Performance Evaluation of Self-Sovereign Identity Use Cases

Alexandre Siqueira
Univ. Federal de São Paulo
Brazil
alexandre.siqueira@unifesp.br

Arlindo F. da Conceição
Univ. Federal de São Paulo
Brazil
arlindo.conceicao@unifesp.br

Vladimir Rocha
Univ. Federal do ABC
Brazil
vladimir.rocha@ufabc.edu.br

***Abstract*—Self-sovereign identity (SSI) enables the creation of user-centric applications where the user has complete control over his data. This research evaluates the performance of SSI-based applications; to do this, we implemented healthcare use cases using Hyperledger Indy and Hyperledger Aries frameworks, deployed it in a cloud environment, and executed their empirical evaluation. The results indicate that the bottleneck is the CPU; a simple setup can support up to 80 concurrent users without relevant errors and 120 simultaneous users before system degradation. Finally, we discuss the system bottlenecks and possible optimization techniques.**

***Index Terms*—SSI, performance evaluation, healthcare use cases, Hyperledger Indy, Hyperledger Aries**

I. Introduction

The development of modern information systems and applications involves searching for better performance, more security, and decentralization. It is not easy to conciliate these factors. One promising approach for addressing these challenges is the usage of self-sovereign identities (SSIs) [1], [2]. The SSI enables users to control their identity and data, sharing specifics about their credentials with whomever they want while securing their private information in their digital wallets. SSI-based systems usually have three components:

- Decentralized Identities (DIDs) [3]: DID is a W3C standard that adopts public key infrastructures and digital signatures to represent digital identities.
- Verifiable Credentials (VCs) [4]: It is also a W3C specification that provides tamper-proof evidence of claims about a subject.
- Blockchain [5]: Using blockchain in SSI solutions is not mandatory, but it is desirable because it can be used as a public key infrastructure aggregating trust to the solution [6]. Blockchain facilitates to verification of the VCs.

An SSI-based application has, in general, the following characteristics: *i*) the DIDs identify users or subjects, *ii*) VCs and VCs' presentations demonstrate data ownership, and *iii*) the blockchain keeps the information secure and auditable.

SSI can be applied in several areas [7], but it has particular potential to contribute to healthcare and e-Health scenarios where the tension between data sharing and privacy is evident. For example, developing a privacy-preserving health data registry presents challenges of secure persistence and access control management [8]. The SSI technology empowers users and patients to control their identity, sharing only the specific information they want while securing their private information in their digital wallets. For example, suppose that a diagnostic lab issues blood test results for an individual (a DID) as a verifiable credential (VC). The credential can comprise clinical information, such as blood type, glucose level, and cholesterol levels. The patient can offer a digital presentation only containing this specific information when asked for blood type.

SSI technology can even offer information without providing any data. To illustrate this idea, let us imagine an application asking a patient if she can donate blood for a person with blood type A+. A digital presentation can answer just "*yes*" or "*no*" to this question without showing the patient's blood type [9].

Thus, SSI technology allows the construction of a new generation of user-centered, information-centered, and privacy-oriented applications. The user defines which information is shared and who can see that information. Notice that the client side of the system provides this guarantee, not the server side. In their study, [10] defined several use cases of SSI in the healthcare context, but it lacked the assurance that those scenarios could work in the real world. In addition, there were no standard metrics for planning/measuring SSI systems and applications deployment. How many blockchain nodes are necessary? Is the process memory or CPU intensive? How many simultaneous clients can the servers support? This paper aims to reduce this lack of knowledge.

The objective was to understand the overall behavior of SSI applications. This challenge means finding latency lower bounds (time per API request) and throughput upper bounds (number of requests). The literature does not provide this information to the best of our knowledge.

We proposed and deployed a fully decentralized, privacy-preserving, blockchain-based SSI system to assess the critical performance factors. Our solution uses open-source frameworks (Hyperledger Indy / Aries) and runs on a cloud environment (Microsoft Azure). [10] implemented and evaluated several healthcare use cases, but this work focuses on the non-functional aspects.

The remaining of the article is structured as follows: Section II presents SSI's fundamental concepts and overall architecture. Section III describes the problem statement. Section IV defines the methodology. Section V describes the

test environment. Section VI exemplifies SSI-based healthcare use cases. Section VII discuss the obtained results. Finally, Section VIII presents our final considerations and points out future works.

II. Fundamentals of Self-sovereign identities

To understand how the SSI works, defining some concepts and principles is essential.

A. Entities and decentralized identifiers (DIDs)

An entity or subject is either a person or an organization that performs a role in a situation. In the context of digital identities, the entities can perform the following roles [4], depicted in Figure 1:

- **Issuer:** an entity that asserts claims about a subject, issuing a VC and transmitting it to a holder.
- **Holder:** an entity that possesses VCs and uses them to create a verifiable presentation to prove a claim;
- **Verifier:** an entity that validates the claims made by a holder by validating the VCs presented by the holder.

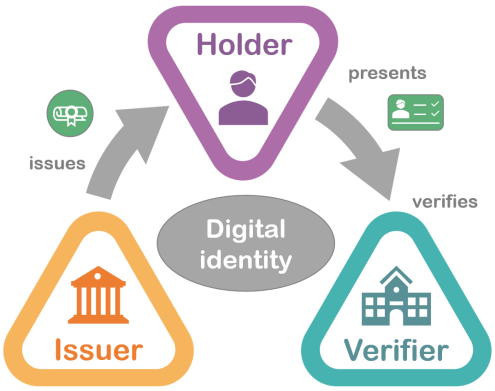

Fig. 1. Entities in a digital identity context, as described by W3C [4]

Entities need identities to represent themselves in a digital context. A digital identity is a set of attributes that allow an entity to be uniquely distinguished from others within a context. Entities may have several digital identities, one for each context, forming what is described by López [11] as their digital persona. The W3C created standards [3] to specify these identities and named them Decentralized Identifiers (DIDs) [12].

