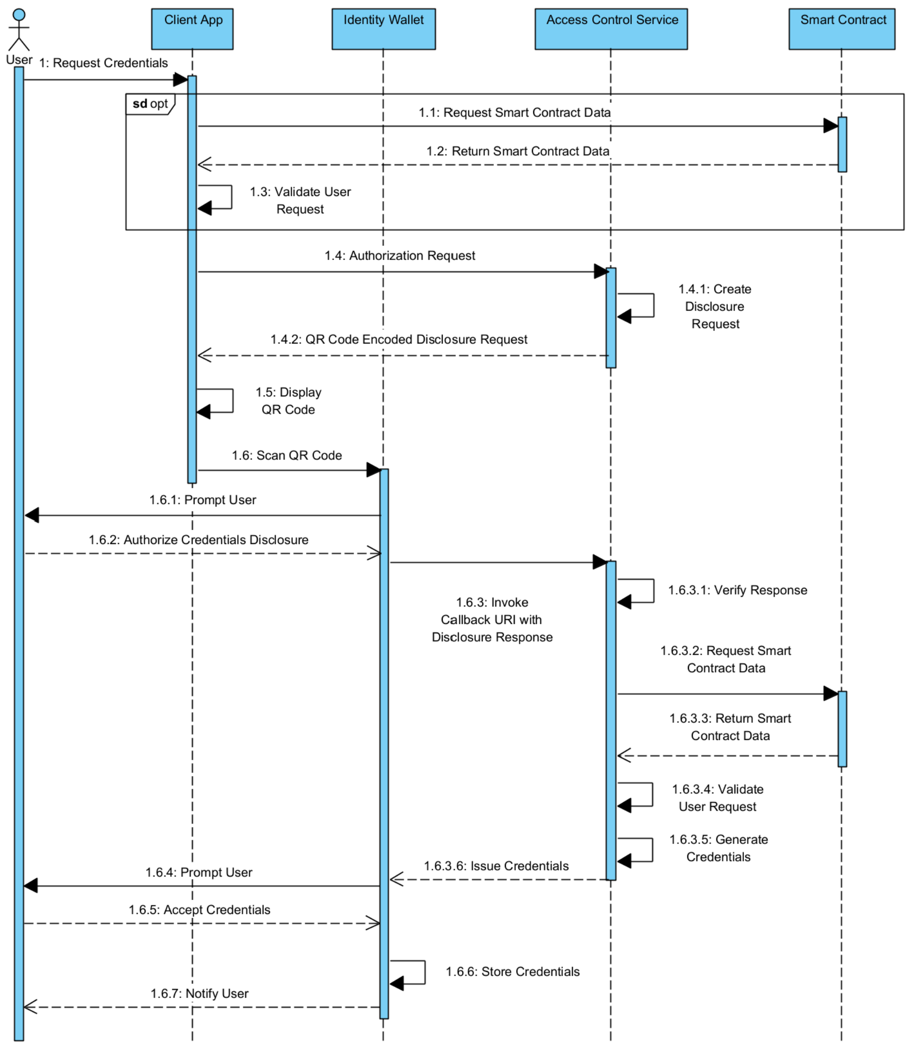

User
Client App
Identity Wallet
Access Control Service
Smart Contract
1: Request Credentials
sd opt
1.1: Request Smart Contract Data
1.2: Return Smart Contract Data
1.3: Validate User Request
1.4: Authorization Request
1.4.1: Create Disclosure Request
1.4.2: QR Code Encoded Disclosure Request
1.5: Display QR Code
1.6: Scan QR Code
1.6.1: Prompt User
1.6.2: Authorize Credentials Disclosure
1.6.3: Invoke Callback URI with Disclosure Response
1.6.3.1: Verify Response
1.6.3.2: Request Smart Contract Data
1.6.3.3: Return Smart Contract Data
1.6.3.4: Validate User Request
1.6.3.5: Generate Credentials
1.6.3.6: Issue Credentials
1.6.4: Prompt User
1.6.5: Accept Credentials
1.6.6: Store Credentials
1.6.7: Notify User

Figure 4.10: UML Sequence Diagram Showing Detailed Credentials Issuance Flow

4. The ACS starts the credentials disclosure flow by creating and sending a Credentials Disclosure Request in the form of a QR code.
5. The Client App displays the QR code so it could be scanned.
6. The user scans the QR code using the Identity Wallet.
7. The Identity Wallet prompts the user to authenticate and possibly disclose requested credentials.
8. The user accepts the credentials disclosure.
9. The Identity Wallet sends Credentials Disclosure Response to the ACS by invoking callback URI.

10. The ACS verifies the response, and if everything is valid, it will read the Smart Contract state.
11. Based on the received data, the ACS decides whether the user is entitled to the requested credentials.
12. If he is, it will generate the credentials and issues them to him via the Identity Wallet.
13. The Identity Wallet prompts the user to accept issued credentials.
14. The user accepts, and the Identity Wallet stores the credentials.
15. In the end, the user is notified about the successful issuance process.

4.6 OpenID Connect Compatibility

This proposed architecture allows for using the access control service as an OpenID Connect Identity Provider.

Elemental OIDC flow steps:

1. Send an authentication request to an identity provider.
2. Create authentication and authorization form from requested scopes and send it to an end-user.
3. User Authenticates himself and authorized requested scopes.
4. Identity provider invokes callback URI with an access token.
5. Relying party exchanges access token for ID token.

In steps two and three, it is possible to use DID authentication instead of a traditional username and password as a drop-in replacement.

Instead of filling in username and password, a user would scan the presented QR code with his identity wallet. He would authorize login via the wallet.

From the implementation perspective, login flow would be based on DID authentication.

The authentication request would trigger DID disclosure flow:

1. Create a disclosure request and present it to a user as a QR code.
2. After the QR code is scanned, the identity wallet prompts the user to accept the login request.
3. When the user accepts the wallet, invokes access control service to pass a disclosure response.
4. If valid, the access control service (as the identity provider) would generate an access code and subsequently invoke the OIDC redirect URI callback to continue the OIDC authentication flow.

Detailed flow is shown in provided UML sequence diagrams (Figures 4.11 and 4.12).

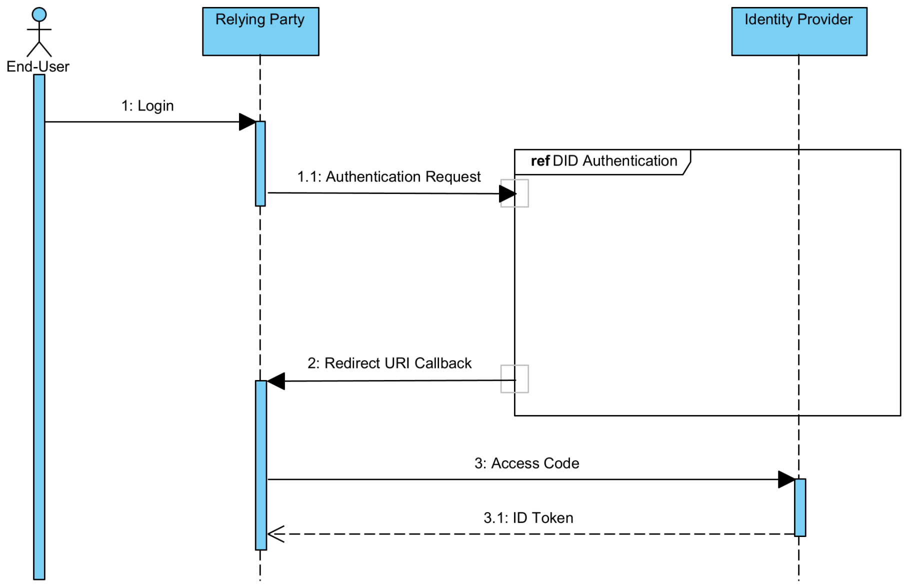


Figure 4.11: General OIDC Authentication Flow

4.7 Chapter Summary

This chapter proposed the architecture of the decentralized application system, containing role-based access control based on the decentralized identity ecosystem.

The system has two primary capabilities. The first is smart contract access control based on DIDs and VCs. The second is issuing credentials according to a smart contract state.

The system consists of four components: Smart Contracts, Access Control Service, Client Application, and Decentralized Identity Module. This chapter provided a detailed explanation of the interaction between these components.

The most significant interaction is the Credentials Disclosure Flow, which describes the process of requesting, disclosing, and verifying credentials.

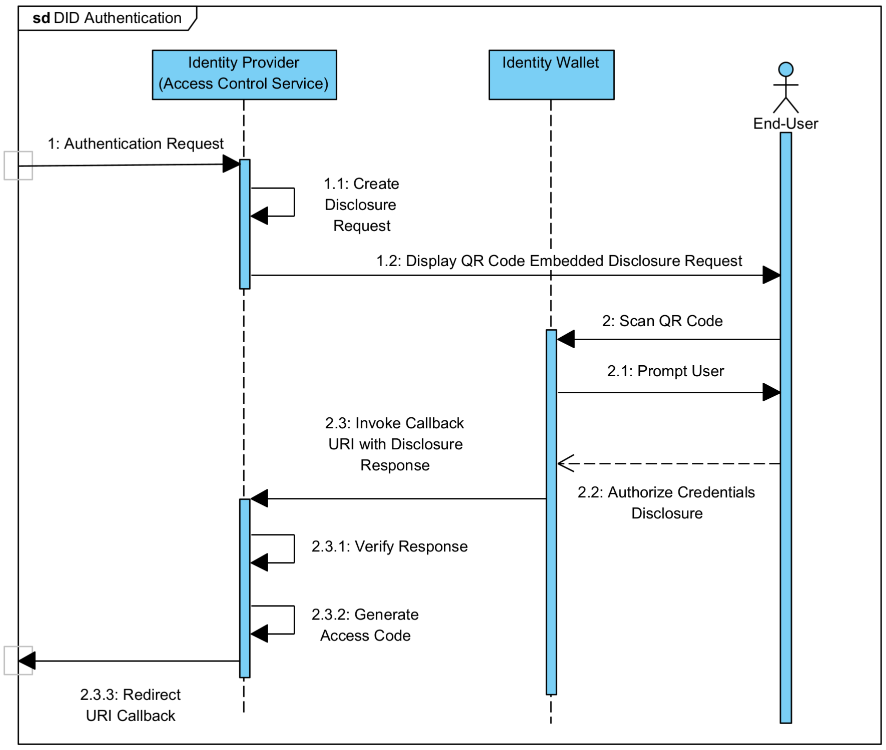

sd DID Authentication
Identity Provider (Access Control Service)
Identity Wallet
End-User
1: Authentication Request
1.1: Create Disclosure Request
1.2: Display QR Code Embedded Disclosure Request
2: Scan QR Code
2.1: Prompt User
2.2: Authorize Credentials Disclosure
2.3: Invoke Callback URI with Disclosure Response
2.3.1: Verify Response
2.3.2: Generate Access Code
2.3.3: Redirect URI Callback

Figure 4.12: Using DID Authentication and VCs for OIDC Authentication

Chapter 5

Proof of Concept Case Study

The last chapter deals with implementing a proof of concept case study based on the architecture proposed in the previous chapter. The case study involves Elections to the European Parliament and the application of decentralized identity.

