École Polytechnique Fédérale de Lausanne

Anonymous Proof-of-Presence Groups for Messaging and Voting

Céline Camacho
Maëlyss Gilliard
Tuomas Juho Antero Pääkkönen
Kilian Schneiter
Adrien Shun Pierre Vauthey
Gabriel Fleischer
Robin Jaccard
Arnaud Wei-Xin Shih
Mohamed Badr Taddist
Jiabao Wen

Semester Project Report

Responsibles

Prof. Bryan Ford
EPFL / DEDIS

Pierluca Borsò
EPFL / DEDIS

Supervisors

Cristina Basescu
Noémien Kocher
Louis-Henri Merino
Pasindu Nivanthaka Tennage
Jean Viaene
Haoqian Zhang

EPFL IC DEDIS
Bâtiment BC
Station 14
CH-1015 Lausanne

January 7, 2022

Abstract

In an online system where people can interact with each other, there is always the question of whether we want a system where the users are anonymous, a system where we can make sure that each user is unique or a system with both properties. This poses a problem especially while implementing an e-voting system, because both properties are looked for, as we want each person to only have one vote, and to be free of outside pressure when voting.
A solution to this problem is Proof-of-Personhood, where we physically give tokens to the users without asking for identities. As the tokens are physically given, we can be sure that every given token belongs to a person, and that no one has multiple tokens.
We continued developing an application that ease the use of the Proof-of-Personhood concept. Our work was to solidify the exisiting codebase, to make sure that it is working correctly and to add both a consensus protocol and a social media feature on top of the already existing application.

Contents

Chapter 1

Introduction

Nowadays, with the evolution of artificial intelligence and machine learning, the capabilities of bots are constantly increasing. CAPTCHAs have tried, for years, to prevent bots from gaining too much power in systems. Those challenges were first made to be easy for humans and hard for bots, but now it is quite the opposite. CAPTCHAs are not only deteriorating the user experience but also failing their security goal. So, how can we prove that we are a real person online? This question is a key aspect of online democracy. In a perfect world, we would like to guarantee "one person, one vote" without identifying the voters.

The first challenge is the anonymity of the ballots. This is crucial as it can alter someone’s vote if they fear peer pressure, retaliation from the government or just does not want to have their vote publicly exposed.

The second and bigger challenge is to make sure that each person has only one vote. The system must be resistant to Sybil attacks - i.e., multi-accounts attacks. Even if a person is very rich or has a lot of computing power, they should have the same voting power as someone else. This also provides accountability as if someone acts as a bad actor, this person would be banned and would not be able to create another account after that.

Solutions exist to fight Sybil attacks and add some accountability but as we will see, they do not fit an online democracy. To achieve democracy, the system must provide certainty that nobody can vote twice, and nobody is wrongly excluded from the system.

Biometric identity - This might seems like a perfect solution to online identity. However, biometric identity has a few flaws.

- Inclusion: Someone can, for instance, lose a hand or an eye while working. This system is not fully inclusive.
- Privacy: There are some privacy concerns too as those biometrics need to be stored somewhere and someone might fear that their eye illness could be exposed to the public.
- Perpetual: Compromised biometrics are an issue because if it happens, you would lose

your identity forever as biometrics can not be replaced.

Proof of investments can mitigate Sybil attack and is very popular in permissionless cryptocurrencies. However, with this system your voting power depends on your investment. The richer you are, the more voting power you can have. This is definitely a solution that does not fit democracy.

Nowadays, almost all online applications have to find the right balance between accountability and anonymity. Some applications might ask for a phone number, some banking platforms for a copy of a passport or an ID card. Unfortunately, none of those solutions both prevent Sybil attacks and are non-privacy invasive.

Proof-of-Personhood

Proof-of-Personhood tries to solve this problem by providing a digital identity linked to a physical person without revealing their identity. How is this achieved? The Proof-of-Personhood concept can be achieved by organizing in-person events and giving present people a token proving they have attended. This token can later be used to prove their personhood.

Chapter 2

Background

2.1 Communication

The communication protocol was already well defined when we began working on the project. As it is used throughout the project, and as we slightly built upon it to implement the communication between two servers, we will present its important elements.

Publish/Subscribe pattern

All our communication between front-end and back-end follows the publish/subscribe pattern. With this pattern, the messages are not destined to one specific person, but are instead published on a channel and are accessible to everyone subscribed to this channel.
The subscriber can also ask to catch up to a channel to be up-to-date with the previously sent messages.
We chose not to use this pattern for the cross-server communication we added, as it was easier to just have: every back-end receives every message.

WebSocket

The communication protocol used in this project is the WebSocket protocol, as it allows for a bidirectional communication between clients and servers.

JSON RPC

To serialize the data sent on our application we are using JSON RPC [1]. It allows to easily specify the type and the parameters of the messages sent, as well as the parameters of their answer, and the possible errors.

2.2 Consensus

A consensus protocol is a mechanism in which multiple machines agree on a singular value, it is useful when multiple values are proposed at the same time to make sure that only one is accepted across the system.

Chapter 3

General Overview

3.1 Project overview

3.1.1 Team organization

The project team followed N-version programming, i.e., developing separate and independent versions of the same program. It consists of two front-ends and two back-ends.

The front-ends consist of TypeScript, using React [Native], targeting mobile- and Web-based execution and Java, using the Android APIs, targeting Android mobile devices.

The back-ends consist of Go, runnable on Mac or Linux servers or embeddable in iOS or Android apps and Scala, using standard system APIs, runnable as a server or embeddable in an Android app.

The students were divided in four subgroups, of originally similar sizes, to work on one technology each. Furthermore, the project itself was split in three parts: consensus, production-ready and social media. At the beginning of the semester, there was at least one student from each technology for each part of the project, apart from consensus in TypeScript, but the situation changed after a few weeks and there were some parts that were not worked on in some technologies. There were three students in the production-ready team, two students working on consensus and five students on social media. Each week consisted of a general meeting with the whole project team, of a technology meeting focusing on the language the students were working with and of a subproject meeting, dedicated to everyone’s own part of the project.

3.1.2 Roles in the system

Each local autonomous organization (LAO) in the system has three different roles. Each one has to be clearly defined to have the system to function properly and to counter an adversary taking one of those roles.

Organizer

The organizer is the most important member of the system. They will be responsible for most of the internal communication and are composed of a front-end (and indirectly of a back-end, even though it works in an automated way, so the organizer role can be managed by one person). They create the LAO which will be the basis of the system as well as the different events of that LAO, like roll calls and elections. The organizer also propagates information through broadcasts in the different channels to the other roles and handles the unfolding of the events. They have the most power in the system, even though the witnesses bring some security against a corrupted organizer.

Attendee

Attendees make up the majority of the participants in the system. They are the ones for whom the events are created and who can enjoy the benefits of the LAOs. Attendees are included in the LAO once they have shown up to a roll call and therefore provided Proof-of-Personhood. They have limited power inside the LAO, as their rights are fixed by the organizer, but are all equal, are all connected to the organizer and can also connect to witnesses to be protected from corrupted servers.

Witness

Witnesses ensure resilient communication inside the system. Each witness has two components: a front-end and a back-end, but like with the organizer role, one person is enough to ensure each witness role. Their purpose is to ensure delivery of the messages despite corrupted or malicious servers, whether those are organizers or other witnesses. For each published message on a channel, each witness also receives it in their back-end, which transfers it to their front-end. The witness then signs the message with their front-end and sends it back to the back-end, which propagates it to all attendees and other servers. This allows all attendees to verify the integrity of received messages and be protected against malicious servers inside the system.

3.1.3 Communication Architecture

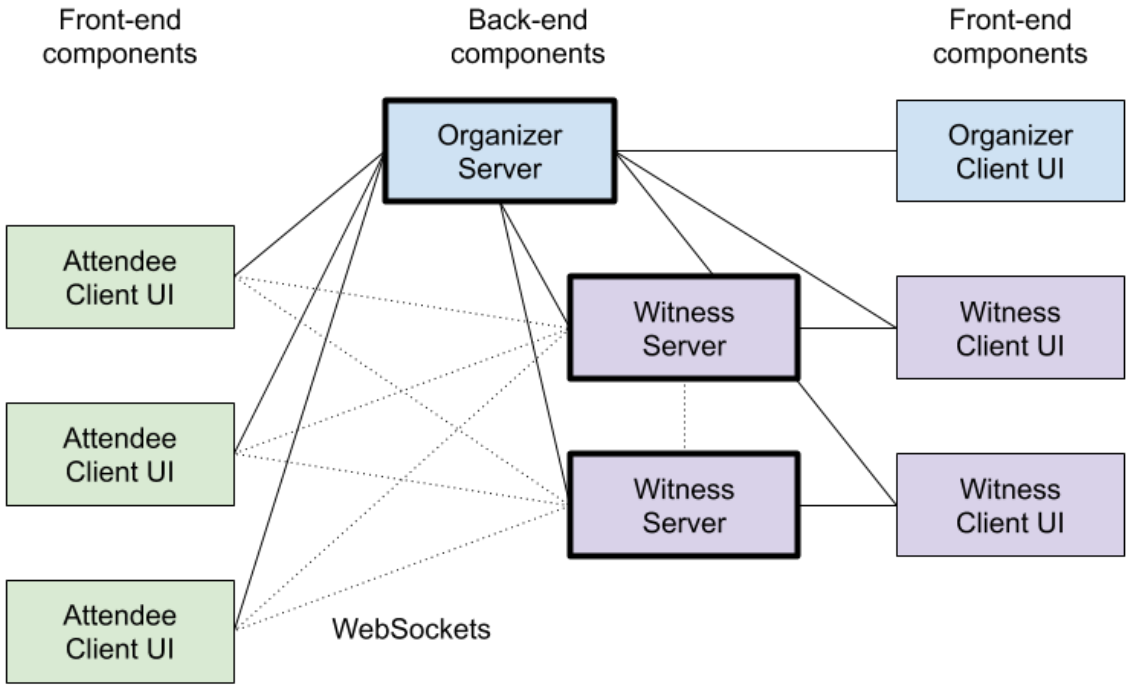

Figure 3.1: PoP architecture in terms of software components

The architecture of the project follows a client-server schema, the attendees being only clients, connected to the servers with WebSockets and the organizers and servers being composed of a client and a server, as mentioned above. The actual communication is done through a publish/subscribe pattern, as the clients can subscribe to channels, publish messages on them and receive the broadcast of a message published on one of the channels they have subscribed to. The servers take care of storing the published messages and of broadcasting the messages to the subscribers.

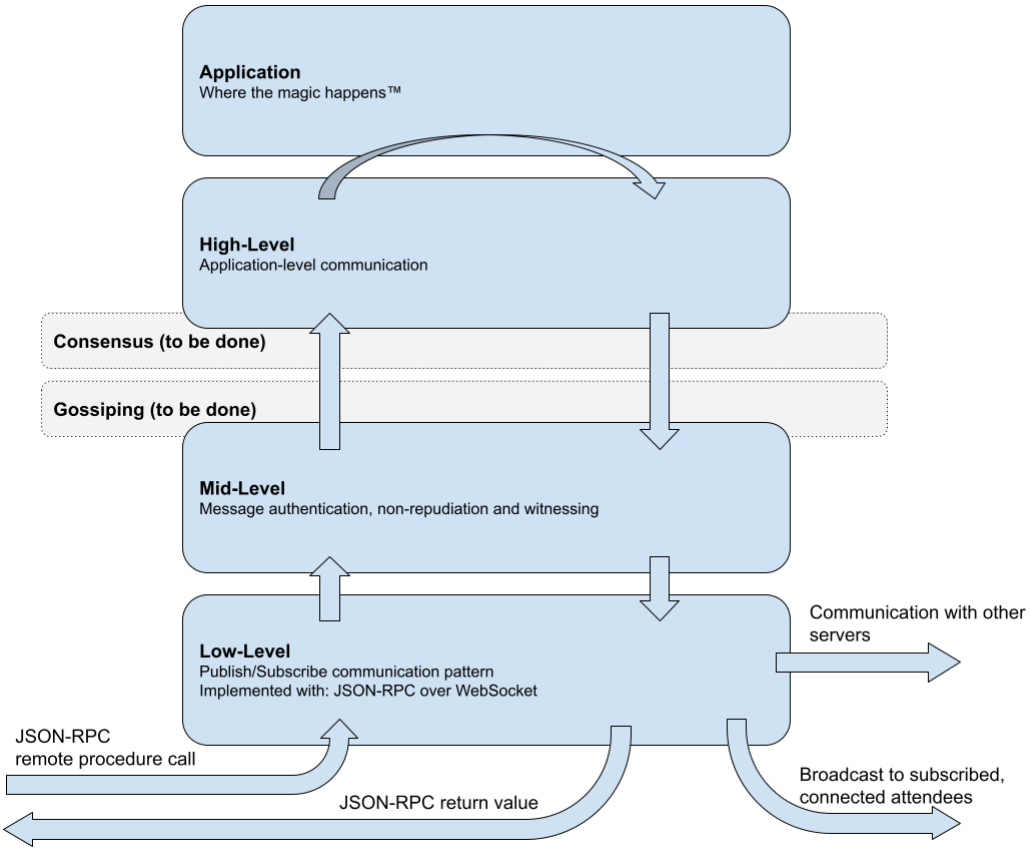

Figure 3.2: How the communication stack works

Figure 3.2 comes from the original protocol design and does not therefore show the advancements in the consensus part.

3.2 System Architecture

As explained above, the project members are divided in four different groups. The advantage of N-version programming is a more reliable software, as the two teams are unlikely to make the same mistakes. An overview of the four implementations is given below.

3.2.1 Back-end 1 - Go

The first back-end of this project is implemented in Go using the version Go 1.16. The instructions to run the server can be found in the README inside the be1-go package on GitHub. It is possible to start an organizer's server as well as a witness' server. The organizer's server is the main operational "hub". Moreover, for operational transparency and robustness, the attendees and witness clients also attempt to maintain connections to each of the witness'

servers. These secondary connections serve both network fault-tolerance and anti-censorship purposes.

Structure of the code

The role of the back-end is to process all messages that it receives and send back an answer to the sender that indicates if the message was processed successfully. The back-end is strict and will reject any message that is not correct. It could be that the message does not follow the protocol or that it is not properly signed or not signed by the expected entity and so on. In any case of a wrong message, an error describing what went wrong will be sent to the client. Here is a global overview of the system.

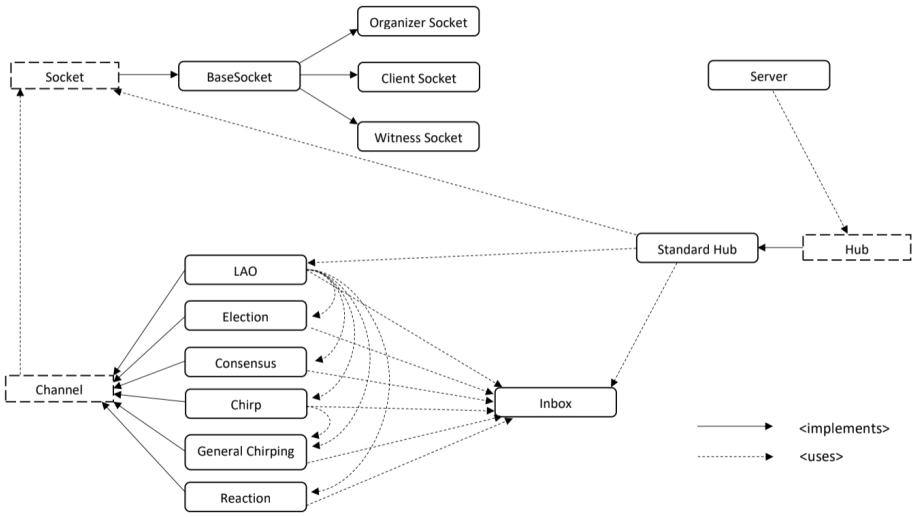

Figure 3.3: Structure of the Go back-end system.

As you can see on Figure 3.3, the Go back-end can be divided into the following main components: the Socket, the Standard Hub, the Channel and the Inbox.

The Socket is the component that communicates with the clients. It receives and forwards all incoming messages to the Standard Hub. It also sends back the answers to the clients after the messages have been processed.

The Standard Hub is the main component of the system. Its role is to perform some validations on the incoming messages and direct them to the right Channel. The Channel component also does some validation. Each channel will receive the messages that are directly addressed to it and will perform some more specific checks than the Standard Hub. It will then write in an Inbox all messages that need to be stored (only the publish messages) so it can answer catch-up queries with all the messages that have been stored.

The following diagram illustrates the path taken by the messages when they arrive from the front-end.

