End-study project report

Tanguy Benard - Yann Cantais

January 10, 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 Preliminary Research

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 focused on Hyperledger Indy, and 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.

2.1 DIDs

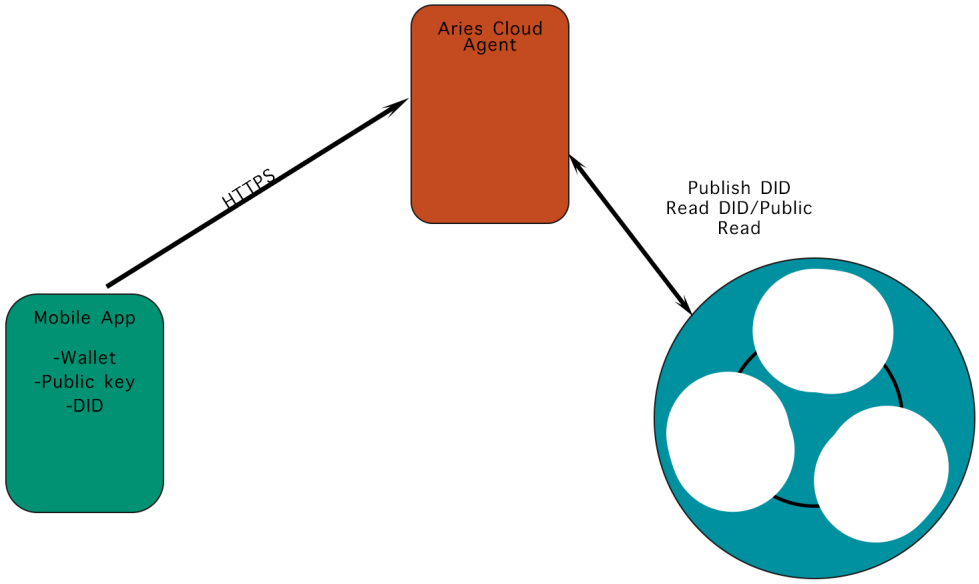
In order for this project to succeed, it is mandatory to give user's control on their identity. The main goal is to be able to communicate with others without having to register through a phone number or a social media account for example. The idea is to keep the messages secured and private.

To achieve this, user's must get an identity wallet and store DIDs in them. Decentralized identifiers (DIDs) are a new type of identifier. Those identifiers enables verifiable, decentralized digital identity.

The controller of a DID is able to prove control over it without requiring permission from any other party. DIDs associate a DID subject with a DID document. This allows trustable interactions, each DID document being able to express cryptographic material, verification methods, or services, which provide a set of mechanisms enabling a DID controller to prove control of the DID.

3 Architecture of the solution

3.1 Overall Architecture



3.2 Indy Pool

The Indy pool is the blockchain network on which every did is written.

3.3 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.

3.4 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.

4 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
```

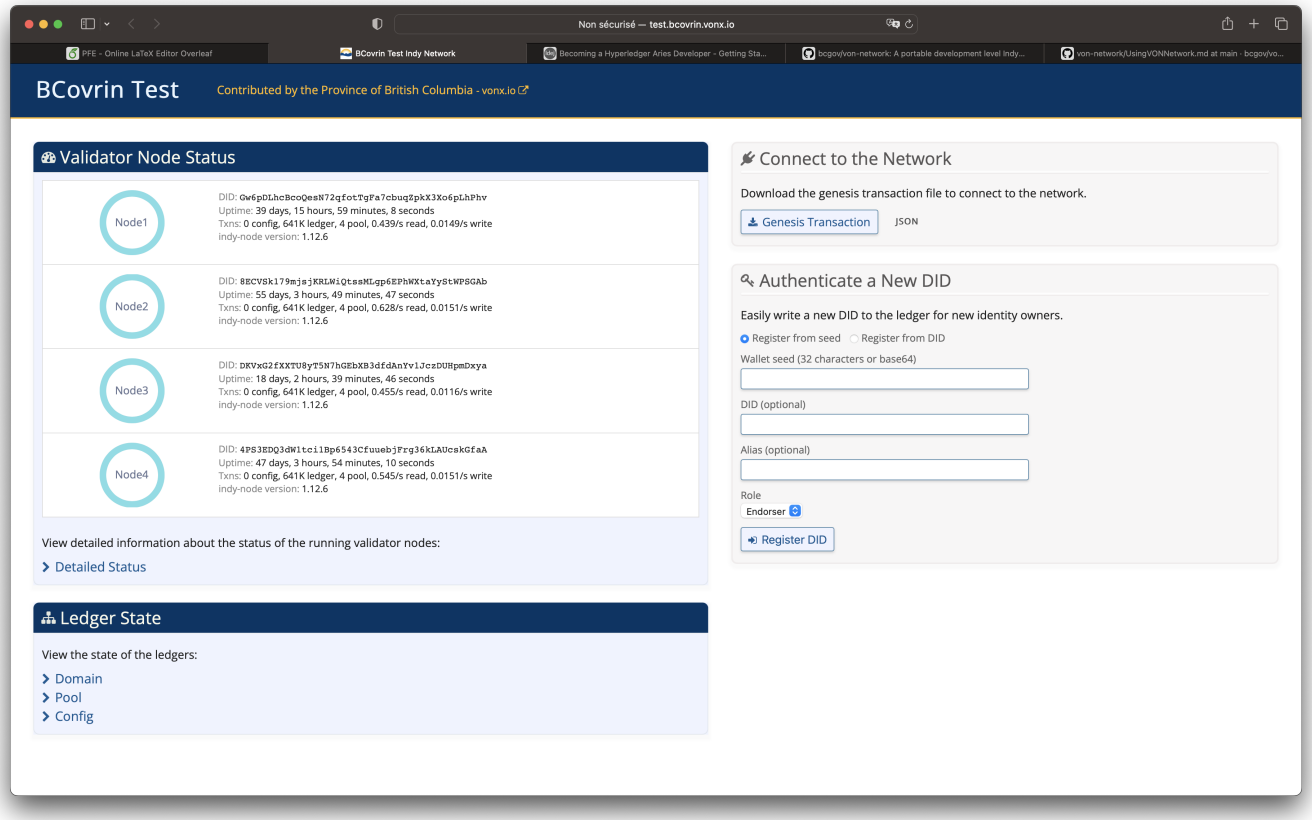
After downloading the repository, the von network can be started with this payload.

