



École Polytechnique Fédérale de Lausanne

Anonymous Proof-of-Presence Groups for Messaging and Voting

by Carlo Musso, Victor Carles, Johann Pluss, Filippo Salmina, Adalsteinn Jonsson, Karim Kabbani, Maxime Laval, François Michel, Luke Patel, Stefan Eric

Project Report

Approved by the Examining Committee:

Prof. Bryan Ford
Thesis Advisor

Pierluca Borsò
Thesis Advisor

Cristina Basescu
Thesis Supervisor

Louis-Henri Merino
Thesis Supervisor

Haoqian Zhang
Thesis Supervisor

EPFL IC DEDIS
Bâtiment BC
Station 14
CH-1015 Lausanne

June 11, 2021

Abstract

Digital identities has always had to compose with a subtle compromise : accountability vs. anonymity. While this issue can be problematic to any online service that support accounts, it becomes crucial when online voting is concerned. "One person, one vote" is a concept that must be upheld in a democracy and one that can be difficult to ensure on a system that guarantees anonymity which is, of course, as critical. To address this, Proof-of-Personhood guarantees both accountability and anonymity to users by running periodic physical meetings in which participants receive digital tokens that attest their digital personhood. This project builds on the work of a previous team of students and engineers which allowed the creation of events and the organization of the previously mentioned physical distribution of tokens event, named roll-calls. Our project adds the possibility of conducting an election - with certain limitations that will be addressed by future students and engineers - as well as storing the tokens in a digital wallet.

Contents

Abstract	2
1 Introduction	5
2 Background	7
2.1 Communication	7
2.1.1 Publish-Subscribe pattern	7
2.1.2 WebSocket	7
2.1.3 JSON RPC	8
2.2 Cryptography	8
3 General Overview	10
3.1 Roles	10
3.1.1 Organizer	11
3.1.2 Attendee	11
3.1.3 Witness	11
3.2 System Architecture	12
3.2.1 Communication Architecture	13
3.3 Existing Code Base	13
3.3.1 Android Front-End	13
3.3.2 Web Front-End	14
3.4 Team Organisation	15
4 Design	16
4.1 Menu	16
4.2 E-Voting	19
4.2.1 Election Setup	19
4.2.2 Cast Vote	23
4.2.3 Election Result	26
4.2.4 Election States in the web-front-end	29
4.2.5 Manage Election	29
4.3 Digital Wallet	30
4.3.1 Introduction	30

4.3.2	Web Interface	32
4.3.3	Android Interface	35
4.4	Witness UI	37
5	Implementation	40
5.1	E-Voting	40
5.1.1	Back-end 1 - Go	40
5.1.2	Back-end 2 - Scala	46
5.1.3	Front-end 1 - React [native]	47
5.1.4	Front-end 2 - Android	50
5.2	Digital Wallet	52
5.2.1	Structure - HD Wallet	52
5.2.2	Structure - Importing and exporting seed	54
5.2.3	Structure - Generating PoP Tokens	54
5.2.4	Structure - Retrieving PoP Tokens	55
5.2.5	Specifics	55
5.2.6	Specifics fe1 - Typescript (React)	55
5.2.7	Specifics fe2 - Java (Android)	57
6	Evaluation	59
6.1	Testing	59
6.1.1	Pop Parties	59
6.1.2	Unit Tests	60
7	Future Work	62
7.1	Election	62
7.1.1	Properties	62
8	Conclusion	63
	Bibliography	64

Chapter 1

Introduction

Voting is a delicate subject. In western democracies, this observation was maybe never easier to make than these days following the U.S 2020 Presidential election. There are a lot of issues with the aforementioned debacle, we will not get into them for E-voting does not address, save for one. Accessibility. In the U.S, for people voting in person, the waiting time for accessing the voting booth can go into the hours which makes it difficult for some to vote since voting day is no Holiday. Moreover, there is racial discrimination associated with disparities in waiting time [3]. This contravenes the "Voting Equality at the Decisive Stage" principle of the democratic process[1]. So, by going digital is this problem solved, is it not ? Well, as often in Computer Science, now that we may have solved a problem, we have created several others engineering wise. Let us explain.

One of the properties of a vote that must be upheld is the anonymity of its caster. If it were not, the citizen would risk retaliation, be it from government or not, sheer social pressure, and even without these more dire consequences it would impede on his privacy. This would affect the integrity of the vote for it would be possible for the vote not to have been freely cast. Here we have our first engineering challenge: guaranteeing that a specific vote cannot be linked to an identity.

This brings us to our second main challenge: identity. Our voting system must be resilient to Sybil, i.e. multi-account, attacks. This would ensure the principle "One person, one vote". The underlying property that we wish to have is accountability. By making sure that an account is linked to a person, we prevent bad actors from operating in impunity for punishment, for example ban, is permanent. Indeed, with that aforementioned link, a banned account means a banned person. Moreover, this also implies that Sybil attacks are impossible for a person to have multiple accounts and for bots to operate any account. There are many ways to add some accountability and fight against Sybil attacks but for voting the requirements is at its paroxysm: we must prevent any citizen from voting twice. This implies that the system must provide certainty that any person can have at most one account. One identifying way that jumps to mind is biometric authentication. Many of us uses this daily to unlock phone, computers

and whatnot. So let us address right away why it would be a bad idea. Here we list some of the problems that arise [4]

- Biometric system errors - even with a slight error rate, this can prevent someone to rightfully vote.
- Compromised biometrics - There is no easy way to replace biometrics which is problematic if one's biometric is compromised. One could, of course, hash the data before submitting it but it runs into a massive problem : biometric systems accept input when they are "close enough" and a good hash function maps two close inputs to dissimilar one.
- Privacy - There are several privacy issues for example tracking and profiling.

Proof of Personhood

Proof of personhood (PoP) is a participation and Sybil attack resistance method for non-trusted users, in which each unique human participant obtains one equal unit of voting power. The purpose of the PoP mechanism is to bind physical entities to virtual identities in a way that enables accountability while preserving anonymity.

One possibility to create one-per-person identities to use in distributed systems are pseudonym parties. This is done leveraging the fact that humans can **physically** be in only one place at a time. Party participants gather periodically at in-person event and receive their PoP token for that event. Notice that no personal identity-related information is ever needed, just the presence of the person at this event.

What makes this system so trustworthy is that once a person has obtained its PoP token, this token can be then provided to any platform which supports it, as a proof of identity. If the user behaves with common sense (thus not giving away its unique personal token) the process with which the token is obtained ensures the fact that the token is unique per person (no Sybil attacks) and that it surely belongs to the same person that was present when the token was distributed.

This project uses Proof-of-Personhood in order to meet this apparent antithetical conjunction of properties of Anonymity and Accountability.

Chapter 2

Background

2.1 Communication

Most of the specifications of the project's architecture were already decided by last semester's team, so this is not per se part of our project. Nonetheless, as it is important to understand our work, we will here provide a quick recap of how the communication between the back-end and the front-end works.

2.1.1 Publish-Subscribe pattern

The communication works following a publish-subscribe pattern. The different clients communicate using a specific channel, e.g the current lao channel. To do so, they first have to subscribe to this channel, so that they can receive the messages published by other clients. Every time a message is published on the channel, it is then broadcast to other clients by the server. To keep the architecture simple, the front-end corresponds to the client, and the back-end to the server.

In case a client subscribes to a new channel and requests old messages, as to be up-to-date with the rest of the subscribers, it can send a catch-up message to the server. Obviously, every time the back-end receives a message from the front-end, it has to notify it with a result, or an error if something went wrong.

2.1.2 WebSocket

The communication protocol for this project is WebSocket. It has been decided that way since WebSockets are easy to use and provide full-duplex communication, whereas HTTP only provides half-duplex. This means that messages can be sent and received both at the same time, making

it really adequate for the publish-subscribe pattern.

2.1.3 JSON RPC

Finally, the data sent over the network is serialized using JSON RPC. This way, we can easily specify the type of the message sent, its parameters (the channel on which the message is sent, the data it contains, the id of the request...), as well as other useful fields such as a result and an error field in case of a server response.

```
→ {
  "jsonrpc": "2.0",
  "method": "publish",
  "params": {
    "channel": "channel_id" /* string */
    "message": {
      "data":
        ...
        /* fields of data */
        ...
    },
    "id": 3 /* unique request identifier */
  }

← {
  "jsonrpc": "2.0",
  "result": 0, /* present on success, absent on error */
  "error": {}, /* absent on success, present on error */
  "id": 3 /* matching request identifier */
}
```

Last semester, the ids, the data and the keys were encoded using base64. This proved to cause problems when parsing the channel ids, as it is split using "/", which could also appear in a base64 encoding. To avoid those problems, we decided to change the encoding to base64URL, which is URL safe, and replaces "/" with "_".

2.2 Cryptography

In this project we use **Ed25519** as our signature scheme. Edwards-curve Digital Signature Algorithm (EdDSA) is a modern and secure digital signature scheme based on twisted Edwards

curves (a popular elliptic curve). With EdDSA we can prove the authenticity of a message very efficiently.

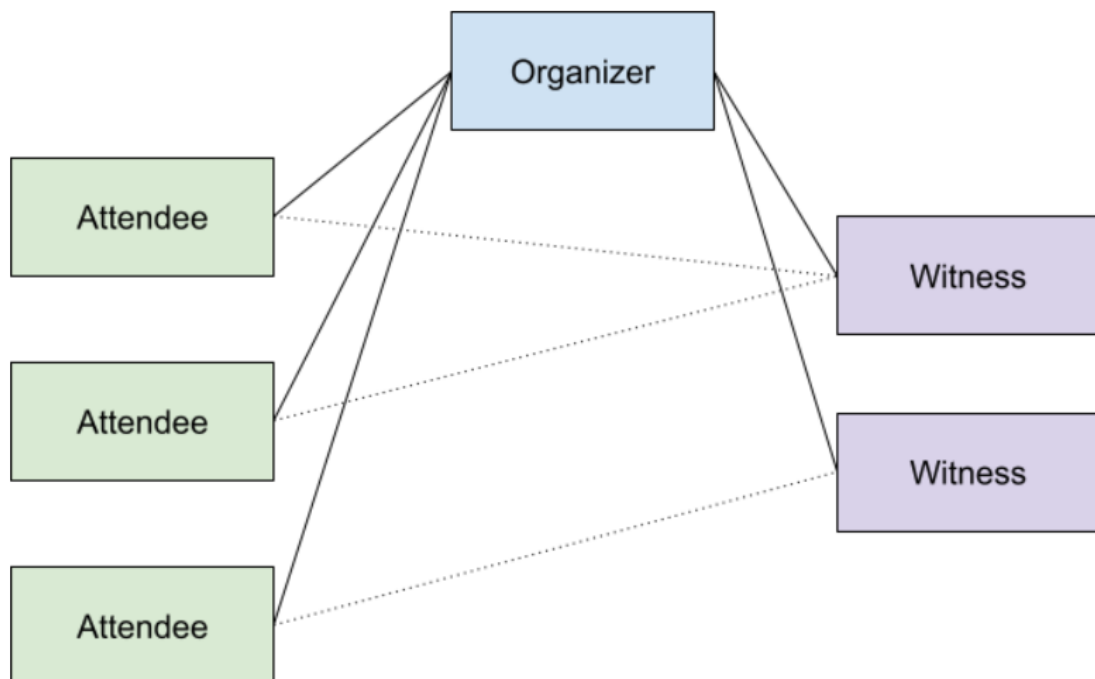
The hash function we used is **SHA256** in all the systems.

Chapter 3

General Overview

3.1 Roles

We will describe the different roles that appear in the system and what their purpose is to make the system function as intended, we of course must take into account that when designing the system that we have a malicious user that has one of these roles, to avoid any possibility of the system failing or making it bias we have the remaining roles and different protocols we follow in order to minimize the chances of biasing or taking over the system.



3.1.1 Organizer

The organizer is going to be the main role in the system a lot of the communication will go through and be caused by the organizer's actions. First of all the organizer has 2 components the front-end and the back-end that are not necessarily on the same device, but since no human action is needed on the back-end side a single person is enough to handle this role. More in detail an organizer will be responsible to create different events on in this PoP context and to propagate the respective creations/updates of events to the rest participants. First of all and maybe the most important event it is going to create is a LAO (local autonomous system) where each other participant can be part of and receive every message passed for it since a channel is going to be created and the publish/subscribe protocol (described above) is going to be applied. We will explain in the upcoming sections more in detail but the organizer can choose to create an election on this LAO channel and by doing this creating a new channel for it. The advantage of having the front-end and the back-end is to have the back-end perform the actions of the front-end and hope to catch and stop some of the illicit activities and prevent a malicious use of the front-end part of the organizer role.

The organizer is connected to the rest of the system through different Web Sockets, each attendee and each witness should receive a message from the organizer once it broadcasts it to a certain channel.

3.1.2 Attendee

Attendees are the main actors of the system their are the reason why events are created/updated, attendees do not have any power on the functionality of the system they just obey the rules that the organizer set and have minimal impact on the technical aspects of the system. On the other hand attendees are actual people that participate in different events that the organizer has created, once an attendee has shown a "proof of personhood" it becomes part of a system and part of a or multiple LAOs. An attendee is a client in this system and is given as much freedom as the organizer gives to them, meaning that the only actions an attendee will have will have to be predetermine by the organizer that will setup an event. A person can become an attendee by being present in human meetings where the organizer that will be the creator of a LAO grants permission to an attendee to enter this LAO. Every attendee is connected to the organizer but can also be connected to some witnesses to attest the organizer's actions, this will be elaborated in the witness section.

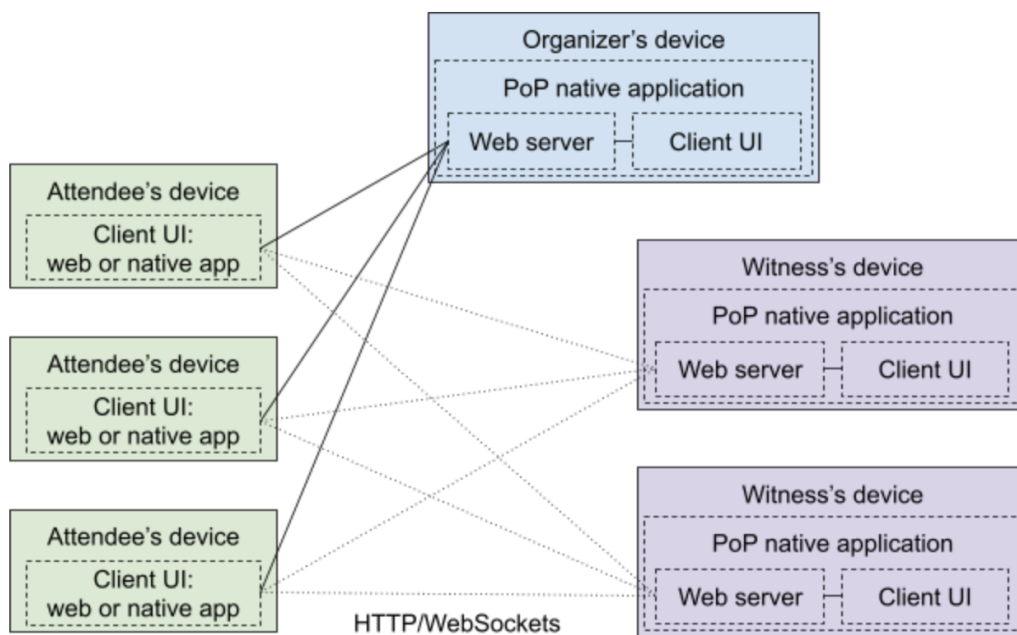
3.1.3 Witness

Witnesses are in the system in order to ensure resilient communication. Like the organizer, the witness has two components: a front-end and a back-end. They are not necessarily on the same

device and no action is needed on the back-end so a single person is enough to manage each witness. The purpose of the witnesses is to ensure message delivery despite malicious servers, whether they be organizers or witnesses. In order to do this, each witness connects to every other server (both witness and organizer) as well as all attendees. When a client or server publishes on a channel, the message is also sent to each witness. These messages are sent from the back-end to the front-end, where they are signed and sent back to the server which sends them on to every attendee and every other server. This allows the attendees to verify the integrity of the messages in case of, for example, a malicious organizer. However multiple optimizations will have to be applied to the witness in order to avoid the all-to-all communication. One optimization will be to limit the classes of messages sent to attendees by subscription channels and topics. Another optimization will be to reduce what servers send to the clients. For example, instead of sending a whole message, they may just send a cryptographic hash ID of the message.

