École Polytechnique Fédérale de Lausanne

Anonymous Proof-of-Presence Groups for Messaging and Voting

by Céline Camacho   Gabriel Fleischer   Sébastien Fulpius
Raoul Gerber   Jean-Baptiste Michel   Romain Pugin
Nicolas Raulin   Ouriel Sebbagh   Alexis Tabin
Maxime Würsch

Semester Project Report

Approved by the Examining Committee:

Prof. Bryan Ford
Thesis Advisor

Pierluca Borsò
Thesis Advisor

Louis-Henri Merino
Thesis Supervisor

Haoqian Zhang
Thesis Supervisor

EPFL IC DEDIS
Bâtiment BC
Station 14
CH-1015 Lausanne

January 15, 2021

Abstract

To protect from Sybil attacks, where one person creates many fake identities, communication tools usually require people to provide some sort of identity, like a credit card or a phone number. However, this causes a privacy problem as it links the virtual identity to a physical one in a privacy-invasive way. Weak identities such as e-mails are better in term of privacy, but do not protect against Sybil attacks and do not provide any form of accountability (e.g. one person, one vote). Proof-of-Personhood solves this problem by proving a digital identity is linked to a person, without revealing its identity. This is done by organizing in-person events and giving present people a token proving they attended. In this project, we started to build an application to implement the Proof-of-Personhood approach. The current version allows users to create an organization and run roll-call events. It is not finished yet, and this project is thought to be continued by other students and engineers.

Contents

Chapter 1

Introduction

In January 2021, users of the messaging application WhatsApp received a message informing them that the application was updating its terms and privacy policy. Allowing the application to increase the amount of data it shares with the other Facebook Companies[1]. A short time after that, a steep rise in downloads of Signal, a cross-platform encrypted messaging service, has been observed. This increase could correspond to people choosing a safer app to communicate [2]. Similarly, during the summer of 2020, the release of the SwissCovid application, a contact tracing app for COVID-19, was in the heart of heated discussions, especially because people were reluctant to the idea of being traced [3].

These two examples show that, nowadays, people care more about their privacy. But most of the applications we use link us to a virtual entity in a privacy-invasive way. They ask the user for its phone number in order to create its account and use their services. Even a reputed safe application like Signal does that in order to achieve more accountability and to protect itself against Sybil attack. These applications trade privacy for security. On the other hand, if weak identities are used, like email addresses or pseudonym public keys, one user could easily create multiple identities by, for example, creating multiple email addresses. Thus the application loses any type of accountability and Sybil attack protection.

The difficulty in finding a way of identifying people online lies in achieving both a non privacy-invasive and sufficiently accountable mechanism. Obtaining accountability and Sybil attack resistance is desired, but we also need to take into account the users' privacy. How nice would it be if we could authenticate users while respecting their privacy and ensuring a high security level? This is the main goal of the Proof-of-Personhood, bind the physical and virtual identities of one user in a way that enables accountability and preserves anonymity. The objective is verifying people rather than identifying them by organizing parties and generating tokens. This is similar to a masquerade ball [4], where there is a real-world gathering which still preserves the attendees' anonymity.

The project is hence a real life application of this idea. The goal of the PoP application is to let users create their own local autonomous organizations and others to attend them. Once the LAO has been created, each user can create different events for each LAO, like meetings, polls or even roll-calls, where attendees have to be physically present. The Proof-of-Personhood is achieved by creating said roll-call events, where users must be physically present to prove their existence. With this application we aim to link virtual and physical identities in a non privacy-invasive way and still achieving some accountability and Sybil attack protection.

Chapter 2

Background

In this chapter, we present several technologies and patterns used in the project.

2.1 WebSocket

WebSocket is a communication protocol. It runs on top of TCP and provides a full-duplex channel. The advantage of WebSocket compared to TCP is that WebSocket allows to send messages, where TCP only support a byte stream.

We choose to use WebSocket instead of HTTP because the server needs to send notifications to the client. It is easier to implement a notification system using WebSocket than using HTTP, as WebSocket provides a full-duplex channel where HTTP only provides a half-duplex channel. As a consequence, if we wanted to implement a notification system using HTTP, the HTTP client would need to poll the server, which is less efficient. Choosing WebSocket instead of HTTP does not affect our system's compatibility as WebSocket is a standard supported by many platforms.

2.2 Publish-Subscribe pattern

The publish-subscribe pattern is a messaging pattern where the sender of a message does not have to explicitly specify the receiver. Instead, when sending a message, the sender specifies a channel. We call this action "Publishing on a channel". A receiver can subscribe to a channel. When a sender publishes a message on a channel, any entity subscribed to that channel will receive the message published to it. For example, we created a channel where all messages related to an event are published.

This pattern is useful in our project because the sender of a message does not know in advance to whom it must send a it. The sender can just publish it on a channel, and subscribers will receive it.

2.3 EdDSA signatures

EdDSA is a digital signature scheme based on twisted Edwards curves, a type of elliptic curves. Signing a message lets the receiver check the authenticity of the said message. Given a message, the owner of the private key can sign the message using EdDSA. Given a signature, the corresponding message and the public key, anyone can verify if the signature is correct. Digital signature scheme are used to prove the authenticity of a message.

In this project, we use Ed25519 as our signature scheme. This is a version of EdDSA using Curve25519, a popular elliptic curve.

Chapter 3

Design

3.1 Overview

3.1.1 Local Autonomous Organization

The core of the PoP project is the Local Autonomous Organization (LAO). This is a kind of group used to organize different types of events. Users of the PoP app can have three different roles in a LAO. First, there is the organizer, who is the creator of the LAO and can run events. Then there are witnesses, who attest events are valid. The organizer and witnesses are specified at LAO creation. Finally there are the attendees, who are attending the events of the LAO. We describe in the following subsections the different types of these events.

3.1.2 PoP event (roll-call)

Running the event

A PoP event is a physical event where people can come to get a PoP token. It is designed in such a way that one person can only obtain a single token. People could then use this token on other services to prove they are a real person. Note that although a PoP token proves you are a person, it is not linked to your identity. Thus services can not identify you using your token.

How does a PoP event work in practice? First, the organizer must setup a server that will be used for the event. They can easily create a server running on their smartphone using the PoP application or run it on a dedicated machine. The organizer of the event sets a location and a time for the event. At the given time, participants (called attendees) come to the event. They either need to install the PoP application, or to use the web version. During the event, they can scan a QR code containing the address of the organizer server and connect to it.

Once every attendee is connected to the server, the roll-call can start. The roll call is the part of the event where attendees obtain a proof they were physically present at the event. Once the roll-call starts, no one else can enter the room where it takes place. The smartphone of attendees show a pseudonymous QR code. The organizer must then scan the QR code of all attendees and write them to the server. Once the roll-call ends, everyone has received a PoP token.

Validity of the Pop event

During the event, witnesses are present. Their role is to attest that the event is valid. They verify no one is entering the room where the roll-call takes place after it started and that an attendee does not obtain multiple tokens. They take pictures and videos of the event to prove that it is a real event, not a fake event the organizer is using to create fake Proof-of-Personhood.

3.1.3 Other events

There are other types of events as the roll-call. However, they are at this stage less important, as it is the roll-call that ensures the Proof-of-Personhood principle. Here is a list of the other events the app should implement, even though it is not the case yet.

- **Meeting:** A gathering of people to discuss together. It requires a name and a start time. The organizer can optionally add an end time, a location and metadata.
- **Poll:** A vote to get the opinion of the participants on a question. It can be either *choose one of N* or *approve any of N*. The result of the poll is sent to the attendees once the poll is closed.
- **Discussion:** A moderated discussion where the attendees can participate. The structure of the discussion is not defined yet.

3.2 Architecture

3.2.1 Architecture overview

The architecture in its simplest form is a client-server architecture. It is designed to be usable everywhere and only requires participant to be on the same local network. No Internet connectivity is needed. When running a PoP event, the organizer starts a server. Every attendee client connects to it and maintains a WebSocket (see section 2.1) connection with it.

On top of WebSocket, we use a publish-subscribe pattern (see section 2.2) to communicate. Clients can subscribe to channels and publish messages on them. When a message is published on a channel a client is subscribed to, the latter receives a notification containing the published message.

The server in our architecture has two roles. The first one is routing messages according to the publish-subscribe pattern. More precisely, when a client wants to publish a message on a channel, it sends it to the server. The server then forwards it to every client subscribed to this channel. The second role is logging every published message on a persistent key-value store, to account for transparency. The log is then used to prove the event took place according to the rules.

3.2.2 Witnesses

The role of witnesses is to make the system resilient against malicious users. If at least one witness is honest, then a user can detect if someone sends invalid messages. Witnesses should fulfill their role by verifying that messages are valid and digitally co-signing them if it is the case.

In the future, witnesses will also run a server. Every attendee client will be able to connect to it the same way they connect to the organizer's server (Figure 3.1). Like the organizer server, the witness server records messages for transparency. It will also address connectivity and censorship issues. If an attendee can not send messages to the organizer server, either because of network issues or of a dishonest organizer censoring messages, they can send them to a witness server, which will then forward them to the organizer server.

3.3 Messaging Protocol

Different participating instances - back-ends and front-ends - need to communicate between each other. This communication follows the Publish-Subscribe pattern described previously, but most importantly, it needs to be standardized. This is achieved by defining and agreeing on a remote procedure call (RPC) protocol. A complete specification of the protocol can be found in the *proto-specs* branch of the project's GitHub repository [5].

3.3.1 JSON-RPC 2.0

The chosen RPC protocol is JSON-RPC 2.0 [6]. It is self-described as stateless and light-weight. Messages using JSON-RPC are written in JSON [7] and can be either requests to perform an action - such as subscribing to a channel - or responses announcing success or failure.