B. Claims, verifiable credentials (VCs), and presentations

The W3C standard [4] defines claims as assertions made by a subject. A VC may contain several claims. When an entity issues a credential to someone, it states that the claims in the credential are valid.

A VC may declare [4], for example, the identity of the holder (name, ID number), the issuer attributes (city government, national agency, university), the type of the credential (driver license, drug prescription), attributes asserted by the issuer (date of birth, bachelor's degree, the patient condition), constraints (expiration date, terms of use), etc.

VCs use cryptographic functions to provide tamper-proof evidence of their authorship. For example, a university could issue a college transcript to a student as a VC containing this student's grades and current degree and sign it with the private key associated with its public DID. This college transcript would also refer to the student's public DID, allowing anyone to verify its authenticity.

C. Architecture of SSI-based systems

There is a range of platforms for developing SSI systems [13]. This research used Hyperledger SSI open-source projects because of their maturity and flexibility. Our architecture comprises several software components distributed in distinct layers, as depicted in Figure 2. These components are detailed in the following sections.

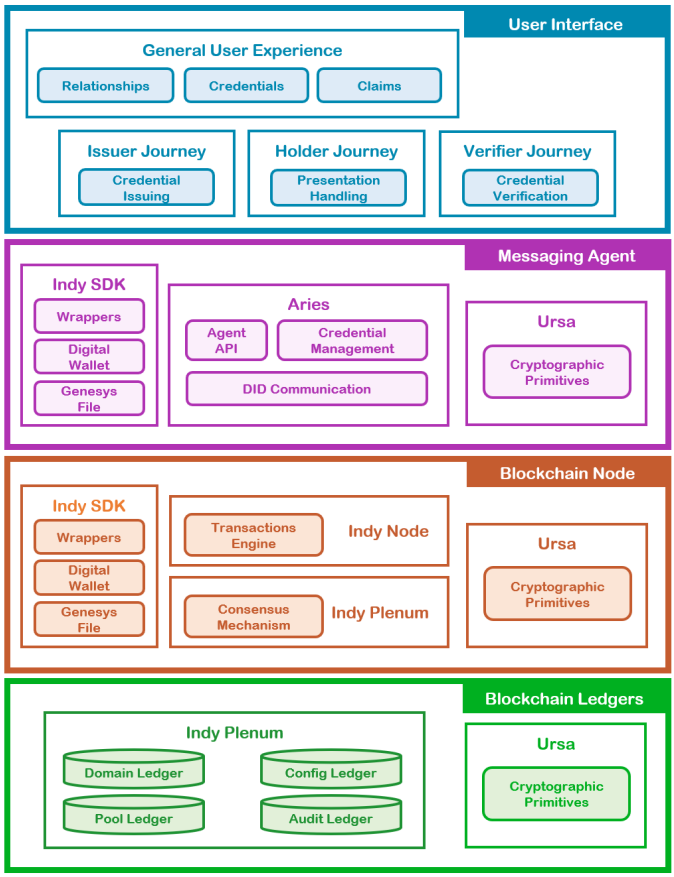

Fig. 2. Component diagram - architectural layers and software packages

1) Hyperledger Indy: This is at the system's core, implementing the fit-for-purpose blockchain network for DID. The Blockchain Ledgers layer uses **Indy Plenum** module to store identities and credentials public information in distributed ledgers.

Indy Plenum is also responsible for implementing the consensus mechanisms that maintain data synchronized in the Blockchain Node layer. The nodes in this layer run **Indy Node** that handles node communication and read/write transactions

in the ledgers and **Indy SDK** libraries for digital wallet management and ledger operation APIs.

Indy Node and Indy SDK modules implement the W3C DID specifications [3], essential for the decentralized identity management capabilities they aim to provide.

2) Hyperledger Aries: This project offers credential management capabilities to the system, implementing W3C VCs specifications [4] and peer-to-peer communication among entities through DIDComm protocols. DIDComm [14] is a standard for secure and private communication between agents (entities in an SSI context), allowing them to connect, maintain relationships, issue credentials, provide proof, and others.

An **Aries agent** is a software component that runs in the Messaging Agent Layer as integration middleware, providing user applications with APIs to engage in SSI interactions with other agents. These APIs are designed to be infrastructure agnostic, allowing Aries agents to integrate with any VC provider. For this research, the selected Aries agents embed Indy SDK libraries to connect to the provided Indy blockchain.

3) Hyperledger Ursa: This project implements functions for cryptographic primitives in a shared library that is used across Hyperledger projects, namely Indy and Aries, to avoid duplicating cryptographic work.

Since many SSI use cases deal with cryptographic key management, **Ursa libraries** spread across Blockchain Ledgers, Blockchain Node, and Messaging Agent architectural layers.

4) General user experience and user journeys: The User interface layer is an abstraction for the healthcare software that uses Messaging Agent APIs to provide specific business journeys for issuers, holders, and verifiers:

- Software applications for issuers must implement **credential issuing** features crafted for their specific needs. e.g., a clinical laboratory must handle medical screenings results as credentials to be issued to a patient's digital wallet;
- Digital wallet software for holders must handle **credentials presentation** so that patients can present their medical information as proof for verifiers;
- Software applications for verifiers must provide **credential verification** capabilities fit for their purpose; for instance, airlines should check traveler's health status while physicians should look up patient's EHR.

Aside from specific user journey features, all applications must implement generic features common for all entities, such as Relationship, Credentials, and Claims management.