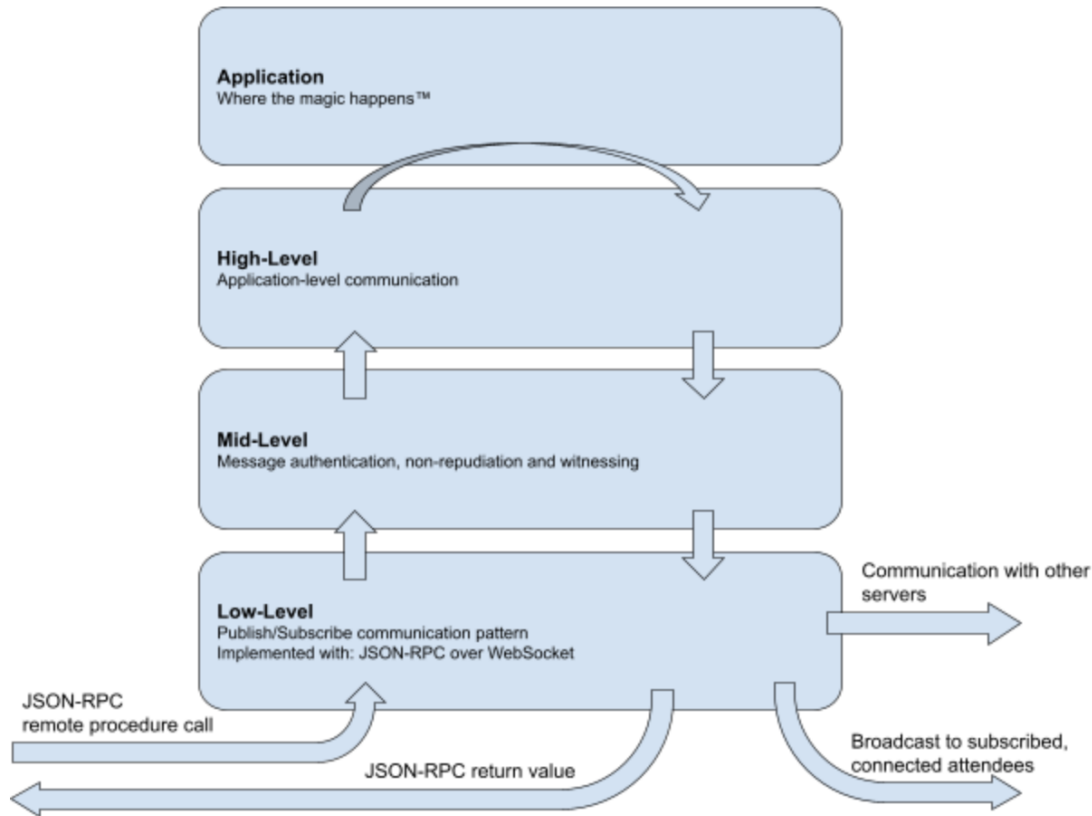
3.2 System Architecture

Here we will show how the different sub-parts work together. As we can see on the following schema, the organizer's and witness' server is to be run on their respective devices, that is on the same device as the client UI. By contrast, attendees only need to run the client.



3.2.1 Communication Architecture

Conceptually, the communication follows the steps that are detailed in the following diagram.



3.3 Existing Code Base

3.3.1 Android Front-End

When we started the project a good part of the code was already present, especially on the communication with the back-end. We could create and connect to a LAO, see the list of LAOs, see the (empty) list of events and fill the form for a new roll-call. But we could not create a roll-call and everything that comes afterwards. We had to complete the roll-call implementation which mainly involved writing new code related to the UI and bug fixing on the UI and the back-end communication side.

3.3.2 Web Front-End

On the web front-end we were given a clean code base in the beginning, which looked well structured and followed best practices such as the *Airbnb ESLint*. Despite all that, for someone who has never learned *React* before, let alone *Redux* it was quite overwhelming in the beginning. Especially since there were over 80 files to go through and most of them weren't sufficiently commented. Also the folder structure was quite confusing since files with very similar names were appearing in different folders. As an example if we only look at the folders with filenames that contain 'Lao' we see this structure:

```
/
├── components
│   ├── eventList
│   │   └── LaoProperties.tsx
│   └── LAOItem
├── ingestion
│   └── Lao.ts
├── model
│   ├── network
│   │   ├── method
│   │   │   └── message
│   │   │       └── data
│   │   │           └── lao
│   │   │               ├── CreateLao.ts
│   │   │               ├── StateLao.ts
│   │   │               └── UpdateLao.ts
│   ├── Lao.ts
│   ├── LaoEvent.ts
│   └── LaoEventBuilder.ts
├── parts
│   └── lao
│       ├── attendee
│       ├── organizer
│       └── witness
├── res
│   └── laoData.ts
├── store
│   ├── reducers
│   │   └── LaoReducer.ts
│   └── stores
│       └── OpenedLaoStore.ts
```

However, these challenges just made the project more interesting to us as the project setting

is probably representative of a realistic software development project in a company.

3.4 Team Organisation

The project followed N-version programming with 2 separate front and back-ends.

Front-ends:

1. **TypeScript** using React [Native], targeting iOS, Android, and Web-based execution
2. **Java** using the Android APIs, targeting Android mobile devices specifically

Back-ends:

1. **Go**, to run either on a Mac or Linux server, or embedded in an iOS or Android app
2. **Scala** using standard system APIs, to run as a server or embedded in an Android app

In the beginning of the semester the Scala back-end wasn't ready for us to work on so we split us up into the three remaining technologies: two students were in the Typescript front-end team, five students in the Java front-end team and three students in the Go back-end team. Further the project was divided into two parts, the e-voting part and the digital wallet part. Seven worked on the e-voting part and 3 on the digital wallet.

Every week, each student had three meetings to attend to. The first was the technology meeting regrouping all the students assigned to a specific technology. The second one was the role meeting (either e-voting or digital wallet). And the last meeting was the general meeting with everyone who works on the project.

Chapter 4

Design

4.1 Menu

We are presenting here how the general user experience of the application looks like focusing on the basic functionalities which are creation of a LAO, subscription to a LAO, creation and process of roll-calls. Because this part does not fit into the e-voting or wallet sections, we decided to create a separate subsection.

There are two different points view for the UI, namely the organizer's and the subscriber's (or attendee's). We are going to highlight the differences between the two.

In Figure 4.1 we show in the respective order of the pictures:

- How we can create a LAO
- The list of LAOs we either created or subscribed to
- The LAO properties which can be edited (only possible for the organizer) and the QR code that can be scanned for users to subscribe

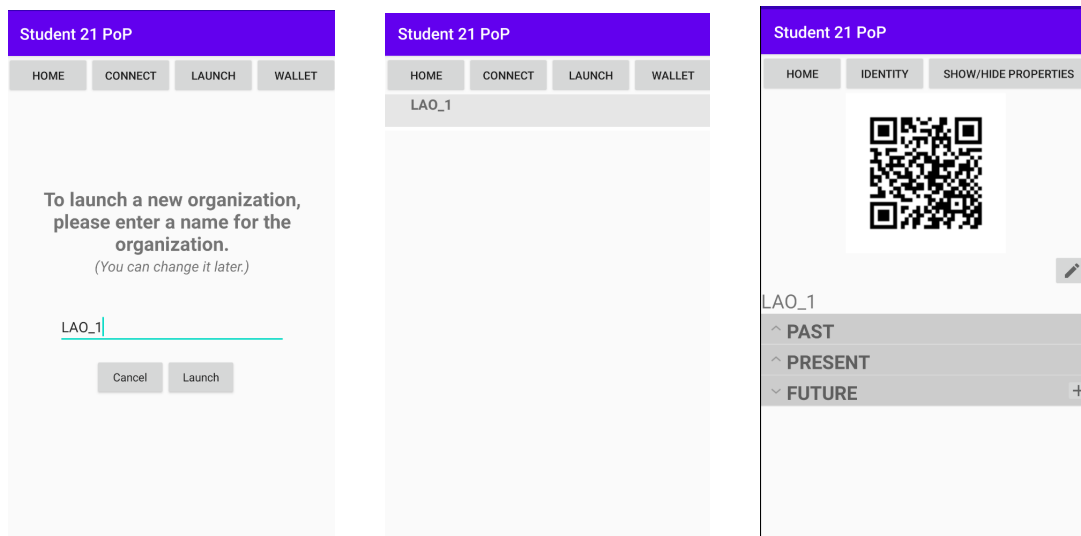


Figure 4.1 – Organizer's UI

After creating and running a few roll-calls, in Figure 4.2 we show in the respective order (organizer's view only):

- The list of events, in this case two roll-calls are closed but could be reopened
- Clicking on the "+" button shows a pop-up to create new events
- The organizer decides to create a roll-call, enters the related information and clicks on "confirm"
- The roll-call is added to the list of events and the LAO clicks on "open" to start the roll-call.

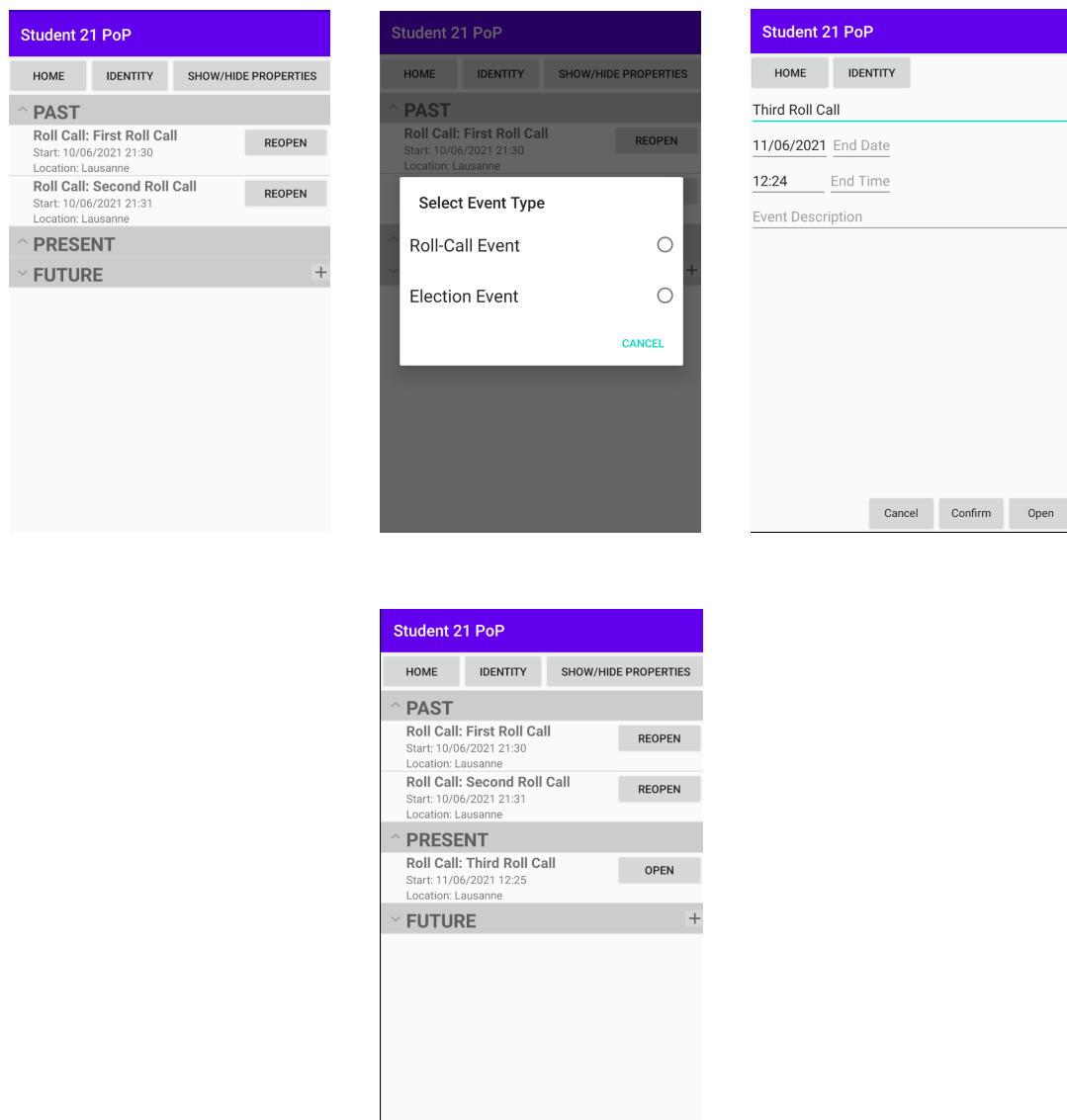


Figure 4.2 – Organizer creates roll-calls

After a user subscribes to the LAO by clicking on "connect" and by scanning the LAO's QR code, what happens when he goes to the list of events is shown in Figure 4.3 in the respective order:

- The new roll-call is open and the user can click on "enter".
- The users public key is displayed as a QR code.
- The organizer scans the attendee and the counter is incremented. After scanning everyone the organizer can close the roll-call.

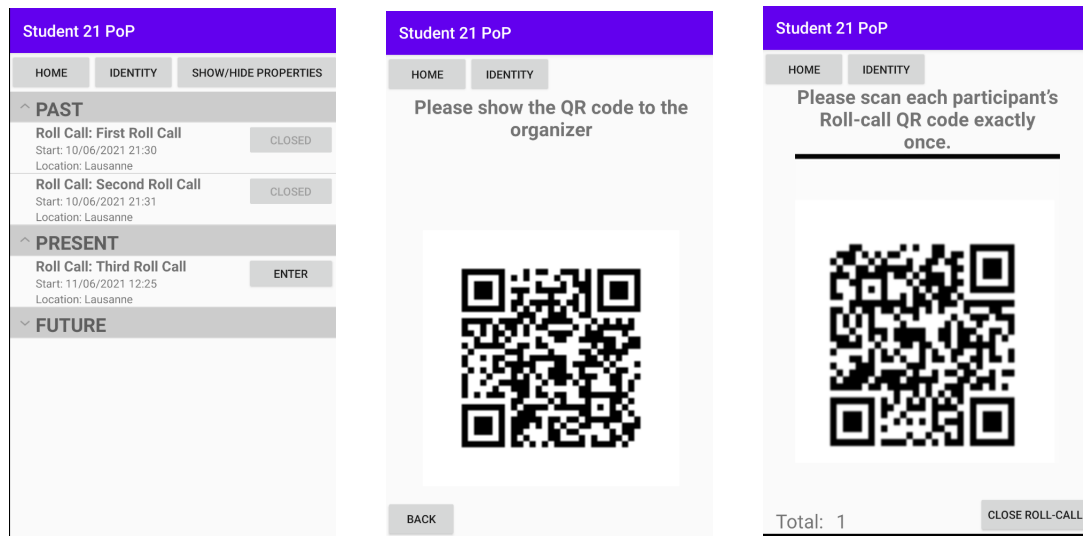


Figure 4.3 – Attendee sees the roll-call, can enter and organizer scans him

4.2 E-Voting

One of the two main goals for this semester's project was to implement E-voting, making it functional on both the web and the android versions.

E-voting would certainly be achievable with proof-of-personhood, as this would guarantee that a voter is a real individual, hence avoiding Sybil attacks that would tamper the results, while keeping them fully anonymous.

The idea is to make the organizer of a LAO able to set up an election, that could contain multiple questions, at a specified time and date. The attendees could then cast their votes when the election is opened, and see the results when the organizer ends the election after past time. To avoid any malicious behavior from the organizer, the witness's goal would be to testify that the votes are legit, by comparing stored hashes when the election is ended.