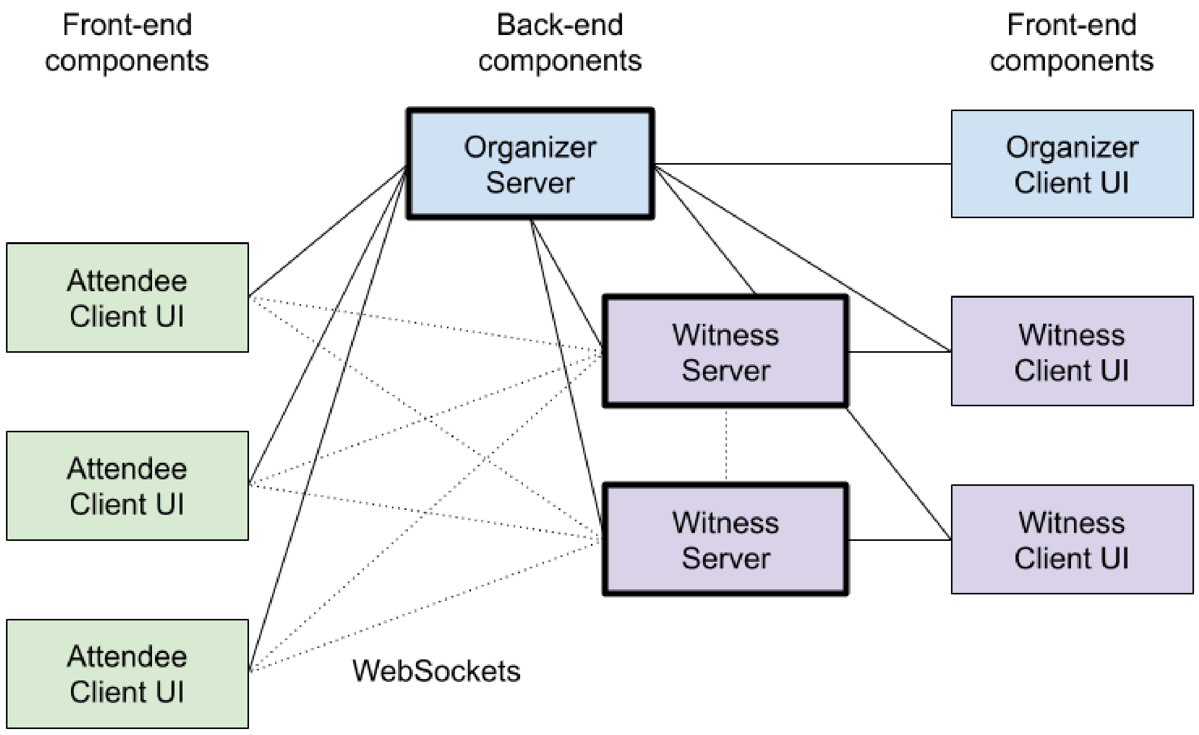

Figure 3.1 – The overall architecture with witnesses. Attendee clients try to maintain a WebSocket connection with the organizer and witnesses servers.

3.3.2 JSON Schema

To standardize the representation of our RPC protocol, we use JSON Schema [8], which allows to specify all three layers described below in JSON, without resorting to examples nor ambiguous language. Instead, with JSON Schema, specifications are written in annotated JSON, expliciting the type of the value expected in each field, potential constants or enumerations and avoiding code duplication by allowing to refer to other JSON Schema files.

3.3.3 Other tools and relevant knowledge

- To sign messages and verify signatures, we are using Ed25519 (section 2.3), an EdDSA signature scheme.
- As JSON does not support binary data, we are first encoding byte arrays in Base64. Note we are only encoding byte arrays (cryptographic keys, signatures and hashes), but not regular strings.
- To generate hashes, we are using SHA-256. As we often hash a combination of items, we need to ensure that it is not possible to game it with tricks like "ab" concatenated with "c" being equivalent to "a" concatenated with "bc". To achieve this we first convert byte arrays to their Base64 string representation and integers to their decimal string representation. Then each string is put between two " and added to a JSON array using compact representation. The hash is then computed on this whole JSON array. It is important to precise that regular strings are not encoded in Base64 for the hash and as such, " and \ characters need to be escaped by adding an additional \ before them.

3.3.4 Request object and layers

Base layer

A request object (also named query in the code) has four core name:value pairs as defined in JSON-RPC 2.0:

- **jsonrpc:** A trivial pair which should always state "2.0".
- **method:** The name of the method to be called remotely. In our case, the five possible values accepted are *subscribe*, *unsubscribe*, *publish* (message outgoing from a sender to a channel), *broadcast* (message outgoing from a channel to its subscribers) and *catchup* (requesting the message history of a channel).

- **params:** The parameters used to perform the action requested by *method*. In our case, it always consists of *channel*, and for the *publish* and *broadcast* methods, it also contains the *message*.
- **id:** The request identifier which allows to match a response to the request.

As a simple example, to subscribe to a channel, one would send a request with *method* = *subscribe* and the relevant channel for the *channel* field of *params*.

Message layer

The *message* field of *params* contains messages sent by *publish* and *broadcast*. It is made of five name:value pairs:

- **data:** The actual content of the message encoded in Base64, detailed below.
- **sender:** The sender's public key encoded in Base64.
- **signature:** The Ed25519 sender's signature on *data*, encoded in Base64.
- **message_id:** The hash of *data* concatenated with *signature*, encoded in Base64.
- **witness_signatures:** An array of JSON objects containing the witness's public key and their Ed25519 signatures on the *message_id*. Each public key, signature and their resulting object are encoded in Base64.

Data layer

The final layer is the data layer. Part of every message, it is the information users want to send. It always starts with the same two name:value pairs, **action:** and **object:** which define together the purpose of the message, such as "create" "LAO" ; "close" "roll_call" ; or "witness" "message". As different purposes require different name:value pairs in this layer, we will not explicit each of them here, but they are defined in the *proto-specs* branch of the GitHub repository [5].

3.3.5 Response object

Every request made with a specified id awaits for a corresponding response. A response object (also named answer in the code) has three core name:value pairs as defined in JSON-RPC 2.0:

- **jsonrpc:** A trivial pair which should always state "2.0".

- **result:** or **error:** Exactly one of those should be present. In case of success, the **result** name:value pair should be used. If the method was *catchup*, then the value should be an array with the message history, otherwise, it should simply state 0 as an ACK. In case of **error**, the name:value pair contains an error object with an integer error code which belongs to the [-5,-1] interval and a short matching description.
- **id:** The same request identifier as in the request which prompted this current response. If the identifier couldn't be decoded, *null* is used in this field.

3.3.6 Create Roll-Call Example

Here is an example of a request which would be sent to announce the creation of a Roll-Call Event. The first JSON shows the base and the message layers, including the Base64 encoded data layer.

```
{
    "jsonrpc":"2.0",
    "method":"publish",
    "params":{
        "channel":"/root/LAO_id",
        "message":{
            "data":"eyJvYmplY3QiOiJyb2xsX2NhbGwiLCJhY3Rpb24iOiJjcmVhdGUiLCJpZCI6Ik
            kyTkhHemx5QlE2czdpVjdYdUtOMTJTOWp2RlRSenpUWFJGVUtHejFVeGs9IiwibmFtZSI6
            Im15X3JvbGxfY2FsbCIsImNyZWF0aW9uIjoxNjEwNzI2NjMyLCJzdGFydCI6MTYxMDcyNz
            YzMiwibG9jYXRpb24iOiJoZXJlIn0=",
            "sender":"4VvizXZE/l77wUmq2e339L9TufzhBT94hofh4Mc/eQ8=",
            "signature":"6Yr4jp835xegEfrKW/UnahycBZvsg0ZsVjugb7Nsl3ymew1ExydJ/gEx4
            UlQRaByM2eU44y91/sPQughezu1Ag==",
            "message_id":"Q1IgsAjomL93rPhQ8iIqm8KXUbMrPr4KfvgPcjTULPw=",
            "witness_signatures":{}
            }
        },
    "id":123
}
```

This second JSON shows the decoded data layer included in the previous JSON.

```
{
    "object":"roll_call",
    "action":"create",
    "id":"9JsRFMjR5Jm7ukWEUlYOEWFwr4fFVcLhrjVwD1DqmuU=",
    "name":"my_roll_call",
    "creation":1610727291,
    "start":1610728291,
    "location":"here"
}
```

The back-end would then proceed to sent a nearly identical message to all subscribers of the channel with only the method being changed to *"broadcast"*. It would as well send the JSON below as a response to the requester.

```
{
    "jsonrpc":"2.0",
    "result":0,
    "id":123
}
```

3.4 Front-end

The front-end design consists of two main user interfaces (UI). The first one is the Home UI. This view manages all actions that are not specific to a LAO. The other UI is used to interact with a specific LAO. This is a general design, there can be minor differences between the implementations of the two front-ends. The design must be seen a goal to achieve. Some parts are not implemented yet by one or the other front-end.

3.4.1 Home UI

The Home UI is the application's initial screen. This is the UI that a user sees when they open the application. It contains a tab bar on the top to navigate between the three screens of this interface. The first one is the Home screen. This screen shows a text explaining the basic usage of the application if the user has not joined any LAO yet. The rest of the time, it displays a list of all LAOs the user created or joined. When the user clicks on one, the app navigates to the Organization UI of this LAO. The figure 3.2 shows the structure of the Home UI.

The next one is the Connect UI. This one manages the connection to a LAO. There are four different steps to connect. The first one asks for permission to use the camera in order to scan the QR code of the LAO. This screen appears only when the application has no access to the camera. Once the permission is obtained, the interface moves on to the next step. This one shows the camera view and enables the user to scan a QR code. If it represents a valid LAO, the Connect UI moves to the third step. This screen shows that the app tries to connect to the scanned LAO.

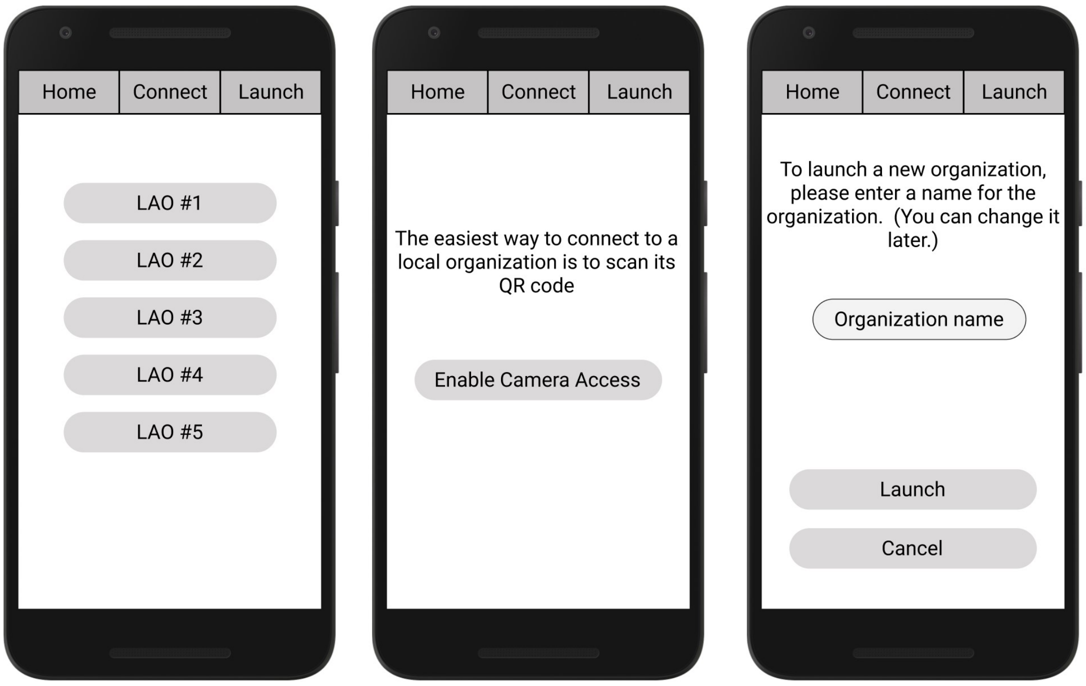
Home
Connect
Launch
LAO #1
LAO #2
LAO #3
LAO #4
LAO #5
The easiest way to connect to a local organization is to scan its QR code
Enable Camera Access
To launch a new organization, please enter a name for the organization. (You can change it later.)
Organization name
Launch
Cancel

Figure 3.2 – The Home UI

The user can stop the process if it takes too long and will be redirected to the scanning step. If the application receives an answer from the LAO, the user is redirected to the last step. This screen shows some information about the LAO (the name, the organizer's fingerprint and the name of each witness along with their fingerprint) as well as buttons to accept or refuse the connection. This connection setup is not a proof-of-presence. Its purpose is to let the user retrieve information about the LAO and to verify that they can access it.