D. Use cases

We have chosen health use cases to validate the user-centric SSI solution because of the dilemma of privacy vs. data sharing. Siqueira [10] implemented six representative use cases of SSI in healthcare: report of allergies, clinical notes, immunization control, laboratory results, drug prescription, and medical procedures. We have chosen immunization control and drug prescriptions from this list of use cases to evaluate because they include revocation steps; more detail about the use case is provided in Section VI.

III. Problem Statement

Dealing with health information as a claim of someone's digital identity enables us to leverage concepts of SSI to create decentralized electronic health record systems that give patients complete control over their health information. However, the bulky nature of health data could expose bottlenecks that impact the overall capacity of SSI-based health registry systems.

This study aims to determine these bottlenecks and provide an empirical baseline for assessing blockchain-based SSI applications and accurately sizing their infrastructure components for real-world deployments.

IV. Methodology

The methodology consisted of deploying a basic architecture and an SSI-based application. The components were deployed in a cloud environment and gradually stressed until any sign of degradation appeared. During this process, we used the tool JMeter [15] to record the execution results, such as CPU usage, number of API requests, and response time (in milliseconds).

The application implements two healthcare scenarios comprising several interactions among the holder, issuer, and verifier. The test environment is realistic, except all its components share the same virtual network. The objective of running them in the same network was to reduce the impact of network latency in the experiments.

The workload employed multiple threads emulating simultaneous users. The number of users (threads) is incremented for each iteration by 20. Each thread executes one entire application cycle per time, where each cycle contains a sequence of API requests. Between each API request was inserted a delay of 2 seconds to avoid network flooding.

V. Test environment

The test environment was designed as a minimal setup that could perform as a production environment. It is imperative to notice that in an actual setup, the user interaction with the wallets implies additional time between system requests. So, our setup can offer a lower bound to a maximum number of concurrent users.

For simplicity, all server components were set up on the Microsoft Azure [16] cloud environment using virtualization techniques for machine provisioning. All machines were deployed in Azure's East US region because this location offered more features at the lowest cost. This setup is represented in the deployment diagram, depicted in Figure 3, explained in the following sections.

A. Network

Hyperledger Indy nodes establish inter-node connections for blockchain ledger synchronization that require a private network to secure. We created the *indylab-vnet* virtual private network (VPN) for that. It was created to isolate Indy inter-node communication and establish a governance control for adding new nodes to the network.

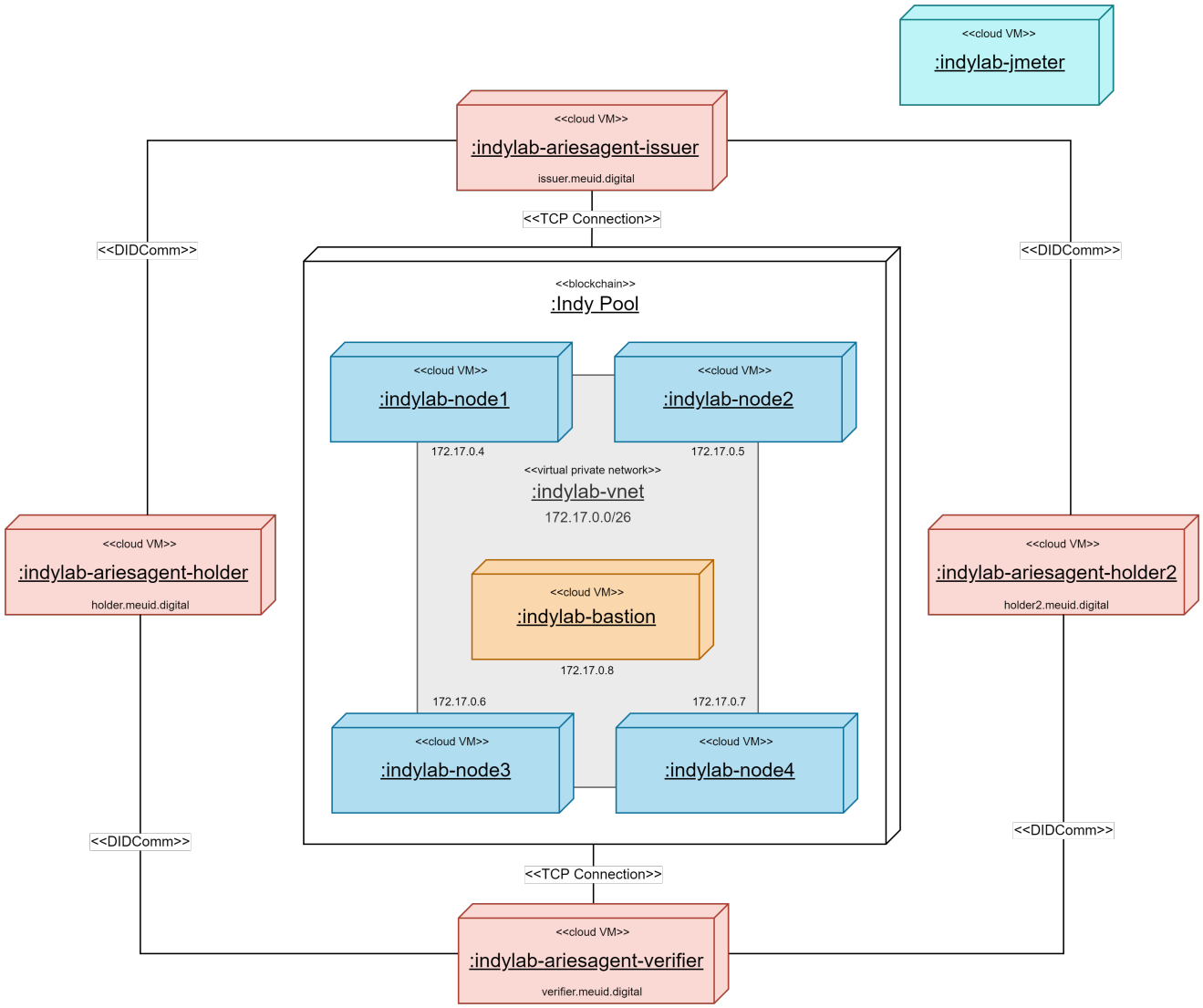

Fig. 3. Deployment diagram - execution environment

This simplified topology, portrayed in Figure 3, overlooks complexities that must be addressed in real-world implementations. For example, network nodes located in distant geographic regions or hosted on different cloud providers would have to connect to the Indy node private network via site-to-site VPNs or another type of network gateway mechanism.