The hardest part of this project is to make the election robust and coherent for every client (android users as much as web users).

4.2.1 Election Setup

The first step is to design some user interface for the election setup. The organizer should be able to choose the number of questions, the number of ballot options per question, the voting method (for now, only plurality), the date/time of the election as well as its name. There can also be a "write in" switch activated, specifying that the attendees can directly write the name of the candidate instead of simply choosing amongst different options.

When the election is setup, an "Election setup" message is sent to the back-end and, upon

successful reception, broadcast to all subscribers, updating their clients with a layout displaying the election.

ElectionSetup:

The organizer A sends an "ElectionSetup" event in the LAO group:

```
→ {
  "jsonrpc": "2.0",
  "method": "publish",
  "params": {
    "channel": "/root/<lao_id>" // LAO channel
    "message": {
      "data": base64({ /* Base 64 representation of the object
        "object": "election", // Constant
        "action": "setup", // Constant
        "id": <hash>, // SHA256("Election"||lao_id||timestamp||name)
        "lao": <lao_id> // ID of the LAO
        "name": <string>, // Name of the election
        "version": <string>, // Features/Implementation identifier
        "created_at": <UNIX timestamp>, // Time created in UTC
        "start_time": <UNIX timestamp>, // Start Time of the election in UTC
        "end_time": <UNIX timestamp>, // End Time of the election in UTC
        "questions": [
          {
            "id": <base64> //Hash("Question"||election_id||question)
            "question": <string>, // Voting topic
            "voting_method": <stringEnum> // Supported Voting Method
            // (e.g. "Plurality", "Approval", ...)
            "ballot_options": [<option1-string>, <option2-string>,
            ...], // Ballot Options
            "write_in": <bool>, // Whether write-in is allowed
          },
          {
            "id": <base64> //Hash("Question"||election_id||question)
            "question": <string>, // Voting topic
            "voting_method": <stringEnum> // Supported Voting Method
            // (e.g. "Plurality", "Approval", ...)
            "ballot_options": [<option1-string>, <option2-string>,
            ...], // Ballot Options
            "write_in": <bool>, // Whether write-in is allowed
          },
          ... /* Any additional questions */
        ]
      })),
      "sender": <base64>, /* Public key of organizer */
      "signature": <base64>, /* Signature by organizer over "data" */
      "message_id": <base64>, /* hash(data||signature) */
      "witness_signatures": [], /*Signature by witnesses(sender||data)*/
    },
    "id": 10 /* unique request identifier */
  }
}

← {
  "jsonrpc": "2.0",
  "result": 0, /* present on success, absent on error */
  "error": {}, /* absent on success, present on error */
  "id": 10 /* matching request identifier */
}
```

[HOME](#)[ORGANIZER](#)[MY IDENTITY](#)[TEST ORGANIZATION](#)

Election Setup

Election

Start time:

06/11/2021 11:26

End time:

06/11/2021 12:26

What to eat?

Voting Method

Plurality

Pizza

Sushi

Hamburgers

Add option

Which time?

Voting Method

Approval

Add option

ADD QUESTION

CANCEL

CONFIRM

Figure 4.4 – Election Setup in the Web Front-end (Version 2)

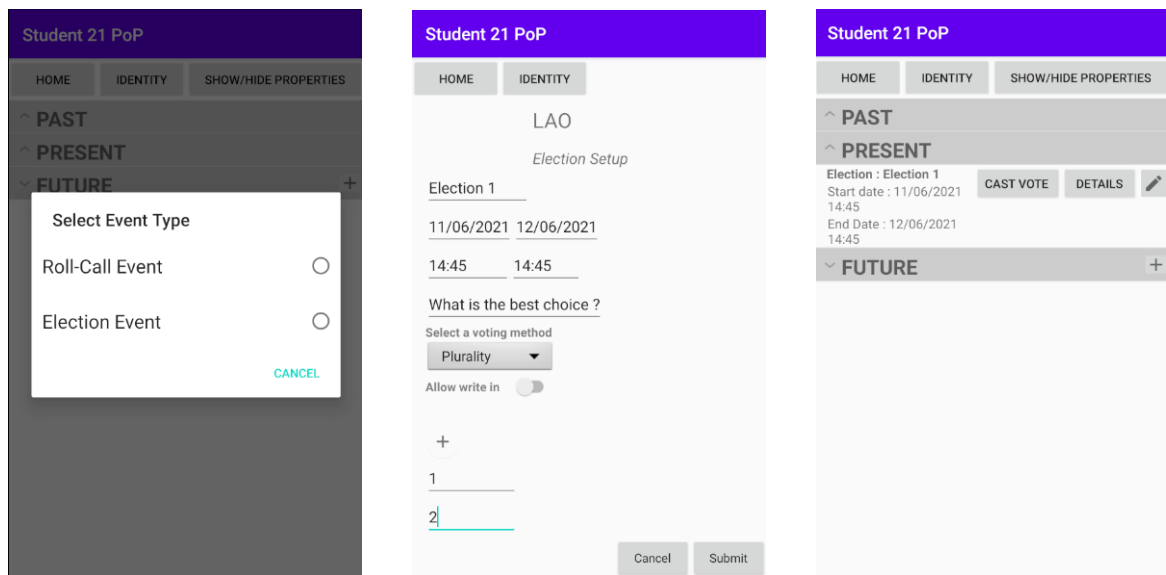


Figure 4.5 – Election Setup in Android Front-end

Depending on the time and date, this layout could either be in a past, present or future section, where each section would have some associated actions.

For instance, in the future section, the organizer would have access to an "edit" button that allows him to change some attributes of the election, or cancel the election. In the present section, the attendees would have access to a button that would bring them to a new page where they can place their vote. In the past section, instead of that, the organizer would have a button to end the election. For the attendees, the results should be accessible once they are ready through the past section as well.

4.2.2 Cast Vote

When clicking on the corresponding button, the attendees would have access to a page where they can place their vote. For each question, the attendee can choose the desired ballot option by simply clicking on it. Once every votes are placed, the attendee can then click on a "cast vote" button to send them. A "Cast vote" message will then be sent to the back-end that will store the votes for the corresponding sender, identified by their public key.

Attendees can cast as many votes as they wish, but only the last one will be taken in consideration. The validity of the vote is determined by its date of creation, one of the fields stored in the message's data, and not the order of reception, to avoid reordering attacks.

The back-end, after sending its answer, should also broadcast the cast vote message. This is because, not only the server, but also every client will store the votes, as it is best not to completely rely on the server. The idea, in a second time, would be for the back-end to broadcast this message only to witnesses, as they are the only attendees that should have a copy of the votes.

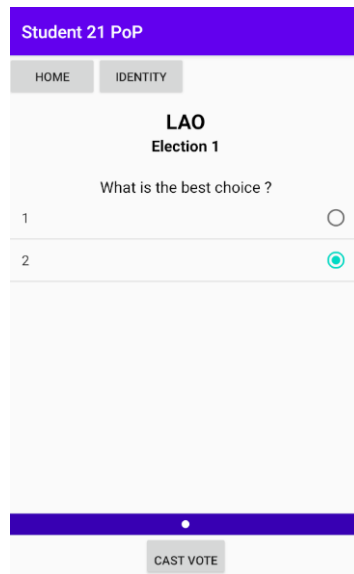


Figure 4.6 – Casting vote in Android Front-end

Casting Vote(s):

The LAO member belonging in L sends a "CastVote" event in the election channel:

```
→ {
  "jsonrpc": "2.0",
  "method": "publish",
  "params": {
    "channel": "/root/<lao_id>/<election_id>" // LAO-Election channel
    "message": {
      "data": base64({ /* Base 64 representation of the object
        "object": "election", // Constant
        "action": "cast_vote", // Constant
        "lao": <lao_id> // ID of the LAO
        "election": <election_id>, // ID of the election
        "created_at": <UNIX timestamp> // Vote submitted time in UTC
                          // and in milliseconds
        "votes": [
          {
            "id": <base64> // Hash("Vote"||election_id||
// question_id||(vote_index(es)|write_in))
// concatenate vote indexes - must use delimiter
            "question": <question_id> // ID of the question
            "vote": [<vote1-index, vote2-index, ...]
                      // Index corresponding to the ballot_options
                      // Absent if write_in is present
            "write_in": <string> // Valid only if write in votes are
          },
          {
            "id": <base64> // Hash("Vote"||election_id||
// question_id||(vote_index(es)|write_in))
// concatenate vote indexes - must use delimiter
            "question": <question_id> // ID of the question
            "vote": [<vote1-index, vote2-index, ...]
                      // Index corresponding to the ballot_options
                      // Absent if write_in is present
            "write_in": <string> // Valid only if write in votes are
          },
          ... /* Additional votes */
        ]
      })),
      "sender": <base64>, /* Public key of the voter */
      "signature": <base64>, /* Signature by voter "data" */
      "message_id": <base64>, /* hash(data||signature) */
      "witness_signatures": [], /*Signature by witnesses(sender||data)*/
    },
    "id": 10 /* unique request identifier */
  }
}

← {
  "jsonrpc": "2.0",
  "result": 0, /* present on success, absent on error */
  "error": {}, /* absent on success, present on error */
  "id": 10 /* matching request identifier */
}
```

4.2.3 Election Result

When the election is past its end time, the organizer can end it by clicking on a "tally votes" button. This made more sense for us than a "terminate election" button, as the election should theoretically be ended once it is in the past section, although it does the exact same thing: ending the election and tallying the votes. Letting the organizer choose when the results should be tallied is still a bit of a flaw in our opinion, as a malicious organizer could decide to never let the attendees see the results, but this is a first-hand simple way to do it. Indeed, making the tallying fully autonomous, based on timestamps, could greatly complexify the system, and would require to take in consideration local zones, delays etc...

When the button is clicked, the organizer's front-end sends an "Election end" message to the back-end, containing a hash of all vote id's stored in the client. The back-end will then proceed to compare the received hash and the computed hash from the server, to see if all the votes were correctly gathered. Upon success, the back-end will broadcast the end message, making every clients update the state of the election. Needless to say, at this point, no more votes received would be taken in consideration in the tallying.

```
→ {
  "jsonrpc": "2.0",
  "method": "publish",
  "params": {
    "channel": "/root/<lao_id>/<election_id>" // LAO-Election channel
    "message": {
      "data": base64({ /* Base 64 representation of the object
        "object": "election", // Constant
        "action": "end", // Constant
        "lao": <lao_id>, // ID of the LAO
        "election": <election_id>, // ID of the election
        "created_at": <UNIX timestamp>, // Vote submitted time in UTC
        "registered_votes": SHA256(<vote_id>, <vote_id>, ...),
      })),
      "sender": <base64>, /* Public key of organizer */
      "signature": <base64>, /* Signature by organizer over "data" */
      "message_id": <base64>, /* hash(data||signature) */
      "witness_signatures": [], /*Signature by witnesses(sender||data)*/
    },
    "id": 10 /* unique request identifier */
  }
}

← {
  "jsonrpc": "2.0",
  "result": 0, /* present on success, absent on error */
  "error": {}, /* absent on success, present on error */
  "id": 10 /* matching request identifier */
}
```



Figure 4.7 – Ending an election and tallying votes

The server will then proceed to count the votes, and broadcast an "Election result" message. The client will order the results and display them in a new page accessible with a "see results" button. The UI for the results should contain all ballot options for each question, with the name of the ballot option next to its corresponding number of votes, ordered by count. The attendees should be able to access the results of an election indefinitely.

```

→ {
  "jsonrpc": "2.0",
  "method": "publish",
  "params": {
    "channel": "/root/<lao_id>/<election_id>" // LAO-Election channel
    "message": {
      "data": base64({ /* Base 64 representation of the object
        "object": "election", // Constant
        "action": "result", // Constant
        "questions": [
          {
            "id": <question_id> // ID of the question
            "result": [<result1-string>, <result2-string>,
              ...] // Results
          },
          {
            "id": <question_id> // ID of the question
            "result": [<result1-string>, <result2-string>,
              ...] // Results
          }
        ],
        "witness_signatures": [], // Signatures before sending message
      }
      "sender": <base64>, /* Public key of organizer */
      "signature": <base64>, /* Signature by organizer over "data" */
      "message_id": <base64>, /* hash(data||signature) */
      "witness_signatures": [], /*Signature by witnesses(sender||data)*/
    },
    "id": 10 /* some unique request identifier */
  }
}

```

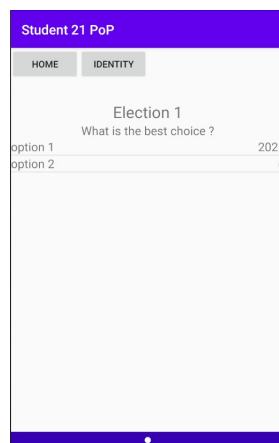


Figure 4.8 – Election results display

4.2.4 Election States in the web-front-end

The election display on the web-front-end can be in one of 6 states. They are described below and the UI is shown in figure 4.9.

- (a) **Not started:** After the election has been created but it hasn't started yet.
- (b) **Running:** When the election has started and hasn't ended. In this state the participants can cast the votes.
- (c) **Finished - Attendee:** When the voting time is over, then the election is finished.
- (d) **Finished - Organizer:** The Organizer sees the "TERMINATE ELECTION / TALLY VOTES" button when the election is finished.
- (e) **Terminated:** When the election is over, the organizer can terminate it by clicking the "Terminate / Start tallying button".
- (f) **Result:** After the election is terminated, the votes get counted and the result is computed, that gets displayed for in form of a bar chart.

4.2.5 Manage Election

When an election is in the Present or Future Category the organizer can click on the little pencil button in order to see some details of the election and modify it's components. For now we just have a UI implementation for an organizer to edit an election properties , and it doesn't make any real change to the election from a back-end point of view. In the future we could imagine the implementation of a message of type UpdateElection that would be sent every time an organizer modifies the properties of a created election . Then this message would have to be signed by the witnesses in order for the modifications to take place.

As you can see in figure 4.10 the organizer have access to buttons where he can modify the election name , the election question , the ballot options, the start time , the end time , the start date and the end date. He also have a button to cancel an election that will put the election end to now and will put it in past events.