5.1 Case Study

5.1.1 EU Elections Case Study

This case study builds on EU elections case study from the paper Process Digitalization using Blockchain – EU Parliament Elections Case Study [13].

Elections to the European Parliament take place every five years, and more than 400 million eligible voters from 27 member states can take part. The European Parliament is the only EU institution elected directly. Therefore the voting process must be accurate, authenticated, transparent, meddle-proof, and verifiable. [13]

The decentralized identity could provide a robust and confidential authentication mechanism suitable for elections.

This case study shows how the proposed access control architecture can be used for voter authentication and elected candidates' privileges assigning during the EU election process modeled using the DasContract.

Proof of concept implementation based on the EU election smart contract generated by the DasContract is part of the case study as well.

5.1.2 Case Study Analysis

In the DasContract model, there are three significant roles: Political party, Candidate, and EU Citizen (eligible voter). The candidate and the voter were chosen as perfect examples for using the decentralized identity.

The original model assumed that eligible EU citizens are already registered somehow, and their account addresses used for voting are already known. Therefore the model was updated to accommodate voter registration for eligible EU citizens. In this new model, EU citizens have to register themselves as voters first before they are allowed to vote. This change is shown in the updated diagram in 5.2.

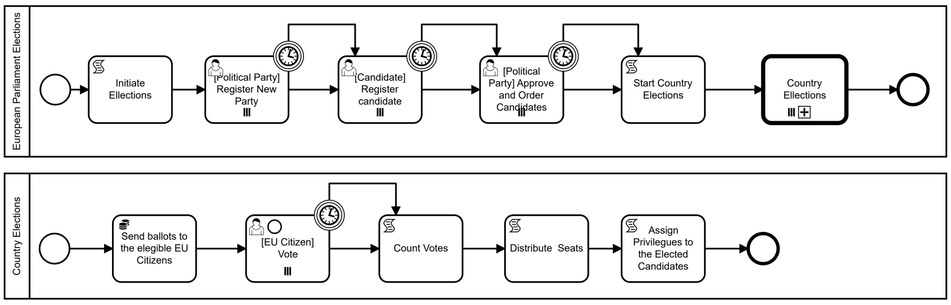

European Parliament Elections
Initiate Ellections
[Political Party] Register New Party
[Candidate] Register candidate
[Political Party] Approve and Order Candidates
Start Country Elections
Country Elections
Country Elections
Send ballots to the elegible EU Citizens
[EU Citizen] Vote
Count Votes
Distribute Seats
Assign Privilegues to the Elected Candidates

Figure 5.1: DasContract Process Model of the EU Elections [13]

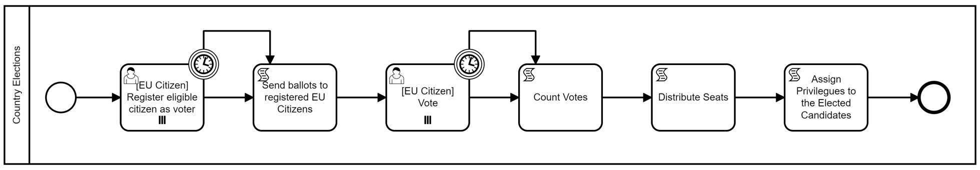

Country Elections
[EU Citizen] Register eligible citizen as voter
Send ballots to registered EU Citizens
[EU Citizen] Vote
Count Votes
Distribute Seats
Assign Privilegues to the Elected Candidates

Figure 5.2: Updated Process Model of the EU Elections

Primary use cases:

- Register Voter (Voter role)
- Vote (Voter role)
- Receive Assigned Mandate (Candidate role)

Only eligible EU citizens can be registered. This is where access control comes in. The Access Control Service grants voter access only to verified users. Verification consists of verifying users' EU member state citizenship and minimum legal age. That is an example of the application of the authorization flow defined in the previous chapter.

Issuing mandate privileges to the elected candidates is another use case that greatly demonstrates the possibilities of using verifiable credentials. The system reads the election results from the elections smart contract. And based on the results, it assigns privileges to the elected candidate by issuing verified credentials. That is an example of the application of the credentials issuance flow defined in the previous chapter.

5.2 Technology Stack

5.2.1 Solidity

Solidity is an object-oriented, high-level language for implementing smart contracts. [92]

The EU elections contract designed in the DasContract language is generated as solidity code. The solidity 0.8.1 compiler was used to generate EVM executable code.

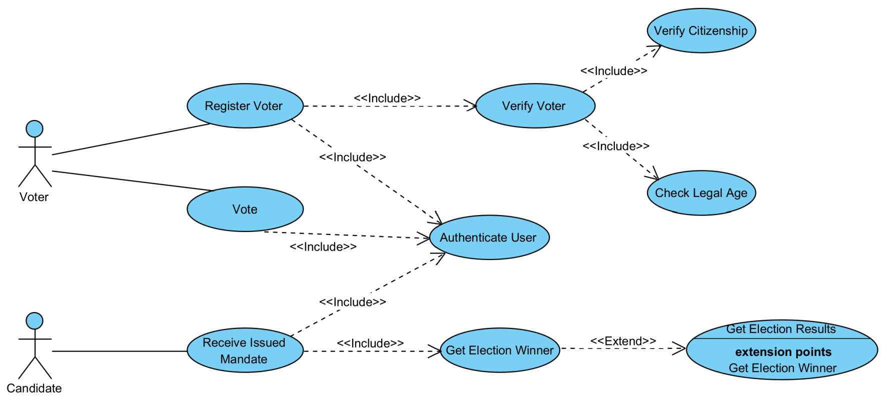


Figure 5.3: UML Use Case Diagram of the EU Elections Case Study

5.2.2 Ganache

Ganache [93] is a personal blockchain. It provides a GUI client for creating and managing local blockchain networks for DApp development and testing. [94]

The Ganache was used for creating testing an Ethereum network, to which smart contracts were deployed.

5.2.3 MetaMask

MetaMask [95] is an Ethereum Wallet in the form of a web browser extension. It connects to an Ethereum network, manages the user's accounts, and allows him to send and receive transactions, exchange tokens, and sign messages. It also provides an Ethereum connection for web 3.0 applications. [96]

The MetaMask was used to manage test accounts and provide an Ethereum blockchain connection for the Client Application.

5.2.4 Remix IDE

Remix IDE [97] is an open-source IDE for smart contract development. It allows for smart contract programming, debugging, and deployment. [98]

It was used for compiling, deploying, and testing smart contracts in the case study.

5.2.5 uPort

Company uPort (now called Serto) provides the Decentralized Identity Module for the system. The uPort offering includes the uPort mobile app as the Identity Wallet and a collection of DID-compliant libraries. [99]

There are only a few complete solutions like this one, and the uPort was chosen because of its mature and tested implementation and very usable app.

5.2.6 Node.js

Node.js is an asynchronous event-driven JavaScript runtime. It is designed to build scalable network applications. [100]

Node.js v14.16.0 was used as a runtime environment for the Access Control Service.

5.2.7 Express.js

Express.js is a minimal Node.js web application framework. [101]

It provides HTTP API of the Access Control Service.

5.2.8 ngrok

The ngrok [102] is a web tunneling service. It exposes a locally running webserver to the internet. [103]

The ngrok is required because the Identity Wallet communicates with the Access Control Service through the internet, so the ACS needs to be accessible from the internet.

5.2.9 Bootstrap

Bootstrap is a web UI framework that allows for fast and easy websites designing. [104]

Bootstrap v5 was used for creating the Client App's UI.

5.2.10 OpenZeppelin

OpenZeppelin is an open-source project that provides a library for secure smart contract development. The library contains a set of solidity smart contracts that can be used for smart contract development. [105]

The EU elections contract is using some of the contracts provided by the OpenZeppelin.

5.2.11 ethers.js

The ether.js is an open-source complete and compact library for interacting with the Ethereum Blockchain and its ecosystem. [106]

In the system, it is used by the Access Control Service and the Client App for signing, verifying signatures, and interacting with smart contracts deployed in the blockchain.

5.2.12 INFURA

The INFURA development suite provides instant, scalable API access to the Ethereum and IPFS networks. [107]

Their HTTP API is used by the Access Control Service's uPort library to access the Ethereum network for DID resolution.

5.3 Implementation

5.3.1 Smart Contract

The Smart Contract component is implemented using the Solidity programming language.

It consists of three contracts:

- ElectionContract
- VotingToken
- AccessControlContract

The election contract and voting token implement a simplified EU election process. They were generated by the DasContract compiler. However, a few changes were necessary for them to be compiled by the current solidity compiler. They also use two contracts provided by the OpenZeppelin library.