B. Blockchain nodes

The Indy Pool comprises four virtual machines that act as blockchain network nodes. Those nodes run Hyperledger Indy Node code on a Ubuntu 16.04 Linux server with two vCPU and 4GB of RAM, connecting via a secure private network. Additional nodes can be added to the network by an administrator if needed.

In addition to the VPN, the blockchain nodes also have public IP addresses that enable Indy client connections. Those IPs are registered in the genesis file, a configuration artifact that acts as the network identifier, distributed among network participants, allowing them to connect to the pool.

C. Messaging agents

There are many data exchange scenarios where this architecture can be applied, and for every scenario, issuers, holders, and verifiers will perform specific business operations. This situation requires a flexible messaging agent setup, so distinct user interface components can run decoupled from their agents. Hyperledger Aries Cloud Agent provides this flexibility by delivering an integration API layer that user applications can consume to interact with the blockchain network and other messaging agents. Aries agents also manage user wallets, using Indy SDK local wallets or PostgreSQL databases to store user credentials.

Aries Cloud Agents can be deployed as containers in a Docker host or Python runners on Ubuntu Linux servers. To keep the test environment simple and to observe the agent's runtime behavior more directly, all Aries Cloud Agents used in the tests were deployed as Python runners. Aries agents run on a Ubuntu 18.04 Linux server with two vCPU and 4GB of RAM.

For testing purposes, the environment comprises four messaging agent servers:

- One **issuer server** that can run different instances of Hyperledger Aries Cloud Agent, according to the test that is currently running.
- One **verifier server** that can run different instances of Hyperledger Aries Cloud Agent, according to the test that is currently running.
- Two **holder servers**, each runs two instances of Hyperledger Aries Cloud Agent.

It is important to note that each holder server runs two Aries instances (4 total), while issuer and verifier servers run one instance each. This contrast was introduced during the tests to adjust its balance, given the excessive load on the holder servers due to the concentration of computational tasks on these machines: e.g., in a typical issuer-holder-verifier transaction, the holder orchestrates the interaction, performing twice as many functions as the other entities. Since real-world scenarios are supposed to have holder entities spread on patients' mobile devices, their impact on the overall performance of the environment would be negligible.

Table I details the specification of Aries agent machines.

D. User interface

Most features of the proposed software are focused on the patient experience. Therefore, the patient wallet is the component that deserves the most thought. Fortunately, some applications provide out-of-the-box SSI digital ledger functionality for testing purposes. During the proof of concept, this study selected the Trinsic [17] wallet for its simplicity of user interface and ability to import external genesis files pointing to custom Hyperledger Indy networks. For the performance/scalability tests, however, the artifacts were replaced by JMeter test runners that emulate the user behavior by consistently consuming the agent APIs in the same sequence as a regular user would do. Details regarding these runners are described in the next section.

E. Test runners

This research adopted Apache JMeter [15] to automate most of its tests. JMeter allows the execution of elaborate test plans as concurrent runners, simulating the typical user load on a system. Test plans specify the step sequence, the number of concurrent runner threads, the pause between steps, and the assertions that must be checked for successful runs.

During the execution of a test plan, JMeter collects runtime data from the running threads, such as elapsed time, iteration status, failed steps, throughput, among others.

Separate test plans were devised to simulate distinct health data exchange scenarios. We evaluated the following aspects:

- average response time
- scalability capabilities
- behavior during stress
- bottleneck entities
- possible impacts in user experience

F. Evaluation criteria

This experimental plan presents evaluation criteria composed of questions whose answers must be registered on a data collection form for that particular test iteration. Those criteria are described in Table II.

The test results provide information that supports the assessment, either confirming the architecture's soundness or improving the experimental plan for subsequent iterations.

VI. Health care use cases

The test scripts sought to represent common healthcare scenarios that SSI technologies could improve. These use case scenarios share specific capabilities that could be tested to assess the fitness of the proposed architecture. For example: if a health use case depends on a particular capability (e.g., expired claims) to be considered functional, and the tests for the named capability fail, then the proposed model does not fit that use case. The following list presents the healthcare scenarios selected for the tests, briefly describing the improvements this architecture could contribute.

- **Immunization records:** Despite previous attempts to create an international standard for proving immunization status, there is no easy way to validate its legitimacy when reported by a person. The success of many efforts to bring society back to normal after the COVID-19 pandemic depends on reliable proof of vaccination, the so-called COVID passports. International Air Travel Association, for example, is adopting SSI technologies to provide a travel pass app [18] that allows travelers to present verifiable proof of negative test results and vaccine shots taken without disclosing personal information. Figure 4 illustrates the model.
- **Control methods for prescription drugs:** prescription drug regulation can benefit from the SSI approach by leveraging Blockchain features such as immutability and decentralized trust to combat counterfeit and identity fraud. In addition, selective disclosure allows patients to prove they are entitled to purchase prescription medicine without exposing their personal data. Figure 5 shows the model.