In figure 4.11 the organizer clicked on the button Edit Election Name that will prompt an edit text we he can enter the new name for the election

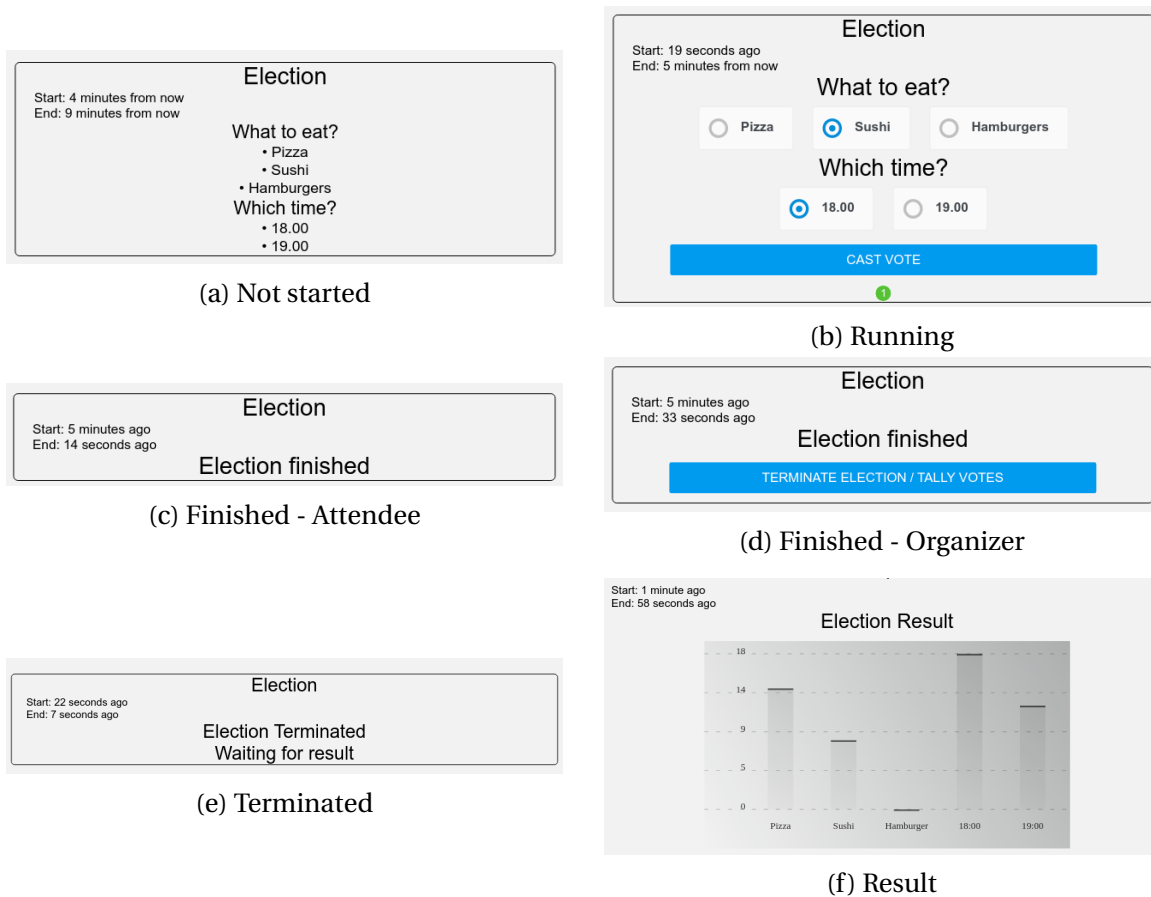


Figure 4.9 – Different Election Display States in the web front-end

4.3 Digital Wallet

4.3.1 Introduction

Proof-of-Personhood can be ensured through roll-calls which consist of in-person events where attendees can prove they exist. By participating in a roll-call the user receives a token consisting of a cryptographic private/public key pair. The token can be used in later events such as an election from the same organizer to attest that they are actually a person and that for example their vote is valid. The goal of this project is to provide the user with a way of collecting and storing these tokens while ensuring security, usability and privacy.

Usability is important because a user might join many LAOs and attend many roll-calls, which means that many tokens may need to be handled and this should not become overwhelming to the user. First, the wallet is initialized by a secret seed that only the user knows and as he participates in roll-calls tokens are added to the wallet. Then he can log out, which will erase the seed from the application and login again possibly on another device by importing the seed

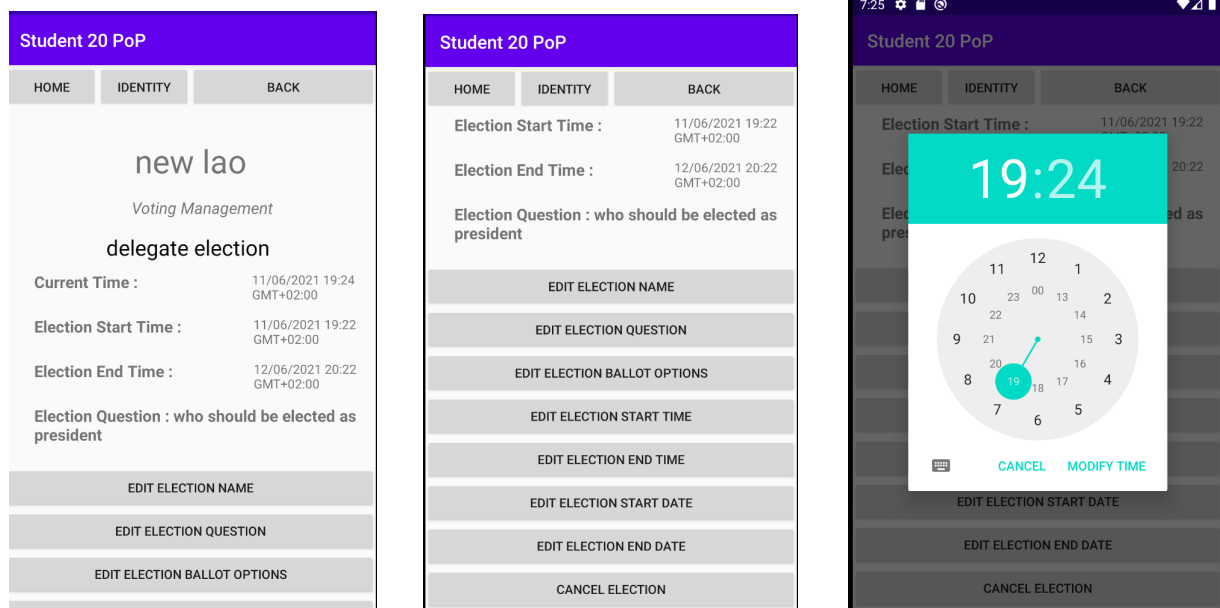


Figure 4.10 – Manage Election

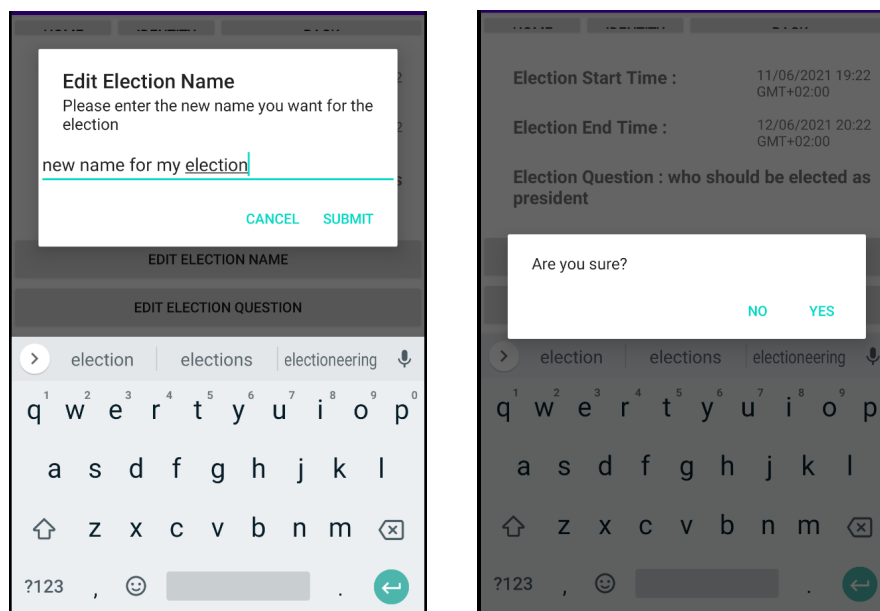


Figure 4.11 – Modify Election Name

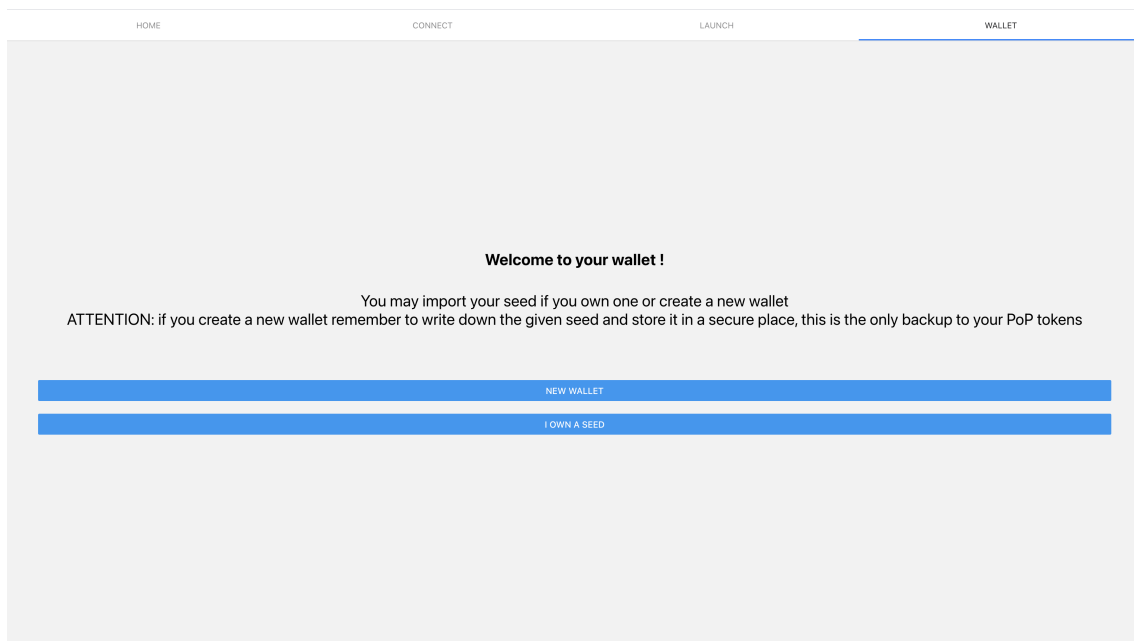
to recover the content of his wallet. The user should be able to save his seed easily. Therefore, instead of asking the user to remember the cryptographic seed, he only needs to remember a sequence of 12 English words that he can enter on his device and the application takes care of recomputing the actual seed.

In the next two subsections we are presenting how each of the two wallets we implemented look and can be used on the Web and Android front-ends.

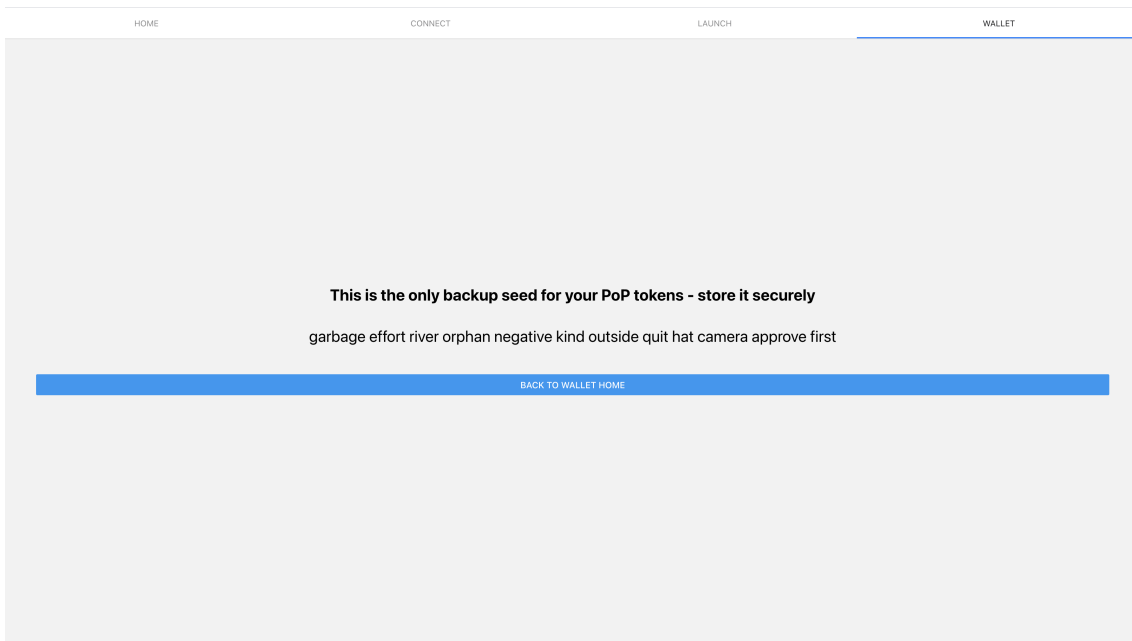
4.3.2 Web Interface

When the PoP application is launched, the user can join LAOs without the need for wallet synchronization. The wallet is mandatory once the user wants to attend a roll call, this since the token's public key used to attend the roll call is produced by the wallet using the secret seed. Wallet synchronization is thus necessary for this step.

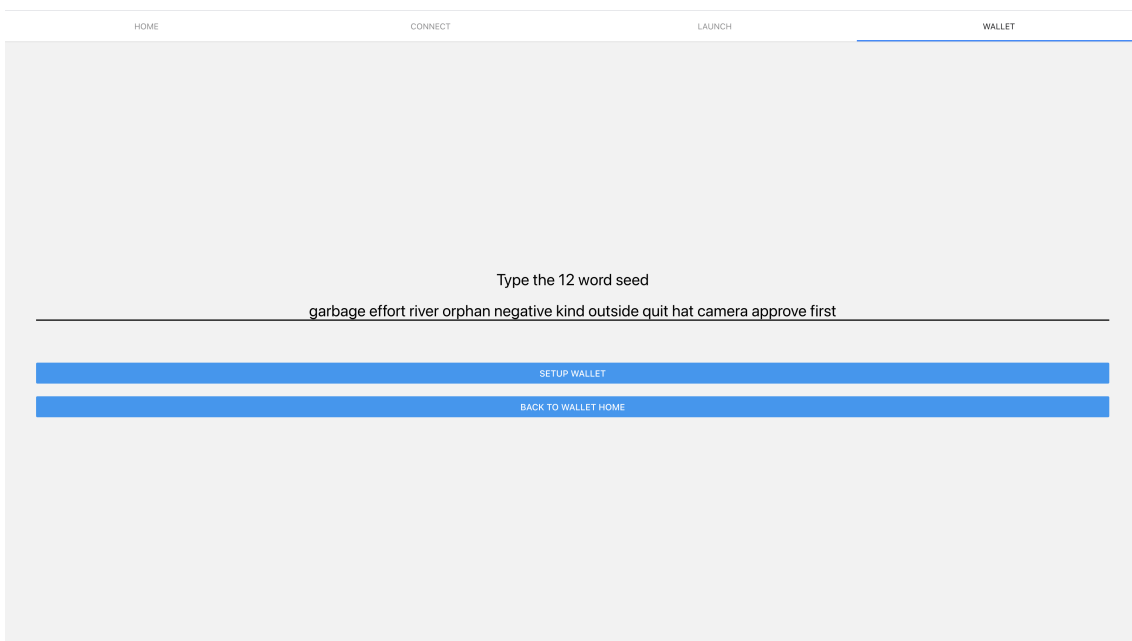
All the details regarding the wallet functionalities are covered in the implementation section, below we observe the synchronization flow of the wallet from a basic design viewpoint.



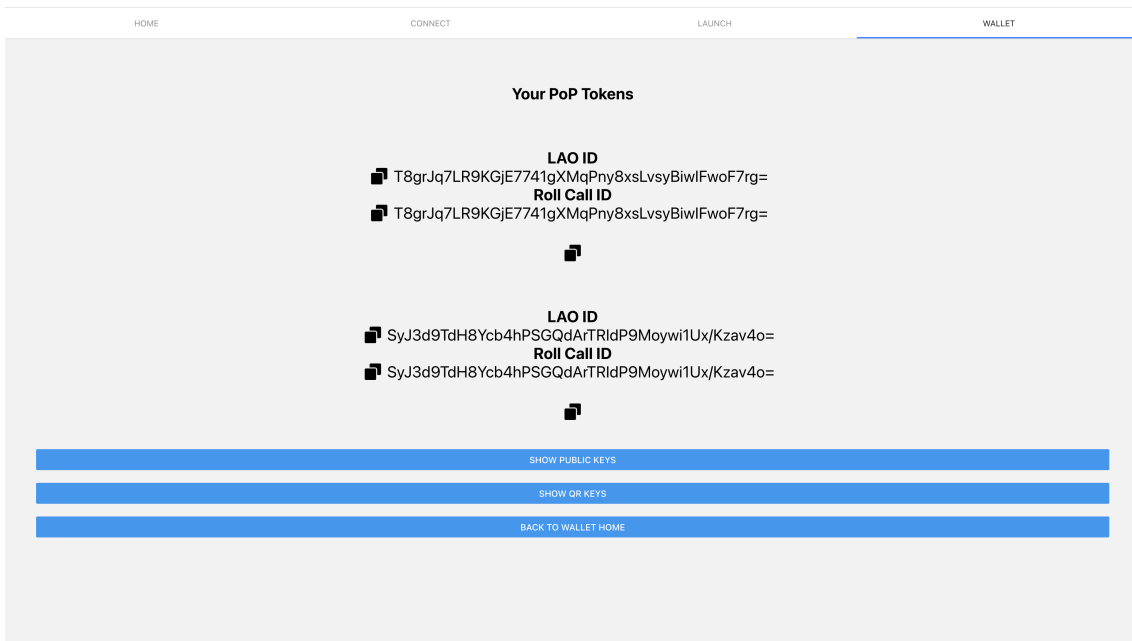
Create a new wallet and obtain your 12-word mnemonic.



