



Università degli Studi di Pisa

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

**Privacy-preserving delivery with
Self-Sovereign Identity - An
implementation with Hyperledger
Indy**

Candidato:

Calogero Turco

Relatori:

Prof.ssa Laura Emilia Maria Ricci

Dott. Andrea Lisi

Anno Accademico 2020-2021

Contents

1 Introduction	5
2 Background	11
2.1 Secure messages using cryptography	11
2.1.1 Public-key cryptography	11
2.1.2 Random Number Generator (RNG) and Collision Free Hash Functions	14
2.2 Blockchain	14
2.3 Bitcoin	15
2.3.1 Privacy in Bitcoin	15
2.3.2 Transactions in Bitcoin	15
2.3.3 Proof Of Work and Nodes	16
2.4 Ethereum	18
2.4.1 Merkle Tree	19
2.5 Self Sovereign Identity	20
2.5.1 SSI and Identifiers	21
2.5.2 Blockchain and SSI	22
2.5.3 Verifiable Credentials and Data Model	22
2.5.4 Selective Disclosure and Zero-knowledge proofs	24
2.5.5 What goes on the blockchain?	25
3 Hyperledger Indy, DID and credentials	28
3.1 Hyperledger-Indy and its history	28
3.2 Indy Architecture	29
3.2.1 Indy ledger(s) and nodes	29
3.2.2 Transactions and Ledgers	30
3.2.3 Nodes and ledgers (Indy-plenum)	31
3.3 Identity in Indy: Agent - Identity Owner	33
3.4 Identifiers in Indy: Decentralized Identifiers	35

3.4.1	NYM	38
3.4.2	DID in Indy and Sovrin	41
3.4.3	Roles	43
3.4.4	DID to DID Communication and Pairwise DID	45
3.5	Verifiable Credentials in Indy	51
3.5.1	What goes on the ledger?	51
3.5.2	Credential Schema and SCHEMA transaction	53
3.5.3	Credential Definition	54
3.5.4	Revocation Registry and Revocation Registry Entry	58
3.5.5	REV_REG_DEF transaction	60
3.5.6	Credential Offer and Credential Request	61
3.5.7	Credentials, their structure, and how MasterSecret links them to their holder	65
3.5.8	Proof request	66
3.5.9	Proof	69
3.5.10	Unlinkability of attributes	71
3.5.11	Portable Credentials: Student Mobility Use Case	72
3.6	Anoncreds Workflow	74
3.6.1	Get a NYM with a role in the ledger	75
3.6.2	Credential Issuing	77
3.6.3	Creating a Proof	78
3.6.4	Verifying Proofs	79
3.6.5	Conclusions	80
4	Privacy preserving deliveries with SSI	82
4.1	Portability versus privacy-preserving authentication	82
4.2	Shipping Verification System: a privacy-preserving solution	83
4.2.1	Definition of the system	83
4.2.2	Store creation	86
4.2.3	Ordering an Item	87
4.2.4	Reserve a Mail Box	87
4.2.5	Delegate a Shipping Deliverer	88
4.2.6	Customer retrieving the items	91
4.3	Shipping Verification System and SSI	92
4.4	Threats - what could go wrong?	94
5	An implementation with Hyperledger Indy	98
5.1	An additional layer of abstraction: IndyLibraries	98
5.2	Steward setup implementation	101

5.2.1	Connection to a pool	102
5.2.2	Generating a Steward NYM	103
5.2.3	Making the Steward endpoint public with an ATTRIB transaction	104
5.3	Store setup as Issuer implementation	106
5.3.1	createConnection - DID to DID setup	108
5.3.2	Assigning Endorser role - NYM transaction	109
5.4	Setup for orders - on ledger Anoncred entities	110
5.4.1	Publish a Schema on the ledger	110
5.4.2	Publish a Credential Definition on the ledger	110
5.5	Give a package to a Shipping Deliverer - Issuing a credential	110
5.6	Reading a proof attribute values - MailBox after receiving proof from Shipping Deliverer	115
5.6.1	Communication with MailBox - Customer to MailBox communication	116
5.6.2	Opening the MailBox - Proof request creation	118
5.7	Fetching attribute for proof - Customer search for Shipping- Credential	119
5.8	Proof - creating a proof to open the MailBox	120
5.8.1	MailBox verification for opening - Verify proof	120
6	Experimental Evaluations	123
6.1	Threat model tests	123
6.1.1	Threat model Scenario: The Shipping Agent receives critical information to impersonate the Customer - credential theft	123
6.1.2	Threat model Scenario: Proof falsification - Customer modifies a proof attributes value	125
6.2	Networks tests	128
6.2.1	Sovrin BuilderNet test - performance on unique trans- actions submission	128
6.2.2	Local Indy pool test - performance on unique transac- tions submission	131
6.2.3	BulderNet and local Indy pool on a specific request repeated	134
6.3	Agent side operations measurements	135
6.3.1	Testing methodology	135
6.3.2	Proof request and proof in different cases	135
6.3.3	Credential measures	136

6.4 Node Validator info and performance measures	137
6.4.1 Testing methodology	138
6.4.2 Contextualize the results: what can be seen?	138
7 Conclusions	141
7.1 Future improvements	143
A Appendix	144
A.1 DID Document	144
A.2 Proof	145
A.3 Valid proof and an alteration of it(non-valid)	146
A.3.1 Valid proof	146
A.3.2 Altered proof(non-valid)	147
A.4 Tables of entities (Chapter 3)	148
A.5 Setup Indy	152
A.5.1 Installing libindy on a Linux based system	152
A.5.2 Run a local pool with Docker	152
A.5.3 Get and use Indyscan containers image for nodes . .	152
A.5.4 Client side with Indy-SDK using Java wrapper	154
A.6 Sovrin BuilderNet transaction test operations	157
A.6.1 Iteration 1	158
A.6.2 Iteration 2	159
A.7 Local Indy pool transaction test operations	161
A.7.1 Rejected SCHEMA request	161
A.7.2 Accepted SCHEMA request	161
A.7.3 Rejected CLAIM_DEF request	162
A.7.4 Accepted CLAIM_DEF request	162
A.7.5 Read requests locally	162
A.8 Local Indy Pool and MainNetCredential Definition update .	163
A.8.1 Local Indy Pool	163
A.8.2 Sovrin MainNet	164
A.9 Example of a NODE transaction and GetValidatorInfo request	164
A.9.1 NODE transaction	164
A.9.2 GetValidatorInfo request	165
Bibliography	170

Chapter 1

Introduction

Can privacy-preserving, portable, and secure credentials be integrated into real-world processes? During this decade, more than ever, the phenomenon of the digital identity of a person scattered into a multitude of accounts is an issue for privacy and security. A few examples are Social Networks, e-commerce, multimedia, and work-related services. However, either having an account with public or private institutions, it is not the person who manages and stores their data, but these important operations are delegated to the institutions in question. **Self-Sovereign Identity** is a paradigm that aims to bring the control of the data back to the user.

A shift to **Self-Sovereign Identity** (SSI) would be a step in the right direction for personal privacy. In SSI, a person stores and keeps all their data, as **verifiable credentials**, in a personal device typically known as *wallet* and uses them only when needed. A credential is so-called verifiable because it contains a **proof**, typically a digital signature used to validate its authenticity. In this case, people, and not service providers, have control of their private data. The credential aforementioned are defined in the **Verifiable Credential Data Model**, whose specification is managed by the World Wide Web Consortium (W3C).

This thesis analyzes how Self-Sovereign Identity can mitigate a real-world threat, focusing on how it can protect the identity of customers in a package delivery scenario. We analyze how the customers' identity in a package delivery scenario can be and how Self-Sovereign Identity can mitigate a real-world threat. In the past, there were cases of online stores databases being compromised, which, in turn, compromised the

identities of their customers. A data breach of the store Home Chef compromised the personal data of its many customers [1], including their home addresses. This could pose a serious threat to a person's security since customers' identities can be obtained from order data. From Eurostat statistics [2], 74% of internet users in Europe ordered from online stores in 2021. Such a large gold mine of data is tempting for data stealers. The same study found that payment security and privacy security were the third cause of people not wanting to buy online, demonstrating that the problem is still open. Moreover, data about a person could be aggregated from various sources and used to compose an identity of said person. In 2020 there was the theft of more than 500.000 accounts on Zoom [3]. All the blame does not lie only on the Zoom company, but the accounts were stolen using login details from a previous leak. This means that the many fragments of a person's identity can be **correlated**, enriching more and more the identikit that a malicious entity could create.

But can the online shop or any service provider be trusted? As said, those providers store user data and have control of them, allowing them to use it. It is not an unfathomable fact to think that those data could be disclosed to the best offer. As such, the relationship between a person and a service provider should be secure and protect the user's privacy. **Decentralized Identifiers** (DIDs) are another component of SSI and might be the right solution for such cases. Their specifications come from W3C, and they allow for privacy-preserving identification. It will be seen how *by using more than one DID a person could communicate with various subjects in an authenticated manner without risking for their identities in different communications to be correlated*.

Figure 1.1 encompasses the principal questions posed in this thesis and their proposed answers.

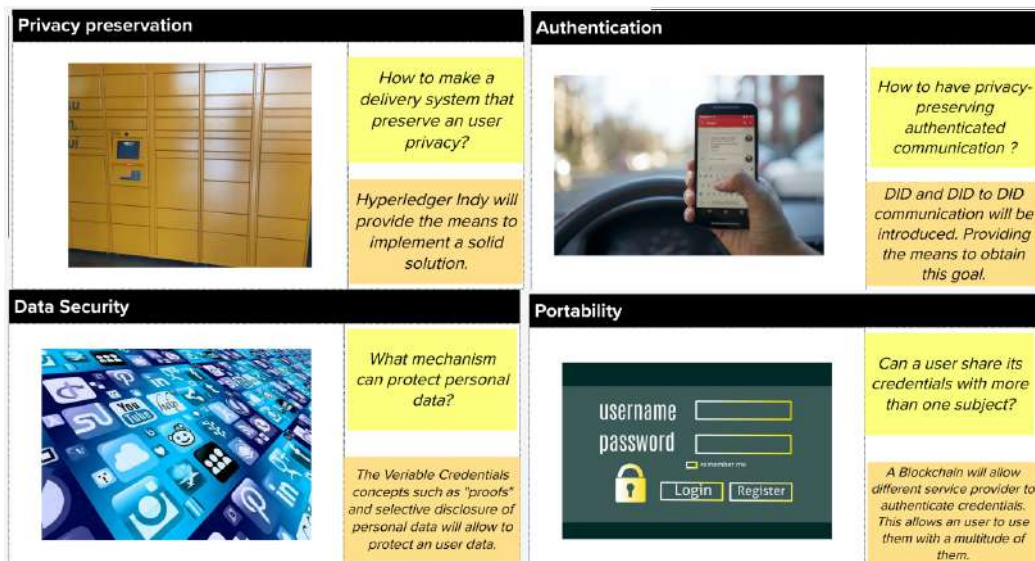


Figure 1.1: Some question with their answer discussed in this thesis.

This thesis begins with the definition of a Self-Sovereign Identity system, unraveling all the concepts and operations involved in a system, continues describing Hyperledger Indy, and terminates presenting the structure, implementation, and evaluation of the use case, a delivery system. The chapters encompass the roadmap as follows:

- Chapter 2 Background:** This chapter provides an overview of the tools and concepts composing the groundwork of the technologies discussed in this thesis. In particular, the chapter starts by focusing on a short description of the cryptography tools used in this thesis and blockchain technology. Then **Self-Sovereign Identity system** is introduced, along with its goals and challenges. Concepts such as **Zero-Knowledge proofs** and **Selective Disclosure** are presented. The first allows a person to present only the proof of possessing a Verifiable Credential, without disclosing it. The second allows the disclosure publicly of just a subset of values from a Verifiable Credential. The **Verifiable Credential Data Model** of the W3C is introduced and compared with alternatives. The chapter shows that the Verifiable Credential Data Model uses a Verifiable Registry as public storage. But it has limited use since any traces of a credential Verifiable Credential creation are absent from the Verifiable Registry. It is used mainly by the Issuer, an authority issuing Verifiable Credentials, using

it to make its identifiers and the format of its credentials public. The chapter contains discussions on identifiers and the pitfalls of having human-readable, secure, and decentralized identifiers. The chapter concludes with a justification for blockchain technology involvement and the blockchain type identified as the best of a Self-Sovereign Identity system, identifying public permissioned blockchains as the best-suited ones.

⚡ **Chapter 3 - Hyperledger Indy and DID:** This chapter describes Hyperledger Indy, the blockchain-based SSI framework that powers the privacy-preserving delivery system implemented in this thesis. The chapter provides a detailed description of the Indy architecture and its components. What sets Indy apart from other blockchain solutions is highlighted in the chapter. As it is mainly catered for SSI and is public permissioned, energy consumption is lower than other blockchains for an Indy system. The chapter then delves into the concept of **agent** as a software representing a user in an Indy SSI system. The chapter introduces **DID** from the W3C specification alongside their **Sovrin specification** and their use in Indy. *DID are an example of human-readable, secure, and decentralized identifiers.* The chapter carefully describes the most important features such as the **Pairwise DID** communication between two users and the **Anoncreds entities** data structures to implement verifiable credentials. A different DID is used for authentication in every communication, protecting the parties' privacy from correlation. The Anoncreds entities are divided into entities for **identity, credentials, revocation, and proofs**. The chapter underlines the division between **public entities** on the blockchain and that **privately exchanged** outside of the blockchain. As Indy is a public permissioned blockchain, there is a distinction between agents having a **public and private identity**. The chapter introduces roles such as Trustee and Stewards, agents managing the Indy system network. The **issuers** of credentials have the role of Endorsers. A strong distinction between them and the **credentials holder** is highlighted, with **the first having their identity public while the latter not**, allowing stronger privacy for credentials holders. Issuers sign transactions toward the Indy system with a private key paired with their DID, with its public key used for signature verification, is stored in the blockchain. Also, credentials

are signed and verified with a different key-pair, where the private counterpart verifies proofs and is on the blockchain. *Credentials holders do not send credentials. It is proof of having them which presented to requesters.* For contextualizing all the introduced concepts, the chapter provides an example of **Anoncreds workflow** applied to the use case of the Erasmus+ Programme to show the **portability** of verifiable credentials.



Chapter 4 Privacy-preserving deliveries with SSI: This chapter models a Shipping Verification System based on shared Mailboxes, such as Amazon Lockers, following the SSI architecture to preserve the privacy of a customer. The chapter lists the main **actors** of the system and their correspondence to the actors of the **Anoncreds workflow** presented in the previous chapter. The system is designed to resist attacks such as **proof falsification** and imposes a few constraints, such as package pickup deadlines, to fit in a realistic scenario. The chapter ends with a **threat model** illustrating the possible treats and **limitations** of the system in question, which highlights the **shortcomings** of the Verifiable Credentials Data Model. *The same problems are in an Indy system since dishonest issuers are problems to a system's trustability.*



Chapter 5 - An implementation with Hyperledger Indy: This chapter illustrates the system implementation of the Shipping Verification System using Hyperledger Indy SDK. The system implementation was deployed on a **local network** and also on a **remote network** such as the Sovrin BuilderNet. The chapter discusses the **implementation**, by describing the SDK and the main implementation decisions.



Chapter 6 - Experimental Evaluations This chapter illustrates security and performance **tests**, showing i) how the implemented system resists threat attacks such as **credential theft** and **proof alteration**, and ii) the measurements. The measurements are divided in **transactions latency** measurements, **client-side** operations **execution time** and Anoncreds entities **size measurements** and **node-side performance measurements**. The final test provides an overview of a node behavior during the Shipping Verification System execution.

▲ **Chapter 7 - Conclusions:** The sixth chapter provides the conclusions of the thesis. It summarizes the discoveries made in this thesis and discusses whether Indy can provide solid solutions for the real world. Finally, it gives some ideas for future improvements of the **Shipping Verification System**.

Let us note that, for the sake of simplicity and readability, personal pronouns are used in this thesis. We use 'he' for simplicity, as a neutral gender pronoun.

Chapter 2

Background

This chapter introduces the technologies and frameworks treated in this thesis. The concept of Self-Sovereign Identity is introduced alongside Verifiable Credentials. This chapter starts with an introduction to cryptographic tools needed to exchange authenticated messages and introduce blockchains. Then, we will see how blockchains make Self-Sovereign Identity possible.

2.1 Secure messages using cryptography

This section provides a brief description of the cryptographic tools used in this thesis.

Cryptography allows two entities to communicate securely, making the messages they exchange safe from any observer. An encryption operation takes a message and a secret key, returning a ciphertext. The decryption operation takes an encrypted message, a key, and returns the original message in plain form. To encrypt and decrypt messages there can be just one key used. In this case, the key is a shared key among the two parties (symmetric cryptography). Alternatively, it can be a different key (asymmetric cryptography).

2.1.1 Public-key cryptography

In asymmetric cryptography, also called public-key cryptography, the messages are encrypted and decrypted using two different keys: a Public Key, which is made public by the receiving party and is used to

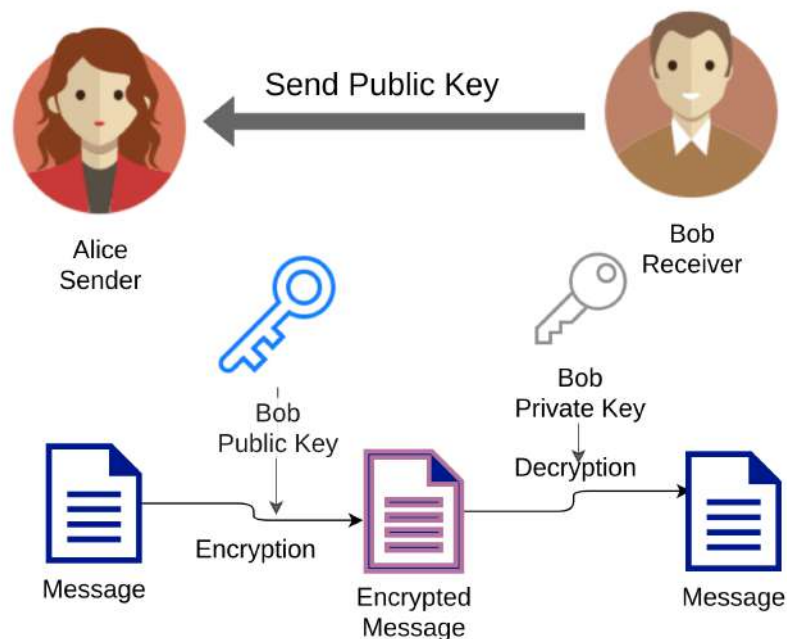


Figure 2.1: Public-key cryptography for confidentiality

encrypt messages; a Private Key, related to a public key, which is the key used to decrypt messages encrypted with the corresponding public key. The private key should remain a secret only known by the key owner (receiver). The sender must receive the authentic receiver public key, as an intruder could create a key pair and substitute his public key for the authentic one while using his generated private key for decryption. Asymmetric keys are mathematically linked, they are generated in such a way that the public key is derived from the private key but not vice versa, i.e. the private key cannot be derived from the public key.

Public-key cryptography for confidentiality

In a message exchange scenario, shown in figure 2.1, the public key of the receiver is used by the sender to encrypt a message. Before the message exchange, the sender and receiver exchange the latter public key using a safe channel, or the sender fetches the public key from a trusted third party. The sender encrypts the message with the public key and sends it. When the encrypted message arrives, the receiver decrypts it with his private key. Only the sender and the receiver know

the content of the message (*confidentiality*).

Digital signature: Non-Repudiation

Asymmetric cryptography can also be used to authenticate messages. The authentication is done through a digital signature, i.e. the sender uses his private key, also known as signing key in this context, to sign a message (as shown in Figure 2.2).

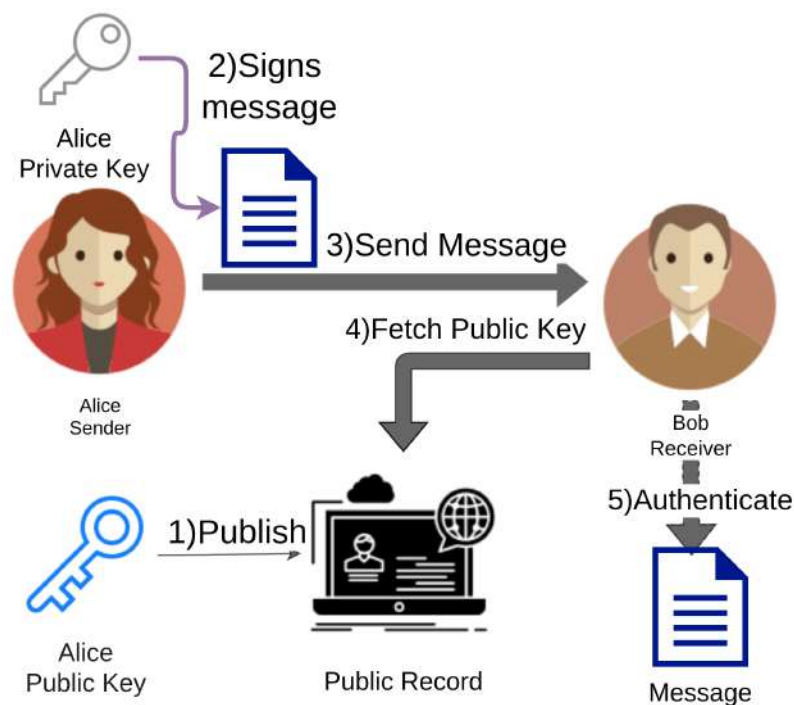


Figure 2.2: Public-key cryptography for authentication

The message receiver uses the public key of the sender to decrypt the signed message. If this operation is successful, it guarantees both: *authentication*, because only the person who knows the signing key can produce a valid signature for the communicated public; *integrity*, because the message cannot be modified along the way without invalidating the signature; *non-repudiation*, i.e. the sender cannot deny the authenticity of the signature since there exists proof that he did it. Combining digital signatures with encryption is possible to perform secure communications.

(EC)DSA

For digital signatures, the Digital Signature Algorithm (DSA) and the Elliptic Curve Digital Signature (ECDSA) are two schemes that implement digital signatures. While they use different classes of groups, DSA uses a cyclic group of primes while ECDSA uses elliptic curve cryptography[4, p.459]. Both maintain the private key secret if used correctly.

2.1.2 Random Number Generator (RNG) and Collision Free Hash Functions

Random Number Generator A Generator g is an algorithm that takes a string as input. Such string is called seed and is used to generate a string of numbers longer than the seed. The generated string looks like a uniform string, where a uniform string is a string composed of elements chosen randomly (with a uniform distribution).

Collision Free Hash Functions Hash functions are defined as functions that take data of arbitrary size as input and return a fixed size output, as with RNG the output is randomly looking.

Collision Free Hash Functions are a category of hash functions, for which it is a requirement to avoid collisions, i.e. applying the hash function to two inputs results in the same output. For an adversary finding a collision would be highly unlikely, i.e. there should not be a polynomial-time algorithm to find it.

2.2 Blockchain

A blockchain is a data structure [5, p.195] that represents a linked list of blocks, each block refers to the previous block in the chain and contains a list of records known as transactions. A blockchain is an implementation of a, typically distributed, ledger and can be seen as an evolution of physical bookkeeping monetary transactions (e.g.: account ledgers). A blockchain serves as a cooperative database distributed among various connected nodes, forming a Peer to Peer (P2P) network, and it is populated with transactions sent by the peers themselves. Each transaction needs to be signed by its sender before being sent to the ledger. Therefore, every transaction has to be validated by all the other

peers before being stored in the blockchain. However, to guarantee each peer stores the same copy of the blockchain, i.e. the same blocks with the same transactions in the same order, the peer proposing the new block is elected through a "consensus algorithm".

2.3 Bitcoin

Bitcoin is the first and most popular Blockchain in the world, which implements the exchange of bitcoins, a cryptocurrency, between two users without relying on a trusted third party, for example, a bank. Instead, the proof that the transaction happened is recorded on the Bitcoin Blockchain as a transaction.

The collection of bitcoins available for exchange is called Unspent Transaction Output (UTXO), and the complete UTXO set represents the total supply of bitcoins available for exchange at the current moment. Every user has a certain amount of UTXO that he can use as input for new transactions. All the funds of a bitcoin user are stored in a wallet, every wallet generates one or more bitcoin addresses from private/public key pairs: bitcoins are "sent" to public keys, and are "spent" using private keys.

2.3.1 Privacy in Bitcoin

Bitcoin users are identified by addresses and hashes of their public keys. When receiving funds, the Bitcoin address of the receiver is made public on the blockchain, meaning that all the fund exchanges can be linked to the same address, the receiver of funds can protect his privacy by using a different bitcoin address for every payment he receives. Since a payment implies a public transaction on the blockchain, the two parties involved in the transaction use their bitcoin address, instead of their public keys, to protect their privacy they can hide their public key, even though the trade is made public the involved parties are not.

2.3.2 Transactions in Bitcoin

In Bitcoin, a transaction is *"a signed data structure expressing a transfer of value. Transactions are transmitted over the bitcoin network, collected by miners, and included into blocks, made permanent on the*

blockchain" [5].

Bitcoin uses the ECDSA algorithm with the elliptic curve **secp256k1** to implement public-key security, and offers the possibility of using **scripts** to allow the creation of transactions valid only when certain conditions are met. Bitcoin scripts are limited since the underlying programming language is not Turing Complete.

2.3.3 Proof Of Work and Nodes

A Bitcoin network, as every blockchain network, (Section 2.2) is composed of nodes, and transactions are sent to the network and propagated to each node. For the time transactions are unverified they are stored in a pool maintained by each node. Roughly every 15 minutes, a batch of transactions is picked from the pool, packed in a block, validated, and inserted into the blockchain. Since nodes receive transactions in a different order, there is a need to validate by verifying their arrival order. Therefore, each node must reach a **consensus** of which transaction arrived first.

A **miner node** is a particular node that participates in the consensus protocol and competes to propose new blocks. A miner node runs specialized hardware to solve the Proof Of Work (PoW) algorithm. When it creates a new block, composed of the transactions enriched with a timestamp, it is concatenated with the node-local copy of the blockchain. A new block will contain the current timestamp and the previous timestamp (as taken from the antecedent block and in hash form), creating a chain of timestamps. At every addition, there is a reinforcement in the structure of the chain of timestamps.

The Proof of Work is calculated by finding the *nonce*. That *nonce* is a value positioned inside the block being hashed together into the blockchain. The resulting hash must match a value with a specified number of zeroes at the beginning. As soon a miner finds such nonce, it attaches it to the block and propagates the block to all the other nodes in the P2P network for validation: if the transaction is correct and the hash is indeed smaller than the difficulty target, the new block is attached to all the other nodes' local blockchains.

This process is complex and resource-consuming (computational-wise). As such it made it necessary to give rewards to the nodes participating. The reward is a special transaction the miner puts into the generated block that includes its bitcoin address, and it consists of a fixed value

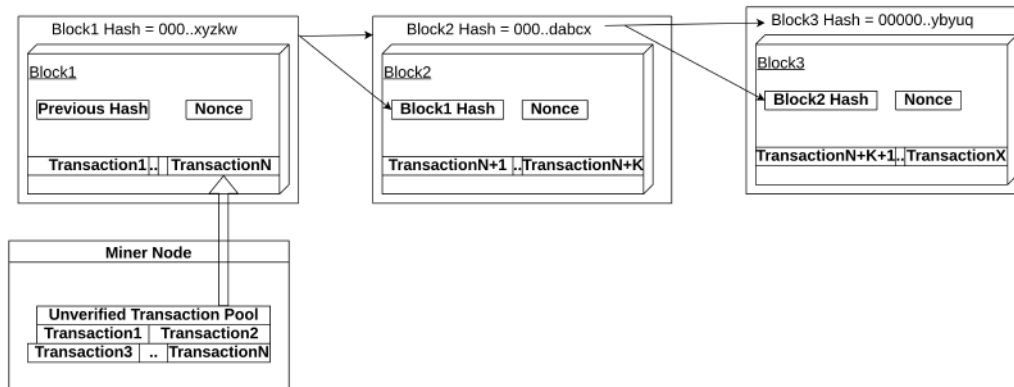


Figure 2.3: Block creation and hash addition in the timestamp server

plus all the sum of the fees specified for all transactions in the block. After the Proof of Work is satisfied, and more blocks are added to the chain, changing a previous block becomes increasingly difficult because it is needed to recalculate the hash for all its subsequent blocks up to the latest.

Bitcoin accounts for the computational power increase of hardware over time and the different number of nodes computing the PoW at a given time. If the number of blocks added to the blockchain on average per hour is high, then the difficulty of generating a Proof of Work increases. Finally, an attacker could try to change the order of the transactions in the blockchain. Assuming he sends a certain sum of bitcoin to a bitcoin address (e.g. a Store) with transaction $t(n+1)$, the transaction will be included in the blockchain, and miners will continue to chain other blocks after that. If the attacker sends another transaction $t(n+2)$ that redirects the same money to him, a miner would recognize that the UTXO used to make the transaction is not of the attacker anymore, **rejecting** the transaction. The attacker could then try to replace $t(n+1)$ with another transaction $t(n+1)'$ in the blockchain, to succeed in his attack, the attacker could start mining and calculating the PoW to put transactions $t(n+1)'$ after transaction $t(n-1)$ in the blockchain. However, the attack would create a **forked blockchain**. Since miners in the case of forks typically consider the longest blockchain as the main blockchain, the attacker cannot succeed in creating the longest chain unless he owns at least 51% of the total hash power. This is known as 51% Attack and can also arise if a group of miners colludes to control the block production. Therefore the group of miners would control the network.

2.4 Ethereum

Blockchains are not only are useful for the creation and exchange of digital currencies. They also allow for wider possibilities.

Ethereum provides blockchain with an integrated programming language that is Turing complete, making it more powerful than the Bitcoin script language, and it allows the creation of Smart Contracts, a cryptographic box that opens only when certain conditions are reached.

The Ethereum state is a set of accounts, Each account has a 20-byte address and is composed of four fields:

1. Nonce: unique randomly generated number used to identify a transaction so that it can be processed only once;
2. Ether balance: Ethereum also provides a cryptocurrency called Ether (ETH) to pay for transaction fees;
3. Account storage: a hash of a tree root, said tree stores data of the account;
4. Account contract code (optional field), it is the hash of the code to be executed when the account receives a message.

There are two types of accounts: a **contract account**, which is controlled by its contract code and when a message is sent to it its internal code activates; an **externally owned account**, which is controlled by a private key and does not have a contract code. The subject that owns the private key can send signed messages.

Transactions are data structures signed and sent by external owned accounts. Transactions are used to start a computation specifying their maximum computational steps (**STARTGAS**) and the fee that the sender of the transaction obliges to pay per computational step (**GASPRICE**).

Transaction code is executed in a virtual environment called Ethereum Virtual Machine (EVM).

Transactions are bound to have fees for their computational processing to protect the network from abuses, miner node offers itself to execute transactions in exchange for a minimum fee value, and when a user requests a transaction he can offer a certain transaction gas-Fee value, the higher the transaction fee the sooner a transaction is executed. For the transactors, there is a trade-off between time

and lowering the gasFees. When transactions are created, a specific amount of gas called GasLimit, similar to STARTGAS, is associated with them. The gasLimit amount will be taken from the transactor balance and any unused gas at the end of the transaction will be refunded.

If an out of gas exception is thrown, the transaction will stop and the computation will be reverted. The miner still receives payment of all the fees per step. This favors the tendency for miners to prefer transactions with lower STARTGAS because lower STARTGAS means fewer computational steps.

The computational steps limit (STARTGAS) is imposed because there is no way to predict the time in which a given computation (figure 2.4) will terminate: this is correlated with the Halting Problem, which is unsolvable. Gas fees are paid for every computational step. Gas is also used for paying the fees of subordinate contracts or message calls and the increase of the memory usage for execution causes the payment of gas fees too.

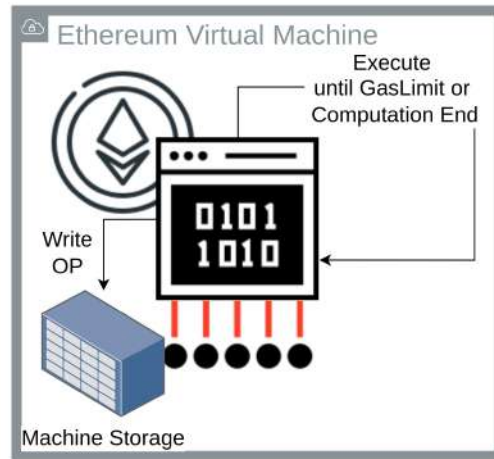


Figure 2.4: Ethereum Virtual Machine(EVM) execution example

2.4.1 Merkle Tree

Both Bitcoin and Ethereum use a Merkle tree to allow for the verification of transactions. This allows verification of transactions without having to download the Blockchain as a whole.

A Merkle tree [6] is a tree data structure using collision-free hash functions for efficient files/data integrity checks. It can be divided into a data block that contains a set of nodes and leaves of the tree. The leaf node contains the hash of a file, and each internal node stores the hash of the content of its children. As a result, the root node stores the "hash of the whole tree". Merkle trees are used to check the integrity and validity of a downloaded file. The efficiency of proving the integrity of a data block lies in not reading all the data of the tree, but instead

processing only a path called Merkle proof, from a leaf up to the root, which is logarithmic in time complexity.

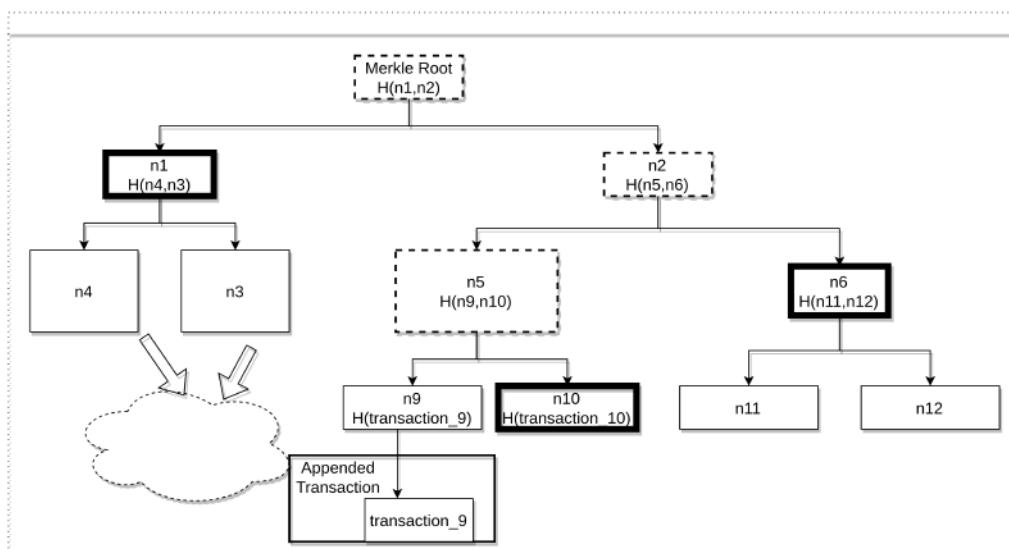


Figure 2.5: A Merkle tree. The highlighted nodes are part of the Merkle Proof, the dashed ones are the nodes which hashes will be calculated.

As seen can be seen in figure 2.5, to check the integrity of transaction_9 a user computes the hash of it, $H(\text{transaction_9})$ getting the parent leaf node (n9); the user then retrieves the sibling (n10), the node (n6), and the node (n1) hashes. Computing the hash of the concatenation of n9 and n10 hashes, $H(n9, n10)$, results in n5 hash. Using the n6 hash value and the calculated $H(n9, n10)$ the hash $H(n5, n6)$ of node n2 is calculated, then by using the n1 node hash, the root hash is calculated. If the result of this process corresponds to the Merkle tree root hash, then the transaction integrity is assured. The list of nodes (n10), (n6), and (n1) is the Merkle proof. As the hash function is a collision-free one (illustrated in section 2.1.2, no other transaction in the place of transaction_9 could have generated the same Merkle tree hash.

2.5 Self Sovereign Identity

Self-Sovereign Identity (SSI) is a concept of digital identity that shifts the identity control from centralized organizations back to the identity owner. Instead of having his own identity stored in an external organization

database, the owner of that identity stores it privately and can disclose it, or parts of it, to an external organization only when needed. This allows a subject to have full control of his data, protecting his privacy. Typically, a subject has more identities associated with more organizations. The SSI paradigm allows identities to be portable, allowing one identity to be used for accessing more than one service. Even if some services allow the portability of identities to another service (e.g., log in with 'external service' pages), the user does not have control of his identity. User identity is still stored and managed by an external organization, and also the extent to which the identity is portable is subject to the organization managing it.

One of the main principles of SSI is that the disclosure of personal user data, which should be kept at a minimum, should include only the minimum required to access the service. Identity should be persistent, but as Christopher Allen says in his 2016 article [7] *"This must not contradict a "right to be forgotten"; a user should be able to dispose of identity if he wishes.*

2.5.1 SSI and Identifiers

Archiving Decentralized SSI is a task now achievable thanks to blockchains. As described by Zooko's Triangle [8, p. 82], formulated by Bryce Wilcox-O'Hearn in his 2001 paper [9], he claims that it is impossible to design a system where identifiers can be human readable, decentralized, and secure. Human readable means that it can be understandable and memorizable by a human. Decentralized means that there should not be a central authority responsible for the identifiers. Secure means that any identifier should be unique, and an identifier should not collide with another one.

One first example of blockchains related to SSI is Namecoin [10]. After the creation of Bitcoin, an alternative coin called Namecoin was created. Namecoin allows the registration of unique names that can be used to create website addresses. Namecoin can be seen as the first bridge of blockchains with SSI, which archived unique identifiers. This can be achieved by assigning a name only to the first requester, meaning that any subsequent request will fail.

Asymmetric cryptography can be used in self-authentication schemes to allow the creation of unique key pairs that can be self-authenticating. Although these pairs are collision-free, the shortcoming of this solution

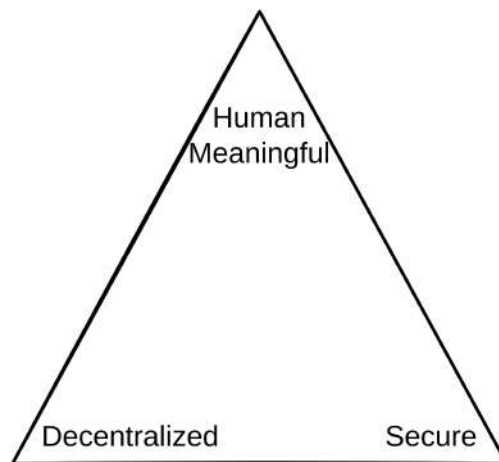


Figure 2.6: Zook Triangle

is that those values are not human readable, a central authority would then be needed to map the key pairs to a human-readable and unique name.

2.5.2 Blockchain and SSI

Blockchain could be used and is used to overcome the limitations expressed by Zook's triangle (see Figure 2.6), which states the trade-offs human-meaningful, decentralized, and secure identifiers (as stated in the previous subsection 2.5.1. Concluding that all three goals are unreachable together. Blockchain can render it possible as Blockchain allows sharing pieces of information publicly and in a safe manner.

2.5.3 Verifiable Credentials and Data Model

SSI relies on the concept of Verifiable Credentials. The main actors and the principal data structures, illustrated in figure 2.7, used in Verifiable Credentials are:

- *Claim*: A claim is a statement about a subject, a single claim could be presented individually.
- *Credential*: A credential is a set of claims and metadata, which includes their validity period and their issuer, issued to a user. A

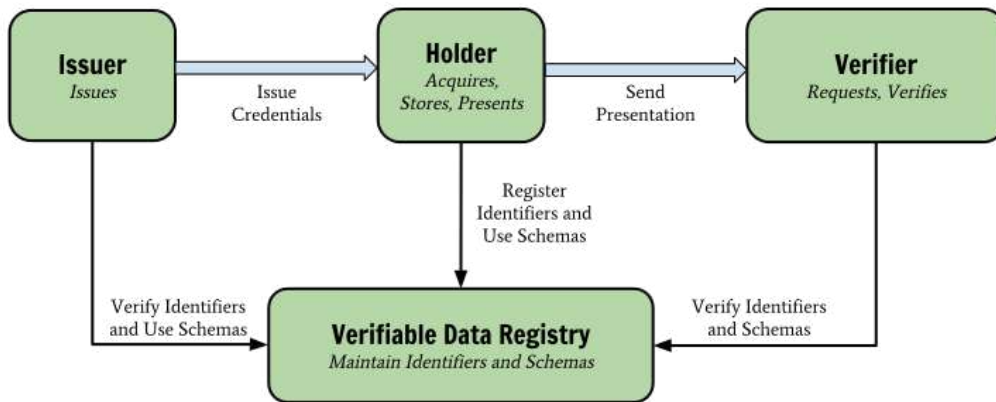


Figure 2.7: Architecture of SSI and Verifiable Credentials [11]

claim is an element of a credential (figure 2.8). A set of claims can be seen as a set of (key, value) pairs, $(attribute \rightarrow value)$.

- *Data verification schema*: is a schema that defines the structure and contents of a credential (e.g.: defines the set of claims that a credential must contain).
- *Issuer*: The entity that issues credentials.
- *Holder* (or *Prover*): The entity that stores claims on a private storage and presents them to a verifier when needed.
- *Verifier*: The entity interested in a particular claim value. The verifier checks the validity of the Verifiable Credential.

For claims to be validated there has to be a method to prove that an Issuer has issued them. This can be solved by using Digital Signatures (subsection 2.1.1).

Verifiable Credential (VC), also called Verifiable Claim in Literature, contains a set of claims, the metadata could specify the validity period for the VC and its claims, and, most importantly, the proof of its authenticity. VCs could also be revoked by specifying a set of rules for their revocation.

Verifiable Presentation (see figure 2.9) consists of one or more claims, which can belong to one or more than credentials that a holder presents

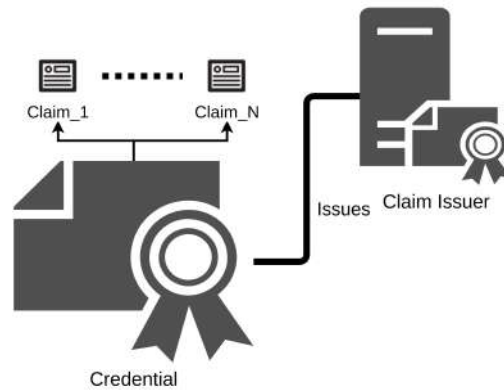


Figure 2.8: Credentials as set of claims

to a verifier, e.g. a service provider, to prove he holds a set of claims requested by the verifier. To be verifiable, a presentation contains metadata with a proof of its authenticity and also the proofs of the authenticity of all VCs inside.

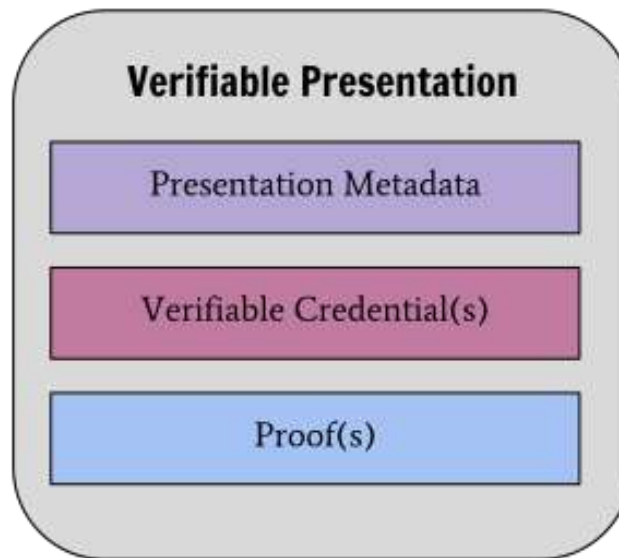


Figure 2.9: Verifiable Presentation structure

2.5.4 Selective Disclosure and Zero-knowledge proofs

To preserve his privacy, the prover could use the mechanism of Zero-knowledge proofs (ZkP) to present just a subset of his claims. A Zero-

knowledge proof is based on cryptographic schemas that allow a prover to prove that a claim has a certain value, without disclosing the value (e.g.: Alice has age>18 instead of presenting Alice age).

A claim is associated with a witness, a piece of confidential information that supports the validity of the claim, and that is kept secret by the prover. As defined by the ZKPROOF standardization initiative: *A proof system is zero-knowledge if it does not leak any information about the prover's witness beyond what the attacker may already know about the witness from other sources.* [12]

2.5.5 What goes on the blockchain?

Some data needs to be on-chain, especially data useful for authentication, such as a public key of an issuer. The W3C Verifiable Credential Working Group (W3C VCWG) data model specification strongly preserves privacy. If a certain credential is created, it is not known to the public since the event is not recorded on a blockchain.

This off-chain model is based on the assumption that the credential Issuer is trusted, and going by that assumption, it gives more privacy to the holders and allows them to write less on the blockchain. Maintaining credentials off-chain preserves the right to be forgotten principle, ..a user should be able to dispose of identity if he wishes, and claims should be modified or removed as appropriate over time. To do this requires a firm separation between identity and its claims: they can't be tied forever [7].

The on-chain model models allow credentials to be stored on-chain, but the degree to which they are made public varies, it usually depends on the type of a blockchain: permissionless, private permissioned, or public permissioned.

Types of Blockchain

- **Permissionless Blockchain:** blockchains like Ethereum and Bitcoin are permissionless blockchains, anyone can connect to them and participate in their consensus protocol and send/receive transactions. Their public nature leads to transaction fees and no privacy since all transactions are visible to the public. Participants are

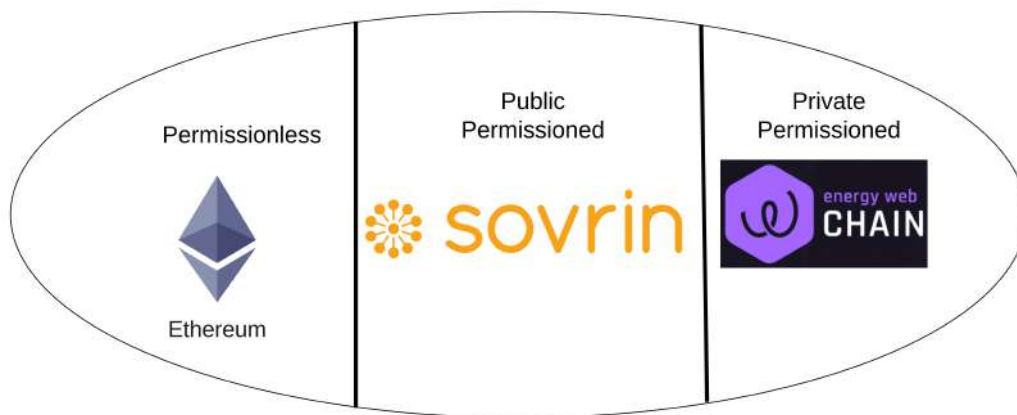


Figure 2.10: Concrete examples the three Blockchain Types

pseudonymous, meaning that compliance with regulations is difficult.

- **Private Permissioned Blockchain:** usually, there are blockchains made by an organization for one specific application, those networks are set up with a finite set of nodes. Even if they allow for greater privacy, they are not transparent and scalability can be difficult. Unlike Permissionless Blockchains, Private and Permissioned usually do not request fees per transaction, but the Service Provider can request a fixed cost for using the network.
- **Public Permissioned Blockchain:** This type of blockchain allows anyone to join, making it more scalable than a private one, usually a consortium starts the network and specifies the rules for joining it. Unlike a permissionless blockchain, every participant is identified.

Finally, hybrid types of blockchain were developed, where both a private blockchain and a public blockchain are used for doing particular transactions: a concrete example of this would be the IBM Food Trust system.

While Blockchain can be classified into three types, the most suitable ones for an SSI system are Public Permissioned networks. Their advantage is having no transaction fees, providing more privacy, and being highly scalable. Given the advantages of public-permissioned blockchain for SSI, the proposed privacy-preserving solution for deliveries is deployed in a public-permissioned blockchain: Hyperledger Indy.

Hypeledger Indy architecture and functionalities are presented in the next chapter.