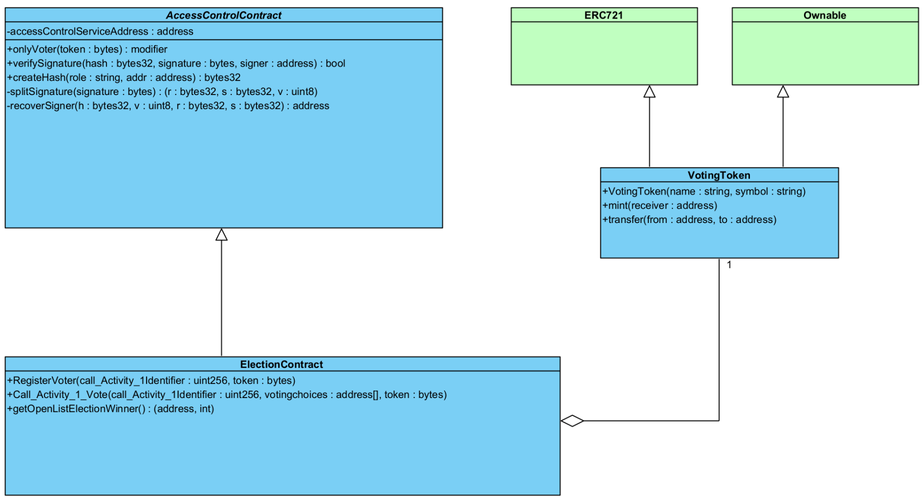

AccessControlContract
-accessControlServiceAddress : address
+onlyVoter(token : bytes) : modifier
+verifySignature(hash : bytes32, signature : bytes, signer : address) : bool
+createHash(role : string, addr : address) : bytes32
-splitSignature(signature : bytes) : (r : bytes32, s : bytes32, v : uint8)
-recoverSigner(h : bytes32, v : uint8, r : bytes32, s : bytes32) : address
ERC721
Ownable
VotingToken
+VotingToken(name : string, symbol : string)
+mint(receiver : address)
+transfer(from : address, to : address)
1
ElectionContract
+RegisterVoter(call_Activity_1Identifier : uint256, token : bytes)
+Call_Activity_1_Vote(call_Activity_1Identifier : uint256, votingchoices : address[], token : bytes)
+getOpenListElectionWinner() : (address, int)

Figure 5.4: Simplified UML Class Diagram of the Smart Contract Component

Because the original election contract does not contain access control, the contract was changed according to the updated process model (see figure 5.2) to emulate the DasContract code generation. This was necessary due to the lack of a usable DasContract editor at the time of the writing.

These changes include adding an access control modifier to the vote function (`Call_Activity_1_Vote`), implementing a whole new function for registering voters (`RegisterVoter`), implementing necessary utility functions, and fixing the control flow.

The original contract did not contain any code for election result aggregation, so a simplified function for finding the election winner was added (`getOpenListElectionWinner`).

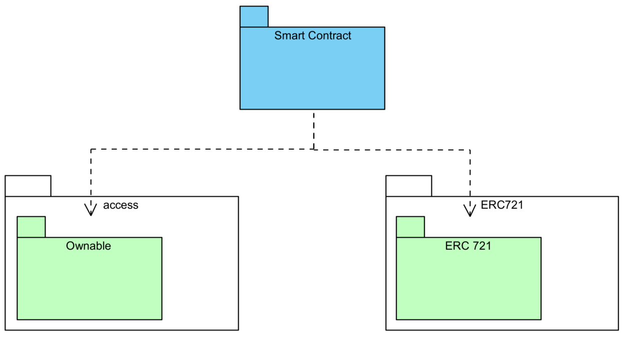


Figure 5.5: UML Package Diagram of the Smart Contract Component

The `AccessControlContract` provides access control for the election contract. The `onlyVoter` modifier is used to restrict access to `Call_Activity_1_Vote`, and `RegisterVoter` functions only for verified users (eligible voters that the Access Control Service issued the access token to). Modifiers in solidity serve as function decorators. They can provide additional functionality a function before or after the function is invoked.

The modifier cryptographically verifies whether the `"VOTER"` role was assigned to the account calling the function. Verification is possible by the `ecrecover` function that can recover the address of a signer account from a signature and original message. In this case, it uses provided signature (the access token) and recreated token data to check if the Access Control Service assigned a voter role to the account.

```
function RegisterVoter(
    uint256 call_Activity_1Identifier,
    bytes memory token)
    isRegisterVotersState()
    onlyVoter(token) public
```

Listing 5: Header of the Voting Contract's RegisterVoter Method Showing onlyVoter Modifier Application

```
contract AccessControlContract {

    modifier onlyVoter(bytes memory token) {
        bytes32 hashedRole = createHash("VOTER", msg.sender);
        bool isValid =
            verifySignature(
                hashedRole,
                token,
                accessControlServiceAddress);

        require(isValid, "INVALID ACCESS");
        _;
    }

    function verifySignature(
        bytes32 hash,
        bytes memory signature,
        address signer) internal pure returns (bool)
    {
        (bytes32 r, bytes32 s, uint8 v) = splitSignature(signature);
        address addressFromSignature = recoverSigner(hash, v, r, s);
        return addressFromSignature == signer;
    }

    function recoverSigner(bytes32 h, uint8 v, bytes32 r, bytes32 s)
        internal pure returns (address)
    {
        bytes memory prefix = "\x19Ethereum Signed Message:\n32";
        bytes32 prefixedHash = keccak256(abi.encodePacked(prefix, h));
        address addr = ecrecover(prefixedHash, v, r, s);

        return addr;
    }
}
```

Listing 6: Core Part of the AccessControlContract

5.3.2 Access Control Service

The Access Control Service is implemented in the JavaScript language as a Node.js web service using the Express.js framework. It uses ethers.js to access the blockchain. It also uses uPort libraries to implement credentials disclosure flow. Because it needs to be accessible from the internet for the uPort app, it requires the ngrok tunneling service to operate.

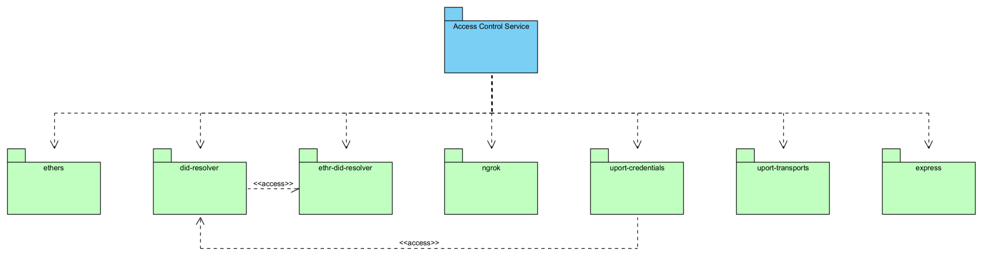


Figure 5.6: UML Package Diagram of the Access Control Service

The access control service provides API for requesting access to the voting contract by the EU citizens eligible to vote and for requesting issued mandate credentials by elected candidates.

APIs for these use cases are built on the credentials disclosure flow and consist of pairs of endpoints. From the pair, one endpoint is for disclosure requests, and the other one serves for disclosure response callbacks. The pair of endpoints follow such convention that the response endpoint has the same path as the request endpoint but with added `/callback` at the end (e.g. `/endpoint` and `/endpoint/callback`).

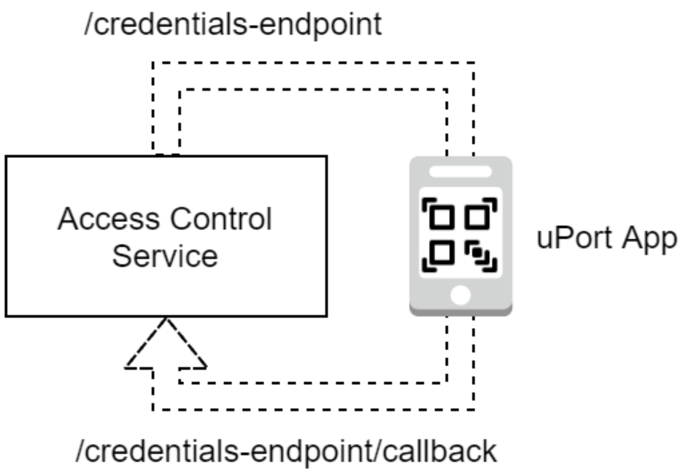


Figure 5.7: Access Control Service Credentials Disclosure API Scheme

Main API:

/verifycredentials - for credentials disclosure flow to verify voters

/token - for requesting voter authorization token after verification

/getmandate - for credential disclosure flow to request issued mandate credentials

Utility API:

/setcontractaddress - to set address of the deployed Smart Contracts

/getcredentials - for credential disclosure flow to request testing citizenship credentials

During voter verification, the credentials disclosure request is created by the createDisclosureRequest method of the Credentials object from the uPort library. It contains a request for the Citizenship ID credentials required by the voter role.

The voter Ethereum address is provided via URL parameter in the request to the credentials disclosure request endpoint.