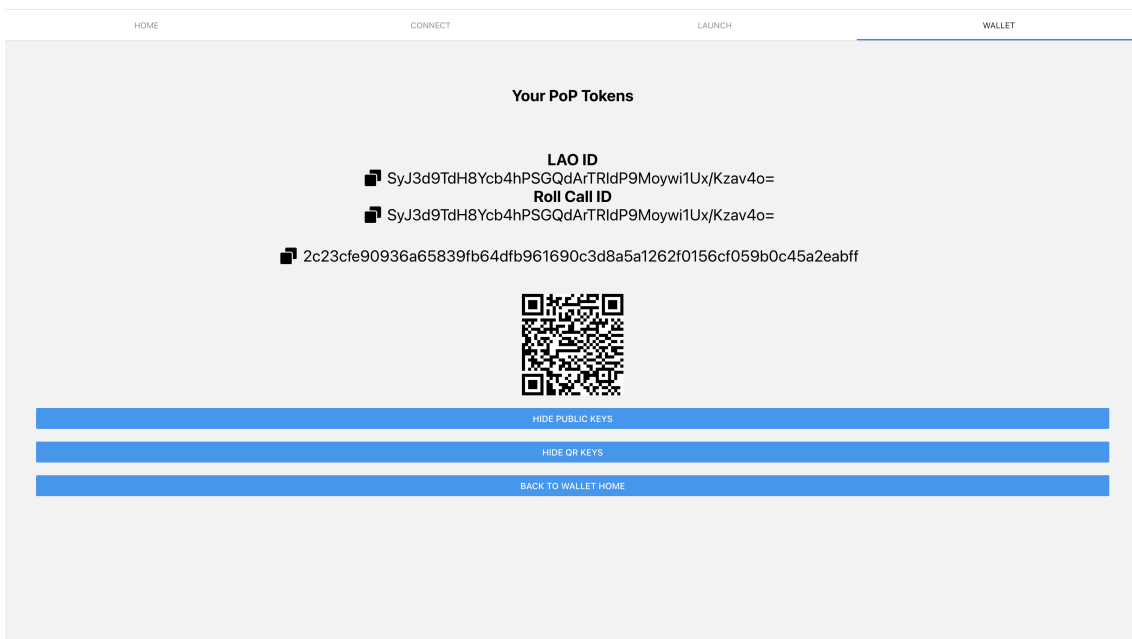
Setup your wallet with the obtained seed.



Once the user's wallet is synced and seed securely stored all tokens associated with a LAO that the user has previously joined are displayed on screen.



The public key of the PoP token (or its QR Code version) can be displayed optionally via the **show public key** (resp. **show QR key**) buttons and can be copied to clipboard via the **copy** button.



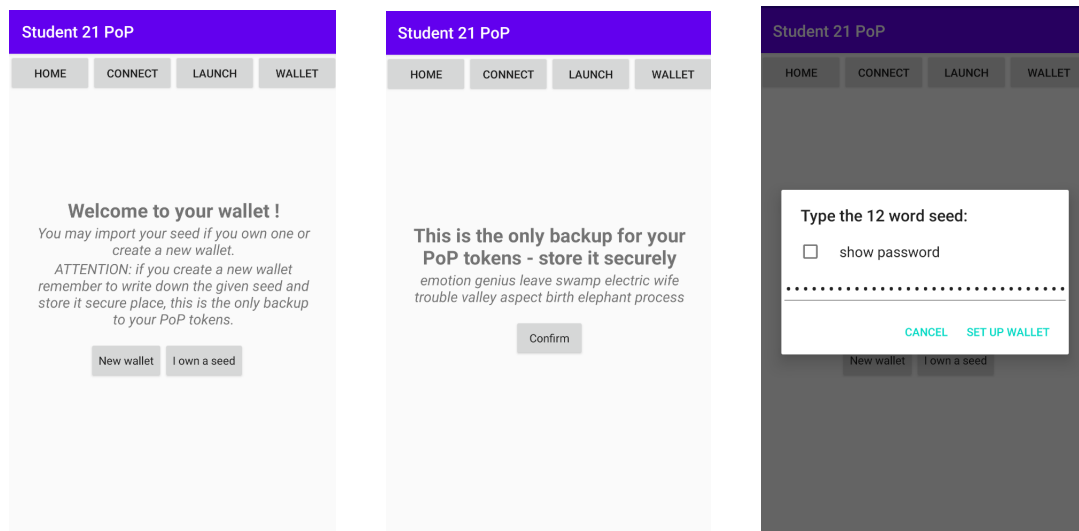


Figure 4.12 – Setting up the wallet

4.3.3 Android Interface

The first time we enter the application there is no initialized wallet which means the user can't participate in roll-calls and receive tokens. On Figure 4.12 the first screen is what is displayed at this point. Then the user can either choose to create a new wallet or import an existing one. In the former case the second screen is displayed and in the latter it is the third one. Note that in the current implementation this is only the step where the user can see the 12-word mnemonic on the interface. We decided to do it this way to reduce the risk of stealing the seed at the cost of a small usability downside.

After the wallet has been setup we get to the wallet home which looks like one of the screens in Figure 4.13 depending on whether the user is subscribed to at least one LAO. Indeed, the second screen displays the list of LAOs the user is subscribed to. Note that the user does not need to setup a wallet before subscribing to a LAO but is required to do it before participating to roll-calls.

When clicking on a LAO we move to the first screen in Figure 4.14 showing the list of closed roll-calls the user attended along with related information. For each of them he can look at the PoP token he received and at the list of the users' public keys who also attended. These correspond respectively to the second and third screen in Figure 4.14.

Lastly, on Figure 4.15 the user can logout which means that the local copy of the seed is destroyed and the content of the wallet is emptied. The user can recover his tokens when importing the seed by using the "I own a seed" button on the welcome screen and he can do it on any other device as well.

Note that in order for the user to display the tokens previously received by a certain LAO, the

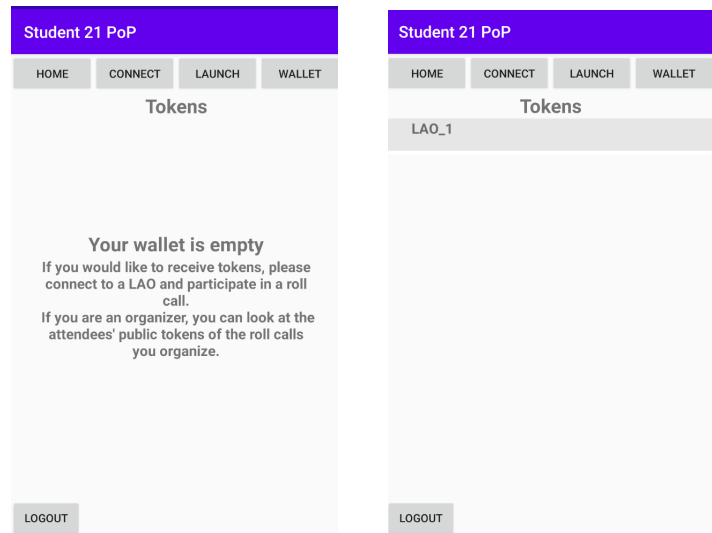


Figure 4.13 – Wallet Home

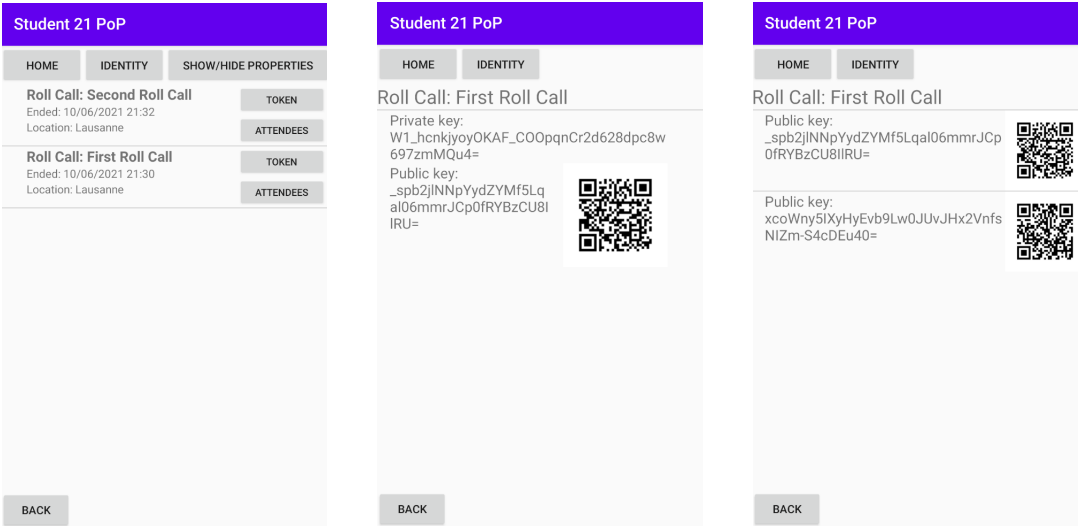


Figure 4.14 – Tokens and List of Attendees

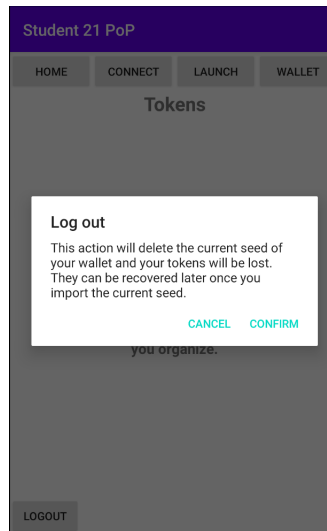


Figure 4.15 – Logout

user needs to be subscribed to the LAO. Therefore if an existing seed is imported on a device which is not subscribed to this LAO, the wallet home will not include the LAO in the list. We chose to have this design because what we want is to recover the tokens for a given seed, but the seed can only be used to generate tokens given a LAO id and a roll-call id. In other words the seed itself does not give us any information if we do not know these two ids. Therefore we need a way of knowing the LAO/roll-call pairs to apply the seed and recover the tokens of the wallet. The best choice we thought of is to require the user to be subscribed to a certain LAO to get its history and recover the tokens the wallet had from the roll-calls of the LAO.

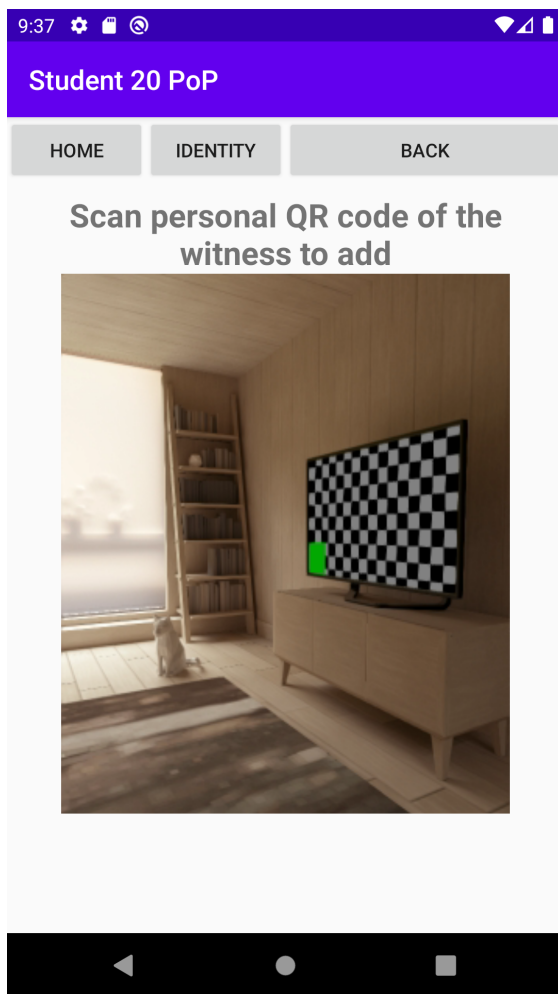
4.4 Witness UI

This semester a Witness UI was implemented that enables the organizer of a Lao to add new witnesses to a Lao by scanning their QR code by going in the Show/Hide properties of the Lao. We also added a new tab Witnesses Messages where the attendee are able to see all the messages that need to be signed. For the attendees that are a witness they will be able to sign those messages manually.

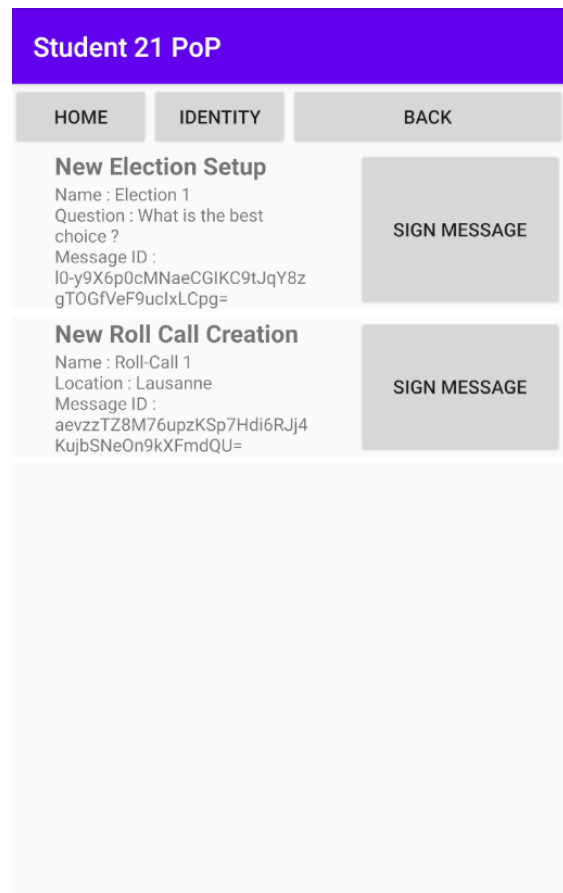
In figure 4.16a we have the camera that opens after the organizer clicked on the add witness button.

In figure 4.16b we can see that after creating a Lao and setting up an election the corresponding messages that need to be signed appear in the tab Witness Messages.

In figure 4.17 the user first tries to sign the message as a Non-Witness which prompts an Alert message , and then tries to sign it as a Witness which succeeds.