Chapter 3

Hyperledger Indy, DID and credentials

This chapter describes Hyperledger Indy (<https://github.com/hyperledger/indy-sdk>). The chapter begins with an overview of Indy architecture, followed by the description of Identity and Decentralized identifiers in Indy, and concluding with a rundown of Indy Verifiable Credentials entities and operations.

3.1 Hyperledger-Indy and its history

Hyperledger Indy offers an ecosystem of libraries, tools, and frameworks based on SSI, but more importantly *"Hyperledger Indy is a special purpose blockchain implemented specifically for Identity on the Internet — enabling certainty about who is talking to whom in a digital transaction"* [13].

Indy is an Open Source Project maintained by the Hyperledger Foundation (which was started by the Linux Foundation). One of Hyperledger's main tenets is to create blockchains and frameworks suitable for businesses specifically designed for them, and not based around cryptocurrency. Indy codebase was created by the company Evernym [14] and then it was donated to the Sovrin Foundation [15]. The Sovrin Foundation deployed the Indy code on various nodes, called the Sovrin Network, thus creating the first and biggest public permissioned blockchain (introduced in section 2.5.5) instance of Indy. As of now, the Sovrin Foundation still supports the Sovrin Network and contributes to the code of Indy. Sovrin offers three networks: BuilderNet, StagingNet, and MainNet.

Respectively for testing, prototyping, and deploying an SSI system. BuilderNet does not have any fees per transaction and the process to obtain an Endorser DID is fairly simple. A short guide for obtaining an Endorser DID for the BuilderNet is in Appendix A.5.

At the core of Indy codebase there are two major components, which can be found on Hyperedgers' GitHub:

- Indy-Node: which contains the code used by a node [16].
- Indy-SDK: which provides client-side code whose main component is libindy: *"The major artifact of the SDK is a C-callable library that provides the basic building blocks for the creation of applications on the top of Hyperledger Indy"* [17].

3.2 Indy Architecture

Indy is a permissioned blockchain whose set of nodes that composes its network is called **pool**. A pool is built starting with a genesis transaction, which sets up the pool state and the p2p network configuration.

As of now, the term *ledger* will be preferred when discussing the Indy blockchain. As said in section 2.2, a blockchain is a type of distributed ledger, and the majority of Indy documentation utilizes the term ledger. This can be attributed to the fact that, unlike Bitcoin/Ethereum, in Indy *the block size is equal to one transaction* [18].

3.2.1 Indy ledger(s) and nodes

Indy does not have just one ledger, instead it has four different ledgers (three can be seen in figure 3.1) each storing a specific type of transaction:

- Domain ledger: It stores transactions relative to the application domain, e.g., schemas, information about identities, or credentials (see Section 3.5).
- Pool ledger: contains transactions relative to the nodes composing the pool. I.e. transactions used for the operations of adding a node to the pool or to change one node state/configuration.

- **Config ledger:** It contains information about the requirements to approve transactions at a given moment, and the configuration of a pool, for example, if the pool is set in read-only mode then all incoming transactions will be discarded.
- **Audit ledger:** This ledger aggregates all the other three ledger transactions sequentially, based on the order of arrival. Audit ledger aids in keeping the other three ledgers synchronized [19].

To lighten the description of the operations, we will use the term ledger without specifying the type unless when strictly needed. One ledger has two main components:

- **Transaction Log:** a sequence of key-value pairs (transaction number, serialized transaction).
- **Merkle Tree:** a tree data structure used to store hashes of transactions.

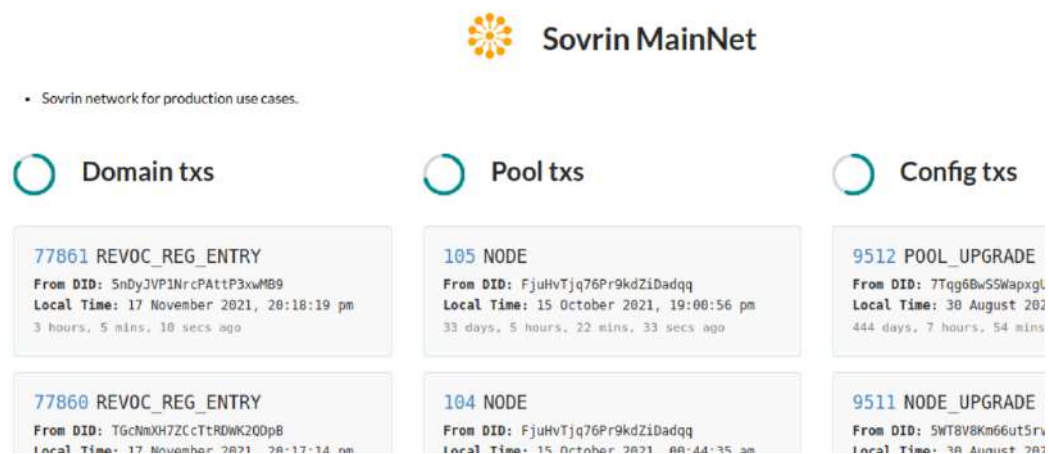


Figure 3.1: A snapshot of the Sovrin MainNet divided into different ledgers storing the corresponding transactions (txs)

3.2.2 Transactions and Ledgers

All transactions are stored on a ledger are serialized in MsgPack format. MSGPack [20] is a data-interchange format similar to JSON, both have the same structure of a key-value map. A key is a string and the mapped value can be a string, an integer, a boolean, an array, or another map.

MsgPack format is lighter in weight than the JSON format, it can encode an equal length map in fewer bytes.

3.2.3 Nodes and ledgers (Indy-plenum)

The ledger implementation is part of the repository Indy-plenum [21], where Indy-plenum contains the same code as Indy-node but only regarding ledger and consensus protocol implementation. All nodes store a copy of the ledger but not all nodes run the consensus protocol. Nodes can be divided into:

- Validator Nodes: nodes who participate in the consensus protocol to validate transactions.
- Observers Nodes: nodes that maintain a copy of the ledger. The copies are consistent with those of the validator node. They are used to offset the workload of validators.

Each node stores a seed (illustrated in subsection 2.1.2) to generate keys and a verification key (verkey), equivalent to a public key, which can be used to validate node-to-node and node-to-client messages. The verkey is an ED25519 key, used in an ED25519 public-key signature system (a public-key signature system based on elliptic curves [22]). A node also has a Boneh–Lynn–Shacham (BLS) key to sign transactions after validating them [23].

Plenum Byzantine Fault Tolerance

Indy has a different approach for transaction validation than Ethereum and Bitcoin. Indy does not use intensive resource-consuming protocols to validate a transaction, but a protocol called Plenum Byzantine Fault Tolerance (PBFT) [24], which is an alteration of Redundant Byzantine Fault Tolerance (RBFT). The RBFT protocol is derived from Practical Byzantine Fault Tolerance[25] which is the most widely known protocol that addresses the Byzantine generals problem. Byzantine Fault Tolerance addresses the problem of malicious nodes approving invalid transactions: the Byzantine generals problem is shown in figure 3.2. The protocol is leader based, it is said that Indy uses a *leader-based byzantine fault-tolerant protocol for ordering transactions - agreeing upon global order between them on all non-faulty nodes, so that nodes will always write*



Figure 3.2: Figure representing the Byzantine generals problem. There are two questions posed by the problem. Can the generals agree on attacking the city? Can the message of agreement sent by one general be trusted? Figure from [26].

them in the same order into their ledgers [27]. The leader decides the order of the transactions and communicates them to all other nodes. The nodes, together with the leader, decide if a transaction is valid or not.

As said in [28] RBFT is: *a new approach to designing robust BFT protocols. In RBFT, multiple instances of a Byzantine Fault Tolerance (BFT) protocol are executed in parallel. Each instance has a primary replica. The various primary replicas are all executed on different machines.*

A transaction to be validated has to have the consensus of $N-f$ nodes with $N=3f+1$. Where f represents the maximum number of malicious nodes allowed to accept transactions nonetheless. If the number of malicious nodes is higher than f the pool stops accepting transactions even if valid. In Indy there are multiple instances of PBFT protocol being executed, each node runs multiple instances of it in parallel. PBFT has some implementation-specific additions to the RBFT protocol. The Master Instance is the one used for the ordering of transactions. Other (backup primaries) instances are used as performance measures. The backup primaries instances have different leaders. If the Master Instance leader (master primary) is too slow, Master Instance is switched to another. The instances run are $f+1$ [26]. As an example, two instances can be seen in figure 3.3. Instead, figure 3.4 shows that a write request needs the consensus of the nodes. The nodes exchange a message for every one of the $f+1$ instances since each instance approves the request inde-

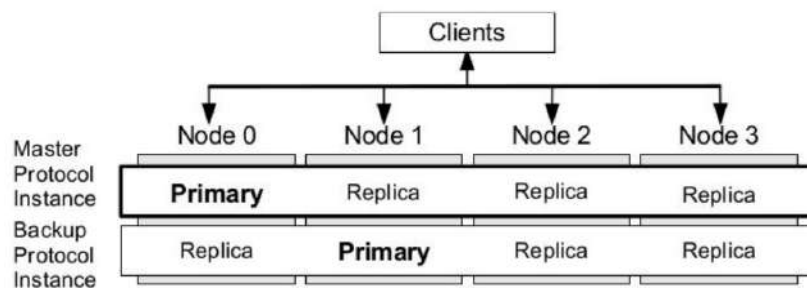


Figure 3.3: Master instance and Backup instance of PBFT. They run in parallel within the nodes. From [26]

pendently [26].

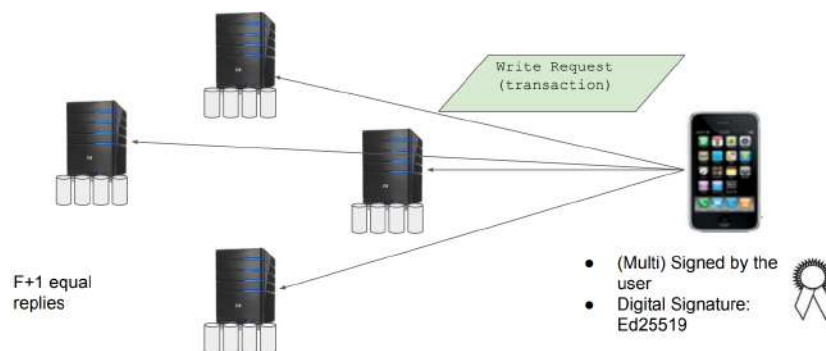


Figure 3.4: Nodes message exchanged following a write request. From [26]

In Indy anyone can read from the ledger, consensus is only needed to write on the ledger. Consensus is not needed for a read request. A node (observer or validator) retrieves the transaction from its Merkle tree and returns it to the requested.

3.3 Identity in Indy: Agent - Identity Owner

An **agent** is a bridge between a user and a SSI system. Generally speaking, an agent is an entity that can be either a person or software and is trusted by an identity owner. An identity owner is someone legally responsible for their actions, it can be a person or an organization.

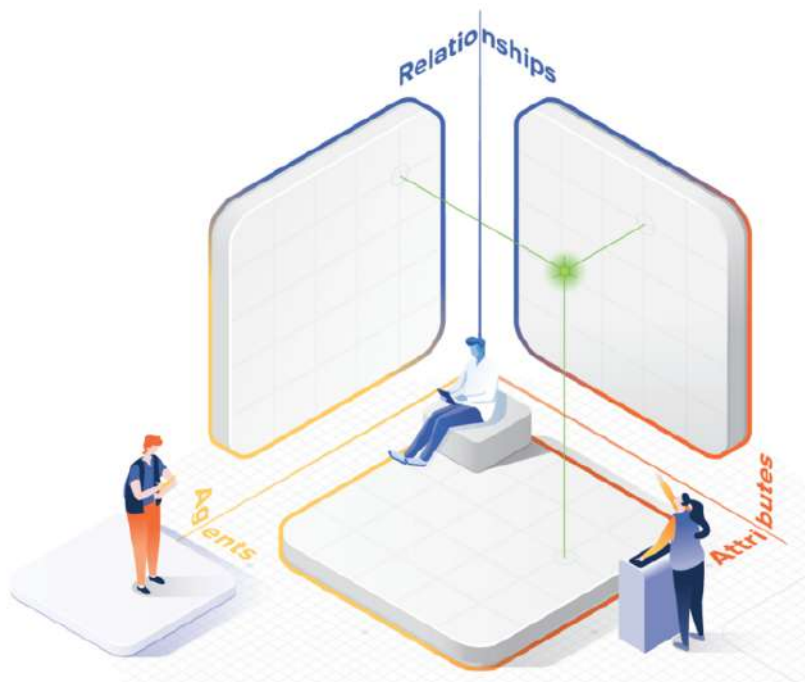


Figure 3.5: Identity as a three-plane, a user shares a set of his attributes to an external subject, using as a proxy a specific agent [29].

Identity Owners delegate their identities to agents, or to put things in perspective, they delegate pieces of their identities to different agents. For example, an agent is a software application in the digital identities context, which can run in different hardware, such as PCs and smartphones, which store different fragments of an Identity Owner identity. An agent stores the identifiers, credentials, and keys of an Identity Owner. The agent has all the functionalities needed to work with bytes, store and manage data, and perform the required cryptographic operations for SSI. Identity, as described in [29], can be seen as a three-way plane, which poses the questions on what an identity owner shares about his identity:

- The *agent-attribute* plane represents the level at which an agent can share attributes about an identity owner. It can be seen as the center plane of figure 3.5.
- The *relationship-attribute* plane indicates how a third party sees the identity owner. The upper-right plane of figure 3.5 encompasses it. For example, while an identity owner is seen by his

company as an employee, a web store sees him as his customer. The relationship-attribute plane also represents the attributes that an identity owner decides to share with an external subject. It has to be noted that if an identity owner uses the same identifier when communicating with different subjects, his identity can be greatly exposed. Attributes the identity owner shares with a subject-A (e.g., government) can be correlated with other attributes shared with subject-B (e.g., bank).

- The *relationship-agent* plane represents the choice of which agent can represent an identity owner in a given situation. Assuming an identity owner has multiple devices (agents), it can delegate specific operations related to their identity to dedicated devices, e.g., buy shares only through smartphones and access public services only through PCs. An identity owner can use different agents and decide how much about his identity it can share and with whom. It represents the upper-left plane of figure 3.5.

An agent in Indy consists of a software using the methods provided by libindy to manage Identity Owner attributes data and provide a bridge between the Identity Owner and the Indy system. An Indy agent can be implemented with the Rust programming language or with the wrappers provided by Hyperledger: Java, Python, iOS, NodeJS, and .Net. All the wrappers call the underlying libindy methods written in Rust. It has to be noted that an Android library for libindy is also provided by Sovrin. We will use the Java wrapper as a reference and for one Indy agent implementation.

Indy agents can have various forms and run on different types of hardware. Also, some existing agents are cloud-based and accessed through a local agent. Agents are not only those managing an Identity Owner's identity but also the validator nodes are agents of the user deploying them [30]. A collection of agents can be found at: <https://github.com/hyperledger-archives/Indy-agent>.

3.4 Identifiers in Indy: Decentralized Identifiers

Decentralized Identifiers (DIDs) are identifiers representing an identity. A DID is a URI divided into three parts: the DID scheme, DID Method,

and a unique method-specific identifier, each separated by a colon (:), i.e. `did:<did-method>:<unique-id>`. An example of DID is shown in figure 3.6, while an example of a Sovrin DID is:

`did:sov:HR6vs6GEZ8rHaVgjj2WodM`

Sovrin DIDs will be discussed in 3.4.2. The scheme part, in this case, is

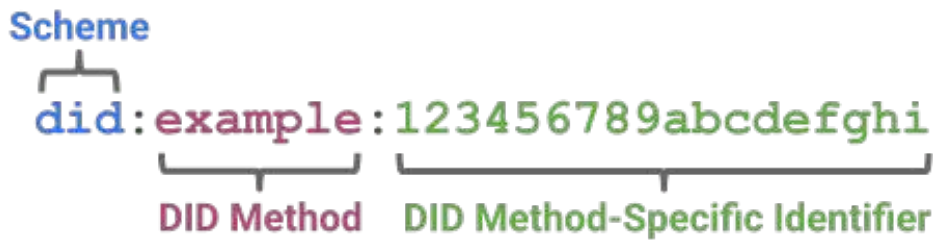


Figure 3.6: DID as string format, as represented by W3C [31]

the string "did". The DID Method specifies the type, or implementation, of a DID, meaning different types of DIDs have different rules regarding their creation and management. The DID method-specific identifier represents a unique value of the DID method type. As per documentation, a DID uniquely refers to a subject, called **DID subject**, who can be a person, an organization, and, differently than an Identity Owner, can also be an object. Usually, a DID subject is not bound to a specific thing, but it can change, i.e. it could be used to represent a role in an organization, and the person behind the role could change.

A DID subject has one or more **DID controllers**, those who can act on behalf of the DID subject. DID controllers are necessary when the subject of the DID is an inanimate object, as said in [32]: *"Typically, the Subject is also the Controller, but in cases of guardianship, agents (human or software), and inanimate Subjects, this is not possible. As such, the Subject has no functional role."* It has to be said that a DID controller could also be an interactive system. If a DID controller is a physical person or an organization then, in this case, the DID controller definition and the Identity Owner definition can be seen as the same. A DID controller of a DID Subject can be the DID Subject himself or another Identity Owner.

Each DID is associated with a **DID Document**, which can be modified only by the DID controllers. A DID Document is a map of `<key, value>`

pairs. The W3C DID specification does define a data model, which represents the entries or a DID Document. Figure 3.7 shows the entries of a DID Document.

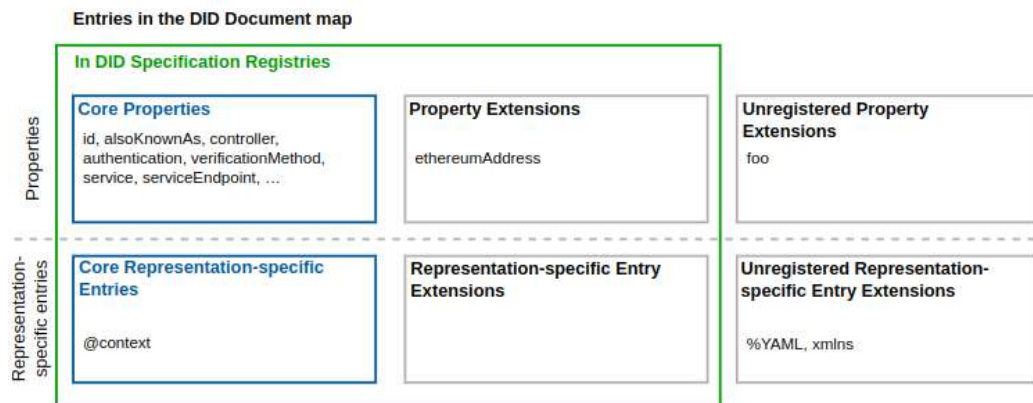


Figure 3.7: DID document entries as defined by the DID Specification Registries [33]: which defines global parameters, properties, and values used by the Decentralized Identifier ecosystem. As can be seen, fields Core properties and extensions structure depend on the DID method implementation.

Core properties [31] represent a collection of properties inside the DID Document. Each entry represents an attribute of the DID subject, where the key represents the relationship between the DID subject and the property "id". The property "id" defines the DID of the DID subject. A DID method specification to conform to the w3 DID specification has the requirement to fit the entry **id** in the Core Propriety section. The Core properties section also can contain the entry **controller**, containing a string or a set of strings, representing one or more DIDs of DID controllers.

A DID Document can be obtained from a DID by a process called **DID resolution** performed by DID resolvers. An example of a DID resolver would be the Universal Resolver¹. The DID resolution process is defined by the DID Method [34], meaning the resolution of a Sovrin DID can be different from the resolution of an Ethereum DID.

¹Universal DID Resolver <https://dev.uniresolver.io/>

Creating a DID in Indy

Indy allows an agent to create an ED25519 private/public key pair and store it in a **Wallet**. The keys are used for signatures and authentication of operations done with the said DID. Indy method for DID creation (*createAndStoreMyDid*) creates a DID that can be derived from a seed, which makes deterministic the DID creation. If the agent creating the DID does not specify a DID seed, a random seed will be used to generate the DID. A DID generated with Indy can be used on the Sovrin Network. It has to be noted that various DID-method specifications define different processes for DID creation. A DID that is created with Indy is composed of only the DID Method-Specific Identifier part of a DID string (figure 3.6), meaning this kind of DID is in an unqualified form. Indy distinguishes between unqualified identifiers and Fully-Qualified identifiers. Indy can work with DIDs from different networks, and it is possible to translate an unqualified DID into a Fully-Qualified one with a specific DID Method [35]. An example of an unqualified DID generated by Indy would be a string: EUw5BvUagdrYdpKh6DJCQ.

3.4.1 NYM

In Indy, each public attribute (part of the DID Document) of the identity of an Identity Owner is contained within a NYM Record. For the Indy DID Method specification, a NYM is defined as *A NYM (short for “Verinym”) is associated with the Legal Identity of an Identity Owner and is a Hyperledger Indy specific term for a data object, which holds DID data of one concrete identity returned during DID resolution [36]*. The creation of a NYM record is done with a transaction, called NYM transaction [18], which is stored in the Domain ledger. NYM transactions serve two purposes: the creation of a NYM record and changes to a NYM record. Changes to a NYM record are modifications of the fields defined by previous NYM transactions related to it.

An NYM transaction is composed of the following fields:

- *dest*: which is the DID of the entity represented by this NYM transaction;
- *role*: the role of the NYM record, different roles allow for different capabilities to the **NYM owner**, i.e. the owner of the DID in the NYM record;

- *verkey*: verkey used by the NYM owner to sign transactions or messages in a communication;
- *alias*: it is an optional field, containing a DID of a different NYM record.

The creation of a NYM record can only be done by agents who have a DID already stored as a NYM record on the Domain ledger. The initial NYM records stored in the ledger are generated by the genesis transaction mentioned in section 3.2. An agent that desires to have their DID stored in the ledger has to request another agent with a NYM record on the ledger. The latter will publish it on behalf of the requester. The agent that owns the NYM is a DID controller as it acts on behalf of the DID subject represented by the DID.

Figure 3.8 shows a NYM transaction in an Indy network executed on a local machine. Transactions of a network can be explored with Indyscan [37]. As can be seen, the *dest* and *from* fields of the transaction contain two different DIDs: the requester and the submitter agents, respectively. It is possible to see the transaction from Indyscan in two formats.

- Enriched data format: the transaction type is inferred by the structure of the transaction. The information shown is only relevant to the specific transaction type;
- Original ledger data: the transaction is stored and read from the ledger.

The different roles in Indy derive from the fact that it is a permissioned blockchain. Each role allows for different capabilities.

As stated in subsection 3.2.3, a consensus on the transaction approval is a requirement for the transaction to be written on the ledger, then to reach a consensus that the transaction submitter signature has to be validated. A valid transaction must be submitted from a NYM owner owning a NYM with an assigned role that allows for the specific type of transaction to be made. The verkey of DID stored in the NYM, which is public and stored in the NYM, is then used from the nodes to verify the transaction. As stated before, NYM transactions are not just for creating a NYM record, but also allow the NYM record owner to modify it. A NYM record verkey could be changed with an operation called key rotation, which is advised to be done periodically and could be useful in case the key secrecy has been compromised [38].

```

{
  "txn": {
    "data": {
      "dest": "3ZPSazYSNGiKP2wcX1t7rt",
      "role": "101",
      "roleAction": "SET_ENDORSER",
      "verkey": "2Psoa7x2pshw7yhkeLJV1hBtQABoEZEeyeBYK57LqFWez",
      "verkeyFull": "2Psoa7x2pshw7yhkeLJV1hBtQABoEZEeyeBYK57LqFWez"
    },
    "metadata": {
      "digest": "6e88ab45a0d1f8bbb32d2137cf6fd37ac848e66fc6a5d5a3e6c79ca20a06f8",
      "from": "Th7MpTaRZVRYnPiabds81Y",
      "payloadDigest": "2c355570ccaf19fe8e7169c70e727f744ba0d665b1381045739a06fd04ed9f4f",
      "reqId": "1639956842834336300"
    },
    "protocolVersion": 2,
    "type": "1",
    "typeName": "NYM"
  },
  "txnMetadata": {
    "seqNo": 193,
    "txnId": "e76653e874583149d510d6bca30c98d6e6855df6fd49f9534f5ef1361292e8b3",
    "txnTime": "2021-12-19T23:34:02.000Z"
  }
}

```

Figure 3.8: This transaction (shown in enriched data format) creates a NYM for with DID dest 3ZPSazYSNGiKP2wcX1t7rt, the role 101 is the role of an Endorser (described in subsection 3.4.3).

A NYM encapsulates a DID and its associated verkey. The verkey is part of an ED25519 key pair, where the signing key is used to sign transactions and/or messages, and the verkey to verify the signature (using a digital signature schema introduced in subsection 2.1.1). The signature verification is also done by nodes, using the verkey, to verify a transaction and reach a consensus (subsection 3.2.3).

ATTRIB TRANSACTION are transactions that enrich a particular NYM transaction with an attribute. The transaction itself can be performed **only** by the **NYM owner**.

The main fields for ATTRIB TRANSACTION are:

- *dest*: the DID associated with the target NYM;
- *raw* OR *hash* OR *enc*: the second field can be one of those three; they are mutually exclusive. A *raw* field represents a JSON map, where a key would be the attribute name, and the value associated with the key is the attribute in string form as raw data. The *hash* field would be the hash of the raw data, the real attribute raw value

is stored in an attribute store. The *enc* field contains the hash of an attribute value that is not stored in an attribute store [18].

Figure 3.9 shows an ATTRIB transaction in the Sovrin MainNet and obtained from Indyscan.io².

```
{
  "txn": {
    "data": {
      "dest": "V17e1FG9pQFbG6hURqQ1Jj",
      "endpoint": "https://aud-dev-idcrypt.rtgs.global/didcomm",
      "raw": "{\\\"endpoint\\\":{\\\"endpoint\\\":\\\"https://aud-dev-idcrypt.rtgs.global/didcomm\\\"}}}"
    },
    "metadata": {
      "digest": "e5519f996037890b2ac35eac444886c4913708cb6e5c58d5477bddafd4b792b0",
      "from": "V17e1FG9pQFbG6hURqQ1Jj",
      "payloadDigest": "d79556d46d1220f1504edd1d6cc37e14a78d202f95b2583a7bdf8043917ec31d",
      "reqId": 1640498285923317000,
      "taaAcceptance": {
        "mechanism": "product_eula",
        "taaDigest": "8cee5d7a573e4893b08ff53a0761a22a1607df3b3fcd7e75b98696c92879641f",
        "time": 1637625600
      }
    },
    "protocolVersion": 2,
    "type": "100",
    "typeName": "ATTRIB"
  },
  "txnMetadata": {
    "seqNo": 79020,
    "txnId": "V17e1FG9pQFbG6hURqQ1Jj:1:b6bf7bc8d96f3ea9d132c83b3da8e7760e420138485657372db4d6a981d3fd9e",
    "txnTime": "2021-12-26T05:58:06.000Z"
  }
}
```

Figure 3.9: The transaction *from* and *dest* fields contain the same DID. It means that it is the NYM owner who is adding an attribute to it. The field *raw* contains a string that defines a service endpoint

3.4.2 DID in Indy and Sovrin

As of the end of 2021/the start of 2022, Hyperledger Indy does not have a DID Method implementation. However, a specification of what an Indy DID Method should be has been proposed [36].

Sovrin DIDs

Sovrin has a working DID Method Specification, as drafted in [39]. Each Sovrin DID conforms to the structure *did:sov:identifier*, where the identifier is a 16-byte string. The identifier string is usually composed of the first 16 bytes of the verkey associated with the DID or derived from

²https://indyscan.io/home/SOVRIN_MAINNET

it. As per Sovrin DID specification, every DID has a 256-bit ED25519 verkey. *All public DIDs and their DID Documents (not in an explicit form) are stored in the ledger.*

The ledger can be queried to obtain a DID Document from the ledger DID-related transactions. The Universal Resolver, as mentioned at the start of this section 3.4, can produce a DID Document from a Sovrin DID. An example of a DID Document, resolved using `did:sov:HR6vs6GEZ8rHaVgJg2WodM` is present at Appendix A.1. The two major components of the DID Document above are:

- **type:** indicates how will this DID will be used, for example, DID-Comm indicates that it will be used in communication from agent to agent (which will be introduced in subsection 3.4.4);
- **service endpoint:** it is a URL to the service endpoint of the DID Subject, e.g.: the address to connect to start communicating with the DID Subject, or a website address. An example of an endpoint is `https://bc-reg-agent.orgbook.gov.bc.ca`. A service endpoint can be added with an ATTRIB transaction, as shown in figure 3.9.

The DID Document is not explicitly stored in the ledger, but it is constructed during the DID resolution from the NYM records and the attributes from ATTRIB transactions associated with them. Those can be obtained from the ledger by anyone who can connect to the pool, using the read requests GET_NYM and GET_ATTRIB. These requests are used in the DID resolution process for a Sovrin DID. The resolution process of the Sovrin DID Method specification can be synthesized from the CRUD Operation Definitions section of the specific [39][5. CRUD Operation Definitions]:

1. Read NYM with GET_NYM request;
2. Read the DID endpoint with a GET_ATTRIB request.

Indy DID Method Specification

While the specification is still in a draft state, it poses those interesting goals [36]:

1. creating a network of Indy networks (Sovrin and others). The DID method-specific identifier is designed to account for different networks. The DID string is designed to have the following structure:

did:indy:<namespace>:<namespace identifier>.

The field namespace specifies the network instance to which the DID belongs, e.g.:

<did>:<indy>:<sov>:<HR6vs6GEZ8rHaVgJg2WodM>;

2. ability to connect to the network identified by the namespace field, by having a way of retrieving their genesis transaction file.
3. self-certifying DIDs. A DID should be derived from the first 16 bytes of their ED25519 verkey, which is said to be unique, as it derives from a random seed. The ledger should enforce the association of verkey with the DID;
4. the DID Document should be obtained only from an NYM record, using the NYM 'from' field value, its verkey, and the namespace of the Indy network instance. The resulting DID document should be a valid JSON file and should pass a verification process, which has not yet been defined.

3.4.3 Roles

As stated in section 3.4.1, a NYM transaction contains a specific role that is given to the corresponding NYM record. An agent that wants to perform a certain operation has to own the NYM with a role that is allowed to do that operation. Owning an NYM record means having the private key associated with such NYM. There are four roles defined in Indy, each identified by a specific number:

- “0” (TRUSTEE): Trustee is the highest-ranking node, which manages the nodes pool. This role allows the addition of new nodes by creating a NYM for a Steward. A Trustee role is the only one that is allowed to write on the Config Ledger, and to perform transactions, such as POOL_CONFIG, which can set the status of the ledger in read_mode. The Trustee can be preconfigured on the ledger with his NYM generated by the genesis transaction. A preconfigured Trustee NYM can be obtained by generating the corresponding DID of that NYM. To generate a specific DID, it has to be known the seed from which it derives.
- “2” (STEWARDS): Like Trustees, Stewards are preconfigured on the ledger. A Steward manages a node that participates in the pool,

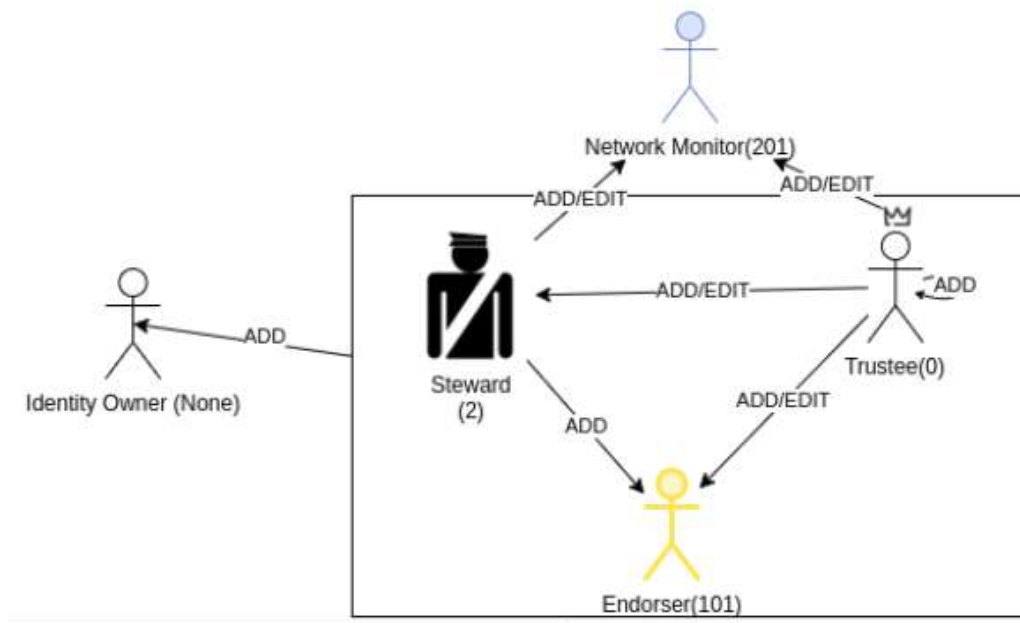


Figure 3.10: An agent with any role can add a NYM record with role `<none>`. Trustees and Stewards can add Endorsers, while Trustees can edit NYM roles and set them to `<none>` or another role, demoting the NYM. Network Monitor can be added by Stewards and Trustees, and they can both edit its role and demote the NYM to `<none>`.

and as such, it has to be approved by a Trustee. A Steward can create a NYM for an entity, such as an Endorser.

- "101" (ENDORSER): The central actor of the Verifiable Credentials system in Indy. It can create credentials and sign them to make them valid, thus corresponding to an SSI Issuer (mentioned in subsection 2.5.3). In previous versions of Indy, the term Trust Anchor was used. Sovrin defines in its glossary a Trust Anchor as *An Issuer who is considered by a Verifier or a Governance Authority to be authoritative for a particular set of Claims or Credentials. A Trust Anchor may be: a) informally recognized as a Trust Anchor by one or more Verifiers, b) formally designated as a Trust Anchor by a Governance Authority, or c) Accredited as a Trust Anchor by an Accreditation authority* [40]. An example of a Trust Anchor could be a University in regards to a graduate diploma. A company can be a Trust Anchor in regards to the salary of a worker towards a bank.

- "201" (NETWORK_MONITOR): This role was created to allow a different agent than a Steward and a Trustee to collect metrics about a node performance. A node like that allows having an agent with fewer responsibilities and capabilities than the aforementioned to monitor the network performance. The performance of a node can be obtained by the node by a VALIDATOR_INFO transaction.
- <none>: In this case, an Identity Owner can have a DID and verkey registered in the ledger, such a case could be useful to register a service endpoint with an ATTRIB_Transaction that adds it on the NYM.

The operations that a certain agent with a NYM can do are defined in a key-value dictionary called AUTH_MAP. The AUTH_MAP can be changed by performing an AUTH_RULE request (which corresponds to a transaction). The type of operations that can be done cannot be changed, but the roles required to perform the existing ones can be changed. We will refer only to the default AUTH_MAP defined by Indy [41], where only a Trustee can perform an AUTH_RULE request.

Even if not having a role on the ledger, an agent owning a DID with a NYM in the ledger and the role <none> can write on the ledger. But only for transactions regarding its NYM (NYM and ATTRIB). For NYM transactions, the NYM created is limited to certain roles, depending on who made the transaction. E.g.: an Endorser can only be added to the ledger by a Steward or a Trustee. Figure 3.10 illustrates auth rules regarding NYM transactions role assignment.

3.4.4 DID to DID Communication and Pairwise DID

As stated before, to store a NYM record on the ledger, an agent without a NYM record in the ledger has to contact another (Steward/Trustee) agent to record it. All agent-to-agent communication takes place outside the boundaries of the Indy system. The need to have a secure channel of communication, where an agent can be identified but at the same time remain anonymous, is fulfilled by **Pairwise DIDs**.

The use of DIDs for communication allows both parties to encrypt and authenticate the messages exchanged. A sender can use the private key associated with their DID to sign the message, while a receiver could use the verkey associated with the DID to verify its authenticity (it corresponds to a digital signature). The DIDs being used in a message

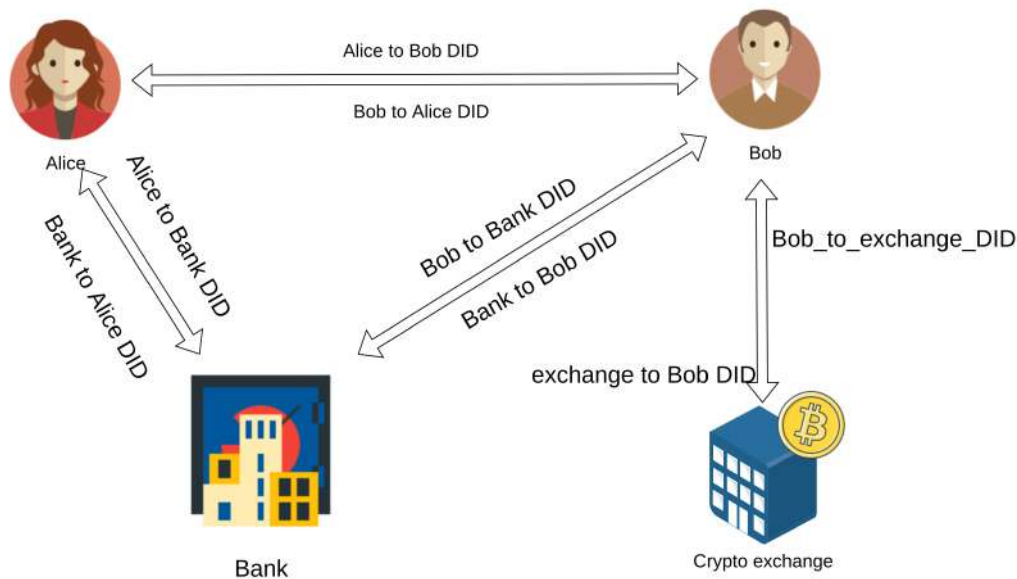


Figure 3.11: All the subjects in this figure communicate using pairwise DIDs as identifiers. Bob can be identified by a cryptocurrency exchange and can communicate with it using a specific DID. Bob’s bank knows only the DID he uses in their communication, so Bob’s identity in relationship with the bank is separated from Bob’s identity in relationship with the crypto exchange. Bob’s bank cannot correlate Bob’s identities.

exchange are called private DIDs, which are not stored on the ledger. This reduces the load on the ledger since fewer data has to be stored in it.

As said in section [42]: *private DIDs are the default in the private relationship between two identity holders, where no one else needs to even know about the relationship, let alone the two DIDs being used.*

The Pairwise Pseudonymous DID also provides an additional layer of privacy, since a user can use one DID for every message exchange he has, making it impossible to correlate his messages and uncover his identity. For conciseness purposes, Pairwise Pseudonymous DIDs will be referred to as Pairwise DIDs. Agents can use a DID for more than one DID to DID communication. Agents with a public NYM record, such as an Endorser or a Steward, could only use DID, the DID inside their public NYM record. Since the identity of such agents is revealed in their operations. For example, a Steward performing a NYM transaction will reveal its DID in the ledger. It is not convenient to hide the Steward DID

from the agent asking for a role.

Such communications store the data in **microledgers** shared between the two nodes involved in the communication. An example of a Pairwise DID being used in a different context can be seen in figure 3.11.

Indy offers a method, *createPairwise*, to link DIDs for pairwise communication. For a DID to DID communication an agent has to create a new DID and link it in the Wallet with the external agent that received the DID. Indy also offers a method, *storeTheirDid*, used to save in the agent Wallet the external agent DID alongside its verkey. In case of a key

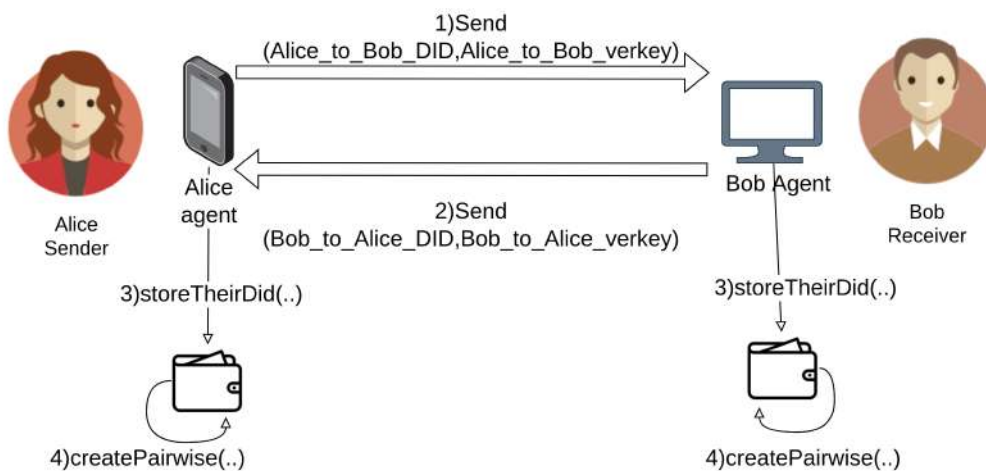


Figure 3.12: Alice and Bob (agents) exchange in a secure channel their pairwise DIDs (1), from the perspective of Alice agent: it stores Bob_to_Alice_DID and the associated verkey (2), and finally it stores Bob DID (3) and creates a link in the Wallet between the two DIDs (4)

rotation, the stored verkey can be updated with the new value (by calling again *storeTheirDid*). Indy does not only offers the cryptographic means to sign and validate messages (*cryptoSign* and *cryptoVerify*), but it also supports the encryption and decryption of a message using asymmetric key cryptography (introduced in section 2.1). The setup of DID to DID communication can be seen in the figure 3.12.

Originally, messages could have been encrypted and decrypted using the methods *authCrypt* and *authDecrypt* (or *anonCrypt* and *anonDecrypt*). With an *authCrypt* operation, the sender uses his **sender verkey**, which is the public key, as the **key for encryption**, also the sender creates a **shared secret key** using the receiver verkey. The shared secret key can be used by the receiver to check the **integrity** of a message. The re-

ceiver can recreate the **shared secret key**, by using the **sender verkey** and his **private key**. The parties' respective verkeys were exchanged and stored during DID to DID communication setup. Private keys were also created during the DID to DID communication setup and during the DIDs creation.

After the communication is set up, the other party verkey can be retrieved from Wallet (with the method `keyForDid()`). The message is then

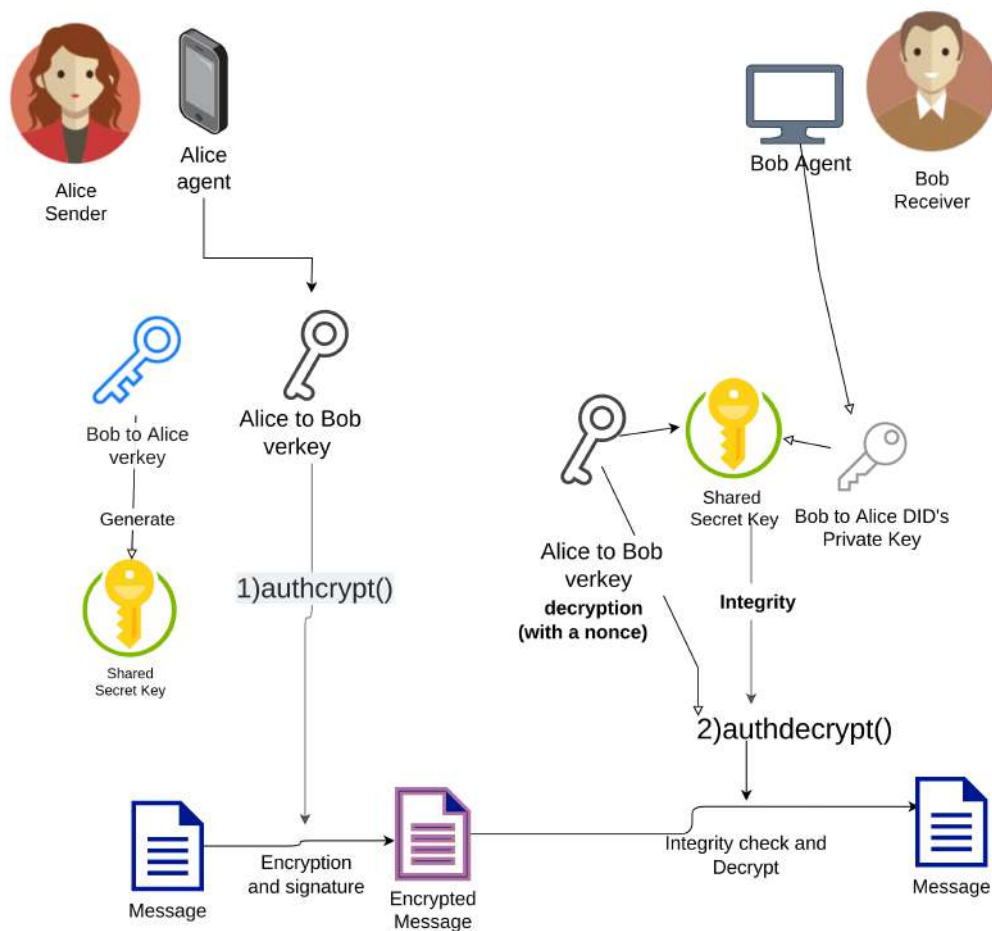


Figure 3.13: Alice uses her verkey to **encrypt** the message. Bob retrieves the pairwise DID with Alice and the associated private key. The secret key is used to validate the integrity of the message and its authenticity. Alice's verkey key and a **nonce** are used by Bob's agent to decrypt the message.

decrypted using the **sender verkey** and a **nonce** (one time use random

value)[43][Line 436] that the parties exchanged. The keys mentioned are all associated with the agent pairwise DID. The documentation says that the value of the nonce is not bound to be confidential [43][Line 436]. The creation of the shared secret key is illustrated in figure 3.13.

Hyperledger documentation highlights the effects of an authcrypt operation: *'auth_crypt(msg, recipient_pubkey, sender_privkey)' – Encrypt only for recipient, achieving confidentiality. Recipient learns sender's pubkey but can't prove to anybody else who the sender is (making the message repudiable). Parties may have had no prior contact, though the sender must discover the recipient's pubkey. The message is tamper evident.* [44][Encryption section].

The methods illustrated above are set to be deprecated. The new methods to use are `pack()` and `unpack()` supporting two different modes: authcrypt mode and anoncrypt mode. In **authcrypt** mode, the sender verkey is inserted into the message in a blinded form. Instead, the sender verkey is absent from the message in **anoncrypt** mode, and the sender's identity is unknown to the receiver.

With the `pack()/unpack()` methods, a Content-Encryption Key (CEK) is used to encrypt and decrypt a message (symmetric encryption cryptography). The CEK and the message are encrypted with the receiver's public key and sent to the receiver [45][section Authcrypt pack algorithm].

Note that the documentation cited considers a multi-agent context, where two Identity Owners communicate through a series of different agents.

