End-study project report

Tanguy Benard - Yann Cantais

January 16, 2023

Contents

1 Introduction

This end study project aims at understanding and building a decentralized and secure message app.

A decentralized messaging application based on Hyperledger Aries and Hyperledger Indy is an innovative communication tool that uses blockchain technology to ensure privacy and security of exchanges. Unlike traditional messaging applications that are hosted on centralized servers, a decentralized messaging application runs on a network of peers that share data transparently and securely.

Hyperledger Aries is an open source platform that provides the tools needed to build decentralized messaging applications and other types of blockchain-based applications. It also offers key management and proof-of-possession capabilities, as well as identity management and data sharing tools. Hyperledger Indy, meanwhile, is a blockchain platform dedicated to decentralized identity management (DID). It allows users to create and manage their own digital identities securely and autonomously, without having to trust a third party.

By combining the features of Hyperledger Aries and Hyperledger Indy, a decentralized messaging application can provide a secure and confidential communication solution that respects users' privacy. Users can send messages and files confidentially and securely, while ensuring that their digital identities remain under their control and cannot be used without their knowledge. This makes it an ideal tool for organizations concerned about privacy and security of their communications.

In sum, a decentralized messaging application based on Hyperledger Aries and Hyperledger Indy is a powerful tool for secure and confidential communication that offers a secure alternative to traditional messaging applications. It allows users to manage their own digital identities autonomously and communicate securely and confidentially, without having to trust a third party.

2 Architecture of the solution

In order for this project to succeed, it is mandatory to give user's control on their identity. The main goal is to be able to communicate with others without having to register through a phone number or a social media account for example. The idea is to keep the messages secured and private.

2.1 DIDs

To achieve this, user's must get an identity wallet and store DIDs in them. Decentralized Identifiers (DIDs) are a new type of identifier for entities that allows them to control their own digital identity. Unlike traditional identifiers, which are controlled by centralized organizations like governments or corporations, DIDs are created and managed by the entities themselves using decentralized technologies.

A DID is a unique identifier that is stored on a decentralized network, such as a blockchain. The controller of a DID is able to prove control over it without requiring permission from any other party. It consists of two parts: a DID Document and a DID Method. The DID Document is a JSON document that contains information about the entity, such as its public key, and a list of services that the entity can be contacted through. The DID Method is a set of rules and protocols that define how the DID is created, managed, and resolved on the decentralized network.

DIDs are a relatively new technology and are still being developed and standardized by the W3C organization. They are being used in several projects and platforms like Sovrin and Hyperledger Indy. As the technology matures and more use cases are developed, it is likely that DIDs will become more widely adopted and integrated into various systems and applications such as digital signing, authentication, and access control.

2.2 Hyperledger Indy and Indy Pool

2.2.1 Hyperledger Indy

Hyperledger Indy is an open-source distributed ledger platform for creating and managing decentralized identifiers (DIDs) and self-sovereign identity (SSI) solutions. It is one of the Hyperledger projects hosted by the Linux Foundation, and it aims to provide a foundation for creating a verifiable, decentralized digital identity infrastructure.

Hyperledger Indy provides a set of components and libraries that can be used to build SSI solutions. Some of the key features and components of Hyperledger Indy include:

- A distributed ledger that can be used to create and manage DIDs. The ledger is implemented using the Sovrin protocol and allows for the creation of decentralized identities that are controlled by the entity themselves, rather than centralized organizations.
- A set of libraries and SDKs that can be used to interact with the ledger and perform different SSI-related tasks, such as creating and managing DIDs, issuing and verifying credentials, and creating secure connections between different entities.

- An **Indy pool** which is a network of nodes running the Indy software and connected to the same ledger. The pool can be used to create and manage DIDs, store public keys, and other related information.
- **Aries protocol and cloud agents (ACA)** which provide a simplified and consistent API that can be used to interact with the ledger and other SSI-related services.

Hyperledger Indy is a widely adopted and flexible platform, it can be integrated with other systems, protocols, and platforms to support various use cases and fit different requirements.

2.2.2 Indy Pool

In the context of our decentralized messaging app, an Indy pool is required in order to store and manage DIDs.

An Indy pool is a group of nodes that run the same version of the Indy software and are connected to the same ledger. Each node in the pool has a copy of the ledger, and they work together to process transactions and maintain the state of the ledger. The nodes can be run by different organizations or individuals, and they can be geographically distributed.

A user can interact with the pool by connecting to any node in the pool, and the node will forward the request to the other nodes. The nodes work together to process the request and reach consensus on the state of the ledger.

The pool can be used to create, read and update DIDs, as well as for storing other related information such as public keys, attributes and credentials. The pool also provides a secure and decentralized way to manage DIDs and ensure that the data on the ledger is accurate and tamper-proof.

Indy pools can be administered and configured to fit specific use cases, such as different levels of trust and security, as well as different governance models. Additionally, the pool can be interfaced with other systems and protocols to provide more functionalities to the DID system, like integration with other identity systems, communication protocols and access control.

2.3 Hperledger Aries and Aries Cloud Agent

2.3.1 Hyperledger Aries

Hyperledger Aries is an open-source project within the Hyperledger community that provides a set of tools and libraries for building decentralized identity (DID) and self-sovereign identity (SSI) solutions.

Aries provides a set of reusable and modular building blocks for creating SSI solutions, including:

- A secure messaging format and protocol that can be used to exchange verifiable credentials and other types of data between different entities.
- A set of tools and libraries that can be used to manage DIDs, credentials, and secure connections with other entities.

Aries Cloud Agent (ACA) which is a software component that runs on a server and provides various services to clients, such as creating and managing DIDs, issuing and verifying credentials, and establishing secure connections with other agents.

Aries aims to be interoperable with different decentralized identity technologies, protocols and ledgers, and it is built to work with other projects in the Hyperledger ecosystem such as Hyperledger Indy. The Aries protocols and libraries can be integrated with other SSI solutions and existing systems, enabling the building of custom SSI solutions.
Aries can be used for a wide range of use cases and industries, for example:

- Verifiable credentials for educational and professional qualifications
- Identity verification for online transactions and access control
- Decentralized identity for Internet of Things (IoT) devices

Like Hyperledger Indy, it is open-source, actively maintained and has a large community of contributors. The code, documentation, and tutorials are available on the Hyperledger Aries GitHub page, and you can also get support from the community and experts on the Hyperledger Aries chat channel.

2.3.2 Aries Cloud Agent

Hyperledger Aries Cloud Agent is a framework for building Verifiable Credential ecosystems. It uses DIDComm messaging and Hyperledger Aries protocols.

DIDComm messaging provides secure and private communication methods on top of the decentralized design of DIDs. It defines how messages integrate with application-level protocols and workflows, and maintains trust seamlessly.

The Aries Cloud Agent will act as a server to process interactions between the different users, such as sending an invitation, accept it or send messages. It is similar to a mail serveur as it will get the messages sent from a client to another. hen the second agent connects to the application, the cloud agent will send him the messages waiting to be read.

Hyperledger Aries provides a shared, reusable, interoperable tool kit designed for initiatives and solutions focused on creating, transmitting and storing verifiable digital credentials. It is infrastructure for blockchain-rooted, peer-to-peer interactions. This project consumes the cryptographic support provided by Hyperledger Ursa, to provide secure secret management and decentralized key management functionality.

Aries Cloud Agent (ACA) is a type of agent software that is used in the context of decentralized identifiers (DIDs) and self-sovereign identity (SSI). ACA is an implementation of the Aries protocol, which is a set of open-source protocols developed by the Hyperledger project to provide a decentralized infrastructure for SSI.

An ACA is a software component that runs on a server and provides various services to clients, such as creating and managing DIDs, issuing and verifying credentials, and establishing secure connections with other agents. It can be thought of as an intermediary that helps users interact with the decentralized infrastructure provided by an Indy Pool (network of nodes running the Indy software and connected to the same ledger).