(a) Add New Witnesses



(b) Witness Messages

Figure 4.16

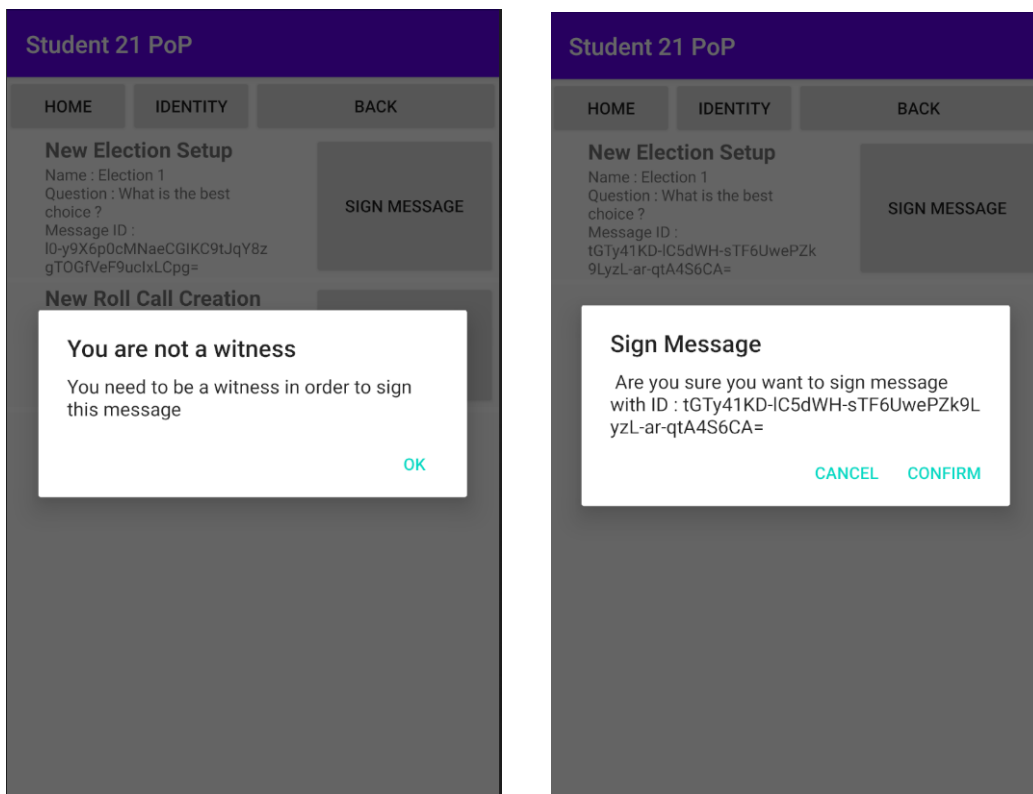


Figure 4.17 – Sign Messages

Chapter 5

Implementation

5.1 E-Voting

5.1.1 Back-end 1 - Go

The first back-end of this project was implemented in Go using the version Go 1.15. All the instructions to run the server can be found in the README file on GitHub. At the beginning of the semester, only the organizer server was implemented and it was only able to communicate with attendee clients. New features were implemented on the organizer server such as handling more messages. All the new features will be described in the subsequent sections. Another type of server was implemented: the witness server. This is a necessary component for the e-voting part of the project. And, the communication between the servers was also implemented.

Organizer Server

The organizer server is launched using the command:

```
./pop organizer -pk <public-key> serve -cp <client-port> -wp <witness-port>
```

<public-key> is the base64url encoded organizer's public key.

<client-port> and <witness-port> are optional parameters with default values 9000 and 9002 specifying on which ports to open websockets for the clients and witnesses respectively. The client websocket opens at `ws://<address>:<port>/organizer/client/` and the witness websocket opens at `ws://<address>:<port>/organizer/witness/`.

Witness Server

The witness server is launched using the command:

```
./pop witness -pk <public-key> serve -org <organizer-address> -cp <client-port>  
-wp <witness-port> -ow <other-witness>
```

<public-key> is the base64url encoded witness's public key.

<organizer-address> must include the organizer's address and the port it is using for witnesses. The default value is `localhost:9002`. If the witness does not succeed in connecting to an organizer, the server does not start.

<client-port> and <witness-port> are optional parameters with default values 9000 and 9002 specifying on which ports to open websockets for the clients and witnesses respectively. The client websocket opens at `ws://<address>:<port>/witness/client/` and the witness websocket opens at `ws://<address>:<port>/witness/witness/`.

<other-witness> can be used as many times as necessary to specify the address and port of any other witness servers currently running to which to connect.

Components

When a server receives a message, it processes the message and then sends an answer back to the sender. The answer of the server can either be positive, if the message was processed successfully, or negative. The answer can be negative because of many reasons, the message does not follow the protocol, some ID or signatures are wrong and so on. In that case the server sends an error to the clients with a detailed description of the error.

As you can see on Figure 5.1 the go back-end can be divided into four main components: The Socket, the Hub, the Channel and the Inbox. The socket is the components that communicate with the clients. It receives all the incoming message and passes it to the Hub. And, it also sends the answer to the client after the processing of the message. The Hub is the central part of the server. It is performing some validations on the message and it determines to which Channel the message was sent to. the Channel component also does some validation on certain messages, then it processes. It might have to write the message to its Inbox (if this is a publish message) and it might have to read all the message of its Inbox (if this is a catchup message).

At the beginning of the semester, the server had only one type of each of the components. But since a witness server is needed for an election and, a Lao and an Election are on separate channels, different types of each component were implements. Each type of the components will be described below.

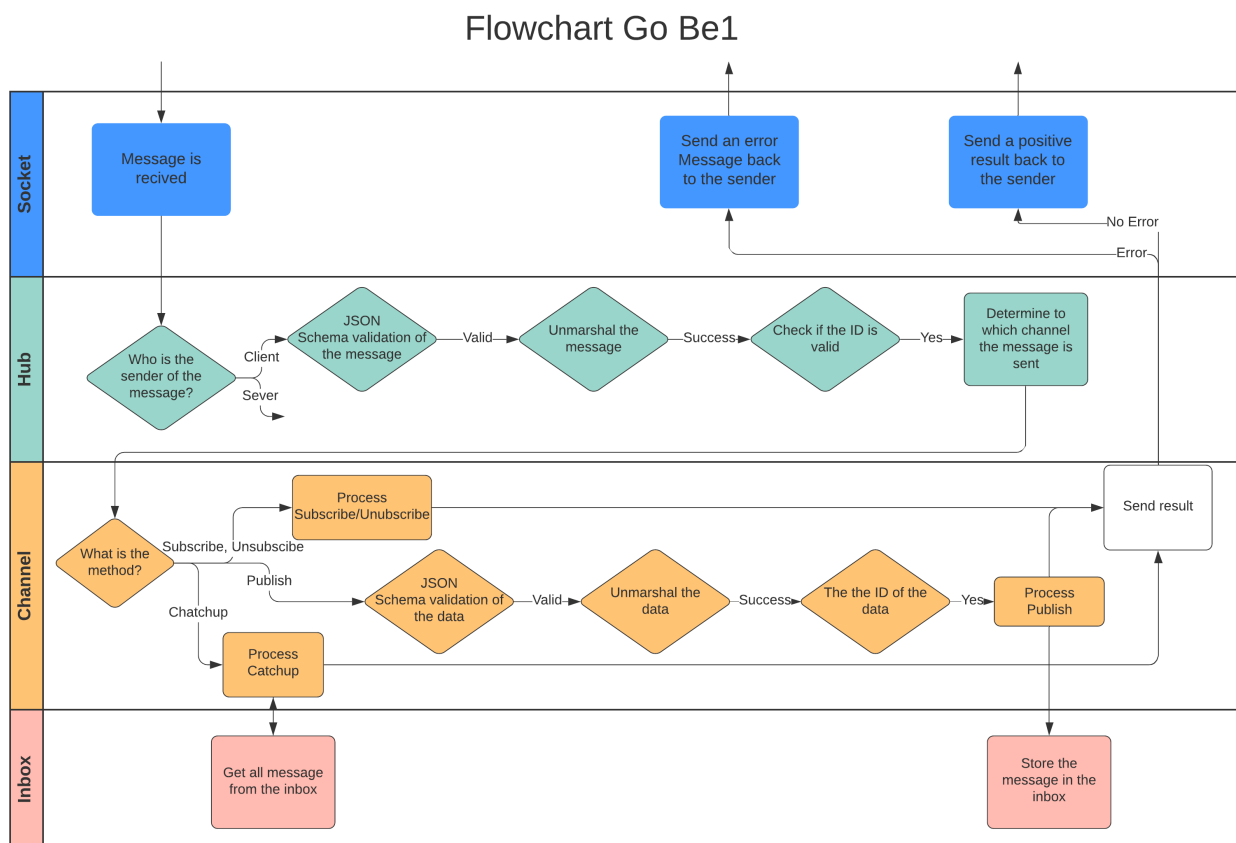


Figure 5.1 – Flowchart of the messages in the go back-end

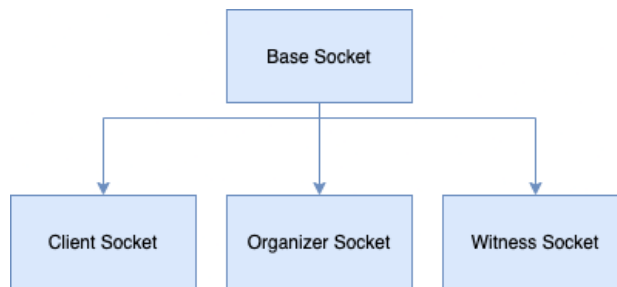


Figure 5.2 – The three different types of socket

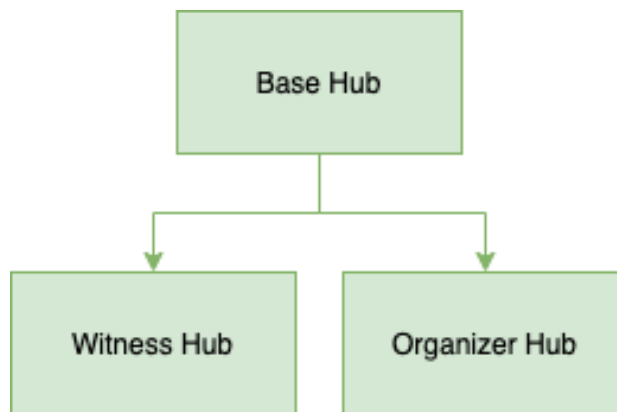


Figure 5.3 – The two different types of hub

Socket Websockets are used for all client-server and server-server connections. The `Socket` interface is used to represent these. Its main functions are `ReadPump()` and `WritePump()` which are called in go routines. The former sends any received messages to the hub and the latter sends messages passed to it by the hub to the server or client that the socket represents. If the message received from the socket is invalid, the hub sends back an error using the `SendError()` method, otherwise it uses the `SendResult()` method. There is a `baseSocket` structure which implements these methods, the sockets representing clients, witnesses and the organizer all function in the same way. For the sake of clarity, however, we use type embedding in order to create the types `ClientSocket`, `WitnessSocket` and `OrganizerSocket`, which contain a pointer to a `baseSocket`, whose `socketType` is defined accordingly. This also allows us to create or modify methods specific to certain types of sockets if necessary.

Hub A Hub has three main methods: `Start`, `Recv` and `RemoveClientSocket`. When we start the server, a new hub is created using the `Start` method. When a socket receive a message from a client it will pass it the the hub by calling the method `Recv`. Each hub has a channel of incoming messages, a list of channel and the public key of the owner of the server.

There is two types of Hub, the `Organizer Hub` and the `Witness Hub`. When an organizer server is created, an `Organizer Hub` is start and when a witness server is created, a `Witness Hub` is

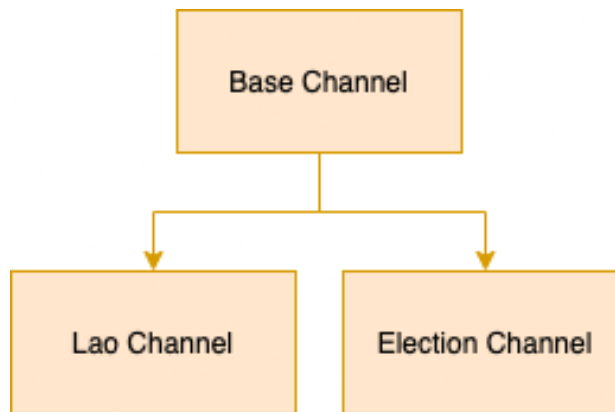


Figure 5.4 – The two different types of channel

started. The two hubs have a lot of elements in common, that's why they have a pointer to a `Base Hub`, which contains all the similarities between the two hubs. The main difference between the two hub is the way they handle the messages. The `Witness Hub` does less checks on the messages than the organizer hub, as long as the message follow the protocol it will accept it and store it.

Each `Hub` try to marshal each incoming message, performs some checks and identify to which of his `Channel` the message was sent to by looking at the "channel" field in the message. And then, it will pass the message to the corresponding `Channel`.

Channel As described in the Background section, there is four different method of message that a client can send to the server: `Publish`, `Subscribe`, `Unsubscribe`, `Catch-up`. And each `Channel` has a corresponding method to handle each of those messages.

The two types of `Channel`, the `Lao Channel` and the `Election Channel`. Both channel have a lot of similarities. They handle `Catch-up`, `Subscribe`, `Unsubscribe` messages in the exact same way and the validations of the messages are done in the same way. That's why they both have a pointer to a `Base Channel`, which gather all those similarities.

When a `Catch-up` message is received, the `Channel` gets all the messages from its corresponding `Inbox`. And, when this is a publish messages, if it passes all the checks, the message is stored inside the `Inbox` of the `Channel`.

Inbox Each `Channel` has an `Inbox` where all the messages are stored. The `Inbox` can store a new message, get a message and add a witness signature to a message that is already stored.

Verification

Two type of verification were implemented during this semester. The first one is the JSON verification and the second one is the ID validation.

The JSON validation was implemented using the `gojsonschema` library [6]. This library was used because it is maintained pretty well and it is one of the fastest and most precise JSON schema in Go [5]. All the JSON schema in the protocol folder are imported into the library at run time and each time an incoming message is handled a by the server, a JSON validation is performed. `gojsonschema` library had some drawbacks. First, there were some problems to load the schema on windows machines. And, the library was trying to fetch the online URL "\$id" when loading the schema, which was a problem because the URL referenced is the master branch.

In a query, two elements can have an ID, the message and the data. Each ID corresponds to a SHA256 hash of some specific fields. A check was added to ensure that each time the ID corresponds to what is described in the specification protocol. If it is not the case, an error is send back to the client.

New events

Two new events were implemented during the semester: the election event and the roll call event. Implementing the roll call event was needed for an election because only the people who participate to an roll call event should be able to vote during an election.

The roll call event is composed of 4 messages: Created, Opened, Closed, Reopened. The handling of each of the messages was implemented. And once the roll call event is over the public keys of all the participants are stored on the `Lao Channel`

The election event being the second implemented event, it is divided into 4 messages `Election Setup`, `Cast Vote`, `Election End` and `Election Result`.

Election Setup: This message gets sent by the organizer on the `lao` channel in order to create a dedicated channel only for the election, the election channel id will simply be `root/lao_Id/election_Id` and will get sent to every subscribed member of the `lao` id so that members of the `lao` could potentially be part of the created election as well. After this is done we the server sets up a new election channel with a few important properties that will be useful from the remaining election messages, since we created a base channel that is also embedded in the `lao` channel we do the same thing with the election one too, other than that we have start and end fields, an ended field indicating if an election has ended, and a map of questions that map the question id to a struct question composed of it's id, the available list of ballot options, a mutex to avoid concurrency problems, the voting method and a map of valid votes that map string

representation of a person's Public Key to a valid vote struct composed of the timestamp which is when the person cast a vote and the list of indexes which is the choices the person made for the particular question. This message also gets stored in the message history of the lao so that if someone wants to catch up on messages it can have the election creation one too. If everything worked until this point a broadcast of the election setup gets broadcast-ed to all the clients, a result message also gets sent to the front-end part of the organizer too.

Cast Vote : This message gets sent on the election channel by the organizer front-end that was created after the election setup message. After doing the necessary sanity checks that is if the received message has all the valid fields for a cast vote message and if the vote cast has a created time that is before the election end time and that number of ballot option chosen is conform with the voting method that is only one vote if approval and one or more if plurality, we updated the valid votes field for the given question received, that is if the sender cast a vote earlier we update it or create a key value pair if this is his first vote cast for the given question, we also check if the question with the certain id exists, that is if the election setup considered it a valid one One more thing we do is that we check that the received vote was sent by a member of the roll call that had to occur previously. We then broadcast this message to all the members of the election channel and respond with a result message to the front-end that sent this message in the first place.

Election End: This message is also sent by the organizer front-end to the election channel received by the back-end. After performing the sanity checks, that is if all the fields of message are valid and the data field is of Election End kind and check if the creation of this message is after the end of the election we check if the registered vote hash received is the same as the one the back-end calculates that is we check that the votes the front-end registers are the same ones the back-end saved. If Everything is in the norms then we broadcast this message to all the clients and send a result message to the sender of the Election End message and save this message in the election history inbox. And We proceed with the Election Result message.