Figure 6 shows an example of the usual use case sequence of actions among Holder, Issuer, and Verifier, with the numbers associated with the actions/interactions mentioned in Table III.

VII. Results and discussion

The following sections present and discuss the test results focusing on the overall performance, scalability capabilities, and revocation behavior.

TABLE I
CLOUD AGENT MACHINE SPECIFICATIONS

Features	Issuer setup	Verifier setup	Holders setup
Host name	indylab-ariesagent-issuer	indylab-ariesagent-verifier	indylab-ariesagent-holder indylab-ariesagent-holder2
OS	Ubuntu Server 18.04	Ubuntu Server 18.04	Ubuntu Server 18.04
Aries instances	Single instance	Single instance	Two instances each
Size	Standard B2s	Standard B2s	Standard B2s
vCPU	2	2	2
RAM	4GB	4GB	4GB

TABLE II
EVALUATION CRITERIA - DIMENSIONS

Dimension	Questions	Test plan
Scalability	Does the proposed architecture scale? Is it affected by latency? How does it affect storage?	Test iterations with incremental load allow measuring the system's working behavior and extrapolating its supported capacity. The ability to scale horizontally and vertically is verifiable by adding more resources to the test environment and testing its effectiveness with further test runs and increased load. Overall latency impact is measured by comparing the average response time while increasing the test load. The same goes for storage impact, measured by checking the size of the blockchain ledger and the entities' local Indy wallets.
Performance	How many API requests/second can it handle? How many simultaneous users can it support?	The number of requests per second and simultaneous users are variables set at the beginning of test runs that influence the overall system performance. The overall system performance is measured by incrementing the number of simultaneous users for every test iteration and monitoring its conditions. Any indications of performance degradation, such as request errors, increased response time, or inconsistent working behavior, are signals of a stressed environment and capacity limit.
Extensibility	Does the system provide Application Programming Interfaces (APIs) for integration with external systems? Does the system support customization mechanisms such as plugins or add-ons that enable functionality modification or improvement?	JMeter test scripts leverage Aries integration APIs to simulate client software interactions with the system, supporting issuer, holder, and verifier client features. Desired features that are not supported out of the box could be achieved by extending Aries's default functionalities. This capability should be observed during test iterations.

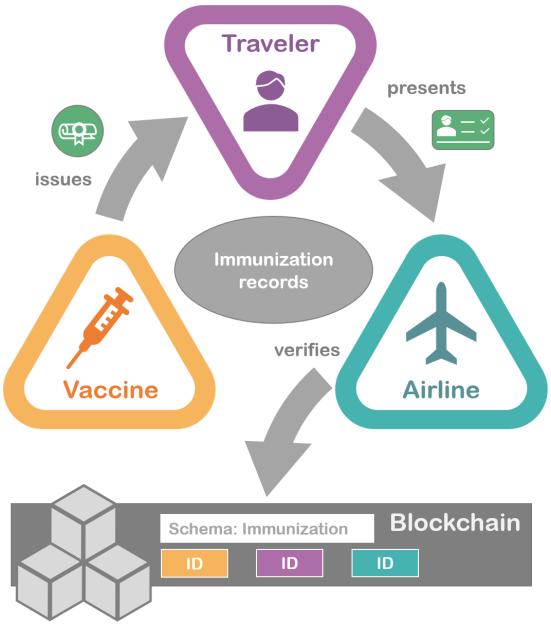

Fig. 4. Immunization records

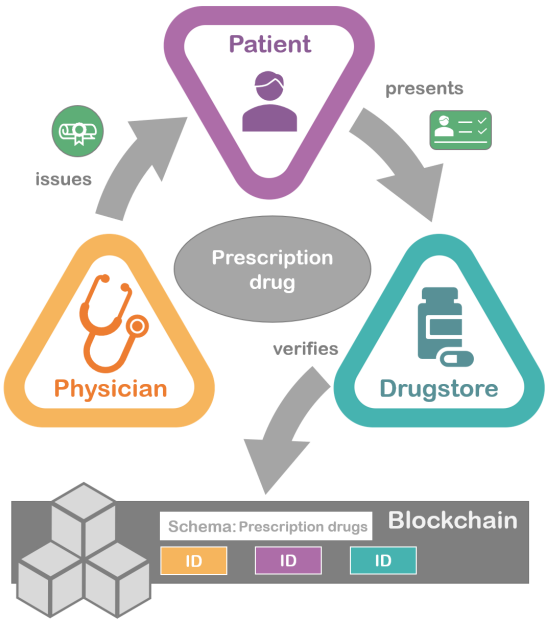

Fig. 5. Prescription drugs

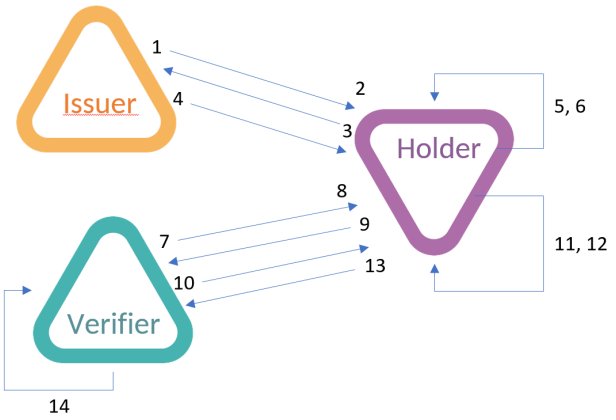

Fig. 6. Usual sequence of actions among Holder, Issuer, and Verifier.