The Launch UI is the last on the Home UI tab bar. This screen allows the user to create and launch a new LAO as its organizer. For now, the creation is very simple. It just asks the user for a name. This screen has been kept simple to avoid feature duplication with the Organization UI. In the future, there may be more requirements to create a LAO and the Launch UI will need to adapt to them.

3.4.2 Organization UI

When the user clicks on a LAO in the Home screen, it opens the Organization UI. As in the Home UI, there is a tab bar on the top to navigate between all the screens with four different sections. The first one redirects to the Home UI. The second one links to the events' list. The information displayed differs with the role of the user, as detailed later. The third one is the Identity UI, and the last one only shows the name of the LAO as seen by the user. The structure of the Organization UI, set on the attendee screen, can be seen in figure 3.3.

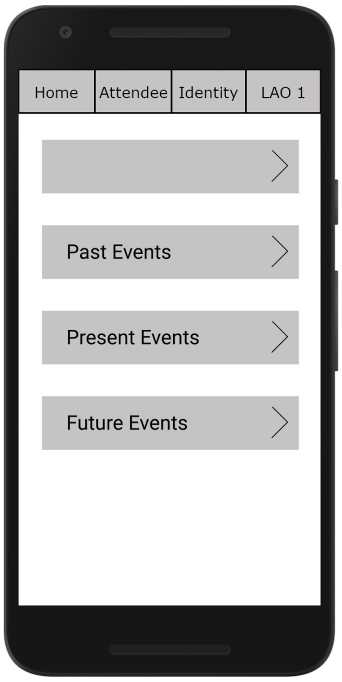

Figure 3.3 – The Organization UI

A LAO contains two types of data: the properties and the events. Properties are time-independent and, for now, there are two different types of them. The first one is the name of the organization. The second one is a list of all the witnesses, with their name and signature. The events are time dependent. Some of them have a specific behaviour in the UI:

- **Roll-call:** It has three different states: *Future*, *Open* and *Closed*. This event does not work the same if you are an organizer, an attendee or a witness. Attendees and witnesses see the start time and the status of the roll-call. When the roll-call is open, a QR code representing them (pseudonymous and roll-call specific) is shown. When the status is *Closed*, the number of participants appears on the screen. For the organizer, the same information, except for the QR code, is displayed. The organizer opens the roll-call by clicking on the corresponding button. A camera view appears to allow the organizer to scan the attendee's QR code and it also displays the number of people already scanned.
- **Poll:** The state of a poll depends on the start time and finish time. The results with progress bar are displayed when the poll is closed.

A user can have one of the three roles: organizer, attendee or witness. As users can have different roles within the LAO, they won't all see the same information. In the next three paragraphs, the main differences are shortly pointed out.

The Attendee UI shows the properties of the LAO and all its events. The screen displays all this information in a list divided into sections. Each of them can be revealed or hidden. The first one contains all the properties of the LAO. The last three show the events in chronological order: one section for past, one for present and one for future events. An event which start time is included between the start and the end time of another event is shown nested in the other. This chronological order is generated by the front-end. For simplicity, all the events in the system are independents. They just contain the required information to create the order.

The structure of the Organizer UI is mainly the same as the Attendee UI. The first difference is that there exists an edit button in the properties section to allow the organizer to change the name of the LAO and to remove or add witnesses. When the user presses the button to add a new witness, the application shows the camera view to scan the witness' QR code. All modifications occur only when the organizer validates the changes. The second difference concerns the events. The organizer sees different information than the attendees, as explained in the description of the events. The last one is a button on the future section to create new events. When the user presses it, he is asked what kind of event he wants to create. The app then displays the appropriate event creation screen.

The Witness UI is very similar to the Attendee UI. Above the list of events, the user sees a button taking them to a screen with the possibility to take pictures and videos to attest that the events took place and that the organizer has not falsified the roll-call neither prevented a person to take part. In the future, this proof will be available to every attendee along with a hash and timestamp to verify its authenticity. Below the button to navigate to the camera view, the application displays the same list as the Attendee UI.

The Identity UI is a form the user can fill with their identity. They can also choose to be anonymous. In this case, all the fields are disabled. If they chose to give their identity, they can provide their name along with other personal information but only the name is mandatory. An identity QR code is shown at the bottom of the screen once the user has set their name. This form is mostly useful for the organizer and witnesses, as they can not remain anonymous. For now, the identity is self-claimed and can be used only in the current LAO.

Chapter 4

Implementation

The project members are divided in four different groups, two working on the two front-end and two working on the back-end. The approach of having multiple teams building the same software in an independent way is called N-version programming. The advantage of this method is to make the software more reliable, as two teams are unlikely to make the same mistakes. The implementation details are explained for each of these groups.

4.1 Back-end 1 in Go

The implementation of the Go language package uses Go 1.15. The instructions to run the server are in the README. The code documentation can be built using the `godoc` command.

4.1.1 Relevant packages

In this section we will talk about the packages that are relevant for the report. Packages defining functions like scanning an array to check if it contains a specific element or packages defining the different message structures as they are defined in the protocol will not be discussed.

Network

As mentioned in subsection 3.2.1, the different instances of the application communicate in a Publish-Subscribe fashion over WebSockets. The *network* package mainly serves this purpose. We chose to use the `gorilla/websocket` [9] package, as it offers more functionalities than the Go standard library package. The library runs a HTTP server and then upgrades the HTTP connection to a WebSocket connection.

In this package, we define a hub. This hub is going to route messages according to the publish-subscribe paradigm. The hub maintains a list of opened WebSocket connections, and closes them automatically if they have more than a 1 second timeout when trying to send them a message. Upon message reception, the hub just forwards it to an actor. There is no checks done on message validity at this step, the actor takes care of that.

Actor

The package *actor* is the package defining the logic behind every actor's action. For the back-end, there are only 2 types of actors: the organizer and the witness. This package implements the actor's reactions to received messages.

To that end, their core function receives an incoming message from the hub and returns the response messages and, if applicable, a broadcast message. To do this, the actor will in turn use the parser package to decode the received JSON string, then the security package to ensure the validity of the data, followed by the database package to store the received message, and finally the parser package again to serialize the messages to be returned. As it learns more about the purpose of the request received, it will distribute the workload to more specific functions.

The features of the witness back-end are expected to evolve in the future, but in the current state, its main use is to store the message history as a back-up and third-party proof. This also implies that the organizer is the only broadcaster of messages for its LAO, so that a received message with the broadcast method is considered invalid by an organizer's back-end server.

Parser

The parser package is primarily focusing on converting JSON strings to Go structures back and forth. Thanks to the `encoding/json` Go library and its `json.Marshal()` and `json.Unmarshal()` functions, the task is simple as long as correct and relevant structures are defined in the *message* package.

`Parser.go` decodes JSON strings into Go `struct`, at each layer of the protocol. As it sometimes is impossible to know beforehand what `struct` type to use, it first parses the JSON string into an intermediate `struct` used to determine what sort of `struct` is required to correctly decode the full message. While `Parser.go` is responsible for checking that the JSON string follows the JSON Schema we defined, it does not have to ensure that the contained information is valid: this is the role of the *security* package.

`Composer.go` is in charge of creating relevant structures and serializing them into messages. It is separated in two cases: Either it must compose a standard response or broadcast a message received with the *publish* method, or it must compose an error response. In both cases the

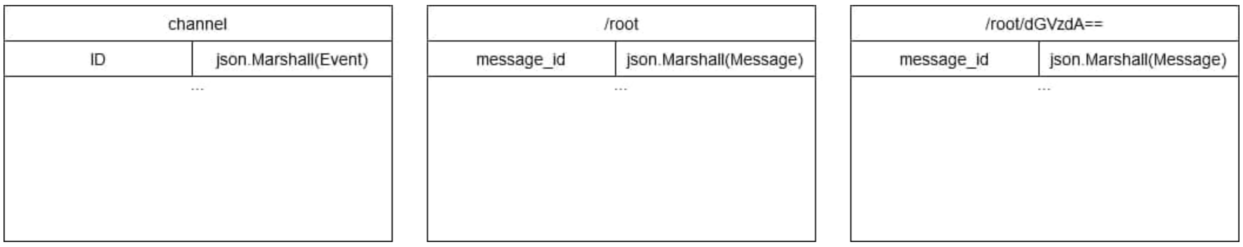

Figure 4.1 – The database structure.

composer will create a message structure, fill it with the wanted information, and parse it with the `json.Marshall()` function into a string before passing it to the hub.

As a side note, the Json-Schema isn't directly incorporated in our system checking the validity of the message's format. This choice was motivated by the fact that `json.Marshall()` and `json.Unmarshall()` already return errors if the received message is invalid. Therefore adding the Schema validation would just make the code slower and more complicated to maintain and understand.

Security

Unsurprisingly, this package defines all the checks to apply on a received message to ensure its validity.

`ValidateMessage.go` is the main file and serves multiple purposes. Upon message reception, the server executes the following verification sequentially. It first checks the *signature* and the *message_id*. This ensures authenticity and integrity as the *data* field of the message is a part of the hash composing the *message_id*. Then, in the case of message containing witness signatures, all of them are also verified and certified to come from people authorized to witness this message.

After having verified the message layer, the embedded data itself is checked according to its type and components. For example, timestamps have to be coherent and to range between defined limits. In the same way, the ids are also verified because all events have a different way to compute it, as explained in the proto-specs branch on Github.

In the particular case of a witness message, which basically contains a witness signature in its data field, the server also verifies that this signature is correct and then adds it in the witness signatures list of the corresponding message in the database.

`SecurityHelper.go` implement the signing and verifying functions using the EdDSA digital signature scheme embedded in the `crypto/ed25519` Go library.

Database

As explained in subsection 3.2.1, every message received on a channel needs to be stored, in a key-value database. For the Go back-end, this means we have key-values pairs, stored inside a container called "bucket". Of course it is possible to have more than one buckets in a single database, and even to have nested buckets. This is then saved locally to the disk. To that end we use the `boltdb/bolt` Go library. Even though it is not maintained anymore, it works well and was simple enough to use to make a first prototype. Before updating it to an equivalent and maintained library, it should be considered to switch to a relational database schema, in order to change the storage engine only once. This change could be motivated by the possibility to index the stored data by another field than the ID. A concrete use would be the ability to return the ten most recent messages faster.