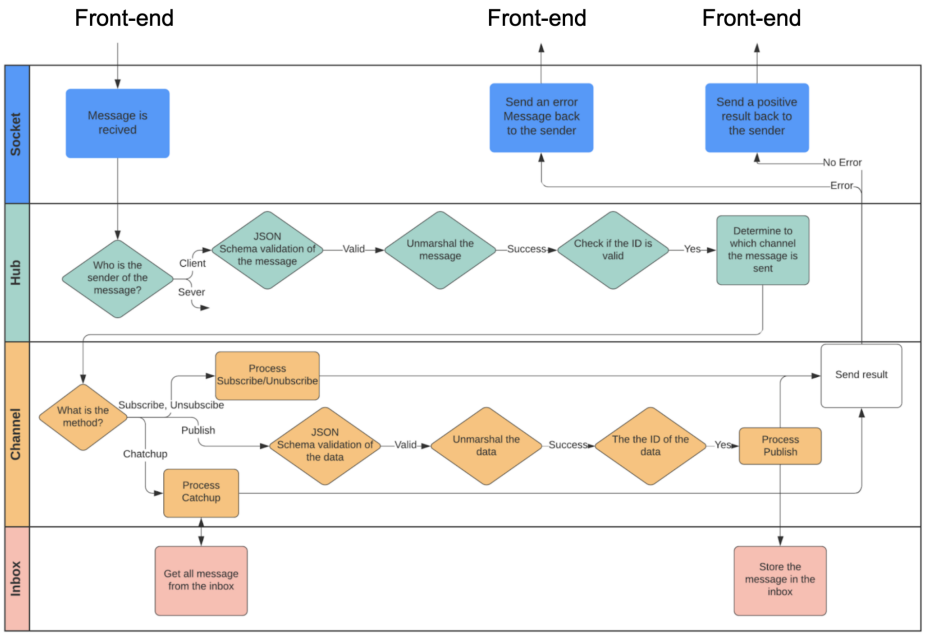

Figure 3.4: Data flow in the Go back-end.
[2]

3.2.2 Back-end 2 - Scala

The second version of the PoP back-end system is implemented in Scala (v2.13.7). The communication protocol between client and server relies on WebSocket over HTTPS, provided mainly by Akka HTTP library [3]. Moreover, the internal architecture of the server follows a data stream pattern among different actors rather than a single instance unit. The stream along with the graph approach also delivers a convenient way of dealing with errors and makes data processing within the actor model manageable and asynchronous. The stream and graph API was also conceived by Akka in their Akka-stream API [4], thus introducing almost no incompatibilities, and matching perfectly our needs given the current requirements.

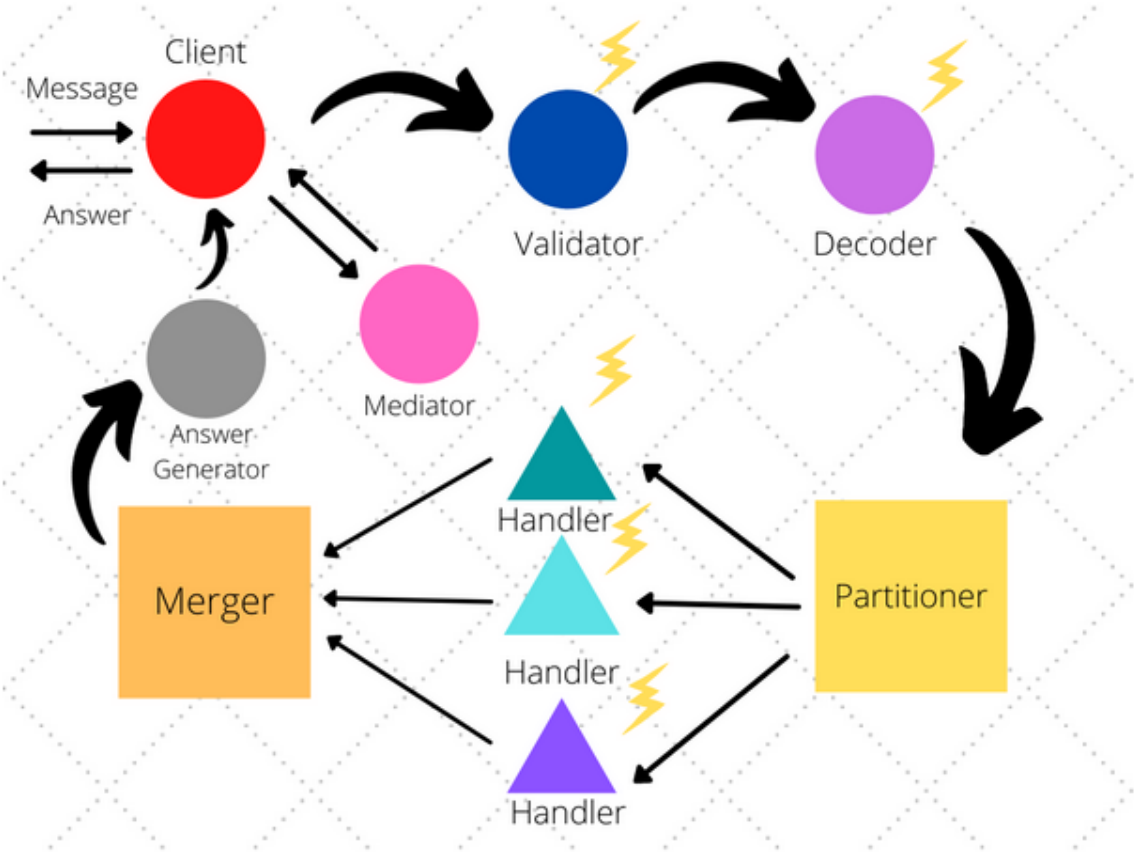

Figure 3.5: Simplified graph of back-end 2

This simplified graph captures the essence of the architecture of the Scala back-end. A more complete graph can be found in the be2-scala directory on GitHub, in the file *docs/README.md*.

Upon receiving a low-level WebSocket message, the Client actor handles it, whether it's a new incoming connection, or a disconnection. It then passes it through the graph to the next node. The Client actor can also deal with formal subscribe/unsubscribe queries by asking a third party *PubSubMediator*, which is an askable actor that keeps track of who is subscribed to which channel and more. It then updates its table accordingly.

The stream of graph messages flows from source (Client actor) to *AnswerGenerator* and back to the client actor again, turning this into a closed coupled source-sink graph. Consequently, the Client actor is also in charge of sending back responses to the front end.

The Client actor models any entity that communicates with the server (i.e organizers, attendees and witnesses).

Every graph message gets processed by every node its flow passes through, making implementation of different phases as independent as possible and hence upgradable for future use cases. The graph nodes can also detect errors while operating with received data and, as result, change the flow of the stream to follow a fault path in order to recover from the error

or report back an error response to the front-end.

For example, if the *Validator* of the JsonRpc requests fails to validate an incoming message that does not correspond to the protocol, it rejects it by notifying the *AnswerGenerator* which generates an error message accordingly and sends it back to the front-end via the Client actor.

Other graph components such as handlers and decoders, were also designed to follow the same approach.

3.2.3 Front-end 1 - TypeScript/React

The web front-end articulates around three major technologies, namely:

- TypeScript for all the logic of the application
- React Native which provides responsive user interface for web and mobile devices
- Redux that allows easy storage by using reducers

The following is strongly inspired by the README of the web front-end on GitHub [5] and introduces the system operation.

Folder organization

The front-end 1 is organized into different modules as follows:

```
/
├─ App.tsx       # the entry point
├─ __mocks__     # test mocks for external libraries
├─ __tests__     # tests
├─ components    # library of reusable UI component (React-based)
├─ ingestion     # module to process incoming messages
├─ jest          # module to allow testing with jest
├─ model         # module containing the data model & objects
│  ├─ network    # ... of the objects sent over the network
│  └─ objects    # ... of the application's entities
├─ navigation    # module dealing with the React navigation
├─ network       # module to do network operations with the back-end
├─ parts         # module containing the application's unique screens
├─ platform      # platform-specific implementation of system functionality
├─ res           # resources
├─ store         # module dealing with application state (Redux-based)
│  ├─ reducers   # module containing the reducers (Redux)
│  └─ stores     # module to access states outside the React environment
└─ styles        # stylesheets
```

At first, this architecture felt quite counter-intuitive. Before working on this project, we were more familiar with feature-centered organization, where the files are sorted by features rather than by usages. For example, to find wallet-specific files, you must browse all modules and search for wallet files instead of going directly in a folder that could be called “Wallet”. We thought about changing it, but since the code was given pretty much untested to us, that would have been a challenge where we could have lost a lot of time. We still think that it would be nice to do it in the future, as this type of organization might be more suitable to work on different features in parallel.
However, this kind of organization has its advantages. Every module is a clear separation of independent actions, and this reflects well the schema in Figure 3.6.

Application state

For the application to react to user’s inputs, it needs to manage its state correctly. Its state is made up of local data, such as user’s cryptographic wallet. More importantly, it contains

the local representation of the whole system and its state, which needs to be consistent. For example, someone being on Android should be able to see the same pages as someone being on the web. In order to achieve this, the user interface displays information from the store module but is not allowed to modify information from it. The view is only updated when a message is received from the back-end and processed.

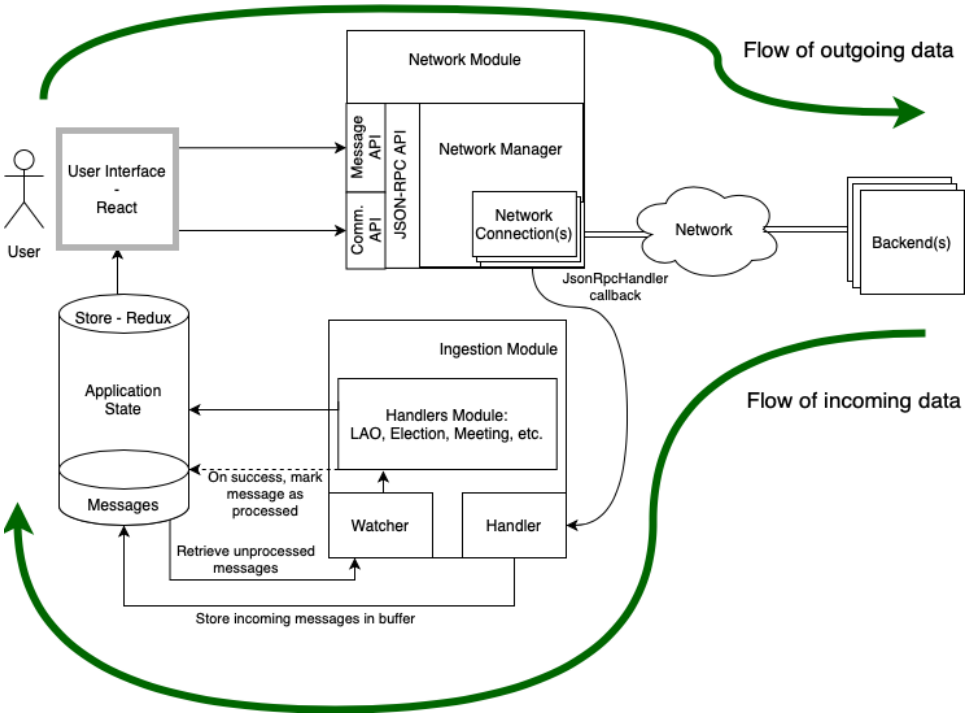

Figure 3.6: Data flow in the web front-end

When the user wants to send a message, let us say publish a post in the social media, they first do the necessary UI operations. Then, the application will send the message to the back-end using the Network Module. The back-end will validate it and propagate it where it is needed in the system. If everything goes well, the message will come back to the user's device and go through the ingestion module. When being ingested, the application will update its state accordingly. By doing so, the store will contain new information that will be automatically displayed on the user's screen.
When it is convenient, the UI could directly modify the application state, but only when this action does not modify the state of the system. We could imagine changing user settings or clearing data in the browser.

Sending messages

As you can see on Figure 3.2, the communication stack of this project is made of multiple layers. The network module contains the logic to encapsulate application-level messages and pass them down the stack. It goes as follows, from the lowest abstraction to the highest:

1. *NetworkManager* and *NetworkConnection* classes abstract away everything related to the WebSocket connections and errors that may appear at that level.
2. *JsonRpcAPI* provides the JSON-RPC API that abstracts the publish/subscribe communication pattern. In it, there are functions to publish a message, subscribe to a channel, and catch up a channel.
3. *MessageApi* allows to generate specific messages to simplify the logic in the application code and reduce duplication.
4. *CommunicationApi* contains functions to deal with communication operations, such as subscribing to a particular channel.

Receiving messages

Sending messages and processing them are completely unrelated in our code, that is why they are separated into independent modules.
On the networking side, the *NetworkManager* and the *NetworkConnection* classes can attach a *JsonRpcHandler* callback to a WebSocket connection. It is called whenever the back-end sends a JSON-RPC request or a notification.
On the message processing side, the *ingestion* module is the one taking care of receiving messages and processing them. It handles the message according to its type, by updating the application state as needed. Finally, it marks them as processed.

User interface

Providing a good user interface, along with a good user experience requires the user interaction pattern to be predictable, coherent, and homogenous across the app. To achieve this, we have a library of reusable components that are all stored in the *components* module. These UI objects are then reused throughout the application and assembled to create the different screens that are in the *parts* modules.
In general, the unique views contained in *parts* oversee all the application-specific logic, because they include the full behavior of that screen. When going down into sub-components, there is mostly the logic that is only dealing with the UI itself.
To link the user interface with the application state, we use React Redux. When the current state of the application is shown on screen, it is done using Hooks. These hooks are linked to the state, and automatically re-render the UI if the state changes.

Coding style

Since TypeScript leaves a lot of freedom to the developer, it takes effort from everyone to maintain a consistent code style. With that in mind, the codebase was developed using a

TypeScript variant of the Airbnb style guide [6]. ESLint [7], a JavaScript/TypeScript linter can run directly in the IDE, allowing it to spot errors and inconsistencies in the code style. Our continuous integration software also executes static analysis using SonarCloud [8], providing feedback against common problems. It tells us if it finds bugs, code smells, duplications and vulnerabilities.

3.2.4 Front-end 2 - Android

The Android front-end uses Java for its codebase.

Project Structure

The project is organized into different modules as follows:

```
app/src
├─ main
│  ├─ AndroidManifest.xml  # file describing essential information about the app
│  ├─ res                  # resources
│  └─ java/P
│     ├─ di                # module containing the hilt modules
│     ├─ model             # module containing the data model & objects
│     │  ├─ network        # ... of the objects sent over the network
│     │  └─ objects        # ... of the application's entities
│     ├─ repository        # module to handle data operations and storage
│     │  ├─ local          # local data source (database)
│     │  └─ remote         # remote data source (connection to back-end)
│     ├─ ui                # module containing the application's screens
│     └─ utility           # module containing the utility classes
├─ androidTest             # tests needing an emulator/android phone
├─ debug                   # module containing debug files
└─ test                    # unit tests
```

The letter *P* represents the project package: *com/github/dedis/popstellar*.

Application design

The application follows the Model-View-ViewModel (MVVM) pattern

- The View encompasses all the activies and their corresponding fragments of the application. As for now, we have an activity for the home page, a LAO detail, the social media and the settings. One activity represents one functionality of the application whereas fragments are an interface contained within an activity.
- The ViewModel manages the data used by the activities and fragments, each activity has its corresponding *ViewModel*.
- The Model encompasses the local and remote data source, model classes and the repository. The last component is useful to simplify data retrieval for the whole application. It is between the ViewModels and both data sources. The *LAOLocalDataSource* corresponds to the database and the *LAORemoteDataSource* corresponds to the connection to the PoP back-end.

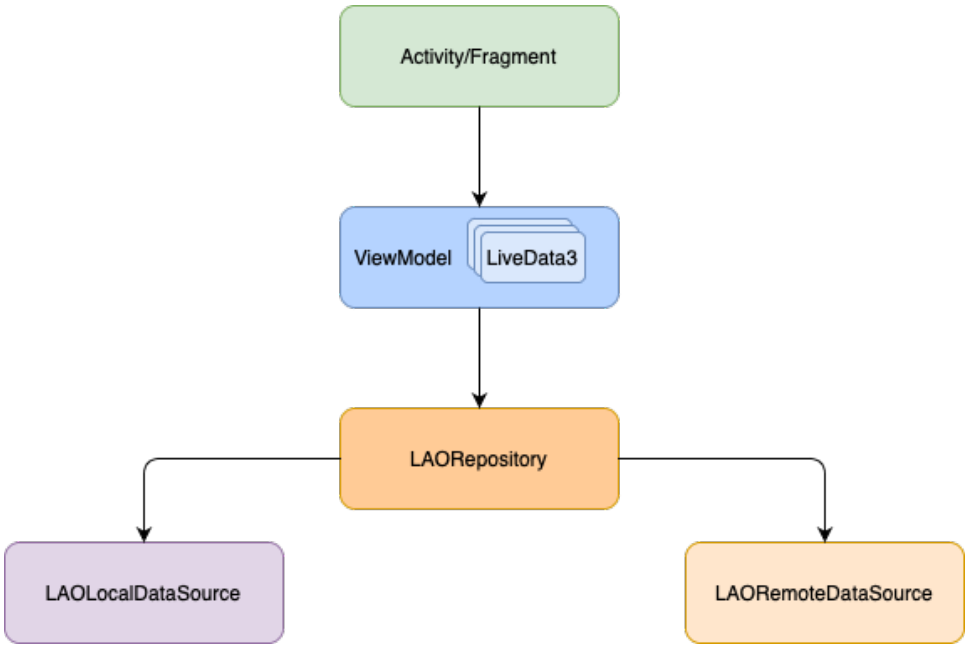

Figure 3.7: Model-View-ViewModel for the project

Application state

The application state of the Android front-end behave the same as in the Web front-end. But instead of a *store* module, we have a *LAORepository*. We also have a list of LAOs instead of a single one.

Sending messages

The communication stack of the PoP project is made of multiple layers. The repository module contains some of the network communication logic, here is how it is organized:

- The *remote* module handles the WebSocket-based connections by using the Scarlet [9] library.
- The *LAORepository* exposes the JSON-RPC API which is used to abstract the publish/subscribe communication. It contains all the functions to publish a message, subscribe to a channel and so on.

The application-level logic to generate messages is directly implemented within a *ViewModel* class, for example the request for sending a chirp is in the *SocialMediaViewModel*.

Getting messages

In the project, receiving messages and processing them are handled differently.
First, the *repository/remote* module observes upcoming messages and notifies the *LAORepository*.
Then, the *LAORepository* forwards them in the handler, which will process the messages using the correct submodule, depending on their types, and update the application state accordinguely.