A. Overall behavior

The first set of tests was focused on testing the system's overall function. This approach established a baseline by selecting the most comprehensive health record scenario and covering it end-to-end, beginning with a small load of simultaneous users and incrementing the load every round. The immunization records scenario was chosen for this first set of tests and configured as a sequence of API calls and result validations in a JMeter script.

Along with the number of simultaneous users, other parameters were specified as part of this baseline: the time interval between API calls was set to 2 seconds with a deviation of 500ms, assuming this average delay acceptable in a real-world interaction. Also, for this baseline, all agents used Indy SDK wallets that are, by default, stored in the local file system.

The following metrics were observed to examine the system condition:

- number of running API requests/second
- average response time
- number of errors during steps
- CPU usage
- memory usage

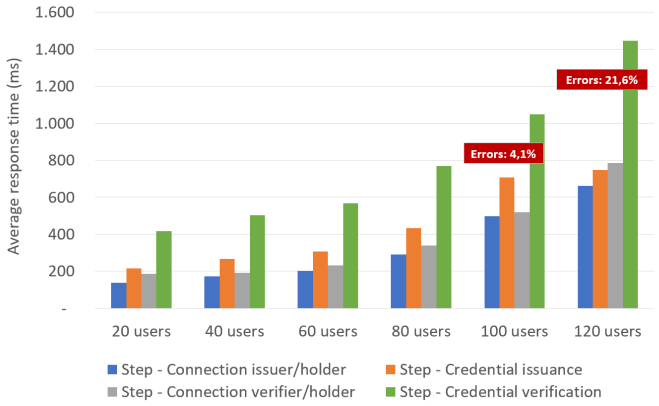

Fig. 7. Immunization scenario with single Indy SDK local wallet

Results of this first round of tests, illustrated in Figure 7, corroborate that the system average response time degrades linearly as the user load increases, along with the Aries agent's CPU usage. They also indicate that user load has a negligible impact on Indy blockchain's computing resources, as shown in Table III, suggesting that Aries agents rely primarily on the information they exchange with peer agents via DIDComm. Indy's public ledger seems to be used by agents only to fetch new schemas and public DIDs for unknown agents in their first interaction.

The test assertions added to the JMeter confirmed the expected results of the functional immunization records tests, demonstrating the system's fitness to the claim proof/disclosure and record expiration functional capabilities.

The system architecture remained consistently stable as the user load increased, reaching up to 80 simultaneous user threads - supporting 14.5 TPS of throughput - running the immunization records test script with no errors. However, with 100 simultaneous user threads, the system started degrading and responding to Aries agent's API requests with errors.

Even though the average response time remained below 2 seconds during the stressful condition, the failed requests were the ones where elapsed time took longer than 2 seconds, implying that the time interval between API requests, as configured in the JMeter script, was not enough to wait for the previous message to complete before triggering the subsequent one. As a result, most failed requests responded with errors similar to the following:

```
POST /issue-credential-2.0/send
HTTP/1.1 403 Forbidden
Connection XYZ not ready
```

The analysis of the server under high load conditions showed that, while Aries' server CPU usage was below 50%, the Python process that ran the Aries agent took 100% of its core CPU (Figure 8), indicating that Aries single-threaded architecture had reached its processing limit.

Table III shows the result details for the first round of tests. Memory consumption metrics were omitted since user load did not affect Indy or Aries servers in this regard. The numbers between parentheses are the maximum and minimum response time in milliseconds.

B. Scalability behavior

The second set of tests focused on optimizing the test environment and assessing the architecture's scalability capabilities. From the settings adopted for the first round, some adjustments were made:

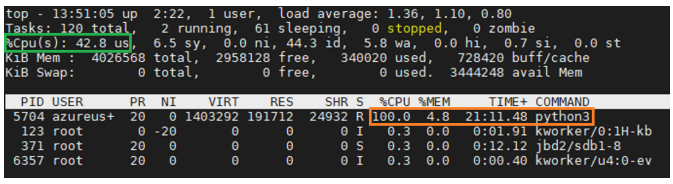

Fig. 8. Aries agent CPU usage under stress

TABLE III
NUMBER OF THREADS *versus* RESPONSE TIME: AVERAGE (MIN–MAX) MS - RAW DATA AND SCRIPTS AVAILABLE [19]