A part of the same process of encryption and decryption of the `pack()` and `unpack()` method corresponds to the asymmetric key cryptography (as in figure 3.14). This process is performed to encrypt and decrypt the CEK and the verkey of the sender, while the whole process incorporates symmetric key cryptography where the CEK is used to decrypt the encrypted message, as seen in figure 3.14. The receiver retrieves from the Wallet the associated private key to the message encryption key (the receiver verkey). The `unpack` method then allows revealing the encrypted CEK. Using the revealed CEK, it is possible to decrypt the original message.

```
{
  "protected": "eyJlbmMiOiJ4Y2....." ,
  "iv": "M1GneQLepxfDbios",
  "ciphertext": "iOLSKIxqn_....." ,
```

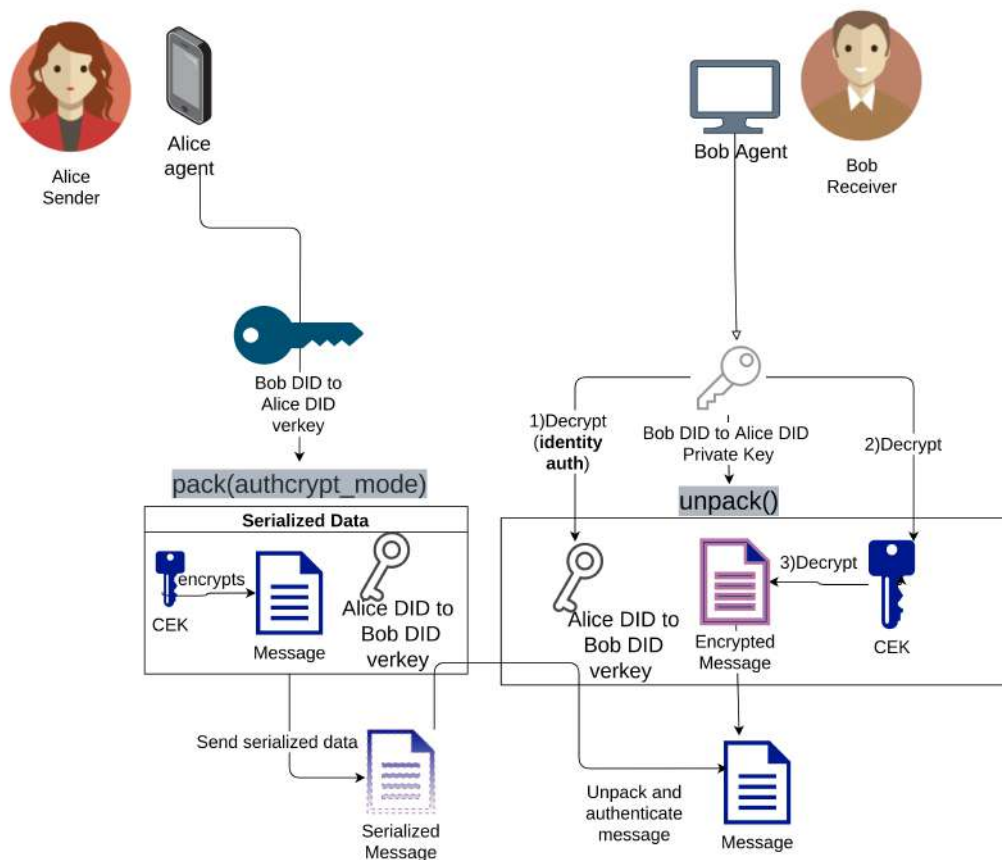


Figure 3.14: This message is encrypted with a Content Encryption Key (CEK). CEK key, Message, and Alice verkey are sent as serialized data. Bob receives the serialized data. Bob with Bob's private key decrypts Alice verkey is used to authenticate Alice's identity. Bob's private key is used to decrypt the CEK key. The CEK key is finally used to obtain the plain message

```
"tag": "gL-lfmD-MnNj9Pr6TfzgLA=="
}
```

Above (taken from [45][section `pack_message()`]) it can be seen the output of a `pack()` operation, where the field *protected* decodes to a structure containing the CEK, while the field *ciphertext* corresponds to the original message encrypted. It has to be noted that Indy pairwise DID communication breaches the concept of non-repudiation mentioned in subsection 2.1.1. As stated in [45]: *Authcrypt mode reveals the sender to the recipient only; it is not the same as a non-repudiable signature.*

The CEK as a shared private key made it possible to authenticate a sender identity while allowing the sender to repudiate a message. An example is made in Hyperledger documentation, by using the private key to encrypt a message as a shared secret key: *"Alice and Carol can negotiate a shared secret and trust one another not to leak it. Thereafter, if Alice sends Carol a message that uses the shared secret (e.g., it's encrypted by a negotiated symmetric encryption key), then Carol knows the sender must be Alice. However, she can't prove it to anyone, because Alice's immediate counter-response could be, 'Carol could have encrypted this himself. She knows the key, it has to be noted that this only works in a pairwise channel [46]."*

All the keywords in this subsection are summarized in Table A.1 in the Appendix.

3.5 Verifiable Credentials in Indy

Having defined what an Identity is in Indy and how two agents communicate, we can now introduce how Verifiable Credentials (discussed in subsection 2.5.3) are implemented in an Indy system.

There is a clear distinction between what goes on the ledger and what does not. As seen in the previous subsection, certain DIDs are private and known only to two parties. The claims owned by an agent in Indy are not public on the ledger, nor is there an association between the claim holder and the claim being made public. Indy adopts the W3C VCWG data model introduced in subsection 2.5.5. All the objects regarding credentials are stored in the Domain ledger as said in subsection 3.2.1.

3.5.1 What goes on the ledger?

Figure 3.15 shows the objects that go on or off the ledger. Such objects will be discussed in this section. As can be seen, credentials offers, credentials requests, proof requests, and even proofs are exchanged privately between the two agents. A credential holder can be an identity owner with a role on the ledger, but usually, a credential holder does not have a NYM record in the ledger. The verifier does not need to have a NYM record on the ledger to verify the requested proof since he only needs to read the ledger. As stated in section 3.2.3 anyone can read the ledger, even a role-less agent. We will see that credentials holders only

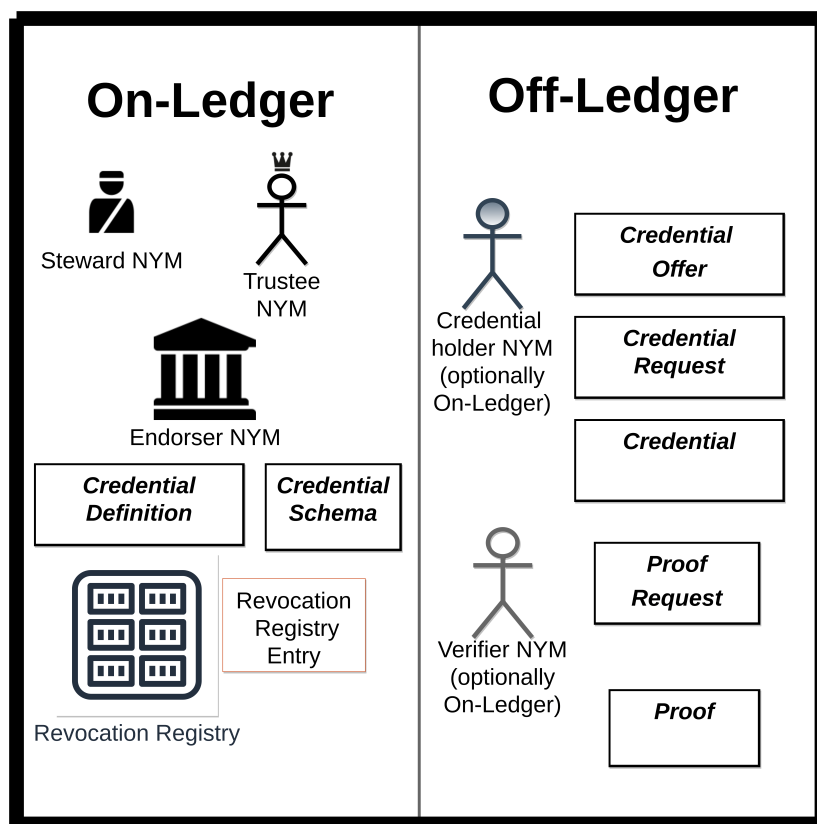


Figure 3.15: This figure divides public entities (On-Ledger) from privately exchanged entities (Off-Ledger). In some cases, the credential holder NYM and verifier NYM can be made public.

need to be able to read the ledger to create credentials requests and proofs. The verifier only needs to be able to read the ledger to verify the requested proof.

There could be cases where a Trustee/Steward/Endorser can be a Credential holder or take the role of a verifier, in such a case we say that the Credential holder/verifier NYM is On-Ledger. Figure 3.16 illustrates the actors allowed to create credentials and the Domain ledger stored objects regarding credentials. Issuers, as in the SSI framework, are represented by Indy agents with an Endorser role.

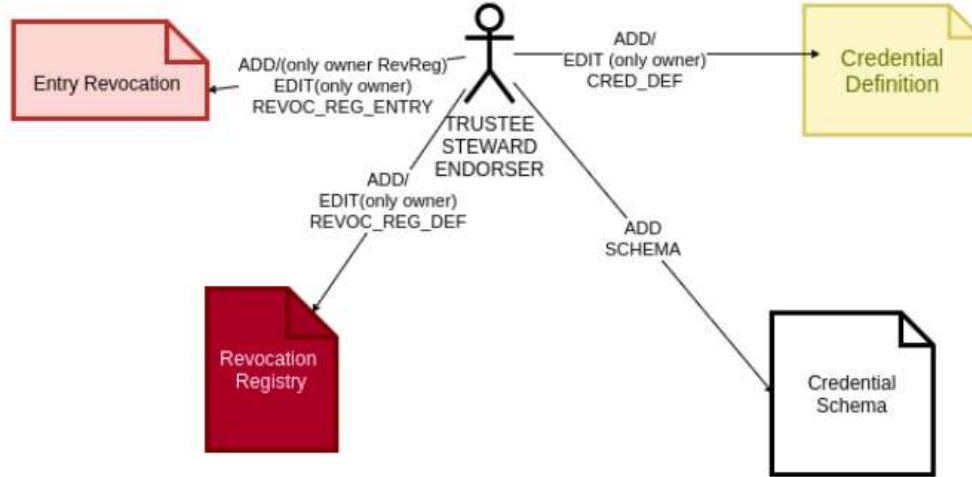


Figure 3.16: As defined by the default AUTH_MAP [41] Stewards, Trustees, and especially Endorsers can create: credentials schema, credentials definitions, revocation registries and entries to the revocation registries.

3.5.2 Credential Schema and SCHEMA transaction

The schema structure was briefly introduced in subsection 2.5.3. A schema defines the structure of a credential as a set of attributes. To create a new Credential Schema, an Endorsers signs and submits a **transaction SCHEMA** [18] that contains a <key,value> map, and is illustrated in figure 3.17 and seen on the ledger in figure 3.18.

A Credential Schema is identified by its transaction sequence number (seqNo) and its transaction Id (txnId), which is a string composed by:

<issuerDID>:<protocolVersion>:<schemaName>:<schemaVersion>.

The protocol version specifies the libindy version used in the pool. A schema is an identifier that contains its name and version. The Credential Schema structure is immutable. Changing a Credential Schema implies creating a new one, keeping the same name but a new version number [47], The credentials already issued with the previous schema will not be affected by it.

A Credential Schema is not bound for use only by its creator. It is possible to use it with many issuers' DIDs. Many other issuers can state to issue credentials with that Credential Schema. This allows having credentials with the same attributes. Before issuing a credential, an

```

{
  "txn": {
    "data": {
      "data": {
        "attr_names": [
          "name", "surname", "studentID"
        ],
        "name": "StudentIdentity",
        "version": "1.0"
      },
      "metadata": {
        "digest": "a1d78ab4345255558bfb78f5dab773973f41e7d456c47428a087d29ec76117ec",
        "from": "7CyGof2G3Gv8ys9AcUpeT9",
        "payloadDigest": "5bc0721c1669882a8e554b73bb75ea3bb85fdca10244ba6be577e38523bf07eb",
        "reqId": "1643221101619110000"
      },
      "protocolVersion": 2,
      "type": "101",
      "typeName": "SCHEMA"
    },
    "txnMetadata": {
      "seqNo": 275,
      "txnId": "7CyGof2G3Gv8ys9AcUpeT9:2:StudentIdentity:1.0",
      "txnTime": "2022-01-26T18:18:24.000Z"
    }
  }
}

```

Figure 3.17: A SCHEMA transaction (in enriched data format) locally created and seen by Indyscan.

issuer has to make public a **Credential Definition**.

3.5.3 Credential Definition

A Credential Definition is derived from a Credential Schema already stored in the ledger. A Credential Definition can be seen as a public declaration from an issuer that he will issue credentials defined by a particular Credential Schema and signed by one of their DIDs (*from*). Locally, a Credential Definition is created with the method `indy_issuer_create_and_store_credential_def` described as *"The method creates a credential definition entity that encapsulates the issuer of the credential DID, the Credential Schema, "secrets" (keys) used for signing credentials and "secrets" used for credentials revocation.*

Credential definition entities contain private and public parts. Private parts will be stored in the issuer's Wallet. The public part will be returned as a Json structure intended to be shared with all Anoncreds workflow actors usually by publishing CRED_DEF transaction to Indy distributed ledger." [48]. A Credential Definition contains the public key

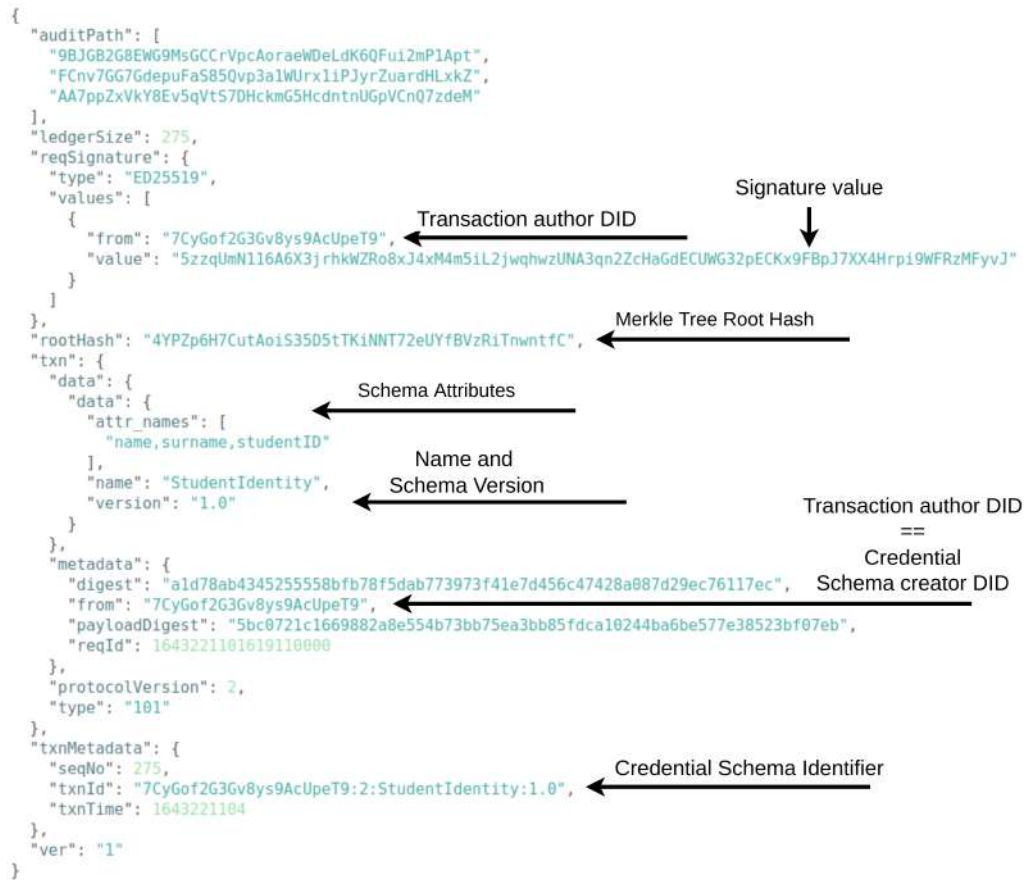


Figure 3.18: A SCHEMA transaction (as stored on ledger) locally created and seen by Indyscan. It can be noted that the transaction holds the Merkle tree root hash. As said in subsect 3.2.1 Indy stores transaction hash in it. This image does not show the full Credential Schema

that can be used to verify credentials issued from it. The public key is used to verify the issuer's signature. The issuer signs credentials with a private key corresponding to the public key present in the Credential Definition. It has to be noted that this key pair is different from the issuer DID key pair. While the Issuer DID key pair is for the signature and validation of transactions towards the pool. The Credential Definition key pair is used in its private part to sign a credential and its public part to verify it. The public part can be retrieved from the ledger and used to verify a credential.

A Credential Definition is created by a CLAIM_DEF transaction, a basic Credential Definition format is shown in figure 3.19. The Credential

```

{
  "txn": {
    "data": {
      "refSchemaAttributes": [
        "name,surname,studentID"
      ],
      "refSchemaFrom": "7CyGof2G3Gv8ys9AcUpeT9",
      "refSchemaId": "7CyGof2G3Gv8ys9AcUpeT9:2:StudentIdentity:1.0",
      "refSchemaName": "StudentIdentity",
      "refSchemaTxnSeqno": 275,
      "refSchemaTxnTime": "2022-01-26T18:18:24.000Z",
      "refSchemaVersion": "1.0",
    },
    "metadata": {
      "digest": "04926548173003d9bace764c19d10a9a6835e136a7d981a19b56229990b73f0",
      "from": "Ak6t91kJBZJ8ZvzYEmypHg",
      "payloadDigest": "7364e4b516e62a8ec5eae0408e7dad60f6e8893ad75413cb2ae91c5b8f95c2b7",
      "reqId": "1643221106096657200",
    },
    "protocolVersion": 2,
    "type": "102",
    "typeName": "CLAIM_DEF",
  },
  "txnMetadata": {
    "seqNo": 276,
    "txnId": "Ak6t91kJBZJ8ZvzYEmypHg:3:CL:275:TAG2",
    "txnTime": "2022-01-26T18:18:27.000Z"
  }
}

```

Figure 3.19 shows a CLAIM_DEF transaction (enriched data format) derived by the Credential Schema from the previous figure 3.17. The transaction structure is as follows:

- Schema Attributes:** "name,surname,studentID"
- Credential Schema Issuer DID:** "7CyGof2G3Gv8ys9AcUpeT9"
- Credential Schema Identifier:** "7CyGof2G3Gv8ys9AcUpeT9:2:StudentIdentity:1.0"
- Schema Name:** "StudentIdentity"
- Schema version:** "1.0"
- Credential Definition Issuer DID:** "Ak6t91kJBZJ8ZvzYEmypHg"
- Inferred transaction type:** "CLAIM_DEF"
- Credential Definition Identifier:** "Ak6t91kJBZJ8ZvzYEmypHg:3:CL:275:TAG2"

Figure 3.19: a CLAIM_DEF transaction (enriched data format) derived by the Credential Schema from the previous figure 3.17. As can be seen, the DID of the Credential Schema creator and Credential Definition Issuer is different.

Definition *txnId*, with structure

<issuerDID>:<Cred.DefTitle>:<signignProtocol>:

<SchemaDef.Txn.Seqno>:<Cred.Def.TAG>

specifies the protocol used to sign the credential when issued. The *txnId*: "LqvCG5UAKAiYqYzzKF3HNf:3:CL:324:TAG2" of figure 3.19 states the digital signature protocol is CL, which stands for Camenisch-Lysyanskaya. As of now, the CL signature protocol is the only supported one for the digital signature of credentials. An Endorser can create multiple Credential Definitions related to a specific Credential Schema, and the TAG is used to distinguish them from each other.

Two fields are holding the public part of two key pairs[18]:

- primary (dict): primary claim public key (the claim term is interchangeable with the term credential in Indy documentation). It is the aforementioned public part of the key pair for credential verification;

```

{
  "auditPath": [
    "Do9nax6AAGmpXjMi7cAjdaVsZnc6rnkvhmsQXPZiCZa",
    "9BJGB2G8EWG9MsGCCrVpcAoraewDeLdK6QFui2mP1Apt",
    "FCnv7GG7GdepuFaS85Qvp3a1WUrx1iPJyrZuardHLxkZ",
    "AA7ppZxVvK8Ev5qVtS7DHckmG5HcdntnUGpVCnQ7zdeM"
  ],
  "ledgerSize": 276,
  "reqSignature": {
    "type": "ED25519",
    "values": [
      {
        "from": "Ak6t91kJBZJ8ZvzYEmypHg",
        "value": "3kYmZpqrB1e2ydfqzdxvGXQC7HqkvYasszfGn7XxbPMgE3J4D1v2LsEtALnYLYmW19gcLKTdExPvHfNcyR0Y3C"
      }
    ]
  },
  "rootHash": "4M98nPJVGTnuJnYc056MJJ3z2VRLBbaUSnkoTSFTuFh6",
  "txn": {
    "data": {
      "data": {
        "primary": {
          "n": "7392985927799274162191392525015406894876119067750179507436795204432580074501988280655313163",
          "r": {
            "master_secret": "31670969853822607789264876080184526392152770139888757397876266758868625479610",
            "name,surname,studentid": "65853815986979947806223024246339585568893492451666766941024790822624"
          },
          "rctxt": "642120014033040160609419365269113631998151198410117375784943022924278493340625902983483",
          "s": "1192661698246237278991889658073311163944563837554112145529193156670275274134791471870645169",
          "z": "4798351556597686909195115924079521318700437465229305286088995210688405168612390076976764435"
        },
        "revocation": {
          "g": "1 1BC2D49255F7FF883889994F349F9308F19603906487D9DF362D619179CFEA2E 1 212F14D20B16A209EC892B",
          "g_dash": "1 03D0CFC5BDFC405EA10F4FA073298EBC8AA2F9DF34A0AD34E6D5275EFE48B380 1 24D96443A7DE87B2F",
          "h": "1 0ACFC9A95DC52A1E0D2C69E61CD7035BA84C4E38B7C0742EAD57C3149BC31A57 1 1D3741178FE6141F409291",
          "h0": "1 186B7409880E429F5871F8F28C8C300563831F4881A1B6178766906A8083828A 1 1730F174892910BFE103A",
          "h1": "1 010ECB6749C16AC1709127CC8457398C8AA982D3EF214F7A7E24CCFCA267B60 1 06FFC697507AC9877783B",
          "h2": "1 160BD2CAC80CC3161592C96747B5C3464FF647A8F938EA8069CC88E9FE6636AA 1 11F1FD2A859ED7260FEDC",
          "h_cap": "1 219ED99A9056C4410EC1EB0C71630996027728F4E845B7A23447027F7DB86FF7 1 202E897017121B9596",
          "htilde": "1 17A9C57C32206D6C8AA32136BE13D9E8BAE738F5CC650ACA287900CB41B19342 1 1089D1C6405D75062",
          "pk": "1 02A4F6C6E623693E3088EA842D053E7277F418B87E5AA71C00845B00060FCB32 1 03EAD3ECCB26D9A219F8B",
          "u": "1 08897E312106D27A69CF3FA47148C84696EA18DA949B68AC8A849BD551CA0AD6 1 1EF68CB8179F6CF77E7F59",
          "y": "1 203A80DFA69299044CC5847199E56CE6C7900042C81CF218FE20A5A667B0AD81 1 17C03151AFCB080F7FB655"
        }
      },
      "ref": 275,
      "signature_type": "CL",
      "tag": "TAG2"
    },
    "metadata": {
      "digest": "04926548173003d9bace764c19d10a9a6835e136a7d981a19b5622990b73f04",
      "from": "Ak6t91kJBZJ8ZvzYEmypHg",
      "payloadDigest": "7364e4b516e62a8ec5eae0408e7dad60f6e8893ad75413cb2ae91c5b8f95c2b7",
      "reqId": 1643221106096657200
    },
    "protocolVersion": 2,
    "type": "102"
  },
  "txnMetadata": {
    "seqNo": 276,
    "txnId": "Ak6t91kJBZJ8ZvzYEmypHg:3:CL:275:TAG2",
    "txnTime": 1643221107
  },
  "ver": "1"
}

```

Transaction signature value

Merkle Tree Root Hash

Credential Definition public key

Revocation credential public key

Transaction author DID

Figure 3.20: a CLAIM_DEF transaction(as stored on the ledger) derived by the Credential Schema from figure 3.18.

The fields primary and revocation represent two public keys of the Credential Definition. The Merkle tree root hash value has changed from the value of figure 3.18.

- revocation (dict): revocation claim public key. It is used to verify the **proof of non-revocation** part in a proof. The aforementioned proof of non-revocation can be seen in appendix A.2 at line 35. This structure is present only if the credential is **revocable**.

Both fields can be seen in figure 3.20 The second entry contains the key of the revocation registry (that we will discuss in the next section).

3.5.4 Revocation Registry and Revocation Registry Entry

A credential issuer can be associated with a Credentials Definition **Revocation Registry** used to keep track of revoked credentials, which is limited in size to a certain number of entries.

To create a revocation registry a REVOC_REG_DEF transaction (shown in figure 3.23) is used. The transaction defines the number of credentials that can be associated with the Revocation Registry. The REVOC_REG_DEF transaction also contains a reference to the Credential Definition with its *txnId*. To illustrate how the Revocation Registry is composed, we will anticipate how credential revocation works using cryptographic accumulators.

Cryptographic accumulators

Cryptographic accumulators form the mathematical basis for revocation in Indy. Given a set of elements $SET=[a,b,c,d]$, the accumulator is the product of all the elements, i.e. $ACCUMULATOR=a*b*c*d$. Not knowing the values of the subset $[a,c,d]$, but knowing the value $ACCUMULATOR$ and a value $WITNESS=a*b*c$, it can be proven that $b \in SET$. To prove that $b \in SET$, it has to be shown that $b*WITNESS=ACCUMULATOR$.

Credential Revocation

As explained in [49], the Revocation Registry contains a cryptographic accumulator associated with a Credential Definition. An issuer can revoke a credential by changing the underlying set which generates a different accumulator. After a credential is revoked, the holder cannot generate the accumulator using the values from his credentials. The Revocation Registry size limit is used to limit the number of elements

of a set and to define the size of a **Tails File** (shown in figure 3.21), which is a structure that contains a value for every credential. The Tails File is indexed by the credential revocation registry id(`cred_rev_id`), a value that is contained in a credential. The `cred_rev_id` is in the range $[0, \text{TailsFile_size}]$.

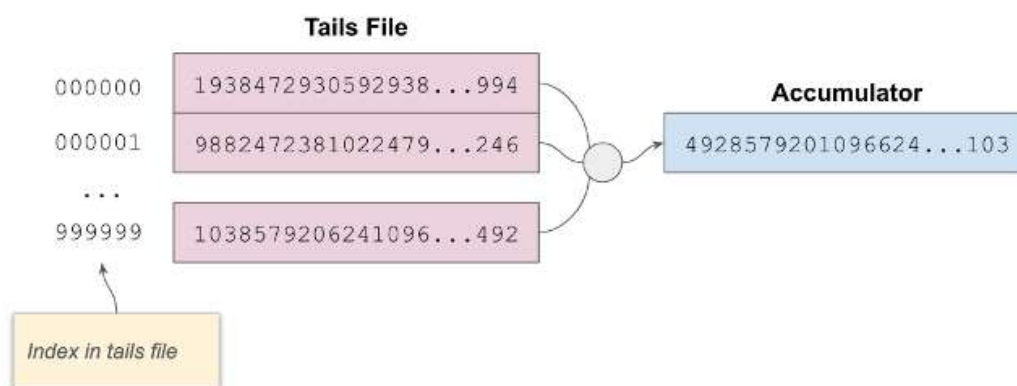


Figure 3.21: The representation of a Tails File, as taken from [49]. It can be seen how each entry is a large number, which is indexed by a credential `cred_rev_id`. The accumulator value is the product of many entries.

The value contained in the Tails File is a large number. The accumulator is the product of all the entry values. The accumulator at a certain time is the product of the values contained in the index of non-revoked credentials. Figure 3.22 shows how the Revocation Registry accumulator values change after a revocation. When a credential is revoked, its indexed value in the Tails File remains stored inside it. Since the value associated with a revoked credential is not used in the accumulator calculation, generating valid proofs using it would fail.

What the issuer does when revoking a credential consists of changing the answer to a mathematical problem. The type of revocation used in Indy and the only supported one right now is Camenisch-Lysyanskaya Accumulator (CL Accum).

The Revocation Registry can be defined by two different issuance types

- **ISSUANCE_ON_DEMAND**: the Revocation Registry has to be updated for every credential issued and revoked.
- **ISSUANCE_BY_DEFAULT**: used when all credentials were already issued, the issued revocation registry is to be updated only after

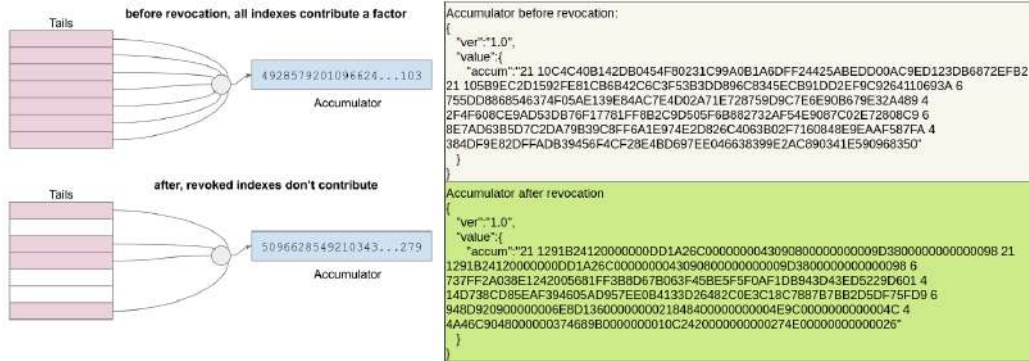


Figure 3.22: The representation left side of the figure represents the tails file and accumulator before and after revocation. The left side is taken from [49]. The right side of the figures shows the change of an accumulator value after a revocation. This example was created in a local indy pool. The accumulator value was taken from a GET_REVOC_REG request result

revocation. The Revocation Registry only keeps track of revoked credentials.

3.5.5 REV_REG_DEF transaction

Figure 3.23 shows a REV_REG_DEF transaction. Since the Tails File is public, its URI is declared in the Revocation Registry Definition. After the recreation of a credential (*issuerCreateCredential*) and after revocation, the accumulator value in the ledger has to be changed. The new accumulator value is returned by credential creation/revocation operation. To update a Revocation Registry accumulator value, a REVOC_REG_ENTRY transaction containing the new accumulator is made. A REVOC_REG_ENTRY transaction is done every time a credential is created and revoked. The REVOC_REG_ENTRY transaction format is different between an issuing and a revoked update as can be seen in figure 3.24 and figure 3.25. It is more efficient to accumulate operations, and then doing a REVOC_REG_ENTRY transaction.

An entry can contain one or more updates called **deltas**. The transaction can contain the field `issued:[index_1,index_2,index_3,...]` and/or `revoked [index_1,index_2,index_3,...]` which contains the index in the Tails File

```

{
  "txn": {
    "data": {
      "credDefId": "4qYoz57efktqW5FBex8BsU:3:CL:2356:TAG2",
      "id": "4qYoz57efktqW5FBex8BsU:4:4qYoz57efktqW5FBex8BsU:3:CL:2356:TAG2:CL_ACCUM:H1",
      "revocDefType": "CL_ACCUM",
      "tag": "Hi",
      "value": {
        "issuanceType": "ISSUANCE_ON_DEMAND",
        "maxCredNum": 5,
        "publicKeys": {
          "accumKey": {
            "z": "1 0EBD5A9D817BB52BB3819C1B704F8F598FF401D9C7F89FCD40C5E00804242A4F 1 0EFA45482114EF37F53C6385A80B984D96483908"
          }
        },
        "tailsHash": "7Adx51fQHjArdwWm7qZU7C3ydfsHLTujeXZAbikQbFn",
        "tailsLocation": "/home/calogero/.indy_client/tails/7Adx51fQHjArdwWm7qZU7C3ydfsHLTujeXZAbikQbFn"
      }
    },
    "metadata": {
      "digest": "a12f7ee93635814f0ff8abce4f37653839c62fc6de1c226a85c999bed199d378",
      "from": "4qYoz57efktqW5FBex8BsU",
      "payloadDigest": "bbae61d697472178a64a29ccc3259148a55ff5629f33e31179f51e5a2bf8f605",
      "reqId": "1636492782057950700"
    },
    "protocolVersion": 2,
    "type": "113",
    "typeName": "REVOC_REG_DEF"
  },
  "txnMetadata": {
    "seqNo": 2361,
    "txnId": "4qYoz57efktqW5FBex8BsU:4:4qYoz57efktqW5FBex8BsU:3:CL:2356:TAG2:CL_ACCUM:H1",
    "txnTime": "2021-11-09T21:19:42.000Z"
  }
}

```

Figure 3.23: REV_REG_DEF transaction example. It can be noted how the public Tails File URI is specified. Also, the Tails File size is specified with the maxCredNum entry. The accumulator value was taken from a GET_REVOC_REG request result. The issuanceType field is set as ISSUANCE_ON_DEMAND

of issued credentials/revoked credentials.

The issued field is absent if the issuanceType of the Revocation Registry is ISSUANCE_BY_DEFAULT.

3.5.6 Credential Offer and Credential Request

When issuing a credential, there is a setup phase between the issuer and the holder. A credential holder is also known as **prover** as he will need to prove to hold one or more credentials. The term prover and holder can be interchanged, but this thesis will generally use the term holder. Upon a DID to DID communication, the holder requests a specific credential from the issuer. The issuer sends the holder a Credential Offer, which contains the Credential Definition *txnID* and the Credential Schema *txnID*. As seen in figure 3.15, a Credential Offer is stored by the holder (e.g. in a Wallet). Below is shown an example of a Credential Offer:

```
{
```


Revocation Registry

Definition Id

accumulator value

Previous accumulator value

When creating a Credential Request, the holder inserts a **MasterSecret** value in a blinded form. The MasterSecret value does not have to be confused with the "master_secret" value present in the previous Credential Offer, which represents a field inside the key_correctness_proof structure of the offer.

MasterSecret

The MasterSecret value is used to link all the credentials inside the Wallet belonging to any agent. When created (*proverCreateMasterSecret*), the MasterSecret is stored inside the Wallet and is used to generate proof with the credentials inside the Wallet. When a credential is issued, the blinded MasterSecret of the holder from the Credential Request is bundled inside the credential. Creating a proof with a credential without having its associated MasterSecret would create a non-valid proof. Below is the Credential Request created from the Credential Offer previously shown:

```
{
  "blinded_ms_correctness_proof": {
    "c": "5801555.....",
    "v_dash_cap": "13754799.....",
    "m_caps": {"master_secret": "140513...."},
    "r_caps": {}
  },
  "blinded_ms": {
    "committed_attributes": {},
    "u": "86259075115557317857656611449.....",
    "ur": "1 0390FA5D2343EDCB570.....",
    "hidden_attributes": ["master_secret"]
  },
  "cred_def_id":
    "MchNFAFVJ8a9qyJvVWBqyG:3:CL:214:TAG2",
  "nonce": "310703307825719878186886",
  "holder_did": "Srvk37m4FSvPufMLcb9oiC"
}
```

As said in [38], MasterSecret *allows for "credential presentations" that are unlinkable to each other*. It can be seen that the fields `blinded_ms_correctness_proof` and `blinded_ms` from the above Credential Request contain information about the blinded Master Secret. Credential Requests are privately exchanged from the holder to the issuer.

3.5.7 Credentials, their structure, and how Master-Secret links them to their holder

After receiving the Credential Request, the issuer issues a credential (*issuerCreateCredential*) and signs it using the CL protocol. The created credential is sent to the holder who stores it in his Wallet (*holderStoreCredential*) alongside the meta-data. Below is shown the Credential Request meta-data, which contains information about the MasterSecret.

```
{
  "master_secret_blinding_data": {
    "vr_prime": "090AF9C0B983A813AFBB16AC479C4",
    "v_prime": "23708815349..."
  },
  "master_secret_name": "0e6a6869-f288-491e...." ,
  "nonce": "310703307825719878186886"
}
```

The field "master_secret_name" contains the identifier of the Master-Secret bundled in the credentials and stored in the holder's Wallet. The credential structure as got from the Wallet is:

```
{
  "cred_info": {
    "cred_rev_id": "1",
    "rev_reg_id":
      "MchNFAFVJ8a9qyJvVWBqyG:4:
      MchNFAFVJ8a9qyJvVWBqyG:
      3:CL:214:TAG2:CL_ACCUM:Hi",
    "referent": "a9eacb97-9d35-4c15-a3ee-
      462519776ecf",
    "schema_id": "MchNFAFVJ8a9qyJvVWBqyG:2:
      StudentIdentity:1.0",
    "cred_def_id": "MchNFAFVJ8a9qyJvVWBqyG
      :3:CL:214:TAG2",
    "attrs": {"name": "Alice"}
  }
}
```

The field "cred_rev_id" represents the index of the issued credential in the Tails File associated with the Revocation Registry. The field referent is the identifier of the issued credential inside the Wallet.

3.5.8 Proof request

When asking a holder for a credential with attributes (claims) that satisfy a set of conditions to access a service, a verifier creates a **proof request**. Such attributes requested might be a subset of the attributes inside a credential, and a proof request can contain attributes from one or more credentials. The holder generates a Zero-knowledge proof (introduced in 2.5.4) to prove that he has all the valid attributes to satisfy the proof request. Some or all of the attributes requested can be presented in a revealed or unrevealed form, depending on the holder's discretion. An example of proof with an unrevealed attribute is in Appendix A.7.5. While with revealed attributes an example is in Appendix A.3.1. Revealed attributes are placed in the `revealed_attrs` and `revealed_attr_groups` structures of a proof. While unrevealed attributes are placed inside the `unrevealed_attrs` structure.

Instead of the plain attributes, a verifier can instead request proofs satisfying a certain predicate. An example of a predicate would be to ask that the credential holder $\text{age} \geq 18$, instead of asking for his age. Predicates have success in the goal of further preserving the holder's privacy, as he can prove to have the requested attribute value without disclosing the value itself.

Predicate proofs and the ability to disclose only certain attributes of a credential allow Indy to perform the **Selective Disclosure** of attributes. As said in [13]: *Selective disclosure of VC data - exposing only the data necessary*.

A proof request cannot be created by using Indy-SDK methods, instead, it is a JSON structure that must match this structure [50]:

```
{
  "name": string,
  "version": string,
  "nonce": string,
  "requested_attributes": {
    "attr_referent": <attr_info>,
    ...,
  },
  "requested_predicates": {
    "<predicate_referent>": <predicate_info>,
    ...,
  },
}
```

```

    "non_revoked": Optional<<non_revoc_interval>>,
    "ver": Optional<str>
}

```

Where the *predicate_info* structure is:

```

predicate_info:
    Describes requested attribute predicate
{
    name": attribute name,
    p_type": predicate type (">=", ">", "<=", "<")
    p_value": int predicate value
    restrictions": Optional<filter_json>,
    non_revoked": Optional<<non_revoc_interval>>
}

```

and the *attr_info* structure is:

```

{
    "name": Optional<string>,
    "names": Optional<[string, string]>,
    "restrictions": Optional<wql query>,
    "non_revoked": Optional<<non_revoc_interval>>,
)
}

```

In *attr_info* the field "name" and "names" are mutually exclusive: "name" specifies just one attribute while "names" specifies a set of attributes [attr_1,attr_2..,attr_n]. The field "names" limits the source of the attributes to only one credential while using more *attr_info* structures in the proof requests allows to have attributes from different credentials. The "restrictions" field is used to restrict to a certain subset or to one, the credentials from which attributes can be taken, and it is a query defined in the Indy Wallet query (wql) language [53]. The allowed fields in the query are [50]:

```

"schema_id": <credential schema id>,
"schema_issuer_did": <credential schema issuer did>,
schema_name": <credential schema name>,
schema_version": <credential schema version>,
issuer_did": <credential issuer did>,
"cred_def_id": <credential definition id>,
rev_reg_id": <credential revocation registry id>

```

We will generally write restrictions as `subquery = subquery, ..., subquery` which means `WHERE subquery AND ... AND subquery` [53].

It was found that even if not clearly documented, it is possible to insert restrictions on an attribute value. Restrictions to attribute value can be a powerful tool since a verifier cannot request a holder to reveal the value of his attribute, which can be presented in an unrevealed form. To put a restriction to an attribute value, the subquery used is

`"attr::attribute name::value":"attribute raw value"`,

where "attribute raw value" corresponds to value requested. It has to be highlighted that an attribute used in a predicate in a proof must remain secret, even if the prover tries to disclose it.

Indy does prohibit disclosing the attribute in a revealed form, but the attribute can be part of the proof as an (unrevealed) attribute and be contained in a predicate.

The field "non_revoked" specifies the time window in which a credential holding the attribute referenced was not revoked. It is possible to prove that the now revoked credentials were non-revoked in the past. It can be done by using the accumulator value of a past timeframe.

A simple proof request linked to previous examples where only one attribute is required would be:

```
{
  "ver": "1.0",
  "name": "reproof1",
  "requested_attributes": {"attr0_referent": {
    "name": "name",
    "restrictions": {
      "issuer_did":
        "MchNFAFVJ8a9qyJvVWBqyG",
      "cred_def_id":
        "MchNFAFVJ8a9qyJvVWBqyG:3:
        CL:214:TAG2",
      "schema_issuer_did":
        "MchNFAFVJ8a9qyJvVWBqyG"
    },
    "non_revoked": {
      "from": 1642436555,
      "to": 1642436555
    }
  }
```

```

    }},
    "non_revoked": {
      "from": 1642436555,
      "to": 1642436555
    },
    "nonce": "833895142217180746822212",
    "version": "1.0",
    "requested_predicates": {}
  }
}

```

3.5.9 Proof

After a proof request is privately exchanged between the verifier and the holder, the holder has to find the credentials that fit the received request.

To create a proof, the holder uses the Credential Schemas, Credential Definitions, and revocation states for the requested credentials. The verifier will not receive the credentials nor directly ask for them in the proof request. The verifier asks for attributes or predicates and can restrict the accepted attributes to a subset of credentials type or a single type of credential. This can be accomplished by using the restrictions field illustrated. What the verifier receives is the proof that the holder has the required credentials containing the attributes in the proof request. The proof structure is as follows [50]:

```

{
  "requested_proof": {
    "revealed_attrs": {
      "requested_attr1_id":
        {sub_proof_index: number,
         raw: string,
         encoded: string},
      "requested_attr4_id":
        {sub_proof_index: number
         raw: string,
         encoded: string},
    },
    "revealed_attr_groups": {
      "requested_attr5_id": {

```

```

        "sub_proof_index": number,
        "values": {
          "attribute_name": {
            "raw": string,
            "encoded": string
          }
        },
      },
    },
    "unrevealed_attrs": {
      "requested_attr3_id":
        {sub_proof_index: number}
    },
    "self_attested_attrs": {
      "requested_attr2_id":
        self_attested_value,
    },
    "predicates": {
      "requested_predicate_1_referent":
        {sub_proof_index: int},
      "requested_predicate_2_referent":
        {sub_proof_index: int},
    }
  }
  "proof": {
    "proofs": [ <credential_proof>,
               <credential_proof>,
               <credential_proof> ],
    "aggregated_proof": <aggregated_proof>
  }
}

```

It can be seen that there are unrevealed and revealed attributes. In the latter case, the proof would imply that the holder has a specific attribute that fits the criteria of the restrictions, but it is put in a blinded form inside the proof. There are also self-attested attributes: those attributes are not part of a credential issued by an issuer, the holder self attests their value. Self-attested attributes can be used when there are no restrictions on attributes. The proof created for the previous proof

request can be seen in Appendix A.2.

The proof is non-valid in the case of :

- Credential revocation: if a credential supporting a proof is revoked, a valid proof can become false. The proof could be valid if the non-revocation interval (can be seen in subsection 3.5.8) of the proof request specifies a time window before its revocation.
- A credential used in the proof has a different Credential Definition than the one the verifier uses to verify the proof.
- A false proof created by modifying a valid proof (or a made-up proof). Integrity checks on the proof are performed during verification.

3.5.10 Unlinkability of attributes

Unlinkability in a two-party relationship means not letting the other party correlate more information about an agent's identity than the one disclosed (this was introduced in subsection 3.3). As in the Hyperledger Ursa documentation: *We say that credentials are unlinkable if it is impossible to correlate the presented credentials across multiple presentations. This is implemented by the holder proving with a zero-knowledge proof that he has a credential rather than showing the credential* [51].

Indy DID to DID communications and Zero-Knowledge proofs allow for two types of unlinkability [54] for an agent attributes.

1. Session Unlinkability: a session starts after two agents establish a channel of communication. The information exchanged between agent A and agent B cannot be correlated to another session. After a session ends and agent B starts another session with Agent A', Agent B cannot correlate the information he has about agent A with agent A'. Pairwise DID change from one session to another. Agent A and agent A' could be the same agent but agent B cannot know it beforehand. When a session is active and if more proofs are exchanged, their attributes can be correlated.
2. RP (Relying Party) unlinkability: during a session agent B cannot know if he is communicating with one agent only. Only the Pairwise DID is known to each agent. Agent A can be switched to agent

A' (for example using a different device), which is given the same Pairwise DID for communicating with B. For example, an Identity Owner such as Alice can use two different devices. The device uses the same DID to communicate with Agent B. Part of the communication is done with Box smartphone using Agent A. Alice can continue the communication from a PC, using Agent A'. Alice's transition from agent to agent is transparent to Agent B.

Entities of this subsection are summarized in table Tables A.4 and Table A.3 in the Appendix.

3.5.11 Portable Credentials: Student Mobility Use Case

This subsection shows how Verifiable Credentials integrate into an already existing system, like the Erasmus+ Programme. We will see how Verifiable Credentials can support student mobility. In the case of the Erasmus+ Programme, decentralization allows for a student to have just one digital identity to share among more universities.

Problem: one student - many identities

As said in subsection 2.5, digital identities are typically controlled by external organizations that store them. The identity owner does not have control of their identity. In the case of the Student Mobility context, a student is set to have many identities for their home university and one or many host universities. Student mobility identity challenges are the subject of the project Erasmus Without Paper funded by the European Commission. The Erasmus Without Paper website states one of its goals: *"Tackling the administrative workload for students and staff. Both mobile students and staff supporting student exchanges need to process a lot of documents and procedures that lead to a huge workload. The workload can be heavily reduced by embracing technology."* [55].

Student identity

From section 3.3, in the relationship-attribute plane, different external subjects see an identity owner in different ways.

The home university of a student sees him as their student and stores a

fragment of his identity. The attributes regarding the career of a student are stored in his university data centers. A student would need to be manually registered to a host university, i.e. he has to create a new fragment of his identity. By using verifiable credentials (introduced in subsection 2.5.3) we can allow the student to share their identity with many universities. Indy allows users to define one or more credentials with different attributes correlated to the student identity. By using Credential Schemas and Credential Definitions (see subsections 3.5.2 and 3.5.3), it is possible to define different formats of student credentials issued by universities. A university can accept credentials from selected external universities.

Student Mobility from the real world to Indy

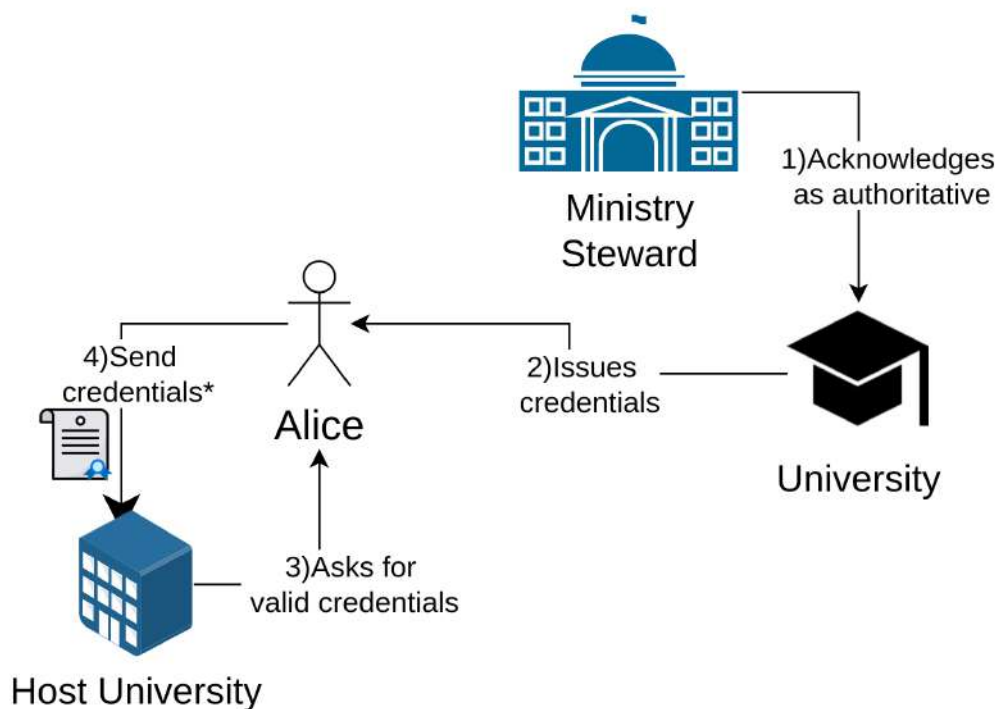


Figure 3.26: Overview of an SSI solution for Student Mobility. Main actors and their interactions are shown.

Figure 3.26 highlights universities recognized as authoritative by a Ministry.