An ACA can provide the following functionalities:

- Acting as a wallet to store the user's DID and associated keys, making them easily accessible to the user.
- Communicating with other agents and ledgers on behalf of the user, providing a simplified and consistent API.
- Handling the security and encryption of the communication, ensuring that the sensitive data is protected.
- Handling interactions with other agents and ledger on behalf of the user, such as creating and managing connections, sending and receiving messages, and completing different types of interactions.
- Provide a web interface that allows users to interact with the agent and carry out different SSI related tasks.

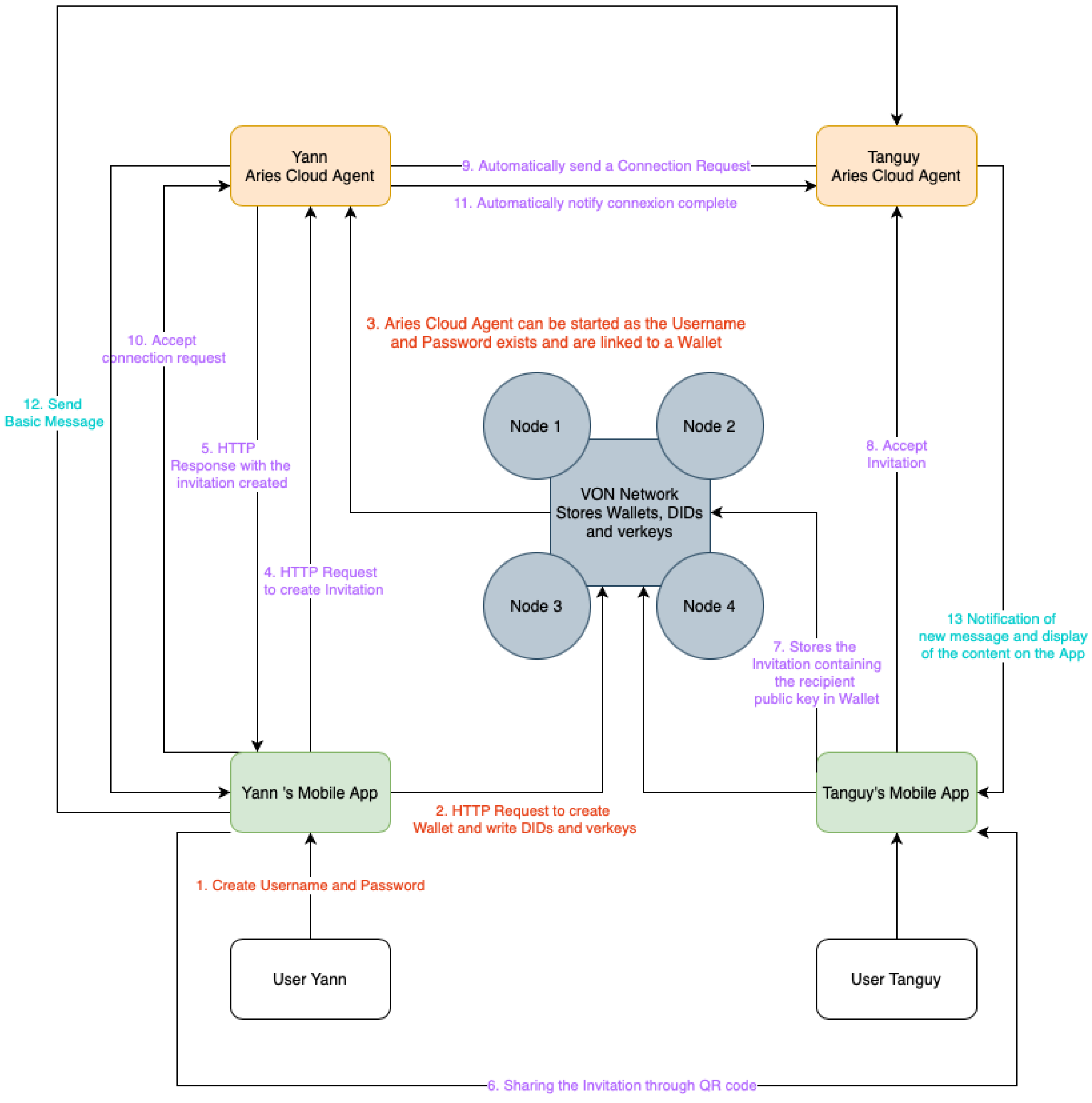
An ACA can be used by individuals, organizations, or service providers, such as identity providers and verifiers, to participate in SSI ecosystems and provide SSI services to their users. It can also be customized and extended to fit specific use cases and integrate with existing systems.

2.4 Mobile App

The application is at the center of our project. This will be installed on user's devices and must have several functionalities:

- Creation of a personal wallet and digital identity (DID).
- Connection to the user's wallet when filling a passcode.
- Sharing of the dids stored on the app's wallet with the Indy Pool
- Sending of an invitation with a qr code through Aries Cloud Agent
- Reception of an invitation and accept or deny it through Aries Cloud Agent
- Selection of the user we want to reach, display the previous conversation and send him a message with Aries Cloud Agent.
- Messages encrypted.

2.5 Overall Architecture


Yann
Aries Cloud Agent
9. Automatically send a Connection Request
11. Automatically notify connexion complete
Tanguy
Aries Cloud Agent
3. Aries Cloud Agent can be started as the Username and Password exists and are linked to a Wallet
10. Accept connection request
12. Send Basic Message
5. HTTP Response with the invitation created
Node 1
Node 2
VON Network
Stores Wallets, DIDs and verkeys
Node 3
Node 4
8. Accept Invitation
4. HTTP Request to create Invitation
13 Notification of new message and display of the content on the App
7. Stores the Invitation containing the recipient public key in Wallet
Yann 's Mobile App
Tanguy's Mobile App
2. HTTP Request to create Wallet and write DIDs and verkeys
1. Create Username and Password
User Yann
User Tanguy
6. Sharing the Invitation through QR code

The Architecture scheme shows how the different components come together. Here, the Mobile Apps are clients and are the only way for users to interact with other bricks.

Once a **User, Yann, creates a Username and a Password (1)**, an HTTP request is automatically sent to the VON Network, **creating a wallet and writing a DID and a verkey in it for him (2)**. Those informations are used to **launch an instance of Aries Cloud Agent (3)**, linked to Yann.

This User can then **create an invitation (4)** by sending an HTTP Request to his Agent which **sends back the invitation (5)**. He then **shares the invitation with a QR code (6)**.

The other user, **Tanguy, receives the invitation (7)**, stores it in his wallet and then **accepts the invitation (8)**.

Automatically, this initiates a **connection request (9)** to Yann. Yann will **accept the connection request (10)** and **notify Tanguy (11)**. Both Users can now **send messages (12)**, which will be received by the other user cloud agent and **sent to the mobile app (13)** as soon as it is connected.

3 Setting up the indy pool

The indy pool is the network of nodes that share data transparently and securely. It is the first thing that need to be settled in order to start the project.

The von network is a portable development level Indy Node network, including a Ledger Browser. It allows a user to see the status of the nodes of a network and browse/search/filter the Ledger Transactions.

It is very convenient for the project for testing purposes as it allows to view and interact with the transactions on a Hyperledger Indy ledger. It typically provides a user-friendly interface for browsing, searching, and filtering the transactions on the ledger, and may also offer additional features such as the ability to view the status of the nodes on the network or to submit new transactions to the ledger. The von network is the software package that provides the core functionality for an indy pool.