After the receiving disclosure response, it is verified by the uPort library. Then the service grants access if the user disclosed Citizenship ID credentials and is old enough to vote. The proof of legal age would be a perfect application of ZKP. However, the uPort does not support ZKP or synthesized credentials, so a user must disclose the date of birth.

```
function validateVoterCredentials(credentials) {
    const citizenIdCredential =
        credentials[roles.VOTER.requirements.credential];
    if (!citizenIdCredential) {
        return false;
    }
    const dateOfBirth = citizenIdCredential.dateOfBirth;
    const age = calculateAge(new Date(dateOfBirth));
    return age >= roles.VOTER.requirements.minimumLegalAge;
}
```

Listing 7: Validation of Citizen ID Credentials

```
credentials.createDisclosureRequest({
        verified: [roles.VOTER.requirements.credential],
        notifications: true,
        callbackUrl: endpoint
            + `/verifycredentials/callback?address=${address}`
});
```

Listing 8: Creating Credentials Disclosure Request

The Authorization token is created by signing hashed token data. The hash algorithm used is the Keccak256.

```
function createHash(role, address) {
    const roleBytes = ethers.utils.toUtf8Bytes(role);
    const addrresBytes = ethers.utils.arrayify(address);
    const data = new Uint8Array([...roleBytes, ...addrresBytes]);
    const hash = ethers.utils.keccak256(data);
    return hash;
}

function makeToken(wallet, hash) {
    return wallet.signMessage(ethers.utils.arrayify(hash));
}
```

Listing 9: Access Token Generation

During mandate credentials issuing, the service verifies that the user is entitled to the credentials based on the election results before it issues the credentials. The createVerification method of the Credentials object is used to create verifiable credentials by providing a plain JavaScript object containing the claims. Then the uPort transport library is used to issue the credentials via the uPort mobile app.

```
function createMandateCredentials(address) {
    return {
        'EU Parliament Mandate': {
            id: 908765439876,
            length: '4 years',
            address: address
        }
    }
}

credentials.createVerification(
    {
        sub: creds.did,
        exp: Math.floor(new Date().getTime() / 1000) + 126227704,
        claim: createMandateCredentials(address),
    }
);
```

Listing 10: Creation of the Mandate Credentials

For testing purposes, the Access Control Service also issues the Citizenship ID credentials. By using the URL parameter isValid, the system can generate credentials of the citizen not eligible to vote (a minor).

5.3.3 Client Application

The Client Application is a simple web application build using HTML, CSS, and JavaScript. For convenience, it is served by the Access Control Service, but it can be served from any web server or CDN provider. Essentially it is a simple web 3.0 application as it uses the MetaMask extension to access an Ethereum blockchain.

It uses the ethers.js library for interaction with smart contracts and message signing. Its UI is built using the Bootstrap framework. It is designed for the voter and candidate role users to test interaction with the system.

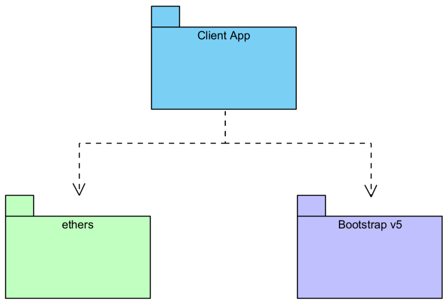


Figure 5.8: UML Package Diagram of the Client Application Component

It provides UI for requesting access to voting and for requesting issued mandate credentials. Its primary purpose is to display credentials disclosure requests embedded in QR codes.

5.4 Deployment and Testing

The system was built and deployed in the local environment to test the capabilities of the proposed architecture.

The Access Control Service was deployed using the node.js runtime environment.

The web application of the Client App ran inside a web browser and was hosted by the AccessControlService. It was done to avoid potential problems with CORS during testing in the local environment.

The Smart Contracts component containing the election contract and the access control contract was compiled and deployed to a testing Ethereum blockchain by the Remix IDE.

The testing Ethereum network was set up using Ganache. Initial testing was unsuccessful due to failing transactions. After some investigation, the cause was discovered - the low gas limit of the network. That led to setting up a new network with a much higher gas limit, which solved the problem.

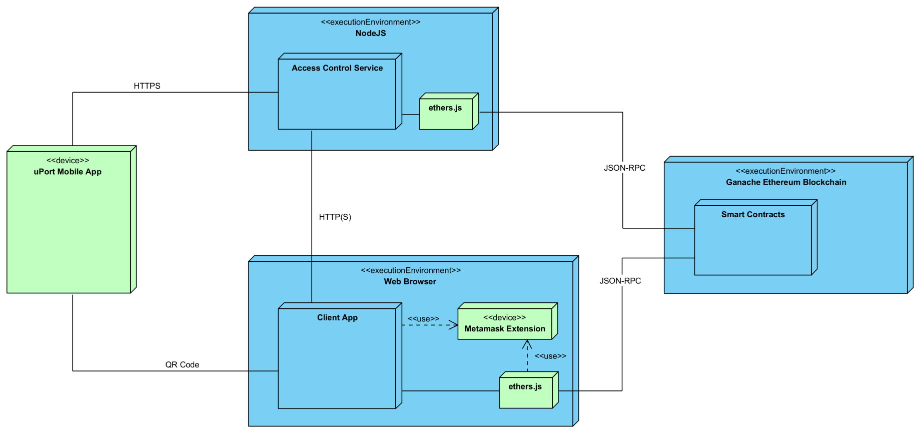


Figure 5.9: UML Deployment Diagram of the System

The Ganache also generated user accounts in the network. The accounts were used during testing to simulate different users or entities interacting with the Smart Contracts. Accounts were designated as `VOTER`, `PARTY`, `CANDIDATE`, and `OWNER`.

VOTER - an EU Citizen

PARTY - a political party participating in the elections

CANDIDATE - a candidate of the political party

OWNER - a owner of the election smart contract

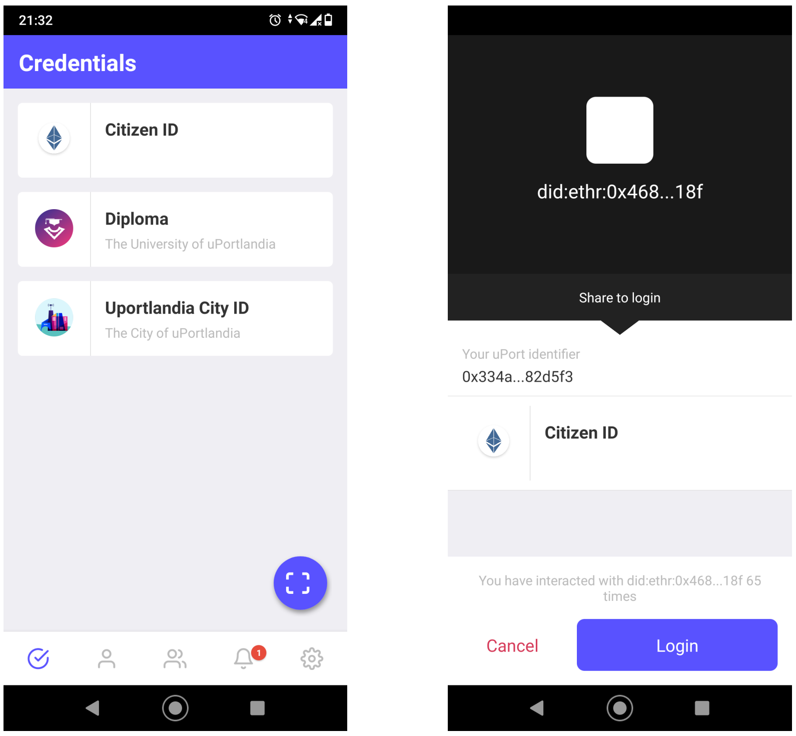
To manage and use these accounts, the MetaMask Ethereum wallet in the form of a web browser extension was used. The MetaMask allows us to interact with the web application (the Client App or Remix IDE) on behalf of the active (connected) account. Switching between the accounts is straightforward when using it. The MetaMask also provided a connection to the testing blockchain for the Client App and the Remix.

The entire election process of the election smart contract was simulated, but only voter registration, voting, and reading election results are done by the Client App. All other interactions were simulated using the interface provided by the Remix.

During testing the Client App, the first step is setup. A user has to connect the MetaMask and set the Smart Contract address. In reality, the address would be known beforehand, but the address keeps changing during the testing, so it requires a manual setting.

Next, the user can request Citizen ID credentials for the testing to be issued to his uPort mobile app. The system can issue valid (citizen is eligible for vote) or invalid (citizen is not eligible to vote due to an insufficient age) credentials.

Then comes the eligible citizen verification. By clicking the Verify Citizenship Credentials, the user starts the Citizenship credentials disclosure flow. After disclosing credentials, the user can get the result by clicking the Get Token button. If the Access Control Service finished the disclosure flow and verified an eligible voter, the Client App displays the Access token. Otherwise, it displays an error.



(a) Received Credentials List

(b) Citizen ID Credential Disclosure Request

Figure 5.10: uPort Identity Wallet Application

With the Access token, the user can register as a voter to the election smart contract and later vote for the selected candidate. That is, however, conditioned by the current state of the smart contract. These activities can only be done if the smart contract is in the corresponding state.

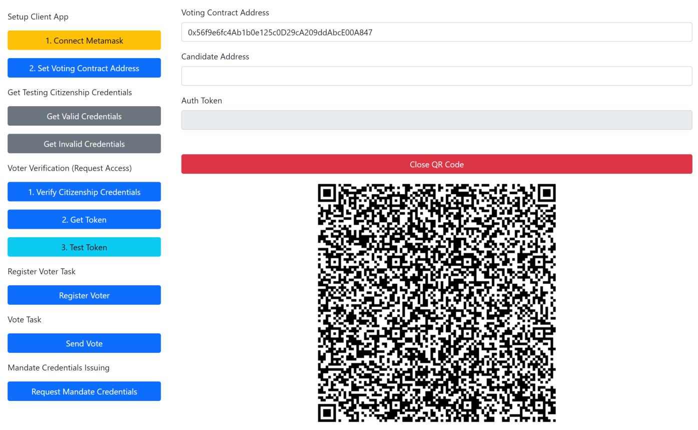
When the election process ends, the user can use the `CANDIDATE` account to request mandate credentials. To verify that the provided account is controlled by its actual owner, the Access Control Service requires the user to authenticate the account by signing some arbitrary message.

5.5 Discussion and Future Development

The proof of concept implementation meets the expectation. It provides access control to smart contracts, and it can issue credentials to the elected candidates exactly as expected from the proposed architecture.

Still, some parts can be improved. The Open ID Connect provider proposed in the architecture was not part of the implementation because it was not needed for the election case study proof of concept. Nevertheless, it could be implemented additionally in the future.