The database schema we developed uses no nested buckets, as it makes the whole process less complex. Everything we store is in the form of a byte array. For each server instance, we create a database with the following structure:

- The bucket "channel": This is a unique bucket that contains one key-value pair for each existing channel, except for the root channel. The key is the channel's ID, and the value is the JSON string representing the current channel's state.
- Channel's bucket : There is one such bucket per channel. The bucket's name is the channel ID and the bucket contains one key-value pair for each message sent on this channel, where the key is the message ID and the value the JSON representation of the message.

4.1.2 Message flow

The figure 4.3 shows how the server reacts upon message reception. The "process content (Database level)" block refers to the process showed in figure 4.2. This flow concerns messages that imply writing to the database. For methods like "catch up", that require reading past inputs, the message flow stays the same. The database flow will read entries from the database, and may return the desired content, *nil*, or an error. If the process encounters no bug, the hub will have to send an array of response to the sender. In some rare cases, it is possible that the received message implies a modification on both a previous message and a channel. If this is the case, we will first edit the previous message, then insert the received message in the database, and finally edit the channel's information.

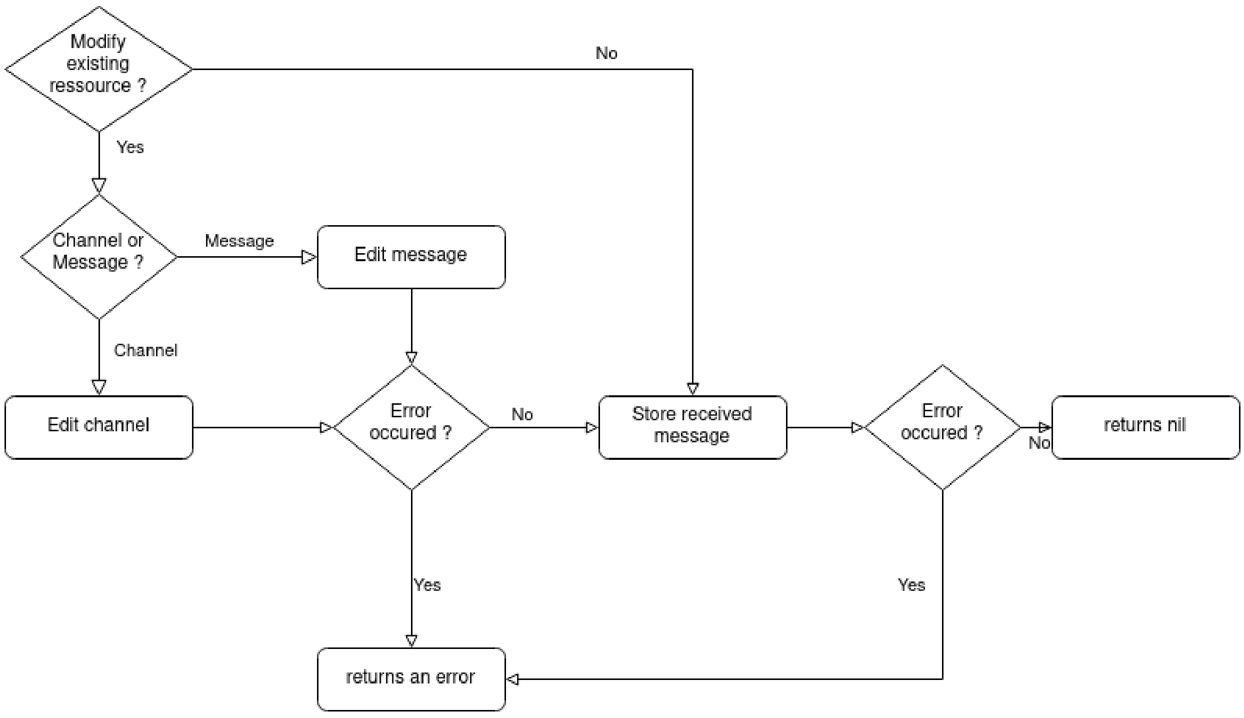

Figure 4.2 – The database flow in the Go back-end.

4.1.3 Debugging and testing

When the server runs, it also runs a simple and minimalist front-end, to do basic debugging. It's contained in the `index.html` file, and is accessible from a web browser on the URI `address:port/test` where `address` and `port` are the address and the port the server is running on. This lets you send text to the back-end over a WebSocket, and prints every received message. We also log every error we encounter using the Go `log` library, as well as success messages upon channel creation.

4.2 Back-end 2 in Scala

The implementation of the back-end in Scala uses Scala 2.13. Instructions to run the server can be found in the README.

4.2.1 JSON parser

The purpose of our JSON parser is separated in two distinct points. Firstly, it should convert any client query into an internal representation of the message which will later be used by the

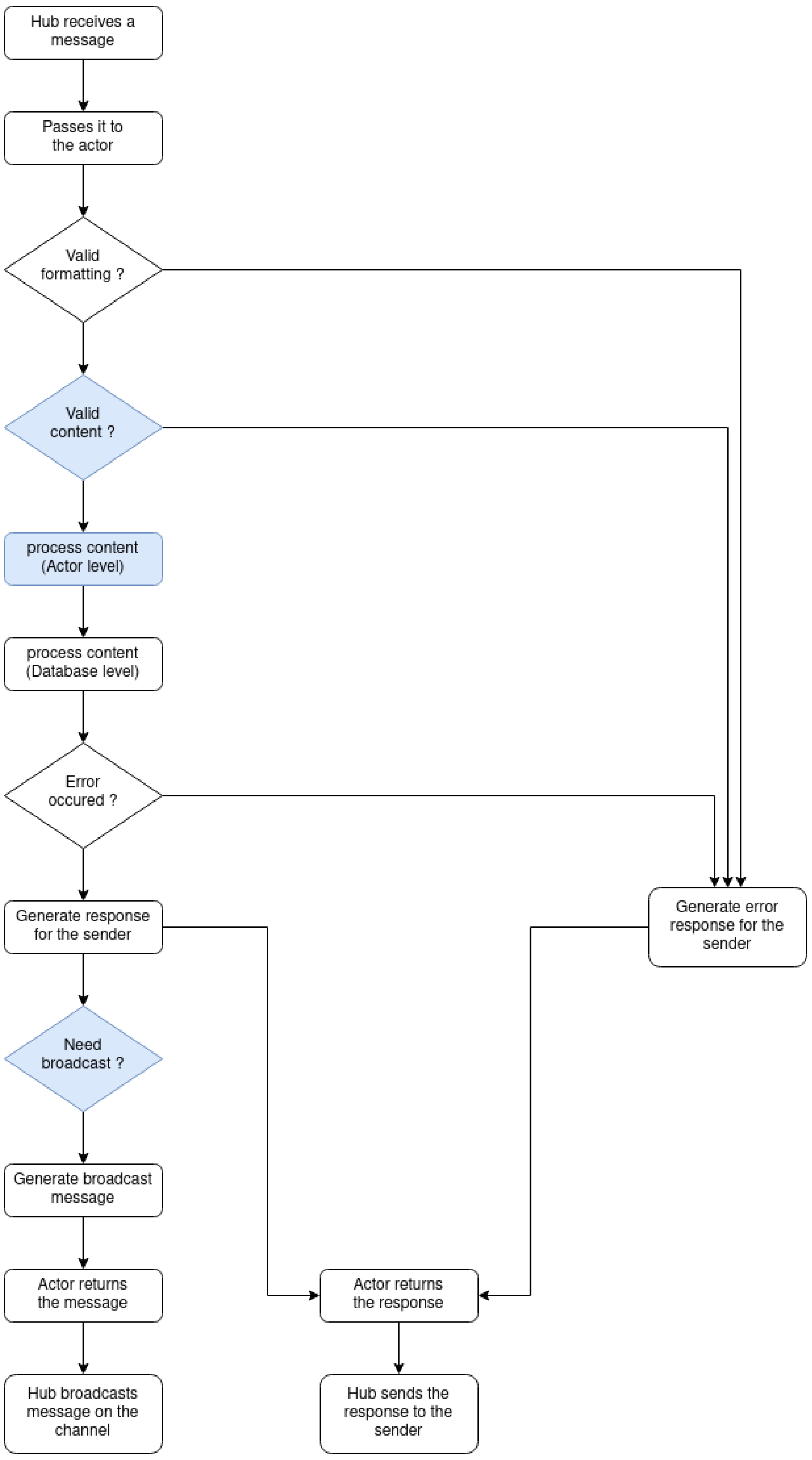
Hub receives a message
Passes it to the actor
Valid formatting ?
Valid content ?
process content (Actor level)
process content (Database level)
Error occured ?
Generate response for the sender
Generate error response for the sender
Need broadcast ?
Generate broadcast message
Actor returns the message
Actor returns the response
Hub broadcasts message on the channel
Hub sends the response to the sender

Figure 4.3 – The message flow in the Go back-end. Blue shapes mean the action or result may differ depending on actor type.

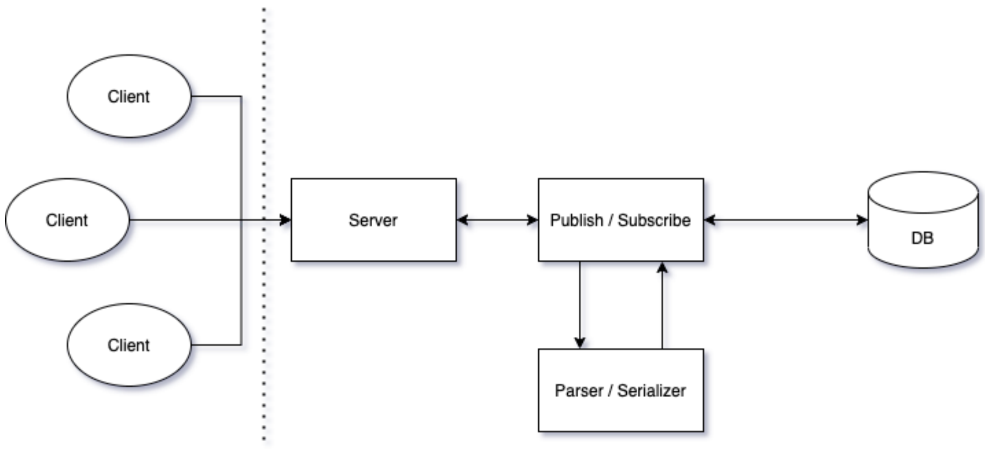

Figure 4.4 – Simplified overview of be2-scala architecture.

publish-subscribe module. Moreover, its second job consists of serializing generated answers to these queries into a JSON string format according to JSON-RPC 2.0 specifications mentioned before.

We started by defining the queries and answers' internal representation as case classes following the tree structure shown in Figure 4.5. First of all, every message is a subtype of `JsonMessage`. This top-level trait branches off in two directions; a `JsonMessageAnswer` is the type of a server answer the parser should serialize in order to be sent over the network. On the other side, every client query is a subtype of `JsonMessagePubSub`. Finally, each publish query such as "create a LAO" or "open a roll-call" is converted to an instance of the case classes that extend the `JsonMessagePublish` class.