Here is how to download the repository and be ready to use it.

```
$git clone https://github.com/bcgov/von-network
$cd von-network
```

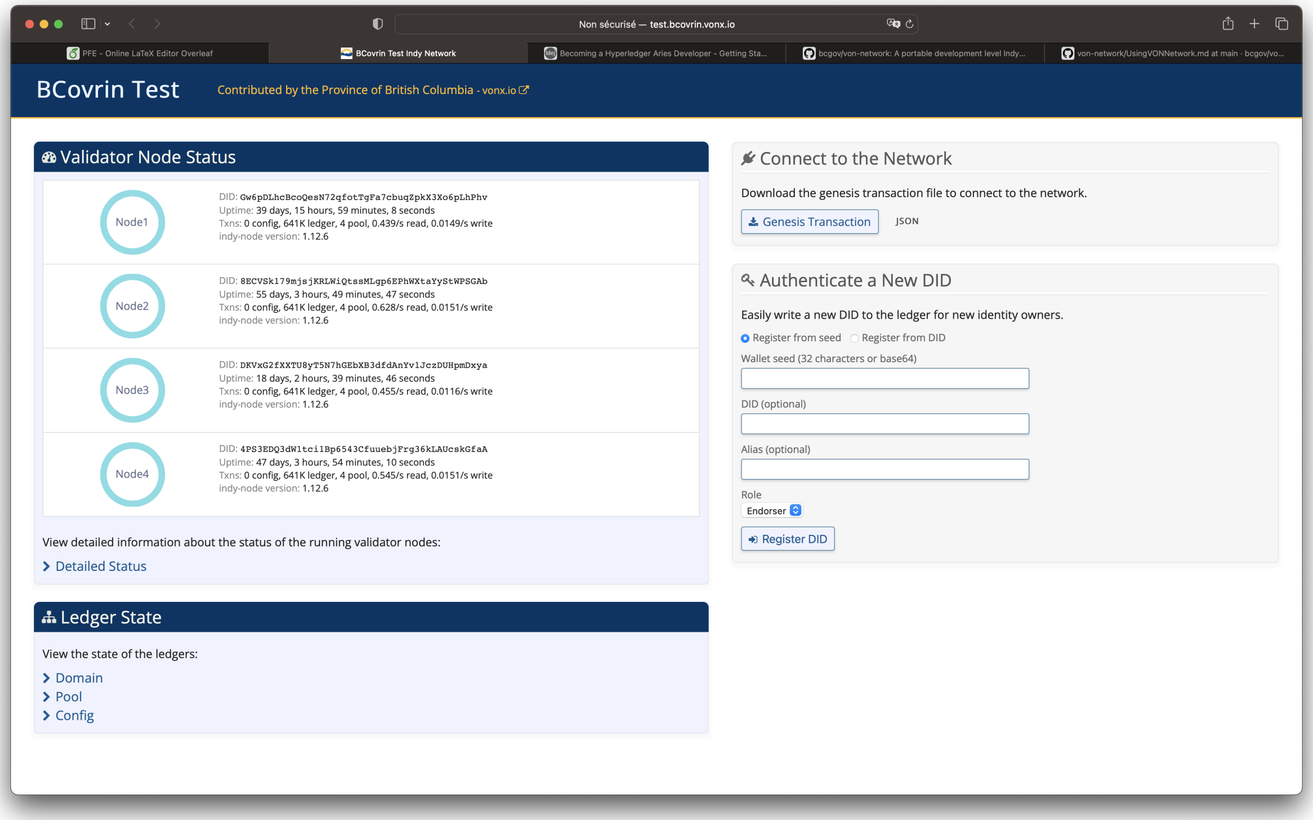
After downloading the repository, the von network can be started with this payload.

```
$./manage start --logs
```

Then, after waiting a few minutes, a status of the network can be displayed by going to

```
http://localhost:9000
```

The content displayed shows the validator node status, the ledger state, the genesis transaction button and an invite to authenticate a new did.



BCovrin Test

Contributed by the Province of British Columbia - vonx.io

Validator Node Status

Node1
DID: Gw6pDLhcBcoQesN72qfotTgFa7cbuqZpkX3Xo6pLhPhv
Uptime: 39 days, 15 hours, 59 minutes, 8 seconds
Txns: 0 config, 641K ledger, 4 pool, 0.439/s read, 0.0149/s write
indy-node version: 1.12.6

Node2
DID: 8ECVSk179mjsjKRLWiQtssMLgp6EPhWXtaYyStWPSGAb
Uptime: 55 days, 3 hours, 49 minutes, 47 seconds
Txns: 0 config, 641K ledger, 4 pool, 0.628/s read, 0.0151/s write
indy-node version: 1.12.6

Node3
DID: DKVxG2fXXTU8yT5N7hGEbXB3dfdAnYv1JczDUHpmDxya
Uptime: 18 days, 2 hours, 39 minutes, 46 seconds
Txns: 0 config, 641K ledger, 4 pool, 0.455/s read, 0.0116/s write
indy-node version: 1.12.6

Node4
DID: 4PS3EDQ3dW1tci1Bp6543CfuuebjFrg36kLAUcskGfaA
Uptime: 47 days, 3 hours, 54 minutes, 10 seconds
Txns: 0 config, 641K ledger, 4 pool, 0.545/s read, 0.0151/s write
indy-node version: 1.12.6

View detailed information about the status of the running validator nodes:
> Detailed Status

Ledger State

View the state of the ledgers:
> Domain
> Pool
> Config

Connect to the Network

Download the genesis transaction file to connect to the network.
Genesis Transaction JSON

Authenticate a New DID

Easily write a new DID to the ledger for new identity owners.
[✓ Register from seed] Register from DID
Wallet seed (32 characters or base64)
DID (optional)
Alias (optional)
Role
Endorser
Register DID

This invite can be used to create a new did, did that is needed in order to establish a connection between to agents.

It is also possible to stop the Indy pool, with the following command:

```
$./manage stop
```

4 Starting the Aries Cloud Agents

For now, we chose to run the Aries Cloud Agent as a standalone, this may change.

The Aries Cloud Agent (ACA-py) is available as a Python Package.

```
$ pip3 install aries-cloudagent
```

It is also mandatory to install libindy, which is an Ubuntu package able to create and manage a wallet. The wallet stores connections such as connection records, credential exchange records and credentials.

Here is the command lines to install libindy:

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 68DB5E88
$ sudo add-apt-repository "deb https://repo.sovrin.org/sdk/deb bionic master
  "

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

4.1 First approach on a connection with the API

Before managing a mobile app, we started a first try directly with the aca-py API. The goal of this was to create a connection between two cloud agents only through two terminals.

Once the two agents are started.

5 Creating the mobile app

A messenger service could be mainly developed first as a web application or a mobile application. We started to try both to see which one was the most convenient. For the web application, we began to initiate a project with the Django framework but realized soon that it was pretty long to handle correctly for beginners (lots of requirements, files. . .). Concerning the mobile application, a Python library called Kivy seemed lighter and simpler to get used to. As we were a little familiar with mobile applications with a previous project with Android Studio, we decided to choose the mobile application.

The main goal of the mobile application is for its user(s) to log in their wallets and begin a conversation with other users after exchanging their DIDs through a QR code.

5.1 The python library used : Kivy

To create a mobile app compatible with the Python Packages of Hyperledger, the Kivy library seemed like the best choice. The previous Python functions developed at the beginning of the project should only be integrated in the Kivy code. Kivy is an open-source Python library that is used to develop cross-platform applications, thus, allowing us to develop the app for Android and iOS.

Kivy uses modern graphical user interface (GUI) libraries such as OpenGL to provide fast and responsive interfaces for users. It was a strong advantage compared to Android Studio for example, where it is needed to compile the Java code during several dozens of seconds in order to view the resulting app. With Kivy, the result takes much less time, about 3 seconds.

5.1.1 The setup

The setting up of an empty Kivy application is a straightforward process. Once the Python library is installed with pip (pip install kivy), a single .py file can contain all that is needed, although we will see that we used several layout files in the project to make it clearer.

5.1.2 Basic Concepts (App, Layouts, Widgets. . .)

App The most important class of a Kivy project is the App class. When a Kivy file is launched, it has to start the app (MyApp for example) by calling the function MyApp.run(). The method build(self) in MyApp has to be coded to initiate the first actions of the app. For an application with several screens (or views) like a messenger application, build has to create a ScreenManager() object, add the different screens, and return it.

Screens For the mobile application, it was needed to create a Screen object for each view. A Screen is built in different parts. The "interface" part is coded in a distinct .kv file (but it could be coded directly in the Screen class). It is written in a language called Kivy Language, which is a declarative language. In this file, similar to .css (web) or .xml (Android) files, we design the interface with layouts (parts of the screen) and widgets (items) of one or multiple screens. We can also define how the widgets should behave and respond to user input (e.g. launching a method on a pressed button). The size, position, and other properties of the widgets are also defined there. The "programmative" part is coded through methods belonging to the Screen class. That is where the methods launched by buttons are written. It is very useful to interact with the Hyperledger library and objects, to pass from a screen to another or to store variables of the main App and accessible to all the screen.