We can then bridge the gap between the real world and Indy starting by defining the Ministry as a Steward. There could be other entities such as Stewards, such as Erasmus National Agencies. Universities will issue credentials, so they will have the role of Endorsers. In this case, the definition of Trust Anchor is used with the role Endorser since it fits the authoritative role of a university (see subsection 3.4.3). University is an authoritative institution to issue credentials to students.

A student could ask their university the attributes defining his career. A university could create one or more Credential Schemas each with different attributes e.g. [name,surname,StudentID,grades average]. In figure 3.26 Alice can be asked to present credentials to enroll in a Host University. Alice will send to the Host University a proof that she holds the required attributes (the concept of proof was introduced in subsection 2.5.3).

Figure 3.27 enriches the architecture showing the Indy ledger, Anoncreds entities, and a System Admin as a Trustee.

Credentials, proofs and restrictions

A host university could also restrict to a certain range Alice's grade average or restrict the home university allowed for a student. If Alice does not belong to a home university with an agreement with the host university, credentials from Alice's home university will not generate a valid proof.

A host university could impose such restrictions using the "restrictions" field of its proof request (discussed in section 3.5.8). The host university could restrict Alice's grade averages without asking for the actual grade average, by using predicates.

3.6 Anoncreds Workflow

In the previous section, we defined the building blocks of how are Verifiable Credentials structured in Indy. We will refer to those building blocks as Anoncreds entities (Anonymous credentials entities). In defining Anoncreds entities we loosely described the workflow to create a proof, from request to proof creation. Now we will dive deep into how the Anoncreds workflow can be integrated into a real-world scenario. We assume Alice wants to study at the University of Copenhagen as an

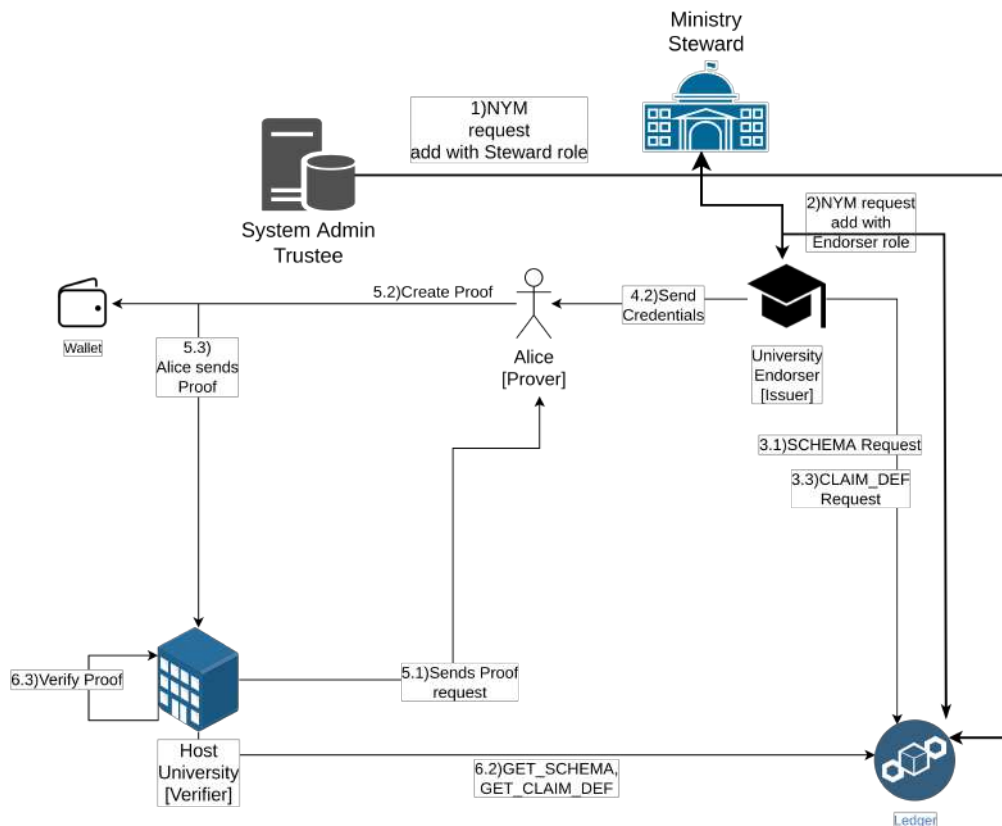


Figure 3.27: From real world to Indy. Addition of System Admin as Trustee. NYM request used to add the University as Endorser. Indy transactions have been added to the operational flow. Also, the Wallet where Alice stores credentials can be seen in the figure.

exchange student. Alice's home university is the University of Pisa. Alice has to present proof of being a student at the University of Pisa to the University of Copenhagen.

3.6.1 Get a NYM with a role in the ledger

Usually, the agents that issue credentials the most are Endorsers. To appoint an Endorser would mean that a Trustee or a Steward create a NYM transaction to register the Endorser DID and verkey on the ledger. This process is equal to the process of a Trustee promoting an agent to Steward, minus using a different role value in the NYM transaction. To become a Steward there needs to be an agreement with a Trustee and

to provide a node to the pool. How to become a Steward depends on the policy chosen by the network Trustees (e.g. Sovrin [56]). Becoming an Endorser would mean reaching an agreement with a Steward to become one, this could be done after establishing a DID to DID communication between the two.

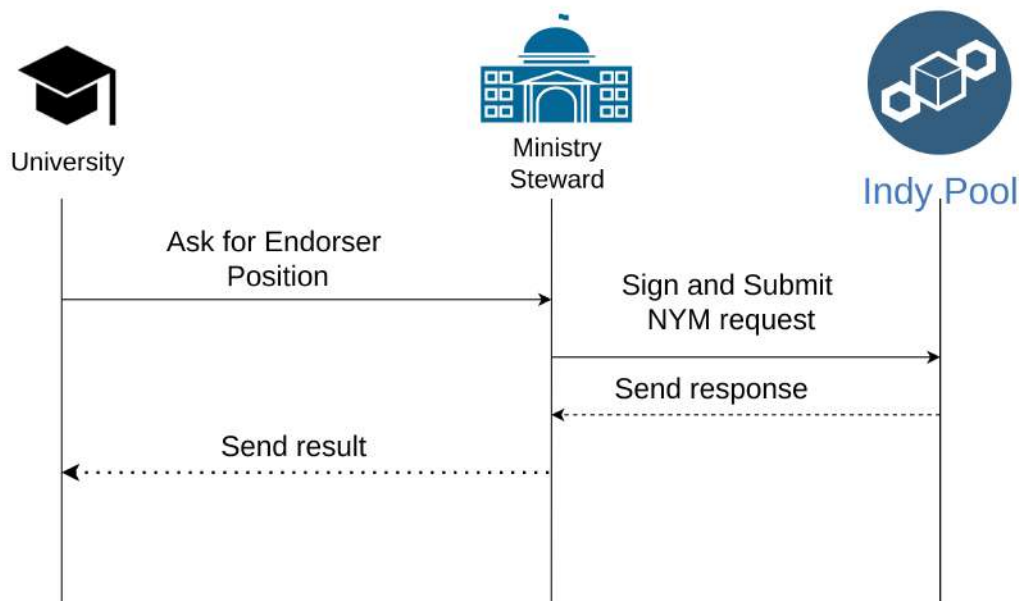


Figure 3.28: A "wannabe" Endorser asks a Steward for the given role. The Steward signs and submits a NYM transaction to the ledger. The operation result is given to the new Endorser (if the transaction is approved).

University of Pisa to issue valid credentials has to be recognized as authoritative by a Ministry. We identify the Italian Ministry of University and Research as Steward. The University of Pisa asks for an Endorser role to the Ministry. Figure 3.28 shows the process of giving an Endorser NYM to the University of Pisa. The role given in the "role" field of the NYM transaction would be 101, as it is the code for the Endorser role. The University of Pisa communicates with the Ministry. The University of Pisa sends the Ministry (Steward) a DID and the associated verkey it has created and stores in its Wallet (*createAndStoreMyDid*). The Ministry (Steward) creates (*buildNymRequest*) a NYM request, which is submitted as a transaction. It signs the transaction with its private key and sends the request to the pool (*signAndSubmitRequest*). The

Ministry (Steward) gets a response from the pool of nodes regarding the transaction. Steward was then able to inform the new Endorser of the given transaction success state result from the pool. After becoming an Endorser, the University of Pisa could make an ATTRIB request to add its endpoint to the ledger.

3.6.2 Credential Issuing

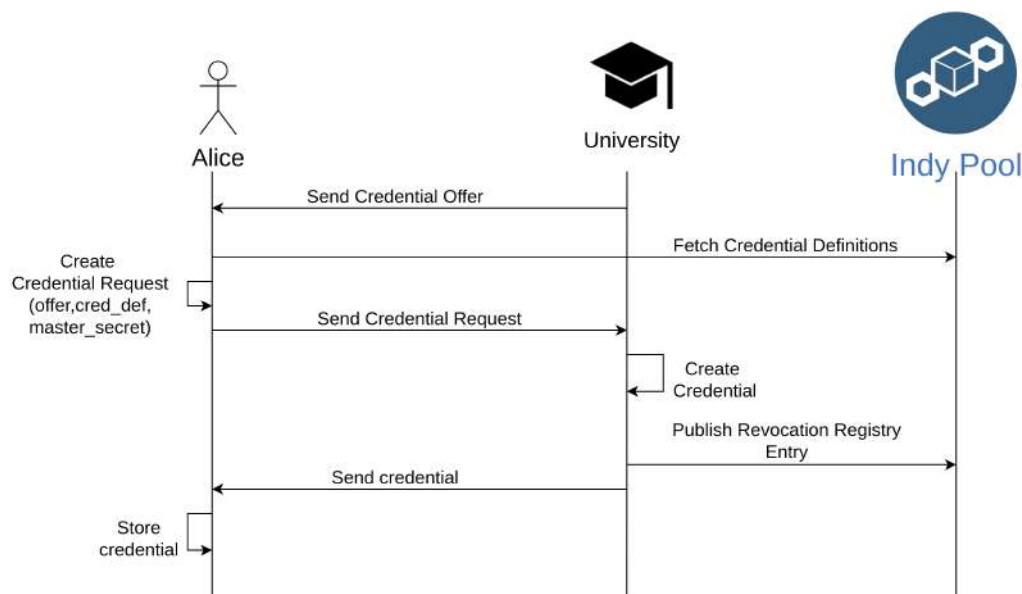


Figure 3.29: Sequence diagram of a credential issuing scenario. Using Student Mobility actors. In this case, the credentials are revocable. An update to the Revocation Registry accumulator is performed.

Alice as an Identity Owner could use her own Indy agent to ask for a "Student Identity credential" from the University of Pisa, in this case, a DID to DID communication can be set up, and the credential will be issued towards the holder_to_Endorser DID. **The DID of a holder during a credential request does not matter** since the DID of the holder is not necessary to generate a valid proof, but only its MasterSecret. As said in subsection 3.5.6, only the MasterSecret links an agent to its credentials.

After the reception of the Credential Offer, Alice fetches the Credential Definition of the offered credentials from the ledger. With a GET_CLAIM_DEF request the Credential Definition is taken from the ledger.

With the creation operation of the Credential Request (*proverCreateCredentialReq*), it is checked that the Credential Offer proposed Credential Definition matches the Credential Definition on the ledger. The check of the Credential Offer is done with the *key_correctness_proof* field of the Credential Offer. This protects the holder, in case of a malicious Endorser or an impostor Endorser who would issue a false credential. A false credential could be a credential without a Credential Definition on the ledger or a credential issued from an agent different than the one who created the Credential Definition.

As can be seen in figure 3.29, When issuing the credential the University of Pisa also updates the Revocation Registry accumulator on the ledger. If the credential is non-revocable this last step is not necessary. After having received the credential, Alice stores it in his Wallet. The issued credential does not contain Alice (holder) DID. As said before the only link between a credential holder and his credentials is his **MasterSecret**.

3.6.3 Creating a Proof

After having her Student Identity credential, Alice starts a DID to DID communication with the University of Copenhagen. In this case, the holder and the verifier communicate with DID to DID communication. The University of Copenhagen does not know the identity of Alice, nor that Alice is a student of the University of Pisa. Only Alice *DID_to_UniversityOfCopenhagen* can be seen from Alice.

After the reception of a proof request, Alice (holder) uses it to search for the credentials to use for the proof.

The Wallet can contain a large number of entries. To get a subset of entries, it can be queried when researching results (*CredentialsSearch.open(Wallet, query)*). The query has the same wql format introduced in section 3.5.8.

After getting a subset of entries, Alice can choose the credential she wishes to use to create the Proof. For example, Alice could have a Student Identity credential from another University but she wishes to use the University of Pisa credential. After the results are returned to Alice (holder) she can select the credentials to use in the proof (*CredentialsSearch.fetchNextCredentials(..)*).

Credentials were queried from the Wallet but were not immediately returned. Instead, a handle to retrieve them in the small-batches is

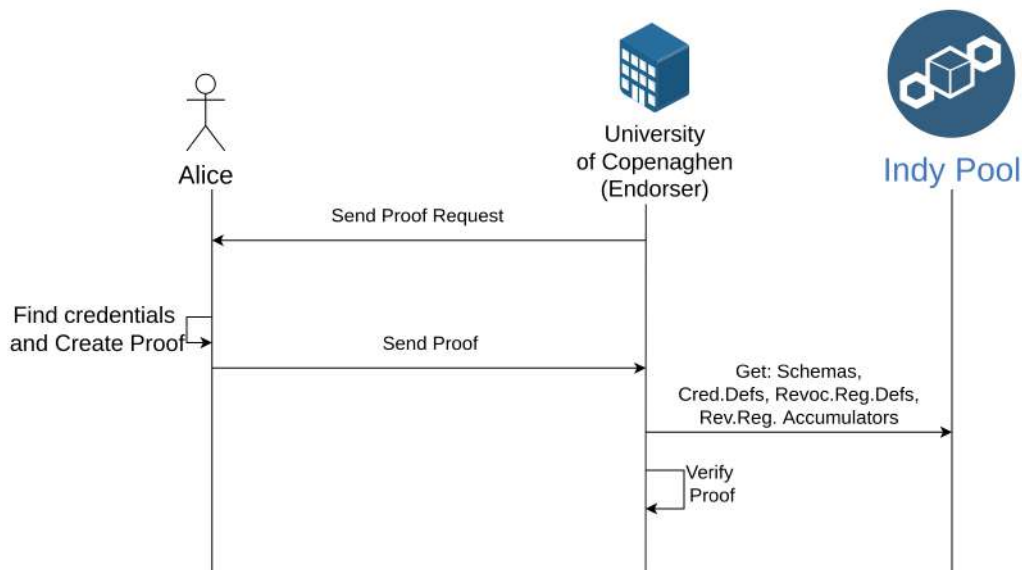


Figure 3.30: Sequence diagram representing: proof request, creation and verification.

given to Alice. The fetched credentials are used to create the proof. Credential Schema and Credential Definition of the fetched credentials are needed to create one proof. In the Student Identity case, the only credential used is the StudentIdentity credential. Alice can then create a proof with the method *proverCreateProof()*. During the proof creation the MasterSecret value will be used to prove that the credential is valid(it will be seen in subsection 5.8).

3.6.4 Verifying Proofs

Figure 3.30 shows how Proofs are received and verified. The computation to verify proof is completely done on the verifier side. After receiving one proof, the University of Copenhagen (verifier) gets all the needed Anoncreds entities to verify it (Revocation Registry accumulators are considered only if the credentials are revocable). All the Credential Schemas (GE_SCHEMA request), Credential Definitions (GET_CLAIM_DEF request), Revocation Registries (GET_REVOC_REG_DEF request) and Revocation Registry Accumulators (GET_REVOC_REG requests) are taken from the ledger.

After the verification is done, a method (*verifierVerifyProof*) returns a

boolean true or false if the proof is valid or not.

3.6.5 Conclusions

Using Verifiable Credentials eases the student enrollment process at a host university, for example filling out forms and searching for certificates from his home university to using credentials she stores in the Wallet. The student Alice can use the same credentials from the University of Pisa to enroll at different Host Universities.

Indy supports not only the portability of credentials but also privacy. Indy allows Alice to reduce the information shared with the Host University. Moreover, different host universities cannot correlate the information received from Alice to information not requested.

As can be seen in figure 3.31, the Student Mobility has the Session and RP unlinkability properties (introduced in subsection 3.5.10).

In figure 3.31, Alice sends two proofs. The first one contains a name and surname. These could be provided by attributes in Alice's birth certificate credentials (issued by a public office).

The second proof is derived from Alice's home university credentials. The proofs received will be correlated with the University of Copenhagen because Alice shares them in the same sessions. There is RP unlinkability since Alice could have changed the device used to communicate with Erasmus University from one proof to another.

Alice could repeat the enrollment process to different Host Universities, using the same credentials given by the University of Pisa. As said in subsection 2.5, identity the SSI paradigm allows identities to be portable. Not only with universities, but Alice can also use his Student Identity credential to create proof destined for various subject services, companies, government, etc.

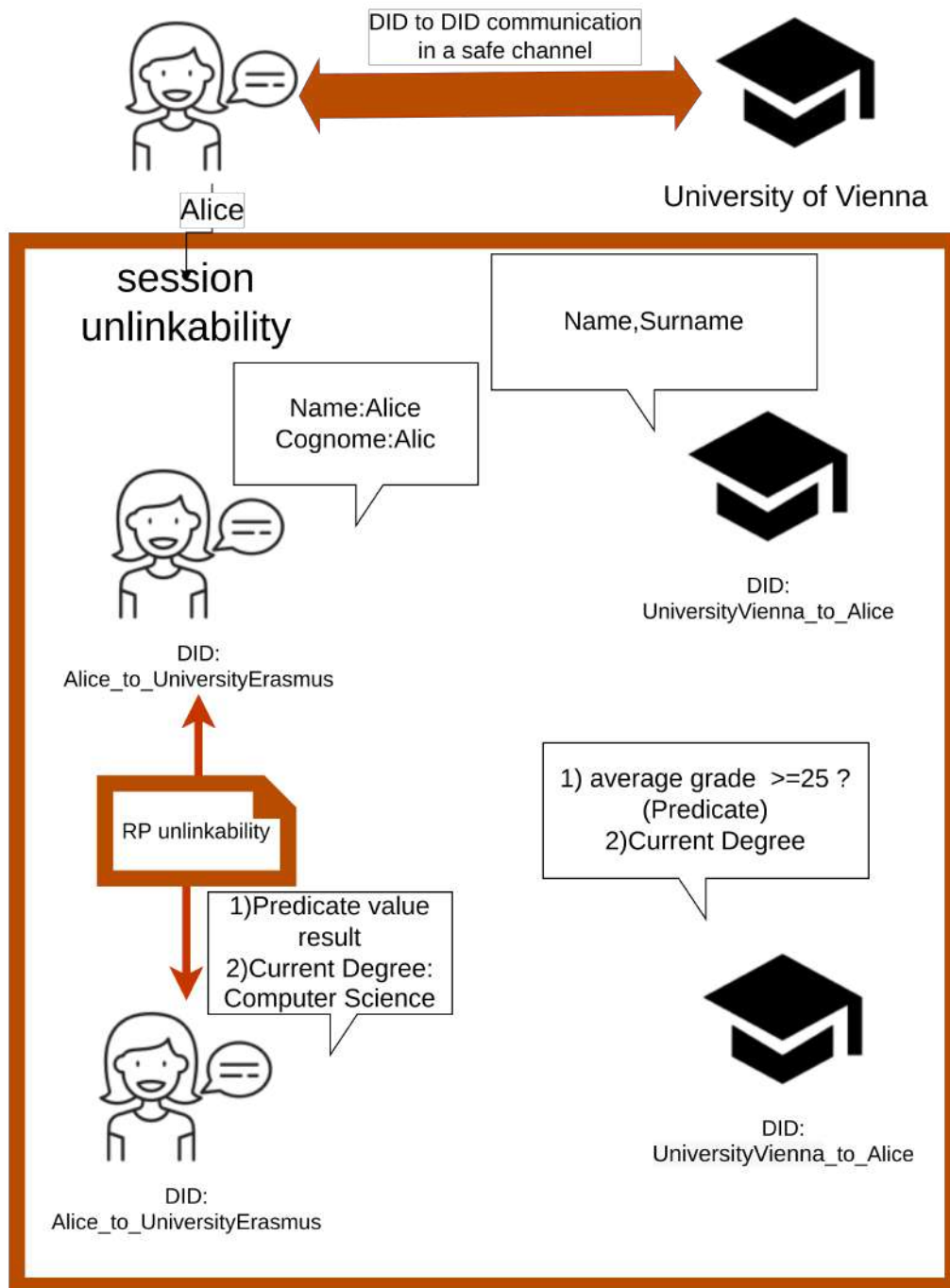


Figure 3.31: Proof presentation being seen as a message exchange. Having established a DID to DID communication, Alice sends proof with the requested attributes in a revealed form. It has to be noted that the University of Copenhagen uses a unique DID to communicate with Alice.

Chapter 4

Privacy preserving deliveries with SSI

This chapter presents the formalization of a privacy-preserving Shipping Verification System with SSI. Moreover, the chapter presents a comparison between the Student Identity case, which tackles the portability of the identity of a student, and the Shipping Verification System, which tackles the anonymous identity and right to be forgotten (from subsection 2.5) problem. Finally, the chapter provides an overview of the actors and their interaction in four phases: store creation, items ordering, handing of delivery to the deliverer, collection of the ordered items.

4.1 Portability versus privacy-preserving authentication

In the previous chapter, we defined the Student Mobility case, which served as an example of credentials that are easily portable and reusable with different verifies. Even if a certain degree of anonymity was provided by Indy with DID to DID communication and Zero-Knowledge proofs, preserving student privacy was not the main goal. Alice has to share her real name and attributes regarding her student career. We will see later in this chapter how Indy allows total to set up a privacy-preserving system for deliveries.

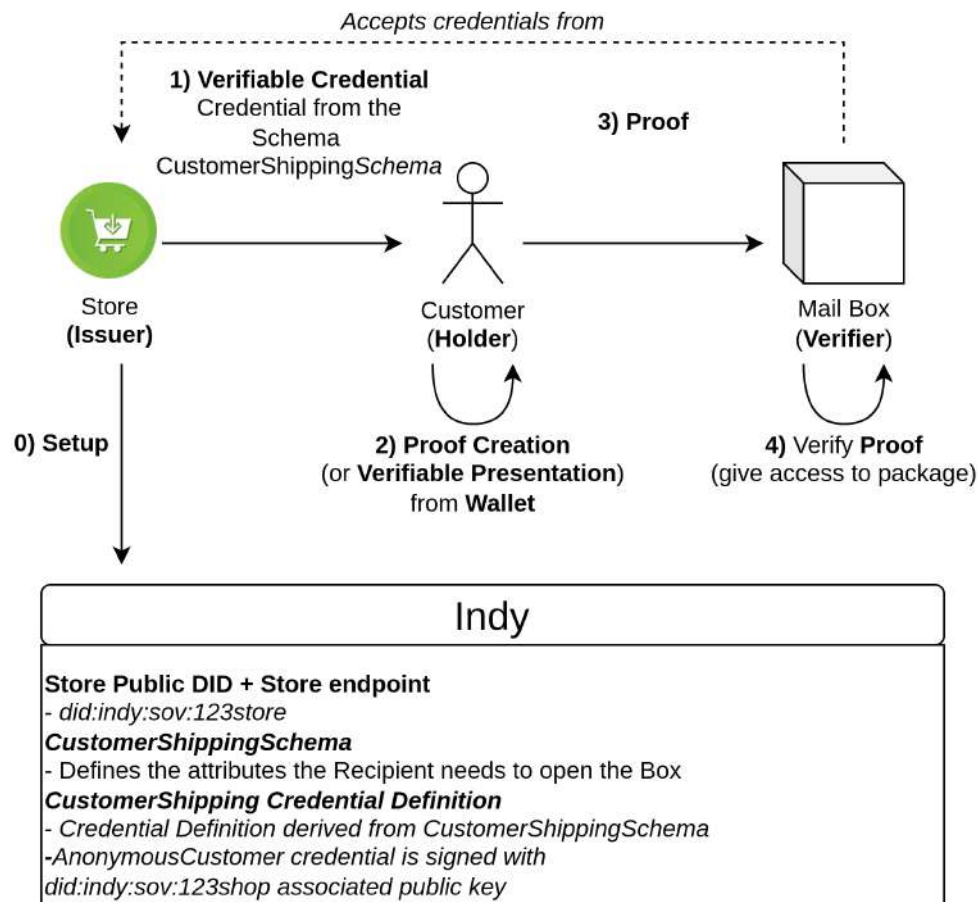


Figure 4.1: First concept for the Shipping Verification System.

4.2 Shipping Verification System: a privacy-preserving solution

In this section, we use Verifiable Credentials intending to preserve the privacy of a customer who ordered a package to be received by mail. We name this as **Shipping Verification System**.

4.2.1 Definition of the system

The Shipping Verification System allows a **Customer** to order from an Online **Store** one or more items. After having received an order, the

Store reserves a **Mail Box** to the Customer. The Mail Box stores the item for a limited time defined by the Store. From the arrival of the item, the Mail Box starts a timer that lasts for the specified time. The aforementioned Mail Box is not a personal Mail Box of the Customer but a parcel locker. Parcel Lockers are pickup points, in the form of a locker and provided with authentication software. Parcel Lockers [57] can be found scattered around a city, usually managed by one company (for example Amazon with its Amazon Lockers). They provide an authentication system (e.g: insertion of a PIN code) to allow only receivers to collect their package (the parcel lockers have more than one storage department).

Figure 4.1 shows the mapping of the real case to Indy. The Store takes the role of an issuer, the Mail Box takes the role of the verifier, while the authentication system is based on Indy proofs. The Customer receives credentials from the Store and uses them to create valid proofs to open the Mail Box. The Mail Box verifies the proofs and if proof is valid, the Mail Box opens the storage department that contains the Customer order.

The Store manages the delivery, reserving a Mail Box and delegating a deliverer. We enrich the first concept of the system by introducing the **Shipping Deliverer**, who is responsible for delivering the item to the Mail Box. The Shipping Deliverer interacts with the Mail Box software and inserts the item into it. To insert an Item inside the Box, the Shipping Deliverer has to prove that he is authorized to do it. The Store creates a credential for the Shipping Deliverer to use for authentication. The Mail Box verifies the proof from the Shipping Deliverer and opens the Box.

Once the item has been deposited, the Customer has to go to the Mail Box before the time frame elapses. The Customer has to prove he was the subject for delivery. After the purchase, the store issues to the Customer a credential for opening the Box after the order is made. Similar to the Shipping Deliverer case, the Customer gives proof of access to the Box. Figure 4.2 shows the system enriched with the Shipping Deliverer.

The rest of the section describes all the steps in detail. In this section, we will omit the credential issuing process to Customer and Shipping Deliverer, as it is the same of subsection 3.6.2.

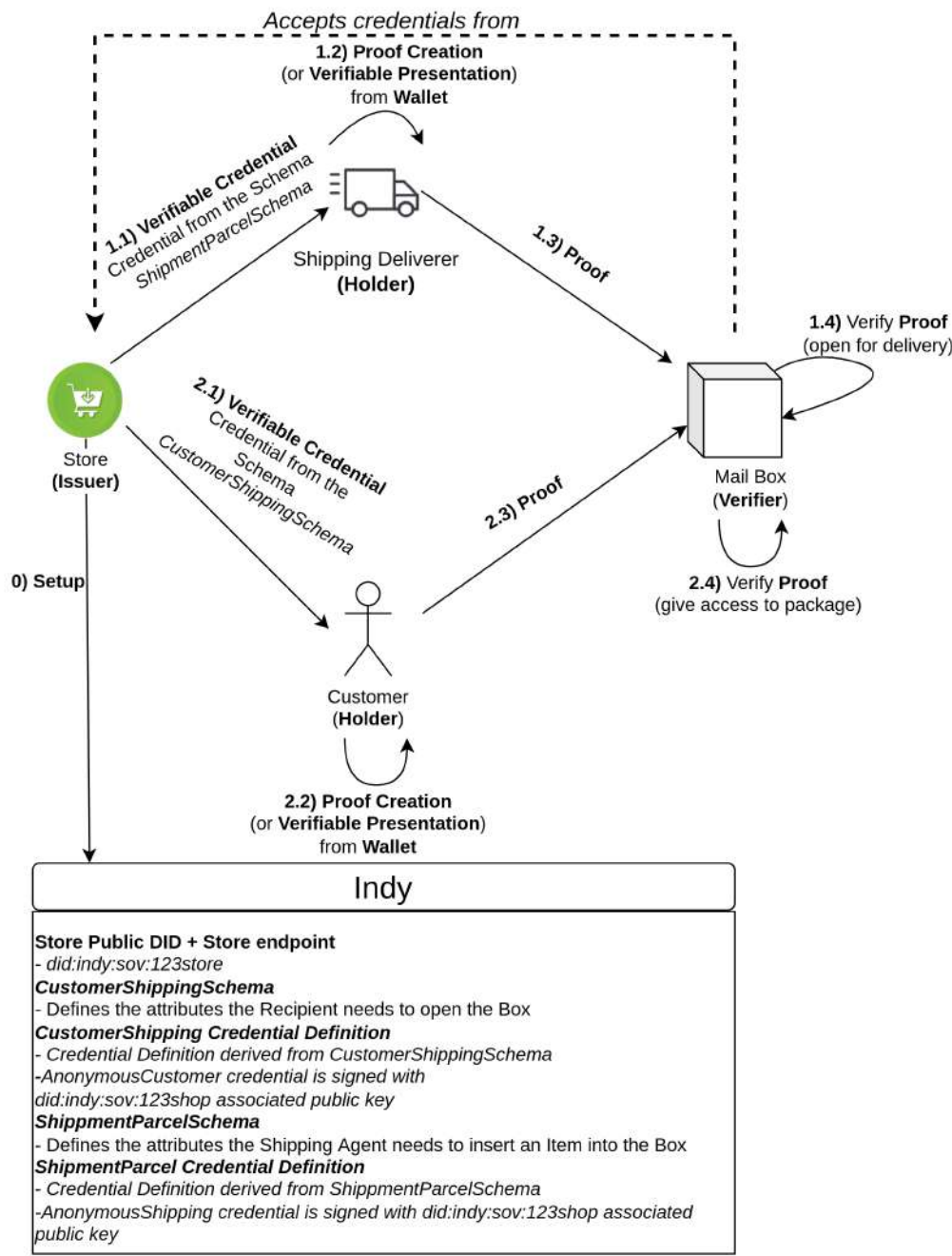


Figure 4.2: Shipping Verification System overview enriched with the Shipping Deliverer. The Store issues two different types of credentials, for Customer and Shipping. The table at the bottom lists the Anoncreds entities from Indy.

4.2.2 Store creation

The Store requires on the ledger a NYM record assigning it the role of Endorser and storing its DID to be available to customers. To get a NYM record on the ledger the Store can connect to the Pool and search for a Steward DID. In this case, a Steward could be any organization running a node.

After getting a Steward DID, the Store makes a GET_ATTRIB request and takes its endpoint from the ledger (figure 4.3). The Store receives the Steward endpoint as a response (if the request was successful).

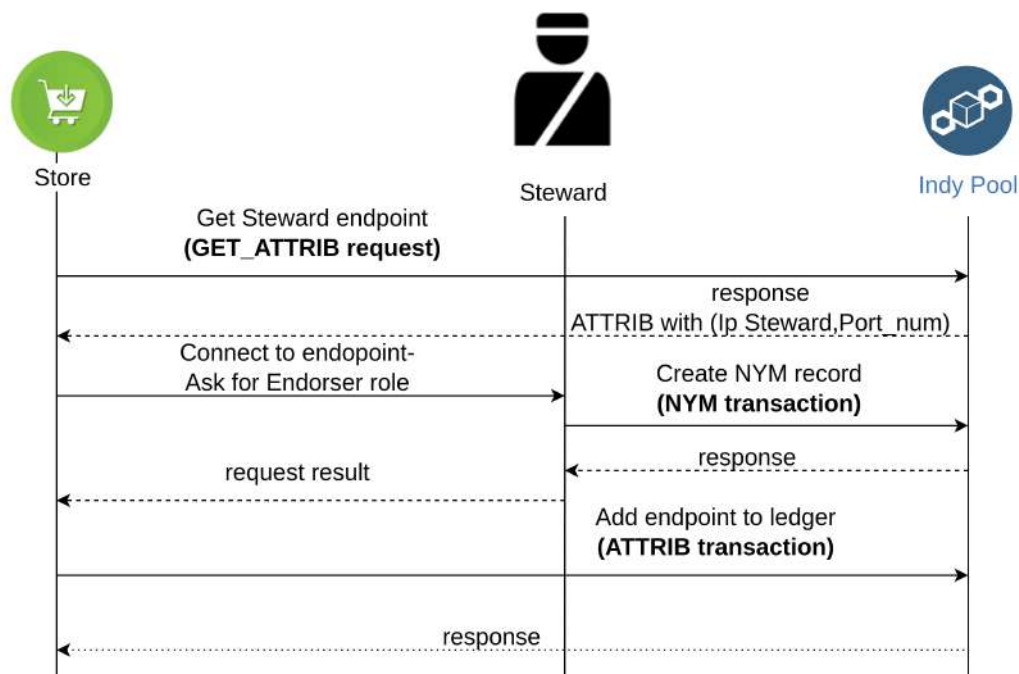


Figure 4.3: Sequence diagram of store creation and setup as an Endorser.

After having received from the Steward the response of successful NYM record creation, Store submits an ATTRIB transaction to the pool. Its endpoint along with its Endorser DID are on its public NYM record. The Store now can be reached for DID to DID communication by Customers. It has to be noted that the Store DID will be public. If the Store does not want to be associated with more than one item order, it has to ask for a different NYM record for every order. We assume that the

Store uses the same NYM for every order.

4.2.3 Ordering an Item

Following the same process of a Store connection with a Steward, a Customer gets the Store endpoint from the ledger.

To preserve his identity, the customer creates a unique DID and establishes a DID to DID communication with the Store. Afterward, the Store sends the Customer the list of available items. This process is illustrated in figure 4.4.

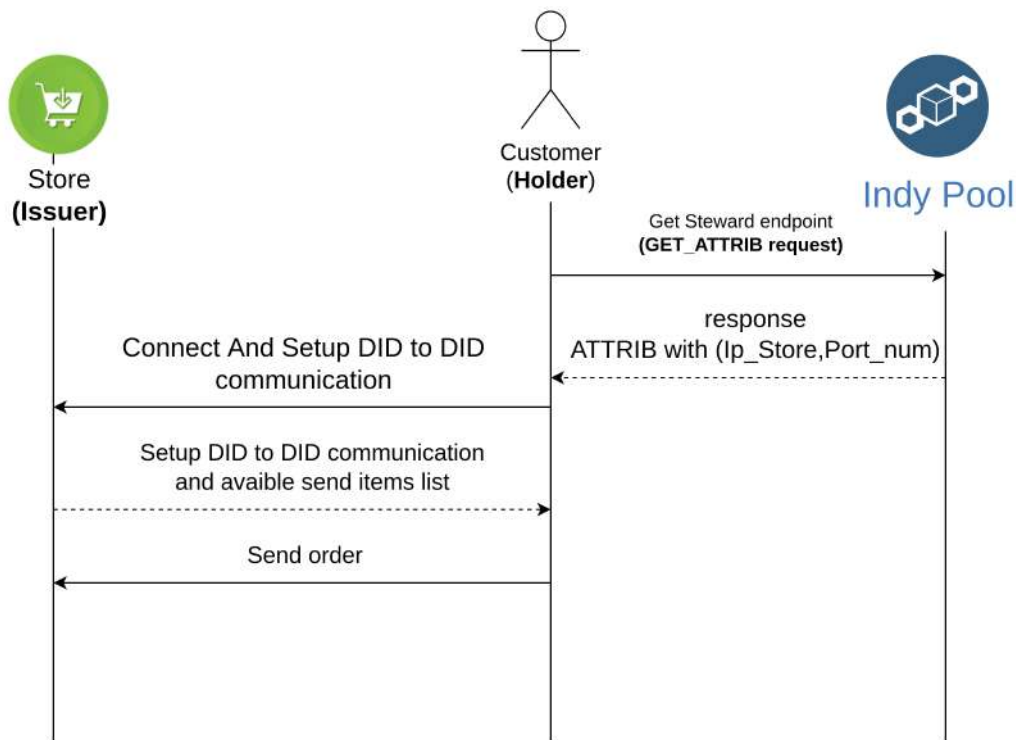


Figure 4.4: The Customer sets up a DID to DID communication with and orders one or more items.

4.2.4 Reserve a Mail Box

After the Store receives an order, it has to reserve a known Mail Box close to the customer. We assume that the Customer does not disclose their address to the Store, but a larger area (e.g. a neighborhood).

Mail Boxes support a software agent that manages the communication with Stores and does not have any information on the ledger (i.e. they have no public DIDs) to not fill the ledger with Mail Boxes NYM records and to speed up the setup of new Mail Boxes. Since a Mail Box does not need a public NYM record, it can be quickly introduced into the system, and it is not necessary to communicate with a Steward for a Mail Box NYM record. We assume the Store owns some Mail Boxes and stores in a Database the endpoint of the Mail Box agents.

The Store chooses a currently empty Mail Box and reserves it to deposit the item ordered by the customer, setting up the Mail Box to be a Verifier for both the Shipping Deliverer and the Customer. Figure 4.5 shows this setup process and the information the Store sends to the Mail Box. After having received a Success response, the Store can delegate

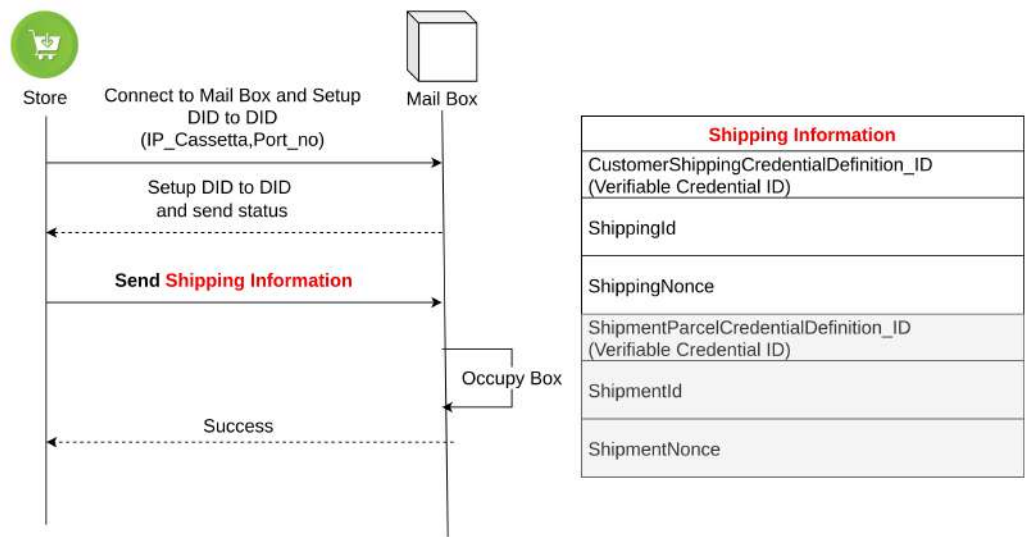


Figure 4.5: After an order is placed. The Store reserves a Mail Box to store the delivery. The Store provides the necessary information for proof verification to the Mail Box (included in **Shipping Information**).

the delivery to a Shipping Deliverer.

4.2.5 Delegate a Shipping Deliverer

To deliver an Item to the Mail Box the Store has to delegate the Shipping Deliverer. We name the credential *ShipmentParcel*, which allows the Shipping Deliverer to open the Mail Box and deposit the item.

ShipmentParcelSchema

The Shipment Parcel credential schema can have different attributes depending on the Store. We identify the following as the minimal subset of attributes that each Store must guarantee. A Store uniquely identifies a delivery by a **ShipmentId** and a **ShipmentNonce**. ShipmentId is the value assigned to the delivery by the Store. To allow the Store to reuse a ShipmentId, for example in case of a lost item, a ShipmentNonce is generated only for the current delivery. The values ShippingId and ShippingNonce (with equal properties) will be created separately for the customer credential.

The Store provides both the ShippingDeliver and the customer of the Mail Box endpoint for communication. The ShipmentNonce is generated with RNG introduced in subsection 2.1.2 or it could be the timestamp from the order time. As can be seen from figure 4.5, ShipmentId and ShipmentNonce are provided to the Mail Box to use for verification.

An attribute **shipmentavailabilitytime** is necessary if the Store wants to control the time frame in which the Mail Box contains the items after delivery. The shipmentavailabilitytime inside the credential guarantees that the ShippingDeliver cannot communicate a different availability time value to the Mail Box. If the Store communicates the availability time without bundling it in the credential, the Shipping Deliverer could give a different value to the Mail Box.

During the verification phase with the Shipping Deliverer, the Mail Box asks in its proof request the availability time specified by the Store. The proof **attr_info** structure can be seen in figure 4.6. To make sure that the delivery is the one the Mail Box expected, restrictions are applied to the credential attributes. The proof request imposes restrictions on the ShipmentId and ShipmentNonce value. The values of the credential must match those on the Shipping Information message received from the Store. The restriction *"attr::attribute name::value":"attribute raw value"* is used to enforce the equality of attributes (introduced in subsection 3.5.8). We assume that the Mail Box agent can be trusted and sets the availability time to the one provided. An example of the credential ShipmentParcelCredential, which satisfies the schema ShipmentParcelSchema, can be seen in figure 4.7.

The Shipping Deliverers can accept more deliveries from more stores before delivering them. The Shipping Deliverer is not bound to handle one delivery at a time. A Shipping Deliverer could stack more deliveries

Proof Request {attr_info}:
Names:{ShipmentNonce,ShipmentId,shipmentavailabilitytime}
restrictions{ ShipmentParcelCredentialDefinition_ID =ShippingInformation.ShipmentParcelCredential.Definition_ID, attr::shipmentid::ShippingInformation.ShipmentId, attr::shippingnonce::ShippingInformation.ShipmentNonce}

Figure 4.6: All the requested attributes are contained in the field Names of in the attr_info structure. The attributes must come from the same credential and respect the restrictions imposed. The restrictions are equality restrictions on attributes of the credential to provided values.

ShipmentParcel credential
shipmentid:(int)Store order ShipmentId
shipmentnonce:(String) Unique Shipment Identifier
shipmentdeclaredvalue:(String) declared items value
shipmentavailabilitytime:(String) available time for items pickup
shipmentweight:(String) Shipment weight

Figure 4.7: ShipmentParcel credential with ShipmentId and Shipment-Nonce, to be used in proof. Shipmentavailabilitytime attribute integrated into the credential. Other attributes regarding the shipment (not needed by Mail Box)

to do at a later time. The process of items to the Mail Box delivery as illustrated in this section can be seen in figure 4.6.

Items delivery

The Shipping Deliverer discloses the shipmentavailabilitytime attribute to the Mail Box. After the proof verification, Mail Box starts a timer that lasts until the end of the available time and opens itself for deposit. The items will be available until the timer reach its end.

Proof Request {attr_info}:
Names:{ShipmentNonce,ShipmentId,shipmentavailabilitytime}
restrictions{ ShipmentParcelCredentialDefinition_ID =ShippingInformation.ShipmentParcelCredential.Definition_ID, attr::shipmentId::ShippingInformation.ShipmentId, attr::shippingnonce::ShippingInformation.ShipmentNonce}

Figure 4.8: Structure of the proof request sent to the Deliverer

4.2.6 Customer retrieving the items

The Store issues a credential `CustomerShippingCredential` to the Customer, similarly to the credential `ShipmentParcelCredential`, to open the Mail Box. When the Customer arrives, the Mail Box requests to the Customer, with a proof request similar to the request shown in figure 4.8, the `ShippingId` and the `ShippingNonce` equal to those sent by the Store as `Shipping Information`. The same mechanism of attributes equality of raw values from the `Shipping Deliverer` proof request is used. We distinguish three cases for Mail Box opening verification failure:

1. Customer does not present a valid proof: the Customer could have the right attributes but not a valid proof (created following the procedures of subsection 3.6.2). In this case, the `Indy verifyproof()` method returns false. An example of non-valid proof would be an altered valid proof with changed attributes. An example of valid proof using a `CustomerShipping` credential and non-valid proof that alters it can be seen in appendix A.3.
2. The customer presents a proof to the wrong Mail Box. It means that the items inside the Mail Box are not those ordered by the Customer. This situation could also happen if the Customer tries to collect its items after the availability time its order expires. The proof presentation is done after the time frame of shipment availability ends, in this case, the Mail Box could contain a package from another delivery. For example, the Customer who holds credentials with `ShippingId_=1` and `ShippingNonce=1234` was late and upon approaching the Mail Box, it submits a proof request asking for with `ShippintId_=2` and `ShippingNonce_=3341`. After receiving

the proof request, the Customer cannot create a valid proof. As he will not find in the Wallet the credentials for creating the proof with the given restrictions.

3. There is the case that the Customer arrives to collect its items after their availability time expires. In this case, the items stored in the Mail Box are the ones from the Customer's order. Even if the proof verification is successful, the Mail Box does not open, as the Customer arrived too late to collect them. The Mail Box always checks that the availability time of the Items did not elapse. If it does, the Mail Box does not open and warns the Customer about late pick-up.

The process from proof request to Mail Box opening can be seen in figure 4.9. The Customer agent could keep stored in his Wallet the *CustomerShippingCredential*, e.g. it could be used to prove he made a purchase, in the case of a warranty. In our case, we assume that the *CustomerShippingCredential* is a one-time credential used to open the Mail Box. The process to search for a credential could be computationally intensive when a Wallet is full. A one-time credential takes unnecessary space after being used. The Customer can delete the credential from the Wallet.

4.3 Shipping Verification System and SSI

Drawing a parallel to the Student Mobility case, the credentials are for one-time use and do not reveal information about a Customer's Identity (anonymity). The *CustomerShippingCredential* has significance only towards one Mail Box and one Store (no portability).

This Shipping Verification System gives to the Customer (and the Shipping Deliverers too) the "right to be forgotten" mentioned in section 2.5. Customers' identity is stored on the Mail Box software only from the time of Items order to the time of item collection. The only information the Customer discloses to the Mail Box is the *ShippingId* and the *ShippingNonce*. The most sensitive information the Customer has to share is the area where he wishes the Mail Box to be. The information shared is kept at a minimum. In this case the Selective Disclosure and Zero-knowledge proof allowed to disclose only the necessary information. There was a Selective Disclosure of the attributes, given the credentials

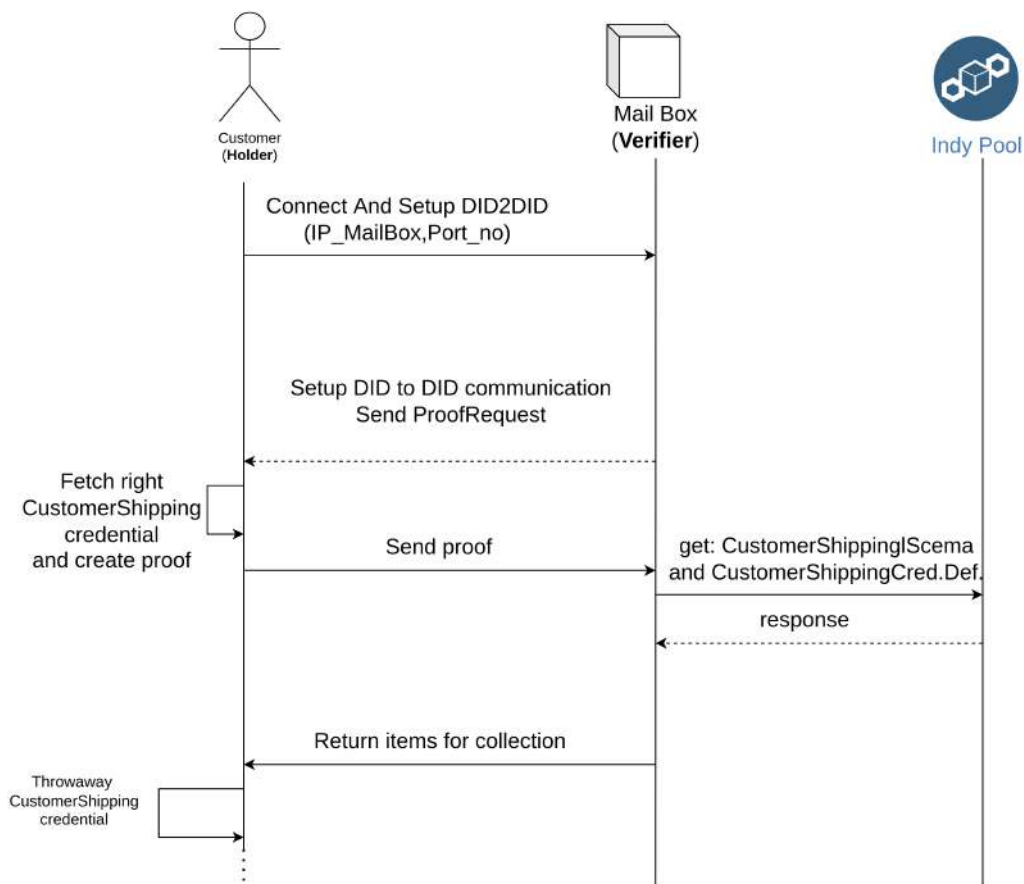


Figure 4.9: The proof verification process is equal to the Shipping Deliverer one. After the verification of the proof, the Mail Box checks that the Customer arrived on time for pick up. After the Customer collects the Item he removes the credential from his Wallet.

only a subset of their attribute was used in the proof.