We then had to choose a parsing library. In our case, we decided to go for spray-json for two main reasons: its API is simple and powerful and is adapted for a JSON-RPC structure. Spray-json both encodes and decodes layer-by-layer which is perfect in order to handle JSON-RPC messages. Moreover, this library is able to parse and serialize case classes automatically provided a single line of code; this is why we prioritised defining query components as case classes over other data structures. Some complex components require other data structures such as a standard class or an enumeration. In this case we define the serialization and parsing procedure explicitly ourselves in the `JsonCommunicationProtocol.scala`.

If the parser succeeds, it returns the serialized or parsed message back to the publish-subscribe module; otherwise, it instead sends a custom message containing a custom error.

4.2.2 Networking and WebSocket

The server only needs to interact with clients using WebSocket. As WebSocket is not standard in Scala, we choose to use the akka-http library, which is based on akka-stream. When creating a

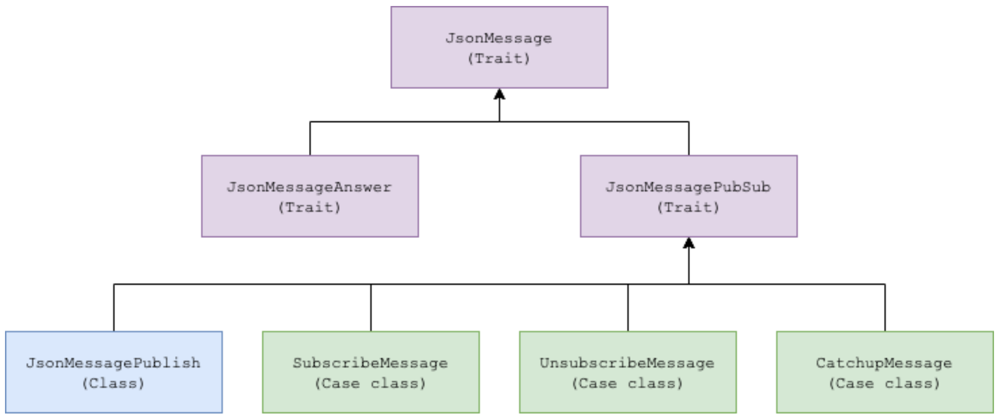

Figure 4.5 – The background depends on the data structure: a purple background stands for a `Trait`, a blue background represents a `Class` and green rectangles are used for children `Case classes`.

WebSocket server, we need to pass it a function building a stream object. When a client connects to the WebSocket server, the server creates a new stream for this client using the function. The stream object receives messages from the client as input and then outputs messages that will be sent back to the client. We explain how we create this flow object in the following section.

4.2.3 Publish-Subscribe

As our data flow is complex, we cannot use simple streams provided by akka-stream. Instead, we use the Graph DSL functionality of akka-stream, which allows us to build arbitrarily complex streams as if we were drawing a graph. The stream we built works as follows: when the stream receives a message, it will first try to parse it using the JSON parser. If there is an error, it will send back an error message to the client. If it is not the case, it will pass it to the publish-subscribe module.

The publish-subscribe module is also a stream object built using Graph DSL. It handles all the RPC defined in the protocol. It receives as input parsed JSON messages, and outputs answers and messages published on a channel. A high-level view of the dataflow is given in Figure 4.6. As most of the diagram is easy to understand, we will focus on more complex parts.

We use two actors common to all clients in our implementation. The Database Actor stores information about the database and handles requests related to it. The Channel Actor does the same for information about channels. The advantage of using actors is their strong integration with akka-stream and that it makes concurrency easy to handle.

When the server receives a subscribe request, it will ask the Channel Actor for a reference to a stream containing messages for a specific channel. This stream corresponds as a part of the hub in the diagram. Then it will connect the output of this stream to the client output stream. Finally,

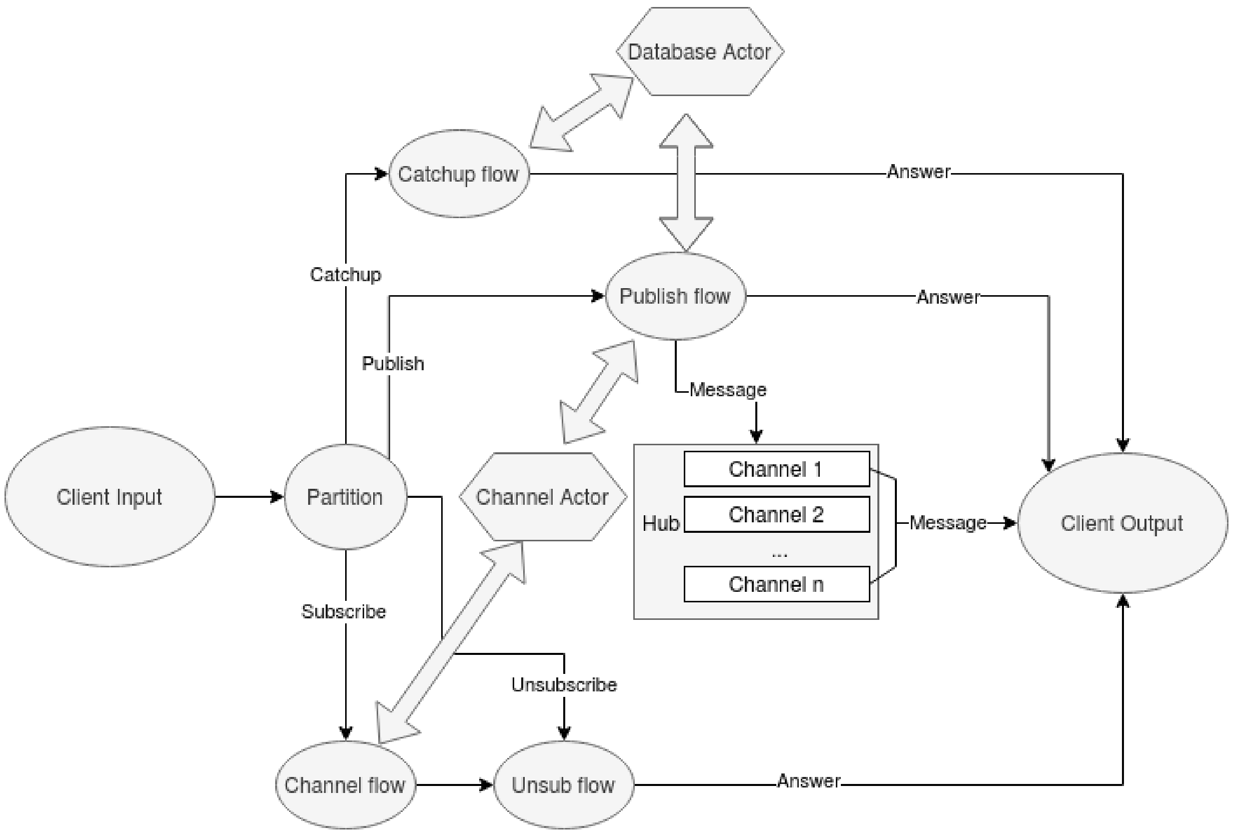

Figure 4.6 – The data flow of the publish-subscribe implementation for be2. Big arrows indicates communication with actors. Streams handling the request of each client use the same instance of the hub, the Channel Actor and Database Actor. In this example, the client subscribed to *channel 1* and *channel n*.

the server stores information needed to disconnect the stream in the case of an unsubscribe request (Unsub flow in the diagram).

When the server receives a publish request, there are multiple steps. First, it checks if the message is valid. If it is valid, it writes it to the persistent key-value store. In some special cases, it does special actions such as creating a channel. Finally, it sends the message to the hub and sends an answer to the client.

4.2.4 Unit tests

We run a set of unit tests using scalatest to verify our code is working according to the specs. For the publish-subscribe module, our tests works as follows: we have a list of messages corresponding to a real use cases. For example creating a LAO and then creating a meeting event. The testing code sends a message to the publish-subscribe and waits for an answer. Once it receives an answer, it sends another message. We repeat this step until we sent all messages. Then we verify if the received messages correspond to the expected answers. On the whole back-end 2, we achieve a line coverage of 84%.

4.3 Front-end 1 in React native

This project is implemented with react-native 0.63 [10]. We have chosen to only support the web version at the end of the semester because we faced some compatibility issues with the storage and the cryptography library between the app on native platform and the web application. To build the project, you need Node.js [11] in version 10 or later. It can be installed on any platform. The detailed information to compile the project is in the READ.ME [12]. We use the coding convention provided by Airbnb [13]. To enforce it, we use ESLint [13]. The list of all external packages used in this project can be found in the `package.json` file in the fe1-web branch [12].

To ease the development process, we build our code with expo. With it we can easily run on any IOS or Android device by just installing the expo app [14]. It is also possible to run the application in a browser. This choice makes the use of all native components inaccessible, for example the camera. With only one command you can switch to a react-native CLI app and get access to the camera and the other native components. This action is irreversible and then requires to compile the code with Android Studio for an android phone or Xcode for an IOS device.

4.3.1 Navigation

The front-end uses react-navigation's library [15] to allow the user to move between screens of the application. We chose this library as it uses the native behaviour of the navigation on both Android and IOS. We use two types of navigators: the tab bar navigator and the stack navigator. In total there are six different navigators. The Home and the Organization UI both use a tab bar navigator. One stack navigator to switch between the Home UI and the Organization UI. A second one that implements the navigator of the Connect UI. Another navigator lets the witness access the camera view to allow him to take pictures and videos. The last one shows the Organizer UI, the roll-call scanning screen or the event's creation screen for the organizer.

4.3.2 Storage

By default, react-native provides only non-persistent storage and the data is a one-way flow from the parent to a child. It is sometimes complex when you need to give data to a deeply nested child. Some data needs to be stored on the device to be reused later. For example, when the user reopens the application, they must see the list of LAOs they previously connected to. So a global persistent storage system was needed to simplify the management of the data in the application.

We chose to use Redux [16] with the addition of redux-persist [17] as global persistent storage. Redux is perfectly incorporated in react native. The library works as follow. There is a store which contains the data. Then you need to connect at least one reducer to this store. A reducer is a function that receives a state and an action and returns a new state. It is used to modify the data. You need one reducer by type of data that you store.

We use five different reducers. One to store the list of LAOs the user previously connected to. Another one for the events of the current LAO. One for the current LAO data shown in the organization UI. We also store the user's private and public keys with a reducer and the last one stores the id of the current open roll-call. The reducer managing the events is quite complex because it needs to order the events in the correct section and to manage the nested events.

4.3.3 Networking and WebSocket

In order to initiate queries such as "create a LAO" or "open a roll-call" to a given server, our app makes use of a set of functions defined in `WebsocketApi.js`. Given a minimal amount of parameters, each of those functions creates the corresponding query, embeds it according to the custom JSON-RPC protocol, and forwards the message to the entity responsible for packet transmission: the WebSocket link.