Election Result: After Election End was processed and sent away we gather all the votes for each question that is stored in the election channel specifications and construct a result message that will represent the winner of the election depending on the voting method, for now we display the candidate followed by the number of votes it has received in descending order in the number of votes. Then we construct the message that will be sent to the front-end and all members of the election channel.

5.1.2 Back-end 2 - Scala

During the semester, the Scala back-end was being refactored and cleaned up in order to make it clearer and to get it up to date with the initial state of the go back-end. Towards the end, it became possible to work on it. We created the election data types , i.e., data types representing election setup, cast vote, election end and election results.

Figure 5.5 – Election Setup Version 1 - Web Front-end

5.1.3 Front-end 1 - React [native]

The web front-end is built in Typescript and uses React [native] and Redux for storage.

Election Setup

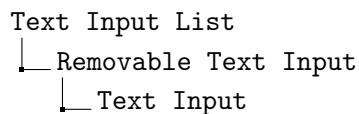
Creating the Election as an Organizer was the first needed functionality in the whole election domain. We started with a basic UI, which we gradually improved based on feedback from my PR-reviewers. Then we implemented the communication logic. The main questions to answer were:

- How to capture the input from the user?
- How to divide the UI into separate components?
- How can I dynamically add more Ballot Options?
- How do I nest components?
- How do I keep state in the components?
- How can I lift the state up (from child to parent component)?

The first election setup version (figure 5.5) was faulty in many regards. Mainly, it didn't allow the user to choose the voting method and the user was only able to remove the last ballot option,

not any arbitrary option. These issues were addressed and fixed in version 2 (figure 4.4) as well as the option to have multiple questions in each election. In order to build the feature of being able to delete each individual ballot option in the election setup individually, we had to get state management under control. Because what seems to be a simple component in figure 4.4, is actually three components which are wrapped inside of each other.

The first component is the **Text Input** field which we also use in the question section, the next component is the **Removable Text Input**, which uses the text input component, and adds a trashcan to the side of it. The final component, **Text Input List** creates a list out of the sub-components and keeps state what input is written where and which trashcan gets clicked. Hierarchy:



The interesting thing is that since the top component keeps the state, it needs to pass down the function which updates the state of the text all the way to the text input component. So each time a user modifies a text field, an *onChange()* method gets triggered inside the **Text Input** component which uses the function from **Text Input List** that was passed down which updates its state. But the function with the state of the trashcan is only passed down to the removable text input components. With this system each component can use the function of the parent component when something changes inside the component.

The top **TextInputList** component also automatically adds a ballot option to the list of options as soon as the user starts writing in the last currently open ballot option field.

This functionality was inspired by Doodle.com (<https://doodle.com/create/options#text>), which have very good usability in their forms.

Communication with the back-end

Since the communication part was still being worked on by our supervisors in the beginning of the semester it was at about a month into the project when we were able to start trying to communicate with the back-end. This was easier said than done, especially since testing was a very tedious process because of the base64 problem described in section 2.1.3 where the back-end couldn't handle channel encoding which contained a slash "/". Therefore, (before we switched to base64url), to test the communication implementation we had to create many *Lao's* until we had one which didn't contain a slash, then we had to create many elections until we had one where the ID didn't contain a slash (since the election id is used in the election channel), and then we were able to debug the other errors. After many tries and errors and debug sessions with the back-end team it finally worked and works consistently now.

Cast Vote

We decided to use check-boxes for the user to click when voting. Similar to the **TextInputList** component, here we built a **CheckboxList** component. The most difficult part was to implement the ability that when the components is called, you can specify how many options should be selectable, and the behaviour of the component changes based on that.

When a user can only select one ballot option: Each time the user clicks a new checkbox, the selected option automatically jumps from the one that was clicked to the new one the user just clicked (regular radio button behavior).

When a user can click two or more options: User has to un-click the previous option in order to click new options if the max amount of clickable options is was implemented.

Election-end

Here, the difficult part was to store the casted votes, in order to hash them as stated in the protocol. We were required to store the last vote of each voter, then sort them by messageID, and finally hash them and send to the back-end. We decided to use the store that was already available, the redux event storage. We added the field "casted-votes" to all election-event and stored for each voter, the last vote of that voter as well as the message ID that came with the vote. Each time a voter votes again, his vote will be overwritten in the storage. The message ID is important as it allows us to sort the votes in the same order, on the front-end and on the back-end.

Election Result

The main challenge in the election result was to get the election ID from the channel. Because the protocol states that in the setup-election, cast-vote, and in the election-end message, the election-ID is present in the message data. But for some reason, in the election results message election ID is not present, which meant that we couldn't read it directly out of the message data. The missing ID fields can be seen in table 5.1.

Therefore I had to find a way to read it directly out of the channel (since the channels are built out of the IDs: /root/laoID/electionID). The problem was that the channel was in a higher data-layer than the election data so accessing it wasn't as straight-forward as it may sound. After some initial struggles to get the channel field where we needed it, we finally figured out how to do it. For each incoming message we stored the channel in the message data. But for the outgoing messages, we made sure that it wasn't included such that it doesn't break the protocol. In the table you can see the protocol specifications. Currently, there are discussions about this on GitHub under this issue: https://github.com/dedis/student_21_pop/issues/309.

object	action	id	lao	name	vers	created	start	end	questions	votes	reg_votes
election	setup	X	X	X	X	X	X	X	X		
election	cast_vote	X	X			X				X	
election	end	X	X								X
election	result								X		

Table 5.1 – Election protocol fields

React Native

Since the project is also supposed to run on iOS and Android phones, we always tried to make our components React Native friendly, meaning they should also work on mobile platforms. There are good cross-platform components available, or almost every UI component needed, except for the date and time picker. We tried to have one date picker which worked for all platforms but didn't arrive at sufficiently good solution. Then we tried another approach: creating three files, one for each of the platforms Android, iOS and the Web. That is possible by changing the file name from *datepicker.tsx* to *datepicker.android.tsx*, *datepicker.ios.tsx* and *datepicker.web.tsx*. Then when it compiles for a specific platform it uses the matching file. When we thought we had figured it out, we tried to compile it on an iPhone but there were still many errors throughout the application which would have taken a significant amount of time to fix. As we were the smaller team of the front-ends, and needed to catch up to the Java-front-end in terms of e-voting functionality we decided to put the react native support on hold and focus on the core E-voting functionality. That's the reason why [native] is shown in brackets throughout this report as it currently needs more work in order to compile it on mobile phones.

5.1.4 Front-end 2 - Android

The second front-end that this project develops is an android application written in java. Here we detail the logical organization of this sub-system.

Activities

An activity is an application component. Currently in our application we have two activities : Home Activity and LaoDetail Activity. The Home Activity represents the home page of the application where the user can access his wallet , launch a new Lao and see all the created Lao. The LaoDetail Activity represents the activity when we open a Lao so it contains everything related to a specific Lao.

Fragments

Fragments are a reusable portion of the app user interface, it contributes its UI to the activity it lives in. It has the advantage to be reusable across multiple activities.

Election Setup The election setup uses EditText to allow the user to input the textual information about the election. Date pickers for intuitive input of dates, Time Pickers a spinner for the different voting methods in the future, and a switch that allows to specify if the organizer wants to allow write in. At the bottom a LinearLayout lets the user add all the ballot options they need. The add button inflates a fragment containing an EditText for the ballot option.

Election Display After an ElectionSetup message is sent to the server and then broadcasted back to every client that are subscribed to the Lao channel , every client will handle that broadcast by updating the Map of elections with this new election. To display the election we then use an BaseExpandableListAdapter class called EventExpandableListViewAdapter that is used to provide data and Views from some data to an expandable list view of events. So in this class that observes the list of elections , everytime a new election is added to the list it will return a View for the corresponding election through the method getView().

Cast Vote This fragment uses the usual TextViews to display the relevant information about the election. Then the lower part of the screen, which is delimited by the space taken by the aforementioned TextViews, is occupied by a ViewPager2 which allow for views to be horizontally swiped through. Each of those views are comprised of a question and the ballot options associated with it. The behaviour of each of theses views is set in a custom adapter that extends RecyclerView.Adapter. The ballot options are displayed in a ListView which is scrollable thus rendering our app capable to handle any - reasonable - number of options. We do not need a custom adapter for this ListView because it only needs to display all the strings in a List object. The ListView can handle any number of selectable candidates therefore paving the way for future improvements of having different voting methods. For now our project only proposes plurality voting which only needs one selected option per question. In order to avoid users missing the fact that the election has several questions, we already added an indication with circles that a swipe is available. This uses CircleIndicator3 library, which is a lightweight solution for this kind of indicators. The fact that we took the option number 3 is because it is designed for use with ViewPager2.

Election Management On this fragment, the user can modify the properties of an election. It uses some TextView to display the properties of the Election such as the start time , the end time and the election title and name. It uses some Button such that the organizer can edit this election properties. It also creates a TimePickerDialog and a DatePickerDialog when the organizer tries

to modifies the time or the date. When the organizer tries to modify the name or the question it creates a AlertDialog whith an EditText that the organizer can enter.

Election Result On this fragment, the user can see the results of an election. The different ballot options and their associated number of votes are displayed on a ListView. This list in itself is contained in a ViewPager2 that will seamlessly allow for future display of election with multiple questions. We define custom adapters, one that set the behaviour inside the ViewPager2 and one for each of the ListView element.

Testing

To tests the objects we created, we use the standard JUnit 5 library. For the moment the model class Lao and Election are completely tested. We also created some tests for the protocol class object such as ElectionQuestion, ElectionSetup, ElectionVote, CastVote, UpdateLao, CreateLao and StateLao. For the Android tests, we were able to create some tests with the help of the Espresso library and especially their integrated tool Espresso Test Recorder that helps you to write some UI tests by trigerring some components/buttons of your application and make sure that it behaves accordingly. Espresso Test Recorder lets you record your interactions with a device and then takes the saved recording and automatically generates a corresponding UI test that you can run to test your app. [2]

5.2 Digital Wallet

The digital wallet structure for both front-ends (Web / Android) is implemented in the same way. The common functionalities are thus explained in the same section, while specifics to the Web or Android environment will be addressed in separated sections. We start explaining how a HD-Wallet is constructed by giving a general idea of its functionalities, explaining how to generate the wallet seed and describing how PoP Tokens are actually generated and recovered.

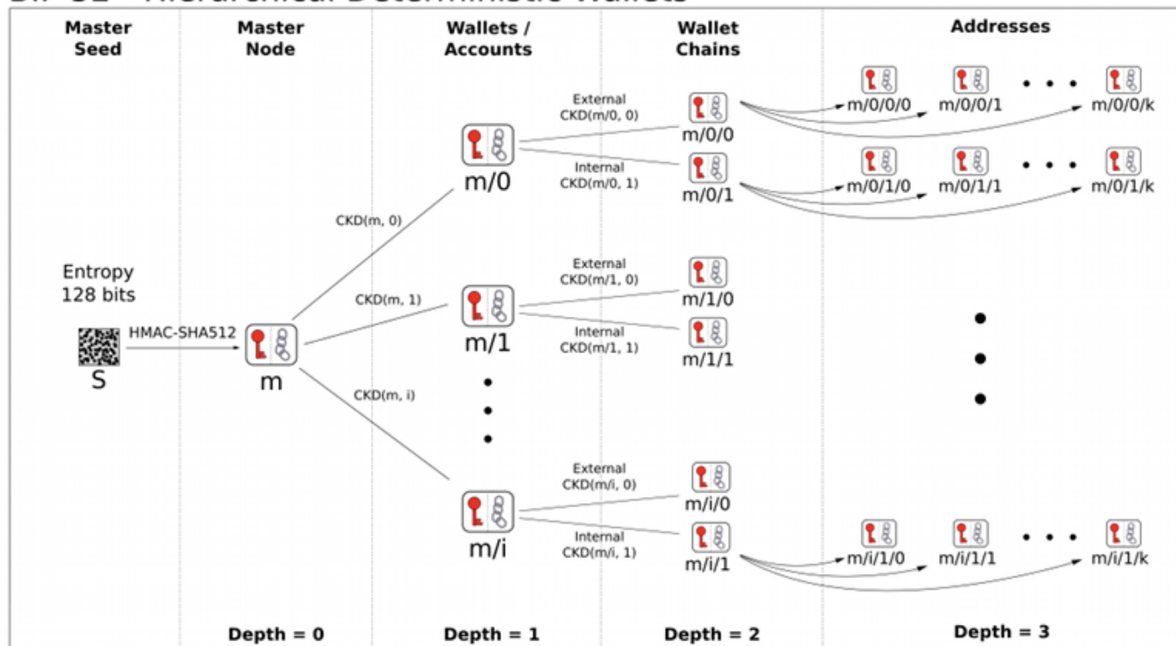
5.2.1 Structure - HD Wallet

One could implement a simple wallet that stores a public key for each roll-call the user participated in, but this implementation would have had one major drawback: it's doesn't scale. Thanks to path derivation we can derive multiple keys from a single secret value, the seed. So that it is only necessary to store the seed.

The hierarchical deterministic wallet (HD Wallet) defined by Bitcoin's BIP-32 is nowadays one of the most advanced form of deterministic wallets. This wallet automatically generates a hierarchical tree-like structure of private/public key pairs allowing to derive all keys from the master key. The master key is the previously mentioned seed.

Derivation diagram

BIP 32 - Hierarchical Deterministic Wallets



Child Key Derivation Function ~ $CKD(x,n) = \text{HMAC-SHA512}(x_{\text{Chain}}, x_{\text{PubKey}} || n)$

Example

A possible path for key-derivation is - **m/1/120/300/...** - used to access the specific key pair in the tree. Each level can be indexed by a 31 bit integer.

Remark

BIP-32 offers two possibilities for path derivation: hardened or non-hardened. With **hardened** derivation there it is not possible to prove that a child public key is linked with a parent public key as opposed to **non-hardened** where this is actually possible. A hardened key has the advantage of being safer since a leak of a private key never risks the compromise of the ancestor keys in the tree. On the other hand a non-hardened one could have interesting use cases where a child key could be linked to a parent/sibling key to prove that they belong to the same owner. This implementation uses the **ED25519** curve instead of the usual secp256k1 and therefore only hardened derivation is supported.

5.2.2 Structure - Importing and exporting seed

Instead of generating a random number which is difficult to remember we encode the seed in a more easier way for humans to securely back-up and retrieve their wallet. The common approach is to use the mnemonic 12-word sentence method standardized by Bitcoin's BIP-39. Essentially, we convert the random seed in a 12-word mnemonic (e.g. wolf juice proud gown wool unfair wall cliff insect more detail hub). This mnemonic is then converted to the actual master key (seed) by **bip39** libraries (explained in more detail in the specific-Web/Android sections).

5.2.3 Structure - Generating PoP Tokens

The PoP tokens are simply a public/private key pair. After initializing the wallet with the corresponding seed we can generate all tokens of the user with the path:

m / purpose / account / LAO_id / roll_call_id

- Purpose is a constant set to 888 to distinguish from the subtrees of other applications.
- Account at the moment has a fixed value set to 0, but in the future if the user can have multiple accounts this parameter can be updated.
- LAO_id and roll_call_id levels are used to distinguish the origin of the various tokens. The path determined by these two parameters will generate the correct private and public keys corresponding to this specific LAO and roll call.

As previously mentioned, each level can be indexed by a 31 bit integer. Unfortunately since both LAO_id and roll_call_id are much longer than 31-bit (SHA256 hash with 32 bytes) it is necessary to convert this id into several levels of derivation path which do not fall in collisions between them.

The chosen approach was therefore to divide the ids into 3-bytes long sub-ids, each sub-id representing an index in the tree-path: thus creating a longer path but guaranteed to have no collisions. In particular, we chose at most 3 bytes as the size because it was the cleanest and most straightforward way to generate the path while keeping the number of bits of each index in the path under 31 per level.