User interface

Similar to the Web front-end, we aim to provide consistent user interface and experience through the application. In order to do so, we created the package */ui* which contains reusable components that create the different interfaces of the application.
The */ui* package consists of views such as the Home and Settings view. It also provides elements such as a *Date Picker*

Dependency injection

To provide an object with the objects it depends on, we use Dependency Injection. In the project, it is handled by Hilt [10], which is an annotation-based framework.

Coding standards

The project follows the Google Java Style Guide [11]. The google-java-format plugin allows very easy formatting.

The values used for the UI are stored in the corresponding xml files in *res/values*.
The *R* class in Android is an auto-generated class which is used to access any resource. For example, the component of a layout can be accessed using *R.id.component_id* or *R.string* for a string.

3.3 Previous Work

From the birth of the project, many features have been added. Some of them were working as expected, and others needed to be fixed. Because we are going to talk about some of them later during the report, here are their descriptions.

3.3.1 Roll call

The roll call is by far the most important concept of our system. This event can be created by the organizer of a LAO and is the necessary operation to provide digital identity to each participant. By setting a location, a time and a date, each attendee will be able to come physically to the event. Then, the organizer must scan each participant's QR code shown on their screen, either on the Web or the Android client. When everyone has been scanned, the organizer can close the roll call. At that point, everyone will receive a PoP token which proves their existence in the Proof-of-Personhood system.

3.3.2 Election

As an organizer of a LAO, you can create an election. This implementation of e-voting uses the PoP token as a way to ensure anonymity and the fact that each person can vote only once.
To set up an election, the organizer can create multiple questions. Each one of them can contain multiple ballot options. Then, the time and date need to be set. When open, all attendees can cast their vote. They can vote any number of times, but only their last vote will count. It is to make sure that everyone can vote on their own without being under the pressure of anyone. When the election ends, all attendees can see the results. Witnesses would be able to guarantee that the votes are legit, by comparing stored hashes when the election is ended.

3.3.3 Digital Wallet

With roll calls, the Proof-of-Personhood system provides a method to prove that all the people that were present exist. At the end of them, all participants receive what we call a PoP token, which is a cryptographic key pair that contains a private and a public key. This token can be used in the system to prove someone's identity and rights. For example, casting an election vote or publishing a post in our social media relies on signing messages with it.
Users are expected to join multiple LAOs and participate in many roll calls. Thus, we needed a way to store PoP tokens while ensuring usability, security and privacy. That is what the digital wallet is for. Initialized by a seed that only the user knows, it is able to generate all tokens of the user by knowing the corresponding LAO and roll call.

3.4 User Experience

This semester was the first time ever where a team of students was dedicated to User Experience. In the past, no one ever thought about it and it can be felt when using the application. This team of two students (Céline and Kilian) chose to work on it in parallel with their respective projects.

In the beginning, they found inconsistencies between the two front-ends. Everyone was informed of these issues at the general meeting, and we took note of them. Despite being motivated, they realized that it would be hard to re-design both front-ends while advancing in their projects. They thought about the time they could use for it and decided, with the authorization of the UX supervisor, to focus on the user experience of the new features of this semester. Then, there would be less modifications to make about these later. Social media was their focus, since having an enjoyable experience on it is really important. They read about the 10 Usability Heuristics for User Interface Design [12] and tried to apply them as much as possible in their work this semester.

Chapter 4

Consensus

4.1 Project Purpose

The goal of this project is to add a consensus to the system, happening when an election is launched, to make sure that the majority of witnesses and organizers are ready for it.
We also wanted the implemented consensus to be made compatible with other features of the program in the future with as few modifications as possible.

4.2 Design

The consensus we implemented is based on the Paxos algorithm. In an instance of a Paxos algorithm, there are proposers, who each propose a value, and acceptors, who accept a value. The proposers are also acceptors. The proposers and acceptors of our consensus are the organizers and witnesses of our system.
Fow now, we only implemented a consensus to accept the start of an election, but we could not make it blocking, as we only worked on the Go back-end and the Android front-end. This means that the election can be started on any system without having a consensus take place.
All messages of the consensus are sent on a new channel, where all witnesses and organizers are automatically subscribed. All of them contain a *consensus_instance* value, identifying the object on which the consensus happens, and a *message_id* value, being the ID of the message starting the consensus.
The consensus was implemented in three consecutive phases.

Phase 1 - Elect and Elect Accept

During this phase we did not implement a consensus protocol, and instead only created a way for the users to propose and accept the start of an election.

The messages sent during this phase follow the pipeline shown in Figure 4.1.

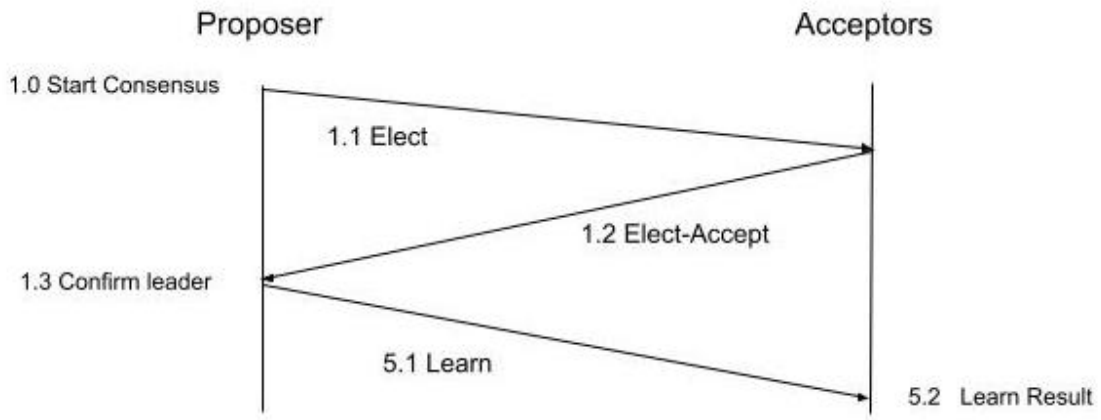

Figure 4.1: Message pipeline of the first phase of consensus

The first message, Elect (Appendix A.1.1), is sent by the proposer's client to inform the acceptors of the start of the consensus and of the value proposed.
The second message, Elect-Accept (Appendix A.1.2), is sent by each of the acceptors' clients to inform the proposer that they accept or refuse the proposed value.
The last message, Learn (Appendix A.1.7), is sent by the proposer's server once the majority of acceptors accept the proposition and their messages are received.

Phase 2 - Paxos

In this phase we implemented the consensus protocol in itself. We chose to implement the Paxos protocol, a well known consensus protocol. All the messages added to the pipeline are generated and sent by the servers, and the pipeline is now the one shown in Figure 4.2. During this pipeline, the servers keep the state of the consensus, that evolves during the process.

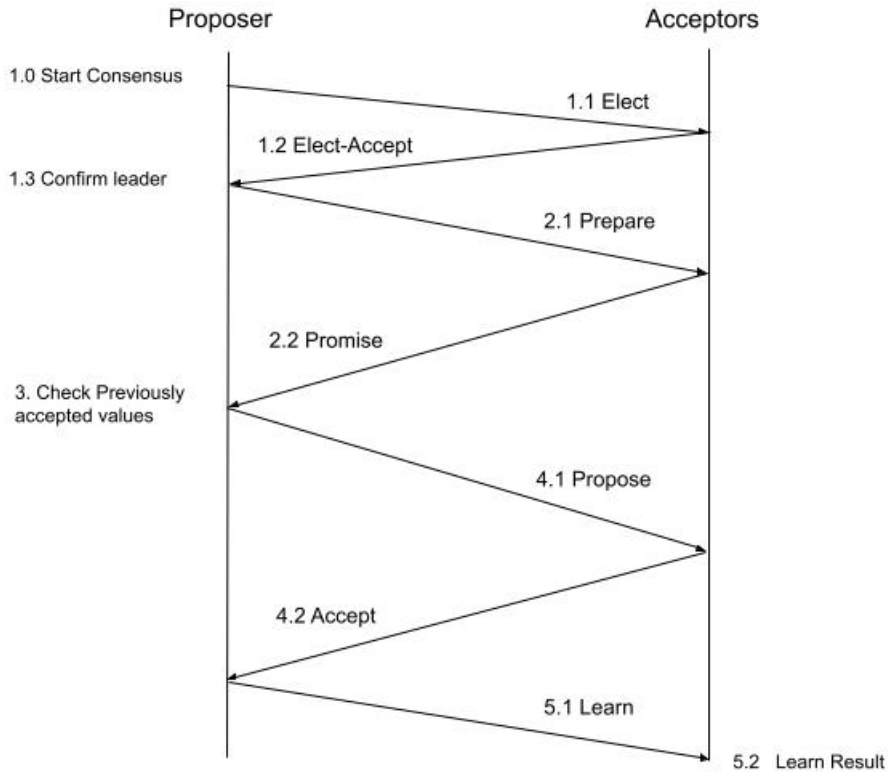

Figure 4.2: Message pipeline of the second phase of consensus

The Prepare (Appendix A.1.3) and Promise (Appendix A.1.4) messages are used to check if a value was previously accepted, and to make sure that only one proposer stays if there are multiple proposers at the same time.
To do this, the proposer will save $proposed\_try = max(promised\_try, proposed\_try) + 1$ and send the new value in the Prepare message.
The acceptors will then check whether their *promised_try* is smaller that the *proposed_try* of the sent message. If it is the case, they will update their *promised_try* to the received value and answer with their *accepted_try* and *accepted_value* in a Promise message.

The Propose (Appendix A.1.5) and Accept (Appendix A.1.6) messages are used to validate the value chosen by the proposer once the previous phase is finished.
To do this, the proposer will check every received *accepted_try*. If one of them is accepted, it will send a Propose message with the highest *accepted_try* and its *accepted_value*. If not, it will send a Propose message with its *proposed_try* and *proposed_value*.
Once the acceptor receives the message, it will check whether the *proposed_try* of the message is greater or equal than the *promised_try* of the server. If it is the case, it changes its *accepted_try* and *accepted_value* to the values of the message and sends an Accept message.

Phase 3 - Timeouts

In this phase, we added a way for the consensus to fail when multiple proposers share the votes, when the proposer fails, or when too many acceptors fail. To do this, we added a timeout in the following situations:

- When an acceptor does not receive a Prepare message after receiving the Elect message. This timeout is longer than the other ones as it needs to wait on a reaction from users and not only from a machine.
- When a proposer does not receive enough Promise messages after sending a Prepare message, it will timeout twice. After the first time, it will continue the consensus as an acceptor, and it will fail the consensus after the second time.
- When an acceptor does not receive a Propose message after sending a Promise message, it will timeout and send a failure.
- When a proposer does not receive enough Accept messages after sending a Propose message, it will timeout twice. After the first time, it will continue the consensus as an acceptor, and it will fail the consensus after the second time.
- When an acceptor does not receive a Learn message after sending an Accept message, it will timeout and send a failure.

When the consensus fails, a Failure message (Appendix A.1.8) is sent to each server and their clients to signal it.

4.3 Implementation

4.3.1 Back-end - Go

Phase 1

For this phase we created a type, *ElectInstance*, saving some values exclusive to the elect instance, consisting of the number of acceptors and the number of positive and negative received Elect-Accept messages.
In addition to broadcasting the message to the subscribed client when receiving Elect-Accept messages, they are stored. Once enough are stored, the proposer will create and send a Learn message.
To send messages, we wrote a goroutine making the server handle the message created by itself to avoid strange logs coming through multiple messages in case of an error.

Phase 2

For this phase we created a second type, *ConsensusInstance*, containing the values used during the consensus. We linked all *ElectInstance* in their respective *ConsensusInstance*, as multiple Elect messages can be sent about the same object.
We modified the created message after enough Elect-Accept are sent to be a Prepare message, and continued the Paxos pipeline. All this work is similar to the one done for the Learn message during phase 1, as the server will each time check wether the conditions are fullfiled to send the next message, and send it if it is the case.

Phase 3

For this phase we created a goroutine launched each time an Elect message is received. This goroutine keeps track of time and creates and sends a failure message in case of a timeout. We also added a channel in ElectInstance, used to inform the timeout goroutine when a message is received.
At the same time, we added a failure once enough negative Elect-Accept messages are received, to avoid having to wait each time for the complete timeout.

Evaluation

We wrote unit test to test the timeouts of the consensus, making sure each time that the correct message is sent in this case. To avoid having tests lasting for multiple minutes while waiting for a timeout, we used an external library [13], which allows us to create a mockable clock instead of using time functions, and if this clock is mocked, it can be freely manipulated to advance time artificially.
We also wrote tests to make sure that when a message is received the correct message is answered, and that the messages are correctly validated when received.

4.3.2 Front-end - Android

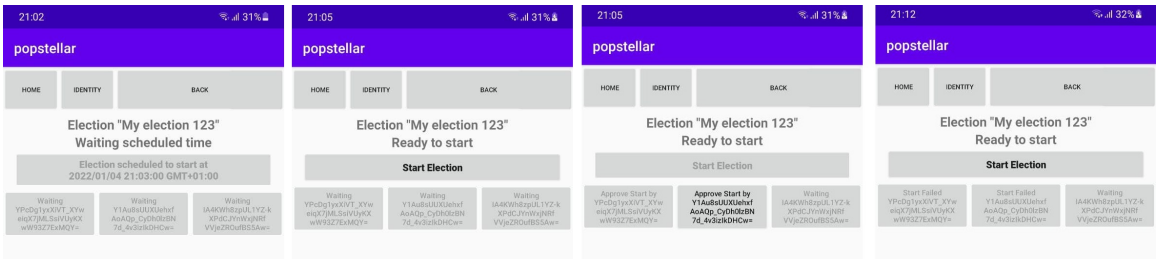

Figure 4.3: Consensus UI for election start

Phase 1

The *ElectionStartFragment* displays information such as election's name, scheduled start time, the current status and all the nodes with their public key. See Figure 4.3 first image.
When the scheduled start time is passed, the UI will be updated, allowing nodes to start. See Figure 4.3 second image.
When a node clicks on "Start Election", it will send an Elect message. All nodes, when receiving this message, will be able to click on the node button, that will send an ElectAccept message. See Figure 4.3 third image.

Phase 2

Phase 2 is only handled in the back-end.

Phase 3

For this phase, all the timeouts are sent by the back-end since clients can disconnect during the process. When a Failure message is received, it will update the state of the corresponding *ElectInstance* and update the UI, allowing the client to try starting the consensus again. For example, node 1 (current user) and node 2 both failed in the last image of Figure 4.3.

4.4 Future Work

- As this semester's work was only done on the Go back-end and the Java front-end, it will need to be done on both the Scala back-end and the Web front-end to have a consensus working with any combination of front-ends and back-ends.
  This will also allow the consensus to be blocking, instead of being easily ignored.
- The consensus protocol can be added to other areas of the system. For example, when changing the name of a LAO or closing an election, but it would need a few modifications to work this way:
  First, the registered value in the state of the consensus needs to be changed from a boolean to a string, as the value can be of any type.
  Then on the Go back-end, the value of the consensus state will need to be defined by the received message and not always be true.
  Then on the front-end, more generic UI needs to be created, for example one with a simple accept/reject value for any basic choice or one allowing node to enter any string value that would be sent in the Elect message.

Chapter 5

Social Media

5.1 Project Purpose

Social interaction is one of the basic human needs. Within a community living together, this is achieved with speech and talks between people and groups of people. But once we move online, Sybil attacks become increasingly impactful, especially in anonymous groups.

Nowadays, more and more businesses use social media and influencers to increase their brand awareness. They hire them to post about their brand and offer money in exchange that varies in function of the audience and popularity of the influencer. This popularity is often measured in terms of followers, likes, and comments. With this system, it is possible to trick companies into thinking that you have a lot of audience but in reality, this audience is full of bots that randomly interact with your post.

This can also cause the spread of misinformation. For example, according to some 2020 research by Carnegie Mellon University, about 45% of tweets concerning COVID-19 on Twitter could have been sent by bots [14]. Being able to get undeserved influence from nowhere (with tens of thousands of fake followers and likes for example) is very problematic.

Up to now, the main way to counter this has cost the users some of their privacy, whether it is by having to give out their phone number, contact details, or even their real identity, with some form of online verification. Since our project is about anonymous Proof-of-Personhood groups, it seemed straightforward that some anonymous means of communication were necessary, but also with protection against Sybil attacks.

The PoP tokens seemed to be the perfect solution for this. Thanks to this system, each roll call participant would only have one anonymous identity inside the local autonomous organization. Using their PoP token inside the system would allow them to post chirps, delete them, follow other users and react to other people’s posts. This enables our social media to be much more meaningful, as every chirp and every reaction will come from a single real person while allowing everyone to keep their anonymity.

5.2 Design

Following the publish/subscribe model, each user's feed is a channel where the owner is the only one authorized to publish and remove chirps. Although a chirp can be removed from the UI, by the design of the pub/sub communication protocol, it will always exist in the historical records of the user's channel. All other users have the right to subscribe to that channel and catch up on past chirps. The name chirp, a sound that birds make, was agreed upon by the social media team after realizing that "tweet", figuring on the original social media UX layout, was a copyrighted term.

Publish a Chirp

Only users with an active PoP token, from the latest roll call, may publish chirps. To enable clients to get real-time knowledge about ongoing chirps on other user feeds, identifying information about all chirps is posted in another channel: */root/<lao_id>/social/chirps*. Each *Chirp* data object consists of the following:

- Text: Max 300 characters (i.e., Unicode code points). The UI needs to enforce this rule. The organizer + witness servers also need to enforce this restriction, rejecting the publication of a chirp that is too long.
- Parent ID: The parent chirp's message ID if it is not the top-level chirp. This is not yet used, however, it will be very useful for the future reply in thread functionality.
- Timestamp: The UNIX timestamp in UTC of the time that the chirp is published. Two chirps cannot be posted in the same second. The organizer and witness servers enforce that the timestamp field is valid (by verifying the timestamp against the server's time including a threshold).