We used two different libraries for the security related operations. Our hashing library is js-sha256 which simply allows us to hash a string using SHA-256. To generate key pairs and sign

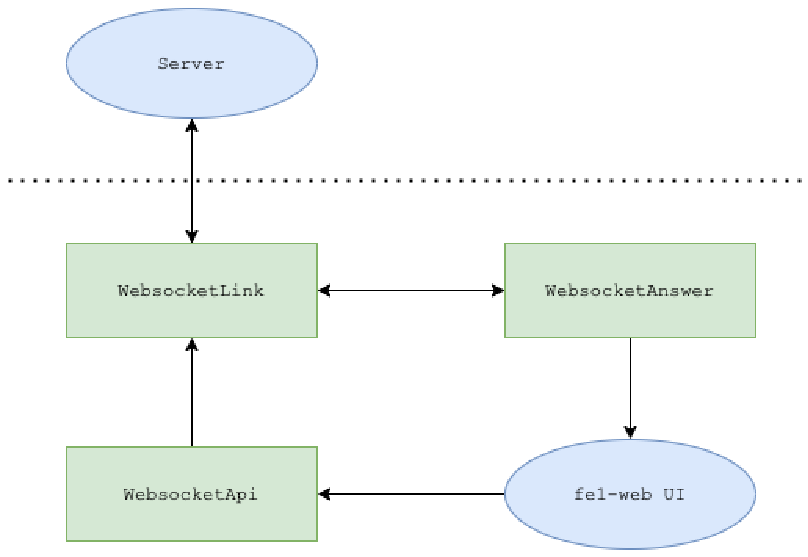

Figure 4.7 – Overview of the client-server communication mechanism

strings, we chose the npm package tweetnacl for its popularity, and because few other libraries support ed25519. Tweetnacl is one of the two packages that prevents us from using the app on native platforms. There exists an alternative named tweet-nacl-react-native-expo, but it does not work on browsers. The second problematic package is related to the browser-only local storage. A solution could be using the asynchronous storage available on all platforms. However, changing from synchronous to asynchronous behaviour so late in the project would require a total rework of the WebSocket section.

`WebsocketLink.js` provides a class that manages network communication, including setting up the WebSocket connection, sending messages over the network and forwarding incoming answers to the suitable entity. We decided to use the W3C WebSocket API for browsers to establish WebSocket connections.

Any new message the link receives is directly transferred to the `WebsocketAnswer.js` module which first checks that said message is valid according to the JSON protocol and then processes the content depending on the type of the query. For instance, when handling a positive server answer to a client query asking for the creation of a new LAO, the module will start by adding the newly created LAO to the list of known LAOs in the local storage, and will then order the UI to display it on the screen as an organizer.

The above hierarchy is represented in Figure 4.7

4.3.4 Date and time picker

For the date and time picker, two different libraries are used. One works only on IOS and Android [18] and the other is for the web application [19]. In the second one, the user can directly set the time and the date. In the native one, the user has to give the date first and then a second picker is shown for the time.

4.3.5 Testing

To verify the type and the structure of objects, we use the prop-types library [20]. It allows us to perform some checks on the validity of the object at runtime. This is useful to have more confidence in the data we receive and helps us in debugging. This library is only used in development.

For unit testing, we used Jest [21] because it was created by the same community as React Native. In addition Jest allows to take snapshots of components to verify that they do not change, eases the testing of Redux and provides mock interfaces if needed. We also created the file `LocalMockServer` which may be used for testing purposes to simulate a local back-end for our application to connect to.

4.4 Front-end 2 in Android

This project is written in Java. All the information to compile the project is found in the README.md [22]. The files are organized in three main packages:

- **Student20_pop:** contains the classes that compose the project.
  - **PoPApplication** is the base class within the Android application. It is the main object that centralizes all other components.
  - **Activities** decide which UI needs to be displayed to the user.
  - **UI** contains all the classes needed to represent the different UIs. All classes inside this package are called fragments.
  - **Model** contains classes used to model the different entities: Lao, Event, Person, Election, Vote, Keys, etc. But also the publish/subscribe communication.
  - **Utility** contains the classes used to represent various utilities like QR code scanning, JSON parsing, network connection, security and the UI utilities.
- **Android Test:** contains the tests that are executed on an android device. This device can be emulated.
- **Unit Test:** contains the tests that can be run without an android device.

4.4.1 PoP application

This class is used to maintain a global application state. It is instantiated before any other class when the process for the package is created. Each instance of the application corresponds to a unique user with their corresponding LAOs. It dispatches every updates of its state to the activities and fragments.

4.4.2 Activities

Activities are the center of control of our application. It is there that we choose which fragment should be displayed.

Main activity

Launched at the start of the app, the main activity shows the home fragment. This activity is used when the user is not yet connected to any LAO, when they want to connect to a new LAO (through the connect fragment) or when they want to create a new LAO.

Organizer activity

Launched when a user clicks on a LAO in the home UI of which they are the organizer or when they create a new LAO (through the launch fragment). The organizer's activity is the most complex one as it handles addition of witnesses and creation of events.

Attendee activity

Launched when a user clicks on a LAO (in the home UI) in which they are attendee or when they connect to a new LAO (through the connect fragment). It displays the different UIs of the attendees.

4.4.3 Fragments

Fragments are used to manage the UIs described in the Design Front-end part (section 3.4).

Main fragments

Home, connect and launch fragments compose the main navigation part of the application. `HomeFragment` displays the list of LAOs the user takes part to. By clicking on a LAO, the user opens the organizer or attendee fragment, depending of their role in this LAO. Their role in each LAO is indicated under the title. The user's public key is compared with the LAO organizer's key. If there's a match, the user is the organizer, otherwise an attendee. `ConnectFragment` simply shows the QR code scanning fragment to let the user connect to a new LAO. We decided to define `LaunchFragment` to create new LAOs and, as its name suggests, to launch them.

Organizer and attendee fragments

`OrganizerFragment` and `AttendeeFragment` are very similar: they are both related to a specific LAO,they both show its properties and display past, present and future events. In addition to that, they both have a button to access the user's identity within that LAO.

The difference resides in the fact that an organizer can edit the LAO's properties, add witnesses and create new events, what attendees can not do.

Identity fragment

`IdentityFragment` is closely related to the LAO the user is currently seeing. For example, a user could want to be anonymous in a LAO and not in another. This means that user's information are not the same for every LAO.

This fragment displays information such as name, email, phone, etc. It also displays a QR code for an organizer to scan in order to add that user to a roll-call event. This QR code currently contains a concatenation of the user's private key and the LAO id but this is subject to change.

Event creation fragments

For each of the event types, we implemented a creation fragment. Each of these fragments extends from an abstract class as events have multiple features in common, like a start date, a start time, etc. These fragments allow the user to create an event with ease.

QR code scanning

When the user needs to scan a QR code, the app shows either the `QRCodeScanningFragment` if it has the camera permission, or the `CameraPermissionFragment` if the permission is not granted yet. Once the user grants the permission, they are redirected to the `QRCodeScanningFragment`.

The user needs to scan QR codes to join a LAO, add a witness to a LAO or add attendees in a roll-call event.

The QR code reading is based on the Google Mobile Vision framework [23] and the classes used in this process are located in the packages `ui.qrcode` and `utility.qrcode`.

Pickers

To let the users easily choose a date (or time respectively) when creating an event, we implemented date and time pickers. If the user does not select a date or a time, the current date and time will be used as default values. This is the reason why we did not add an explicit "now" button as it is mentioned in the UI specification. By doing this, the UI stays simpler and easier to use.

4.4.4 Models

We created classes to model the different events, a local autonomous organization (LAO), a set of public and private keys, a person, a vote and an election. Some of these classes are not used yet as the protocol is still a work in progress.

4.4.5 Protocol models

The package `model.network` contains classes used to model the protocol messages as java objects. It is split into three categories based on the protocol definition: Base Layer, Message Layer and Data Layer.

The base layer objects are located in the packages `method` and `answer`. The message layer objects are located in the `method.message` package. And the data layer objects are in the `method.message.data` package.

Base layer

Based on the protocol, we created five objects, one for each message. To modularize the code, we added two more super classes, Message and Query.

Message defines the behavior of a base layer message and Query defines the behavior of a message expecting an answer and thus sending a unique id.

In the protocol of the base layer, the field `method` is used to know which object is sent and received. To easily map the values with their respective class types, we created the enumeration `Method` which contains every possible method values and their types.

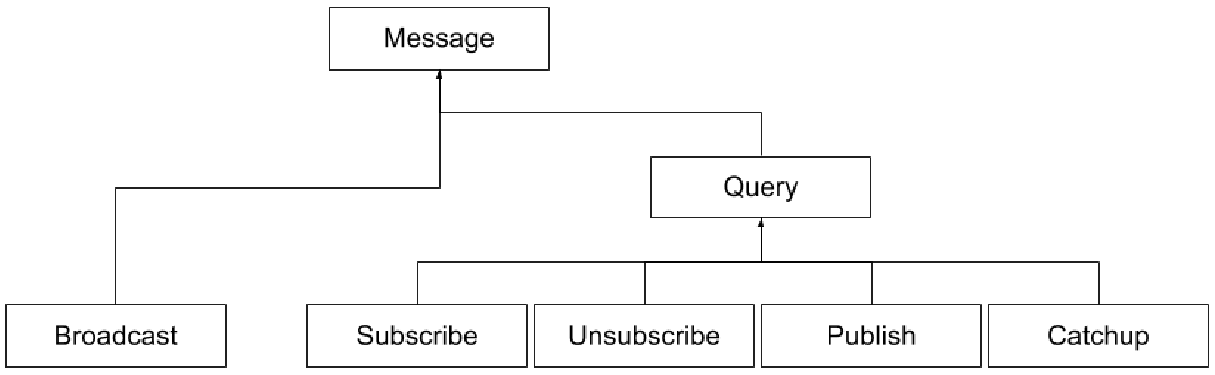

Figure 4.8 – The classes diagram

Answers

We defined two main objects: `Error` and `Result`. They are both children of the class `Answer`. We also created the class `ErrorCode` to represent the error field of the `Error` object. It contains the error code and its description.

Message layer

The message layer contains one object: `MessageGeneral`. It is held by the `Publish` and `Broadcast` base layer messages and holds `Data` messages. It links the base layer and the data layer as stated by the protocol.

Data layer

Based on the protocol, we created ten objects. Three for the LAO, two for the meeting, one for witnessing and four for the roll-call. Each object extends the `Data` class which defines the abstract methods `getObject()` and `getActions()` used to fill the respective JSON fields of the protocol.

The static map messages is used to convert the pairs (Object, Action) to the actual object classes.

4.4.6 JSON parsing

The JSON parsing is based on Google's Gson library [24]. We chose it mostly for convenience as it is a library we had already worked with. But after achieving the JSON parsing, it is now obvious that it was not the best choice.

The two main problems are that there is no clean way to differentiate which class should be parsed based on the fields present in the JSON object and that there is also no way to mark a field as required or optional.

To solve these issues, we had to create custom JSON serializers that are located in : `utility.json`

As the protocol is defined in a JSON Schema, we created tests to validate the JSON generated by the app with the schema. To achieve that, we used NetworkNT's json-schema-validator library [25].