Currently, the token request after credential verification has to be sent manually, but it could be automated.


Setup Client App
1. Connect Metamask
2. Set Voting Contract Address
Get Testing Citizenship Credentials
Get Valid Credentials
Get Invalid Credentials
Voter Verification (Request Access)
1. Verify Citizenship Credentials
2. Get Token
3. Test Token
Register Voter Task
Register Voter
Vote Task
Send Vote
Mandate Credentials Issuing
Request Mandate Credentials
Close QR Code

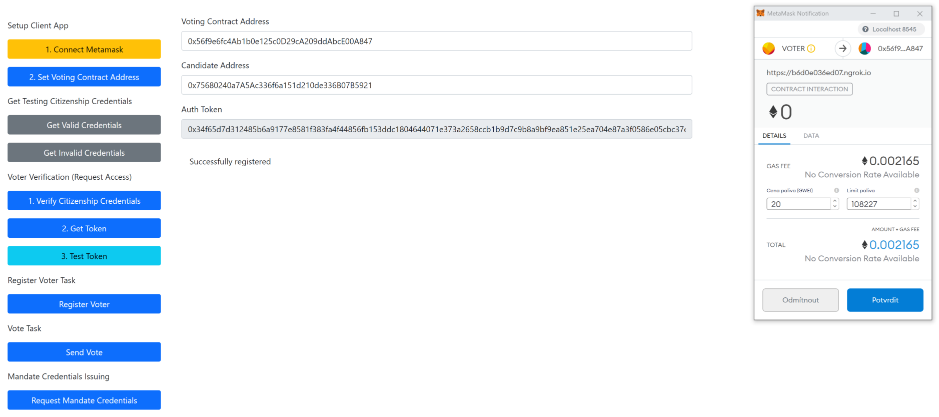
Voting Contract Address

0x56f9e6fc4Ab1b0e125c0D29cA209ddAbcE00A847

Candidate Address

Auth Token

Figure 5.11: The Client App with Displayed QR Code with Credentials Disclosure Request



Voting Contract Address

0x56f9e6fc4Ab1b0e125c0D29cA209ddAbcE00A847

Candidate Address

0x75680240a7A5Ac336f6a151d210de336B07B5921

Auth Token

0x34f65d7d312485b6a9177e8581f383fa4f44856fb153ddc1804644071e373a2658ccb1b9d7c9b8a9bf9ea851e25ea704e87a3f0586e05cbc37

Successfully registered

Localhost 8545

VOTER → 0x56f9...A847

https://b6d0e036ed07.ngrok.io

CONTRACT INTERACTION

0

GAS FEE 0.002165
No Conversion Rate Available

Cena paliva (GWEI) 20

Limit paliva 108227

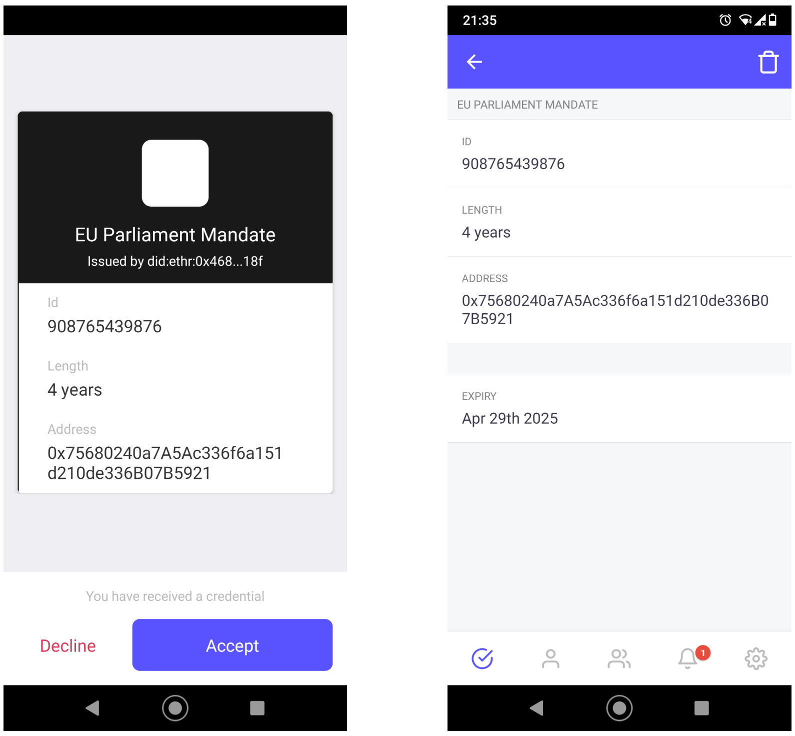
AMOUNT + GAS FEE

TOTAL 0.002165
No Conversion Rate Available

Figure 5.12: The Client App with MetaMask Extension in Process of Calling the Vote Function

Also, when using this indirect access control method, the address of the account a user chooses for smart contract interaction is part of the authorization process. Even though the DID is a universal identifier, the address from some particular blockchain is still required. That is probably unavoidable, but it is still required to find a way of proving that a user authenticated using DID is the owner of an account he provided.

In this proof of concept, a user can create an access token for another user if he avoids the Client App and uses the ACS API directly. One possible solution would be using challenge-response authentication (signing an arbitrary message and letting the ACS verify it), just like it is used now to authenticate elected candidates before issuing mandate credentials.


EU Parliament Mandate
Issued by did:ethr:0x468...18f

Id
908765439876

Length
4 years

Address
0x75680240a7A5Ac336f6a151d210de336B07B5921

You have received a credential

Decline | Accept

21:35

EU PARLIAMENT MANDATE

ID
908765439876

LENGTH
4 years

ADDRESS
0x75680240a7A5Ac336f6a151d210de336B07B5921

EXPIRY
Apr 29th 2025

(a) EU Parliament Mandate Credentials Issuance

(b) Issued EU Parliament Mandate Credentials Detail

Figure 5.13: uPort Identity Wallet Application

Another and possibly better solution is to get an address from verifiable credentials. For example, a government could issue verifiable credentials containing the verified blockchain address of a citizen for official occasions (e.g., elections).

Finding other solutions for the problem of tying the blockchain account to DID-based identity may be the subject of further research.

Apart from this security concern, there is also an issue with the current DID ecosystem. As it is bleeding-edge technology, it is still very actively evolving and changing. Many standards are still in the phase of a draft or a proposal, which means they are likely to change. Also, there are not many libraries or SDKs that implement DID standards. And those which do, provide mostly low-level API, which is not very stable.

Due to this, it is almost certain that the decentralized identity module will require a technological update if it should work in the future as well.

Even though the indirect access control should prevent unnecessary data storing in a smart contract, voters' addresses are still stored inside the election contract due to the contract's design. However, when used with a different smart contract, the access control service is able to replace blockchain address-based access lists and therefore prevent address storing inside the contract.

Furthermore, the system shares many disadvantages with smart contracts due to them being part of the system. That includes cumbersome development and testing, high cost (if deployed to a public blockchain), or a block gas limit.

The main contribution of the proof of concept is that the proposed architecture is usable. The architecture was designed to be flexible and platform-independent so that it could serve at least as a framework or an inspiration for other similar solutions.

Currently, using the decentralized identity for production use, especially with sensitive data, is not advised. Nevertheless, with an active community and corporate backing, the decentralized identity ecosystem will grow and mature. Moreover, in the future, with available mature DID implementations and production-ready blockchain, the decentralized identity will surely be viable authentication and authorization alternative to the current methods.

5.6 Chapter Summary

At the beginning of the chapter, the EU election case study was analyzed to find the possible application of the proposed access control system architecture.

The next part consisted of a description of technologies used for implementation.

The following part contained a description of the Access Control Service, Smart Contracts, and the Client App components implementation, including dependencies.

The last part described how the system is deployed and tested, together with proof of concept discussion and recommendations for future development.

Conclusion

This thesis provided a comprehensive review of the decentralized identity, decentralized applications, and DasContract domain-specific language, including their underlying technologies like blockchain, smart contracts, and zero-knowledge proofs.

The thesis also provided an overview of the current authentication methods like OAuth, OpenID Connect, and SAML, together with a detailed comparison of these methods with the DID-based authentication method.

The third chapter of the thesis described how the decentralized identity could be used to solve the problem with authorization on public permissionless blockchains like Ethereum by providing access control. It outlined possible ways to implement role-based or attribute-based access control mechanisms. Additionally, it showed how the decentralized identity could be integrated into DasContract to specify smart contracts RBAC.

The thesis proposed the architecture of a decentralized application system containing a role-based access control mechanism based on decentralized identity. The architecture has three parts. The Smart Contracts that contain business logic that requires access control. The Access Control Service that grants access to the Smart Contracts and issues new credentials. And the Client App uses access granted by the service to interact with the contracts. Additionally, the Access Control Service can be used as an OpenID Connect provider.

The proposed architecture was demonstrated on proof of concept case study implementation. The proof of concept case study involved simplified Elections to the European Parliament process modeled in the DasContract. The decentralized identity proved itself useful for verifying eligible voters and issuing mandate credentials to elected candidates. The implementation satisfied the requirement, but a few shortcomings were found. Most of them could be solved by further development as outlined in the case study conclusion.

The proof of concept case study provided valuable insight into the application of the decentralized identity. The decentralized identity is a bleeding-edge technology that is not yet ready for production use. However, it is proved its potential, and in a few years, it could be commonly used for official communication with government institutions.

Because the decentralized identity is a new and still evolving branch of computer science, it provides many possibilities for further research.

For example:

- Try to find other solutions for the problem of tying the blockchain account to a DID-based identity.
- Test this access control method on other permissionless blockchains other than Ethereum.
- Try to integrate this access control method with permissioned blockchains.
- Experiment with other methods outlined in the third chapter, especially with the possibility of using oracles.
- Explore verifiable credentials revocation mechanism.
- Test the real-world cybersecurity of the proposed architecture.
- Explore the legal aspects of the decentralized identity application, like eIDAS or GDPR.