The complete JSON to add a chirp can be seen in Appendix A.2.1.

Possible errors are:

- The maximum amount of characters is reached.
- The parent ID does not exist.
- The timestamp is out of bounds.

After validating the chirp, the organizer's server propagates the above message on the channel it is meant for, but it also creates the following message and sends it to the universal */root/<lao_id>/social/chirps* channel.

The complete JSON to notify the addition of a chirp can be seen in Appendix A.2.2.

At first, we thought that this *notify_add* message would be the one we would be using to display chirps on the user feed. As you can see, there are only three pieces of information about the chirp being published: its ID, the channel where it has been sent, and the time it was posted. We figured out that it was not the message to enable chirps' display. It was only there for information purposes. The message we are using to show chirps is thus the one above (add chirp), which is sent to the social channel of the sender.
In the future, we can imagine that these messages could be used to know what are the most trending channels in a LAO and to show a feed to someone that has not followed anyone yet.

Remove a chirp

As with the adding functionality, only users with an active PoP token from the latest roll call may remove chirps. What is more, only the owner of the chirp can remove that chirp.
To enable clients to get real-time knowledge about ongoing removals on other user feeds, identifying information about all chirp removals is posted in the channel: */root/<lao_id>/social/chirps*.

The complete JSON to delete a chirp can be seen in Appendix A.2.3.

Possible errors:

- Unknown chirp
- Timestamp out of bounds

After validating the chirp removal, the organizer's server propagates the above message on the channel it is meant for, but it also creates the following message and sends it to the universal */root/<lao_id>/social/chirps* channel.

The complete JSON to notify the deletion of a chirp can be seen in Appendix A.2.4.

Add Reactions

Anyone with an active PoP token for a LAO may react to chirps. For simplicity and readability reasons, we established that there would be only three types of reactions: thumbs up, thumbs down, and hearts. All reactions are posted in a single channel: */root/<lao_id>/social/reactions* to enable users to have a general overview and see which chirps are being popular.

Each reaction contains the following:

- Reaction codepoint: An Emoji: https://unicode.org/emoji/charts/full-emoji-list.html
- Chirp ID: The ID of the chirp that the sender is reacting to.

- Timestamp: UNIX timestamp in UTC of the time that the user reacted.

The complete JSON to add a reaction can be seen in Appendix A.2.5.

When discussing reactions, we had to know how we would represent emojis in our system. There were two possibilities: either we represented them using Unicode encodings, like "`\uD83D\uDC4D`" for the thumbs up, or directly use the emoji.
The first possibility has some downsides. Having to escape the characters in the JSON files included in the communication protocol is not practical and may lead to errors. Moreover, the serialization of the emoji's representation needs to be done correctly, so that doing a round-trip would be possible without losing information. This kind of string is less readable too, meaning that it would be harder to debug when looking at the messages sent on the network. This method, however, ensures that all systems would represent reactions the same way, providing coherence and consistency.
Using emojis directly seemed easier and clearer. But before making this decision, we had to make sure that each system was able to serialize emojis correctly. This solution may not provide the same consistency as the first one, but having different Unicode representations within systems was no longer a problem considering that they all would be displayed as emojis anyway. After doing some research, we saw that systems could serialize them, so we agreed on using this process.

A second discussion happened when we thought about the channels to which reactions are sent. For all of us, it did not feel optimal to post all reactions in a general channel. It means that all the users would receive all reaction messages of their LAO, even of channels they are not subscribed to. Hence, we made some propositions and debated about the one we were going to use. These were:

1. **Keeping the old protocol**

   Because we did not have that much time left, keeping the old protocol was the easiest solution. It would mean having all reactions posted on a general reaction channel. Front-ends have to store reactions by chirp IDs, even for chirps they are not directly storing (because the user is not following the channel where it was published). This way, the number of reactions would be up to date at all times, and following a new user would directly show the correct number of reactions for each chirp. On top of that, it is possible to know what are chirps people react the most to, which allows us to show the most trending ones to all users.
   The only downside is that storing all of them is not optimal and does not scale well with the number of users in a LAO.

2. **Having one reaction channel per user**

   When following someone, you would automatically subscribe to the corresponding user's social channel. With this method, you would only receive reactions from the channels you truly care about and would not store all reactions of the system.
   There were three variants of this: the general reactions channel could be erased, it could

be replaced by a current message sent daily by the back-end, or it could be kept. If deleted, we would not be able to make statistics about the chirps that trigger the most reactions. If kept, it means that every reaction would be sent twice, which is not optimal either. Having a message sent every twenty-four (or less) hours that maps each chirp to its actual number of reactions may seem like the best solution, but it would need a lot of implementation on the back-end side and also take away some control from the front-ends (with respect to decentralization).

3. **Having one reaction channel per chirp**

   Each chirp would have a channel dedicated to its corresponding reactions. Again, it would mean that you only get the reactions from the channel you are subscribed to, disabling the possibility to know the most trending chirps. This would imply a lot of channels to create and subscribe to. Knowing exactly when to subscribe is unclear. We could do it automatically when following someone, or allow users to choose explicitly the ones they want to keep up to date.
   This mechanism might bring more value to the general channel since you would be able to subscribe to reactions of chirps from your own feed.

In the end, we put optimization aside, thinking that it was not our top priority at that moment, and stuck with the old protocol. It is the only one that is easy to implement, while ensuring that reactions would be displayed the same way for all users no matter when they subscribed to the channel.

Delete Reactions

Even though this was not implemented during the semester due to a lack of time, we had a protocol drawn up for reaction deletions. Only the sender of the reaction is allowed to delete it.
Each deletion request contains the following:

- Message ID: The ID of the request that the sender wants to delete.
- Timestamp: UNIX timestamp in UTC of the time that the user sent the request.

The complete JSON to delete a reaction can be seen in Appendix A.2.6.

5.2.1 User Experience

Nowadays, social media applications are made so that the user experience is as pleasant as possible. It has multiple benefits. For example, an inviting application is an application that the user will open often and for a long period of time, allowing for more advertising possibilities. Economic purposes aside, the PoP social media needs to be pleasing. The user interface must

be instinctive, and react correctly to users' inputs while providing clear feedback of what is happening if needed.

When our supervisors presented this project, they gave us a UX layout for the application. It was designed to be a mix of Twitter and Reddit, as you can see in Figure 5.1.

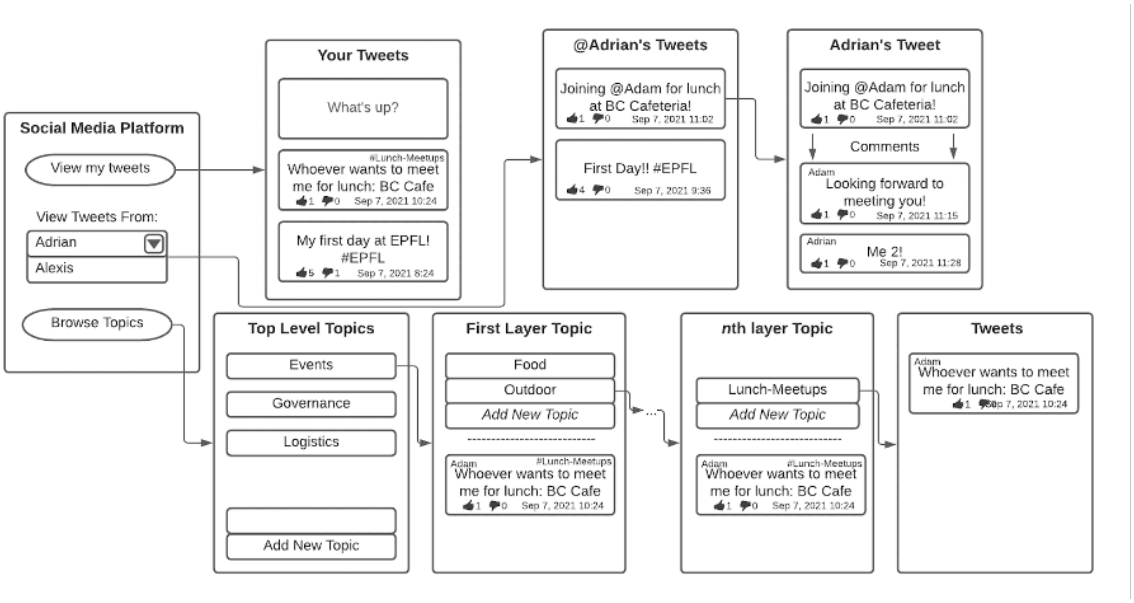

Figure 5.1: First UX layout for the social media

When we saw it, we immediately thought that it was not really appropriate for this type of social media. Since we have an instantaneous communication application where everything goes fast, these topics organized in layers felt awkward to us. With this behavior, someone would have to browse multiple topics to access something precise. This process might be more suitable for forum-type of applications, where questions and answers need to be carefully sorted. Moreover, it would probably need some moderation if users were able to create topics wherever they want. Otherwise, topics may appear multiple times in multiple layers, and it would be hard to find what you search for.
We decided to manage topics using hashtags. Having hashtags seemed better because of the simplicity of the design. Users can just look up a topic in the search bar, and every message that contains this hashtag would be displayed. In the future, we can imagine the possibility of directly following topics to stay up to date.

The home screen on the left needed some modifications too. In most applications, it is the screen where users spend most of their time on. It needs to display important information and allow easy navigation. The "View Tweets from" scrolling menu is not scalable and does not adapt to anonymization. Every user will only be identified with their public key, which is not suitable for an alphabetically sorted list. What we preferred was to create profile pages (like "@Adrian's Tweets") that can be accessed by searching for a user in a search menu, or by clicking on a user's profile picture.

A navigation bar displayed at all times at the top or the bottom of the application was an obvious addition to us. It is common in modern applications and provides easy navigation in the user interface.
With all these changes in mind, we designed a new UX layout for the application.

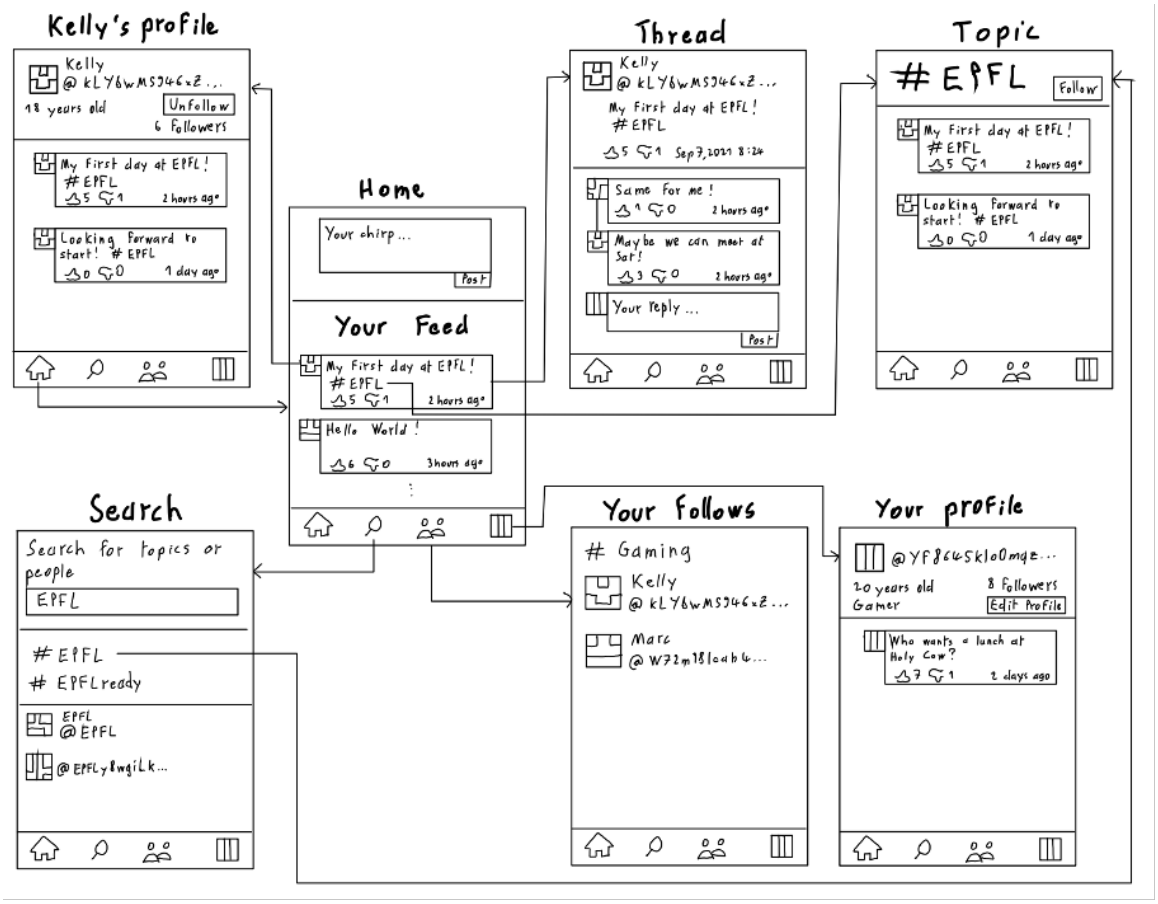

Figure 5.2: Second UX layout for the social media

Not all these pages have been implemented and are ready to use, but this can be a model for future students working on this social media to understand our goal in terms of UX and functionalities. The home screen becomes the place where you can see the chirps of your feed (i.e.: of the people you follow), publish chirps and navigate through the application. The menu bar at the bottom allows you to go to “Home”, “Search”, “Your follows” and “Your profile” easily and at all times. On the Web front-end, people are identified by an image generated with their public key [15] so that it is visually easier to differentiate multiple users. The page “Your follows” allows users to see every person and topic they are following, for easy and quick access to them.
We believe that this UX is nice and suitable for this social media application, and would encourage next students to keep going in this direction.

5.3 Implementation

5.3.1 Back-end 1 - Go

The main logic to receive messages was already implemented by the previous students working on the project. However, the messages related to social media were not yet recognized by the system. To implement those new features, three new channels have been created. The first one of those channels is the channel to which the clients send their new chirps in the first place. It is also to this channel that they will send a message when they want to delete a chirp. There is one channel of this type per user that owns a PoP token. Those channels are created at the end of each roll call for each attendee of the roll call. The second of those channels is a universal chirp channel. It is the role of the server to feed this channel by broadcasting all the chirps that are posted on the individual channels. The server makes sure to remove the content of the chirps, and only send the metadata of each chirp to the universal channel, so the messages are very light. There is only one channel of this type per LAO and it is created during the creation of the LAO. All clients can subscribe to it in order to have an overview of the activity of the other attendees. The third of those channels is the reaction channel. Every reaction and deletion of a reaction is sent to this channel. There is only one channel and its creation happens when a new LAO is created. Those channels are respectively named the *chirp.Channel*, the *generalChirping.Channel* and the *reaction.Channel* in the Go implementation.

Message Data

A new Go struct has been created for each new message data added to the protocol (*chirp_add_publish.json*, *chirp_notify_add.json*, *chirp_delete_publish.json*, *chirp_notify_delete.json*, *reaction_add.json*, and *reaction_delete.json*). A *verify()* function has also been added for each new Go struct. This function verifies the data of the messages without any information about the current state of the system. It will for instance check that a timestamp is positive, that a field is encoded in base64URL if it needs to, or that an ID (which is a Hash of some other fields) is correctly computed.

Verification

Many checks are performed upon receiving any new message. Most of those checks happen in the channels, so that the checks can be more precise because the type of the message is known. The back-end always checks that the new chirps are sent to the right channel and signed by a valid PoP token. It also checks that the message has not already been sent or that the chirp/reaction to be deleted has been sent in the past. Each field of the messages is being checked, so that they correspond to the protocol. The implementation is as strict as possible in order to refuse all incorrect messages.

5.3.2 Back-end 2 - Scala

Since the main components handling each message received by the server were already implemented during the last semesters, there were not many major additions to the back-end concerning the social media. However, there were many modifications done to all the intermediary handlers and verifiers in order to be able to process the new social media-related requests.

New events and *MessageData*

For the needs of the social media feature, 5 new events were implemented: 3 client-side and 2 server-side ones. The client-side additions are adding a chirp, deleting a chirp and adding a reaction. What is new in the social media subproject compared to other semesters is the need for server-side broadcasts. Since the server needs to broadcast the chirp's ID to the general *posts* channel in response to each chirp, we also need a broadcast event for each corresponding client-side event.
To represent these, we created new objects (*AddChirp.scala*, *NotifyAddChirp.scala*, *DeleteChirp.scala*, *NotifyDeleteChirp.scala* and *AddReaction.scala*) extending *MessageData* inside the new *. . . /message/data/socialMedia* folder. To have matching *MessageData*, we needed to create new *ObjectTypes*, *CHIRP* and *REACTION*, inside *ObjectType.scala*, and new *ActionTypes* for each action (*ADD*, *NOTIFY_ADD*, *REMOVE* and *NOTIFY_REMOVE*), inside *ActionType.scala*.
We also had to create new requests (all named *JsonRpcRequest<NameOfObject>*) extending *JsonRpcRequest* inside the new *. . . /requests/socialMedia folder*. The handling of all these new events is done in the modified components described below, in a similar way (excluding the new broadcast functionality) as the previous events.