The MailBox agent never receives the credentials from the Shipping Deliverer and the Customer. Instead only the derived proof is transmitted to the Mail Box. This could theoretically protect the Customer from an attacker, wanting to steal his credential. The credential stored never leaves the Customer's Wallet. As an example, the Shipping Deliverer was able to not disclose the shipment declared value to the Mail Box, even if the requested attribute were from the same credential (figure 4.7) containing it.

Indy credential and proof requests with restrictions on Credential Def-

inition and values allow us to use the `ShippingId` and `ShippingNonce` to identify an order. Alterations to the `ShippingId` and `ShippingNonce` values from the credential would render it not valid.

4.4 Threats - what could go wrong?

Given the fact that credentials are traceable on the ledger and the use of DID to DID communication, there could be two major problems in the actors' relationships:

- Lack of non-repudiation. As said in subsection 3.4.4, with DID to DID communication the concept of non-repudiation is not valid. This could be a problem in a case of a malfunctioning Mail Box. Since the communication with a Mail Box agent is DID to DID, it is impossible to prove that a faulty Mail Box sent a certain message. A faulty Mail Box agent could declare that the Mail Box is empty, while it is not and the timer has not elapsed. The Mail Box agent cannot be declared liable for its wrongdoings.
- Single point of failure. If the Indy pool goes down, the Mail Box cannot fetch Anoncreds entities for proof verification. The unreliability of the ledger also means no way to get the endpoint of a Steward. With the pool down the Store cannot publish credentials definitions to create a credential on the ledger. In this case, it is better to minimize the ledger involvement. For example, the Store could give the Box the Anoncreds entities required to verify the proofs.

This section investigates the system behavior analyzing possible malicious actions. To better list the threats we introduce the following actors:

Alice honest customer who would like to try anonymous delivery.

PennyStore honest store. Has a NYM record on the ledger after being approved by a Steward.

ScamStore malicious store.

VictorBox honest Mail Box agent.

WendyBox malicious/malfunctioning Mail Box agent.

Trudy malicious actor, tries to impersonate a customer.

OscarShippingDeliver ShippingDeliver that tries to not follow the Store directives.

JohnShippingDeliver honest ShippingDeliver, follows Store instructions.

Table 4.1 and table 4.2 list possible malicious actions performed by attackers, under specific assumptions, and describe the countermeasures. The following subsection highlights the commonalities of threats and an evaluation of them.

Threats contextualized and conclusions

As highlighted by the first two entries of table 4.1 the Shipping Verification System requires the Stores to be honest. This characteristic of the need for authorities, to be honest, can be expanded to all Verifiable Credentials Data Model systems and in turn any Indy system. Steward, Trustees, and Endorser have to be trustable for a system to work. It could be possible to make the system more reliable and protect the Customers from dishonest Stores by using interledger approaches as in the second entry of table 4.1.

The third entry of table 4.1 and the second and third entry of table 4.2 show how the system is robust against credentials theft and how it provides solid privacy preservation.

Conclusions

The system is in regards to Verifiable Credentials. The system's main objective is preserving the privacy of a Customer. The objective is accomplished without threats being an issue. While issues of Store and deliver trustability can be generalized to all e-commerce services. *The system relies on Steward's judgment in appointing honest Stores as issuers.*

Assumptions	Attacker	Goal	Countermeasure
Stores are accurately scrutinized by Steward before being granted a NYM record	Scam Store	Use the system to collect orders and payments without delivering the items	Any Steward will not allow in the system a store that does not to some trustfulness criteria
ScamStore has passed the trustfulness check. Alice is honest	Scam Store	Collect orders and payments without delivering items or sell to multiple customers the same item	To protect herself Alice could ask to use a Smart Contract (introduced in section 2.4). Using a Blockchain like ethereum or Hyperledger Fabric. The Smart Contract would give a payment to ScamStore only after the Mail Box has been opened and the items collected by Alice
Alice, PennyStore, VictorBox are honest.	Trudy	Trudy sniffs Alice's credential and tries to use it	Since Trudy does not have Alice MasterSecret she cannot create a valid proof. It will be shown in subsection 6.1.1 that without the MasterSecret is not even possible to store the credential before proof creation.
PennyStore is honest	OscarShippingDeliver	OscarShippingDelivers does not deliver the shipment.	PennyStore could make an untrusted Shipping Agent pay a fee before delivery. After the delivery is done, a Smart Contract could give the Shipping Deliverer his payment plus the fee refund

Table 4.1: Threat table

Assumptions	Attacker	Goal	Countermeasure
PennyStore is honest	WendyBox, Trudy	WendyBox gives the items to Trudy	The only countermeasure would be to check for a Mail Box integrity before reserving it for a delivery
Alice is honest, VictorBox is honest.	ScamStore	Tries to use the credential issued to Alice to open Victor Box	As in case 4 any proof without Alice MasterSecret would fail
JohnShippingDeliver is honest	Trudy	Trudy tries to uncover JohnShippingDeliver identity to get the Shipment before delivery	Using secure channels and DID to DID communication, it is not possible to uncover its identity

Table 4.2: Threats table

Chapter 5

An implementation with Hyperledger Indy

This chapter focuses on the Shipping Verification System implementation. There is a setup phase required for the necessary components (Indy-SDK, libindy, nodes, Docker, Indyscan, Java Wrapper). The setup process can be seen in appendix A.5. The codebase illustrated in this chapter can be found at github.com/caltr98/IndyWorking¹. The implementation is based on a local Indy pool. Nonetheless, after a certain stage of development, the implementation was deployed on the Sovrin BuilderNet too. References to the Anoncreds entities in Sovrin (seen with Indyscan) as hyperlinks will be visible to PDF readers. Meanwhile, the links will be in the footnotes for paper readers.

5.1 An additional layer of abstraction: IndyLibraries

Indy Java wrapper is split into packages each containing .java classes with methods relevant to the package. The list of packages is

- anoncreds: provides methods to implement the Anoncreds workflow (illustrated in subsection 3.6).
- blob_storage: provides methods to access the Tails file, introduced in subsection 3.5.4.

¹<https://github.com/caltr98/IndyWorking> - IndyWorking repository

- **cache**: Allows to access saved results, e.g. ledger read request results in the Indy Java wrapper cache. In case of a cache miss, an operation to retrieve the result is performed.
- **crypto**: contains the cryptographic methods provided by Indy, including the *packMessage()* and *unpackMessages()* for DID to DID communication (from subsection 3.4.4).
- **did**: provided methods to create and manage DIDs.
- **ledger**: provided methods to perform read requests or write requests (transactions).
- **metrics**: provides methods to access the underlying libindy performance metrics.
- **non_secrets**: provides methods to query data from Wallet.
- **pairwise**: contains methods to manage pairwise DID.
- **payments**: contains methods related to the Indy SDK payment interface.
- **pool**: provided methods to configure and start communication with a pool of nodes.

Agents can use any of the methods present in the classes contained in those packages. Providing an additional layer of abstraction in the implementation allows for the creation of agents that are specialized in a particular Indy role (roles from subsection 3.4.3).

During the Shipping Verification System implementation, to create such a layer, the package **IndyLibraries** was developed, a component of the IndyWorking repository linked at the start of this chapter. IndyLibraries are composed of classes that are mapped to a specific Indy role. Each class can call only a selected subset of the methods defined by the wrapper. Every java class is a specialization of another. As can be seen by figure 5.1 a base agent is defined in class Agent of IndyLibraries. An agent defined by the Agent class provides the methods for the operations allowed to an agent without a NYM record in the ledger or a none role (the roles distinctions are illustrated in subsection 3.4.3). Other classes extend it by providing more methods for the capabilities allowed by the agent role. Trustees and Endorsers can do all the Agent operations and

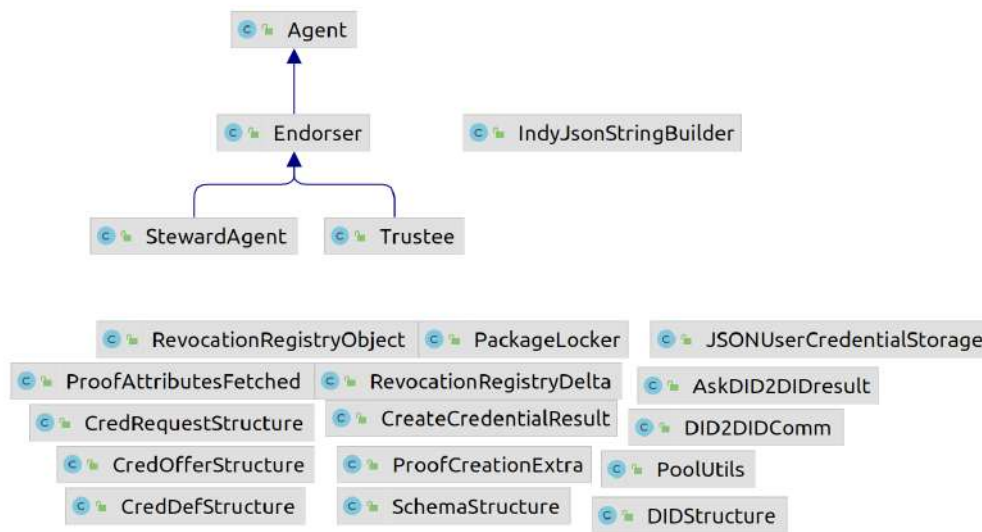


Figure 5.1: Class Diagram providing an overview of the IndyLibraries classes

more, and as such, they are a specialization of the class Agent.

Class [IndyJsonStringBuilder](https://github.com/caltr98/IndyWorking/blob/master/src/main/java/IndyLibraries/IndyJsonStringBuilder.java)² contains methods to create custom Json structures. Various Json structures are arguments for Indy Java wrapper methods. Another class to highlight is **DID2DIDComm**, which provides static methods to set up DID to DID communication and send/receive messages in a DID to DID Communication. The Json structure provided for the methods must precisely respect the format defined by the wrapper. A slight error in the Json format could cause a never-ending stream of errors. Other classes are used to aggregate the results from Indy Java methods. In the following sections, the Indy Java wrapper will generally be named the **wrapper** for conciseness.

The following sections describe the implementation of the **Shipping Verification System with Hyperledger Indy**, presented in section 4.2 and using code snippets from the codebase linked at the beginning of this chapter 5. Only the processes of figure 4.2 will be covered. Hyperlink references pointing to a method code (on GitHub) are also highlighted in the chapter. Each agent runs a different process and communicates through the use of Java sockets. To allow for total privacy preservation,

²<https://github.com/caltr98/IndyWorking/blob/master/src/main/java/IndyLibraries/IndyJsonStringBuilder.java> -IndyJsonStringBuilder class

it is advised that agents use mechanisms to not disclose their IP, such as a proxy, VPN, or Tor browser for connections.

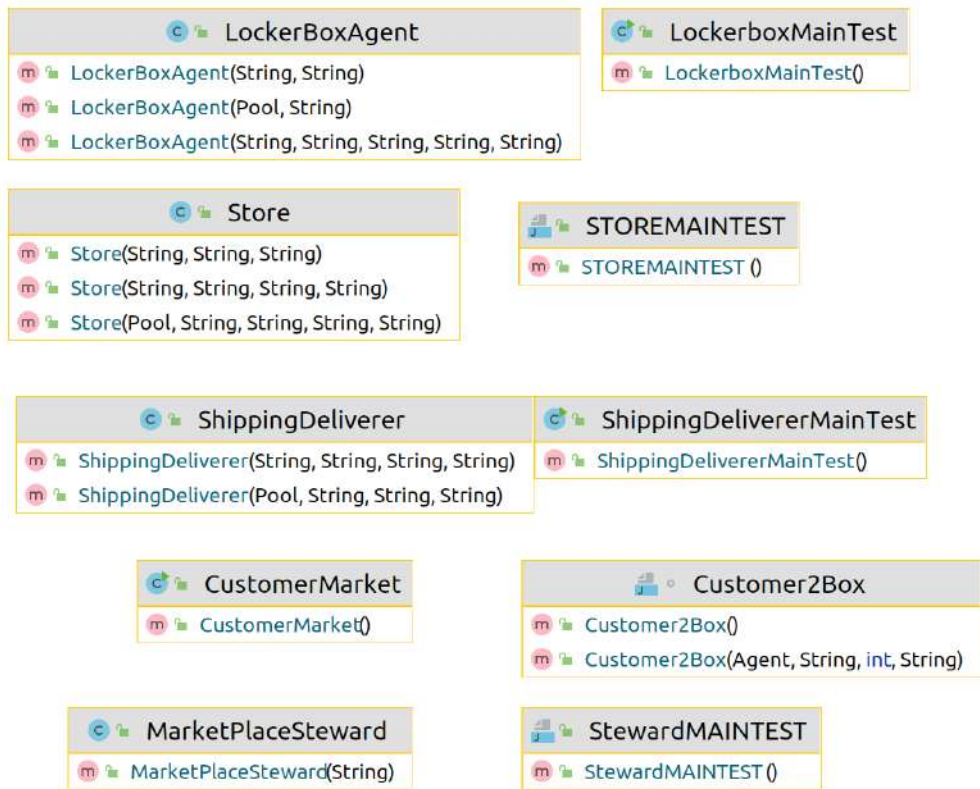


Figure 5.2: Class Diagram providing an overview of the Shipping Verification System implementation. The agents' classes are alongside their process main class.

5.2 Steward setup implementation

Every system based in Indy has Stewards deploying a node. Stewards do not actively participate in the Shipping Verification System. Nonetheless, a Steward agent appointing Stores as Endorser is essential to the system.

A Steward agent is defined in the class `MarketPlaceSteward`³. The

³<https://github.com/caltr98/IndyWorking/blob/master/src/main/java/MarketPlaceSteward.java> - MarketPlaceSteward class

Steward process uses as Main classe the class `StewardMAINTEST`⁴. The `MarketPlaceSteward` class diagram can be seen in figure 5.2.

`MarketPlaceStewardTaskHandler` is a task that runs on a thread that handles foreign agent requests. It locally setups the DID to DID communication with the and for the Store, it forwards its NYM requests to the ledger. Inner Class `MarketPlaceStewardRequestsHandler` is a task running on the main thread of the Steward agent. `MarketPlaceStewardRequestsHandler` task receives requests and forwards them to `marketPlaceStewardTaskHandler` task. The `MarketPlaceSteward` class diagram in detail can be seen in figure 5.3.

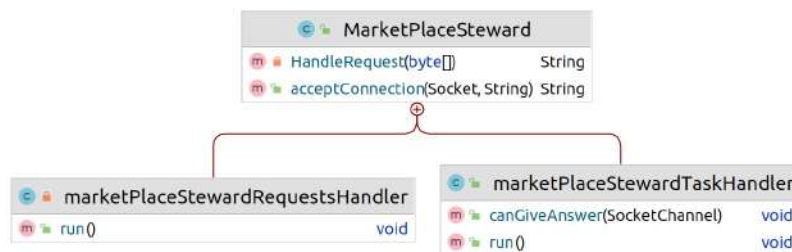


Figure 5.3: Class diagram of `MarketPlaceSteward`.

In the genesis transaction of the pool, some NYM records can be predefined. A NYM record contains a DID and its verkey. As said in subsection 3.4.3 a DID can be generated by the seed from which it derives. An agent having the DID seed can generate it and its pair of keys. After the pool setup, it is possible to retrieve the DID of predefined agents with this seed mechanism. The Steward needs to connect to the pool before, being it can approve Stores as Endorsers. The connection process is illustrated in the following subsection.

5.2.1 Connection to a pool

An agent has to create a pool configuration, which returns a handle, and then connects to the pool. The Steward agent constructor method (`public MarketPlaceSteward(String poolName)`) contains the operation to create a configuration and subsequently connect to the pool.

The method `createPoolLedgerConfig()` called inside the constructor to

⁴<https://github.com/caltr98/IndyWorking/blob/master/src/main/java/MarketPlaceSteward.java> - `StewardMAINTEST` class

produce a configuration. To create a configuration the genesis transaction of the pool is needed, and the configuration data is stored internally by libindy. The configuration is referred to by its name. The config parameter consists of a Json structure containing a runtime pool configuration. Setting the field `config="{ }"` allows to use of the default pool runtime configuration. Providing a different value to the field `config` would have generated a custom configuration.

The `PoolUtils`⁵ class in an adaptation an Hyperledger sample [58] has two methods to create a configuration. The method `createPoolLedgerConfig(String poolName, String pathToGenesis)` allows the creation of a configuration using a text file stored in the local file system. The method `createPoolLedgerConfig()` uses method `createGenesisTxnFile()` to write the Indyscan pool genesis transaction, allowing it to be used in a configuration creation. The `createPoolLedgerConfig()` method calls a wrapper method (`Pool.createPoolLedgerConfig()`⁶) to create a configuration.

```
Pool.createPoolLedgerConfig(DEFAULT_POOL_NAME, createPoolLedgerConfigJSONParameter.toJson()).get();
```

In this case `DEFAULT_POOL_NAME` is "INDYSCANPOOL". The configuration could have a different name than the pool.

5.2.2 Generating a Steward NYM

For INDYSCANPOOL the Steward DID is generated from a seed⁷, while the Steward object is created inside the `MarkerPlaceSteward` constructor. Before creating a DID, the Steward has to create and open a Wallet.

Wallet creation and opening

Wallet creation is implemented by the methods in the `IndyLibraries.Agent`⁸ class:

```
public Boolean CreateWallet(String WalletName, String
    WalletKeyPassword)
```

⁵<https://github.com/caltr98/IndyWorking/blob/master/src/main/java/IndyLibraries/PoolUtils.java> - PoolUtils class

⁶<https://github.com/hyperledger/indy-sdk/blob/master/wrappers/java/src/main/java/org/hyperledger/indy/sdk/pool/Pool.java> - Indy wrapper Pool class

⁷The seed used is 00000000000000000000000000000000Steward1.

⁸<https://github.com/caltr98/IndyWorking/blob/master/src/main/java/IndyLibraries/Agent.java> - Agent class

```
public Wallet OpenWallet(String WalletName, String
    WalletKeyPassword)
```

Such methods call the methods from the [wrapper Wallet class](#)⁹, i.e. [Wallet.createWallet\(..\)](#) and [Wallet.openWallet\(..\)](#).

DID creation based on seed

After a Wallet is created, the following method from `IndyLibraries.Agent` class creates and stores the Steward DID in the Wallet.

The method [createDID\(\)](#) creates a DID derived from a predefined Steward seed.

5.2.3 Making the Steward endpoint public with an ATTRIB transaction

After a DID creation, the Steward constructor executes the following code before terminating.

```
//add endpoint to NYM Object in Ledger to talk with Steward
//inserting address 127.0.0.1 for localhost
indySteward.addENDpointToNYM("did2did","127.0.0.1:"+
    listeningAddress.getPort());
Thread th = new Thread(new marketPlaceStewardRequestsHandler(pool
    ,AgentListenChannel,this));
System.out.println("gettingPoolStats" + indySteward.
    GetvalidatorInfo());
th.start();
```

Method [addENDpointToNYM](#) is from [IndyLibraries.Endorser](#)¹⁰ class, and is inherited by [IndyLibraries.StewardAgent](#)¹¹ (figure 5.2 shows the class hierarchy).

The first example of transaction submission is given in the [addENDpointToNYM](#) method. The function code is represented in figure 5.4. The wrapper (from [Ledger](#) class) method [buildAttribRequest\(\)](#) creates

⁹<https://github.com/hyperledger/indy-sdk/blob/master/wrappers/java/src/main/java/org/hyperledger/indy/sdk/Wallet/Wallet.java> - wrapper Wallet class

¹⁰<https://github.com/caltr98/IndyWorking/blob/master/src/main/java/IndyLibraries/Endorser.java> - Endorser

¹¹<https://github.com/caltr98/IndyWorking/blob/master/src/main/java/IndyLibraries/StewardAgent.java> - StewardAgent class

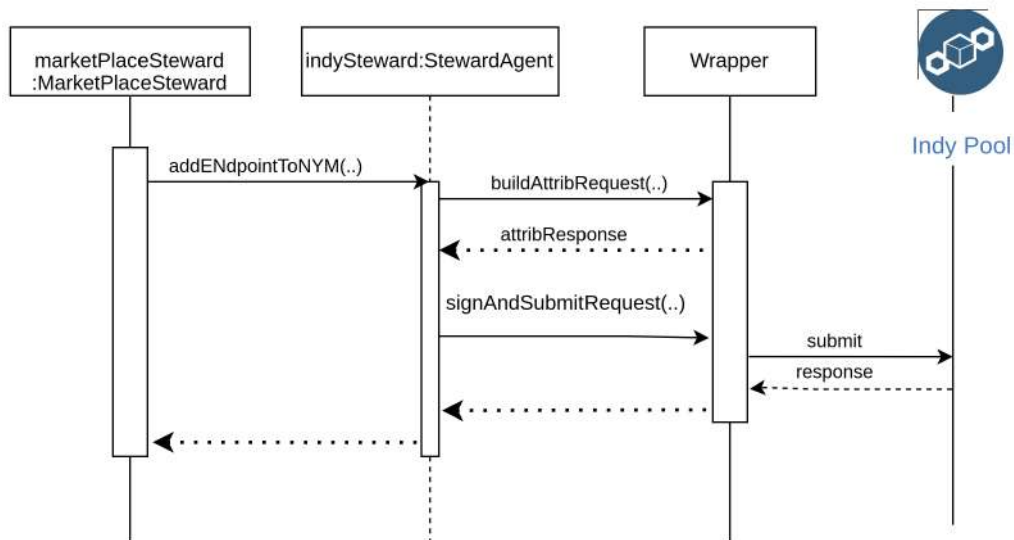


Figure 5.4: Sequence diagram representing the process of creating an ATTRIB transaction (for endpoint publication). The sequence diagrams include the performed invocation of methods and the communication with the ledger. In this figure, the Steward agent is decomposed into marketPlaceSteward and StewardAgent.

an **ATTRIB** request to create a transaction. A common pattern with transaction requests is:

1. Build Request;
2. call the method `signAndSubmitRequest()`. This method is from the wrapper class `Ledger`¹².

Make Steward ready for connections

After having placed its endpoint on the ledger, the Steward creates a Thread object that will run the server-side of the Steward, accepting connections from the store and delegating the requests to other threads. Steward is also capable of obtaining Validator Info files from the nodes. This will be seen in subsection 6.4.1.

¹²<https://github.com/hyperledger/indy-sdk/blob/master/wrappers/java/src/main/java/org/hyperledger/indy/sdk/ledger/Ledger.java> - wrapper Ledger class

5.3 Store setup as Issuer implementation

A Store agent is implemented by the [Store](#)¹³ class (as can be seen in figure 5.5). [MarketPlaceStoreTaskHandler](#) is a task that needs to be executed in a thread to handle customer requests. An object [Store2ShippingDeliverer](#) handles the communication with a **ShippingDeliverer**. [Store2Box](#) handles the communication with a **MailBox**. Finally, [StoreRequestsHandler](#) is a Task that listens to incoming requests and creates new [MarketPlaceStoreTaskHandler](#) tasks to handle them. The Store agent process runs as Main the class [STOREMAINTEST](#)¹⁴. The Store class has two constructors, one is used when the Store is created for the first time and does not have a DID on the ledger. The second constructor retrieves the already created Endorser DID from Wallet and uses it, without contacting the Steward. The first constructor will be used as a reference. The Store creates its own DID with a similar procedure to the one illustrated in section 5.2.2 A fragment of the Store constructor:

```
public Store(String poolName,String storeName,String WalletPassword){
    .....
    byte[] request=StoreIndy.writeMessage(StoreIndy.
        askForEndorserRoleRequest(true),store_to_stewardDID2DID);
    StoreIndy.sendD2DMessage(request,connectionTOSTeward.
        socket());
    //read response
    String response=new String((StoreIndy.waitForMessage(connectionTOSTeward.
        socket()),
        Charset.defaultCharset());
```

This code fragment illustrates the operations to ask for a [DID](#)¹⁵ with Endorser role (same process from figure 3.28). It has to be noted that the Store and Steward agents communicate with a DID to DID communication. The method [getEndPointFromLedger\(\)](#) parses the endpoint from another method internal to the Agent class called method [getATTRIBFromLedger\(\)](#) from the same class.

¹³<https://github.com/caltr98/IndyWorking/blob/master/src/main/java/Store.java>-Store class

¹⁴<https://github.com/caltr98/IndyWorking/blob/master/src/main/java/STOREMAINTEST.java> - STOREMAINTEST

¹⁵A Store DID on Sovrin <https://indyscan.io/tx/SOVRIN{ }BUILDERNET/domain/14705>

CHAPTER 5. AN IMPLEMENTATION WITH HYPERLEDGER INDY107

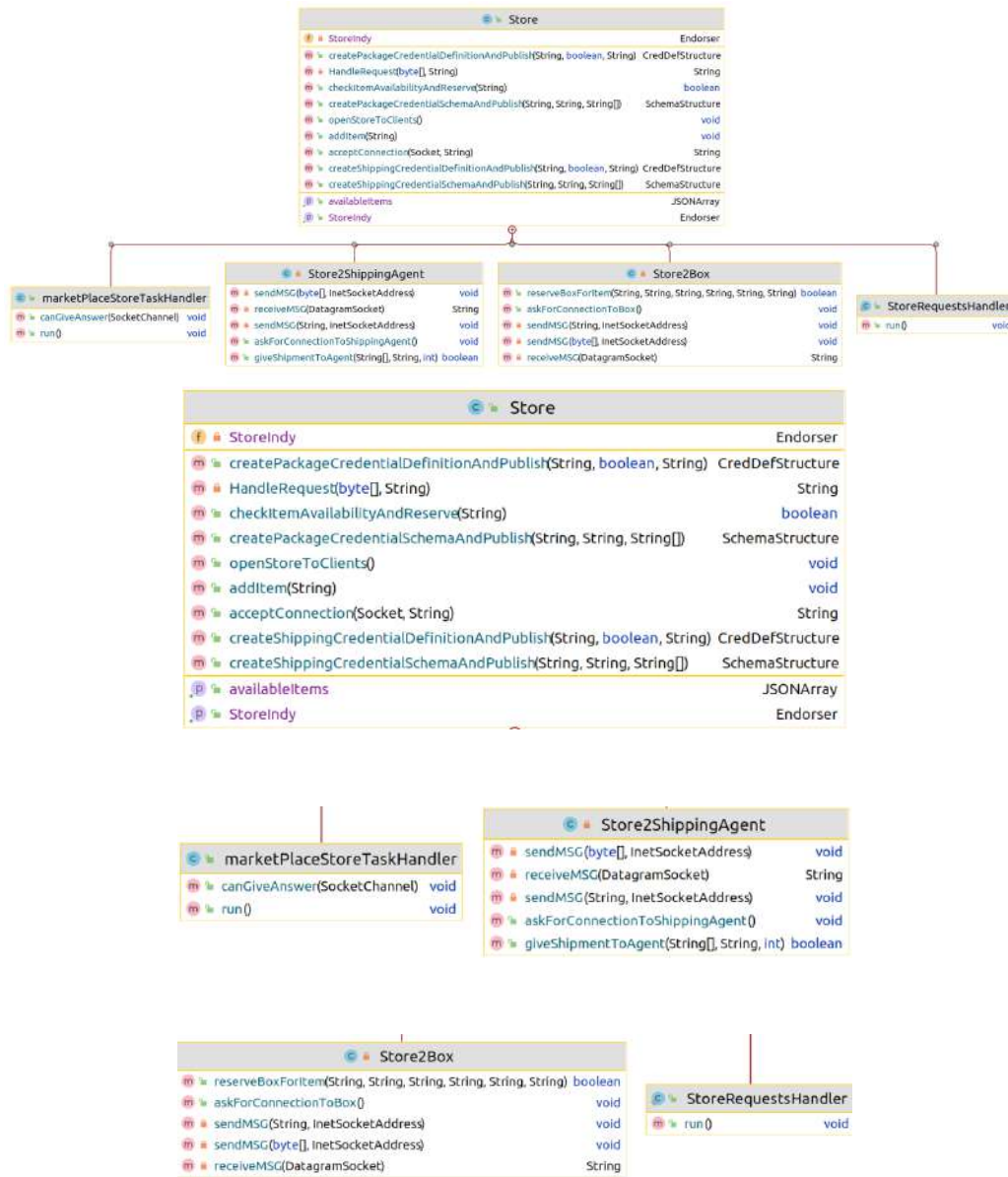


Figure 5.5: Store class diagram with its inner classes.

```

public String getATTRIBFromLedger(String destDID, String optRaw, String optHash, String
    optencryptedval) {
    .....
    String getAttribRequest = Ledger.buildGetAttribRequest(this.mainDID.didName, destDID,
        optRaw, optHash, optencryptedval).get();
    String s = Ledger.submitRequest(this.poolConnection, getAttribRequest).get();
}
  
```

The code fragment above shows another pattern of Indy used for reading requests: i) Build read request; 2) [submitRequest\(\)](#) Since the request in this case does not require a signature, it is possible to hide the submitter DID. To hide the submitter DID, the parameter `submitterDid`, in the build read request method must be null. In the above case the `mainDID.didName` provided is null. Then the "default Libindy DID" will then be used. [59].

5.3.1 createConnection - DID to DID setup

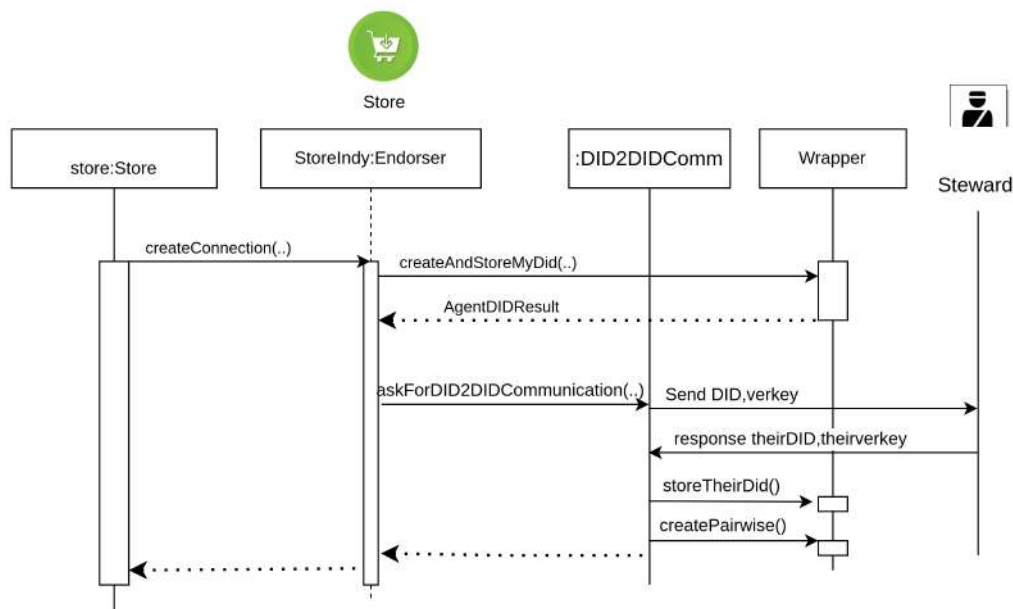


Figure 5.6: Sequence diagram representing DID communication setup from the store agent side.

The method [createConnection](#) from `IndyLibraries.Agent` is used to setup DID to DID communication with a TCP connection. DID creation is done without specifying the seed value. The new DID, alongside its verkey, is passed to the method [askForDID2DIDCommunication\(\)](#) from `IndyLibraries.DID2DIDComm`¹⁶. The methods `storeTheirDid` and `createPairwise` were already discussed in subsection 3.4.4 and are now

¹⁶<https://github.com/caltr98/IndyWorking/blob/master/src/main/java/IndyLibraries/DID2DIDComm.java> - DID2DIDComm class

used, thus implementing the DID to DID communication setup from figure 3.12.

The method `askForEndorserRoleRequest()` returns the result from method `create_nym_endorserRequest(..)` (from class `IndyJsonStringBuilder`). The Json structure is not defined by Indy, it was created to provide a format for the Endorser role request whose main components are:

```
{
  "request": "NYMENDORSER"
  "fields": {
    "did" = <did_value>
    "verkey" = <verkey_value>
    .....
  }
}
```

To send a DID to DID message, the method `writeDID2DIDMSG()` from `DID2DIDComm` is called. In the message is encrypted with the method `packMessage` (from wrapper class `Crypto`¹⁷), which corresponds to the `pack()` method of subsection 3.4.4). This message was sent as an array of bytes to the Steward. Afterward, the Steward reads the message using the `unpackMessage()` method from the wrapper. The whole process is illustrated in figure 3.13.

5.3.2 Assigning Endorser role - NYM transaction

The process of creating a NYM transaction is similar to the ATTRIB transaction process (from section 5.2). As can be seen in the following Json structure, the NYM requests specify the role and DID of the Store as dest.

```
{
  "identifier": "Th7MpTaRZVRYnPiabds81Y",
  "protocolVersion": 2,
  "operation": {
    "role": "101",
    "type": "1",
    "dest": "S9uZc5yYQ6ErjcBaegfmg",
    "verkey": "Ei9uLpHCVejbpeCgTT1Q8PaHYUyLFyzdtYdj67uKLeM"
  },
  "reqId": 1643060751386667292
}
```

¹⁷<https://github.com/hyperledger/indy-sdk/blob/master/wrappers/java/src/main/java/org/hyperledger/indy/sdk/crypto/Crypto.java> - wrapper `Crypto` class

5.4 Setup for orders - on ledger Anoncred entities

After being granted a [NYM](#) record, the Store agent has to write on the ledger its [endpoint](#)¹⁸. The schemas [ShipmentParcelSchema](#)¹⁹ and [ShippingSchema](#)²⁰, used to specify the attributes of a verifiable credential, needs not to be created and stored on the Indy ledger.

5.4.1 Publish a Schema on the ledger

The method calls can be seen in figure 5.7. The Credential Schemas needed for the system are public on the ledger (in a Schema transaction as in figure 3.17). The Store then creates the Credential Definition that stores its DID public key used to sign the credentials.

5.4.2 Publish a Credential Definition on the ledger

As for the Credentials Schemas, the [ShipmentParcel Credential Definition](#)²¹ and [CustomerShipping Credential Definition](#)²² derived Credential Definition are created with the same process (that can be seen in figure 5.8). With this last step, the Credential Definition discussed in 3.5.3 is public on the ledger.

5.5 Give a package to a Shipping Deliverer - Issuing a credential

As said in subsection 4.2.5, a ShipmentParcel credential is issued to a Shipping Deliverer. Upon a delivery request to the Shipping Deliverer,

¹⁸Store endpoint on Sovrin https://indyscan.io/tx/SOVRIN_BUILDERNET/domain/15203

¹⁹ShipmentParcelSchema on Sovrin https://indyscan.io/tx/SOVRIN_BUILDERNET/domain/15168

²⁰ShippingSchema on Sovrin https://indyscan.io/tx/SOVRIN_BUILDERNET/domain/15199

²¹ShipmentParcel Credential Definition on Sovrin https://indyscan.io/tx/SOVRIN_BUILDERNET/domain/15197

²²CustomerShipping Credential Definition on Sovrin https://indyscan.io/tx/SOVRIN_BUILDERNET/domain/15194

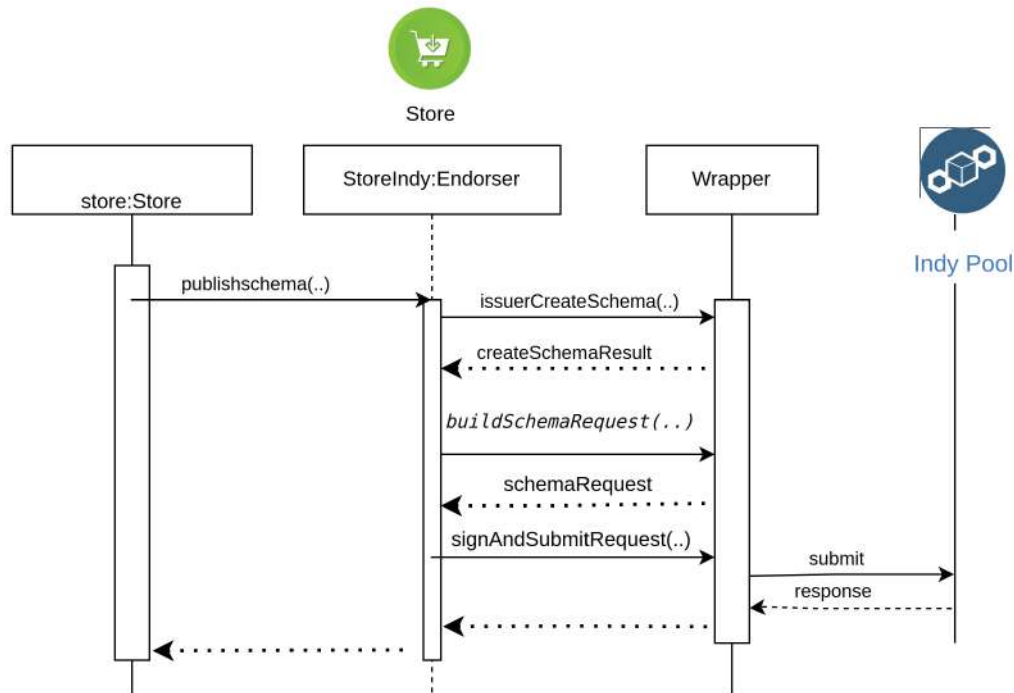


Figure 5.7: Sequence diagram representing the method calls to perform a SCHEMA transaction.

a Credential Offer is created (figure 5.9) and sent to its agent. Subsequently, the credential request was created by the Shipping Deliver agent. The necessary MasterSecret to bundle in the request (as said in subsection 3.5.6), if not created before, is created alongside the credential. The methods to create a Credential Request, started by the method `DeliveryCredRequest(..)` of `Shipping Deliverer`²³, can be seen in figure 5.10.

With the method `returnIssuerCreateCredentialNonRevocable(..)` of `IndyLibraries.Endorser`, the Store issues the credential (in figure 5.11). The created credential in this implementation have the attributes:

```

shipmentid = "42"
shipmentnonce = "4316897532576543..."
shipmentdeclaredvalue="54_euros"
shipmentweight = "4kg"
shipmentavailabilitytime = "2_days"

```

²³<https://github.com/caltr98/IndyWorking/blob/master/src/main/java/ShippingDeliverer.java> - Shipping Deliverer class

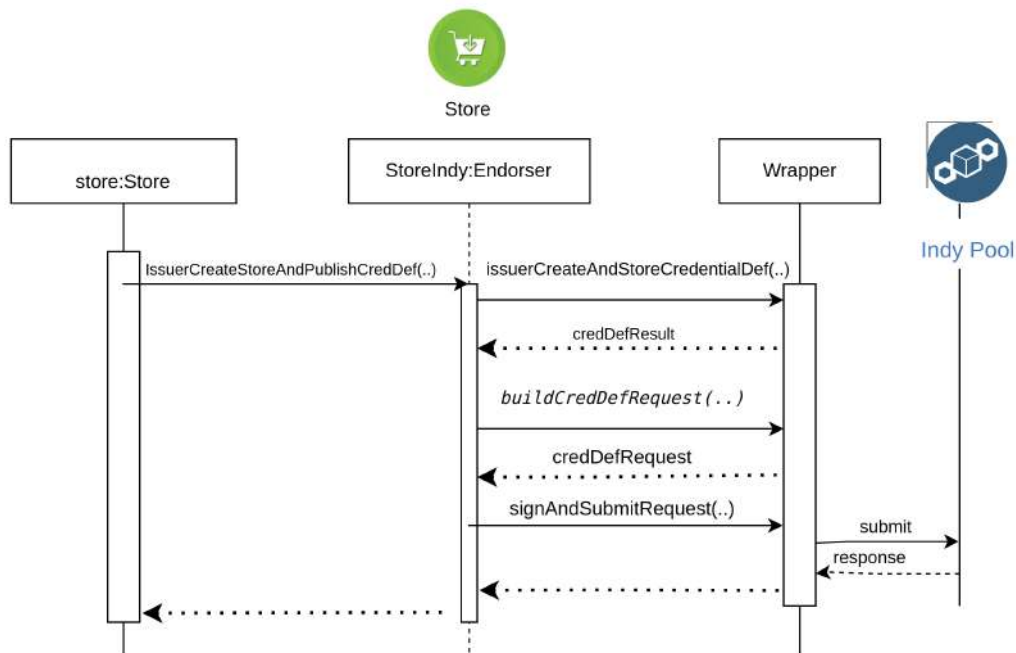


Figure 5.8: Sequence diagram representing the method calls to perform a CLAIM_DEF transaction.

The credential is created in the following method call of [giveShipmentToAgent](#), alongside the Mail Box endpoint:

```

store2shippingAgent.giveShipmentToAgent( new String[]{ String.valueOf(currentshipmentid),
(int)((Math.random()*79 )+1 )+"kg",
(int)((Math.random()*999 )+1 )+"euros",
nonceForShipment, String.valueOf(((int) (Math.random()*71) +1))},
useBOX.boxAddress.getAddress().getHostAddress(),useBOX.boxAddress.getPort());

```

The credential is then sent to the Shipping Deliverer, which stores it. To store the credential of the Shipping deliverer, the method used is [DeliveryCred](#). Below is its method call:

```

public String DeliveryCred(String credential,String LockerBoxAddress, int LockerBoxPort){
    String storeCredential=
        IndyShippingDeliverer.storeCredentialInWallet(null,DeliveryCredentialOffer.credDef.
            credDefId,DeliveryCredentialRequest.credReqMetadataJson,
            credential,DeliveryCredentialOffer.credDef.credDefJson,null);
    ....
}

```

Where [storeCredentialInWallet\(\)](#) calls the method [proverStoreCredential\(\)](#) from the wrapper (Anoncreds class).

Among other Anoncred entities, the metadata from the Credential Re-

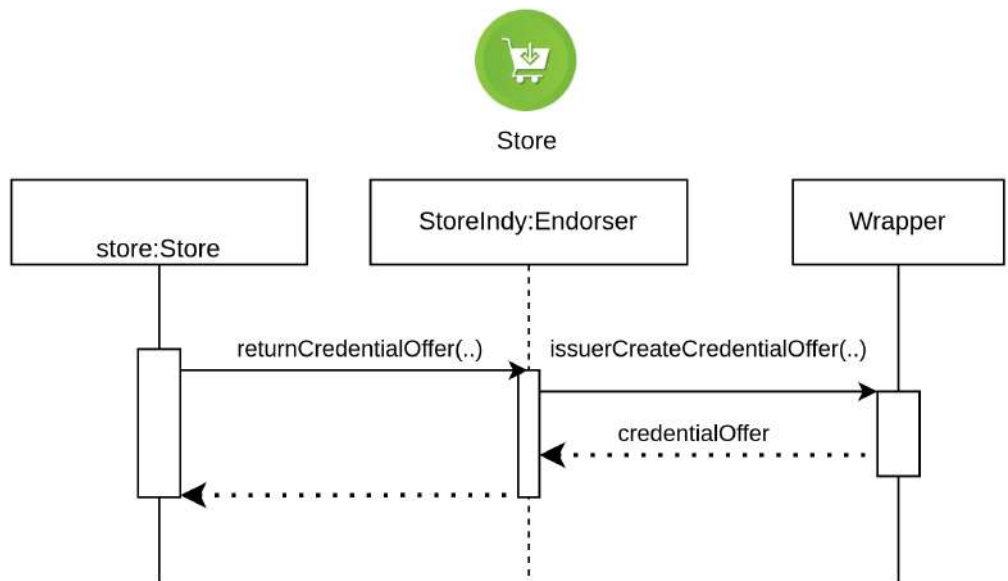


Figure 5.9: Sequence diagram representing method calls to create a Credential Offer.

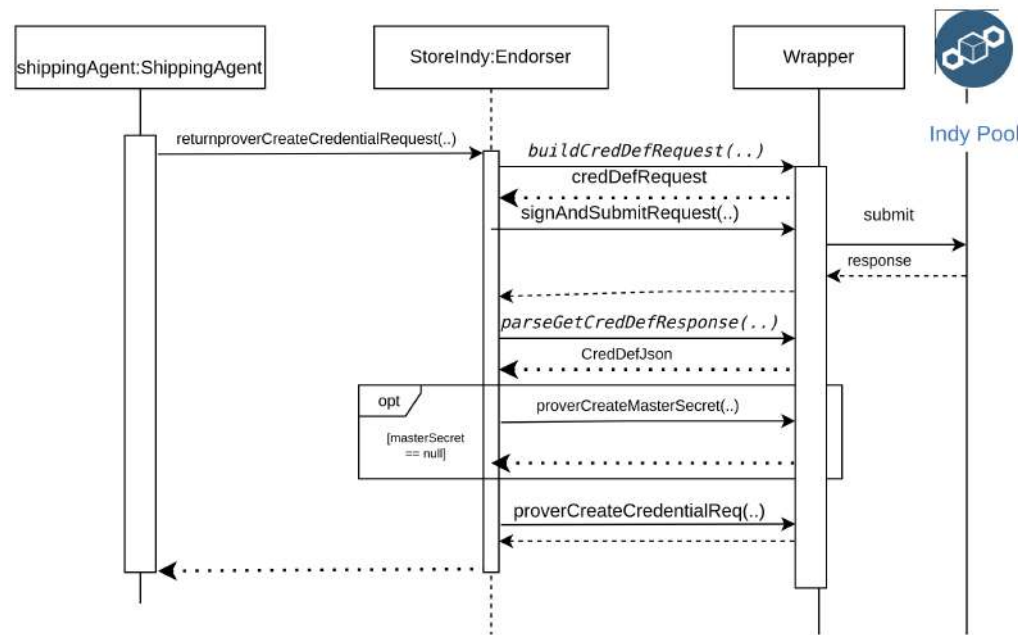


Figure 5.10: Sequence diagram representing method calls to create a Credential Request.