Bibliography

1. JOHNSON, Peggy. *Partnering for a path to digital identity - The Official Microsoft Blog* [online]. 2018 [visited on 2021-04-02]. Available from: `https://blogs.microsoft.com/blog/2018/01/22/partnering-for-a-path-to-digital-identity`.
2. ERBEN, Luk. Drn ttky a Holocaust: superobanka Jacoba Lenze. *Root* [online]. 2013 [visited on 2021-04-02]. Available from: `https://www.root.cz/clanky/derne-stitky-a-holocaust-superobcanka-jacoba-lenze`.
3. KLEIN, Dirk de. *The Civil Servants part in the Dutch Holocaust.* [Online]. 2018 [visited on 2021-04-02]. Available from: `https://dirkdeklein.net/2018/07/11/the-civil-servants-part-in-the-dutch-holocaust`.
4. VIJAYAN, Jai. 2017 Data Breach Will Cost Equifax at Least $1.38 Billion. *Dark Reading* [online]. 2020 [visited on 2021-04-02]. Available from: `https://www.darkreading.com/attacks-breaches/2017-data-breach-will-cost-equifax-at-least-%5C$138-billion-/d/d-id/1336815`.
5. GATES, Mark. *Blockchain: ultimate guide to understanding blockchain, bitcoin, cryptocurrencies, smart contracts and the future of money.* CreateSpace, 2017. ISBN 9781547090686. OCLC: 1013543660.
6. ASHARAF, S.; ADARSH, S. *Decentralized computing using blockchain technologies and smart contracts: emerging research and opportunities.* Hershey, PA: Information Science Reference, 2017. ISBN 9781522521938.
7. NAKAMOTO, Satoshi. Bitcoin: A peer-to-peer electronic cash system [online]. 2008 [visited on 2021-03-19]. Available from: `http://www.bitcoin.org/bitcoin.pdf`.

8. MODI, Ritesh. *Solidity programming essentials: a beginner's guide to build smart contracts for Ethereum and blockchain.* Birmingham Mumbai: Packt, 2018. ISBN 9781788831383.

9. IYER, Kedar; DANNEN, Chris. *Building Games with Ethereum Smart Contracts: Intermediate Projects for Solidity Developers.* 1st ed. 2018. Berkeley, CA: Apress : Imprint: Apress, 2018. ISBN 9781484234921.

10. FRANKENFIELD, Jake. *Permissioned Blockchains* [online]. 2020 [visited on 2021-04-12]. Available from: `https://www.investopedia.com/terms/p/permissioned-blockchains.asp`.

11. SKOTNICA, Marek; PERGL, Robert. Das Contract - A Visual Domain Specific Language for Modeling Blockchain Smart Contracts. In: AVEIRO, David; GUIZZARDI, Giancarlo; BORBINHA, José (eds.). *Advances in Enterprise Engineering XIII* [online]. Cham: Springer International Publishing, 2020, vol. 374, pp. 149–166 [visited on 2021-03-21]. ISBN 9783030379339. Available from DOI: `10.1007/978-3-030-37933-9_10`.

12. LY, Thon. A Complete Mental Model for Ethereum dApp Development. *Medium* [online]. 2018 [visited on 2021-03-31]. Available from: `https://medium.com/heartbankacademy/a-complete-mental-model-for-ethereum-dapp-development-5ce08598ed0a`.

13. SKOTNICA, Marek; APARÍCIO, Marta; PERGL, Robert; GUERREIRO, Sérgio. Process Digitalization using Blockchain: EU Parliament Elections Case Study: in: *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development* [online]. SCITEPRESS - Science and Technology Publications, 2021, pp. 65–75 [visited on 2021-03-21]. ISBN 9789897584879. Available from DOI: `10.5220/0010229000650075`.

14. COLLINS, Patrick. What is a blockchain oracle? What is the oracle problem? Why can't blockchains make API calls? This is everything you need to know about off-chain dat | Better Programming. *Medium* [online]. 2020 [visited on 2021-03-30]. Available from: `https://betterprogramming.pub/what-is-a-blockchain-oracle-f5ccab8dbd72`.

15. SOLIDITY BY EXAMPLE. *First Application | Solidity by Example | 0.7.6* [online]. 2021 [visited on 2021-03-19]. Available from: `https://solidity-by-example.org/first-app`.

16. FRAIT, Jan. *Generating Ethereum Smart Contracts from DasContract Language.* Prague, 2020. Available also from: `https://dspace.cvut.cz/bitstream/handle/10467/90034/F8-DP-2020-Frait-Jan-thesis.pdf`. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology.

17. SKOTNICA, Marek; KLICPERA, Jan; PERGL, Robert. Towards Model-Driven Smart Contract Systems – Code Generation and Improving Expressivity of Smart Contract Modeling. In: *Proceedings of the 20th CIAO! Doctoral Consortium, and Enterprise Engineering Working Conference Forum 2020 co-located with 10th Enterprise Engineering Working Conference (EEWC 2020)* [online]. CEUR Workshop Proceedings (CEUR-WS.org), 2020 [visited on 2021-03-21]. Available from: `http://ceur-ws.org/Vol-2825/paper1.pdf`.

18. DROZDÍK, Martin. *Open-Source Legal Process Designer in .NET Blazor*. Prague, 2020. Available also from: `https://dspace.cvut.cz/bitstream/handle/10467/88271/F8-BP-2020-Drozdik-Martin-thesis.pdf`. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology.

19. KLICPERA, Jan. A novel way of conducting legal contracts - Jan Klicpera - Medium. *Medium* [online]. 2021 [visited on 2021-03-22]. Available from: `https://medium.com/@honzaklicpera/a-novel-way-of-conducting-legal-contracts-be54ceda39ad`.

20. LÓPEZ-PINTADO, Orlenys; GARCÍA-BAÑUELOS, Luciano; DUMAS, Marlon; WEBER, Ingo; PONOMAREV, Alexander. Caterpillar: A business process execution engine on the Ethereum blockchain. *Software: Practice and Experience*. 2019, vol. 49, no. 7, pp. 1162–1193. Available from DOI: `https://doi.org/10.1002/spe.2702`.

21. ORLENYSLP. *Caterpillar* [online]. 2021 [visited on 2021-03-22]. Available from: `https://github.com/orlenyslp/Caterpillar`.

22. LANTZ, L.; CAWREY, D. *Mastering Blockchain*. O'Reilly Media, 2020. ISBN 9781492054658.

23. BULTEL, Xavier; DREIER, Jannik; DUMAS, Jean-Guillaume; LAFOURCADE, Pascal; MIYAHARA, Daiki; MIZUKI, Takaaki; NAGAO, Atsuki; SASAKI, Tatsuya; SHINAGAWA, Kazumasa; SONE, Hideaki. Physical Zero-Knowledge Proof for Makaro. In: IZUMI, Taisuke; KUZNETSOV, Petr (eds.). *Stabilization, Safety, and Security of Distributed Systems*. Cham: Springer International Publishing, 2018, pp. 111–125. ISBN 978-3-030-03232-6.

24. ROBERT, Léo; MIYAHARA, Daiki; LAFOURCADE, Pascal; MIZUKI, Takaaki. Physical Zero-Knowledge Proof for Suguru Puzzle. In: DEVISMES, Stéphane; MITTAL, Neeraj (eds.). *Stabilization, Safety, and Security of Distributed Systems*. Cham: Springer International Publishing, 2020, pp. 235–247. ISBN 978-3-030-64348-5.

25. WU, Huixin; WANG, Feng. A Survey of Noninteractive Zero Knowledge Proof System and Its Applications. *The Scientific World Journal* [online]. 2014, vol. 2014, pp. 1–7 [visited on 2021-03-24]. ISSN 2356-6140, ISSN 1537-744X. Available from DOI: `10.1155/2014/560484`.
26. LAVRENOV, Dmitry [online]. 2019 [visited on 2021-03-24]. Available from: `https://www.altoros.com/blog/zero-knowledge-proof-improving-privacy-for-a-blockchain`.
27. ZCASH. *What are zk-SNARKs? | Zcash* [online]. 2019 [visited on 2021-03-24]. Available from: `https://z.cash/technology/zksnarks`.
28. ALLEN, Christopher. The Path to Self-Sovereign Identity. *CoinDesk* [online]. 2016 [visited on 2021-03-27]. Available from: `https://www.coindesk.com/path-self-sovereign-identity`.
29. WERBACH, Kevin. *The blockchain and the new architecture of trust.* Cambridge, MA: MIT Press, 2018. Information policy series. ISBN 9780262038935.
30. YOUNG, Kaliya. *The domains of identity: a framework for understanding identity systems in contemporary society.* New york: Anthem Press, 2020. Anthem ethics of personal data collection. ISBN 9781785273704.
31. DECENTRALIZED IDENTITY FOUNDATION. The Rising Tide of Decentralized Identity - Decentralized Identity Foundation - Medium. *Medium* [online]. 2017 [visited on 2021-03-27]. ISSN 2163-4663. Available from: `https://medium.com/decentralized-identity/the-rising-tide-of-decentralized-identity-2e163e4ec663`.
32. REED, Drummond. *Decentralized Identifiers (DIDs): The Fundamental Building Block of SSI* [online]. 2018 [visited on 2021-03-27]. Available from: `https://ssimeetup.org/decentralized-identifiers-did-fundamental-block-self-sovereign-identity-drummond-reed-webinar-2`.
33. SPORNY, Manu; SABADELLO, Markus; REED, Drummond. *Decentralized Identifiers (DIDs) v1.0* [online]. 2021-03 [visited on 2021-03-28]. Candidate Recommendation. W3C. Available from: `https://www.w3.org/TR/2021/CR-did-core-20210318/`.
34. GROUP, W3C Credentials Community. *A Primer for Decentralized Identifiers* [online]. 2020 [visited on 2021-03-28]. Available from: `https://w3c-ccg.github.io/did-primer`.
35. SABADELLO, Markus. *DID Resolution: Given a DID how do I retrieve its document?* [Online]. 2018 [visited on 2021-03-27]. Available from: `https://ssimeetup.org/did-resolution-given-did-how-do-retrieve-document-markus-sabadello-webinar-13`.
36. DECENTRALIZED-IDENTITY. *universal-resolver* [online]. 2021 [visited on 2021-03-31]. Available from: `https://github.com/decentralized-identity/universal-resolver`.