Modified components

To be able to handle the new requests inside the back-end, new cases to handle and validate were added to several components, all inside the *ParamsWithMessageHandler* structure when referring to the server graph in *docs/README.md* on GitHub, since the rest of the request is already well handled by the server. New cases were added to *MessageDecoder.scala* to get typecasted *JsonRpcRequests* out of the generic *JsonRpcRequests*.
We also added new cases inside *DataBuilder.scala* to be then used by the *parseMessageData* function of *MessageDecoder*. We also added new components, *SocialMediaValidator.scala* and *SocialMediaHandler.scala*, in order to validate and handle *JsonRpcRequests*, in the same fashion as for other event types. To be able to use these components, we modified respectively *Validator.scala*, to add a case for social media requests and *ParamsWithMessageHandler.scala*, by adding a port and a case for social media requests handling.
We also added new *JsonFormats* inside *MessageDataProtocol.scala*, to allow conversions between the new social media objects and JSON. In order to implement the attendee channel

functionality, we had to modify *RollCallHandler.scala*. It now reads the *LaoData* object inside the *handleCloseRollCall* function to get the current attendee list and then uses the *CreateChannel* function from *DbActor* while iterating over the list to create a new channel for each attendee.

Channel types

Even though the channel type verification is not fully implemented yet, it has been made possible, focusing especially on social media. In order to be able to check that a chirp is sent to the right type of channel (on one's own channel, not on the main LAO channel for example), the *ChannelData* object of each channel stored in the database holds an ObjectType value, allowing to check easily whether a request can be sent to a channel. This will be done inside *SocialMediaVerifier.scala*.

Verification

Currently, the social media validator only checks the staleness of the timestamp of the sent request, whether the sender is in the attendee list (i.e. holds a valid PoP token) and whether they are sending to their own channel, enforced with the channel name consisting of their PoP token.

5.3.3 Front-end 1 - TypeScript/React

The three major goals we focused on on the web front-end are publishing chirps, removing chirps, and posting reactions. To do so, we had to implement the user interfaces and their corresponding logic.

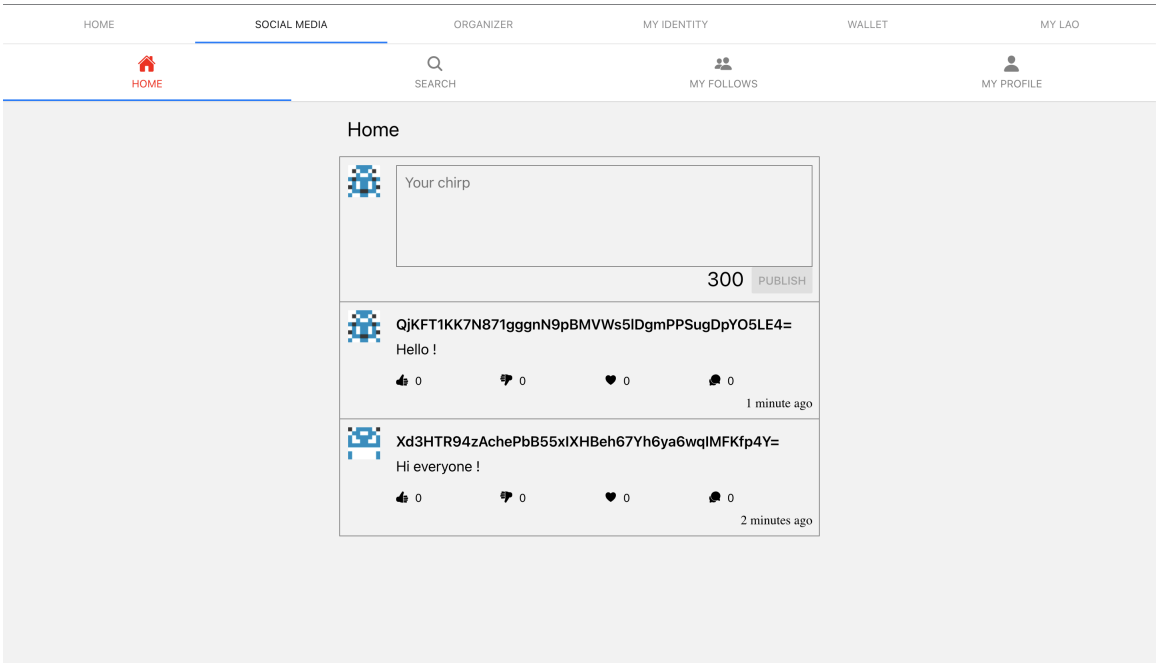

Figure 5.3: Social media Home screen

Publish a chirp

Users can publish chirps on the social media home page. The social media section will only be visible after creating a LAO and the button for publishing a chirp is clickable only after having participated in a roll call; these ensure that only attendees with a valid pop token can publish chirps. Since we limited the number of characters of a chirp to 300, the publish button becomes disabled if we reach this threshold.

Figure 5.4: Publishing less than 300 chars

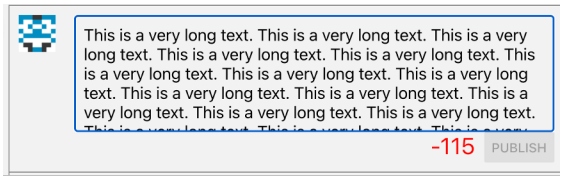

Figure 5.5: Publishing more than 300 chars

The data add chirp request sent to the back-end follows the same logic as in the previous implementation for other messages: we created the data according to the protocol and added all the necessary functions to validate and send the query.
Every attendee follows themselves automatically at the end of a roll call, so they can see all their published chirps directly on the home page. However, if someone wants to receive chirps from others, they have to follow them manually. On the search page, users just have to click on the follow button for each person they want to follow in the list of attendees of the last roll call.

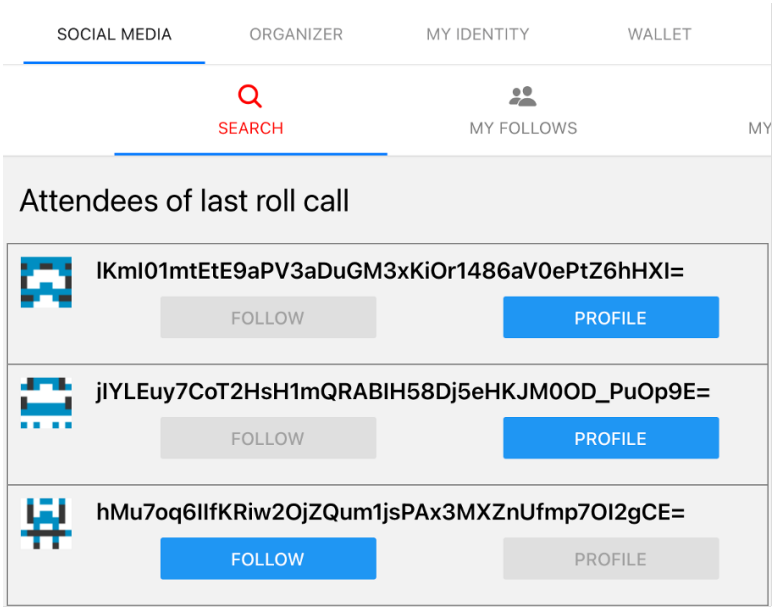

Figure 5.6: List of attendees in the Search page

After receiving a response from the back-end meaning the requested add chirp is valid, the ingestion processes the message using Redux for storage. We create the corresponding chirp object in *ChirpHandler*, then dispatch the add action to *SocialReducer*.
We decided to let our data structure be composed of three components: *allIdsInOrder*, a list containing all chirp IDs in order of sending time, *byId* maps a chirp ID to its *ChirpState*, and *byUser* maps a sender to the list of chirp ID they sent. *byId* and *byUser* are both *Record<>* that allow us to retrieve random elements in O(1). The *byUser map* is useful when we want to search for a particular user, especially when displaying someone's profile. The list *allIdsInOrder* enables us to show the chirps in the right order. We perform a binary search algorithm when inserting an element, for it to achieve better time efficiency. In the end, we make a list of the state of all chirps and display them using a React component called *ChirpCard*.

Figure 5.7: A chirp you can delete

Remove a chirp

We pass the current user's public key to each *ChirpCard* component, so it can verify if the user is the sender of the chirp. Only the sender can see the delete button. Once the sender clicks on the button, a request for deleting the chirp will be generated and sent to the back-end.

There is a flag in the *Chirp* object called *isDeleted* which is going to indicate if the chirp is deleted or not. Initially, this value is set to false. It turns to true once we receive a message from the back-end telling us to delete the chirp. We set it in the *SocialHandler*, we also replace the text with an empty string. Later, when we delete a chirp in *SocialReducer*, we do not make it disappear completely from the storage. We simply update the state of the chirp in the *byId* map. In this way, a deleted chirp will still have records in our database and its replies will be kept.
The *isDeleted* flag helps us to differentiate a published chirp and a deleted chirp when displaying a *ChirpCard*. A deleted chirp will only have a string in grey saying “This chirp was deleted” and the reply field. It is no longer possible to see the text and the reactions for this chirp.

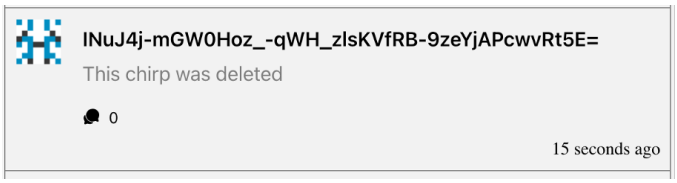

Figure 5.8: A deleted chirp

Broadcast

We also implemented and tested the necessary process for transforming the broadcast of an addition and a deletion of chirp into message objects to use in our system. When the back-end sends those messages, they are directly converted into these objects. We have not implemented their ingestion logic yet but they will mainly be used for knowing the trending channels in the future.

Post a reaction

Since all the reactions go on a general reaction channel (one per LAO), we made everyone subscribe to the reaction channel automatically after the roll call for simplicity. We have buttons for thumbs up, thumbs down, and heart in our *ChirpCard* component. Each button will send an *addReaction* request with the corresponding reaction codepoint to the back-end.

Similar to *addChirp*, *ReactionHandler* will create a reaction object after receiving the response from the back-end, then it will call the *SocialReducer* to add the reaction to the storage. We created another entry called *reactionsByChirp* in the *SocialReducer*, which maps a chirp ID to the pair of the reaction codepoint and the list of users’ public keys. In this way, if we see that a user already added a reaction to a chirp, we will not count it multiple times, to achieve a meaningful reaction within the scope of a roll call.
We retrieve the counter for each reaction with the help of a mapping from each chirp ID to the pair of the reaction codepoint and the length of the list of users’ public keys. We recover the

entry for each *ChirpCard* using the ID of its corresponding chirp.
We added some additional checks when a chirp is displayed without having reactions to it, because we do not store chirps that do not have reactions in *reactionsByChirp*. If not handled carefully, this would lead to some undefined behaviors. The number of each reaction is displayed beside the corresponding reaction on the *ChirpCard*.

5.3.4 Front-end 2 - Android

In order to use our social media in Android, we needed to create a new activity and its related fragments so it is separated from the rest of the application, similar to when we are in a LAO. The dedicated *ViewModel* contains all variables and methods needed to change screens, display chirps from a specific LAO, send chirps to the back-end, and so on. We have to handle the start of the activity differently whether we were in the *HomeActivity* or in the *LaoDetailActivity*. For both of them, we put an extra ("OPENED_FROM") in the intent to specify which activity we are coming from. We can directly specify the LAO ID and name when we are joining the social media from one of them. It is then set at the start of the activity. If you are not part of any LAO, clicking on the social media button will make a toast appear saying “You are not in a LAO”, making the user aware of the requirement needed to access it.

We should be able to swap LAO from which chirps are displayed. It is done by having a menu in the *ActionBar* in which we create a new item for each LAO joined or launched so far. When you have chosen one, its name is added in the *ActionBar*, making it easier to track which LAO you have set your social media to. Usage of this functionality can be seen in Figure 5.10.

To display the chirps, we have a *ChirpCard* in which we will feed information about the chirps to display. It is then used in a *ListView* to make the social media’s feed. Changing the tab is done with the *BottomNavigationView*, apart from the tab to send a chirp, which is accessible from the home page. When writing a chirp, if you try to send it while you have not chosen a LAO beforehand, it will not do anything and display a toast. Right next to the button, we have the number of characters left, which is limited to 300. It will turn red when you write beyond the limitation and disable the button.

Data and object

To use the social media, we needed new messages data : *add*, *notify_add*, *delete* and *notify_delete* for the chirp. Those are data that are sent within a message. The server receives them and responds to the client. Then, we handle them to make the proper changes. We also created a *Chirp* object in order to store it in the LAO. The object contains the ID of the chirp, which is just the message ID, the channel to which it was sent, the sender, the text, its timestamp, a flag to know whether the chirp was deleted or not and the parent ID. With all this information, we can use it in our *ChirpCard* which is then displayed in our feed.