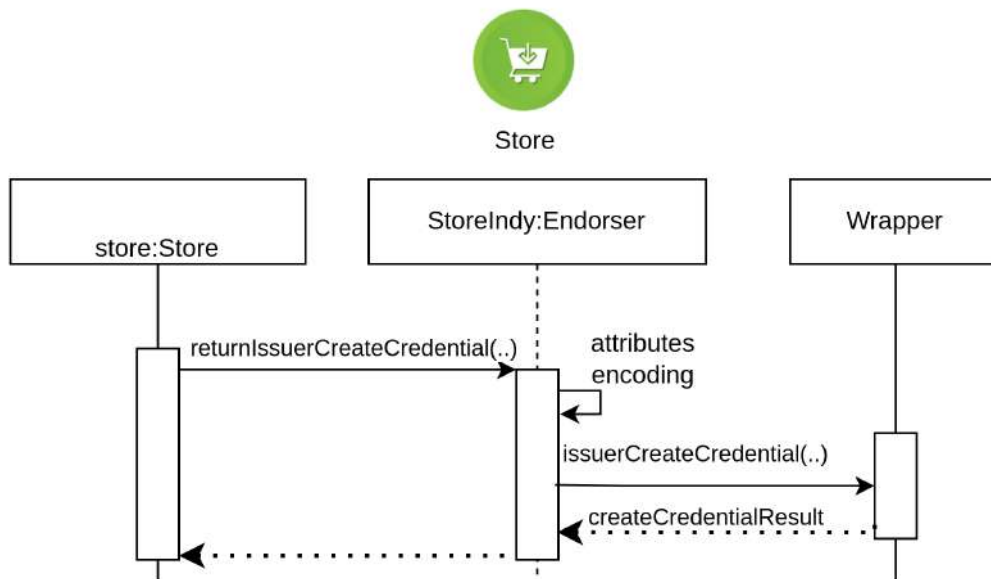


Figure 5.11: Sequence diagram representing the method calls to create a Credential.

quest is stored. The metadata contains the field `master_secret_` which identifies in the Wallet the `MasterSecret` used in the Credential Request. The process to issue a credential to the Customer is similar to the Shipping Deliverer one. The only difference lies in the attributes required by the schema, which are the `ShippingId` and `ShippingNonce` and a description of the items (e.g their names). Therefore, the process of creating a `CustomerShipping` credential description will be omitted. Instead, we will focus on the Customer as a prover, i.e. the moment the customer creates proof to present to the Mail Box to open it. Similarly, the same actions performed by the Shipping Deliverer will be omitted. However, what happens after a Shipping Deliverer presents one valid proof is described in the next section.

5.6 Reading a proof attribute values - Mail-Box after receiving proof from Shipping Deliverer

The MailBox agent has to save the timeframe to keep the items once they arrive. As said in subsection 4.2.5, the timeframe is a value contained in the ShipmentParcelCredential, and part of the attributes requested by the MailBox. The MailBox agent object class diagram ([LockerBoxAgent](#)²⁴) can be seen in figure 5.12. The method `insertItemInBox()` is used by the Shipping Deliverer to present a valid proof to open the MailBox.

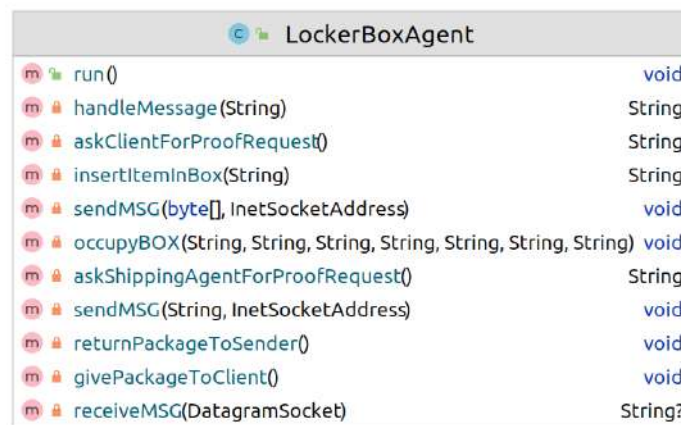


Figure 5.12: LockerBoxAgent class diagram. This method being run waits for requests and handles the requests received in a single thread.

As can be seen in figure 5.13, the MailBox agent reads the `shipment-availabilitytime` value from the proof and starts a timer. The purchased items will be available to the customer for collection until the shipping availability time elapses. Note that in this case the `shipment-availabilitytime` time is expressed within hours. The `insertItemInBox` method reads the proof fields as a Json structure. The given `shipment-availabilitytime` time is summed up to the current moment timestamp to create a timestamp: `timestampreturnitem`. The `timestampreturnitem` represents the deadline date for item pickup.

²⁴<https://github.com/caltr98/IndyWorking/blob/master/src/main/java/LockerBoxAgent.java>
- LockerBoxAgent class

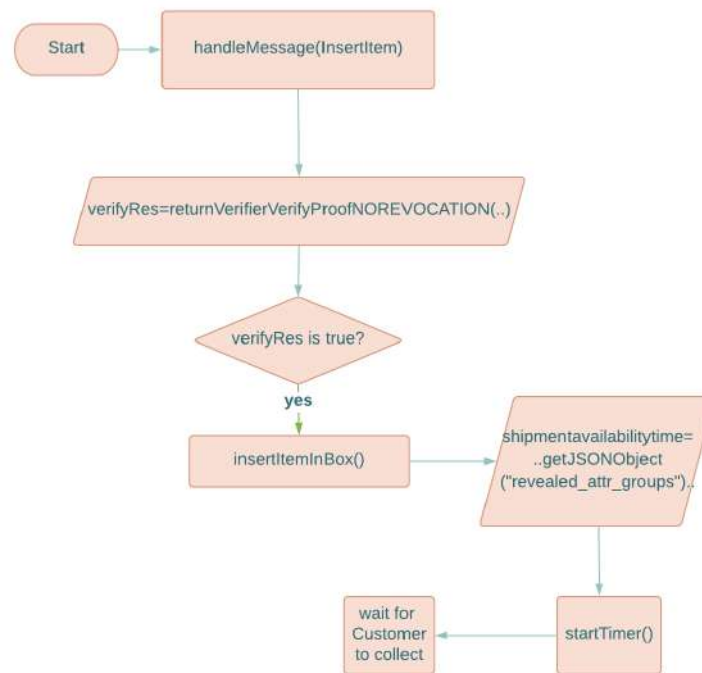


Figure 5.13: Flowchart diagram of MailBox arrival of an order. After proof verification, the timer starts.

5.6.1 Communication with MailBox - Customer to MailBox communication

The CustomerMarket object (representing the Customer agent) uses objects of inner class Customer2Box (class diagram in figure 5.14). This class contains methods to contact a specific Box and data regarding the Customer to Box communication. The MailBox software agent is supposed to be light weighted and communication with it involves fewer messages exchanged than the Customer/Store agent. As TCP connections have extra overhead with the TCP connection handshake and to maintain the ordering of messages, UDP is preferred in this implementation. As such, DatagramSockets are used for communication and the methods for DID to DID communication are slightly different than in section 5.2.3. The method [*createDID2DIDCommunication\(\)*](#) from the class `IndyLibraries.Agent` sends the handshake message, created by [*setupDID2DIDCommunicationAsk*](#), in a `DatagramPacket`. It is useful to create only the message independently of the channel of communication. The invoked calls can be seen in figure 5.15.

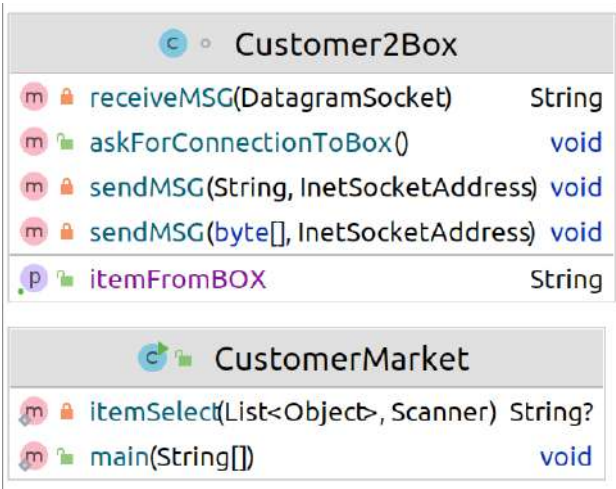


Figure 5.14: Class diagram of CustomerMarket (Customer agent main class) and Customer2Box class(for handling communication with Mail Box)

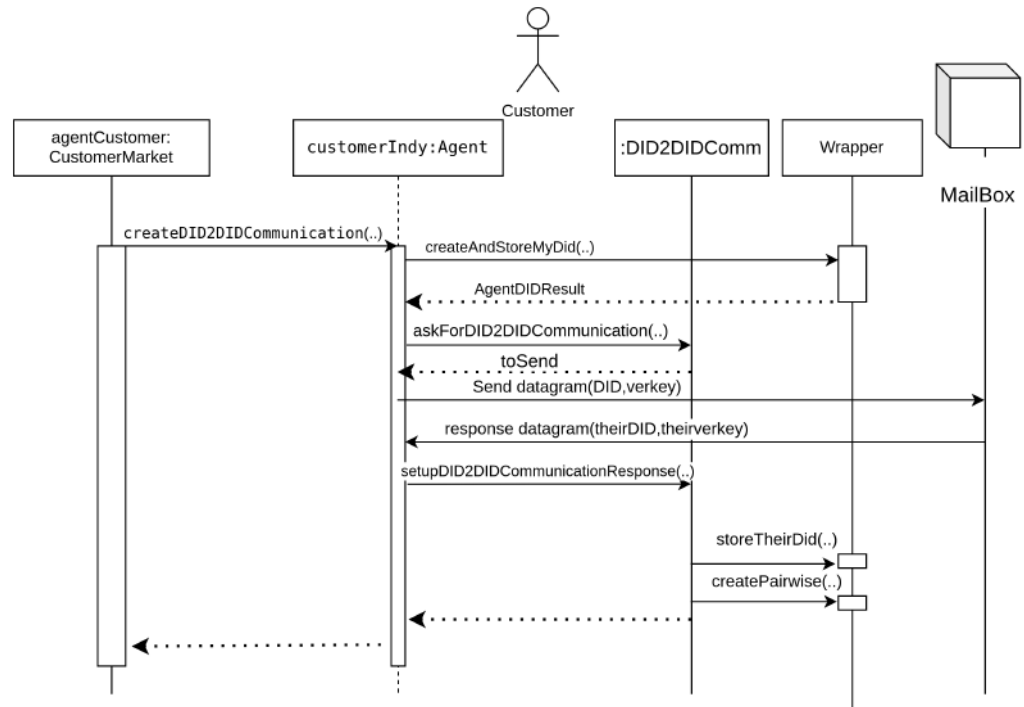


Figure 5.15: Sequence diagram representing DID to DID communication setup with a stateless protocol for communication.

5.6.2 Opening the MailBox - Proof request creation

As said in subsection 3.5.8 proof request creation does not have an associated method offered by Indy-SDK. To create a proof request, the method `generateAttrInfoForProofRequest` from `IndyJsonStringBuilder` is used to create an `attr_info` structure (introduced in section 3.5.8). The method allows for specifying all the possible restrictions for a given attribute or a set of attributes. In this case, there are two equality restrictions to impose on attributes `ShippingId` and `ShippingNonce`, as said in subsection 4.2.5.

An example of the `attr_info` generated by the method is below:

```
{
  "names": [
    "shippingnonce",
    "shippingid"
  ],
  "restrictions": {
    "issuer_did": "4F19BBTtpFRSwtFJQQ2B2t",
    "cred_def_id": "4F19BBTtpFRSwtFJQQ2B2t:3:CL:17:tag",
    "attr::shippingnonce::value": "1191773072",
    "attr::shippingid::value": "1"
  }
}
```

The given attribute info is coupled with a Nonce generated by the wrapper. The complete proof request is created with the method `createProofRequest()` from `IndyJsonStringBuilder`, which groups in a `Json` structure all the `attr_info` structures. An example of a proof request produced by this method is:

```
{
  "ver": "1.0",
  "name": "GetItemProof1643153583626",
  "requested_attributes": { "attr0_referent": {
    "names": [
      "shippingnonce",
      "shippingid"
    ],
    "restrictions": {
      "issuer_did": "4F19BBTtpFRSwtFJQQ2B2t",
      "cred_def_id": "4F19BBTtpFRSwtFJQQ2B2t:3:CL:17:tag",
      "attr::shippingnonce::value": "1191773072",
      "attr::shippingid::value": "1"
    }
  } },
  "nonce": "483392572069903751230186",
  "version": "1.0",
  "requested_predicates": {}
}
```

5.7 Fetching attribute for proof - Customer search for ShippingCredential

After the reception of a proof request, the operations in 3.6.3 are executed to create a proof. The method calls used in the process to create a proof can be seen in figure 5.16.

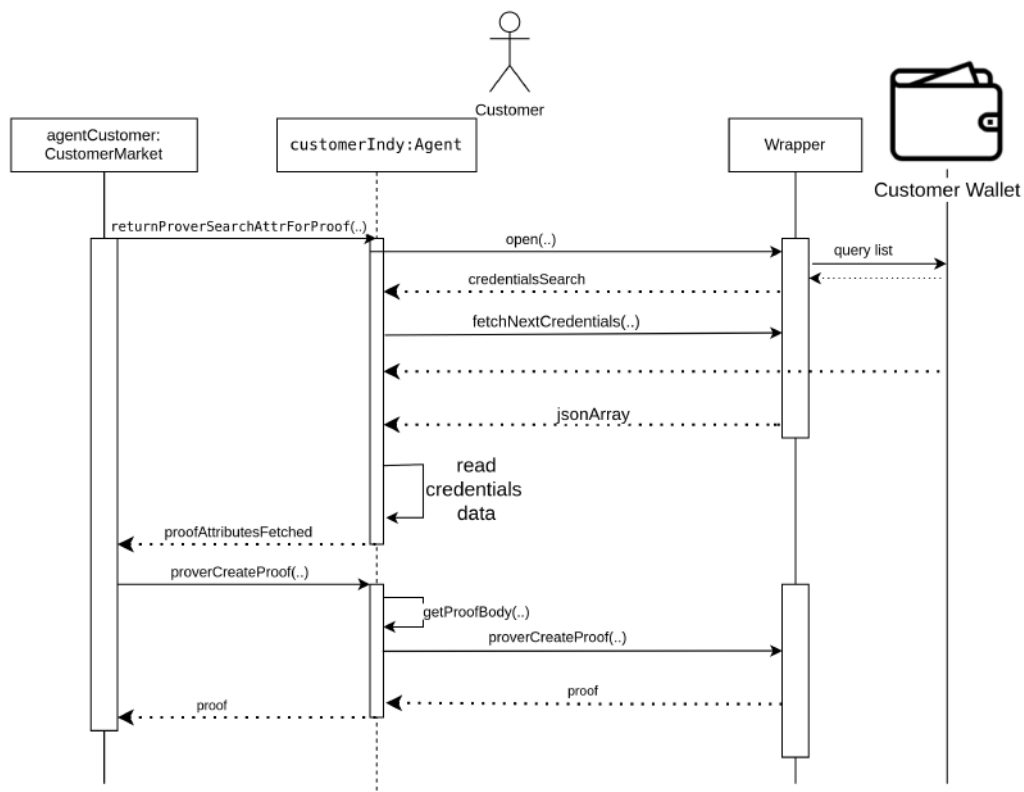


Figure 5.16: Sequence diagram representing the method calls to create a proof. Each method will be discussed in this subsection

The method to search for the request attributes in the Wallet is [*returnProverSearchAttrForProof\(\)*](#) and it is contained in `IndyLibraries.Agent`. The proof search is illustrated in subsection 3.6.3.

Inside the method *returnProverSearchAttrForProof()* the method *open* is called to get the object *CredentialsSearch*, which is used to fetch the credentials in batch. For every attribute requested in the proof request, the credentials compatible with the specified restrictions are returned.

```
credentialsSearch = CredentialsSearchForProofReq.open(this.mainWallet,
proofRequestJson, null).get();
```

CHAPTER 5. AN IMPLEMENTATION WITH HYPERLEDGER INDY120

The credentials can be obtained from the Wallet as a Json array.

```
jsonArray = new JSONArray(credentialsSearch.fetchNextCredentials(attrRef, 100).get());
```

The method `fetchNextCredentials` from the wrapper returns a string that represents as a Json array the credentials queried. In our specific case, only one credential exists per proof request. The restrictions equalities are defined on `shippingNonce`, `shippingId`, and the Store identifier. As said in the section 4.2.5, the pair `shippingNonce` and `shippingId` is unique for a Store. One of the fields obtained from the `jsonArray` is the credential identifier in the Wallet. To create a proof, the credentials identifiers associated with the attributes of the proof are needed.

```
credentialIdForAttribute = jsonArray.getJSONObject(0).getJSONObject("cred_info").getString("referent");
```

Finally, when `credentialsSearch` is no longer needed, it has to be closed:

```
credentialsSearch.close();
```

5.8 Proof - creating a proof to open the Mail-Box

Before creating the proof, a particular Json structure must be composed. The `getProofBody()` from `IndyJsonStringBuilder` composes the attributes with their credential Wallet identifier.

The proof creation is performed with the method `proverCreateProof` from the `Anoncreds` class wrapper.

5.8.1 MailBox verification for opening - Verify proof

The proof verification operations are shown in figure 4.9. As said in subsection 4.2.6, the proof can be valid but the item availability time could have elapsed. There are two defined methods in `Agent.IndyLibraries` to verify the validity of a proof are two and their signature is:

```
returnVerifierVerifyProof(String proofRequestJson, String proofJson, String schemas,String  
    credentialDefs, String revocRegDefs,String revocRegs)  
  
returnVerifierVerifyProofNOREVOCATION(String proofRequestJson, String proofJson, String  
    schemas,String credentialDefs)  
{
```

The first method is designed to verify proofs that are related to credentials that are revocable. In the `CustomerShippingCredential` case, the

CHAPTER 5. AN IMPLEMENTATION WITH HYPERLEDGER INDY121

credential is not revocable, meaning the first method is not the most suitable for it. As such the `returnVerifierVerifyProofNOREVOCATION` method is used to verify proof by the MailBox agent.

A fragment of code from the `returnVerifierVerifyProofNOREVOCATION` method code showing the invocation of the wrapper method `verifierVerifyProof`:

```
public Boolean returnVerifierVerifyProofNOREVOCATION(String proofRequestJson, String
proofJson, String schemas, String credentialDefs) {
    Boolean valid = false;
    ...
    try {
        valid = Anoncreds.verifierVerifyProof(proofRequestJson, proofJson, schemas,
        credentialDefs, revocRegDefs, revocRegs).get();
    }
}
```

After the verification with `verifierVerifyProof` from the wrapper (Anoncreds class), the boolean "valid" is returned, which represents the proof validity if true, otherwise the proof is not valid if false.

The proof verification is done in the method `handleMessage()` of LockerBoxAgent. The figure 5.17 summarizes the code run by the MailBox agent when an open request arrives.

After verifying the proof, if it is valid, the Mail Box agent checks for the time of proof presentation. If the proof presentation is within the time frame of availability, the Mail Box agent sends a Success message to the Customer. Else, if the proof is valid but the Customer is late, a warning is sent. The implementation is shown in the following fragment of code, thus terminating the overview of the Shipping Verification System implementation is complete.

```
boolean verifyRes=
IndyLockerBox.returnVerifierVerifyProofNOREVOCATION(this.lastProofRequestToCustomer,
proofReceived,schemasReceived,creddefsReceived);
if(verifyRes) {
    //CHECK openingTime is within the allowed time limits
    byte[] msg=null;
    long currentOpeningTime = System.currentTimeMillis();
    String openBoxResult="validproof";
    if(currentOpeningTime>=this.timestampcanopen && currentOpeningTime<=this.
timestampreturnitem){
        openBoxResult="Success";
    }
    else{
        openBoxResult="WrongTime";
    }
}

if(openBoxResult.equals("Success")){
    msg = IndyLockerBox.writeMessage("Success", addresstodid.get(currentTalker));
    this.isOccupied = false ;
}
```

CHAPTER 5. AN IMPLEMENTATION WITH HYPERLEDGER INDY122

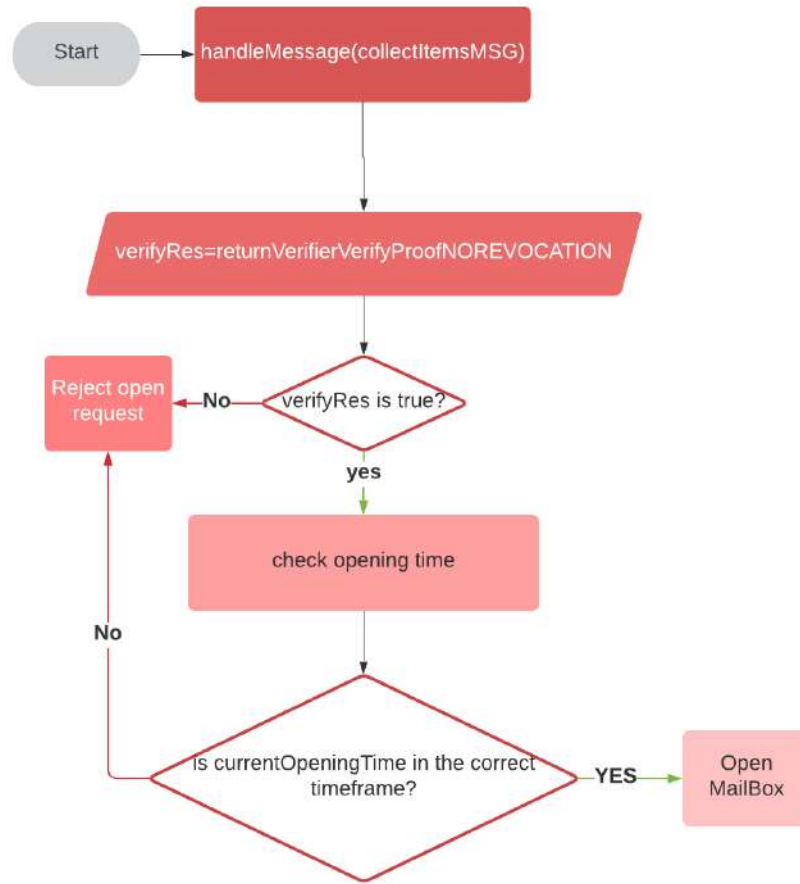


Figure 5.17: Flowchart diagram of MailBox checking proof and time for opening.

```
    sendMSG(msg, currentTalker);  
    return "Success_proof_is_:\n"+proofReceived;  
  }  
  else{  
    msg = IndyLockerBox.writeMessage("WrongTime", addresstodid.get(currentTalker));  
    sendMSG(msg, currentTalker);  
    return "Failure_proof_is_valid_but_Customer_arrived_too_late";  
  }  
}
```

Chapter 6

Experimental Evaluations

This chapter illustrates security evaluations on HyperLedger Indy and the Shipping Verification System, described in the previous chapter. Furthermore, it includes performance analysis of the Sovrin BuilderNet and a local pool of nodes. Also, agent-side operations performance and Anoncreds entities dimensions are illustrated. Finally, the metrics collected by a node of the Shipping Verification System are shown.

6.1 Threat model tests

This section presents a threat model to show how the system resists malicious users.

6.1.1 Threat model Scenario: The Shipping Agent receives critical information to impersonate the Customer - credential theft

This subsection illustrates the unlikely scenario, a credential received by the wrong holder. This case is correlated with the third entry of Table 4.1. All the test agents classes can be found on [IndyWorking](https://github.com/caltr98/IndyWorking/tree/master/src/main/java) repository.¹

¹Agents classes – <https://github.com/caltr98/IndyWorking/tree/master/src/main/java>

Testing methodology - Assumptions

To test this case changes were made to agents: Store, ShippingDeliverer, and CustomerMarket agents. The faulty Customer is identified as [sloppyCustomer](#). The sloppyCustomer sends, alongside the Credential Request for the ShippingCredential, the Credential Request metadata (mentioned in section 5.5). Credential Request metadata is a field that should not be disclosed to the issuer. It links the credentials being requested to the MasterSecret in the Wallet. The faulty (or malevolent) Store is identified as sloppyStore. The malicious Shipping Deliver is represented by the agent of the class scammerShippingDeliver. The malicious agent tries to store the credential as it was created for it, it could allow the malicious agent to create a valid proof with the credential. The sloppyStore sends the scammerShippingDeliver the credential and Credential Request metadata that sloppyCustomer sent to it.

Contextualize results: What does this unlikely scenario demonstrate?

This experiment shows how the MasterSecret is the key factor to credentials security. A holder without a MasterSecret of a credential cannot produce a valid proof.

Scenario

The scenario is represented in figure 6.1. To create a proof, a holder **must** have the credentials of the attributes stored in his Wallet. For storing credentials, the method *storeCredentialInWallet()* from *IndyLibraries.Agent* is used.

The scammerShippingDeliverer having all the data needed to store the credential in his Wallet tries to store it using the wrapper method. The scammerShippingDeliverer Wallet does not contain the MasterSecret, which is in a blinded form inside the Credential Request.

The method *storeCredentialInWallet()* tries to find the MasterSecret identified by *master_secret_name*. As said in section 5.5, the MasterSecret is contained in the Credential Request metadata. Since the scammerShippingDeliverer does have the MasterSecret identified, the method call fails, in Java throwing an exception: **WalletItemNotFoundException: No value with the specified key exists in the**

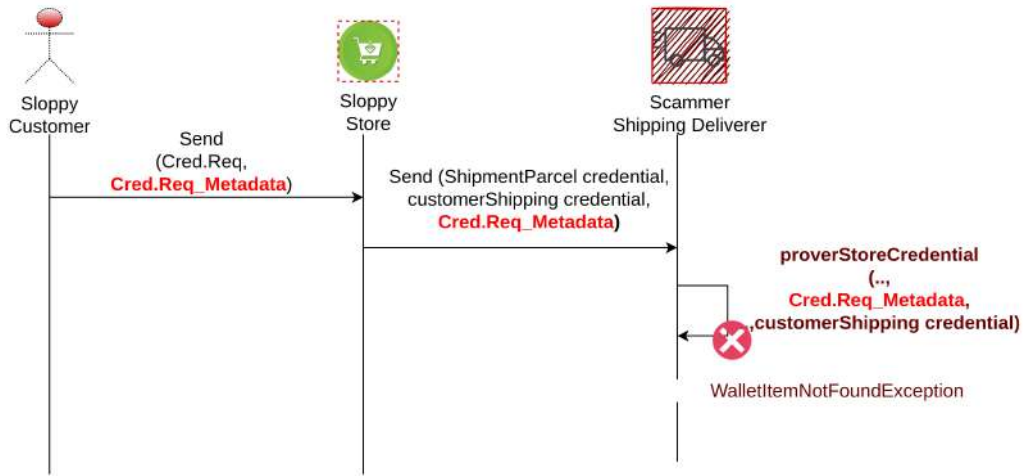


Figure 6.1: Sequence diagram of the interaction between sloppyCustomer, sloppyStore and scammerShippingDeliverer. In bold red is the action that compromises the security of the order.

Wallet from which it was requested. If the scammerShippingDeliverer tries to craft a MasterSecret with the same identifier, another exception is thrown. 6.2 The UrsaCryptoError exception containing this message can be seen in figure: **UrsaCryptoError: Invalid Signature correctness proof $q \neq q'$** . This message indicates that checks on the credential are done when the holder stores it. In this case, the libindy internal commands recognize that the MasterSecret does not correspond to the one stored in the credential.

Conclusions

It is not possible to store the credential of another agent. This protects the holder (the customer in this case) from having their credentials shared with other subjects. It was uncovered that cryptography operations are done on the credential before being stored (using UrsaCrypto).

6.1.2 Threat model Scenario: Proof falsification - Customer modifies a proof attributes value

This subsection illustrates the alteration of a valid proof, i.e. a holder tries to modify a valid proof to create another valid proof. This case is loosely correlated with the third entry of Table 4.1.

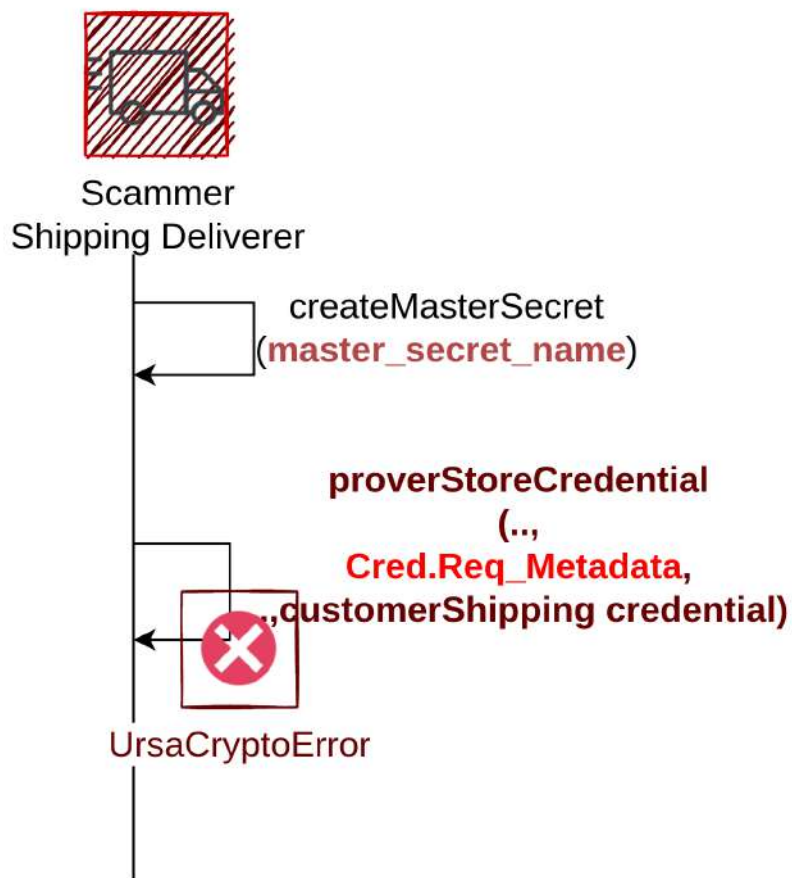


Figure 6.2: Sequence diagram of the interaction between `scammerShippingDeliverer` trying to replicate a `MasterSecret`.

Testing methodology - Assumptions

Agent `scammerCustomer` was instantiated. and does legitimately order an item. Once the item is collected, it stores the valid proof used to retrieve it. This case requires the assumption that the `scammerCustomer` can retrieve the `ShippingId` and `ShippingNonce` of a subsequent delivery to the same Mail Box.

Contextualize results: Does Indy recognize a rigged proof?

This experiment shows that modifying a valid proof does not produce another valid proof.

Scenario

The scenario is represented in figure 6.3. The scammerCustomer receives the ShippingId and ShippingNonce of the order currently contained in the Mail Box from an external entity. The external entity providing the data is defined as a Sensible Data Provider. For testing purposes, the Sensible Data Provider is the programmer inserting by terminal the aforementioned values (retrieved from the Store output on the terminal) in the scammerCustomer process. Generating proof with

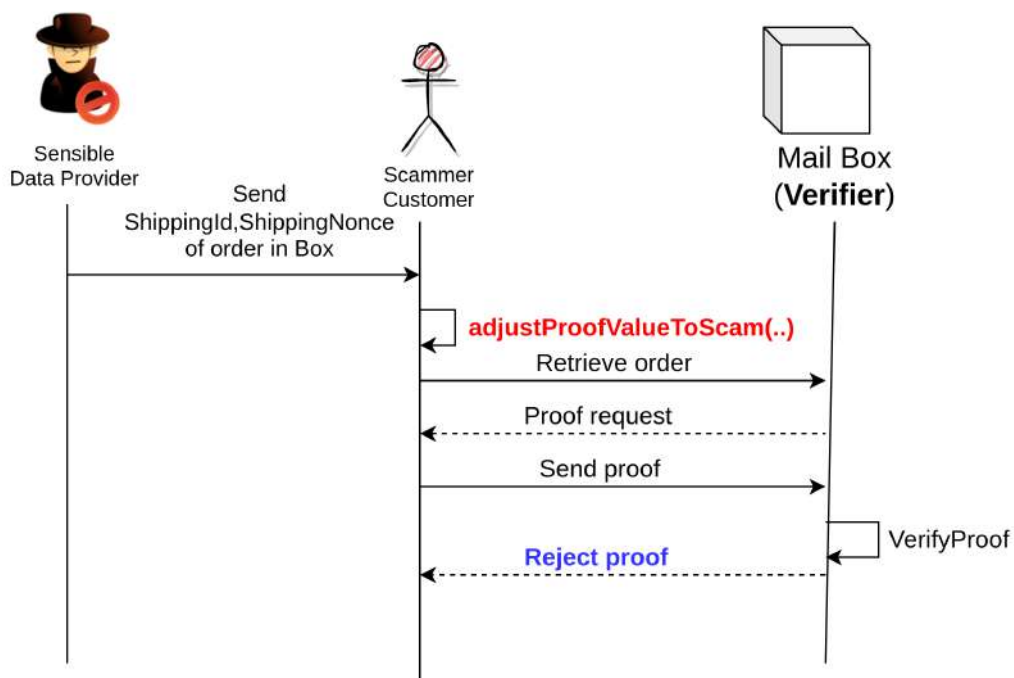


Figure 6.3: Sequence diagram of a Customer who tries to modify his proof with data received from an external subject. The Sensible Data Provider represents an entity that stole the ShippingId and ShippingNonce from the Store.

the requested ShippingNonce and ShippingId is not possible. Therefore, scammerCustomers can try to modify their relative fields in the old proof to match the new values. Upon verification, the proof results are false. The wrapper returns false and the libindy logger prints in the command line the following message:

```

Logger:gen 31, 2022 7:24:56 AM org.hyperledger.indy.sdk.LibIndy$Logger$1 callback
INFO: /home/indy/.cargo/registry/src/github.com-1ecc6299db9ec823/ursa-0.3.2/src/cl/
verifier.rs:313 | Verifier verify proof -> done
  
```

The method `verifierVerifyProof()` returns false. There are no exceptions thrown, the structure of the false proof corresponded to a valid proof structure, but not its contents.

Conclusions

Agents cannot craft a proof if they do not own valid credentials supporting it.

6.2 Networks tests

This section presents the tests performed to evaluate the response time of requests on the **Sovrin network** and the computational time of requests on a **local testnet** deployed through Docker.

6.2.1 Sovrin BuilderNet test - performance on unique transactions submission

This subsection presents the measurements on the Sovrin BuilderNet, deployed in 2019 and destined for testing purposes.

Testing methodology

The goal of this section is to uncover the responsiveness of the BuilderNet in four cases:

1. Rejected SCHEMA transaction: creating a SCHEMA transaction that does not respect the correct format. It has to be noted that the transaction format is loosely checked by the client-side libraries, before submission (in our case the wrapper). As such, a minor modification to a valid transaction was performed. The SCHEMA transaction has a field version (as can be seen in figure 3.17). An incorrect SCHEMA transaction request is created using a version number in the form XX (e.g 11). The pool of nodes only accepts numbers of the format XX.X or X.XX. The list of attributes is {"lot", "of", "attributes", "abcdefghijklmnopqrstuvwxyABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789", "1"}.

2. Accepted SCHEMA transaction: creating a valid SCHEMA transaction in respect of the format. The list of attributes is unchanged.
3. Rejected CLAIM_DEF transaction, For the creation of a rejected CLAIM_DEF transaction request, it is used the Credential Schema from the previous case. However, to render it invalid, a sequence number different than the Credential Schema sequence number on the ledger is provided in the request creation. This allows the creation of a Credential Request but with field values incorrect.
4. Accepted CLAIM_DEF, in this case, the SCHEMA is the one created in case 2 too. The Credential Definition defined by the transaction is also revocable (as the one from the Rejected CLAIM_DEF case). It means that the CLAIM_DEF will have two key fields instead of one, primary and revocation (as said in subsection 3.5.3,. As can be seen in figure 3.19, a CLAIM_DEF transaction contains the sequence number of its corresponding SCHEMA transaction. The Credential Schema used is the same as in the previous case, but CLAIM_DEF transactions specify a different TAG field (in figure 3.20) for every transaction. The TAG field is different for every transaction, making it a Credential Definition creation and not an update.

Each case was tested by repeating the request build and transaction submission 20 times in a row, which also allowed measurement of how the BuilderNet handles a rapid series of **different** requests. Two iterations of the test are used as a reference, each on two different days and at two different hours of the day. Each iteration includes 20 transactions submissions. For both of the iterations, the average value of the transactions submission and response time are presented. The results were calculated using the difference between the two calls of `System.currentTimeMillis()`.

This test is indicative of the performance of Indy with a ledger of well over 1500 transactions on domain ledger, over 90 transactions on pool ledger, and more than 60 transactions on config ledger for a total of more than 1950 transactions, which are all aggregated as copies in the audit ledger. The redundancy in the audit ledger takes the stored transactions total to twice as much as 1950 previously accounted. This means that the node stores well over 3900 transactions. The measurements with the BuilderNet can give an insight on pool performance a pool with **17**

nodes[60][BuilderNet page panel] as in January of 2022 (all validators). It was possible to measure the time from transaction signature to pool response and the number of bytes exchanged between the local Indy agent and a Sovrin node. Using Wireshark and its name resolution, the IP of a node related to the BuilderNet was distinguishable from other communications IPs.

In this case, the node conversation measured was with node: `sovrin1.internet-sicherheit.de`.

The summary is shown in Table 6.1 computed from the data in appendix A.6.

Operation Iteration 1	Avarage (ms)	Variance (ms)	[MIN, MAX] (ms)	Bytes Exchanged Agent(A) to Sovrin(S) Sovrin(S) to Agent(A)	Operation Iteration 2	Avarage (ms)	Variance (ms)	[MIN, MAX] (ms)	Bytes Exchanged Agent(A) to Sovrin(S) Sovrin(S) to Agent(A)
Sovrin BuilderNet Rejected SCHEMA request	175,95	25186,0475	[91, 506]	A to S 18k S to A 10k	Sovrin BuilderNet Rejected SCHEMA request	59,68	26511,62	[90, 632]	A to S 18k S to A 10k
Sovrin BuilderNet Accepted SCHEMA request	3402,45	1710615,21	[2989, 8592]	A to S 73k S to A 83k	Sovrin BuilderNet Accepted SCHEMA request	200,70	195653,52	[1104, 3374]	A to S 30k S to A 42k
Sovrin BuilderNet Rejected CLAIM_DEF request	1386,85	1263779,29	[451, 4722]	A to S 229k S to A 25k	Sovrin BuilderNet Rejected CLAIM_DEF request	2907,20	190724,27	[1059, 3066]	A to S 231k S to A 24k
Sovrin BuilderNet Accepted CLAIM_DEF request	1461,15	383093,93	[723, 2562]	A to S 243k S to A 274k	Sovrin BuilderNet Accepted CLAIM_DEF request	1683,40	460268,04	[974, 3930]	A to S 237k S to A 252k

Table 6.1: Two iterations of 20 requests of 4 type on **SovrinBuilderNet**.

Contextualize results: BuilderNet measures the results - how do they matter?

As said in section 3.1 Sovrin deploys three networks: BuilderNet, StagingNet and MainNet. BuilderNet has had the fewest transactions of all three Sovrin networks. The other two networks are StagingNet and MainNet, which were deployed earlier than BuilderNet.

The result of those tests displays a remote Indy network performance.

Conclusions

It is possible to extract some key findings from those results.

- The time it takes for the pool to answer the agent is significantly shorter if the SCHEMA transaction fails. We can assume that

the structural check on requests is on average 188ms, with the maximum measured at 632ms.

- After seeing the discrepancy between an accepted CLAIM_DEF in iteration 1 and 2, further tests were performed showing that the time taken to perform a CLAIM_DEF transaction is strongly variable.
- When a transaction is accepted (SCHEMA or CLAIM_DEF), the bytes incoming from the pool to the agent are more than when a transaction fails.

Precisely when a transaction is successful, the pool sends a successful message alongside the transaction as formatted on the ledger. Agents should be prepared to receive several bytes more than what they sent.

- The size of a CLAIM_DEF transaction is greater than a SCHEMA transaction. We can assume that it is the product of its more complex structure. It has to be considered that the CLAIM_DEF contains two public keys, as seen in figure 3.20.
- The answer of a failed CLAIM_DEF transaction takes more time than a failed SCHEMA transaction. The nature of the failure is an erroneous transaction sequence number. During testing, it appears that the transaction identified by the sequence number provided in the request has been checked. If the id does not correspond to the SCHEMA transaction Id provided, the transaction fails. If the transaction Id does not correspond to the Credential Schema request, the transaction is invalid. A transaction is proven invalid when the transaction Id does not correspond to the Credential Schema Id provided in the request.
- The BuilderNet experience seems seamless overall between requests and results. From a human point of view, there was no interruption between request and response.

6.2.2 Local Indy pool test - performance on unique transactions submission

This subsection describes the measures in a **local Indy pool**. The results will be confronted with the measures of the previous subsection.

Reading operations from the ledger will also be considered.

Testing methodology

The goal of this subsection is to uncover the local Indy pool responsiveness. The cases are the same as the previous subsection.

The test can also be indicative of the performance of Indy in the ledger with 500 and more transactions on domain ledger, 4 transactions on Pool ledger, 0 transactions on Config ledger, for a total of at least 504 transactions, all aggregated as copies in the domain ledger. A node stores over 1004 transactions. The final aim of this test is to highlight the performance of a single operation with a series of different requests. Read request performance was also measured, which were considered five types of read requests. The results are shown in Table 6.2a and can be seen in Appendix A.7.5. One iteration of the test is presented in appendix A.7. Table 6.2b gives an overview of the results.

Read requests on a Local Pool	Avarage (ms)	Variance (ms)	[MIN, MAX] (ms)
Get Schema request	11,27	6.86	[6, 13]
Get Credential Definition	15,62	23,98	[23, 7]
Get Revocation Registry delta	25,62	45,12	[38, 14]
Get Revocation Registry definition	10,125	26,98	[5, 21]

Operations Iteration 1	Avarage (ms)	Variance (ms)	[MIN, MAX] (ms)
Local pool Rejected SCHEMA request	9,90	21,25	[3, 18]
Local pool Accepted SCHEMA request	2912,85	489344,13	[70, 3045]
Local pool Rejected CLAIM_DEF request	1299,00	1000281,36	[41, 3018]
Local pool Accepted CLAIM_DEF request	929,40	955522,04	[84, 2545]

(a) Overview of read transactions performance on 8 requests of 4 types on a local pool.

(b) One iteration of 20 requests of 4 types on a **local pool**.

Table 6.2: Local pool read/write requests

Contextualize results: local Indy pool measures results - how do they matter?

This experimentation removes the overhead given by remote network communication. In turn, the computational power of the nodes is reduced. Each node is set up in a docker container. The local pool has four nodes (all validators), whose consensus will be reached faster than the BuilderNet, given that nodes are fewer and communicate locally. Requests are received one at a time from only one agent, differently than the BuilderNet where the number of requests at a given time is not controllable.

Conclusions

By comparing the results collected locally to those of the previous subsection, some key findings can be highlighted.

- Confirming the results of the previous subsection. Integrity check on SCHEMA transaction requests is fast. This measure can help us understand the overhead of remote communication with BuilderNet. The two iterations were completed in around 3.5-4 seconds with BuilderNet. Locally, the operation was completed in around 0.2 seconds. With the BuilderNet iterations, the network adds at least 3 seconds of overhead.
- The accepted SCHEMA transaction highlighted a certain pattern in the batch of transactions. Locally, the first transaction being sent receives a response much faster than the subsequent ones. The first transaction takes less than 0.1 seconds, while the subsequent all-around 3 seconds.
It could mean that the first submission of a series is threatened differently than the rest. It may be that limitations are imposed on subsequent write operations. Or most probably, the pool takes time to perform internal operations caused by the transaction after accepting it.
- The time to perform 20 successful SCHEMA transactions is around 60 seconds. Both locally and on BuilderNet. This happens with the CLAIM_DEF transaction too, taking around 30 seconds to perform it. Computational power offsets the network overhead.

- A failed CLAIM_DEF as in the previous subsection, takes more time than a failed SCHEMA transaction. The operation seems to be at least 9 seconds faster locally.

6.2.3 BuilderNet and local Indy pool on a specific request repeated

In this subsection measures on a CLAIM_DEF transaction request submitted multiple times will be compared operating on a local Indy pool and the BuilderNet.

Testing methodology

Create a single CLAIM_DEF request and measure the time of the *signAndSubmitRequest()* for the request created 20 times in a row. The transaction will be accepted, even if the data of a Credential Definition is updated. Results can be seen in appendix A.8. The Table 6.3 shows the comparison between the local pool and theBuilderNet performances.

CLAIM_DEF requests signAndSubmitRequest	Avarage (ms)	Variance (ms)	[MIN, MAX] (ms)
BuilderNet	541,35	140139,92	[86, 1190]
Local pool	15,62	25,62	[2, 81]

Table 6.3: Confrontation of a local pool to BuilderNet to 20 submissions of the same (CLAIM_DEF)request.

Contextualize results: measure a batch of subsequent transactions from the same request - why does it matter?

In this case, the submission of requests is much faster than the previous one. Agents do not have to build a new request for every submission.

Conclusions

Even given the sample size of the two iterations, it is possible to extract some key findings from those results. Locally, the first transaction is much slower than the subsequent, probably due to caching of recent transactions. Finally, both locally and on BuilderNet, the execution time of subsequent identical requests is less than the time to handle a series of unique requests (Credential Definition requests)

6.3 Agent side operations measurements

In this section, the measurement will focus on operations not involving the ledger. As seen in section 3.6, the majority of operations of the Anoncreds workflow do not involve the ledger.

6.3.1 Testing methodology

The measurements were collected using the method *collectMetrics()* included in the package Metrics (mentioned in section 5.1) of the wrapper. With *collectMetrics()* the metrics are directly collected from the libindy command *collect_metrics*. For every type of command, a counter is returned. The time to execute the commands is expressed in milliseconds. The time metrics of a type of command are an aggregation of all the command executions. The dimensions of Anoncreds entities (as Strings) were measured using the method *getBytes()* provided by Java and accessing its field length. Table 6.4 express the results in table form. The data used as a reference are in appendix 6.3.2. The credentials used in this test are revocable (an additional check) and have two attributes.

6.3.2 Proof request and proof in different cases

Off ledger operations metrics This subsection presents a collection of measurements regarding agent side operation during the creation of a proof.

Proof Request format	Proof Request Size(bytes)	Proof Size (bytes)	Holder create proof command (ms)	Verifier verify proof
1 attribute requested 1 restriction (Credential Schema ID)	485	35498	0*(40)	42
1 attribute requested 2 restrictions (Cred.Schema_Id, Cred.Def_Id)	910	35555	34	34
2 attributes requested individually (can belong to differents credentials)	913	35582	41	39
2 attributes from the same credential. 2 restrictions(IssuerDID, Cred.Def_Id)	608	35418	34	34
1 predicate	532	76405	77	78
2 predicates	907	117781	117	115

Table 6.4: Proof creation and verification metrics. *The collect_metrics command always gives 0ms as time execution for a 1 attribute proof. It is unclear why it has this behavior. At Java execution level the difference between System.currentTimeMillis() showed 40ms as time of execution.

Conclusions

- The addition of an attribute is minimal on the proof/proof request size.
- The proof dimensions are in the order of kilobytes
- Predicates can rapidly make a proof size bigger. A second predicate added 41376 bytes to a proof.
- The time to verify/create a proof containing predicates is at least twice as long as a proof with only attributes.

6.3.3 Credential measures

Dimensions on credentials with different attributes are considered. All the Anoncred entities relevant to the the credential creation process are

being measured.

The table 6.5 shows the difference in sizes of the Anoncred entities.

Credential Creation	Credential Offer size (bytes)	Credential Request size (bytes)	Revocation Registry entry size (bytes)	Credential size (bytes)
1 attribute	2398	35498	868	5330
2 attributes	3106	35555	5390	5390

Table 6.5: Size of Credentials

Time measurements were performed on a credential with two attributes. The values in Table 6.6 are the average of 7 iterations.

Average Credential creation (2 attributes)	Credential Offer (Issuer)	Credential Request (Holder)	Credential creation (Issuer)	Credential store (Holder)	Build Credential Definition (Issuer)	Build Credential Schema (Issuer)
Time (ms)	1 (*)	31	66	29	12	1(*)

Table 6.6: Time of execution to create Credentials. *The resulting time was always 1

Conclusions

- The credential issuing process is more demanding (computationally wise) to the issuer.
- The holder has to create a Credential Request and also store the credential. These operations together account for 60ms of time.
- Overall a proof presentation (no predicates) and the credential creation process has the same cost (computationally wise).

6.4 Node Validator info and performance measures

This section discusses the result of the *GetValidatorInfo()* method (mentioned in section 5.2.2). The measures were performed on nodes used

in the local implementation Shipping Verification System. The measurement results returned are updated in the nodes every minute [61].

6.4.1 Testing methodology

The method `GetvalidatorInfo()` from `IndyLibraries.StewardAgent` is used for obtaining measures about the performance of pool nodes. The method creates a **GetValidatorInfo** Request.