4.4.7 Network

WebSocket connection

The WebSocket connection is built using the Project Tyrus [26]. It generates a client (or server) endpoint with an annotated class. It makes the code very easy to read and write.

Parallelization

As WebSocket queries should not be made on the main thread, we chose to use `CompletableFutures` from `java.util.concurrent` to handle parallel and blocking computations.

Requests

Some of the sent messages expect answers. So, as the protocol defines it, the app generates a unique id for each of them, using an `AtomicInteger`. It then stores the requests while waiting for the correct answer.

To avoid any memory leak due to lost messages, we implemented a timeout system to make sure requests could not be stored forever.

The app calls a routine periodically that iterates over each pending request to check if their waiting time does not exceed the threshold. As the threshold and the routine's period are the same in the app, let's call it T, a request can timeout between T and $2T$ seconds, which is not very precise. But this is not crucial and it is a lot lighter this way.

4.4.8 Testing

For standard testings, we are using the well known library that is JUnit [27] and a small project called ConcurrentUnit [28] which is very helpful to test concurrent behavior.

For Android tests, they run on an Android device or emulator. We use the Espresso library [29] that allows us to emulate human interactions with the interface.

Chapter 5

Evaluation

5.1 Cross testing

Now that we described our implementations, we can now explain our testing strategies. Given that the work was done by four groups, two front-ends and two back-ends, cross testing was very important. To ensure compatibility, we tested every back-end with every front-end. We decided to test the LAO creation and roll-call event creation.

5.1.1 Major bugs found

As the LAO creation was the first implemented feature, it was also the first to be tested. And therefore the one that made us discover most bugs. Many of these bugs came from the fact that even though we had a precise protocol definition, every team had a different interpretation and implementation of it.

This impacted mostly the message ID and the LAO or event ID, as they are computed by hashing concatenated strings. However, it took us a long time to agree on a concatenation standard, and to implement it correctly in every project.

5.2 Future work

There are obviously some improvements, enhancements, changes and new features to be added in such a project. Cross-testing between the front-ends and back-ends as well as usability testing are an important part of it. From user interface details to high-level features that will make the app achieve its inherent goals, this part describes both short-term and long-term work that has not been covered by our team, but will be required or desired in the future.

5.2.1 Events

There are many high-level objectives that the app aims to achieve. The first one is to support all the event types. For now, the only event supported by both the front-ends and the back-ends of the app is the roll-call, which is the heart of the proof-of-presence concept. In the future, the app should also support the other types of events. The meeting event, which is called by the organizer to make sure the attendees gather to a defined place. Poll events will offer the possibility to vote on various subjects. At first, this feature will broadcast votes in clear text but should, as a later extension, use cryptography to enforce privacy. Finally, discussion events will support messaging in a gossip-based fashion. Ideally, this feature will support blocking of other attendees to avoid spam and denial-of-service attacks in a way that encourages accountability and forgiveness. All these events require protocol specifications, as well as models to be stored in the back-end and a way to be created in the user-interface for the organizer, and displayed on the attendee's side.

5.2.2 Other high level objectives

In the long term, a desirable feature could be to support multi-organizer LAOs. As other alternative or extension, the back-end could be enhanced to produce classic blockchains and having it mirrored by witnesses to make it more transparent. Another alternative will be to make attendees automatically become witnesses when they chose not to be anonymous, making sure that the volunteer witnesses are distinct from the official witnesses chosen by the organizer.

It should also be mentioned that we plan on having a third front-end implementation for iOS in Swift.

5.2.3 Low level

A lot of lower-level enhancements and features can be added to improve the app's usability, and to make it usable both as a minimal viable product and as an app used by the public. There are some desirable features that have not been implemented this semester that could improve the app in the short term. For example, there should be a way to join a LAO by entering an URI instead of scanning a QR code. This feature is particularly useful for the web-based front-end, as not all front-end devices have a camera. Less urgent features include, but are not limited to, a way to set reminders for a particular event, the support of push notifications, a way to hide or delete a LAO from the home screen, and more personalized ways to display a LAO with icons or status messages. An important improvement will be to avoid name collision between LAOs, as this could represent an attack. Some of these UI changes will have an impact on the back-end and the protocol, which will have to be updated accordingly.

5.2.4 Testing

Testing constitutes an essential part of future work. Cross-testing should continue to be regularly conducted to make sure every new step in the project works between every front-end and back-end. As this app is aimed to have users, it is important to think about usability testing, giving people who have not worked on this project a prototype of the app and gathering their feedback. There should be a mock Proof-of-Presence event to test the app in real-life conditions.

Chapter 6

Related Work

6.1 A question of privacy and democracy

This idea of verifying people, rather than identifying them is not at its birth stage. In fact, in a previous work [30], the DEDIS lab came up with a more complex version of PoP to bind physical entities to virtual identities in a way that enables accountability while preserving anonymity.

Here, we touch a much bigger problem than privacy, and everyone is concerned as the world is relentlessly going more and more digital.

6.1.1 Redemocratizing permissionless cryptocurrencies

In the previous version of Pop, the goal of the team was more blockchain oriented. Indeed, permissionless blockchain-based cryptocurrencies commonly use Proof-of-Work (PoW) or Proof-of-Stake (PoS) to ensure their security. These two protocols have some advantages, e.g. they prevent double spending attacks but they also have drawbacks: PoW is a huge (wasted) electricity consumer and PoS is a kind of shareholder corporation where people who possess more assets are able to mint new coins faster than less-privileged participant. As a consequence, the rich become even richer.

The problem to tackle was Sybil attack resistance, but also to ensure a fair and widely accessible wealth creation process. Thus they created a new cryptocurrency with a PoP protocol a bit different from ours. The idea was to link virtual and physical identities in a *real-world event* while preserving users' anonymity. At the party every attendee is issued one and only one *Proof-of-Personhood* token, without needing to disclose any identifying information.

To understand the differences between this protocol and ours, we will explain how it works.

Assumptions and Threat Model

The basis of the protocol was also the pseudonym party which was organized by a set of volunteer organizers. Each organizer was an independent person in charge of an independent server denoted as a *conode*, which together form a collective authority or *cothority*. They assumed an any trust model in which attendees need to trust only one organizer and their conode. As we will explain, the security was ensured by a protocol named *Collective Signing* (Cosi) which enables an authority to request the validation of a statement by a decentralized group of witnesses. The resulting collective signatures are comparable in size and verification cost to individual signatures and can be correct only if every organizer co-signed.

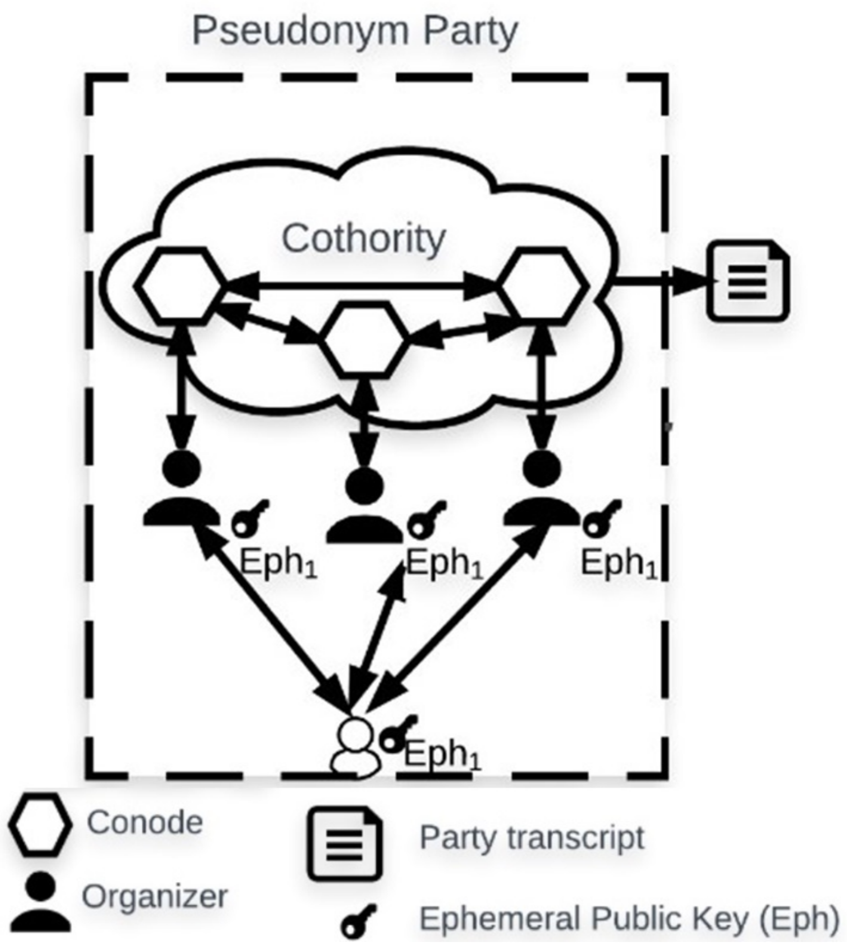

Figure 6.1 – Configuration of a party

The reason why people don't need to trust other conodes relies on the peer to peer network of the cothority. Indeed, at the end of the party, the organizers send the collected data (that should be the same for every one) to their respective conode. This data is then internally broadcasted inside the cothority, ensuring that every conode received all the information. Finally, as we can see in the figure 6.3 the token issued for each participant will be partly generated by a transcript containing all the conodes' signatures.

Party flow

First the organizer decides all the details about the party; e.g. the location, the start time, a set of observers who will record the party. Then they write them out on a configuration file and make it public for the people willing to join the party.

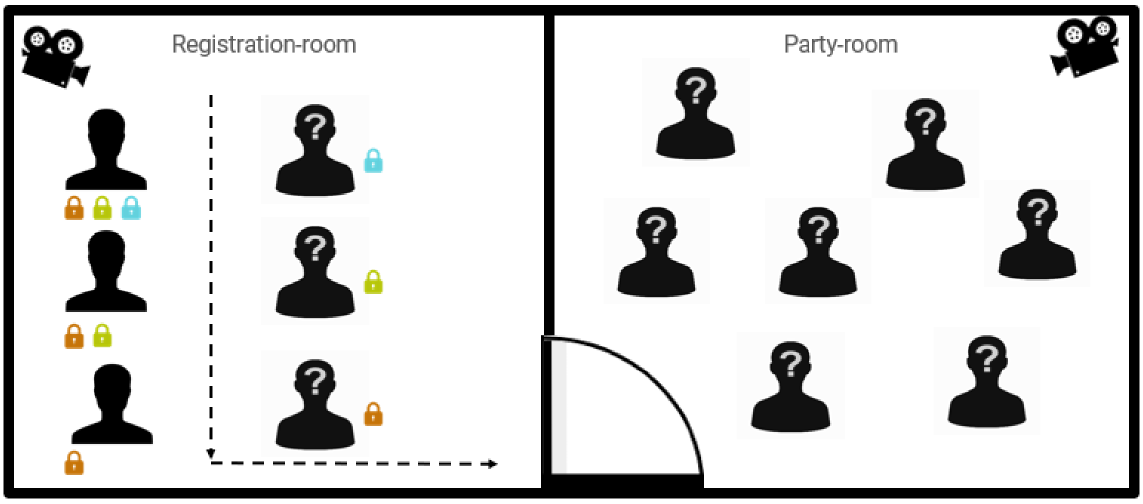

Figure 6.2 – Configuration of a party

Before their entry in the registration room each participant receives a set of three keys, one public key and one private key, not to be used at the party and an ephemeral public key to be used only at the party.