5.2.4 Structure - Retrieving PoP Tokens

This implementation has the advantage of not needing to store every single generated token. In order to determine which PoP tokens a user owns we need:

- The wallet's master secret (seed).
- The event history of all the LAOs the user is connected to.

To retrieve all tokens the user owns, we iterate through the history of roll-call events in all of the LAOs the user is connected to and for each LAO_id and roll_call_id we test if the generated token's public key is present in the list of scanned public keys during the roll call.

5.2.5 Specifics

The Typescript and Java front-ends have implemented the same wallet **structure** but using different libraries for seed encryption/decryption and token generation. Furthermore both front-ends have a specific protocol to securely store the wallet seed.

5.2.6 Specifics fe1 - Typescript (React)

On the Typescript front-end the Digital Wallet needs to communicate with an IndexedDB browser database in order to securely store the secret key which is used to encrypt the wallet seed.

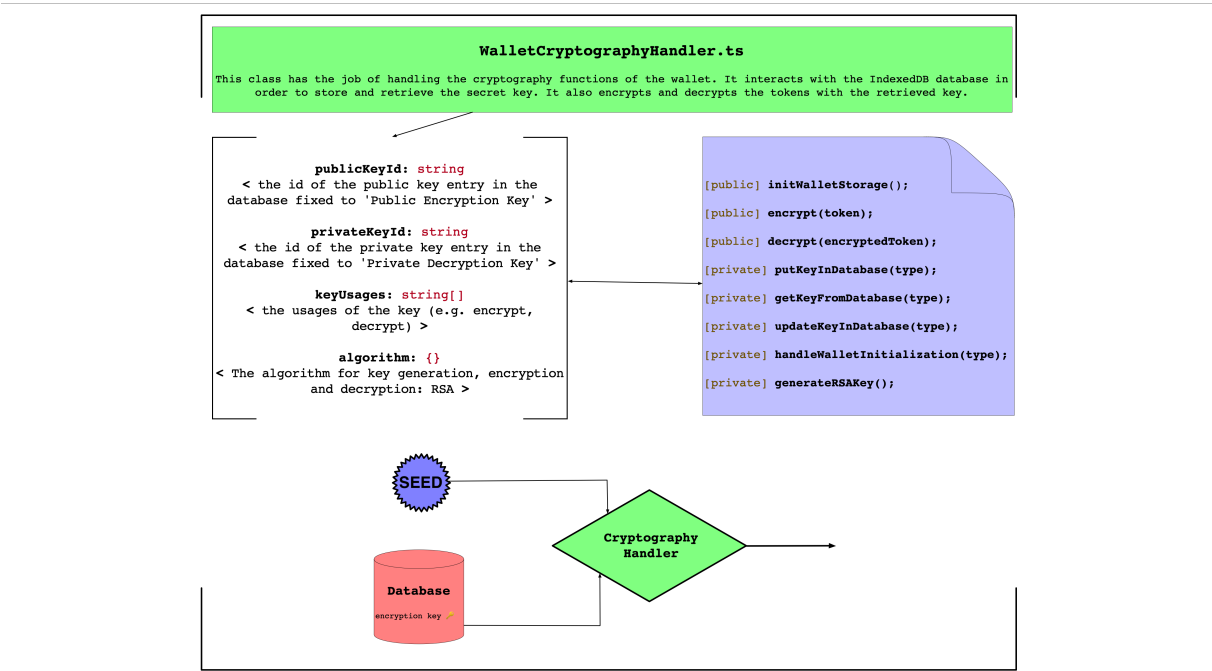
IndexedDB is used over other browser databases since it enables the storage of more complex objects with respect to strings or numbers. In this case it is needed to store an object of type **CryptoKey**.

The secure storing of the seed happens in the following way:

- The user inserts the 12-word mnemonic and sets up the wallet.
- A **key-value storage** on IndexedDB is created and **two RSA keys** are stored: the private decryption key and the public encryption key.
- The seed is then encrypted with this RSA key and stored in the Redux state. **The plaintext seed is never stored or cached.** Each time it is needed for token generation, its encrypted version is retrieved from Redux storage, then decrypted and used on the go.

All the above mentioned process is managed by the WalletCryptographyHandler.

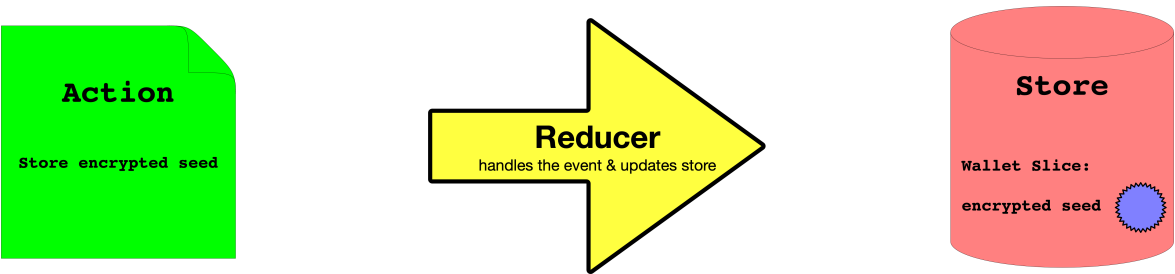
Implementation diagram



Seed storage

After encrypting the seed, this is stored in the Redux state. The redux state gives a single source of data-truth across the application. Once the user inputs the seed: **it triggers a store action which is handled by a reducer updating the wallet slice of the global state of the application.**

Redux state diagram



External libraries

- The utilized library for everything regarding **cryptography** is **SubtleCrypto** (link), offered by the Web Crypto API.
- To create a new wallet, the user is given a 12-word mnemonic. The generation of these 12-words, the dictionary-checks and the process to convert the mnemonic to the master key (seed) is managed by the **bip39** library (link).
- Given a correct path and a correct seed, the wallet has to be able to generate and retrieve the PoP tokens using the **ED25519 curve**. The public/private key pair generation is managed by the **ed25519-hd-key** library (link).

5.2.7 Specifics fe2 - Java (Android)

UI Implementation

In this subsection we are giving an overview of how the code of the wallet UI is organized in the Android implementation. The global application has two activities namely the HomeActivity and the LaoDetailActivity. The first one handles the functionalities that are general and do not involve LAO specific tasks, such as connecting a LAO or displaying the list of LAOs. The second one handles LAO specific tasks such as displaying the events of the LAO. This is the reason why we have the following placement of fragments into the activities.

HomeActivity: This activity has three fragments for the wallet, they are part of this activity because they do not correspond to a specific LAO

- **WalletFragment:** this fragment displays the first screen of the wallet where we can either choose to setup a new wallet or to use an existing seed. If we choose the second option a popup appears to enter the seed, which calls the importSeed function of the wallet. If we choose the first one we move to the next fragment.
- **SeedWalletFragment:** it displays the 12-words sequence representing the seed and the user should remember it. This fragment calls the exportSeed function so that the wallet encodes the seed into these words.
- **ContentWalletFragment:** after setting up the wallet this fragment displays the list of LAOs the user is subscribed to and if he clicks on a LAO we change to the LaoDetailActivity which handles LAO specific actions. If the list of LAOs is empty it displays a message with a little explanation of why this screen is empty or what it would contain once it gets filled. In order to display the list, this fragment builds a ListView using the LAOListAdpater which is the same one that was already implemented for the HomeFragment.

LaoDetailActivity: In this activity there are three fragments for the wallet.

- **LaoWalletFragment:** it contains the list of roll-calls the user attended from the LAO of the activity. To display this list the fragment uses the WalletListAdapter. For each roll-call it shows the date and time when it ended, the location and a "token" and an "attendees" button. The first button opens the RollCallTokenFragment and the second one opens the AttendeesListFragment.
- **RollCallTokenFragment:** it displays the token that the user received from the roll-call, namely the private key and the public key are displayed in base64url encoding and the public key is also displayed as a QR code.
- **AttendeesListFragment:** it displays the list of the public keys of the roll-call's attendees in base64url encoding and as a QR code. This fragment uses the AttendeesListAdapter for its ListView.

These fragments in the LaoDetailActivity contain a "BACK" button to ease user navigation.

Seed storage

To increase the security of the application instead of saving the seed in clear in the code we write them in the shared preference on Android. Thanks to the AEAD encryption/decryption we can assure the confidentiality and integrity of the data.

External libraries

One of the core library used to implement the wallet is **java-stellar-sdk** ([link](#)). This library allows to derive the key pairs as explained above (BIP-32). After having spent a good amount of time looking a library that meets the criterion for the ED25519 curve, we agreed that this library was the most reliable choice. The only problem was due to the fact that the specific function that we needed was in a class that is not accessible because its scope is package-private. Since the class contained about fifty lines of code we decided to directly include them in our code by putting them in a dedicated sub-package of the project with their corresponding license. Another important library used in the wallet class is **NovaCrypto** ([link](#)), allowing to generate and validate the mnemonics.

Chapter 6

Evaluation

6.1 Testing

6.1.1 Pop Parties

The goal of these PoP parties was to identify bugs when running a roll call on Android devices and on the web while running the server. Since everything was tested in more ideal scenarios we needed cross evaluations between the different front-ends(Java and Typescript) while the server was running in parallel, this way we got to more real life usage of such an application.

During the first PoP party we had a lot of things that went wrong, somewhat expected since everyone was working on it's part with not a lot of testing with the rest of the team in much detail, so first of all the web front-end was not able to run the general setup was not working so this needed to be inspected in more depth after the PoP party but was definitely not compatible with the APK that was running on different android devices. Next we tested if the roll call can work with everyone with an Android phone and one organizer, this also had some strange behaviour since only 2 out of 4 phones were able to join the LAO created by the organizer for the 2 other there was an error when scanning the LAO QR code provided by the organizer, this turned out to be as we think only a connection problem the phones were not connected to the internet or the connection was lost in those few moments. Next think we noticed was that the roll call couldn't be closed normally. Next think that was identified was the fact that once the app goes in the background(we open another app or just pause this one) the connection to the app was interrupted as well and the organizer was not recognized for it's role once we go back in the app and all the previous work could not be continued. The last problem/inconvenience encountered was the fact that once the organizer scans the attendees it can no longer show the QR code for the new incoming attendees that want to participate in the roll call.

During the second PoP party things were starting to improve, the web front-end was running

as well as the APK for the Android users but new problems were encountered. First of all the content of the the QR was not the same on the 2 front-ends and thus not cross compatible while running a roll call. One more bug was found and that is the the proposed start should be before the proposed end. Next thing we found was that the server allowed attendees that were not part of the roll call to cast votes. Following this we also identified that the check if the organizer was the one creating the election or a lao was not made and this anyone could perform those actions. One more problem was that not all catch up messages were not sent to the attendee that put the app in the background meaning that the server was not saving the entire message history.

During the third PoP party, the cross compatibility for joining a roll call between the web and java front-end still persisted. Creating an election seemed to work pretty well but a strange bug appeared when trying to create election a few times in a row, the server crashed and this was a bug that did not appear later on.

The last pop party was the smoothest yet, we could run a roll call both on the web and on android phones with the APK installed. We also tested on the android front-end that the wallet works well, after everyone has been scanned and after closing the roll-call, the attendees could go to their wallet and see their token as well as the other participants' public keys. After logging out the wallet and importing the seed again, the token could be retrieved. However since not everything was set in time we couldn't run cat votes yet on the APK but that seemed to work fine on the web.

6.1.2 Unit Tests

Wallet - Web front-end

Unit tests are provided for both the **WalletCryptographyHandler** and the **HD-Wallet** class (both explained above). The first test (cryptography) addresses how the seed is encrypted/decrypted and stored, while the second test (HD-Wallet) addresses the correct generation of the PoP tokens given a master key (seed) and LAO_id & roll_call_id.

Wallet - Android front-end

In order to test the implementation of the wallet we made some Android tests. More precisely we separately tested the logic of the wallet and its UI.

For the logic we made sure that the wallet recovers the same tokens it previously generated. What we did is to first create a wallet, then export the seed and generate a token for a given (LAO_id, roll_call_id) pair. By exporting we mean reading and saving the seed somewhere (e.g. on a piece of paper). Then we check that the same token (public/private key pair) is generated

for the same ids pair when importing the wallet again. By importing we mean setting up a new wallet initialized with the same seed.

For the UI thanks a the Espresso library we can check that the content (the button, the text, etc) of the wallet related fragment are display and are in the right position.

Wallet - Cross-testing

In order to be sure the 2-version programming of the Wallet is well respected and coherent throughout both front-ends cross-testing was done.

Given this initial 12-word mnemonic:

garbage effort river orphan negative kind outside quit hat camera approve first

both front-ends produce the same hexadecimal seed:

**010ac98c615c31a20a6a9fcb71c94642abdd4f662d148f81d61479c8f125854b
ac9c0228f6705cbdd96e27ffb2d4e806d152c875a5484113434d1d561e42a94d**

The utilized libraries for mnemonic/seed generation are thus coherent. Then deriving the PoP token from the following LAO_ids and roll_call_ids (with the above seed):

- **T8grJq7LR9KGjE7741gXMqPny8xsLvsvBiwIFwoF7rg=**
- **SyJ3d9TdH8Ycb4hPSGQdArTRIdP9Moywi1Ux/Kzav4o=**

gives the two public keys:

- **7147759d146897111bcf74f60a1948b1d3a22c9199a6b88c236eb7326adc2efc**
- **2c23cfe90936a65839fb64dfb961690c3d8a5a1262f0156cf059b0c45a2eabff**

which are coherent in both front-ends.

Chapter 7

Future Work

7.1 Election

At this stage, we allow only for one question per election, no write-in which are all features that could be added by future teams. Moreover we only offer the possibility of conducting plurality voting but in the future we could imagine more options and more complex voting form that could range from proportional to liquid - or viscous - voting forms.

7.1.1 Properties

At this point, our system only guarantees the integrity of an election. It is arguably the single most important characteristics because without it, any result is meaningless.

Privacy

For now, our system conducts the whole election process with clear message being communicated. A future improvement would be to encrypt the votes being sent, and to shuffle before tallying and decrypting which would unlink the vote from the identity of its sender.

React Native support

As it stands the web front-end doesn't compile for the native applications. If that would be added, then the application could also be used as a native iOS app besides the Android app.

Chapter 8

Conclusion

In this project, we built upon an existing codebase. Already able to create LAOs, we fixed the roll-call feature which allowed us to conduct pop-parties. Moreover, we added the possibility to run elections in which all verified attendees can participate. To be a verified attendee, users can go to a roll-call meeting where they get a token. We developed a secure and user friendly E-Wallet in which they can store all the tokens they received from different roll-calls.

We all learned valuable lessons during this project, not only useful coding skills, but also how to work efficiently in a developing team. Many of these lessons, we would not have learned during 'traditional' theoretic university courses and we are sure they will serve us well in the future.

Bibliography

- [1] Robert Dahl. *Democracy and its critics*. New Haven, Connecticut: Yale University Press, 1989.
- [2] Android Studio User Guide. *Create UI tests with Espresso Test Recorder*. URL: <https://developer.android.com/studio/test/espresso-test-recorder>. (accessed: 09.06/2021).
- [3] Devin G. Pope Keith Chen Kareem Haggag and Ryne Rohla. “Racial Disparities in Voting Wait Times: Evidence from Smartphone Data”. In: *The Review of Economics and Statistics* (2020), pp. 1–27. DOI: https://doi.org/10.1162/rest_a_01012.
- [4] Sharath Pankanti Salil Prabhakar and Anil K. Jain. “Biometric Recognition: Security and Privacy Concerns”. In: *IEEE Security and Privacy Magazine* (2003), pp. 33–42. DOI: <http://dx.doi.org/10.1109/MSECP.2003.1193209>.
- [5] Wild Blue Ventures. *Go JSON Schema Validator Benchmarks*. URL: <https://github.com/wbvinc/go-jsonschema-validator-benchmarks>.
- [6] xeipuuv. *gojsonschema: An implementation of JSON Schema for the Go programming language*. URL: <https://pkg.go.dev/github.com/xeipuuv/gojsonschema>.