GetValidatorInfo Request

`GetValidatorInfo` has a unique pattern compared to the requests in this chapter. It does not create a transaction, meaning it is not sent for verification to the whole pool. Steward agents do not submit `GetValidatorInfo` requests to the pool, instead they submit them to nodes individually.

```
public String GetvalidatorInfo(){
    try {
        ...
        String requestResult=submitAction(this.poolConnection,SignedvalidatorInfoReq,null,-1).
        get();
    }
```

The method (from wrapper class `Ledger`) `submitAction(Pool pool,String requestJson,String nodes,int timeout)` is used to perform the request. This is a request submitted individually to the nodes specified in the "String nodes" parameter. If the nodes parameter is null, all nodes receive the request. Figure 6.4 shows the operation to perform the request on a node. Steward performs a `GetValidatorInfo` request after an iteration of item orders and collections in the Shipping Verification System.

After request, the Steward does receive a response from all four nodes in the local pool. For reference, only the result of node 3 can be seen in the Appendix A.9.2.

6.4.2 Contextualize the results: what can be seen?

Non performance contents

The validator info contains information about the current status of the pool. The field "Quorum" gives insights into the number required for node approval to perform internal operations. It can be seen how the factor **f** mentioned in subsection 3.2.3 is equal to 1 in this pool. The

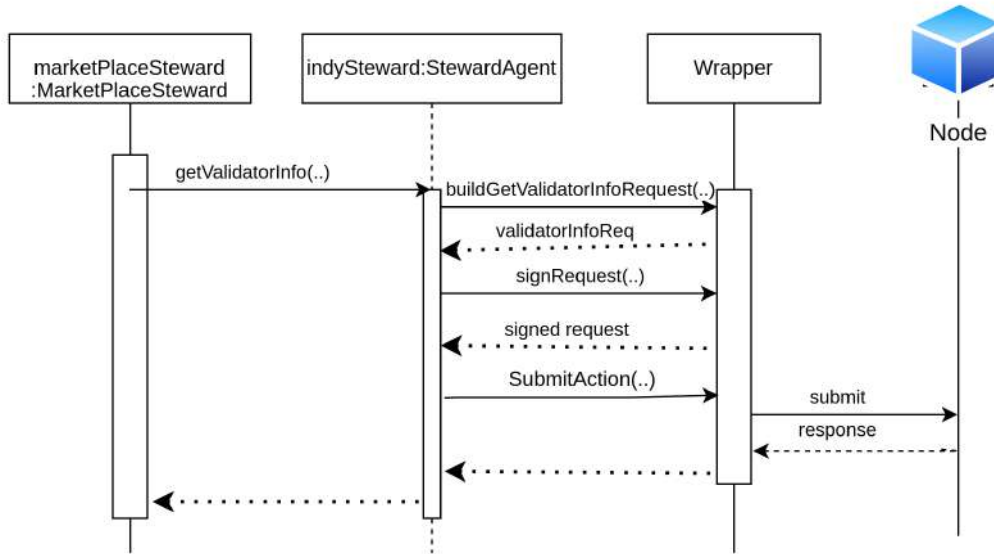


Figure 6.4: Sequence diagram representing the calls to perform a GetValidatorInfo request.

factor f represents the maximum number of malicious nodes allowed in the pool to keep accepting transactions. The "Read_only" field indicates if the pool is set in read-only mode (as said in subsection 3.4.2, which can be set by the Trustee). Finally, there is a BLS key field containing the BLS key of the node mentioned in section 3.4.2.

Performance measures

Various excerpts of the Validator info from Appendix A.9.2 are shown below.

```

1 "Metrics":{
2   "instances started":{
3     "0":9972.932849743,
4     "1":9972.93820727
5   },

```

As can be seen, the nodes run two instances of the consensus protocol. As said in section 3.2.3 (and can be seen in figure 3.3) more than one instance is in execution. The aforementioned subsection also mentions that multiple instances are used to measure the performance of the Master Instance with other instances. The Master Instance is an instance of the consensus protocol used to order transactions. The backup instances

act as a benchmark to it(as said in subsection 3.2.3).

```

1  "ordered request counts":{
2    "0":8,
3    "1":8
4  },
5  ...
6  "ordered request durations":{
7    "0":11.44397719,
8    "1":4.268943218
9  },
10 "throughput":{
11   "0":0.0029512197,
12   "1":0.0029512196
13 },

```

Based on the excerpt above, it is possible to see the time to complete 8 transactions. The "ordered request durations" structure contains the cumulative execution time for the two instances. In the "ordered request counter" the two instances counted 8 requests. It can be seen that the backup instance "1" was faster than the default Master Instance ("0"). The low performance of instance "0" has a consequence, as nodes can vote to change Master Instance. This operation is called View Change. In this case, the Master Instance was switched to instance "1". As indicated in the field View_No below.

```

1  "View_No":1,
2  ....
3  "IC_queue":{
4    "2":{
5      "Voters":{
6        "Node4":{
7          "reason":25
8        }
9      }
10   }
11 },

```

The IC_queue field shows that a message of requested view change was received by node 3. Node 4 asked for a View Change. The "reasons":25 entry indicates: 25 - master primary has too low performance [62].

Conclusions

Validator info requests can provide valuable insights into node performance. They provide an understanding of the pool status, which is valuable to a Steward agent. Meanwhile, it is also seen how the pool monitors its internal performance to switch the Master Instance.

Chapter 7

Conclusions

This work done during this thesis was studying the Hyperledger Indy technology and using it to implement a use case, i.e. a **SSI** solution for **privacy preservation** to a delivery system, from its design to its implementation. As seen, the system can easily be deployed on various Indy **pools**, on a local network, or a global network (as **Sovrin**). Tests on the system showed its solidity in borderline cases such as proof tampering. The performance appeared to be solid, both in local pool measurements and with the Sovrin MainNet. The management of a pool seemed to be well supported with utility transactions such as **ValidatorNode** requests.

One of the main challenges was the understanding of the Indy system. While Indy offers wrappers for various programming languages, it was not developer-friendly. For example, Java wrapper methods often relied on Json structures as a parameter. If a Json structure has a discrepancy with the expected one an exception is thrown. Understanding the cause of the error was not aided by the Wrapper. The code documentation provided appears to be great on certain methods and concepts. On other occasions, information was difficult to find. It occurs that certain operations would throw an unexpected exception. Inserting an attribute as a revealed attribute and as a **predicate** in a proof request was not allowed. This occurrence is not documented as clearly as it should.

Overall, **Indy** and its internal library **libindy** are powerful and efficient tools to develop an SSI system. All functionalities promised by Indy are available. Also, promising forward steps are being done on an Indy DID Method specification. Implementing an API on top of Indy, such as Indylibraries, is a viable solution to minimize errors. The ease of use is

better after a certain point in the learning curve of the Indy-SDK. The Indy architecture allowed to minimize the **ledger** involvement. The majority of **Anoncreds entities** were exchanged privately. **Zero knowledge proofs** allow customers to retrieve their orders without disclosing their identity. Using **selective disclosure** only the required **attributes** need to be disclosed from a credential. It has to be said that without ledger involvement a system level of trust depends on its actors. If **Endorsers**, **Stewards**, and **Trustees** are not malicious, the system can be trusted. This can be seen as a trade-off between decentralization

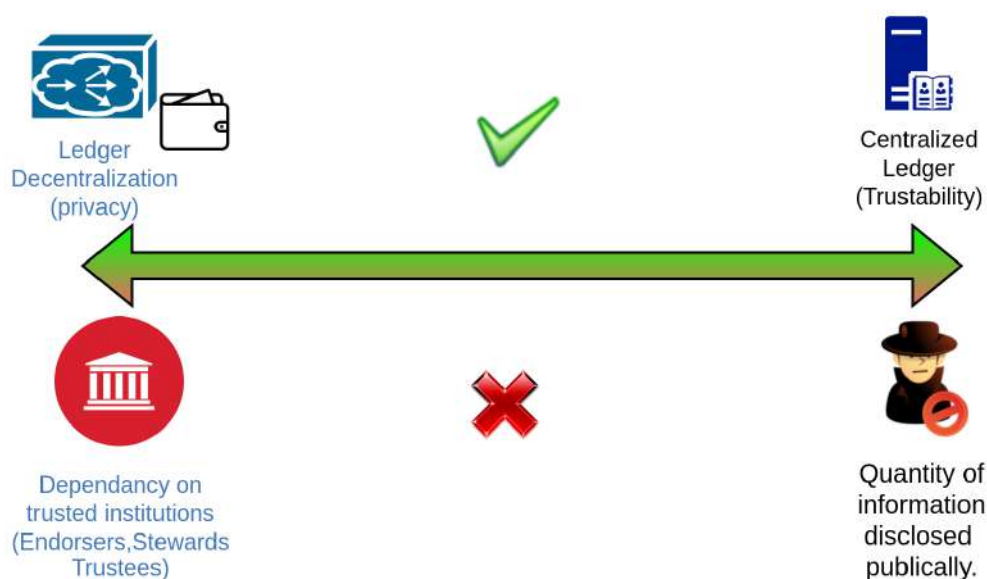


Figure 7.1: Figure representing the tradeoff between privacy in a system and trustability of a system.

from the ledger and reliance on agents with a **role** (Figure 7.1). **Hybrid solutions** with other blockchains can be used as insurance for "NYM-less" agents.

The anonymity of agents during their private communications was preserved using **DID**. Pairwise DID communication made also possible to authenticate the origin of the message exchanged. The attributes of an **Identity Holder** are preserved from correlation. This is the product of session **unlinkability**. **DID** not only were used in agent-to-agent communication. **DID** allowed for agents to have a **role** on the ledger (with a **NYM** public record). The documentation on Indy architecture

and concepts is rich but too sparse and in different sources. It can be lackluster in certain areas. In some cases, it was necessary to refer to the code documentation.

A case of **credential portability** was also illustrated with the Student Identity case. It was seen how the **Plenum Byzantine Fault-Tolerant Protocol** is a valid alternative to the energy-demanding Proof Of Work (of Bitcoin and Ethereum). Indy and **Public Permissioned** are perfectly suitable for an SSI system. Zooko's triangle was proven wrong in this case too. DID identifiers, credentials, and proofs are all human-readable, decentralized, and architected to be secure.

7.1 Future improvements

Who controls the controllers? Preventing malevolent behavior from Endorsers, Stewards, and Trustees could pose a challenge. With regards to the Shipment Verification System, the Store could pose a threat. For example, by selling the same item to more than one customer. The first improvement of the Shipment Verification System could be solving this issue.

Exploring the further capabilities of the Indy-SDK could also provide various new features. As said in the introduction chapter, some people do not use online shops for fear of payment security or privacy. Indy can provide a solution for payment security as well. As Indy provides a [payments API](#)¹. The system could also allow its customers to cancel an order, even if it is currently being shipped. This feature could include credential revocation. This way, the Mail Box can instantly be made available for new orders, and the credentials to deliver the items that belong to the Shipping Deliverer can be revoked, meaning that any attempt to deliver the canceled order can only fail.

In the future, the agent of the customer code could be adapted into an Android application since it is written in Java and Indy supports Android too.

Finally, an Indy public permissionless blockchain could be deployed, creating a consortium (as Sovrin) that starts the network and specifies the rules to join for the online shops.

¹<https://github.com/hyperledger/indy-sdk/tree/master/libnullpay> Indy payments API

Appendix A

Appendix

A.1 DID Document

DID Document of did:sov:HR6vs6GEZ8rHaVgjjg2WodM

```
1  {
2  "@context": [
3    "https://www.w3.org/ns/did/v1",
4    "https://w3id.org/security/suites/ED25519-2018/v1",
5    "https://w3id.org/security/suites/x25519-2019/v1"
6  ],
7  "id": "did:sov:HR6vs6GEZ8rHaVgjjg2WodM",
8  "verificationMethod": [
9    {
10     "type": "ED25519VerificationKey2018",
11     "id": "did:sov:HR6vs6GEZ8rHaVgjjg2WodM#key-1",
12     "publicKeyBase58": "9wvq2i4xUa5umXoThe83CDgx1e5bsjZKJL4DEWvTP9qe"
13   },
14   {
15     "type": "X25519KeyAgreementKey2019",
16     "id": "did:sov:HR6vs6GEZ8rHaVgjjg2WodM#key-agreement-1",
17     "publicKeyBase58": "3mHtKcQFEzqeUcnce5BAuzAgLEbqKaV542pUf9xQ5Pf8"
18   }
19 ],
20 "authentication": [
21   "did:sov:HR6vs6GEZ8rHaVgjjg2WodM#key-1"
22 ],
23 "assertionMethod": [
24   "did:sov:HR6vs6GEZ8rHaVgjjg2WodM#key-1"
25 ],
26 "keyAgreement": [
27   "did:sov:HR6vs6GEZ8rHaVgjjg2WodM#key-agreement-1"
28 ],
29 "service": [
30   {
31     "type": "endpoint",
32     "serviceEndpoint": "https://bc-reg-agent.orgbook.gov.bc.ca"
33   },
34   {
35     "type": "did-communication",
```

```

36     "id": "did:sov:HR6vs6GEZ8rHaVgjjg2WodM#did-communication",
37     "serviceEndpoint": "https://bc-reg-agent.orgbook.gov.bc.ca",
38     "priority": 0,
39     "routingKeys": [],
40     "accept": [
41         "didcomm/aip2;env=rfc19"
42     ],
43     "recipientKeys": [
44         "did:sov:HR6vs6GEZ8rHaVgjjg2WodM#key-1"
45     ]
46 },
47 {
48     "type": "DIDComm",
49     "id": "did:sov:HR6vs6GEZ8rHaVgjjg2WodM#didcomm-1",
50     "serviceEndpoint": "https://bc-reg-agent.orgbook.gov.bc.ca",
51     "accept": [
52         "didcomm/v2",
53         "didcomm/aip2;env=rfc19"
54     ],
55     "routingKeys": []
56 }
57 ]
58 }

```

A.2 Proof

Proof from proof request in subsection 3.5.9

```

1  {
2      "identifiers": [{
3          "rev_reg_id":
4              "MchNFAFVJ8a9qyJvVWBqyG:4:
5              MchNFAFVJ8a9qyJvVWBqyG:3:
6              CL:214:TAG2:CL_ACCUM:Hi",
7          "schema_id":
8              "MchNFAFVJ8a9qyJvVWBqyG:2:
9              StudentIdentity:1.0",
10         "cred_def_id":
11             "MchNFAFVJ8a9qyJvVWBqyG:3:CL:214:TAG2",
12         "timestamp": 1642436585
13     }],
14     "proof": {
15         "proofs": [{
16             "primary_proof": {
17                 "eq_proof": {
18                     "e": "7230853...",
19                     "m2": "220881...",
20                     "v": "...."
21                 },
22                 "a_prime": "...",
23                 "revealed_attrs": {},
24                 "m": {
25                     "master_secret":
26                         "5455930....",
27                     "name":
28                         "11689661...."
29                 }
30             }
31         }
32     ]
33 }

```

```

28         }
29     },
30 },
31     "ge_proofs": []
32 },
33     "non_revoc_proof": {
34         "x_list": {
35             "o_prime": "122E...",
36             "c": "117A7C...",
37             .....
38             .....
39             .....
40         },
41         "c_list": {
42             "a": "...."
43             ....
44             ....
45         }
46     }
47 },
48     "aggregated_proof": {
49         "c_hash": "129557301682..",
50         "c_list": [
51             [4....], [...], ...]
52 },
53     "requested_proof": {
54         "predicates": {},
55         "self_attested_attrs": {},
56         "revealed_attrs": {},
57         "unrevealed_attrs":
58             {"attr0_referent":
59                 {"sub_proof_index": 0}}
60 }
61 }

```

A.3 Valid proof and an alteration of it(non-valid)

A.3.1 Valid proof

```

1 {
2     "identifiers": [{
3         "rev_reg_id": null,
4         "schema_id": "YRDrcLTz9CB4Ly8PbtLB2i:2:customershipping:2.0",
5         "cred_def_id": "YRDrcLTz9CB4Ly8PbtLB2i:3:CL:545:tag",
6         "timestamp": null
7     }],
8     "proof": {
9         "proofs": [{
10             "primary_proof": {
11                 "eq_proof": {
12                     "e": "14166405919962..",
13                     "m2": "12273050539646..",

```

```

14         "v": "769929433217019.." ,
15         "a_prime": "35961018082087..",
16         "revealed_attrs": {
17             "shippingid": "1",
18             "shippingnonce": "-1378345762"
19         },
20         "m": {
21             "lockerboxid": "32444994215700160525708734341..",
22             "master_secret": "138279688441911069560519766..",
23             "itemname": "11478480223977800037182545334765..",
24             "storedid": "90888880386224687264967501586615..",
25             "diddest": "315858724533007130991142677322979.."
26         }
27     },
28     "ge_proofs": []
29 },
30     "non_revoc_proof": null
31 },
32     "aggregated_proof": {
33         "c_hash": "83042424..",
34         "c_list": [[1,28,221, 187,...]]
35     }
36 },
37     "requested_proof": {
38         "predicates": {},
39         "self_attested_attrs": {},
40         "revealed_attrs": {},
41         "revealed_attr_groups": {"attr0_referent": {
42             "values": {
43                 "shippingid": {
44                     "raw": "1",
45                     "encoded": "1"
46                 },
47                 "shippingnonce": {
48                     "raw": "-1378345762",
49                     "encoded": "-1378345762"
50                 }
51             },
52             "sub_proof_index": 0
53         }},
54         "unrevealed_attrs": {}
55     }
56 }

```

A.3.2 Altered proof(non-valid)

The following manufactured proof was produced by the agent scammer-Customer, its implementation can be found in the repository [GitHub IndyWorking](#).

```

1 {
2     "identifiers": [{
3         "rev_reg_id": null,
4         "schema_id": "YRDrcLTz9CB4Ly8PbtLB2i:2:customershipping:2.0",
5         "cred_def_id": "YRDrcLTz9CB4Ly8PbtLB2i:3:CL:545:tag",

```

```

6      "timestamp": null
7    },
8    "proof": {
9      "proofs": [{
10         "primary_proof": {
11           "eq_proof": {
12             "e": "14166405919962..",
13             "m2": "12273050539646..",
14             "v": "769929433217019",
15             "a_prime": "35961018082087",
16             "revealed_attrs": {
17               "shippingid": "2",
18               "shippingnonce": "690638353"
19             },
20             "m": {
21               "lockerboxid": "32444994215700160525708734341..",
22               "master_secret": "138279688441911069560519766..",
23               "itemname": "11478480223977800037182545334765..",
24               "storedid": "90888880386224687264967501586615..",
25               "diddest": "315858724533007130991142677322979.."
26             }
27           },
28           "ge_proofs": []
29         },
30         "non_revoc_proof": null
31       ]},
32     "aggregated_proof": {
33       "c_hash": "83042424..",
34       "c_list": [[1,28,221, 187,...]]
35     },
36   },
37   "requested_proof": {
38     "predicates": {},
39     "self_attested_attrs": {},
40     "revealed_attrs": {},
41     "revealed_attr_groups": {"attr0_referent": {
42       "values": {
43         "shippingid": {
44           "raw": "2",
45           "encoded": "2"
46         },
47         "shippingnonce": {
48           "raw": "690638353",
49           "encoded": "690638353"
50         }
51       },
52       "sub_proof_index": 0
53     }},
54     "unrevealed_attrs": {}
55   }
56 }

```

A.4 Tables of entities (Chapter 3)

	Description	Area
DID	String identifier. It can be used to identify a transaction maker. It can also be used as an identifier in a communication.	Global
Pairwise DID	DID used in a two way communication to authenticate messages.	Global
DID Indy	Indy DID method specification is still in draft form.	Global
DID Sov	Sovrin DID method specification. Used in Sovrin network. fully qualified DID example: did:sov:HR6vs6GEZ8rHaVgJg2WodM	Global
DID in Indy SDK	Unqualified DID, containing an identifier such as HR6vs6GEZ8rHaVgJg2WodM. It can be converted to fully qualified DID. Supports DID from many DID method specification. It can be created with createAndStoreMyDid().	Indy
Verkey/Verification key	Public key paired with a DID. Used by parties to verify a signature with the paired DID.	Indy
Agent (Software)	Program used communicate with an SSI system.	Global
Ledger	Collection of transactions.	Indy
Pool	set of nodes (verifiers and not), storing the ledger.	Indy

Table A.1: Table of general concepts

Entity	Description	Area	Write request(Transaction)	Read request
NYM	Transaction containing a DID and its verkey, it creates a NYM record.	Indy	NYM	GET_NYM
NYM record	An object on the ledger containing pieces of information related to a particular DID.	Indy	-	-
ATTRIB	Transaction to add information in a NYM record. Usually used to add an endpoint to DID.	Indy	ATTRIB	GET_ATTRIB

Table A.2: Identity-related entities

	Description	Sender	Receiver
Credential Offer	Offer of a credential to an agent. It identifies the Credential Definition	Issuer	Holder
Credential Request	Request of a credential to an issuer.	Holder	Issuer
MasterSecret	Link of a credential to its holder. Needed to create proof.	Holder	Issuer (inside Credential Request in blinded form)
Credential	Credential issued from a prover to an holder. Issued from a Credential Definition.	Issuer	Holder
Attribute	Single component of a credential.	-	-
Predicate	Statement about an attribute. Allows to verify that an attribute belongs to a range without asking to disclose it.	-	-
Proof Request	Request of a set of attributes/predicates. Sent by a verifier to an holder	Verifier	Holder
Proof	Zero Knowledge Proof an holder. Used to prove he has the attributes and valid credentials to satisfy a proof request	Holder	Verifier

Table A.3: Anoncreds entities (off ledger)

	Description	Write request (transaction)	Read request
Credential Schema	Public definition on the ledger of a credential format. It defines the credential attributes. Can be used by more than one Issuer.	SCHEMA	CLAIM_SCHEMA
Credential Definition	Public statement on the ledger from an Issuer. It specifies that credentials from a Credential Schema will be issued. Can define the credential as revocable or non-revocable.	CLAIM_DEF	GET_CLAIM_DEF
Revocation Registry	Structure associated with a Credential Definition.	REVOC_REG_DEF	GET_REVOC_REG_DEF
Accumulator	A value used to calculate if a credential was revoked or not.	REVOC_REG_ENTRY	GET_REVOC_REG
Revocation Registry Delta	Accumulator and sets of revoked credentials/ issued credentials in a Revocation Registry.	(consequence of REVOC_REG_ENTRY)	GET_REVOC_REG_DELTA

Table A.4: Anoncreds entities (on the ledger)

A.5 Setup Indy

The end goal of this appendix is to set up Indy SDK for use and use in a local pool.

A Linux-based distribution system will be used as a reference.

But the provided references will mention other systems(Windows, macOS,..).

A.5.1 Installing libindy on a Linux based system

The component of Indy SDK needed in our case is just libindy.

Installing Indy SDK on a Linux-based system consists in adding a repository to APT.

After adding the repository, following the guide in [17][section Installing the SDK] is needed only the command

```
$ sudo apt-get install -y libindy
```

Indy repository could generate some APT errors. It is possible to remove the repository after installation.

A.5.2 Run a local pool with Docker

Docker is needed(or at least it is the advised solution) to run a local pool.

It is possible to [install Docker](#)¹ following the instruction of its website(APT installation is supported).

Our pool will run on Docker containers.

A.5.3 Get and use Indyscan containers image for nodes

Instead of using the test nodes provided by Hyperledger [17][section How to start local nodes pool with docker], Indyscan nodes are going to be created.

As we saw in chapter 3 Indyscan allows the reading of transactions stored on the ledger.

To use Indyscan nodes it is needed to clone the Indyscan repository from GitHub [37]. It is also possible to download it as ZIP and directly unpack it.(Figure A.1)

¹<https://docs.docker.com/engine/install/ubuntu/>- install Docker

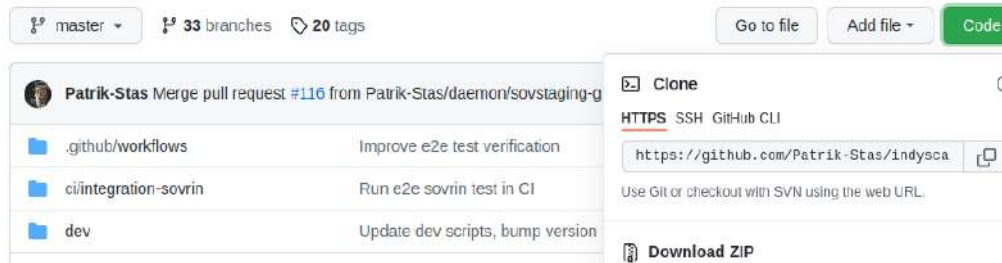


Figure A.1: Download as ZIP after having pressed the code button.

If it is desired to clone the repository: install installed [git](#)² and to run the command :

```
git clone https://github.com/Patrik-Stas/Indyscan.git
```

After the Indyscan repository is on the filesystem, using as current working directory `/PathToIndyscan/indyscan/start`. The command needed to build the containers using Indyscan images are:

```
$ docker-compose -f docker-compose.es.yml -f
docker-compose.yml up -d
```

It has to be noted that using the above command is a different approach to the documented procedure [37].

After docker containers are running, the Indyscan pool has to be added to the list of known Indy networks by the system[37].

To do so from `/PathToIndyscan/indyscan/` as the working directory it is needed to run the script

```
$ ./init-local.sh
```

. The network identified as INDYSCANPOOL will be added to the list of Indy known networks.

Before being able to connect to the local Indyscan pool, there needs to be an addition to file `/etc/hosts`.

```
'127.0.0.1 indyscanpool'
```

This forwards the connection to the pool running in the localhost, by linking the pool name to the loopback address.

To see the contents of the ledgers like in figure 3.1 it is needed to navigate to **127.0.0.1:3707**.

²<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git> - Install Git

A.5.4 Client side with Indy-SDK using Java wrapper

Subsection A.5.1 provided the operations to set up libindy on a local machine. In this subsection, the Java wrapper setup process will be illustrated.

The Hyperledger provided setup is on its on a [GitHub page](#)

The Indy wrapper is composed of classes with methods that call the underlying libindy commands.

All the Indy wrapper classes are available in a library "org.hyperledger.indy".

The Indy wrapper also uses SLF4J for logging. SLF4J allows the redirection on standard output of messages/errors and results from the underlying libindy commands.

To integrate the Indy wrapper library in a Java project it is advised to use Maven. Maven allows defining the libraries to import in a file pom.xml.

IntelliJ IDEA provides a [Maven](#)³ integration but it can be possible to use it with other IDE. The pom.xml used in the following implementation can be found on IndyWorking([pom.xml](#)) repository at GitHub and is the advised one to use.

Old versions of the Indy library can lead to unexpected behavior or missing classes.

The SLF4J version also changes the amount of information that can be seen on the standard output.

To use the Indy wrapper it is needed to build the Indy SDK from source. It can be done following the [Hyperledger](#)⁴ guide.

To do so Rust must be installed in the local machine. Rust can be installed from its official [website](#)⁵. The step Build indy from the guide can be done without using the git command. By downloading the Indy SDK repository (or only the libindy directory). And performing the cargo build operation using the libindy directory as the current working directory. This process produces the shared libraries libindy.so.

If the Java wrapper raise a NullPointerException performing operations with libindy: it may be necessary to move the libindy.so to

³<https://maven.apache.org/install.html> - Install Maven

⁴Builder Indy SDK <https://github.com/hyperledger/indy-sdk/blob/master/docs/build-guides/ubuntu-build.md#setup-indy-sdk-build-environment-for-ubuntu-based-distro-ubuntu-1604>

⁵<https://www.rust-lang.org/tools/install> - Install Rust

```
/usr/local/lib.
```

Alternatively it is possible to set a Java environment variable to the path of libindy.so, defined as

```
LD{_}LIBRARY{_}PATH=/directorydilibindy/libindy.so
```

On IntelliJ IDEA this could be done with the Run/Debug Configurations window, as in Figure A.2

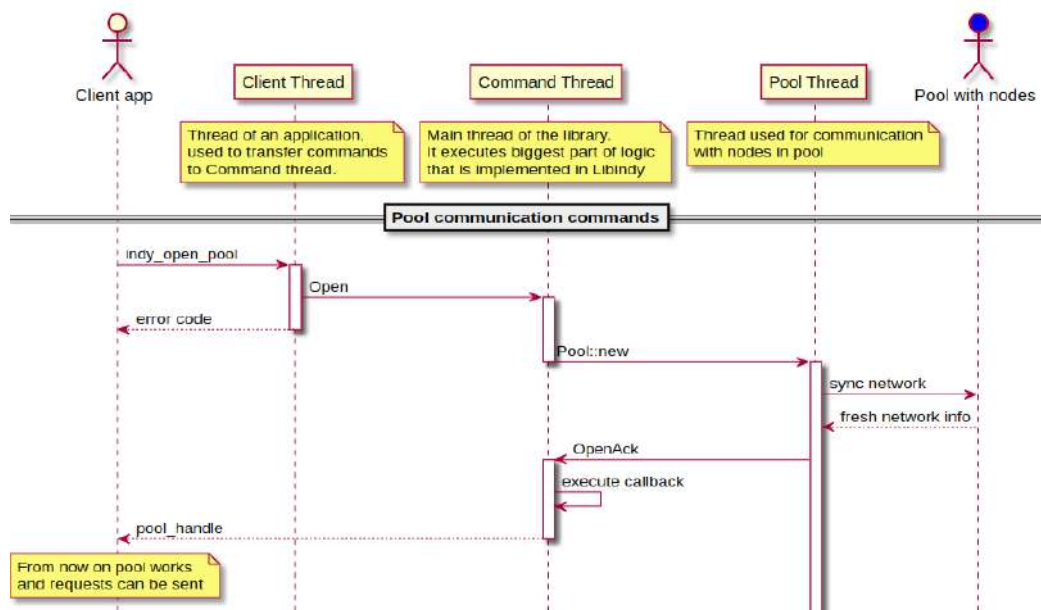


Figure A.2: Run/Debug Configurations window with environment variable addition.

Libindy and Java Future interface

Libindy does perform operations on different threads. The threads work on a task is done in a pipeline. A thread forwards certain types of tasks to another. Each thread executes a callback to the previous thread of the pipeline, returning a result. The threads consist in:

- **Client Thread:** receives a command from a Client, such as a Java wrapper method call and forwards it to the Command Thread. The client agent can resume its activity. The result can be retrieved asynchronously, when ready.

- **Command Thread:** does the majority of operations. It forwards computationally intensive operations to the Expensive Thread. If an operation is related to the pool, it forwards it to the Pool thread.
- **Expensive Thread:** Does computationally intensive operations. Operations involving cryptography such as the creation of a Credential Definition.
- **Pool thread:** Sends transactions to the pool and awaits results before performing a callback.

Figure A.3 is an example of the threads pipeline.

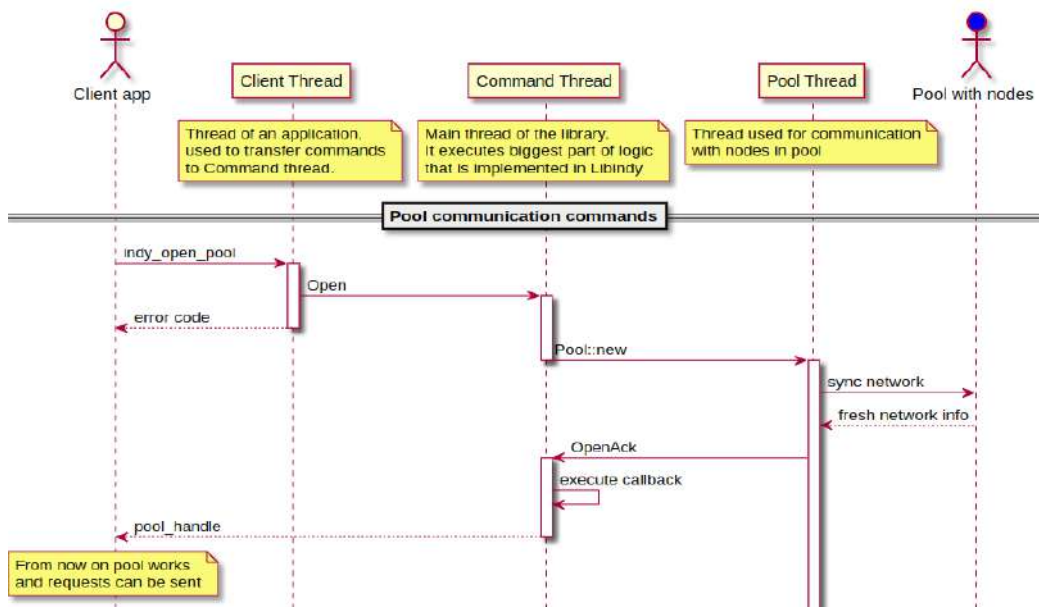


Figure A.3: Portion of sequence diagram[63] corresponding communication with the pool. The operation requested is to open a connection to the pool. After the Pool finishes handshake operations with the pool, a sequence of callback starts. A pool handle is returned to the client agent, which uses it to communicate with the pool.

The Indy-SDK Java wrapper returns for libindy command a method calling it. The Java methods return as a result an object of type `CompletableFuture< T>`. An object CF of type `CompletableFuture<T>` can be set as complete by a thread that has a reference to it with `CF.complete(return_object)`. The client side thread

can check for CF completion with `CF.isDone()` or can wait for the value returned with `CF.get()`.

Sovrin BuilderNet setup

The website [Selfserve](https://selfserve.sovrin.org/)⁶ allows to register a DID created with libindy. To program on the Indy wrapper for Java

1. Follow the following [guide](#)⁷.
Creating a DID with a specific seed.
2. After having obtained the DID. Create a DID with the Indy Java wrapper with the specified seed.
3. In indy-cli type the command `show-tta` and before accepting find the field `Digest:"TAA_DIGEST_VALUE"`.
4. Use the Digest with the method [appendTxnAuthorAgreementAcceptanceToRequest\(\)](#) of Indy wrapper (class Agent).

The methods of `IndyLibraries` (introduced in section 5.1) create requests for the Sovrin MainNet or any other networks which require TTA digests. The methods are: `addEndpointToNYMOnSovrin()`, `publishschemaOnSovrin()`, `IssuerCreateStoreAndPublishCredDefOnSovrin()`, `IssuerCreateStoreAndPublishPrefinedSchemaCredDefOnSovrin()` of `IndyLibraries.Endorser` class.

A.6 Sovrin BuilderNet transaction test operations

Results of the experiments of subsection 6.2.1

⁶selfserve - <https://selfserve.sovrin.org/>

⁷Get on Sovrin DID - <https://ldej.nl/post/connecting-acapy-to-development-ledgers/>

A.6.1 Iteration 1

Rejected SCHEMA request

Time of all operations: 4014ms(4 sec) 4 created TCP conversations.
Bytes agent to Sovrin 18k. Bytes Sovrin to agent 10k.

```
1. SCHEMA request on Sovrin presubmit: 1643329752865 postsubmit: 1643329753350 delta:485
2. SCHEMA request on Sovrin presubmit: 1643329753361 postsubmit: 1643329753453 delta:92
3. SCHEMA request on Sovrin presubmit: 1643329753460 postsubmit: 1643329753543 delta:83
4. SCHEMA request on Sovrin presubmit: 1643329753548 postsubmit: 1643329753652 delta:104
5. SCHEMA request on Sovrin presubmit: 1643329753655 postsubmit: 1643329753747 delta:92
6. SCHEMA request on Sovrin presubmit: 1643329753750 postsubmit: 1643329754241 delta:491
7. SCHEMA request on Sovrin presubmit: 1643329754248 postsubmit: 1643329754340 delta:92
8. SCHEMA request on Sovrin presubmit: 1643329754343 postsubmit: 1643329754438 delta:95
9. SCHEMA request on Sovrin presubmit: 1643329754441 postsubmit: 1643329754541 delta:100
10. SCHEMA request on Sovrin presubmit: 1643329754544 postsubmit: 1643329754636 delta:92
11. SCHEMA request on Sovrin presubmit: 1643329754638 postsubmit: 1643329755144 delta:506
12. SCHEMA request on Sovrin presubmit: 1643329755150 postsubmit: 1643329755241 delta:91
13. SCHEMA request on Sovrin presubmit: 1643329755248 postsubmit: 1643329755340 delta:92
14. SCHEMA request on Sovrin presubmit: 1643329755347 postsubmit: 1643329755450 delta:103
15. SCHEMA request on Sovrin presubmit: 1643329755457 postsubmit: 1643329755569 delta:112
16. SCHEMA request on Sovrin presubmit: 1643329755572 postsubmit: 1643329756062 delta:490
17. SCHEMA request on Sovrin presubmit: 1643329756067 postsubmit: 1643329756162 delta:95
18. SCHEMA request on Sovrin presubmit: 1643329756167 postsubmit: 1643329756273 delta:106
19. SCHEMA request on Sovrin presubmit: 1643329756279 postsubmit: 1643329756384 delta:105
20. SCHEMA request on Sovrin presubmit: 1643329756387 postsubmit: 1643329756494 delta:107
```

Accepted SCHEMA request

Time of all operations: 60316ms(60 sec) 11 created TCP conversations.
Bytes agent to Sovrin 73k. Bytes Sovrin to agent 43k.

```
1)SCHEMA request on Sovrin presubmit: 1643326587006 postsubmit: 1643326587865 delta:8592
2)SCHEMA request on Sovrin presubmit: 1643326587875 postsubmit: 1643326590926 delta:3051
3)SCHEMA request on Sovrin presubmit: 1643326590930 postsubmit: 1643326593965 delta:3035
4)SCHEMA request on Sovrin presubmit: 1643326593973 postsubmit: 1643326596991 delta:3018
5)SCHEMA request on Sovrin presubmit: 1643326596998 postsubmit: 1643326600008 delta:3010
6)SCHEMA request on Sovrin presubmit: 1643326600015 postsubmit: 1643326603053 delta:3038
7)SCHEMA request on Sovrin presubmit: 1643326603066 postsubmit: 1643326606066 delta:3000
8)SCHEMA request on Sovrin presubmit: 1643326606073 postsubmit: 1643326609105 delta:3032
9)SCHEMA request on Sovrin presubmit: 1643326609112 postsubmit: 1643326612103 delta:2991
10)SCHEMA request on Sovrin presubmit: 1643326612109 postsubmit: 1643326615098 delta:2989
11)SCHEMA request on Sovrin presubmit: 1643326615106 postsubmit: 1643326620217 delta:5111
12)SCHEMA request on Sovrin presubmit: 1643326620223 postsubmit: 1643326623273 delta:3050
13)SCHEMA request on Sovrin presubmit: 1643326623279 postsubmit: 1643326626321 delta:3042
14)SCHEMA request on Sovrin presubmit: 1643326626326 postsubmit: 1643326629370 delta:3044
15)SCHEMA request on Sovrin presubmit: 1643326629379 postsubmit: 1643326632389 delta:3010
16)SCHEMA request on Sovrin presubmit: 1643326632394 postsubmit: 1643326635403 delta:3009
17)SCHEMA request on Sovrin presubmit: 1643326635408 postsubmit: 1643326638405 delta:2997
18)SCHEMA request on Sovrin presubmit: 1643326638411 postsubmit: 1643326641439 delta:3028
19)SCHEMA request on Sovrin presubmit: 1643326641443 postsubmit: 1643326644436 delta:2993
20)SCHEMA request on Sovrin presubmit: 1643326644440 postsubmit: 1643326647449 delta:3009
```

Rejected CLAIM_DEF request

Time of all operations: 27737ms(28 sec) 10 created TCP conversations.
Bytes agent to Sovrin 229k. Bytes Sovrin to agent 25k.

```
1. CLAIM_DEF request on Sovrin presubmit: 1643332333666 postsubmit: 1643332334294 delta:628
2. CLAIM_DEF request on Sovrin presubmit: 1643332334305 postsubmit: 1643332339027 delta:4722
3. CLAIM_DEF request on Sovrin presubmit: 1643332339037 postsubmit: 1643332342054 delta:3017
4. CLAIM_DEF request on Sovrin presubmit: 1643332342064 postsubmit: 1643332345054 delta:2990
5. CLAIM_DEF request on Sovrin presubmit: 1643332346091 postsubmit: 1643332348068 delta:1977
6. CLAIM_DEF request on Sovrin presubmit: 1643332354092 postsubmit: 1643332354688 delta:596
```

```

7. CLAIM_DEF request on Sovrin presubmit: 1643332356002 postsubmit: 1643332357693 delta:1691
8. CLAIM_DEF request on Sovrin presubmit: 1643332359004 postsubmit: 1643332360690 delta:1686
9. CLAIM_DEF request on Sovrin presubmit: 1643332364200 postsubmit: 1643332364804 delta:604
10. CLAIM_DEF request on Sovrin presubmit: 1643332368344 postsubmit: 1643332368811 delta:467
11. CLAIM_DEF request on Sovrin presubmit: 1643332373617 postsubmit: 1643332374272 delta:655
12. CLAIM_DEF request on Sovrin presubmit: 1643332375271 postsubmit: 1643332377260 delta:1989
13. CLAIM_DEF request on Sovrin presubmit: 1643332378774 postsubmit: 1643332380274 delta:1500
14. CLAIM_DEF request on Sovrin presubmit: 1643332383126 postsubmit: 1643332383577 delta:451
15. CLAIM_DEF request on Sovrin presubmit: 1643332385594 postsubmit: 1643332386589 delta:995
16. CLAIM_DEF request on Sovrin presubmit: 1643332388184 postsubmit: 1643332389586 delta:1402
17. CLAIM_DEF request on Sovrin presubmit: 1643332392022 postsubmit: 1643332392641 delta:619
18. CLAIM_DEF request on Sovrin presubmit: 1643332395718 postsubmit: 1643332396290 delta:572
19. CLAIM_DEF request on Sovrin presubmit: 1643332400029 postsubmit: 1643332400623 delta:594
20. CLAIM_DEF request on Sovrin presubmit: 1643332404138 postsubmit: 1643332404720 delta:582

```

Accepted CLAIM_DEF request

Time of all operations: 29223ms(30 sec) 11 created TCP conversations.
 Bytes agent to Sovrin 243k. Bytes Sovrin to agent 274k.

```

1. CLAIM_DEF request on Sovrin presubmit: 1643331316975 postsubmit: 1643331317892 delta:917
2. CLAIM_DEF request on Sovrin presubmit: 1643331319999 postsubmit: 1643331320907 delta:908
3. CLAIM_DEF request on Sovrin presubmit: 1643331324044 postsubmit: 1643331324913 delta:869
4. CLAIM_DEF request on Sovrin presubmit: 1643331327507 postsubmit: 1643331328302 delta:795
5. CLAIM_DEF request on Sovrin presubmit: 1643331329579 postsubmit: 1643331331294 delta:1715
6. CLAIM_DEF request on Sovrin presubmit: 1643331334730 postsubmit: 1643331335603 delta:873
7. CLAIM_DEF request on Sovrin presubmit: 1643331338014 postsubmit: 1643331338737 delta:723
8. CLAIM_DEF request on Sovrin presubmit: 1643331339685 postsubmit: 1643331341748 delta:2063
9. CLAIM_DEF request on Sovrin presubmit: 1643331342944 postsubmit: 1643331344787 delta:1843
10. CLAIM_DEF request on Sovrin presubmit: 1643331346207 postsubmit: 1643331347797 delta:1590
11. CLAIM_DEF request on Sovrin presubmit: 1643331349155 postsubmit: 1643331350832 delta:1677
12. CLAIM_DEF request on Sovrin presubmit: 1643331352036 postsubmit: 1643331353819 delta:1783
13. CLAIM_DEF request on Sovrin presubmit: 1643331354345 postsubmit: 1643331356825 delta:2480
14. CLAIM_DEF request on Sovrin presubmit: 1643331359593 postsubmit: 1643331360466 delta:873
15. CLAIM_DEF request on Sovrin presubmit: 1643331361254 postsubmit: 1643331363540 delta:2286
16. CLAIM_DEF request on Sovrin presubmit: 1643331364509 postsubmit: 1643331366537 delta:2028
17. CLAIM_DEF request on Sovrin presubmit: 1643331367003 postsubmit: 1643331369565 delta:2562
18. CLAIM_DEF request on Sovrin presubmit: 1643331372975 postsubmit: 1643331373861 delta:886
19. CLAIM_DEF request on Sovrin presubmit: 1643331375406 postsubmit: 1643331376866 delta:1460
20. CLAIM_DEF request on Sovrin presubmit: 1643331380288 postsubmit: 1643331381180 delta:892

```

A.6.2 Iteration 2

Rejected SCHEMA request

Time of all operations: 3533ms(3.5 sec) 4 created TCP conversations.
 Bytes agent to Sovrin 18k. Bytes Sovrin to agent 10k.

```

1. Schema Definition on Sovrin presubmit: 1643397952466 postsubmit: 1643397953098 delta:632
2. Schema Definition on Sovrin presubmit: 1643397953108 postsubmit: 1643397953209 delta:101
3. Schema Definition on Sovrin presubmit: 1643397953215 postsubmit: 1643397953317 delta:102
4. Schema Definition on Sovrin presubmit: 1643397953320 postsubmit: 1643397953424 delta:104
5. Schema Definition on Sovrin presubmit: 1643397953426 postsubmit: 1643397953553 delta:127
6. Schema Definition on Sovrin presubmit: 1643397953556 postsubmit: 1643397954113 delta:557
7. Schema Definition on Sovrin presubmit: 1643397954122 postsubmit: 1643397954214 delta:92
8. Schema Definition on Sovrin presubmit: 1643397954219 postsubmit: 1643397954421 delta:202
9. Schema Definition on Sovrin presubmit: 1643397954424 postsubmit: 1643397954558 delta:134
10. Schema Definition on Sovrin presubmit: 1643397954564 postsubmit: 1643397954666 delta:102
11. Schema Definition on Sovrin presubmit: 1643397954671 postsubmit: 1643397955161 delta:490
12. Schema Definition on Sovrin presubmit: 1643397955166 postsubmit: 1643397955273 delta:107
13. Schema Definition on Sovrin presubmit: 1643397955279 postsubmit: 1643397955380 delta:101
14. Schema Definition on Sovrin presubmit: 1643397955383 postsubmit: 1643397955481 delta:98
15. Schema Definition on Sovrin presubmit: 1643397955487 postsubmit: 1643397955603 delta:116
16. Schema Definition on Sovrin presubmit: 1643397955606 postsubmit: 1643397956140 delta:534
17. Schema Definition on Sovrin presubmit: 1643397956146 postsubmit: 1643397956253 delta:107
18. Schema Definition on Sovrin presubmit: 1643397956259 postsubmit: 1643397956349 delta:90

```

19. Schema Definition on Sovrin presubmit: 1643397956355 postsubmit: 1643397956473 delta:118
 20. Schema Definition on Sovrin presubmit: 1643397956478 postsubmit: 1643397956578 delta:100

Accepted SCHEMA request

Time of all operations: 58889ms(59 sec) 11 created TCP conversations.
 Bytes agent to Sovrin 30k. Bytes Sovrin to agent 42k.

1. SCHEMA request on Sovrin presubmit: 1643399776008 postsubmit: 1643399777112 delta:1104
 2. SCHEMA request on Sovrin presubmit: 1643399777130 postsubmit: 1643399780139 delta:3009
 3. SCHEMA request on Sovrin presubmit: 1643399780142 postsubmit: 1643399783152 delta:3010
 4. SCHEMA request on Sovrin presubmit: 1643399783159 postsubmit: 1643399786192 delta:3033
 5. SCHEMA request on Sovrin presubmit: 1643399786198 postsubmit: 1643399789248 delta:3050
 6. SCHEMA request on Sovrin presubmit: 1643399789255 postsubmit: 1643399792270 delta:3015
 7. SCHEMA request on Sovrin presubmit: 1643399792280 postsubmit: 1643399795390 delta:3110
 8. SCHEMA request on Sovrin presubmit: 1643399795397 postsubmit: 1643399798355 delta:2958
 9. SCHEMA request on Sovrin presubmit: 1643399798361 postsubmit: 1643399801371 delta:3010
 10. SCHEMA request on Sovrin presubmit: 1643399801377 postsubmit: 1643399804343 delta:2966
 11. SCHEMA request on Sovrin presubmit: 1643399804349 postsubmit: 1643399807412 delta:3063
 12. SCHEMA request on Sovrin presubmit: 1643399807418 postsubmit: 1643399810442 delta:3024
 13. SCHEMA request on Sovrin presubmit: 1643399810447 postsubmit: 1643399813474 delta:3027
 14. SCHEMA request on Sovrin presubmit: 1643399813479 postsubmit: 1643399816518 delta:3039
 15. SCHEMA request on Sovrin presubmit: 1643399816524 postsubmit: 1643399819624 delta:3100
 16. SCHEMA request on Sovrin presubmit: 1643399819631 postsubmit: 1643399822576 delta:2945
 17. SCHEMA request on Sovrin presubmit: 1643399822582 postsubmit: 1643399825639 delta:3057
 18. SCHEMA request on Sovrin presubmit: 1643399825644 postsubmit: 1643399829018 delta:3374
 19. SCHEMA request on Sovrin presubmit: 1643399829024 postsubmit: 1643399831988 delta:2964
 20. SCHEMA request on Sovrin presubmit: 1643399831993 postsubmit: 1643399835024 delta:3031

Rejected CLAIM_DEF request

Time of all operations: 58144ms(58 sec) 10 created TCP conversations.
 Bytes agent to Sovrin 231k. Bytes Sovrin to agent 24k.