API Request	Number of simultaneous threads					
	20 threads	**40 threads**	**60 threads**	**80 threads**	**100 threads**	**120 threads**
# of scenario executions	200	684	752	1040	1113	992
Step - Connection issuer/holder	**139 ms**	**173**	**202**	**292**	**498**	**662**
01 - Issuer invitation	52 (37-222)	65 (34-457)	86 (35-546)	137 (35-867)	329 (19-1333)	528 (39-1674)
02 - Holder receives invitation	29 (21-133)	37 (20-427)	39 (39-446)	51 (20-746)	56 (19-747)	43 (19-444)
03 - Holder acceptance	58 (38-170)	71 (40-631)	76 (38-715)	104 (41-810)	113 (40-1097)	90 (40-588)
Step - Credential issuance	**215**	**267**	**307**	**433**	**707**	**748**
04 - Issuer issues credential	52 (32-243)	65 (33-561)	88 (35-590)	152 (39-893)	407 (44-1949)	520 (32-2331)
05 - Holder queries credential	26 (18-176)	31 (17-547)	39 (17-744)	53 (17-658)	56 (4-850)	30 (3-751)
06 - Holder stores credential	137 (102-474)	171 (102-830)	179 (101-893)	228 (100-1347)	243 (103-1379)	197 (102-1103)
Step - Connection verifier/holder	**187**	**190**	**231**	**340**	**519**	**786**
07 - Verifier invitation	105 (38-940)	84 (36-656)	113 (35-938)	181 (36-1234)	346 (35-3239)	652 (38-2504)
08 - Holder receives invitation	29 (20-127)	35 (19-362)	43 (19-409)	52 (19-509)	54 (20-896)	43 (19-408)
09 - Holder acceptance	53 (41-137)	71 (39-664)	75 (40-497)	106 (38-1138)	118 (41-1304)	91 (39-1564)
Step - Credential verification	**417**	**503**	**569**	**770**	**1.048**	**1.447**
10 - Verifier requests proof	42 (19-425)	52 (18-502)	72 (19-692)	107 (20-1015)	207 (19-2446)	296 (19-2375)
11 - Holder queries requests	19 (8-111)	43 (9-446)	49 (9-401)	75 (9-640)	79 (9-835)	42 (9-510)
12 - Holder queries eligible credentials	20 (11-101)	43 (13-762)	51 (13-534)	78 (12-749)	86 (13-961)	43 (12-718)
13 - Holder sends proof	163 (128-309)	171 (126-714)	174 (129-690)	209 (130-892)	224 (128-1302)	200 (130-939)
14 - Verifier validates proof	174 (141-334)	196 (139-731)	222 (138-879)	301 (145-1389)	452 (145-3146)	866 (142-2367)
Throughput (average number of completed requests/second)						
Value	**3.8**	**7.5**	**11.1**	**14.5**	**17.6**	**20.3**
CPU Usage (average %)						
Indy servers	**11.9%**	**12.7%**	**12.5%**	**12.7%**	**12.7%**	**12.6%**
Aries server - issuer	**9.1%**	**13.9%**	**18.2%**	**28.5%**	**36.5%**	**35.3%**
Aries server - verifier	**6.6%**	**11.9%**	**17.1%**	**25.7%**	**35.4%**	**35.6%**
Errors (average %)						
Failed requests	**0%**	**0%**	**0%**	**0%**	**4.1%**	**21.6%**

- The JMeter script running configuration was modified to skip further API requests of a given user thread if a prior request failed. This setting provided more reliable results since some error requests could be considered in the metrics, even knowing that their effect would be inevitably flawed given the errors in the previously required API requests of the chain;
- To counteract the Aries single-threaded architecture limitation, the issuer and verifier servers were adjusted to run two Aries agent instances sharing the same wallet. An NGINX [20] reverse proxy was added to the server to balance the API requests between Aries local instances;
- Aries agent's digital wallet mechanism was switched from the default Indy SDK local storage to a local PostgreSQL database instance. This change was necessary to support concurrent access to a single digital wallet, a feature currently not supported by the Indy SQLite database implementation [21];

The test results showed that working with concurrent Aries agents locally on issuer and verifier servers allowed them to handle CPU usage more efficiently in the two vCPU configurations and, consequently, support more simultaneous users with the same hardware. With this configuration, the environment could sustain 120 simultaneous user threads (23.3 requests/second of throughput) without errors, peaking at 73% of CPU usage.

An unexpected event occurred during this test round, causing verifier agents to crash after some testing time. The causes for the Aries agent to halt could not be determined, but the agent logs showed similarities with a known issue identified by the Lissi Identity team [22].

C. Revocation behavior

The third set of tests was meant to validate the revocation capabilities of Indy and Aries components. The environment followed the same configurations used in the last iteration, apart from the following changes:

- A different health scenario was selected to evaluate distinct functional capabilities. The prescription drugs scenario was represented in a different JMeter test script.
- A new server component had to be added to the environment: a Tails [23] revocation server, used by Aries agents to check for revoked credentials. This component was installed in the Indy bastion server.

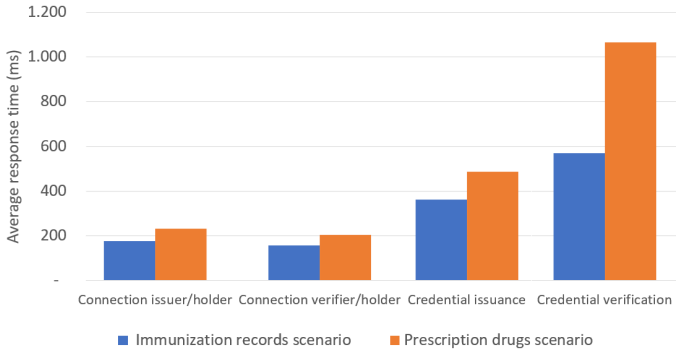

Fig. 9. Performance impact in credential revocation scenario with 60 simultaneous users

The test results shown in Figure 9 confirmed that handling revocation records raise the complexity of credential issuance and validation processes. This behavior could be observed in the overall response time of the Aries API requests for this scenario compared with the results of previous scenarios.

The prescription drugs scenario presented some implementation challenges that out-of-the-box SSI interaction could not solve. For example, Indy credentials do not support single-use capabilities, which could be used to implement a medical prescription that would be consumed upon presentation at a drugstore. Also, standard DID methods do not allow Indy credentials to be revoked by entities other than those that issued them. This prevents the alternative where the drugstore could revoke the drug prescription after verifying it.

The last test consisted of shutting down the Indy nodes, one by one, and verifying the behavior. As expected, Indy nodes could not ensure consensus if one of the four nodes was missing. The error message observed in the Aries error logs was:

```
POST /present-proof-2.0/records/../send-presentation
HTTP/1.1 400 Bad Request
Exception raised by ledger transaction:
Error: Pool timeout.
Caused by: Consensus is impossible.
```

After shutting down the other Indy nodes, Aries agents could not operate and logged the following error message:

```
Shutdown in progress.
```

VIII. Conclusions

Self-sovereign identities technology allows the creation of decentralized applications where the data owner controls his data (credentials). During the COVID-19 pandemic, we saw the world's first relevant example of this application. And we believe that more use cases will emerge in the following years.