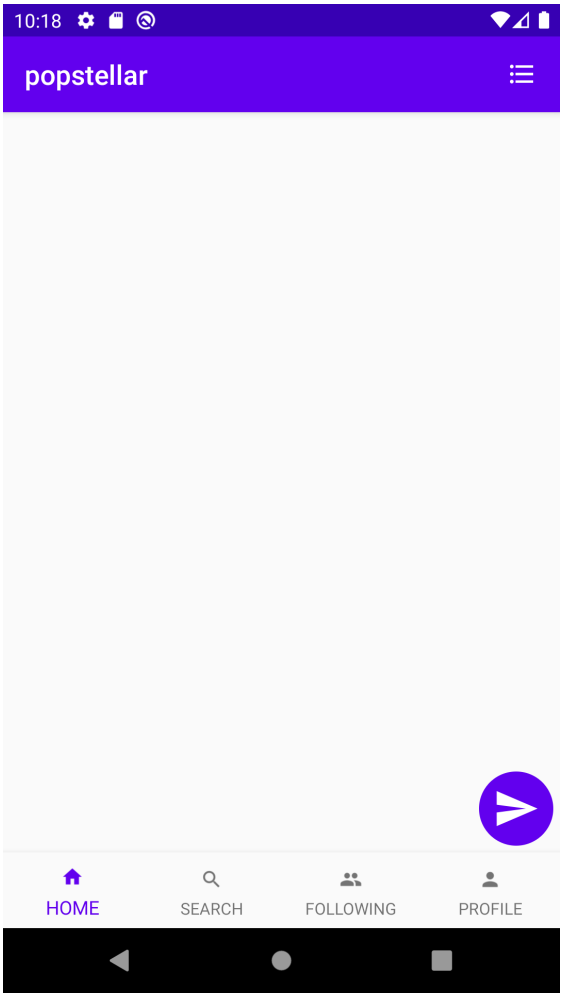

Figure 5.9: Home page at launch

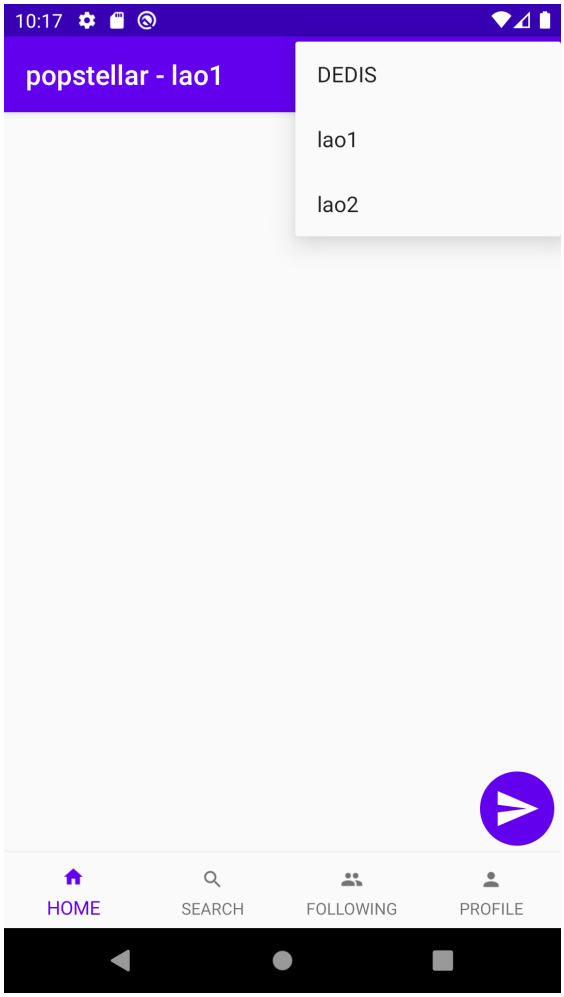

Figure 5.10: List of LAOs

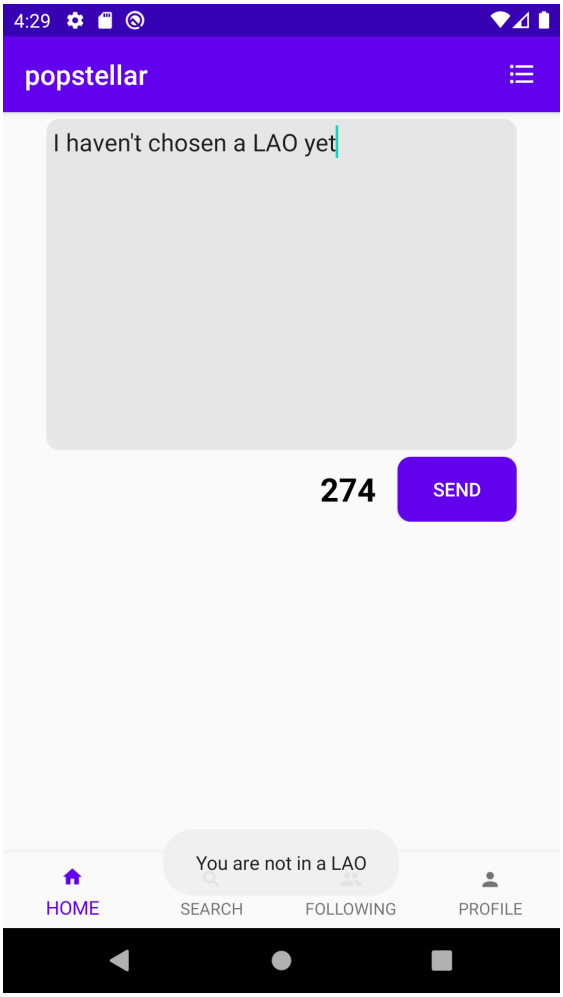

Figure 5.11: No LAO chosen when sending

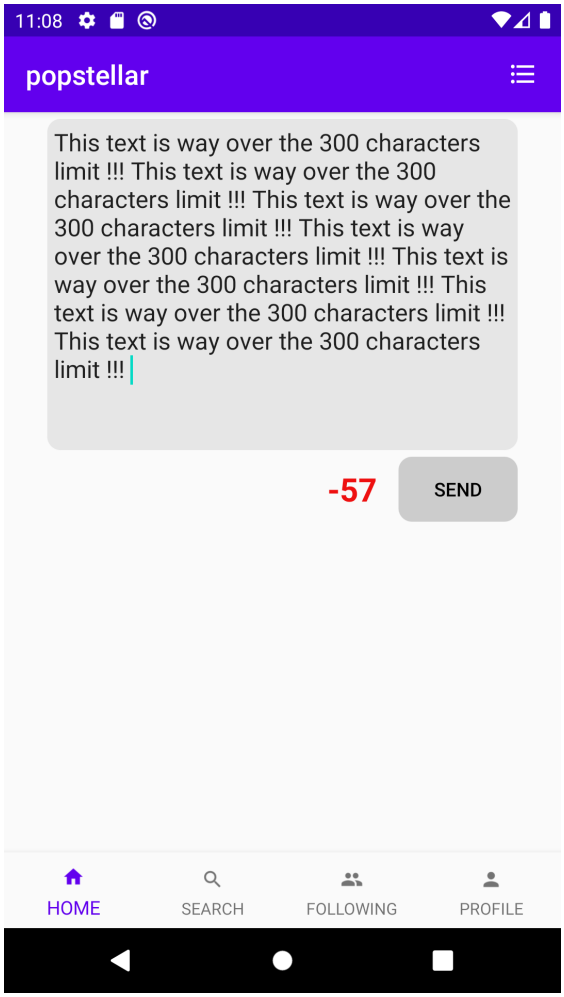

Figure 5.12: Over the 300 characters limit

Publish a chirp

Posting a chirp will send the data *addChirp* to the server. When the message is verified, we receive an answer which will be handled to create the corresponding *Chirp* object. The message is sent in the designated channel specific for each user. You are automatically subscribed to your own channel, meaning you will see your chirps on your home feed. Subscribing to a peer means subscribing to their channel, doing so will send a catch-up message to retrieve all their previous chirps in order to display them in the feed.

Remove a chirp

The owner of a chirp can choose to delete it, only them have the button to do so. Clicking on it will send a message to the back-end. After this, we will handle the response to set the deleted flag to true and the text to an empty string. The chirp will not be completely deleted as it will still be stored and displayed in the feed, but it will only show a text saying it was indeed deleted.

Broadcast

Similar to the Web front-end, we are not doing anything yet when receiving a message with action *notify_add* and *notify_delete*. It will probably be useful in the future to know what are the trending channels and so on.

5.4 Future Work

This semester, we achieved the implementation of a social media platform with the basic functionalities of posting and deleting chirps and reacting to them.

- A first functionality to implement would be the possibility to remove a posted reaction. The way to do this has already been stated in the protocol, but it could not be implemented in all systems due to a lack of time.
- Replies could also be added to the social media. We have already prepared for this with the possibility of storing the parent chirp ID inside each *AddChirp* message.
- Topics are another possible feature to add, to group the different chirps relating to the same subject together and potentially the possibility to tag other users to identify them in a chirp.
- Another functionality could be to have different accounts for different fields of interest, but all linked to one PoP token. It would allow users to separate our hobbies and passions, while still only being able to like or follow users once.

- We could also imagine the possibility of having persistent accounts over subsequent roll calls, by linking the previous PoP token account to the new one, instead of losing access to everything linked to our previous token. Obviously, this would be by choice, in order to allow users to have a fresh start after a roll call if they feel like it.

Chapter 6

Production-Ready

6.1 Project Purpose

The purpose of this project is to turn the Proof-of-Personhood system into a deployable working unit, and provide a suitable user experience. The PoP project was under development by students for almost three semesters now, but no stable version of it was released yet. The system was always run locally on the engineers/production side but was never delivered to run correctly in the field. Shifting from development to production readiness is a major step that requires more thinking from an engineering perspective. And this is what this subproject is all about. Adding testing and deployment phases, respectively, therefore completing the software development life cycle of the PoP project.

Even if, theoretically, the code structure holds, it does not imply in any sense that its equivalent implementation will also be matching the project expectations. Production ensures that bugs and misbehavior that were not discovered in development are detected and reported back to the software development phase, hence, adding another layer of testing towards a good quality and bug-free system. Furthermore, readiness also involves great user experience, putting user interfaces under tests and continuous upgrade to serve user expectations is mandatory. It also makes sure that server applications and user interfaces are coherent as they evolve to respect future project requirements.

A major key of production readiness is debugging, the easier the system is to debug, the closer it is getting to be deployable. Debugging the system is not a simple task as the whole difficulty, certainly, resides in diagnosing exposed bugs when the system is put under work by actual PoP users. Capturing faulty behaviors and tracing bugs is a crucial step to fix them. Many software technologies and frameworks such as logging and testing are needed to ensure such establishment. And this subproject aims to make use of such technologies as much as it is required to achieve its ultimate goal of high quality software.

Figure 6.1: Usual development cycle

6.2 Initial state of the System

At the start of the semester, the team had to choose where they would put their effort into. To do that in the most objective way, a complete overview of the project had to be done.

Here is an overview of the state of the system. In the appendix, you will find a detailed explanation of the notation B.1.1 and a justification of the attributions. B.1.4

	BE1 - Go	BE2 - Scala	FE1 - Web	FE2 - Android
Measures				
Lines of code	1802	1365	2472	5812
Test Coverage	22.5%	20.5%	24.9%	27.3%
Technical Debt	2h	5h	27h	25h
Code smells	43	104	134	267
Features				
LAO Creation	Operational	Operational	Operational	Operational
Roll call	Operational	Not Working	Not Working	Operational
Re-open a Roll call	Operational	Operational	Unimplemented	Operational
Election	Incomplete	Unimplemented	Not Working	Not Working
Message Validation	Unimplemented	Operational	Operational	Unimplemented
Digital Wallet			Not Working	Operational
Cross-server Comm	Unimplemented	Unimplemented		
Consensus	Unimplemented	Unimplemented	Unimplemented	Unimplemented
Chirp Creation	Unimplemented	Unimplemented	Unimplemented	Unimplemented
Chirp Deletion	Unimplemented	Unimplemented	Unimplemented	Unimplemented
Add Reaction	Unimplemented	Unimplemented	Unimplemented	Unimplemented
Remove Reaction	Unimplemented	Unimplemented	Unimplemented	Unimplemented

Table 6.1: Overview of the initial state of the System.

Test Coverage data is not measured

This is an important issue as one cannot know if a part of the system is under test or not. The data seen in the graph 6.1 is the one computed after the coverage was added.

The number of tested modules is very small in all subsystems
The coverage of the project does not go above 25%. This is not enough to provide any sort of reliability.

The protocol requirements are not precise enough and most systems implement it differently
One of the most recurrent issues when testing the system as a whole with all subprojects combined is that they are not perfectly compatible. Either the protocol is not followed or it is too vague and two distinct implementations came out of it.

The systems are not robust and very easy to break
The systems mostly fill the simple behaviors, but they are easy to break, and the errors are not always correctly raised to the user or logged.

6.3 History and Decisions

6.3.1 Many problems, little oversight

The main issue that we faced was getting a good oversight of the project across multiple subsystems. It was obvious that each system lacked stability and suffered from code smells and technical debt, mainly due to the complexity of the code in some files. As it was evident that we would not be able to achieve a 100% reliable software due to the limited time, we had to invest, objectively, our efforts in the areas that needed urgent recovery in order to construct a solid codebase that is ready for deployment.
Nonetheless, we strongly believed that, as a team, we would attain better software quality by the end of the semester.

To achieve this, we had to find what issues with the current state of the system were the most urgent to fix. So our first task was to get a deep understanding of the codebase, even if this task was carried on continuously throughout our whole journey, we needed to get a great grasp at first of different subsystems.

As we were performing manual testing of the system, many bugs were triggered here and there. We then realized that the project required more automated testing and bug traceability, mainly via logging. Moreover, important blocking issues were caused by incompatibilities among the different subprojects, preventing the system from functioning properly.

6.3.2 3 weeks in

After taking the time to apprehend the system architecture and spot weaknesses in the subsystems, we decided to invest time on setting up an automated code analysis tool that would continuously generate reports about the emerging codebase.
SonarCloud [8] analysis made things even clearer: we need to cover more code, hence our goal for the next few weeks was to increase the code coverage of all the systems. And therefore, numerous refactorings across all systems needed to be done by the whole PoP team to make the code testable and pay back a significant part of the system technical debt. Race to code coverage had begun.

6.3.3 Half semester in: Age of testing

Interestingly, the code coverage started to increase gradually in all systems. As a consequence, the production-ready team decided to integrate the coverage in the GitHub actions to monitor code coverage in the repository, setting the threshold to a minimum of 50% of code coverage for newly added code to the repository, hence taking a step forward towards a CI/CD system. This initiative incited all team members to test their freshly implemented functionalities in the project, therefore detecting bugs at an early stage in the development.

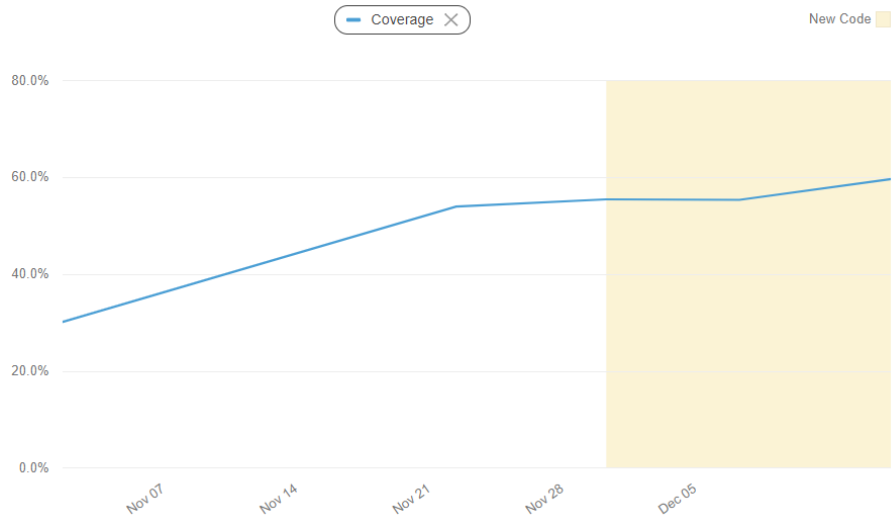

Figure 6.2: Go Back-end - Coverage progression over the semester

6.3.4 Final decisions

As the code coverage rate for each individual system was gradually increasing and functionalities started to become more stable due to unit testing and extensive debugging, the team was ready to take testing to the next level.

Our new concurrent objective was to set up a tool to enable functional testing, meaning we were looking forward to testing individual features within each subsystem. Even if each entity within a certain subsystem is working as a unit, it should nonetheless be put under integration testing and functionality testing afterwards.
Our plan was straightforward: first set up the tool, then run the back-end/front-end server, execute the tests and finally generate and interpret the test report to improve the quality of each system.

Thanks to Mr. Borsò's lectures on testing, we managed to agree on Karate testing framework [16] as a tool that will help us cover most of our testing needs. Consequently, for the rest of the semester, the plan was to keep unit testing going and also set up Karate to implement feature test scenarios. The system tests involved both back-ends and front-ends.

6.3.5 Karate Pros

Karate is an open-source general-purpose test-automation framework. It gets updated frequently and has support for WebSockets connections, web browser automation and application automation through a full Appium support [17]. Moreover,

- The tool can be used for both back-end and front-end. This makes the test suites extremely maintainable, as only one tool is needed for all systems.
- Tests are written in Gherkin, and therefore can even be understood by anyone who does not have a deep understanding of the programming language.
- The framework is easy to extend and customize, and it is in Java, a language learned by all IC EPFL students in their first year.
- New students will not have to learn a new language to understand and extend the test suite.
- By using clever abstractions, the same tests can be run on both versions of the front-end/back-end. This is the main reason why we chose Karate. It divides the work by half in the sense of "Write once, test anywhere", making sure that both systems have consistent behaviors.
- Karate generates test reports automatically at the end of each test session.
  Therefore, by sorting tests by features, the generated report can quickly give an overview of what feature is working and what is not. This is one of the main issue that we faced when starting the semester: getting a complete oversight of the system.

Karate and SonarCloud are complementary.
On one hand, SonarCloud provides static analysis of the codebase to give an overview of the functionalities of the system and on the other hand, Karate provides functional testing to keep the codebase healthy and bug-free.

6.3.6 Implementation

Karate is based on Cucumber / Gherkin. Therefore, the tests are in fact *Scenarios* grouped inside feature files. The idea is that each feature has multiple scenarios, and those scenarios together provide a thorough status of the reliability of that feature.

One important fact to note is that Karate can create calls to other *Features* or *Scenarios*. This is greatly used in our subproject to modularize procedures like starting a server before a test or launching the application in an emulator.

Back-ends

To test the back-ends, the servers are abstracted away, and the tests only see the common interface in the form of the Server class that provides every needed functionalities like starting the server, stopping it, and so on.

Common procedures are modularized and called at the start of each test (in the Background part), hence making the tests as concise as possible, capturing essential statements only.

Front-ends

The front-end idea is similar. Abstract away differences and make tests as easy to understand as possible.
Karate uses selectors to find the elements in the views and perform assertions, clicks, scrolls, etc.

To achieve this, the project uses enhanced page objects. It contains the selectors, but also some procedures that are different from both systems. As an example, both systems do not set the URL of the organizer's server the same way. So a procedure is defined inside the page object to set that URL.

Framework Extensions

Karate offers a mock WebSocket implementation, but it only supports one receiving message. We thus created two utility classes : *MultiMessageWebSocketClient* et *MultiMessageWebsocketServer*.

They can send and receive messages but most importantly, they offer access to a *MessageBuffer* where all the received messages are stored in-order.

Those messages can be retrieved from the buffer and optionally removed as well. But as the order of the messages is not always guaranteed, the buffer also offers retrieving based on filters. That way, one can retrieve the first message that is a *CreateLao* for example.

6.4 Current state of the project

The table is again just an overview of the system. The appendix contains the details about the notation B.1.1 and the justification of the attributions B.1.4.

	BE1 - Go	BE2 - Scala	FE1 - Web	FE2 - Android
Measures				
Lines of code	3291 (+1489)	1460 (+95)	2726 (+254)	6268 (+456)
Test Coverage	69.9% (+47.4)	31% (+11)	32.6% (+7)	43% (+16)
Technical Debt	0h (-2)	5h	25h (-2)	31h (+6)
Code smells	2 (-41)	107 (+3)	132 (-2)	300 (+33)
Features				
LAO Creation	Operational	Operational	Operational	Operational
Roll call	Operational	Operational	Operational	Operational
Re-open a Roll call	Operational	Operational	Unimplemented	Operational
Election	Operational	Unimplemented	Operational	Operational
Message Validation	Operational	Operational	Operational	Incomplete
Digital Wallet			Operational	Operational
Cross-server Comm	Operational	Unimplemented		
Consensus	Operational	Unimplemented	Unimplemented	Operational
Chirp Creation	Operational	Operational	Operational	Operational
Chirp Deletion	Operational	Operational	Operational	Operational
Add Reaction	Operational	Operational	Operational	Unimplemented
Remove Reaction	No Data	Unimplemented	Unimplemented	Unimplemented

Table 6.2: Overview of the current state of the System.

The system's features are now mostly operational, which is a great step forward. Even if manual testing proved the system usable, it does not mean it is reliable.

A lot of work has been done to consolidate the project's code and make it more robust, maintainable and reliable. But there is still a lot of room for improvement. A section is dedicated to this part: Code Consolidation.

Even though the project is way more tested, this can still be greatly improved, especially regarding edge cases: most tests are only focused on basic functionalities and simple scenarios. Special cases are frequently not tested, and this is a big issue

This is expected as our first goal was to produce a usable application and then focus on the edge cases. But the lack of time forced us to only focus on the first part of the procedure.

Even though the project is going in the right direction, it still needs a lot of work before it can be described as maintainable, reliable and robust.