1. CLAIM_DEF request on Sovrin presubmit: 1643396246238 postsubmit: 1643396247297 delta:1059
 2. CLAIM_DEF request on Sovrin presubmit: 1643396247309 postsubmit: 1643396250303 delta:2994
 3. CLAIM_DEF request on Sovrin presubmit: 1643396250312 postsubmit: 1643396253323 delta:3011
 4. CLAIM_DEF request on Sovrin presubmit: 1643396253333 postsubmit: 1643396256335 delta:3002
 5. CLAIM_DEF request on Sovrin presubmit: 1643396256343 postsubmit: 1643396259345 delta:3002
 6. CLAIM_DEF request on Sovrin presubmit: 1643396259353 postsubmit: 1643396262370 delta:3017
 7. CLAIM_DEF request on Sovrin presubmit: 1643396262381 postsubmit: 1643396265366 delta:2985
 8. CLAIM_DEF request on Sovrin presubmit: 1643396265376 postsubmit: 1643396268378 delta:3002
 9. CLAIM_DEF request on Sovrin presubmit: 1643396268387 postsubmit: 1643396271416 delta:3029
 10. CLAIM_DEF request on Sovrin presubmit: 1643396271424 postsubmit: 1643396274463 delta:3039
 11. CLAIM_DEF request on Sovrin presubmit: 1643396274472 postsubmit: 1643396277430 delta:2958
 12. CLAIM_DEF request on Sovrin presubmit: 1643396277438 postsubmit: 1643396280504 delta:3066
 13. CLAIM_DEF request on Sovrin presubmit: 1643396280512 postsubmit: 1643396283467 delta:2955
 14. CLAIM_DEF request on Sovrin presubmit: 1643396283475 postsubmit: 1643396286473 delta:2998
 15. CLAIM_DEF request on Sovrin presubmit: 1643396286482 postsubmit: 1643396289486 delta:3004
 16. CLAIM_DEF request on Sovrin presubmit: 1643396289490 postsubmit: 1643396292588 delta:3098
 17. CLAIM_DEF request on Sovrin presubmit: 1643396292597 postsubmit: 1643396295510 delta:2913
 18. CLAIM_DEF request on Sovrin presubmit: 1643396295517 postsubmit: 1643396298520 delta:3003
 19. CLAIM_DEF request on Sovrin presubmit: 1643396298524 postsubmit: 1643396301522 delta:2998
 20. CLAIM_DEF request on Sovrin presubmit: 1643396301525 postsubmit: 1643396304536 delta:3011

Accepted CLAIM_DEF request

Time of all operations: 33668ms(33 sec) 11 created TCP conversations.
 Bytes agent to Sovrin 237k. Bytes Sovrin to agent 252k.

1. CLAIM_DEF request on Sovrin presubmit: 1643397089851 postsubmit: 1643397091171 delta:1320
 2. CLAIM_DEF request on Sovrin presubmit: 1643397094468 postsubmit: 1643397095532 delta:1064
 3. CLAIM_DEF request on Sovrin presubmit: 1643397096905 postsubmit: 1643397098493 delta:1588

```

4. CLAIM_DEF request on Sovrin presubmit: 1643397099147 postsubmit: 1643397101529 delta:2382
5. CLAIM_DEF request on Sovrin presubmit: 1643397103023 postsubmit: 1643397104578 delta:1555
6. CLAIM_DEF request on Sovrin presubmit: 1643397105318 postsubmit: 1643397107619 delta:2301
7. CLAIM_DEF request on Sovrin presubmit: 1643397109118 postsubmit: 1643397110638 delta:1520
8. CLAIM_DEF request on Sovrin presubmit: 1643397111987 postsubmit: 1643397113658 delta:1671
9. CLAIM_DEF request on Sovrin presubmit: 1643397117040 postsubmit: 1643397118364 delta:1324
10. CLAIM_DEF request on Sovrin presubmit: 1643397120228 postsubmit: 1643397121517 delta:1289
11. CLAIM_DEF request on Sovrin presubmit: 1643397125883 postsubmit: 1643397127240 delta:1357
12. CLAIM_DEF request on Sovrin presubmit: 1643397127984 postsubmit: 1643397129994 delta:2010
13. CLAIM_DEF request on Sovrin presubmit: 1643397132521 postsubmit: 1643397133497 delta:976
14. CLAIM_DEF request on Sovrin presubmit: 1643397134390 postsubmit: 1643397136517 delta:2127
15. CLAIM_DEF request on Sovrin presubmit: 1643397138660 postsubmit: 1643397142590 delta:3930
16. CLAIM_DEF request on Sovrin presubmit: 1643397144034 postsubmit: 1643397145589 delta:1555
17. CLAIM_DEF request on Sovrin presubmit: 1643397148641 postsubmit: 1643397149749 delta:1108
18. CLAIM_DEF request on Sovrin presubmit: 1643397151240 postsubmit: 1643397152746 delta:1506
19. CLAIM_DEF request on Sovrin presubmit: 1643397155800 postsubmit: 1643397156774 delta:974
20. CLAIM_DEF request on Sovrin presubmit: 1643397157697 postsubmit: 1643397159808 delta:2111

```

A.7 Local Indy pool transaction test operations

A.7.1 Rejected SCHEMA request

Time of all operations: 198ms(0.2 sec)

```

1. SCHEMA transaction local presubmit: 1643463477137 postsubmit: 1643463477155 delta:18
2. SCHEMA transaction local presubmit: 1643463477168 postsubmit: 1643463477182 delta:14
3. SCHEMA transaction local presubmit: 1643463477184 postsubmit: 1643463477197 delta:13
4. SCHEMA transaction local presubmit: 1643463477199 postsubmit: 1643463477213 delta:14
5. SCHEMA transaction local presubmit: 1643463477218 postsubmit: 1643463477221 delta:3
6. SCHEMA transaction local presubmit: 1643463477222 postsubmit: 1643463477235 delta:13
7. SCHEMA transaction local presubmit: 1643463477236 postsubmit: 1643463477246 delta:10
8. SCHEMA transaction local presubmit: 1643463477248 postsubmit: 1643463477255 delta:7
9. SCHEMA transaction local presubmit: 1643463477257 postsubmit: 1643463477262 delta:5
10. SCHEMA transaction local presubmit: 1643463477264 postsubmit: 1643463477271 delta:7
11. SCHEMA transaction local presubmit: 1643463477272 postsubmit: 1643463477285 delta:13
12. SCHEMA transaction local presubmit: 1643463477287 postsubmit: 1643463477290 delta:3
13. SCHEMA transaction local presubmit: 1643463477291 postsubmit: 1643463477301 delta:10
14. SCHEMA transaction local presubmit: 1643463477306 postsubmit: 1643463477319 delta:13
15. SCHEMA transaction local presubmit: 1643463477322 postsubmit: 1643463477338 delta:16
16. SCHEMA transaction local presubmit: 1643463477342 postsubmit: 1643463477355 delta:13
17. SCHEMA transaction local presubmit: 1643463477357 postsubmit: 1643463477366 delta:9
18. SCHEMA transaction local presubmit: 1643463477368 postsubmit: 1643463477371 delta:3
19. SCHEMA transaction local presubmit: 1643463477372 postsubmit: 1643463477382 delta:10
20. SCHEMA transaction local presubmit: 1643463477383 postsubmit: 1643463477387 delta:4

```

A.7.2 Accepted SCHEMA request

Time of all operations: 58257ms(59 sec)

```

1. SCHEMA transaction local presubmit: 1643464521939 postsubmit: 1643464522009 delta:70
2. SCHEMA transaction local presubmit: 1643464522020 postsubmit: 1643464525014 delta:2994
3. SCHEMA transaction local presubmit: 1643464525016 postsubmit: 1643464528037 delta:3021
4. SCHEMA transaction local presubmit: 1643464528039 postsubmit: 1643464531045 delta:3006
5. SCHEMA transaction local presubmit: 1643464531047 postsubmit: 1643464534050 delta:3003
6. SCHEMA transaction local presubmit: 1643464534052 postsubmit: 1643464537062 delta:3010
7. SCHEMA transaction local presubmit: 1643464537064 postsubmit: 1643464540089 delta:3025
8. SCHEMA transaction local presubmit: 1643464540090 postsubmit: 1643464543089 delta:2999
9. SCHEMA transaction local presubmit: 1643464543090 postsubmit: 1643464546124 delta:3034
10. SCHEMA transaction local presubmit: 1643464546126 postsubmit: 1643464549140 delta:3014
11. SCHEMA transaction local presubmit: 1643464549142 postsubmit: 1643464552161 delta:3019
12. SCHEMA transaction local presubmit: 1643464552163 postsubmit: 1643464556089 delta:3926
13. SCHEMA transaction local presubmit: 1643464556091 postsubmit: 1643464559108 delta:3017
14. SCHEMA transaction local presubmit: 1643464559112 postsubmit: 1643464562122 delta:3010
15. SCHEMA transaction local presubmit: 1643464562123 postsubmit: 1643464565125 delta:3002

```

```

16. SCHEMA transaction local presubmit: 1643464565127 postsubmit: 1643464568172 delta:3045
17. SCHEMA transaction local presubmit: 1643464568174 postsubmit: 1643464571167 delta:2993
18. SCHEMA transaction local presubmit: 1643464571169 postsubmit: 1643464574192 delta:3023
19. SCHEMA transaction local presubmit: 1643464574194 postsubmit: 1643464577222 delta:3028
20. SCHEMA transaction local presubmit: 1643464577224 postsubmit: 1643464580242 delta:3018

```

A.7.3 Rejected CLAIM_DEF request

Time of all operations: 18588ms(19 sec) 10 created TCP conversations.

Bytes agent to Sovrin 231k. Bytes Sovrin to agent 24k.

```

1. CLAIM_DEF transaction local presubmit: 1643465187439 postsubmit: 1643465187480 delta:41
2. CLAIM_DEF transaction local presubmit: 1643465187491 postsubmit: 1643465190487 delta:2996
3. CLAIM_DEF transaction local presubmit: 1643465190491 postsubmit: 1643465193509 delta:3018
4. CLAIM_DEF transaction local presubmit: 1643465194677 postsubmit: 1643465196507 delta:1830
5. CLAIM_DEF transaction local presubmit: 1643465197451 postsubmit: 1643465199525 delta:2074
6. CLAIM_DEF transaction local presubmit: 1643465200715 postsubmit: 1643465202536 delta:1821
7. CLAIM_DEF transaction local presubmit: 1643465203925 postsubmit: 1643465205542 delta:1617
8. CLAIM_DEF transaction local presubmit: 1643465207115 postsubmit: 1643465208547 delta:1432
9. CLAIM_DEF transaction local presubmit: 1643465210648 postsubmit: 1643465211557 delta:909
10. CLAIM_DEF transaction local presubmit: 1643465216962 postsubmit: 1643465217026 delta:64
11. CLAIM_DEF transaction local presubmit: 1643465217861 postsubmit: 1643465220028 delta:2167
12. CLAIM_DEF transaction local presubmit: 1643465224222 postsubmit: 1643465224270 delta:48
13. CLAIM_DEF transaction local presubmit: 1643465225818 postsubmit: 1643465227285 delta:1467
14. CLAIM_DEF transaction local presubmit: 1643465232599 postsubmit: 1643465232648 delta:49
15. CLAIM_DEF transaction local presubmit: 1643465234419 postsubmit: 1643465235661 delta:1242
16. CLAIM_DEF transaction local presubmit: 1643465237588 postsubmit: 1643465238668 delta:1080
17. CLAIM_DEF transaction local presubmit: 1643465242144 postsubmit: 1643465242195 delta:51
18. CLAIM_DEF transaction local presubmit: 1643465243749 postsubmit: 1643465245210 delta:1461
19. CLAIM_DEF transaction local presubmit: 1643465245672 postsubmit: 1643465248222 delta:2550
20. CLAIM_DEF transaction local presubmit: 1643465251400 postsubmit: 1643465251463 delta:63

```

A.7.4 Accepted CLAIM_DEF request

Time of all operations: 33668ms(33 sec) 11 created TCP conversations.

Bytes agent to Sovrin 237k. Bytes Sovrin to agent 252k.

```

1. CLAIM_DEF transaction local presubmit: 1643465849681 postsubmit: 1643465849772 delta:91
2. CLAIM_DEF transaction local presubmit: 1643465850831 postsubmit: 1643465852795 delta:1964
3. CLAIM_DEF transaction local presubmit: 1643465853517 postsubmit: 1643465855807 delta:2290
4. CLAIM_DEF transaction local presubmit: 1643465857882 postsubmit: 1643465858814 delta:932
5. CLAIM_DEF transaction local presubmit: 1643465861765 postsubmit: 1643465861849 delta:84
6. CLAIM_DEF transaction local presubmit: 1643465866306 postsubmit: 1643465866391 delta:85
7. CLAIM_DEF transaction local presubmit: 1643465870149 postsubmit: 1643465870242 delta:93
8. CLAIM_DEF transaction local presubmit: 1643465871061 postsubmit: 1643465873251 delta:2190
9. CLAIM_DEF transaction local presubmit: 1643465876714 postsubmit: 1643465876798 delta:84
10. CLAIM_DEF transaction local presubmit: 1643465880324 postsubmit: 1643465880408 delta:84
11. CLAIM_DEF transaction local presubmit: 1643465882704 postsubmit: 1643465883424 delta:720
12. CLAIM_DEF transaction local presubmit: 1643465885973 postsubmit: 1643465886438 delta:465
13. CLAIM_DEF transaction local presubmit: 1643465888936 postsubmit: 1643465889452 delta:516
14. CLAIM_DEF transaction local presubmit: 1643465890798 postsubmit: 1643465892459 delta:1661
15. CLAIM_DEF transaction local presubmit: 1643465896978 postsubmit: 1643465897064 delta:86
16. CLAIM_DEF transaction local presubmit: 1643465897614 postsubmit: 1643465900077 delta:2463
17. CLAIM_DEF transaction local presubmit: 1643465903857 postsubmit: 1643465903944 delta:87
18. CLAIM_DEF transaction local presubmit: 1643465904368 postsubmit: 1643465906963 delta:2595
19. CLAIM_DEF transaction local presubmit: 1643465915121 postsubmit: 1643465915210 delta:89
20. CLAIM_DEF transaction local presubmit: 1643465916205 postsubmit: 1643465918214 delta:2009

```

A.7.5 Read requests locally

This subsection presents the time of read request to the ledger. The Credential Schema has to read has two attributes. The Credential

Definition allows revocation(as said in subsection 3.5.3 has two keys).

```
Get Schema pre: 1643665411893 post: 1643665411899 delta: 6
Get Schema pre: 1643666574360 post:1643666574367 delta: 7
Get Schema pre: 1643667974584 post:1643667974591 delta: 7
Get Schema pre: 1643671856482 post:1643671856494 delta: 12
Get Schema pre: 1643732095866 post:1643732095876 delta: 10
Get Schema pre: 1643732095866 post:1643732095876 delta: 10
Get Schema pre: 1643735506172 post:1643735506185 delta: 13
Get Schema pre: 1643736019600 post:1643736019607 delta: 7
```

```
Get CredentialDefinition from ledger prerequest: 1643736033573postrequest : 1643736033596delta: 23
Get CredentialDefinition from ledger prerequest: 1643736033611postrequest : 1643736033629delta: 18
Get CredentialDefinition from ledger prerequest: 1643736033637postrequest : 1643736033644delta: 7
Get CredentialDefinition from ledger prerequest: 1643736033649postrequest : 1643736033661delta: 12
Get CredentialDefinition from ledger prerequest: 1643736033665postrequest : 1643736033681delta: 16
Get CredentialDefinition from ledger prerequest: 1643736033687postrequest : 1643736033706delta: 19
Get CredentialDefinition from ledger prerequest: 1643736033710postrequest : 1643736033727delta: 17
Get CredentialDefinition from ledger prerequest: 1643736033732postrequest : 1643736033745delta: 13
```

As said in 3.5.5, updates on the Revocation Registry are called deltas. To obtain the accumulator value of a Revocation Registry in a particular timeframe, a delta has to be read from ledger.

```
Read Revoc.Reg.Delta. pre: 1643665412024 post:1643665412038delta:14
Read Revoc.Reg.Delta. pre: 1643736033841 post:1643736033866delta:25
Read Revoc.Reg.Delta. pre: 1643737001233 post:1643737001257delta:24
Read Revoc.Reg.Delta. pre: 1643737001270 post:1643737001300delta:30
Read Revoc.Reg.Delta. pre: 1643737001303 post:1643737001341delta:38
Read Revoc.Reg.Delta. pre: 1643737001344 post:1643737001368delta:24
Read Revoc.Reg.Delta. pre: 1643737001370 post:1643737001396delta:26
Read Revoc.Reg.Delta. pre: 1643737001398 post:1643737001422delta:24
```

```
Revoc.Reg.Definition pre: 1643665412048 post:1643665412055delta:7
Revoc.Reg.Definition pre: 1643666592406 post:1643666592417delta:11
```

```
Revoc.Reg.Definition pre: 1643667608635 post:1643667608649delta:14
Revoc.Reg.Definition pre: 1643667992672 post:1643667992679delta:7
```

```
Revoc.Reg.Definition pre: 1643671876126 post:1643671876131delta:5
Revoc.Reg.Definition pre: 1643732119258 post:1643732119266delta:8
```

```
Revoc.Reg.Definition pre: 1643732119258 post:1643732119266delta:8
Revoc.Reg.Definition pre: 1643665411945 post:1643665411966delta:21
```

A.8 Local Indy Pool and MainNetCredential Definition update

CLAIM_DEF transaction submitted 20 times in a row.

A.8.1 Local Indy Pool

Time of all operations: 186ms(0.2sec)

```
1. CLAIM_DEF transaction local presubmit: 1643469866971 postsubmit: 1643469867052 delta:81
2. CLAIM_DEF transaction local presubmit: 1643469867062 postsubmit: 1643469867068 delta:6
3. CLAIM_DEF transaction local presubmit: 1643469867068 postsubmit: 1643469867073 delta:5
4. CLAIM_DEF transaction local presubmit: 1643469867073 postsubmit: 1643469867082 delta:9
5. CLAIM_DEF transaction local presubmit: 1643469867083 postsubmit: 1643469867094 delta:11
6. CLAIM_DEF transaction local presubmit: 1643469867095 postsubmit: 1643469867103 delta:8
7. CLAIM_DEF transaction local presubmit: 1643469867103 postsubmit: 1643469867114 delta:11
8. CLAIM_DEF transaction local presubmit: 1643469867114 postsubmit: 1643469867120 delta:6
9. CLAIM_DEF transaction local presubmit: 1643469867120 postsubmit: 1643469867124 delta:4
```

```

10. CLAIM_DEF transaction local presubmit: 1643469867125 postsubmit: 1643469867135 delta:10
11. CLAIM_DEF transaction local presubmit: 1643469867135 postsubmit: 1643469867138 delta:3
12. CLAIM_DEF transaction local presubmit: 1643469867139 postsubmit: 1643469867142 delta:3
13. CLAIM_DEF transaction local presubmit: 1643469867142 postsubmit: 1643469867145 delta:3
14. CLAIM_DEF transaction local presubmit: 1643469867145 postsubmit: 1643469867148 delta:3
15. CLAIM_DEF transaction local presubmit: 1643469867148 postsubmit: 1643469867150 delta:2
16. CLAIM_DEF transaction local presubmit: 1643469867150 postsubmit: 1643469867158 delta:8
17. CLAIM_DEF transaction local presubmit: 1643469867158 postsubmit: 1643469867164 delta:6
18. CLAIM_DEF transaction local presubmit: 1643469867165 postsubmit: 1643469867167 delta:2
19. CLAIM_DEF transaction local presubmit: 1643469867167 postsubmit: 1643469867170 delta:3
20. CLAIM_DEF transaction local presubmit: 1643469867170 postsubmit: 1643469867172 delta:2

```

A.8.2 Sovrin MainNet

Time of all operations: 10827ms(10sec)

```

1. CLAIM_DEF Definition on Sovrin presubmit: 1643469213704 postsubmit: 1643469214562 delta:858
2. CLAIM_DEF Definition on Sovrin presubmit: 1643469214575 postsubmit: 1643469214661 delta:86
3. CLAIM_DEF Definition on Sovrin presubmit: 1643469214663 postsubmit: 1643469215013 delta:350
4. CLAIM_DEF Definition on Sovrin presubmit: 1643469215014 postsubmit: 1643469215117 delta:103
5. CLAIM_DEF Definition on Sovrin presubmit: 1643469215118 postsubmit: 1643469215337 delta:219
6. CLAIM_DEF Definition on Sovrin presubmit: 1643469215338 postsubmit: 1643469215622 delta:284
7. CLAIM_DEF Definition on Sovrin presubmit: 1643469215623 postsubmit: 1643469215898 delta:275
8. CLAIM_DEF Definition on Sovrin presubmit: 1643469215898 postsubmit: 1643469217088 delta:1190
9. CLAIM_DEF Definition on Sovrin presubmit: 1643469217088 postsubmit: 1643469217510 delta:422
10. CLAIM_DEF Definition on Sovrin presubmit: 1643469217511 postsubmit: 1643469217986 delta:475
11. CLAIM_DEF Definition on Sovrin presubmit: 1643469217987 postsubmit: 1643469218992 delta:1005
12. CLAIM_DEF Definition on Sovrin presubmit: 1643469218993 postsubmit: 1643469219391 delta:398
13. CLAIM_DEF Definition on Sovrin presubmit: 1643469219391 postsubmit: 1643469219610 delta:219
14. CLAIM_DEF Definition on Sovrin presubmit: 1643469219610 postsubmit: 1643469220006 delta:396
15. CLAIM_DEF Definition on Sovrin presubmit: 1643469220007 postsubmit: 1643469220744 delta:737
16. CLAIM_DEF Definition on Sovrin presubmit: 1643469220744 postsubmit: 1643469221641 delta:897
17. CLAIM_DEF Definition on Sovrin presubmit: 1643469221641 postsubmit: 1643469222131 delta:490
18. CLAIM_DEF Definition on Sovrin presubmit: 1643469222132 postsubmit: 1643469223251 delta:1119
19. CLAIM_DEF Definition on Sovrin presubmit: 1643469223251 postsubmit: 1643469224432 delta:1181
20. CLAIM_DEF Definition on Sovrin presubmit: 1643469224433 postsubmit: 1643469224556 delta:123

```

A.9 Example of a NODE transaction and Get-ValidatorInfo request

A.9.1 NODE transaction

An example of NODE transaction (addition of a node to the pool) from the Sovrin BuilderNet Pool ledger.

```

1  {
2    "txn": {
3      "data": {
4        "data": {
5          "alias": "OgNode",
6          "blskey": "3bcg83S97wRd...",
7          "blskey_pop": "R4BwfzJJpe...",
8          "client_ip": "62.150.213.170",
9          "client_ip_geo": {
10           "city": "",
11           "country": "KW",
12           "eu": false,
13           "location": {
14             "lat": 29.3375,
15             "lon": 47.6581
16           },
17           "region": "",
18           "timezone": "Asia/Kuwait"

```

```

19     },
20     "client_port": 9702,
21     "node_ip": "62.150.213.170",
22     "node_ip_geo": {
23         "city": "",
24         "country": "KW",
25         "eu": false,
26         "location": {
27             "lat": 29.3375,
28             "lon": 47.6581
29         },
30         "region": "",
31         "timezone": "Asia/Kuwait"
32     },
33     "node_port": 9701,
34     "services": [
35         "VALIDATOR"
36     ]
37 },
38 "dest": "5aNBs6DTToRDNuXamiswvPhvoGxoLbdEL5XTLdZrv6Xf"
39 },
40 "metadata": {
41     "digest": "46cea093ccb0ab3296f7a453...",
42     "from": "3mvH5YWuYRV2AAE6V7Ch6",
43     "payloadDigest": "91eeecadfd878352b0a60724077b92...",
44     "reqId": 1568897081348430600
45 },
46 "protocolVersion": 2,
47 "type": "0",
48 "typeName": "NODE"
49 },
50 "txnMetadata": {
51     "seqNo": 23,
52     "txnTime": "2019-09-19T12:39:43.000Z"
53 }
54 }

```

A.9.2 GetValidatorInfo request

The following is the result of a validator info request. The result is from a node in a pool of four nodes.

```

1  {
2      "result":{
3          "identifier":"Th7MpTaRZVRynPiabds81Y",
4          "data":{
5              "Pool_info":{
6                  "Unreachable_nodes":[
7                      ],
8                  "Unreachable_nodes_count":0,
9                  "Total_nodes_count":4,
10                 "Quorums":{"f": 1, 'checkpoint': Quorum(2), 'ledger_status_last_3PC': Quorum(2), 'strong': Quorum(3), '
n': 4, 'prepare': Quorum(2), 'timestamp': Quorum(2), 'reply': Quorum(2), 'view_change': Quorum(3),
'bls_signatures': Quorum(3), 'propagate': Quorum(2), 'same_consistency_proof': Quorum(2), '
observer_data': Quorum(2), 'ledger_status': Quorum(2), 'view_change_ack': Quorum(2), '
backup_instance_faulty': Quorum(2), 'commit': Quorum(3), 'election': Quorum(3), 'consistency_proof
': Quorum(2), 'view_change_done': Quorum(3), 'weak': Quorum(2)}",
11                 "Read_only":false,
12                 "Reachable_nodes_count":4,
13                 "Blacklisted_nodes":[
14                     ],
15                 "Suspicious_nodes":"",
16                 "f_value":1,
17                 "Reachable_nodes":[
18                     [
19                         "Node1",
20                         null
21                     ],
22                     [
23                         "Node2",
24

```

```

25         0
26     ],
27     [
28         "Node3",
29         1
30     ],
31     [
32         "Node4",
33         null
34     ]
35 ],
36 },
37 "response-version": "0.0.1",
38 "Memory_profiler": [
39 ],
40 ],
41 "Extractions": {
42     "upgrade_log": "",
43     "journalctl_exceptions": [
44         ""
45     ],
46     "stops_stat": null,
47     "node-control status": [
48         ""
49     ],
50     "indy-node_status": [
51         ""
52     ]
53 },
54 "Update_time": "Wednesday, December 15, 2021 10:55:22 PM +0000",
55 "Software": {
56     "OS_version": "Linux-5.13.0-21-generic-x86_64-with-Ubuntu-16.04-xenial",
57     "Indy_packages": [
58         "ii indy-node 1.12.1~dev1172 amd64 Indy node",
59         "ii indy-plenum 1.12.1~dev989 amd64 Plenum Byzantine
60         Fault Tolerant Protocol",
61         "ii libindy-crypto 0.4.5 amd64 This is the
62         shared crypto library for Hyperledger Indy components.",
63         "ii python3-indy-crypto 0.4.5 amd64 This is the
64         official wrapper for Hyperledger Indy Crypto library (https://www.hyperledger.org/projects).",
65         ""
66     ],
67     "sovrin": "unknown",
68     "Installed_packages": [
69         "timeout-decorator 0.4.0",
70         "sortedcontainers 1.5.7",
71         "six 1.11.0",
72         "sha3 0.2.1",
73         "setuptools 45.2.0",
74         "semver 2.7.9",
75         "rlp 0.5.1",
76         "pyzmq 18.1.0",
77         "python-rocksdb 0.6.9",
78         "python-dateutil 2.6.1",
79         "Pympler 0.5",
80         "Pygments 2.2.0",
81         "psutil 5.4.3",
82         "portalocker 0.5.7",
83         "pip 9.0.3",
84         "packaging 19.0",
85         "orderedset 2.0",
86         "libnacl 1.6.1",
87         "jsonpickle 0.9.6",
88         "ioflo 1.5.4",
89         "intervaltree 2.1.0",
90         "indy-plenum 1.12.1.dev989",
91         "indy-node 1.12.1.dev1172",
92         "indy-crypto 0.4.5-23",
93         "distro 1.3.0",
94         "base58 1.0.0",
95         "wheel 0.29.0",
96         "wcwidth 0.1.6",
97         "ujson 1.33",
98         "pyparsing 2.0.3",
99         "PyNaCl 1.0.1",
100        "pycparser 2.14",
101        "prompt-toolkit 0.57",

```

```

99         "ply 3.7",
100         "msgpack-python 0.4.6",
101         "leveldb 0.1",
102         "cffi 1.5.2"
103     ],
104     "indy-node": "1.12.1.dev1172"
105 },
106 "Hardware": {
107     "HDD_used_by_node": "25 MBs"
108 },
109 "Node_info": {
110     "Client_protocol": "tcp",
111     "Replicas_status": {
112         "Node3:1": {
113             "Last_ordered_3PC": [
114                 1,
115                 3348
116             ],
117             "Stashed_txns": {
118                 "Stashed_PrePrepare": 0,
119                 "Stashed_checkpoints": 0
120             },
121             "Primary": "Node3:1",
122             "Watermarks": "3340:3640"
123         },
124         "Node3:0": {
125             "Last_ordered_3PC": [
126                 1,
127                 10384
128             ],
129             "Stashed_txns": {
130                 "Stashed_PrePrepare": 0,
131                 "Stashed_checkpoints": 0
132             },
133             "Primary": "Node2:0",
134             "Watermarks": "10300:10600"
135         }
136     },
137     "BLS_key": "3WFpdbg7C5cnLYZw...",
138     "Committed_ledger_root_hashes": {
139         "0": "b'HFvSQsKyt4juFN1t54SpQFRNfU4HLVxSeLr8EMCNage6'",
140         "1": "b'2F9z1kHDzxe1rv3XtytSMVWCERJgneWmpGPXwaYdUkV'",
141         "2": "b'GKot5hBsd81kMupNCXHagbhv3huEbxAFMLnpcX2hniwn'",
142         "3": "b'5ZwF4qbohEG3HmBi1Ep9cdEj6wP6XVbnnF3NFGx8ner4'"
143     },
144     "Requests_timeouts": {
145         "Ordering_phase_req_timeouts": 0,
146         "Propagates_phase_req_timeouts": 0
147     },
148     "Catchup_status": {
149         "Last_txn_3PC_keys": {
150             "0": {
151                 "Node1": [
152                     null,
153                     null
154                 ],
155                 "Node4": [
156                     null,
157                     null
158                 ]
159             },
160             "1": {
161                 },
162             "2": {
163                 },
164             "3": {
165                 "Node1": [
166                     null,
167                     null
168                 ],
169                 "Node4": [
170                     null,
171                     null
172                 ]
173             }
174         }
175     }

```

```

176         },
177         "Number_txns_in_catchup":{
178             "0":0,
179             "1":0,
180             "2":0,
181             "3":0
182         },
183         "Waiting_consistency_proof_msgs":{
184             "0":null,
185             "1":null,
186             "2":null,
187             "3":null
188         },
189         "Received_LedgerStatus": "",
190         "Ledger_statuses":{
191             "0":"synced",
192             "1":"synced",
193             "2":"synced",
194             "3":"synced"
195         }
196     },
197     "Node_protocol":"tcp",
198     "Node_ip":"0.0.0.0",
199     "Mode":"participating",
200     "Node_port":9705,
201     "Freshness_status":{
202         "0":{
203             "Has_write_consensus":true,
204             "Last_updated_time":"2021-12-15 22:54:58+00:00"
205         },
206         "1":{
207             "Has_write_consensus":true,
208             "Last_updated_time":"2021-12-15 22:55:22+00:00"
209         },
210         "2":{
211             "Has_write_consensus":true,
212             "Last_updated_time":"2021-12-15 22:54:58+00:00"
213         }
214     },
215     "Uncommitted_ledger_txns":{
216         "0":{
217             "Count":0
218         },
219         "1":{
220             "Count":0
221         },
222         "2":{
223             "Count":0
224         },
225         "3":{
226             "Count":0
227         }
228     },
229     "Client_ip":"0.0.0.0",
230     "Name":"Node3",
231     "Uncommitted_ledger_root_hashes":{
232     },
233     },
234     "Committed_state_root_hashes":{
235         "0":"b'F9iUf4RuMmUzj7o4zsNGYeyjztvPbUJ1S967kCzzkVvf'",
236         "1":"b'Agpz1u2MpES7oxF8njbt97qpN8Bz4RGvR77Hi9TJCwD1'",
237         "2":"b'DfNLmH4DAHTKv63YPFJzuRdeEtVwF5RtVnvKYHd8iLEA'"
238     },
239     "Count_of_replicas":2,
240     "Metrics":{
241         "instances_started":{
242             "0":9972.932849743,
243             "1":9972.93820727
244         },
245         "master throughput ratio":1.0000000393,
246         "avg backup throughput":0.0029512196,
247         "Omega":20,
248         "total requests":8,
249         "average-per-second":{
250             "write-transactions":4.993907E-4,
251             "read-transactions":0.9811154801
252         }
253     },

```

```

253         "uptime":16019,
254         "ordered request counts":{
255             "0":8,
256             "1":8
257         },
258         "transaction-count":{
259             "ledger":3106,
260             "audit":10384,
261             "pool":4,
262             "config":0
263         },
264         "master throughput":0.0029512197,
265         "Delta":0.1,
266         "max master request latencies":0,
267         "client avg request latencies":{
268             "0":null,
269             "1":null
270         },
271         "ordered request durations":{
272             "0":11.44397719,
273             "1":4.268943218
274         },
275         "throughput":{
276             "0":0.0029512197,
277             "1":0.0029512196
278         },
279         "Lambda":240
280     },
281     "View_change_status":{
282         "View_No":1,
283         "Last_complete_view_no":1,
284         "VCDone_queue":{
285
286         },
287         "VC_in_progress":false,
288         "IC_queue":{
289             "2":{
290                 "Voters":{
291                     "Node4":{
292                         "reason":25
293                     }
294                 }
295             }
296         },
297         "Last_view_change_started_at":"1970-01-01 00:00:00"
298     },
299     "Client_port":9706,
300     "verkey":"5LCf9oRXkDBNG61PpSsBnWDLhyCZWzYyHajSwusoSLZCsAre9a5BGL",
301     "Uncommitted_state_root_hashes":{
302         "0":"'b'F9iUf4RuMmUZj7o4zsNGYeyjztvPbUJ1S967kCzzkVvf'",
303         "1":"'b'Agpz1u2MpES7oxF8nJbt97qpN8Bz4RGvR77Hi9TJCwD1'",
304         "2":"'b'DfNLmH4DAHTKv63YPFJzuRdeEtVwF5RtVnvKYHd8iLEA'"
305     },
306     "did":"'DKVxG2fXXTU8yT5N7hGEbXB3dfdAnYv1JczDUHpmDxya'"
307 },
308 "Protocol":{
309
310 },
311 "timestamp":1639608922
312 },
313 "type":"119",
314 "reqId":1639608922578596764
315 },
316 "op":"REPLY"
317 }

```

Bibliography

- [1] Dashlane. *Home Chef One of 11 New Data Breaches Resulting in Millions of User Records for Sale on the Dark Web*. URL: <https://blog.dashlane.com/home-chef-data-breach-millions-of-user-records-for-sale-on-the-dark-web/> (visited on 02/05/2022).
- [2] Eurostat. *E-commerce statistics for individuals*. URL: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=E-commerce_statistics_for_individuals (visited on 02/05/2022).
- [3] Dashlane. *500,000 Zoom Accounts on the Dark Web*. URL: <https://blog.dashlane.com/500000-zoom-accounts-for-sale-on-the-dark-web/> (visited on 02/05/2022).
- [4] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014. ISBN: 9781466570269. URL: <https://www.crcpress.com/Introduction-to-Modern-Cryptography-Second-Edition/Katz-Lindell/p/book/9781466570269>.
- [5] Andreas M Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies*. Second ed. "O'Reilly Media, Inc.", 2017.
- [6] Alin Tomescu. *What is a merkle tree*. URL: <https://decentralizedthoughts.github.io/2020-12-22-what-is-a-merkle-tree/> (visited on 01/03/2022).
- [7] Cristopher Allen. *The Path to Self-Sovereign Identity*. URL: <http://www.lifewithalacrity.com/2016/04/the-path-to-self-sovereign-identity.html> (visited on 12/31/2021).

- [8] Alexander Mühle et al. "A survey on essential components of a self-sovereign identity". In: *Comput. Sci. Rev.* 30 (2018), pp. 80–86. DOI: [10.1016/j.cosrev.2018.10.002](https://doi.org/10.1016/j.cosrev.2018.10.002). URL: <https://doi.org/10.1016/j.cosrev.2018.10.002>.
- [9] Zooko Wilcox-O’Hearn. *Names: Decentralized, Secure, Human-Meaningful: Choose Two*. 2001. URL: <http://web.archive.org/web/20120204172516/http://zooko.com/distnames.html> (visited on 12/31/2021).
- [10] Harry A. Kalodner et al. "An Empirical Study of Namecoin and Lessons for Decentralized Namespace Design". In: *14th Annual Workshop on the Economics of Information Security, WEIS 2015, Delft, The Netherlands, 22-23 June, 2015*. 2015. URL: http://www.econinfosec.org/archive/weis2015/papers/WEIS_2015_kalodner.pdf.
- [11] David Chadwick Manu Sporny Dave Longley. *Verifiable Credentials Data Model v1.1*. URL: <https://www.w3.org/TR/vc-data-model/> (visited on 12/28/2021).
- [12] ZKProof. *ZKProof Community Reference. Version 0.2*. Dec. 2019. Updated versions at <https://zkproof.org>.
- [13] Tracy Kuhrt Stephen Curran. *Introduction to Hyperledger Indy*. URL: <https://github.com/hyperledger-archives/education/blob/master/LFS171x/docs/introduction-to-hyperledger-indy.md#hyperledger-indy-in-context> (visited on 02/01/2022).
- [14] Everynym. *Everynym about page*. URL: <https://www.evernym.com/about-evernym/> (visited on 01/02/2022).
- [15] Sovrin. *Sovrin FAQ*. URL: <https://sovrin.org/faqs/> (visited on 01/02/2022).
- [16] Hyperledger. *indy-node*. URL: <https://github.com/hyperledger/indy-node/> (visited on 01/02/2022).
- [17] Hyperledger. *indy-sdk*. URL: <https://github.com/hyperledger/indy-sdk/> (visited on 01/02/2021).
- [18] Hyperledger. *Transactions*. URL: <https://hyperledger-indy.readthedocs.io/projects/node/en/latest/transactions.html> (visited on 01/02/2022).

- [19] Hyperledger. *Audit Ledger*. URL: https://hyperledger-indy.readthedocs.io/projects/plenum/en/latest/audit_ledger.html (visited on 01/04/2022).
- [20] msgpack.org. *MsgPack specification*. URL: <https://github.com/msgpack/msgpack/blob/master/spec.md> (visited on 01/04/2022).
- [21] Hyperledger. *indy-plenum*. URL: <https://github.com/hyperledger/indy-plenum> (visited on 02/02/2022).
- [22] Svetlin Nakov. *Elliptic Curve Cryptography (ECC)*. URL: <https://cryptobook.nakov.com/asymmetric-key-ciphers/elliptic-curve-cryptography-ecc#curve25519-x25519-and-ed25519> (visited on 01/08/2022).
- [23] Hyperledger. *start_nodes.md*. URL: https://github.com/hyperledger/indy-plenum/blob/f812a72197bc0740d4c6b75b796beff8f88c73f9/docs/source/start_nodes.md (visited on 01/04/2022).
- [24] Hyperledger. *Hyperledger Architecture, Volume 1*. URL: https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf (visited on 01/04/2022).
- [25] Miguel Castro and Barbara Liskov. "Practical byzantine fault tolerance and proactive recovery". In: *ACM Trans. Comput. Syst.* 20.4 (2002), pp. 398–461. DOI: [10.1145/571637.571640](https://doi.org/10.1145/571637.571640). URL: <https://doi.org/10.1145/571637.571640>.
- [26] Evernym. *Understanding Hyperledger Indy Ledger*. URL: <https://wiki.hyperledger.org> (visited on 01/04/2022).
- [27] Hyperledger. *Indy Node troubleshooting guide*. URL: <https://github.com/hyperledger/indy-node/blob/master/docs/source/troubleshooting.md> (visited on 02/02/2022).
- [28] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. "RBFT: Redundant Byzantine Fault Tolerance". In: *2013 IEEE 33rd International Conference on Distributed Computing Systems*. 2013, pp. 297–306. DOI: [10.1109/ICDCS.2013.53](https://doi.org/10.1109/ICDCS.2013.53).
- [29] Daniel Hardman. *Three Dimensions of Identity*. URL: <https://medium.com/evernym/three-dimensions-of-identity-bc06ae4aec1c> (visited on 01/05/2022).

- [30] Daniel Hardman. *Aries RFC 0004: Agents*. URL: <https://github.com/hyperledger/aries-rfcs/blob/main/concepts/0004-agents/README.md> (visited on 01/06/2022).
- [31] Markus Sabadello Manu Sporny Dave Longley et al. *Aries RFC 0004: Agents*. URL: <https://www.w3.org/TR/did-core/> (visited on 01/14/2022).
- [32] Adrian Gropper Kim Hamilton-Duffy Ryan Grant. *Use Cases and Requirements for Decentralized Identifiers*. URL: <https://www.w3.org/TR/did-use-cases/> (visited on 01/14/2022).
- [33] Manu Sporny Orie Steele. *DID Specification Registries*. URL: <https://www.w3.org/TR/did-spec-registries/> (visited on 01/21/2022).
- [34] Dmitri Zagidulin Markus Sabadello. *DID resolution*. URL: <https://w3c-ccg.github.io/did-resolution/> (visited on 01/21/2022).
- [35] Hyperledger. *Fully-Qualified identifiers*. URL: <https://github.com/hyperledger/indy-sdk/blob/master/docs/how-tos/fully-qualified-did-support.md> (visited on 01/21/2022).
- [36] *Indy DID Method*. URL: <https://hyperledger.github.io/indy-did-method/> (visited on 01/14/2022).
- [37] Patrik Stas. *Indyscan*. URL: <https://github.com/Patrik-Stas/indyscan> (visited on 01/14/2022).
- [38] Jason Law Drummond Reed, Daniel Hardman, and Mike Lodder. *DKMS (Decentralized Key Management System) Design and Architecture V3*. URL: <https://github.com/hyperledger/indy-sdk/blob/master/docs/design/005-dkms/DKMS%5C%20Design%5C%20and%5C%20Architecture%5C%20V3.md#81-key-rotation> (visited on 01/14/2022).
- [39] *Sovrin DID Method Specification*. URL: <https://sovrin-foundation.github.io/sovrin/spec/did-method-spec-template.html> (visited on 01/14/2022).
- [40] Sovrin. *Sovrin Glossary V3*. URL: https://docs.google.com/document/d/1gfIz5TT0cNp2kxGMLFXr19x1uoZsruUe_0glHst2fZ8/edit#heading=h.pvtmpaxhni76 (visited on 01/14/2022).
- [41] Hyperledger. *Default AUTH_MAP Rules*. URL: https://github.com/hyperledger/indy-node/blob/master/docs/source/auth_rules.md (visited on 01/16/2022).

- [42] Andrew Tobin (Evernym). *Sovrin: What Goes on the Ledger?* URL: <https://www.evernym.com/wp-content/uploads/2017/07/What-Goes-On-The-Ledger.pdf> (visited on 01/15/2022).
- [43] Hyperledger. *Indy SDK for Node.js*. URL: <https://github.com/hyperledger/indy-sdk/blob/master/wrappers/java/src/main/java/org/hyperledger/indy/sdk/crypto/Crypto.java> (visited on 01/16/2022).
- [44] Daniel Hardman. *Aries RFC 0006: SSI Notation*. URL: <https://github.com/hyperledger/aries-rfcs/blob/master/concepts/0006-ssi-notation/README.md> (visited on 02/09/2022).
- [45] Stephen Curran Kyle Den Hartog and Mike Lodder Sam Curren. *Aries RFC 0019: Encryption Envelope*. URL: <https://github.com/hyperledger/aries-rfcs/blob/main/features/0019-encryption-envelope/README.md> (visited on 01/15/2022).
- [46] Sovrin. *Repudiation*. URL: <https://github.com/sovrin-foundation/protocol/blob/master/janus/repudiation.md> (visited on 01/16/2022).
- [47] Hyperledger. *Anoncreds Design (indy-node)*. URL: <https://github.com/hyperledger/indy-node/blob/master/design/anoncreds.md> (visited on 01/16/2022).
- [48] Hyperledger. *002-anoncreds*. URL: <https://github.com/hyperledger/indy-sdk/tree/master/docs/design/002-anoncreds> (visited on 01/16/2022).
- [49] Hyperledger. *How Credential Revocation Works*. URL: <https://github.com/hyperledger/indy-sdk/blob/master/docs/concepts/revocation/cred-revocation.md> (visited on 01/16/2022).
- [50] Hyperledger. *Anoncreds.java*. URL: <https://github.com/hyperledger/indy-sdk/blob/master/wrappers/java/src/main/java/org/hyperledger/indy/sdk/anoncreds/Anoncreds.java>.
- [51] Michael Lodder Dmitry Khovratovich. *Anonymous credentials with type-3 revocation*. URL: <https://github.com/hyperledger/ursa-docs/blob/main/specs/anoncreds1/anoncreds.tex> (visited on 01/17/2022).

- [52] Michael Lodder Dmitry Khovratovich. *Anonymous credentials with type-3 revocation*. URL: <https://github.com/hyperledger-archives/indy-crypto/blob/master/libindy-crypto/docs/AnonCred.pdf> (visited on 01/17/2022).
- [53] Hyperledger. *Wallet Query Language*. URL: <https://github.com/hyperledger/indy-sdk/blob/master/docs/design/011-wallet-query-language/README.md> (visited on 01/16/2022).
- [54] H.J.M. Joosten H.J. Jongsma. *Technical Report Studybits*. URL: https://www.scienceguide.nl/wp-content/uploads/2018/03/studybits_-_technical_report_en_v2.pdf (visited on 01/14/2022).
- [55] ErasmusWithoutPaper. *About*. URL: <https://www.erasmuswithoutpaper.eu/about> (visited on 01/18/2022).
- [56] Drummond Reed Gary Zimmerman. *Becoming a Sovrin Steward*. URL: <https://www.evernym.com/wp-content/uploads/2017/07/Becoming-a-Sovrin-Steward.pdf> (visited on 01/16/2022).
- [57] Robbie Whelan. *Parcel-Delivery Firms and Retailers Experiment With Package Pickup Points*. URL: <https://www.wsj.com/articles/parcel-delivery-firms-and-retailers-experiment-with-package-pickup-points-1431966679> (visited on 01/31/2022).
- [58] Hyperledger. *PoolUtils.java*. URL: <https://github.com/hyperledger/indy-sdk/blob/master/samples/java/src/main/java/Utils/PoolUtils.java>.
- [59] Hyperledger. *Ledger.java*. URL: <https://github.com/hyperledger/indy-sdk/blob/master/wrappers/java/src/main/java/org/hyperledger/indy/sdk/ledger/Ledger.java#L465> (visited on 01/24/2022).
- [60] Sovrin. *SSI Metrics Dashboards*. URL: <https://sovrin.org/ssi-metrics-dashboards/> (visited on 02/07/2022).
- [61] Hyperledger. *Extension of validator_info*. URL: https://github.com/hyperledger/indy-node/blob/master/design/validator_info.md#cur-description (visited on 02/02/2022).

- [62] Hyperledger. *Indy Node troubleshooting guide Useful fields in validator info output*. URL: <https://github.com/hyperledger/indy-node/blob/master/docs/source/troubleshooting.md#useful-fields-in-validator-info-output> (visited on 02/02/2022).
- [63] Hyperledger. *Threading model*. URL: <https://github.com/hyperledger/indy-sdk/blob/master/docs/architecture/threading-model.md> (visited on 02/02/2022).