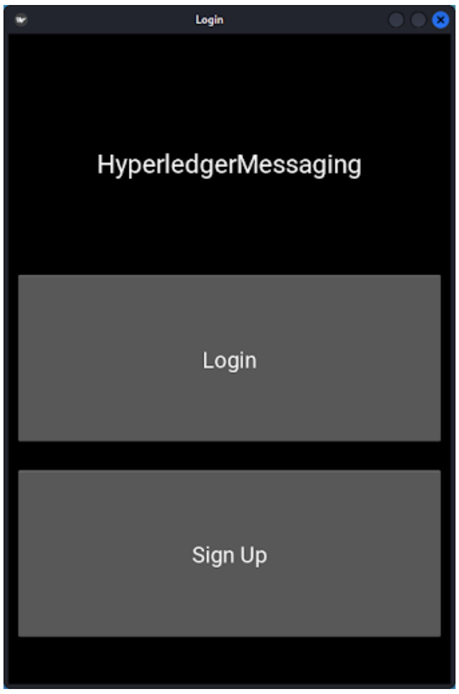
Widgets Widgets are basically the resources of a Screen. They could be Button (to launch functions or transitions between screens), Label (to display a text to the user), TextInput (to interact with the user). Widgets can be implemented to a Screen with a .kv file or directly in a method, using the add_widget() function. The widgets of a screen can be listed or identified with an id, and thus, modified or removed. Widgets were very convenient for all the changes that the screens needed to have during their use (new message, notifications indicating a new message or invitation,...).

5.2 The basic architecture of the application

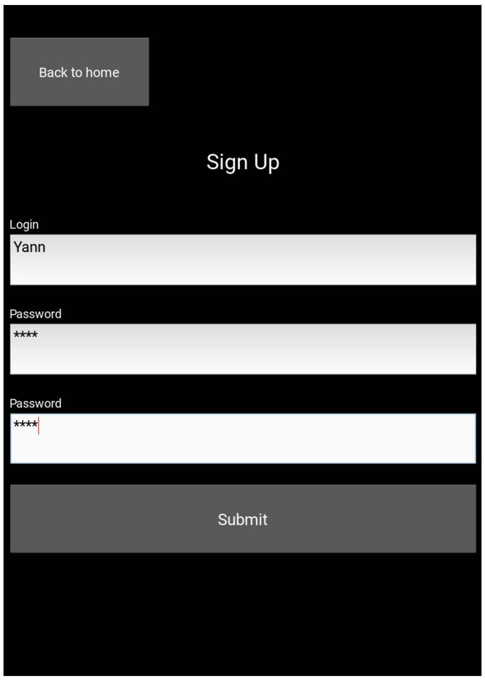
At first, we needed to develop the basic architecture of the application (different screens, inputs, messages shown,...), with no link to an Indy block-chain.

Although, having previously apprehended Hyperledger Indy by coding lines that allowed us to create, open and close wallets, we implemented these functions along the development of the basic application and especially for the Login and Sign Up pages. We knew that we will need to have an opened wallet during all the actions that the user will trigger (send and receive invitations, messages...)

When the user starts the application, he enters the Home screen. In this Home screen, 2 buttons are displayed : Login and Sign Up, pointing to the 2 corresponding screens.



At first, the user must sign up. When he clicks on the button, the screen changes and looks like this.



By sending the login and password through the Submit button, the application will try to create an Hyperledger wallet using this method:

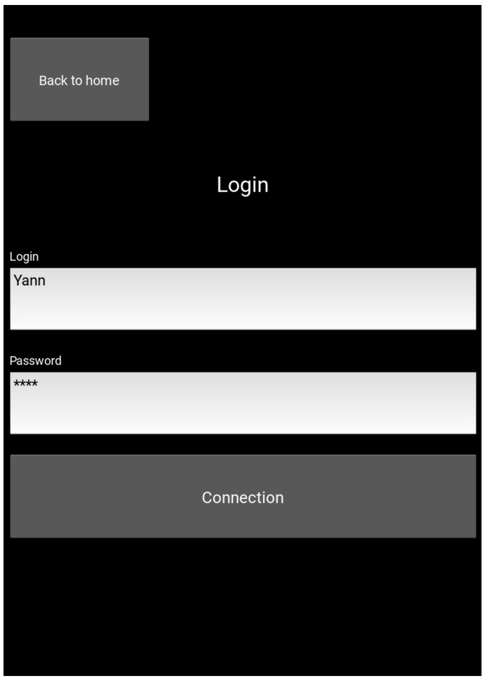
```
async def sign(me, key):
wallet_config = '{"id": "%s-wallet"}' % me
wallet_credentials = '{"key": "%s"}' % key
wallet_handle = None
try:
    await wallet.create_wallet(wallet_config, wallet_credentials)
    wallet_handle = await wallet.open_wallet(wallet_config,
wallet_credentials)
    success = True
except:
    success = False
if not success:
    success, wallet_handle = await login(me, key)
return success, wallet_handle
```

If it succeeds, the wallet is created, opened and closed (not shown on this part of the code), and the user is sent to the Login page, with login and password already pre-filled. The resulting variable of the successful connection, wallet_handle, is stored in the App and could be used by another screen if an action is required to be logged in the wallet.

If it does not, it will mean that the user already exists but it is the wrong password, and it will tell the user by an adequate message below the submit button.

If the user already exists but the good password is sent, the user will be sent to the Login with login and password pre-filled. The button Back to home sends the user back to the previous screen.

The Login screen is very similar to the SignUp screen. On this screen, the application will try to open a Hyperledger wallet with the method **wallet.open_wallet (id = login and key = password)**.

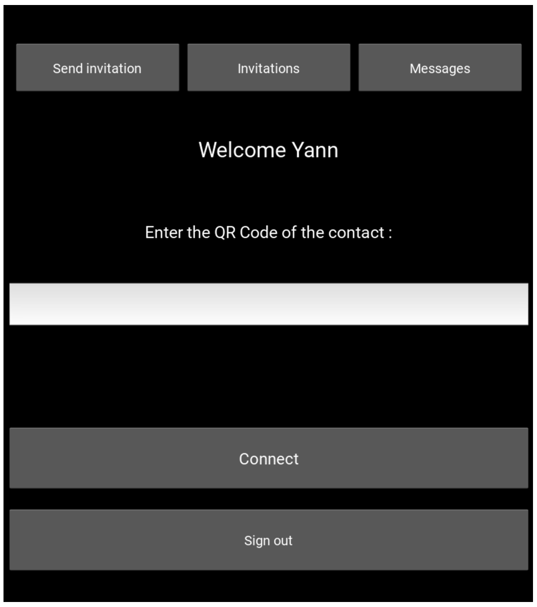


```
    async def login(me, key):
    wallet_config = '{"id": "%s-wallet"}' % me
    wallet_credentials = '{"key": "%s"}' % key
    print(wallet_config, wallet_credentials)
    wallet_handle = None
    try:
        wallet_handle = await wallet.open_wallet(wallet_config,
   wallet_credentials)
        success = True
    except:
        success = False
    return success, wallet_handle
```

If it succeeds, the wallet is opened and the user is logged in. The variable wallet_handle is also stored in the App.

If it does not, a message is shown below the Connection button, indicating the user that he types a wrong login or password. Like the SignUp screen, the button Back to home sends the user back to the Home screen.

When the user is logged in, he is redirected to the Connected screen :



In this screen, multiple actions can be launched : sending/receiving QR codes or invitations, going to the chat page of the contacts of the user or signing out.

Connection between users (including the sending/receiving of the QR code and the invitation) will be explained in the part 6 of the report. Messaging between users will be explained in the part 7.

6 Process the connection between users through the app

To enable users to send messages to each other, it is needed to implement a process for establishing connections between their mobile wallets. On the VON Network, connections are established after a back and forth of invitations sent and accepted. To start the establishment of the communication, we used the principle of the QR code, containing a Recipient Public Key of the sender of the QR code and other information indicating where to send the invitation. The QR code allow users to share it physically when 2 persons meet and want to contact themselves later. By scanning this QR code, the invitee will be able to process a connection between him and the inviter and be able to communicate.