6.5 Future Work

It is clear to the sight of the project's state that the production-ready team cannot retire. A lot of work is still to be done. Here is a non-exhaustive, yet thoughtful, list of tasks that should be targeted:

- Add more Karate scenarios especially for election, and the social media features.
- The assertions on data parts of the messages are not complete. A utility class providing functions to convert data from their base64 form must be added.
- An easy way to validate signatures and hashes should also be created.
- Add Karate to the continuous integration pipeline in the CI.
- Generate test reports automatically and make them accessible from the CI.
- Seek to elevate code coverage for all subsystems .
- Integrate Gatling [18] in Karate to simulate multiple connections (organizers, attendees and witnesses) and hence testing the distributed aspect of the whole system.
- Improve the maintainability of the systems.
- Provide a better user experience. Especially when an error occurs, try as much as possible to recover and notify the user that something went wrong.

Chapter 7

Server Communication

7.1 Project Purpose

In our PoP application, the goal is to have multiple servers, being either witness or organizer servers, to make sure that the organizer cannot refuse access to other people.
To do this, each server needs to be able to communicate to the others to make sure that either they are all in the exact same state or that they share the necessary messages, and can catch up on all channels when necessary.

7.2 Design

This part of the project was not meant to be worked on this semester, but became necessary for the consensus part of the project as it happens between servers and not between clients. The best way to move forward with the communication in the time we had at our disposition was the simplest one: broadcast each message to every other server and keep the same state on every server.
Only the Go back-end was being worked on for this part of the project, therefore there are system specific information only about this one.

7.2.1 Back-end - Go

There was already a way for servers to open a connection between each other, but they did nothing with the received messages. Therefore the steps needed for this connection to work were the following:

- A server needs to be able to process messages from another server and to act accordingly. The message that may be sent by a server to another server are those following the

Answer, Broadcast, Publish and Catch-up schemas.

- A server needs to keep the state of the system and share it when a connection with a new server is being established.
  It is done in two steps. First, every channel sent with the root of the server as destination (for now only the messages creating a LAO) are stored on the server, and having each server send a Catch-up request on the root channel of the other server when a connection between two servers is created. Then having a server request a Catch-up each time it creates a LAO following a message received by another server, be it as part of a Catch-up answer or a Broadcast.

- A server needs to broadcast every Publish message it receives to the other servers the first time it is received.
  We did it by keeping on each server the list of all server sockets of this server. By having it, whenever it receives a Publish message, it sends a Broadcast message containing the received message to all other servers.

7.3 Implementation

7.3.1 Back-end - Go

Message Processing

The basis of the message processing is the same whether the message comes from a client or another server, meaning that most functions relating to the message processing can be reused. There is no difference between a message coming from a client or a message coming from a server, except that a client should not send a message following the Broadcast or the Answer schemas.
We therefore wrote a function *handleMessageFromServer* mirroring the already existing *handleMessageFromClient* but accepting Answer and Broadcast messages. In addition, we wrote the necessary functions to handle both Answer and Broadcast messages.

Catch up to another server

To catch up to a server, we gave each hub an inbox containing all messages received on the root channel of the server. When a new connection is established between two servers, each server will send a Catch-up request on the root of the other server, which is answered using the content of the new inbox.
To have a server catch up on the content of a channel, we implemented the same mechanism, but sending the Catch-up query when a channel is created to the originator of the creation, if it is a server, to the newly created channel instead of the root channel.
When receiving an answer, if the answer contains the same ID as an unanswered Catch-up

query sent by the server, the server will handle all messages contained in the answer to the query.

Broadcast message to each server

To broadcast a message to every server, we added a list of sockets to the hub, where we add a socket everytime a connection with a new server is established. We created a function called at the beginning of the handling of a Publish message, which sends a broadcast message containing the message.
To avoid having a server broadcasting a message when it receives multiple Publish messages having each the same ID, we added an inbox to the hub storing all received message, and send an error when trying to broadcast an already received message instead of broadcasting and handling it a second time.

7.4 Evaluation

Practical Tests

We did practical tests of our implementation by having up to three servers connected to each other, with clients on each server, and checked that the clients could participate together in our application. We also verified that each message was only sent once per server, and that there were no communication loops.

Unit Tests

We wrote two unit tests.
The first one, *Test_Handle_Server_Catchup*, checks that when a catch-up is required by a server, the correct list of message will be sent back to the server.
The second one, *Test_Handle_Answer*, verifies that answer and result messages are correctly handled by the server.

7.5 Future Work

- As the work we did this semester is only on the Go back-end, it still need to be done on the Scala back-end.
- For now, there is a lot of wasted traffic as when a server sends a message by itself (specifically for the consensus), it will be broadcast by every other server. The system could therefore be optimized by selecting which message to broadcast, and to which server, instead of broadcasting everything to everyone.

Chapter 8

Code Consolidation

Throughout the semester, a lot of code consolidation were done. Here is a descriptive overview of the achieved work.

8.1 Back-end 1 - Go

The codebase we were given had a good structure with a consistent style and was well documented and we believed that no major refactoring was needed. However, some clean-up and code consolidation were still needed, here are the main changes.

- Testing - At the beginning of the semester only a small part of the codebase was covered by unit tests. During the semester, we added many unit tests to reach around 70% of code coverage by the end of it. Those tests helped to consolidate the system and remove some old bugs. In addition to static analysis, a good coverage also increases the reliability of the software.
- Code Smells - We removed as many code smells as we could throughout the semester. The goal was to increase the overall quality of the code. Removing the code smells will also help to diminish the probability of bugs and failures in the future.
- Code duplication - At the beginning of the semester, we had two separated implementation of the witness server and the organizer server, although the differences between the two are small. We decided to merge those implementations and to condition on the type of *Hub* the few differences. At start, we had two implementations of a *Hub*: one for an organizer and one for a witness. We now have only one implementation of a *Hub* with an attribute *HubType*. We also merged the entry point for starting an organizer server or a witness server as they were also really similar.
- Code complexity - When a message is being processed by a server, it goes through multiple cases which can be long, and the error handling can add a lot of code complexity.

To reduce this, we created a registry for the message handling, which replaces the case by a map, and that can be used in every channel.

- Formatting - Go is opinionated about coding style and guidelines. Since this semester, this coding style is now enforced by the Continuous Integration and the codebase can be formatted using the "make check" command.

8.1.1 Future Work

- Code duplication - There is still room to improve the code duplication. The channels, for instance, have some common methods and the code could be optimized to take advantage of those similarities.
- Code complexity - The registry is for now only used in the consensus channel. It should also be used in the other channels.

8.2 Back-end 2 - Scala

The Scala back-end was missing some crucial functionalities at the beginning of the semester, even though the necessary data structures themselves were implemented for the most part. In addition to some refactoring, we added some features already present in back-end 1.

- Json Schema verification - We added a JsonSchema verifier for both JsonRpcRequests and the message content itself. Until now, this was not checked, but it now makes the server more robust against improperly crafted requests. To implement this, we have used the *networknt/json-schema-validator* [19] library, which contains everything we need for the validation process. We had some issues with the location of the protocol's JSON files, since they are located in the main folder, but this was resolved with a *sbt* configuration, which copies the files to an accessible place and allows to get a completely standalone *jar* file ready to be deployed in any JVM setup.
- Elections - We added the *CastVoteElection* object, required by the protocol for the elections, since it was the only remaining message to be added from last semester. This was more of an introductory task, allowing us to see how to create a new message object. Even though the election has still not been fully implemented on Scala, this now gives all the tools to people working on the project in the future to do so, as everything is available. We also added a new data structure, *ElectionData*, allowing us to store the election status (still ongoing or not), the election questions and the election votes, which can be used to implement the handling of the result later on. We also completed the other handlers apart from the one for the result.
- Database refactoring - We modified the entire database structure, from one database per channel to one database for the entire server. This choice was made in order to

manage the channels and the messages inside in an easier way. Since we use *Leveldb*, a key-value database, the channels are now represented by keys in the database, in the form */root/lao_id/channel*. We made this decision since the previous implementation with multiple files made catching up a single message really difficult. If one did not know the ID of a message, it would have taken at least O(n) time, where n is the number of messages inside a channel. This was problematic for checking the message sending rights, for checking who the LAO owner is and would have also been a problem for elections and consensus. The single file implementation was agreed upon during a Scala meeting, since it was the most robust one. The other choices were to either keep the database as it was (and have really complex functions to get a single message) or have the useful messages rewritten in the database with specific channel keys (but this would have caused duplication).

- LAO data storage - We added a *LaoData* object to help us manage the LAO better. It contains the owner public key, the LAO's keypair, the list of PoP tokens of the attendees of the last roll call and the list of witnesses (even though the consensus functionality does not work yet in Scala, it can be useful in the future). This allows us to check whether the sender of a message has a valid PoP token, whether someone is allowed to modify the LAO or start events and makes it way easier to sign the broadcast messages sent by the server. There are also added functions inside the *DbActor* object, which manages the database queries, in order to have easy access to this new object inside each LAO.

- Channel data storage - We also added a *ChannelData* object to help us manage the channels. The *ChannelData* object contains the type of a channel (an *ObjectType*, for example *ELECTION* for an election channel) and the list of messages sent to the channel. This makes the catch-up functionality very easy to implement and allows us to check whether a message is allowed to be sent in the current channel. Like for *LaoData*, there are also added functions in the *DbActor* object. Both new objects help to make the code more robust, as they allow us to enforce the protocol better and help preventing unwanted parties from sending inappropriate messages to the server.

- Roll Call - We also added the roll call functionality. Until now, the roll call related messages were only processed with a simple forward inside the handler, but now, the list of attendees' public keys is stored inside the database, in the *LaoData* object, which now allows our LAO to actually work as it is supposed to and have a valid PoP token list, which is very important for the project and adds security in the back-end.

- PoP Tokens - A very important functionality which was missing until this semester was the actual usage of the PoP tokens on the back-end 2. Until now, not much was checked about the sender, but now, using a function which checks the attendee list from the new *LaoData* object, we can check whether the senders of the messages have a valid PoP token (i.e. whether they have attended the last roll call). This allows the enforcement of the attendee role and makes the server way more robust against external threats. The owner role has also been made concrete with a similar function checking whether a sender's public key is the same as the LAO's owner's (similarly stored in the *LaoData* object).

- Database cache - Another minor functionality we added was the cache inside *DbActor*, for the *LaoData* object for each LAO. This choice was made, since it is the object which is queried the most frequently (at each received request), but the implementation could be extended to other objects too, if necessary. Instead of having to query each time the object from the database, we use a simple mutable map, with the keys being the same as in the database, the ID of each LAO, and the values being the LaoData objects. This can prevent database read errors and therefore make the server more robust.
- Base64 data: additional checks - Additional checks were added to the construction of the *Base64Data* to check whether the given string really is Base64.
- Immutable vs mutable: *Message*, *MessageDataParser* and *JsonRpcRequest* - These classes were assumed immutable as one would typically do in a functional programming language, however, they were not. Necessary modifications were done to turn them into immutable classes.
- Scala build and Jar packaging - We upgraded to the latest *sbt* version and optimized build settings, to run the Scala back-end optimally and resolving library deprecations (as much as possible), as well as optimized build settings for *sbt 1.6.1*[20]. We also added a *sbt* compatible plugin to package the code as a .jar, ready to be deployed in DEDIS servers.
- *DataBuilder* refactoring - We refactored the *DataBuilder* to use newly implemented utility type classes *DataRegistry* and *DataRegistryModule* in order to have Json schema validation during the build.
- Validation for data Json schemas - In the new classes *DataSchemaValidator* and *DataBuilder*, the message datas get validated right before parsing/decoding or rejected otherwise.

8.2.1 Future Work

- Elections - The election result functionality still needs to be implemented in Scala, as this was not done during the semester due to time constraints. The data structures for this are already ready (at least partially, maybe students will need to add some other necessary fields in the *ElectionData* object).
- Code quality - The code's quality can still be improved, especially by reducing code smells, since now with SonarCloud one can get access to very precise metrics.

8.3 Front-end 1 - Web

- Roll call - Having a working roll call was the first needed functionality in the whole FE1 system, as all the social media activities work only after a roll call when every attendee obtains a valid PoP token. We first fixed a bug in the *requestOpenRollCall* to send the request data to the back-end. Then, we redesigned the layout when opening a roll call.

The organizer is redirected to a new page dedicated to handling the roll call process. We used a QrReader to scan the attendees. Every time the organizer scans a participant, a message will pop up on the top center of the page indicating the participant is successfully added to the set of all attendees. A badge on the top of the close button keeps track of the number of attendees and is thus incremented by one each time someone is scanned.

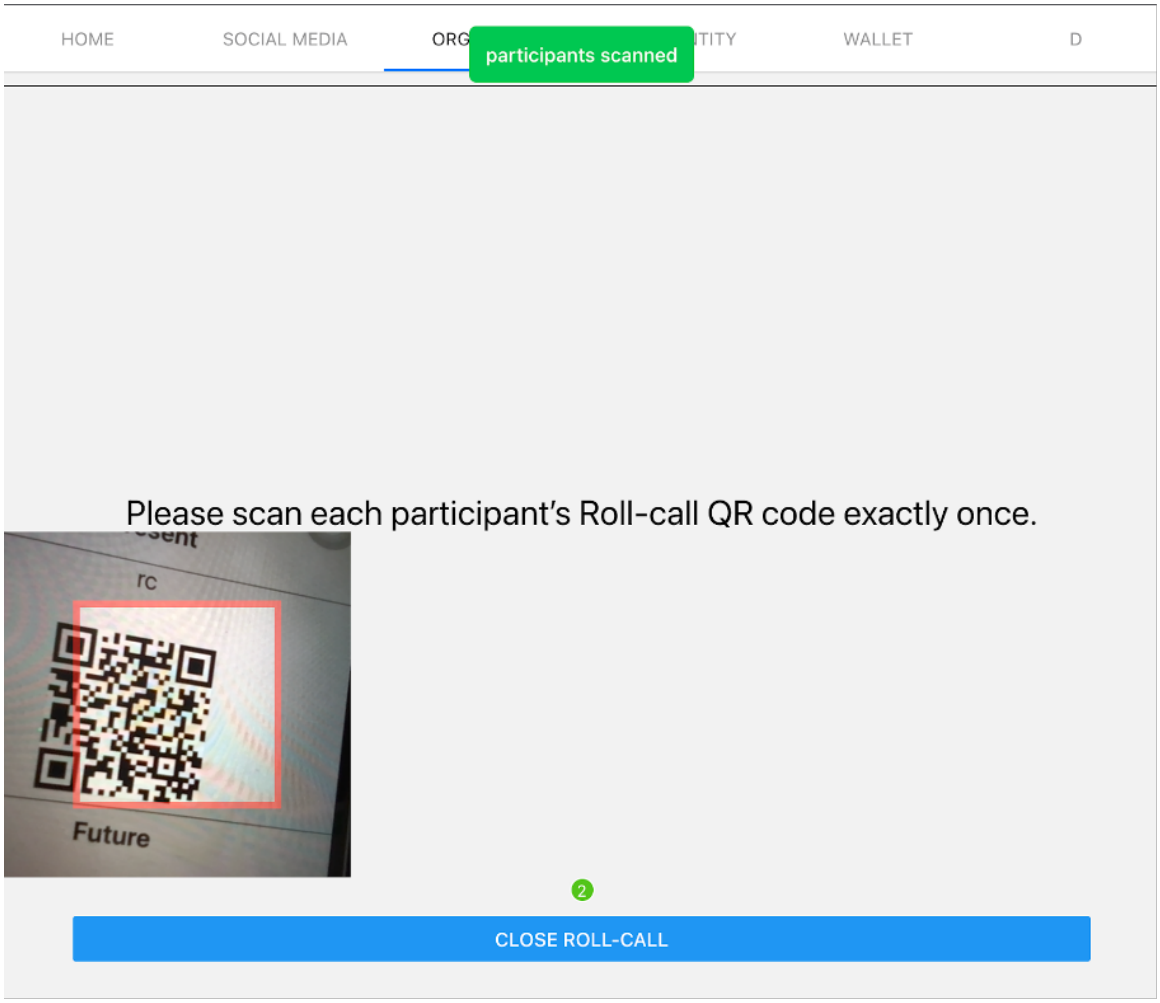

Figure 8.1: UI when opening a roll call as an organizer

To make sure that every participant will only be added once, we use a set for the attendees. The close roll call button will send a request to the back-end, with a table containing all the attendees' public keys and the updated Id of the roll call. Then, the UI navigates back to the organizer navigation tab home. We can see from the status of the roll call changed to closed, and a list of the PoP tokens' public keys of all the attendees.

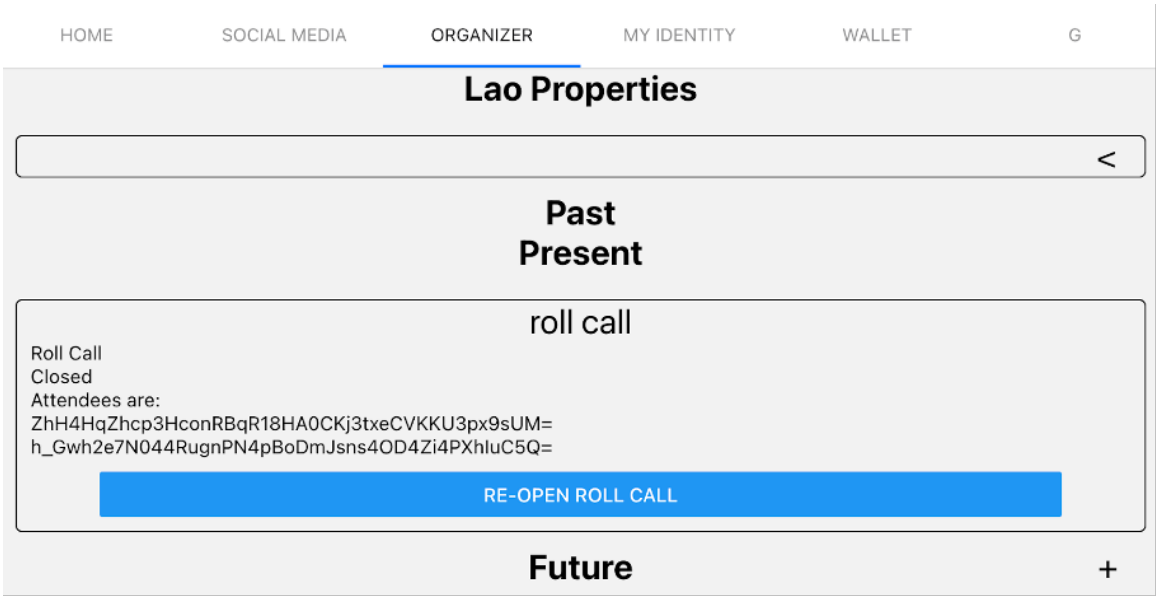

Figure 8.2: UI of a closed roll call.