37. SABADELLO, Markus. A Universal Resolver for self-sovereign identifiers. *Medium* [online]. 2017 [visited on 2021-03-31]. Available from: `https://medium.com/decentralized-identity/a-universal-resolver-for-self-sovereign-identifiers-48e6b4a5cc3c`.

38. PREUKSCHAT, Alex. *SELF-SOVEREIGN IDENTITY: decentralized digital identity and verifiable credentials.* S.l.: O'REILLY MEDIA, 2021. ISBN 9781617296598. OCLC: 1236259321.

39. RUFF, Tyler. *Verifiable Credentials 101 for SSI* [online]. 2018 [visited on 2021-03-27]. Available from: `https://ssimeetup.org/verifiable-credentials-101-ssi-tyler-ruff-webinar-11`.

40. BURNETT, Daniel; NOBLE, Grant; ZUNDEL, Brent; LONGLEY, Dave; SPORNY, Manu. *Verifiable Credentials Data Model 1.0* [online]. 2019-11 [visited on 2021-03-28]. W3C Recommendation. W3C. Available from: `https://www.w3.org/TR/2019/REC-vc-data-model-20191119/`.

41. OTTO, Nate; LEE, Sunny; SLETTEN, Brian; BURNETT, Daniel; SPORNY, Manu; EBERT, Ken. *Verifiable Credentials Use Cases* [online]. 2020 [visited on 2021-03-28]. Available from: `https://w3c.github.io/vc-use-cases`.

42. SABADELLO, Markus. *Introduction to DID Auth for SSI* [online]. 2018 [visited on 2021-03-27]. Available from: `https://ssimeetup.org/introduction-did-auth-markus-sabadello-webinar-10`.

43. SABADELLO, Markus; HARTOG, Kyle Den; LUNDKVIST, Christian; FRANZ, Cedric; ELIAS, Alberto; HUGHES, Andrew; JORDAN, John; ZAGIDULIN, Dmitri. Introduction to DID Auth. In: *Rebooting the Web of Trust VI.* 2018.

44. LONGLEY, Dave; SPORNY, Manu. *Credential Handler API 1.0* [online]. 2021 [visited on 2021-03-29]. Available from: `https://w3c-ccg.github.io/credential-handler-api`.

45. JONES, J. C.; LUNDBERG, Emil; BALFANZ, Dirk; CZESKIS, Alexei; KUMAR, Akshay; LINDEMANN, Rolf; HODGES, Jeff; JONES, Michael. *Web Authentication: An API for accessing Public Key Credentials - Level 2* [online]. 2021-02 [visited on 2021-03-29]. Proposed Recommendation. W3C. Available from: `https://www.w3.org/TR/2021/PR-webauthn-2-20210225/`.

46. BASART, Ivan; CASATI, Egido; JONES, Michael B.; JUNGE, Andrés; STARK, David; TERBU, Oliver; ZAGIDULIN, Dmitri. Using OpenID Connect Self-Issued to Achieve DID Auth. In: *Rebooting the Web of Trust VIII.* 2019.

47. DECENTRALIZED IDENTITY FOUNDATION. DIF & OIDF - Decentralized Identity Foundation - Medium. *Medium* [online]. 2020 [visited on 2021-03-29]. Available from: https://medium.com/decentralized-identity/dif-oidf-9753b9bd0093.

48. TERBU, Oliver; BASART, Ivan; HARTOG, Kyle Den; LUNDKVIST, Christian; STARK, David; ZAGIDULIN, Dmitri; STROCKIS, Danny; STEELE, Orie. *Self-Issued OpenID Connect Provider DID Profile v0.1* [online]. 2020 [visited on 2021-03-29]. Available from: https://identity.foundation/did-siop.

49. TERBU, Oliver. Using OpenID Connect with Decentralized Identifiers. *Medium* [online]. 2019 [visited on 2021-03-29]. Available from: https://medium.com/decentralized-identity/using-openid-connect-with-decentralized-identifiers-24733f6fa636.

50. KALIYA-IDENTITYWOMAN. Where to begin with OIDC and SIOP - Decentralized Identity Foundation - Medium. *Medium* [online]. 2020 [visited on 2021-03-29]. Available from: https://medium.com/decentralized-identity/where-to-begin-with-oidc-and-siop-7dd186c89796.

51. REED, Drummond. *Decentralized Key Management (DKMS): An Essential Missing Piece of the SSI Puzzle* [online]. 2018 [visited on 2021-03-27]. Available from: https://ssimeetup.org/decentralized-key-management-dkms-essential-missing-piece-ssi-puzzle-drummond-reed-webinar-8.

52. HYPERLEDGER. *indy-sdk* [online]. 2021 [visited on 2021-03-29]. Available from: https://github.com/hyperledger/indy-sdk/blob/677a0439487a1b7ce64c2e62671ed3e0079cc11f/doc/design/005-dkms/DKMS%5C%20Design%5C%20and%5C%20Architecture%5C%20V3.md.

53. BUCHNER, Daniel. Rhythm and Melody: How Hubs and Agents Rock Together. *Medium* [online]. 2019 [visited on 2021-03-30]. Available from: https://medium.com/decentralized-identity/rhythm-and-melody-how-hubs-and-agents-rock-together-ac2dd6bf8cf4.

54. (DIF), DID Communication Working Group. *DIDComm Messaging Specification* [online]. 2021 [visited on 2021-03-31]. Available from: https://identity.foundation/didcomm-messaging/spec.

55. KALIYA-IDENTITYWOMAN. Understanding DIDComm - Decentralized Identity Foundation - Medium. *Medium* [online]. 2020 [visited on 2021-03-31]. Available from: https://medium.com/decentralized-identity/understanding-didcomm-14da547ca36b.

56. DECENTRALIZED-IDENTITY. *universal-registrar* [online]. 2021 [visited on 2021-03-31]. Available from: https://github.com/decentralized-identity/universal-registrar.

57. SECURE DATA STORAGE WORKING GROUP. *DIF - Secure Data Storage Working Group* [online]. 2021 [visited on 2021-03-31]. Available from: https://identity.foundation/working-groups/secure-data-storage.html.

58. SECURE DATA STORAGE WORKING GROUP. *Confidential Storage 0.1* [online]. 2021 [visited on 2021-03-31]. Available from: https://identity.foundation/confidential-storage.

59. HARDMAN, Daniel. Peer DIDs moving to DIF's ID Working Group - Decentralized Identity Foundation - Medium. *Medium* [online]. 2020 [visited on 2021-03-31]. Available from: https://medium.com/decentralized-identity/peer-dids-moving-to-difs-id-working-group-7f1664bcbf30.

60. DEVENTER, Oskar; LUNDKVIST, Christian; CSERNAI, Márton; HARTOG, Kyle Den; SABADELLO, Markus; CURREN, Sam; GISOLFI, Dan; VARLEY, Mike; HAMMANN, Sven; JORDAN, John; HARCHANDANI, Lovesh; FISHER, Devin; LOOKER, Tobias; ZUNDEL, Brent; STEPHEN CURRAN. *Peer DID Method Specification* [online]. 2021 [visited on 2021-03-31]. Available from: https://identity.foundation/peer-did-method-spec.

61. HARDMAN, Daniel. *Peer DIDs: a secure and scalable method for DIDs that's entirely off-ledger* [online]. 2019 [visited on 2021-03-27]. Available from: https://ssimeetup.org/peer-dids-secure-scalable-method-dids-off-ledger-daniel-hardman-webinar-42.

62. DECENTRALIZED IDENTITY WEB DIRECTORY. *Decentralized Identity Web Directory* [online]. 2020 [visited on 2021-03-31]. Available from: https://decentralized-id.com/#organizations.

63. DECENTRALIZED IDENTITY FOUNDATION. *DIF - Decentralized Identity Foundation Members* [online]. 2021 [visited on 2021-03-31]. Available from: https://identity.foundation/#members.

64. HELMY, Nader. Overview of Decentralized Identity Standards - Decentralized Identity Foundation - Medium. *Medium* [online]. 2020 [visited on 2021-03-31]. Available from: https://medium.com/decentralized-identity/overview-of-decentralized-identity-standards-f82efd9ab6c7.

65. MCDONALD, Malcolm. *Web security for developers.* San Francisco: No Starch Press, Inc, 2020. ISBN 9781593279950.

66. WILSON, Yvonne; HINGNIKAR, Abhishek. *Solving identity management in modern applications: demystifying OAuth 2.0, OpenID connect, and SAML 2.0.* APress, 2019. ISBN 9781484250945. OCLC: 1158466530.

67. OKTADEV. *OAuth 2.0 and OpenID Connect (in plain English)* [online]. 2018 [visited on 2021-02-10]. Available from: https://www.youtube.com/watch?v=996OiexHze0&ab_channel=OktaDev.

68. SPILCA, Laurentiu. *Spring Security in Action* [online]. S.l.: Manning Publications, 2020 [visited on 2021-03-10]. ISBN 9781617297731. OCLC: 1224915715.

69. HARDT, D. *The OAuth 2.0 Authorization Framework* [online]. RFC Editor, 2012-10 [visited on 2021-03-10]. RFC, 6749. RFC Editor. ISSN 2070-1721. Available from: `http://www.rfc-editor.org/rfc/rfc6749.txt`.

70. SPASOVSKI, Martin. *OAuth 2.0 identity and access management patterns* [online]. Packt Publishing, 2013 [visited on 2021-03-10]. ISBN 9781783285594. OCLC: 900918626.