6.1 Prerequisite

The instance of the VON Network and at least one instance of an Aries Cloud Agent must be running in order to begin the connection.

To ease the development part, every user have to launch his own Cloud Agent but it could not be used in production. Indeed, each user of the application can not run a server permanently.

Concerning the exchanges between the different actors (user, agent and network), we focused on the security between agents and not between the user and its own agent. We know that if this application have to be used on the Internet, other security mechanisms have to be implemented between users and agents. An agent is not necessarily well-intentioned, and it is necessary to not give him an all-access to the user's wallet. This is not what we developed in the project but we are aware of the security problem.

To establish the connection between users, it is then mandatory to have two instances of Aries Cloud Agent already running on a computer which can be the same that hosts the VON Network.

Here is the payload that is used to start an exemple agent called Yann. A DID has already been created in the previous steps in Yann's wallet.

```
    aca-py start \
  --label Yann \
  -it http 0.0.0.0 8000 \
  -ot http \
  --admin 0.0.0.0 11000 \
  --admin-insecure-mode \
  --genesis-url http://localhost:9000/genesis \
  --seed Yann0000000000000000000000000000 \
  --endpoint http://localhost:8000/ \
  --debug-connections \
  --public-invites \
  --auto-provision \
  --wallet-type indy \
  --wallet-name Yann \
  --wallet-key secret
```

Listing 1: Starting Yann Cloud Agent

The Seed: Yann0000000000000000000000000000 is the seed that has been used to previously create the DID on the VON Network.

The Cloud Agent labellised "Tanguy" is created the same way. The seed also corresponds to a DID previously created on the VON Network.

```
    aca-py start \
  --label Tanguy \
  -it http 0.0.0.0 8000 \
  -ot http \
  --admin 0.0.0.0 11000 \
  --admin-insecure-mode \
  --genesis-url http://localhost:9000/genesis \
  --seed Tanguy00000000000000000000000000 \
  --endpoint http://localhost:8000/ \
  --debug-connections \
  --public-invites \
  --auto-provision \
  --wallet-type indy \
  --wallet-name Tanguy \
  --wallet-key secret
```

Listing 2: Starting Tanguy Cloud Agent

In these two payloads, the genesis-url http://localhost:9000/genesis corresponds to the endpoint of the VON Network running. The endpoint of the ACA-py API is http://localhost:11000. This endpoint will be used through the application to send http requests to the several agents in order to perform actions.

For our purpose, we created a script to launch a cloud agent. After a user registered his username and password and before he connects, we just start the corresponding agent in the terminal :

```
    $python3 launch_agent.py username password
```

This will simulate the cloud agent running continuously.

6.2 Generate the QR code of invitation

The next step is to generate a QR code with Yann's mobile app. Yann is the user that initiates the Connection Protocol by creating an invitation, here Yann. This is the message that must be sent.

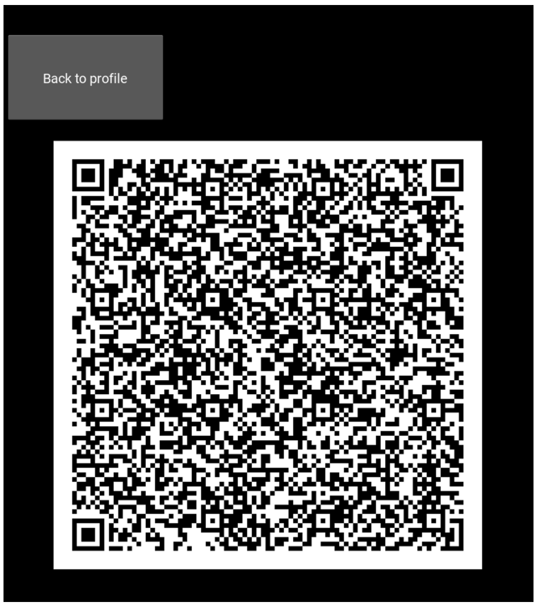
```
    curl -X 'POST' \
  'http://0.0.0.0:11000/out-of-band/create-invitation' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "@type": "did:sov:BzCbsNYhMrjHiqZDTUASHg;spec/out-of-band/1.0/invitation",
  "handshake_protocols": [
    "did:sov:BzCbsNYhMrjHiqZDTUASHg;spec/didexchange/1.0"
  ]
}
'
```

This is a POST request targeting the endpoint **out-of-band/create-invitation** with the data filled in a json format. In this query, the endpoint have to be changed depending on the location of the user's agent. Concerning the DID, we did not succeed to change it. The one specified by the Connection Protocol (BzCbsNYhMrjHiqZDTUASHg) does not seem to be replaceable by the DID the user have to send without having an error while sending the QR code.

The response of the server should look like this in the API:

```
    {
  "invitation_url": "http://localhost:8000/?oob=
    eyJAdHlwZSI6ICJkaWQ6c292OkJ6Q2JzTlloTXJqSGlxWkRUVUFTSGc7c3BlYy9vdXQtb2Y
    ...",
  "invi_msg_id": "d95ff44a-caee-49b9-aa2d-1f7d3e0b7c40",
  "state": "initial",
  "oob_id": "7a65693c-08f3-47f5-8bbe-ab8e8e420c1f",
  "trace": false,
  "invitation": {
    "@type": "did:sov:BzCbsNYhMrjHiqZDTUASHg;spec/out-of-band/1.1/invitation
    ",
    "@id": "d95ff44a-caee-49b9-aa2d-1f7d3e0b7c40",
    "handshake_protocols": [
      "did:sov:BzCbsNYhMrjHiqZDTUASHg;spec/didexchange/1.0"
    ],
    "services": [
      {
        "id": "#inline",
        "type": "did-communication",
        "recipientKeys": [
          "did:key:z6MkswEpHemJGkX8ikpiyk5GEedEEgwxJPNxw1XfeiP8hRYa"
        ],
        "serviceEndpoint": "http://localhost:8000/"
      }
    ],
    "label": "Yann"
  }
}
```

Yann can create the QR code through the "Send Invitation" tab of his profile page. By clicking on it, the QR code will be generated. Sometimes, we noticed that the QR code generated was not readable by the Python library used. To fix this problem, we implemented a while loop that generates the QR code and checks that the QR code is readable before exiting this loop and displaying the code.



To generates the QR code, the previous POST request has to be sent to the user's agent. The response will be used as the text behind the QR code, but with small changes such as label (name of the user that will include his DID) and verkey.

In order to be more in control of the security mechanisms that the simple use of the API, we decided to create a new DID for each new invitation. The creation of a new DID generates a public key (verkey) that can be sent with the QR code by modifying the json received by the API response.

So in our mobile application, each user receiving a new invitation of Yann has a different public key of Yann. The different public keys are stored directly in the wallets of the QR code receivers and do not pass through the agent. This public key will be used to encrypt the content of the messages that will be send. We did not implement any mechanisms that will ensure that the messages sent are encrypted with the public key desired (the one sent to the user that we talk to) but it could be a new function to add to the mobile application.