During the party, the whole process is really similar to what we did. There are two steps:

- A registration phase during which each participant is anonymously checked by all the organizers collecting the attendee's ephemeral public key.
- The party right after, where there might be an actual party or something incentive for people to go there.

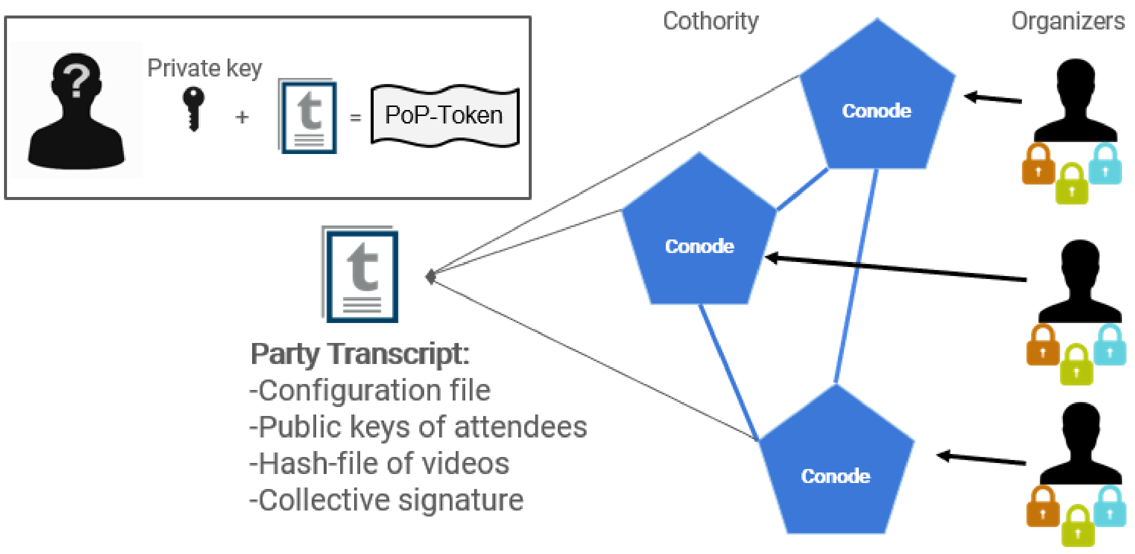

Figure 6.3 – Delivery of tokens

By the end, the organizers upload all the ephemeral public keys collected at the party to their respective conodes. With them, a final transcript is created and a collective signature by the cothority is inserted inside as mentioned before. Each attendee finally ends up with a PoP-Token, generated by their private key and the party transcript.

This protocol was initially designed for PoPCoin, a newly created cryptocurrency that was supposed to lead to a continuously fair and democratic wealth creation process paving the way for an experimental basic income infrastructure. It was an ambitious project and, as said earlier, what we did is based on a different approach. That is, offering this possibility to have a reliable and secure system in any kind of organization, anywhere in the world and with the minimum dependencies on external resources.

6.1.2 New perspective of democracy

The idea of democracy in a world constantly evolving is not something uniformly established. In fact, even the definition of democracy is subject to debate. For example, The Council Of Europe defined the idea of democracy with two key principles that are *individual autonomy* and *equality* whereas Robert Dahl has a different approach that we will explain later.

Concerning the work that we made, one can see the links with this strong concept through the possible uses of this application.

To have a glimpse of it, one can observe that the actual system that we have many democratic countries is based on hierarchy. That is, everything needs to be approved by an authority established (in general) by the people. But in the digital world, this authority itself can be avoided by having a peer to peer system; e.g. the blockchain, ensuring itself the respect of the rules. In the real world, one can barely imagine such a system. Nevertheless, thanks to the recent technological advances we now can imagine a decentralized democracy.

Talking about politics, in his famous book [31], Pr. Robert A. Dahl came up with 5 criteria to measure the ideal representative democracy. Here they are :

- effective participation
- equality in voting
- enlightened understanding
- control of the agenda
- inclusion

Without diving too much into the details, one can see that in a decentralized democracy, everybody would have adequate and equal opportunities for placing questions on the agenda and for expressing their preferences throughout the decision making process. *One person one vote*. That is, thanks to the absence of authority, roughly all the criteria would be satisfied. Indeed, the third one could lean to the problems that we are actually facing; e.g.all citizens should be more or less equally aware of the pros and cons of subjects so they can make a choice that serve best their interests.

It looks like an utopia but the advent of tools and protocols like the Proof-of-Personhood could definitely lead to a decentralized and more democratic approach.

6.1.3 Other related Project

Without going through all the different works that have been done in this subject, especially in the DEDIS lab. Here are two other papers that haven't been published yet:

- The first one [32] explores how technology has failed to support robust democracy but could eventually help us make better collective choices. Going through digital deliberation and voting systems since making good decisions relies on the availability of good information.
- A related longer-term project to build on Proof-of-Personhood to create cryptocurrencies ("PoPcoin") implementing "democratic money" is already in progress. Some of the principles of the conceived PoP-based democratic money are informally developed in this draft [33].

Chapter 7

Conclusion

In this project, we built an app reinforcing the privacy of users. By attending a PoP event, users can obtain a digital identity which is not linked to their physical identity while still being accountable. While the app offers very basic functionalities, we have a strong starting point which can be easily extended. For example, we can easily add new features to the protocol by adding new messages to the data layer. No change of other layers is needed. Future work is needed to add missing functionalities such as other types of events and to improve the usability.

Bibliography

[1] WhatsApp LLC. *WhatsApp Privacy Policy.* URL: https://www.whatsapp.com/legal/updates/privacy-policy/?lang=en. 04.01.2021.

[2] Zoe Kleinman. *WhatsApp users flock to rival message platforms.* URL: https://www.bbc.com/news/technology-55634139. 12.01.2021.

[3] Serge Vaudenay. *The Dark Side of SwissCovid.* URL: https://lasec.epfl.ch/people/vaudenay/swisscovid.html. 04.07.2020.

[4] Bryan Ford and Jacob Strauss. *An Offline Foundation for Online Accountable Pseudonyms.* URL: https://bford.info/pub/net/sybil.pdf.

[5] *JSON protocol specification.* URL: https://github.com/dedis/student20%5C_pop/tree/proto-specs (visited on 01/15/2021).

[6] JSON-RPC Working Group <json-rpc@googlegroups.com>. *JSON-RPC 2.0 Specification.* 2013. URL: https://www.jsonrpc.org/specification (visited on 01/15/2021).

[7] JSON. *Introducing JSON.* URL: https://www.json.org/json-en.html (visited on 01/15/2021).

[8] JSON Schema. *JSON Schema.* 2019. URL: https://json-schema.org/ (visited on 01/15/2021).

[9] Joachim Bauch Gary Burd. *Gorilla WebSocket.* URL: https://github.com/gorilla/websocket (visited on 01/15/2021).

[10] Facebook and community. *React Native.* https://reactnative.dev/. [Online; accessed 13-January-2021]. 2021.

[11] Ryan Dahl. *Node.js.* https://nodejs.org/en/. [Online; accessed 13-January-2021]. 2021.

[12] React-native Front-end. *Readme.* https://github.com/dedis/student20_pop/tree/fe1-web. [Online; accessed 13-January-2021]. 2021.

[13] Airbnb. *Airbnb JavaScript Style Guide.* https://github.com/airbnb/javascript. [Online; accessed 13-January-2021]. 2021.

[14] Expo's team. *Expo.* https://expo.io/home. [Online; accessed 13-January-2021]. 2021.

[15] Community. *React Navigation.* https://reactnavigation.org/. [Online; accessed 13-January-2021]. 2021.

[16] Dan Abramov. *Redux*. `https://redux.js.org/`. [Online; accessed 13-January-2021]. 2021.

[17] Zack Story. *Redux-persist*. `https://github.com/rt2zz/redux-persist`. [Online; accessed 13-January-2021]. 2021.

[18] Vojtech Novak and Luan Curti. *React Native DateTimePicker*. `https://github.com/react-native-datetimepicker/datetimepicker`. [Online; accessed 13-January-2021]. 2021.

[19] hackerone. *React Datepicker*. `https://reactdatepicker.com/`. [Online; accessed 13-January-2021]. 2021.

[20] Facebook. *Prop-types*. `https://github.com/facebook/prop-types`. [Online; accessed 13-January-2021]. 2021.

[21] Christoph Nakazawa and Facebook. *Jest*. `https://jestjs.io/`. [Online; accessed 13-January-2021]. 2021.

[22] Android Front-end. *ReadMe*. [Online; accessed 13-January-2021]. URL: `https://github.com/dedis/student20_pop/blob/fe2-android/README.md`.

[23] Google. *Mobile Vision*. [Online; accessed 13-January-2021]. URL: `https://developers.google.com/vision`.

[24] Google. *Gson Library*. [Online; accessed 13-January-2021]. URL: `https://github.com/google/gson`.

[25] Network New Technologies Inc. *Json Schema Validator*. [Online; accessed 13-January-2021]. URL: `https://github.com/networknt/json-schema-validator`.

[26] Community. *Project Tyrus*. [Online; accessed 13-January-2021]. URL: `https://tyrus-project.github.io/`.

[27] Community. *JUnit*. [Online; accessed 13-January-2021]. URL: `https://junit.org/`.

[28] Jonathan Halterman. *ConcurrentUnit*. [Online; accessed 13-January-2021]. URL: `https://github.com/jhalterman/concurrentunit`.

[29] Google. *Espresso*. [Online; accessed 13-January-2021]. URL: `https://developer.android.com/training/testing/espresso`.

[30] Maria Borg, Eleftherios Kokoris-Kogias, Philipp Jovanovica, Linus Gasser, Nicolas Gailly, and Bryan Ford. "Proof-of-Personhood: Redemocratizing Permissionless Cryptocurrencies". In: *IEEE Security & Privacy on the Blockchain (IEEE S&B)*. 2017.

[31] Robert Dahl. "Democracy and Its Critics". In: *Yale University Press*. 1989.

[32] Bryan Ford. *Technologizing Democracy or Democratizing Technology?A Layered-Architecture Perspective on Potentials and Challenges*. `https://bford.info/pub/soc/dt2-chapter.pdf`. [preliminary work-in-progress; may becomepart of a future book]. 2020.

[33] Bryan Ford. *Democratic Value and Money for Decentralized Digital Society*. `https://arxiv.org/pdf/2003.12375.pdf`. [preliminary work-in-progress; may becomepart of a future book]. 2018.