```
$./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.



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
```

5 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
```

5.1 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.

6 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.

6.1 The python library used : Kivy

To create a mobile app compatible with the Python Package 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.

6.1.1 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.

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

The 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.

6.1.3 Advanced Concepts (transitions)

6.2 The basic architecture of the application

The different screens of the application are :
In the Home screen, 2 buttons are displayed : Login and Sign Up, pointing to the 2 corresponding screens.
By sending the login and password through the Submit button, the application will try to create a Hyperledger wallet with the method

```
wallet.create_wallet (id = login and key = password)
```

If it succeeds, the wallet is created, opened and the user is logged in. 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 directly logged in. 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). 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.

7 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. This typically involves using digital identifiers and cryptographic keys to verify the identity of the users and establish a secure channel for communication.

7.1 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. For now, we want every user to have his own cloud agent but it may change in the future. It is then mandatory to have two instances 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 Yann00000000000000000000000000000 \
--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: Yann00000000000000000000000000000 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 Tanguy0000000000000000000000000 \
--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.

7.2 Sending an invitation

The next step is to send an invitation to Tanguy with Yann's mobile app. The ***inviter*** is the user that initiates the protocol by sending an invitation message, here Yann. Yann has an invitation tab in his app, when he clicks on the button, he sends an http request to his cloud agent. Here is the Python code that is used in the app.

```
    r = requests.post("http://localhost:11000/out-of-band/create-invitation
", json={"@type": "did:sov:BzCbsNYhMrjHiqZDTUASHg;spec/out-of-band/1.0/
invitation",
                "@id": "c927b4a7-1901-433e-ac3f-16158431fd0a",
                "handshake_protocols": [
                "did:sov:BzCbsNYhMrjHiqZDTUASHg;spec/didexchange/1.0"
                 ],
                 "label": "Yann",
                 "service": [
                 "did:sov:UpFt248WuA5djSFThNjBhq"
                 ]
                 })
        print(r.status_code, r.reason)
        print(r.text[:300] + '...')
```

Listing 3: Sending an invitation

This is a POST request targeting the endpoint **out-of-band/create-invitation** with the data filled in a json format.

- @type : The Initial Connection Key
- @id :
- Routing Keys
- handshake_protocols
- Label
- service

  0160: Connection Protocol

7.2.1 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

Once it is done, we want the App transforms the content of the json used into a qrcode. The idea is to be able to share it physically when you meet a person that you want to contact later. By scanning this qr, the invitee will be able to process a connection between him and the inviter and be able to communicate.

Figure 1: QR code of the invitation Json

7.3 Connection request

Tanguy receives the invitation request by scanning the QR code. He must then accept the connection invitation. He will use the information present in the invitation message to prepare a new request going to the inviter cloud agent, Yann.

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.
    }
  }
}
```

7.4 Verify the digital identifiers

After the users have exchanged their digital identifiers, you will need to implement a process for verifying that the identifiers are valid and belong to the correct users. This typically involves using the indy-sdk or other relevant libraries to query the indy pool and confirm that the digital IDs are registered and associated with the correct users.

7.4.1 Connection Response

After receiving the connection request, ***the inviter*** process it. She checks if the information presented with the key used match.

If she is willing to accept the connection, she will keep the information received in her wallet. The inviter will then send a connection response to the invitee's cloud agent using the newly information 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.

7.5 Generate cryptographic keys

Once the digital identifiers have been verified, you will need to generate cryptographic keys for the users' mobile wallets. This typically involves using the indy-sdk or other relevant libraries to generate a private/public key pair for each user, and then storing the keys securely in the users' mobile wallets.

7.5.1 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".

7.6 Establish a secure channel

After the cryptographic keys have been generated, you can use them to establish a secure channel for communication between the users' mobile wallets. This typically involves using the indy-sdk or other relevant libraries to encrypt and decrypt the messages using the users' keys, and then transmit them over the network.

8 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.

8.1 Basic Message Protocol

Aries RFC 0095: Basic Message Protocol 1.0
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?"
}'
```

8.2 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 the python code 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"
```

8.3 Encode the messages

The first step is to encode the messages that are sent between the users' mobile wallets. 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.

As we want the messages to be kept secret we may have to use another protocol: Aries RFC 0019: Encryption Envelope

There are two layers of messages that combine to enable **interoperable** self-sovereign agent-to-agent communication.

DIDComm Encrypted Envelope are messages sent between identities to establish a connection, issuing a verifiable credential, sharing a chat,etc.

A DIDComm Encrypted Envelope is an envelope around a plaintext message to permit secure sending and routing. A plaintext message going from its sender to its receiver passes through many agents, and an encryption envelope is used for each hop of the journey.

8.4 Decrypt the messages

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.

8.5 Store the messages on the indy pool

Once the messages have been decrypted, you will need to implement a mechanism for storing them securely on the indy pool. This typically involves using the indy-sdk or other relevant libraries to write the messages to the pool's ledger or storage system, using the appropriate permissions and policies.

8.6 Retrieve the messages from the indy pool

Finally, you will need to implement a mechanism for retrieving the messages from the indy pool when the indy pool has received it.