Here is the Python code that is used in the app :

```
 MyDID, MyVerkey = await did.create_and_store_my_did(wallet_handle =
wallet_handle, did_json = "{}")
 while not GoodQR:
        print('Creating invitation link')
        r = requests.post("http://localhost:"+str(app.second_port)+"/out
-of-band/create-invitation", json={"@type": "did:sov:
BzCbsNYhMrjHiqZDTUASHg;spec/out-of-band/1.0/invitation",
                            "handshake_protocols": [
                            "did:sov:BzCbsNYhMrjHiqZDTUASHg;spec/
didexchange/1.0"
                             ]
                             })
        json_response = json.loads(r.text)
        json_invite = json_response["invitation"]
        json_invite["recipientKeys"] = [MyVerkey]
        json_invite["label"] = app.username+":"+MyDID
        img = qrcode.make(json_invite)    img.save('QRCode'+app.username
+'.png')
        wimg = Image(source='QRCode'+app.username+'.png')
        time.sleep(2)
```

```
            img=cv2.imread("QRCode"+app.username+".png")
            det=cv2.QRCodeDetector()
            invitation, pts, st_code=det.detectAndDecode(img)
            if len(invitation) > 10:
                GoodQR = True
```

Listing 3: Generating the QR code

Once this invitation is registered, the ACA-py instance will log that it has created an invitation:

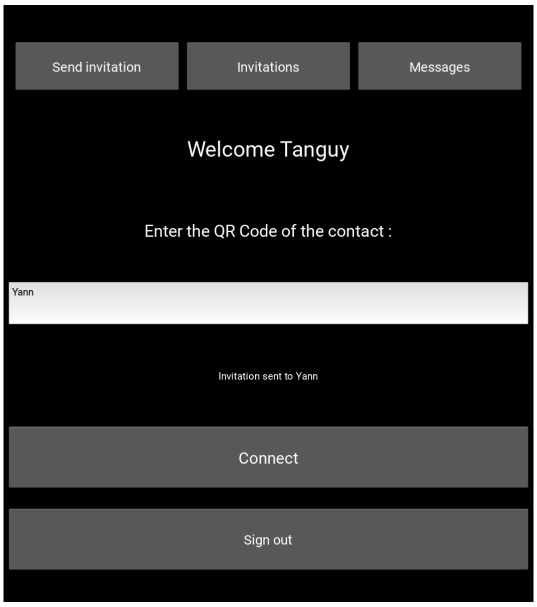
```
    Created new connection
    connection: {'routing_state': 'none', 'invitation_key': '
   b4eygXuTvzdSv1XK3ZwQtzdB8t79gcC2W46uDUz45XC', 'accept': 'manual', '
   updated_at': '2021-03-11 08:01:16.546248Z', 'created_at': '2021-03-11
   08:01:16.546248Z', 'connection_id': '9ebac177-a3d4-4a74-be42-82f4e0cafefa
   ', 'state': 'invitation', 'invitation_mode': 'once', 'their_role': '
   invitee', 'rfc23_state': 'invitation-sent'}

Added Invitation
    connection: {'routing_state': 'none', 'invitation_key': '
   b4eygXuTvzdSv1XK3ZwQtzdB8t79gcC2W46uDUz45XC', 'accept': 'manual', '
   updated_at': '2021-03-11 08:01:16.550301Z', 'created_at': '2021-03-11
   08:01:16.546248Z', 'connection_id': '9ebac177-a3d4-4a74-be42-82f4e0cafefa
   ', 'invitation_msg_id': '638728b4-63b1-4a9a-82b8-c07d72925196', 'state':
   'invitation', 'invitation_mode': 'once', 'their_role': 'invitee', '
   rfc23_state': 'invitation-sent'}
```

Listing 4: Output from Yann's AcaPy terminal

6.3 Connection request

Tanguy receives the invitation of Yann by scanning the QR code. In our App, as we develop the test application on a computer, Tanguy simply need to enter the name of the user that generated the QR Code and click on "Connect". The application will search for a PNG file named QRCodeYann.png and will scann it.



Automatically, information included in the invitation message will be used to 1. store the public key of Yann in the wallet and 2. send a connection request going to the inviter cloud agent, Yann. The connection request is splitted into 2 requests, receive-invitation and accept-invitation.

Here is the Python code that is used in the app :

```python
        img=cv2.imread("QRCode"+contact+".png")
        det=cv2.QRCodeDetector()
        invitation, pts, st_code=det.detectAndDecode(img)
        invitation = json.loads(invitation)
        TheirVerkey = invitation["recipientKeys"][0]
        TheirDID = invitation["label"].split(":")[1]
        await store_their_did(app.username, app.login, contact, TheirDID,
   TheirVerkey)
        r = requests.post("http://0.0.0.0:"+str(app.second_port)+"/out-of-
   band/receive-invitation", json = invitation)
        response = json.loads(r.text)
        connection_id = response["connection_id"]
        r = requests.post("http://0.0.0.0:"+str(app.second_port)+"/
   didexchange/"+connection_id+"/accept-invitation")
```

Listing 5: Sending of the connection request

Tanguy, the invitee will provision a new DID according to the DID method spec. This means creating a matching DID and key. This DID contains the following:

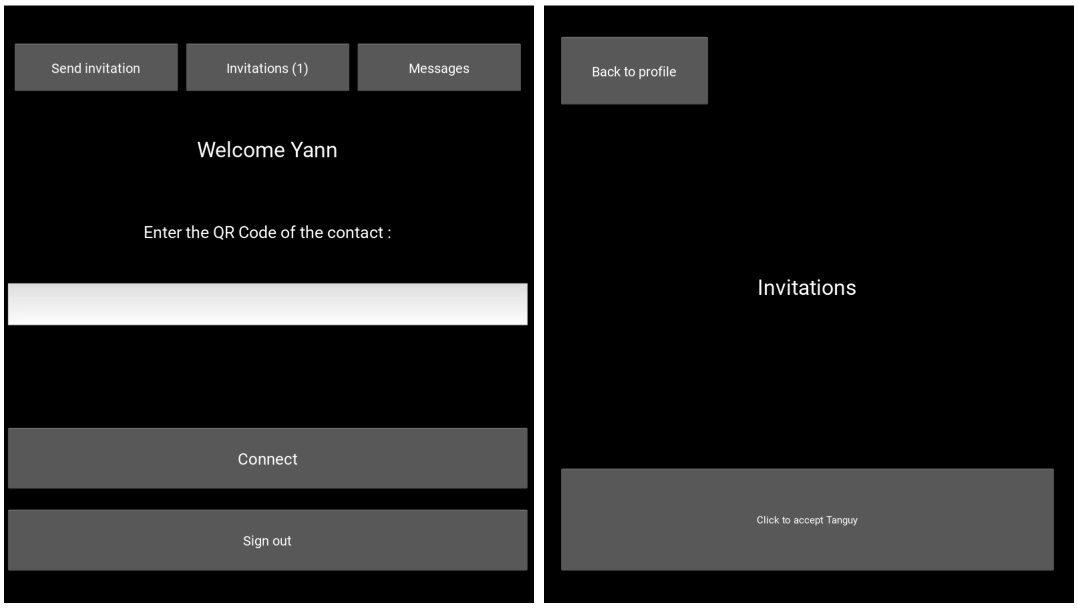
- Label
- Tanguy's DID
- Tanguy's DIDDoc
- Relationship Keys
- Routing Keys
- Service Endpoints

The newly provisioned DID and DID document is presented in the connection_request message as follows:

```
    {
  "@id": "5678876542345",
  "@type": "https://didcomm.org/connections/1.0/request",
  "label": "Tanguy",
  "connection": {
    "DID": "B.did@B:A",
    "DIDDoc": {
        "@context": "https://w3id.org/did/v1"
        // DID document contents here.
    }
  }
}
```

6.3.1 Connection Response

After receiving the connection request, Yann, the inviter process it. The Aries Cloud Agent checks if the information presented with the key used match.



After the connection request is sent by Tanguy, a notification will appear in the tab "Invitations" in the profile of Yann. If he is willing to accept the connection, he will click on "Click to accept Tanguy" in the Invitations tab and it will trigger the establishment of the a connection by sending a connection response to Tanguy.

To do that, the program starts by listing all Yann's connections with the API of his Cloud Agent. Then, if a connection matches a state "request-received", it will add buttons (labeled by the name of the sender) that will accept the connections if he clicks on it.