71. RAIBLE, Matt. *What the Heck is OAuth?* [Online]. 2017 [visited on 2021-02-10]. Available from: `https://developer.okta.com/blog/2017/06/21/what-the-heck-is-oauth`.

72. ANWAR, Haseeb. The Complete Guide to OAuth 2.0 and OpenID Connect Protocols. *Medium* [online]. 2020 [visited on 2021-02-10]. Available from: `https://medium.com/better-programming/the-complete-guide-to-oauth-2-0-and-openid-connect-protocols-35ebc1cbc11a`.

73. SAKELLARIOS, Christos. ASP.NET Core Identity Series – OAuth 2.0, OpenID Connect & IdentityServer. *chsakell's Blog* [online]. 2019 [visited on 2021-02-10]. Available from: `https://chsakell.com/2019/03/11/asp-net-core-identity-series-oauth-2-0-openid-connect-identityserver`.

74. BOYD, Ryan. *Getting started with OAuth 2.0*. Beijing ; Sebastopol, Calif: O'Reilly, 2012. ISBN 9781449311605. OCLC: ocn764382903.

75. BIHIS, Charles. *Mastering OAuth 2.0: create powerful applications to interact with popular service providers such as Facebook, Google, Twitter, and more by leveraging the OAuth 2.0 Authorization Framework*. Packt Publishing, 2015. ISBN 9781784395407. OCLC: 1010940989.

76. RICHER, Justin. *End User Authentication with OAuth 2.0 — OAuth* [online]. [N.d.] [visited on 2021-02-10]. Available from: `https://oauth.net/articles/authentication`.

77. FOUNDATION, OpenID. *OpenID Connect Core 1.0 incorporating errata set 1* [online]. 2014-11 [visited on 2021-03-10]. Tech. rep. OpenID Foundation. Available from: `https://openid.net/specs/openid-connect-core-1_0.html`.

78. AUTH0. *JSON Web Tokens - jwt.io* [online]. Auth0, 2021 [visited on 2021-02-16]. Available from: `https://jwt.io/introduction`.

79. JONES, M.; BRADLEY, J.; SAKIMURA, N. *JSON Web Token (JWT)* [online]. RFC Editor, 2015-05 [visited on 2021-03-10]. RFC, 7519. RFC Editor. ISSN 2070-1721. Available from: `http://www.rfc-editor.org/rfc/rfc7519.txt`.

80. SIRIWARDENA, Prabath. *Advanced api security: securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*. Berkeley, California: Apress, 2014. ISBN 9781430268185.

81. CAMPBELL, B.; MORTIMORE, C.; JONES, M. *Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants* [online]. RFC Editor, 2015-05 [visited on 2021-03-10]. RFC, 7522. RFC Editor. ISSN 2070-1721. Available from: `http://www.rfc-editor.org/rfc/rfc7522.txt`.

82. PINGIDENTITY. *SAML 2.0: How It Works* [online]. [N.d.] [visited on 2021-02-10]. Available from: `https://www.pingidentity.com/en/resources/client-library/articles/saml.html`.

83. COMPUTING, VMware End-User. *SAML 2.0: Technical Overview* [online]. 2019 [visited on 2021-02-10]. Available from: `https://www.youtube.com/watch?v=SvppXbpv-5k&ab_channel=VMwareEnd-UserComputing`.

84. ČVUT - VÝPOČETNÍ A INFORMAČNÍ CENTRUM. *Single Sign On (jednotn pihlaovn) | IST | esk vysok uen technick v Praze* [online]. 2021 [visited on 2021-05-02]. Available from: `https://ist.cvut.cz/nase-sluzby/single-sign-on`.

85. AUTH0. *Role-Based Access Control* [online]. [N.d.] [visited on 2021-04-21]. Available from: `https://auth0.com/docs/authorization/rbac`.

86. AUTH0. *What is Role-Based Access Control (RBAC)?* [Online]. Auth0, [n.d.] [visited on 2021-04-23]. Available from: `https://auth0.com/intro-to-iam/what-is-role-based-access-control-rbac`.

87. OKTA. *What Is Attribute-Based Access Control (ABAC)?* [Online]. Okta, Inc., 2020 [visited on 2021-04-23]. Available from: `https://www.okta.com/blog/2020/09/attribute-based-access-control-abac`.

88. BRAENDGAARD, Pelle. *EIP-1812: Ethereum Verifiable Claims* [online]. 2019-03 [visited on 2021-04-23]. Draft. Ethereum. Available from: `https://eips.ethereum.org/EIPS/eip-1812`.

89. NETWORK, Aventus. Working with Strings in Solidity. *Medium* [online]. 2018 [visited on 2021-04-23]. Available from: `https://blog.aventus.io/working-with-strings-in-solidity-473bcc59dc04`.

90. OPENZEPPELIN. *Access Control - OpenZeppelin Docs* [online]. 2021 [visited on 2021-04-23]. Available from: `https://docs.openzeppelin.com/contracts/2.x/access-control`.

91. LIU, Bowen; SUN, Siwei; SZALACHOWSKI, Pawel. SMACS: Smart Contract Access Control Service. In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2020, pp. 221–232. Available from DOI: `10.1109/DSN48063.2020.00039`.

92. ETHEREUM. *Solidity — Solidity 0.8.4 documentation* [online]. 2021 [visited on 2021-04-26]. Available from: `https://docs.soliditylang.org/en/v0.8.4`.
93. CONSENSYS SOFTWARE. *Ganache | Truffle Suite* [online]. 2021 [visited on 2021-04-26]. Available from: `https://www.trufflesuite.com/ganache`.
94. CONSENSYS SOFTWARE. *Ganache | Overview | Documentation | Truffle Suite* [online]. 2021 [visited on 2021-04-26]. Available from: `https://www.trufflesuite.com/docs/ganache/overview`.
95. METAMASK. *MetaMask* [online]. 2021 [visited on 2021-04-26]. Available from: `https://metamask.io`.
96. METAMASK. *Introduction | MetaMask Docs* [online]. 2021 [visited on 2021-04-26]. Available from: `https://docs.metamask.io/guide`.
97. ETHEREUM. *Remix - Ethereum IDE & community* [online]. 2020 [visited on 2021-04-26]. Available from: `https://remix-project.org`.
98. ETHEREUM. *Welcome to Remix's documentation! — Remix - Ethereum IDE 1 documentation* [online]. 2021 [visited on 2021-04-26]. Available from: `https://remix-ide.readthedocs.io/en/latest`.
99. UPORT. *uPort Developer Portal* [online]. 2020 [visited on 2021-04-26]. Available from: `https://developer.uport.me`.
100. NODE.JS. *About | Node.js* [online]. 2021 [visited on 2021-04-26]. Available from: `https://nodejs.org/en/about`.
101. EXPRESS. *Express - Node.js web application framework* [online]. 2021 [visited on 2021-04-26]. Available from: `https://expressjs.com`.
102. NGROK. *ngrok - secure introspectable tunnels to localhost* [online]. 2021 [visited on 2021-04-26]. Available from: `https://ngrok.com`.
103. NGROK. *ngrok – documentation* [online]. 2021 [visited on 2021-04-26]. Available from: `https://ngrok.com/docs`.
104. BOOTSTRAP CONTRIBUTORS. *Bootstrap* [online]. 2021 [visited on 2021-04-26]. Available from: `https://getbootstrap.com`.
105. OPENZEPPELIN. *Contracts - OpenZeppelin Docs* [online]. 2021 [visited on 2021-04-26]. Available from: `https://docs.openzeppelin.com/contracts/4.x`.
106. ETHERS. *Documentation* [online]. 2021 [visited on 2021-04-27]. Available from: `https://docs.ethers.io/v5`.
107. INFURA. *Ethereum API | IPFS API & Gateway | ETH Nodes as a Service | Infura* [online]. 2021 [visited on 2021-05-02]. Available from: `https://infura.io`.

Appendix A

Acronyms

ABAC Attribute-Based Access Control

ABI Application Binary Interface

API Application Programming Interface

BPMN Business Process Model and Notation

CDN Content Delivery Network

CORS Cross-Origin Resource Sharing

CRUD Create, Read, Update, Delete

CSS Cascading Style Sheets

CTU Czech Technical University in Prague

dApps Decentralized Applications

DApps Decentralized Applications

DEMO Design & Engineering Methodology for Organizations

DID Decentralized Identifier / Decentralized Identity

DIF Decentralized Identity Foundation

DKMS Decentralized Key Management

DKPI Decentralized Public Key Infrastructure

DNS Domain Name System

DSL Domain Specific Language

EOA Externally Owned Account

EVM Ethereum Virtual Machine

GUI Graphical User Interface

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
JSON	JavaScript Object Notation
JSON-RPC	JSON Remote Procedure Call
JWT	JSON Web Token
OIDC	OpenID Connect
OIDF	OpenID Foundation
P2P	Peer-to-Peer
PGP	Pretty Good Privacy
PoS	Proof of Stake
PoW	Proof of Work
RBAC	Role-Based Access Control
REST	Representational State Transfer
RPC	Remote Procedure Call
SAML	Security Assertion Markup Language
SDK	Software Development Kit
SIOP	Self-Issued OpenID Connect Provider
SMACS	Smart Contract Access Control Service
SSH	Secure Shell
SSI	Self-Sovereign Identity
SSO	Single Sign-On
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URN	Uniform Resource Name
VC	Verifiable Credential
VP	Verifiable Presentation
W3C	World Wide Web Consortium
XML	Extensible Markup Language
ZKP	Zero-knowledge Proof

Appendix B

Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
src ...................................... the directory of source codes
  thesis .............. the directory of LaTeX source codes of the thesis
  proof-of-concept .......... proof of concept implementation sources
    access-control-service .......... Access Control Service sources
    smart-contracts ........................ Smart Contracts sources
    client-app ................................. Client App sources
text .......................................... the thesis text directory
  thesis.pdf ........................... the thesis text in PDF format
video ....... the directory containing screen recording of the PoC testing
```