This work innovates by evaluating personal health features as credentials, and mapping intricate healthcare entity relations in simple issuer-holder-verifier interactions. We also identified practical limitations of the SSI model. The developers can use these results to project future systems and applications.

Our experiments offer a baseline for achievable performance. It says that up to 80 concurrent users can be supported even in a simple setup. After 80 simultaneous users, some level of performance degradation is expected. Our results have methodological limitations; the most critical limitation is that there are additional latencies between system components in actual setups due to human interaction and network latency. We expected that user interaction time would increase the number of concurrent users. On the other hand, network latency can create new problems, such as timeouts and connection loss. It is worth mentioning that the performance data depends on the setup, but our results can be an orientation.

In future work, we will improve the evaluation method by inserting adaptive backoff time intervals between API requests to get a more precise result in the maximum number of concurrent users. Another alternative is to implement a pool of requests, making it easier to control the number of requests. Another open question that needs to be investigated is precisely why the process is CPU bound in the Aries agent; in the future, we want to refine this finding.

Stressed conditions tend to impact Aries agents much more than Indy blockchain nodes. The metrics that should be monitored to detect system degradation are: increased average API response time, CPU usage, and failed API requests.

Since Aries agents' processes are CPU-bound, adding more agents to an Aries cluster (combining several instances and load balancer) efficiently scales the environment horizontally. Adding more CPU resources to Aries agents is also possible but conceivably limited due to I/O and memory bottlenecks that could happen despite appearing negligible in our tests. Another possible extension of this work is to improve the scalability. One possibility is using several instances of Aries' agents and a load balancer before the agents.

Acknowledgment

The work was partially supported by SmartMed Project, Research Council of Norway, project number: 288106.

This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq proc. 465446/2014-0, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, FAPESP proc. 14/50937-1, and FAPESP proc. 15/24485-9.

References

[1] A. Preukschat and D. Reed, *Self-Sovereign Identity*. Manning Publications, 2021, kindle edition.

[2] R. Soltani, U. T. Nguyen, and A. An, "A survey of self-sovereign identity ecosystem," *Security and Communication Networks*, vol. 2021, 2021.

[3] W3C, "Decentralized Identifiers (DIDs) v1.0 - Core architecture, data model, and representations," 2021. [Online]. Available: https://www.w3.org/TR/did-core/

[4] ——, "Verifiable Credentials Data Model 1.0 - Expressing verifiable information on the Web," 2019. [Online]. Available: https://www.w3.org/TR/vc-data-model

[5] ComputerWorld, "How blockchain makes self-sovereign identities possible," 2018. [Online]. Available: https://www.computerworld.com/article/3244128/how-blockchain-makes-self-sovereign-identities-possible.html

[6] K. Werbach, *The blockchain and the new architecture of trust*. Mit Press, 2018.

[7] M. S. Ferdous, F. Chowdhury, and M. O. Alassafi, "In search of self-sovereign identity leveraging blockchain technology," *IEEE Access*, vol. 7, pp. 103 059–103 079, 2019.

[8] E. J. De Aguiar, B. S. Faiçal, B. Krishnamachari, and J. Ueyama, "A survey of blockchain-based strategies for healthcare," *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, pp. 1–27, 2020.

[9] B. A. Alzahrani, "Self-protected content for information-centric networking architectures using verifiable credentials," *Telecommunication Systems*, pp. 1–10, 2022.

[10] A. Siqueira, A. F. da Conceição, and V. Rocha, "User-centric health data using self-sovereign identities," *IV Workshop em Blockchain: Teoria, Tecnologias e Aplicações (WBlockchain)*, 2021. [Online]. Available: https://doi.org/10.5753/wblockchain.2021.17135

[11] M. López, "Self-sovereign identity: The future of identity: Self-sovereignity, digital wallets, and blockchain," LACChain, 2020.

[12] P. Windley, "Blog: Decentralized identifiers," 2019. [Online]. Available: https://www.windley.com/archives/2019/02/decentralized_identifiers.shtml

[13] A. Satybaldy, M. Nowostawski, and J. Ellingsen, "Self-sovereign identity systems: Evaluation framework," in *IFIP International Summer School on Privacy and Identity Management*. Springer, 2019, pp. 447–461.

[14] Hyperledger, "Aries RFC 0005: DID Communication," 2019. [Online]. Available: https://github.com/hyperledger/aries-rfcs/tree/master/concepts/0005-didcomm

[15] A. S. Foundation, "Apache JMeter Website," 2022. [Online]. Available: https://jmeter.apache.org

[16] Microsoft, "Microsoft Azure Website," 2022. [Online]. Available: https://portal.azure.com

[17] Trinsic, "Identity wallets," 2021. [Online]. Available: https://trinsic.id/identity-wallets/

[18] IATA, "IATA - travel pass initiative," 2021. [Online]. Available: https://www.iata.org/en/programs/passenger/travel-pass/

[19] Github, "Aries Performance Test - Raw data and test scripts," 2022. [Online]. Available: https://github.com/alex-siqueira/aries-performance-test/tree/main/test-scripts

[20] NGINX, "Nginx website," 2022. [Online]. Available: https://nginx.org

[21] Hyperledger, "Credential definition not in wallet - issue #1480 - github repository," 2021. [Online]. Available: https://github.com/hyperledger/aries-cloudagent-python/issues/1480

[22] L. Identity, "System crashes after "record not found" exception - acapy load test results - github repository," 2022. [Online]. Available: https://github.com/lissi-id/acapy-load-test-results

[23] P. of British Columbia, "Indy Tails Server - Github Repository," 2022. [Online]. Available: https://github.com/bcgov/indy-tails-server