Here is the Python code that is used in the app :

```
class Invitations(Screen):
    def on_enter(self):
        app = App.get_running_app()
        Invitations = []
        r = requests.get("http://0.0.0.0:"+str(app.second_port)+"/
   connections").text
        r_json = json.loads(r)
        r_json = r_json["results"]
        for item in r_json:
            if item["rfc23_state"] == "request-received":
                Invitations.append(("Click to accept "+item["their_label"],
   "http://0.0.0.0:"+str(app.second_port)+"/didexchange/"+item["
   connection_id"]+"/accept-request"))
        for item in Invitations:
            text, url = item
            button = Button(text=text, on_press=lambda btn: self.
   button_pressed(item), size_hint=(1, 0.3))           self.ids.
   invitation_layout.add_widget(button)

    def button_pressed(self, item):
        text, url = item
        response = requests.post(url)
        print(response.text)
        self.manager.transition = SlideTransition(direction="right")
        self.manager.current = 'connected'
```

Listing 6: Listing the connections requests received

```
{
  "@type": "https://didcomm.org/connections/1.0/response",
  "@id": "12345678900987654321",
  "~thread": {
    "thid": "<@id of request message>"
  },
  "connection": {
    "DID": "A.did@B:A",
    "DIDDoc": {
      "@context": "https://w3id.org/did/v1"
      // DID document contents here.
    }
  }
}
```

The above message is required to be signed. The connection attribute above will be base64URL encoded and included as part of the sig_data attribute of the signed field. The result looks like this:

```
    {
  "@type": "https://didcomm.org/connections/1.0/response",
  "@id": "12345678900987654321",
  "~thread": {
    "thid": "<@id of request message>"
  },
  "connection~sig": {
    "@type": "https://didcomm.org/signature/1.0/ed25519Sha512_single",
    "signature": "<digital signature function output>",
    "sig_data": "<base64URL(64bit_integer_from_unix_epoch||
   connection_attribute)>",
    "signer": "<signing_verkey>"
  }
}
```

This connection response must be signed by the invitee (?? the inviter ?).

6.3.2 Response Processing

When the ***invitee*** receives the response message, he verifies the sig_data provided. After validation, he updates his wallet with the new connection information.

After the Response is received, the connection is technically complete. This remains unconfirmed to the ***inviter*** however. The ***invitee*** sends a last message to the ***inviter*** to confirm the connection acknowledgement. In our case, this will be a message saying : Connection complete with “Agent”.

7 Implement the messaging functionality

After having managed the connection between users, it is now time to focus on sending and receiving messages between the users' mobile wallets. In order to do that, the app will use the basic message protocol and the Aries Cloud Agent API.

7.1 Basic Message Protocol

"The Aries RFC 0095: Basic Message Protocol describes a stateless, easy to support user message protocol. It has a single message type used to communicate." (**basicmessage**) Here is the basic structure of a basic message. The field "content" contains the message.

```
    {
    "@id": "123456780",
    "@type": "https://didcomm.org/basicmessage/1.0/message",
    "~l10n": { "locale": "en" },
    "sent_time": "2019-01-15 18:42:01Z",
    "content": "Your hovercraft is full of eels."
    }
```

There are two roles in this protocol: sender and receiver. Each agent is both sender and receiver. In order to send a message, the app sends a POST request to the admin url endpoint of the other.

Each user has its own connection id which corresponds to the connection with the other, in this case if we suppose :

- connexion_id Yann → Tanguy : 403f89d5-d6e8-4e95-8383-7389fcc3e449
- connexion_id Tanguy → Yann : bba96390-7869-42be-9a19-e03bda8db476

Then, Yann must send this POST request in order to reach Tanguy.

```
        curl -X 'POST' \
  'http://0.0.0.0:11000/connections/403f89d5-d6e8-4e95-8383-7389fcc3e449/
   send-message' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "content": "Hello ca va?"
}'
```

Tanguy must send this POST request in order to reach Yann.

```
    curl -X 'POST' \
  'http://0.0.0.0:11000/connections/bba96390-7869-42be-9a19-e03bda8db476/
   send-message' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "content": "Oui et toi?"
}'
```

7.2 Receiving the messages

To receive the message, the app must use the webhooks. These webhooks are records for every action involving the cloud agents. When ACA-Py is started with the command line parameter **–webhook-url url_to_specify**

```
    aca-py start \
--label Yann \
-it http 0.0.0.0 8000 \
-ot http \
--admin 0.0.0.0 11000 \
--admin-insecure-mode \
--genesis-url http://localhost:9000/genesis \
--seed Yann000000000000000000000000000 \
--endpoint http://localhost:8000/ \
--debug-connections \
--public-invites \
--auto-provision \
--wallet-type indy \
--wallet-name Yann \
--wallet-key secret
--webhook-url <url_to_specify>
```

POST requests are sent to the specified URL each time a record is created or updated, i.e. each time an action is performed through the API.

The path to these records differs with each record. The subject of the record is indeed added to the URL, for example: **https://localhost:10000** becomes **http://localhost/10000/webhooks/topic/basicmessages/** when a connection record is updated.

Here is a simple Python code that can be used to retrieve a request that was sent to our cloud agent.

```
    from flask import Flask, request

    app = Flask(__name__)
    @app.route('/webhooks/topic/basicmessages/', methods=['POST'])
    def webhook():
        #Process the webhook request
        data = request.get_json()
        print(data)
        return "OK"
```

8 Encrypt the messages

As we want the messages to be kept secret, the messages that are sent between the users' mobile wallets must be encrypted. This typically involves using the indy-sdk or other relevant libraries to encrypt the messages using the users' cryptographic keys, and then encode the encrypted messages in a format that can be transmitted over the network.

For this purpose, we may have to use another protocol: Aries RFC 0019: Encryption Envelope

An Encryption Envelope is a message sent between identities to establish a connection, issuing a verifiable credential, sharing a chat,etc. It follows from the didcomm protocol such as basic messages.

In our case, we will use this envelope to wrap it around a plaintext message. This will permit secure sending and routing. In real life, the nodes of the Indy Pool are not stored locally on the same computer. Plaintext messages have to pass through many agents in many different geographical spaces, and an encryption envelope ensure confidentiality for each step of the transmission from the sender to the receiver.

In our application, the envelop does not sign the transaction and thus does not prove completely the identity of the sender (even if the sender has to be logged in the right wallet).
But, as it was explained before, one way to ensure that the sender is the one that he claims to be, the receiver could verify the public key used to encrypt the message, or by signing the envelop with the argument "sender_verkey".

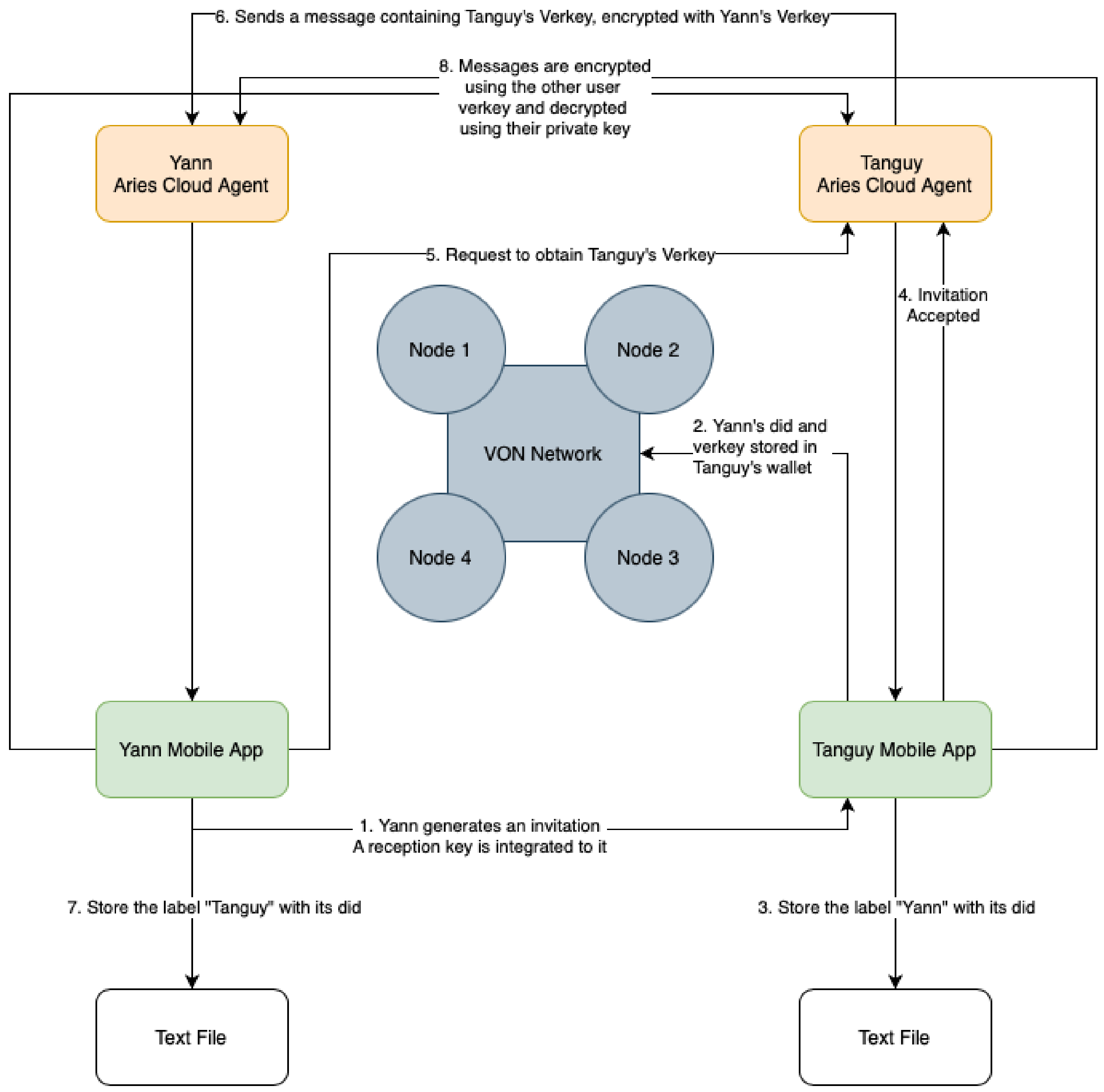
```
async def send_encrypt(user, wallet_handle, DIDs, message, contact):
    verkey = None
    for DID in DIDs:
        if ":" not in DID:
            continue
        DID = DID.split(":")
        name = DID[0]
        their_did = DID[1]
        if name == contact:
            verkey = await did.key_for_local_did(wallet_handle, their_did)
            break
    print(verkey)
    encrypted_message = await crypto.pack_message(wallet_handle =
   wallet_handle, message = message, recipient_verkeys = [verkey],
   sender_verkey=None)
    return verkey, encrypted_message
```

Here is the code used to encrypt the messages with the function pack(). The arguments of this function are:

- The user sending the message.
- wallet_handle: This is the handle to the wallet that contains the sender's secrets. item DIDs : The DIDs of the sender and the receiver.
- message: This is the basic message that a user wants to send to another user. It is passed as a string. As our message is a JSON object it should be converted to string format first.
- contact : The name of the receiver of the message.

The **verkey** is the receiver's verkey which is passed as a string. This verkey is used to encrypt the message and is used as input to the encryption algorithm.

8.0.1 Process


6. Sends a message containing Tanguy's Verkey, encrypted with Yann's Verkey
8. Messages are encrypted using the other user verkey and decrypted using their private key
Yann Aries Cloud Agent
Tanguy Aries Cloud Agent
5. Request to obtain Tanguy's Verkey
4. Invitation Accepted
Node 1
Node 2
VON Network
2. Yann's did and verkey stored in Tanguy's wallet
Node 4
Node 3
Yann Mobile App
Tanguy Mobile App
1. Yann generates an invitation A reception key is integrated to it
7. Store the label "Tanguy" with its did
3. Store the label "Yann" with its did
Text File
Text File

When Yann generates an invitation, a "Reception Key" is integrated to it. Yann sends this Reception Key to Tanguy.

Tanguy receives a did and a verkey that he will store in his wallet. So that Tanguy can access this DID coming from Yann, the application creates a text file which contains a label "Yann:" with its did.

Tanguy retrieves the verkey that has just been stored and encrypts a message with this key. The function crypto.pack_message **encrypts the message**. The program then returns the variable **verkey** and the **encrypted message**.

Once Yann has sent a qr code, Tanguy can then send an encrypted message. For the moment, Yann cannot yet encrypt his messages with this verkey because Tanguy does not have the associated decryption key.

When Yann wants to have a reception key, when he opens the conversation, if he doesn't already have Tanguy's verkey, he sends him a plain text request in the form of a message in the text which will automatically create a response from Tanguy. Tanguy's agent will automatically create a did and a verkey and use the create did and verkey function. This agent then sends its encrypted verkey with Yann's verkey. Yann decrypts the message and retrieves the verkey which he writes in a text file. Each time one of the two users reconnects with the conversation, he can read the verkey of the opposite user written in the text file.

8.0.2 Security

As our App is working right now, an attacker(1) could possibly detect what type of message is used to ask for the receiver(2) verkey. Despite this, he(1) could not receive the message that the other user(2) sends to him because the user (2) would not have any verkey and therefore could not send back his encrypted verkey. As every invitation contains a different random verkey, this attack could not be used to get verkeys of new users with a verkey that another user we are already connected with sent to us.
It seems pretty secure. The only way to bypass this would be to get the qrcode of the invitation between two other users.

8.1 Decrypt the messages

```
 async def decrypt_message(wallet_handle, encrypted_message):
 encrypted_message = json.dumps(encrypted_message)
 encrypted_message = str.encode(encrypted_message)
 print(encrypted_message)
 decrypted_message = await crypto.unpack_message(wallet_handle,
encrypted_message)
 return decrypted_message
```

After the messages have been encoded, you will need to implement a mechanism for decrypting them when they are received by the recipient's mobile wallet. This typically involves using the indy-sdk or other relevant libraries to decrypt the messages using the recipient's private key, and then decode the encrypted message to reveal its original content.

8.2 Notifications

The Application works like emails.
Firstly, Yann writes a message and sends it to the cloud agent of Tanguy.
This cloud agent acts as a server. It receives the message and store it with the other messages that Tanguy did not receive yet. As soon as Tanguy's mobile app is launched and connected, the cloud agent sends the pile of messages to the client, the app.
Tanguy's client receives the pile of messages.
We implemented a notification system so that each time the cloud agent sends a pile of messages, a notification is displayed on the app, counting the number of messages that have not been red yet.

9 Security Considerations

9.1 Link between a user and one cloud agent

Using the Aries Cloud Agent library, each user have his own cloud agent running. Every cloud agent running can be acessed regarding his port. It can be a potential flaw as it could be possible to link someone identity regarding the port number of the linked agent. During our development, it was possible to specify the port we wanted to link our wallet with.

If we wanted our project to be put in production, users should not be able to chose their ports as it would lead to critical flaws.

For example, if Tanguy logs himself with the port number of Yann, in his contacts, he will see Tanguy, which is a non sense.

9.2 Nothing across the von network

After having shared some messages between two users, it is interesting to look if something has been written on the Von Network. For security reasons, nothing must be written on the network, because as a blockchain, one of the main principle of the pool is that everybody can read on it. It would not be secured to have all of the conversations leaked on the pool.

Fortunately, after looking in the domain tab, nothing has changed.

10 Overall Experience

We were able to create a messaging application with decentralized identity using Hyperledger Indy and Aries. It was still difficult to understand the different building blocks of this project, mainly because of the different GitHub documentations that changed and varied with the ever-changing tools.

We got very promising results and think we delivered a test application that could be adapted to production.

10.1 Difficulties encountered

During the course of this project, we encountered several challenges that we thought would be worth mentioning. First of all, the main challenge was the lack of time when it came to managing a project of this size. Indeed, we had to learn how to work with several tools composed of several thousand lines of code in a very short time.

During this project, we had to create a messaging application that did not require you to give your identity, your phone number or to create or use an account. To do this, we had to discover the Hyperledger project which is a suite of frameworks aimed at deploying enterprise-grade blockchain solutions.

The first trouble we had was to understand the project. When we heard "decentralized messaging app" we thought the messages needed to be stored on a blockchain, but it did not make any sense as it would be readable by anybody.

We started our project by searching and learning content about Hyperledger Indy, the blockchain and how it worked. It was quite difficult to dive in this subject as we had few knowledge on it.

We looked for how it worked in detail with the help of online documentation on the official Hyperledger website or github. At this time, the problem that we did not really have a global vision. We were trying to know the details of every pieces of the project, without really knowing how to set them together.