However, we found that testing was not as intuitive as the implementation. Especially, we needed to not only mock the LAO but also the QR code reader for the camera, the toast message for showing notifications when the scan succeeds, the *requestCloseRollCall* data to send to the back-end, and the navigator when we go back to the home page after closing a roll call.

- Event time - When creating events, their start time could sometimes be set to be before their creation time. This may lead to the end time being before the start time. To handle this, we adapted the times accordingly before sending the creation message to the back-end. If the times are heavily affected, the user will be informed of the changes with a pop up message.
- Organizer's PoP token - Because the organizer needs to be present to administrate a roll call, it was logical that they should get a token at the end of it. To resolve this, we generated their token directly when opening a roll call and put their PoP token's public key automatically in the list of attendees before scanning anyone. Hence, parts of the application that use the PoP token no longer need to differentiate between being an attendee or an organizer.
- Repairing Wallet - When we tried to use the PoP token's private and public key for message signature, we realized that the private key was not generated correctly. Its length was half of what it should be, so the signature was not working. By going through the cryptographic algorithm, we found the problem and fixed it. Right now, the PoP token's key pair is fully usable
- Message signatures - When working on social media, we had to find a way to sign messages using the PoP token's public key. It was not done before us, so its whole implementation was missing. We added the corresponding logic and created a map

that contains all properties of the messages, sorted by their type. This way, adding new properties and retrieving them will be easy and fast when wanting to deal with a certain type of message. The election cast vote message is now signed with the user's PoP token to provide anonymity, as it is stated in the protocol.

- UI & UX - Some user interface elements did not allow for a good user experience, so we made some changes for them to become easier to use.
  - DatePicker - The DatePicker for choosing the date and the time of an event was not working as we expected. It was possible to select dates and times from the past, and the start and end time were not set by default when creating an event. Now, past dates are displayed in grey and there are modular default durations for each type of event.
  - Copying text - When using the application for the first time, we did not realize that the seed needed to be copied to initialize the wallet. Then, a copy button has been added which copies the seed directly in the clipboard. All the texts that were there to be copied have been modified to text inputs with an unmodified value, as it is more common nowadays to copy text from a box. In addition, all texts in text inputs are directly selected when clicking on them, allowing easy modification and selection.
  - Address of a LAO - For changing the address when launching a LAO, you had to modify a hard-coded constant, which was not user-friendly at all. We added a text input in the Launch screen so that the address can be directly set when creating an organization.
  - Disabling swipes - On the web, it was possible to navigate the application using swipes (like you would do on mobile applications like Snapchat). It led to issues because when trying to select some text, it navigated to the right or the left screen. On top of that, this type of navigation seems counter-intuitive on the web. That is why we disabled it.
  - Modals - There was not any component to display pop-up messages in the application. Two types of generic modals – the UI term for a pop-up message – have been added to the components' folder. One is for showing information that is going to be dismissed, and the other one is to ask for the user's confirmation when needed.

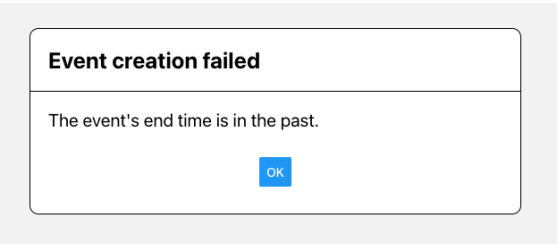

Figure 8.3: Modal to dismiss

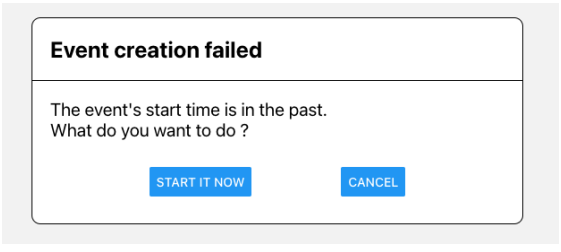

Figure 8.4: Modal for confirmation

- Refactoring - When browsing the React components, we saw that there was not a generic text input class. Text inputs generally have two forms in our front-end: either being one

one-line at the bottom or being in a box. All one-line inputs have the same stylesheet and behavior, so we created a generic class for them.
Generally, when making a pull request that became larger and larger because of comments addressing, we preferred to refactor what could be refactored in a separated PR. As an example, the modals we added were not generic at first. They were only event-specific, and have been modified afterwards.

- Documentation - Along with the documentation we added on the new features, we took care of documenting past code that we went through and understood. For example, we described all existing files in *extmodel/network/method/message/data*. These are all the types of messages that we have in our system. When past documentation did not meet TSDoc's requirements [21], we modified it as well.

- Tests - The project lacked a library to test React components. We added it [22] and could test components using Jest. For example, we tested some buttons that are used in multiple parts of the application.
  When adding new features that relied on code given to us, we sometimes tested the past code first to be sure that it works as intended. To be able to test some classes and functions, we needed to use Jest mocks. We started with basic tests and were then able to test more complex classes by mocking external dependencies. There was not a lot of mocks at the beginning, but now, there are a lot of examples to take inspiration from.

8.3.1 Future Work

- Incomplete features - Even though most of the required functions already exist, the implementation for re-opening a roll call and closing an election is still missing. For now, the only way to close an election is to wait for it to finish.

- LAO connection issue - We have an issue when connecting to the same LAO from the same device. After connecting to a LAO, closing, and re-opening, the window navigates back to the home page instead of the LAO page.

- Improve roll call - Our current implementation does not verify if the scanned data is a valid pop token in roll call. For example, if we scan the QR code of a LAO, we will receive the same message notifying the scan succeeds as we scanned the pop token of an attendee. It would be nice if in the future we can catch this fault and display "invalid PoP token" in an error message, without adding the data to the set of attendees.

- Tests and documentation - Since the tests were not explicitly required before, a lot of existing codes do not have their tests. We added tests and documentation during the semester but there are still many that need to be done.

- Code smells - There are 132 code smells according to the CI, future work should always aim at reducing this statistic.

- UI design - The overall design of the application should be improved. For example, a button that takes the width of the whole screen is not perfect in terms of user experience.

8.4 Front-end 2 - Android

The codebase is huge and has its flaws. Especially the way the UI was tested, for some it was not easy to read through and even harder to write tests. To make the code more robust and easier to maintain, we refactored the handlers to make it more scalable.
We also added Hilt to the project to handle dependency injection, added types for handling keys and a settings tab.
There were several major bugs that crashed the app on a single button click that needed to be resolved.

- Tab settings - To change the server URL on which the application was connected, we had to change it manually in the injection file before building the APK. Hence, the idea to create a tab setting, currently it only contains a try to modify the server URL. It is a temporary solution as, in the future, the application will connect to a server by scanning a QR code.
- Refactoring handlers - Handlers were using booleans to know whether we should enqueue the result to reprocess the message or not. We replaced them by using errors, created for this purpose. The result is now enqueued when catching an error. We also created a registry to improve scalability as they could be many more (action, object) pairs in the future, which would be time-consuming to maintain as it was.
- Incoming message verification against the protocol's JSON-SCHEMA [23] - Implemented a verification system of incoming messages against the protocol's schema.
- Testing UIs - Previous interface tests were a total mess, they seriously needed a test framework to be readable and easier to implement. To simplify them, we created a folder, *androidTest/pages* containing page object of the activities and fragments. A page object is used to centralize the selection of the items in the views.
  This makes writing and reading tests easier. We also created a *FragmentScenarioRule* which is a test rule that creates a fragment for each test. It allowed us to remove the before and after function as it is already defined in the rule.
- Hilt - Originally, the injection was done manually through the Injection class.
  It was a working strategy, but it made the tests hard to write, as it was difficult to mock specific dependencies in the integrated tests.

  Hilt is an annotation based dependency injection tool that is created specifically for Android.
  The tool is therefore built for the MVVM[1] architecture.
  With this tool, it is very easy to swap an injected dependency during a specific test with a mocked one, allowing developers to write tests easily.

[1] Model-View-ViewModel

For more information about Hilt, it is advised to read its dedicated documentation on the Android website. [10]

- Security type refactor - The Android subproject was using strings as a type for almost everything. This harmed the readability of the project, and a great refactor was done to create a dedicated type for each purpose: PublicKey, MessageID, Signature, etc.

8.4.1 Future Work

There are still a lot of flaws in the application that needs to be addressed :

- Create types for the IDs - The security type refactoring is a great step forward, but types can be improved, especially for the different IDs in the system.
  It would render the code easier to read and more maintainable if each ID had their own type.
- Create a dedicated WebSocket connection service. - Currently, it creates a WebSocket connection at startup and the connection is killed when the application ends. To change the URL dynamically, the settings tab uses a custom RequestBuilder.
  This is not a good system, and it would be a lot better to create a dedicated service that would provide creation and completion of WebSocket connections.
- Add more verifications on incoming messages - The received messages are now verified against the protocol's schema, but most verifications are still to be done.
  The validation of the sender's signature, the verification of the message ID value, the LAO ID, the election ID, etc.
  This issue should be addressed as it draws a great hole in the system's security.
- Discard bad messages after some time - Currently, when an incoming message cannot be handled, it is added back into the queue as it might be caused by a reorder issue.
  But there is no rule to discard the message. If a message is simply bad and will never be handled correctly, it is kept in the queue forever.
  This is an important issue as it creates a massive memory leak.
- Refactor the *LAORepository* into multiple sub-services - The *LAORepository* has too many responsibilities, which makes it hard to test and to mock.
  It should be split into multiple services.
- Add more unit tests - A big chunk of the codebase is still not tested.
- Address naming convention mistakes in some classes - Some classes are using C-like names that does not follow the usual Android style guidelines.
  It is mostly present in the different *ViewModel* classes.

Chapter 9

Conclusion

In this project, we continued to build on the codebase of the PoP application, where we already had working LAOs, and partly working roll calls and elections.
We worked a bit to have fully working roll calls and to have a working election on three out of four systems.
We did a lot of work to solidify this basis by adding tests and test coverage requirements for all our work, by reformatting the code in multiple places and by correcting some bugs.
We also began the implementation of a consensus protocol to add a layer of security when multiple organizers are trying to modify things at the same time and to share some of the power of the organizer to the witnesses, as well as implementing some cross server communication.
Finally, we implemented a social media, to allow the users having taken part in a roll call to interact with each other.

Thanks to Nicolas Pierre Raulin for his help given through the whole semester.

Chapter 10

Appendices

Appendix A

JSON RPC Messages

A.1 Consensus Messages

A.1.1 Consensus Elect

```
{
"jsonrpc": "2.0",
"method": "publish",
"params": {
    "channel": "/root/<lao_id>/consensus/", /* LAO Social Platform */
    "message": {
        "data": base64({ /* base64 representation of this object */
            "object": "consensus",
            "action": "elect",
            "instance_id": "..base 64..", /* unique id of the consensus instance */
            "created_at": 1631280815, /* UNIX Timestamp in UTC */
            "key": {
                ""type": "election", // what object the consensus refers to
                "id": election_id, // what object id the consensus refers to
                "property": "state" // what property of the object the value refers to
            }
            "value": "started" // the value proposed in the consensus
        }),
        "sender": "..base64..", /* Public key of sender */
        "signature": "..base64..", /* Signature by sender over "data" */
        "message_id": "..base64..", /* hash(data||signature) */
        "witness_signatures": [ /* Array of witness signatures */
            {
            witness: "...base64...", /* Public key of witness */
            signature: "...base64...",/* Signature of (message_id) */
            }, ...
        ]
    }
```

```
 },
 "id": 8 /* unique request identifier */
}
```

A.1.2 Consensus Elect-Accept

```
{
"jsonrpc": "2.0",
"method": "publish",
"params": {
    "channel": "/root/<lao_id>/consensus/", /* LAO Social Platform */
    "message": {
        "data": base64({ /* base64 representation of this object */
            "object": "consensus",
            "action": "elect_accept",
            "message_id": "..base 64", /* message_id of the "elect" message
            "instance_id": "..base 64..", /* unique id of the consensus instance */
            "created_at": 1631280815, /* UNIX Timestamp in UTC */
            "accept": boolean, // indicating whether the proposal is accepted (true)
                or rejected (false)
        }),
        "sender": "..base64..", /* Public key of sender */
        "signature": "..base64..", /* Signature by sender over "data" */
        "message_id": "..base64..", /* hash(data||signature) */
        "witness_signatures": [ /* Array of witness signatures */
            {
            witness: "...base64...", /* Public key of witness */
            signature: "...base64...",/* Signature of (message_id) */
            }, ...
        ]
    }
 },
 "id": 8 /* unique request identifier */
}
```

A.1.3 Consensus Prepare

```
{
"jsonrpc": "2.0",
"method": "publish",
"params": {
    "channel": "/root/<lao_id>/consensus/", /* LAO Social Platform */
    "message": {
        "data": base64({ /* base64 representation of this object */
            "object": "consensus",
            "action": "prepare",
```

```
            "message_id": "..base 64", /* message_id of the "elect" message
            "instance_id": "..base 64..", /* unique id of the consensus instance */
            "created_at": 1631280815, /* UNIX Timestamp in UTC */
            "value": {
                proposed_try: int
            }
        }),
        "sender": "..base64..", /* Public key of sender */
        "signature": "..base64..", /* Signature by sender over "data" */
        "message_id": "..base64..", /* hash(data||signature) */
        "witness_signatures": [ /* Array of witness signatures */
            {
            witness: "...base64...", /* Public key of witness */
            signature: "...base64...",/* Signature of (message_id) */
            }, ...
        ]
    }
 },
 "id": 8 /* unique request identifier */
}
```

A.1.4 Consensus Promise

```
{
"jsonrpc": "2.0",
"method": "publish",
"params": {
    "channel": "/root/<lao_id>/consensus/", /* LAO Social Platform */
    "message": {
        "data": base64({ /* base64 representation of this object */
            "object": "consensus",
            "action": "promise",
            "message_id": "..base 64", /* message_id of the "elect" message
            "instance_id": "..base 64..", /* unique id of the consensus instance */
            "created_at": 1631280815, /* UNIX Timestamp in UTC */
            "value": {
                "accepted_try": int
                "accepted_value": bool
                "promised_try": int
            }
        }),
        "sender": "..base64..", /* Public key of sender */
        "signature": "..base64..", /* Signature by sender over "data" */
        "message_id": "..base64..", /* hash(data||signature) */
        "witness_signatures": [ /* Array of witness signatures */
            {
            witness: "...base64...", /* Public key of witness */
            signature: "...base64...",/* Signature of (message_id) */
```

```
            }, ...
        ]
    }
 },
 "id": 8 /* unique request identifier */
}
```

A.1.5 Consensus Propose

```
{
"jsonrpc": "2.0",
"method": "publish",
"params": {
    "channel": "/root/<lao_id>/consensus/", /* LAO Social Platform */
    "message": {
        "data": base64({ /* base64 representation of this object */
            "object": "consensus",
            "action": "propose",
            "message_id": "..base 64", /* message_id of the "elect" message
            "instance_id": "..base 64..", /* unique id of the consensus instance */
            "created_at": 1631280815, /* UNIX Timestamp in UTC */
            "value": {
                "proposed_try": int
                "proposed_value": bool
            }
            "acceptor-signatures": ["..base64.."], /* Signatures of all the received
                PROMISE messages*/
        }),
        "sender": "..base64..", /* Public key of sender */
        "signature": "..base64..", /* Signature by sender over "data" */
        "message_id": "..base64..", /* hash(data||signature) */
        "witness_signatures": [ /* Array of witness signatures */
            {
            witness: "...base64...", /* Public key of witness */
            signature: "...base64...",/* Signature of (message_id) */
            }, ...
        ]
    }
 },
 "id": 8 /* unique request identifier */
}
```

A.1.6 Consensus Accept

```
{
"jsonrpc": "2.0",
```

```
"method": "publish",
"params": {
    "channel": "/root/<lao_id>/consensus/", /* LAO Social Platform */
    "message": {
        "data": base64({ /* base64 representation of this object */
            "object": "consensus",
            "action": "accept",
            "message_id": "..base 64", /* message_id of the "elect" message
            "instance_id": "..base 64..", /* unique id of the consensus instance */
            "created_at": 1631280815, /* UNIX Timestamp in UTC */
            "accept": boolean, // indicating whether the proposal is "value": {
                "accepted_try": int
                "accepted_value": bool
            }
        }),
        "sender": "..base64..", /* Public key of sender */
        "signature": "..base64..", /* Signature by sender over "data" */
        "message_id": "..base64..", /* hash(data||signature) */
        "witness_signatures": [ /* Array of witness signatures */
            {
            witness: "...base64...", /* Public key of witness */
            signature: "...base64...",/* Signature of (message_id) */
            }, ...
        ]
    }
 },
 "id": 8 /* unique request identifier */
}
```

A.1.7 Consensus Learn

```
{
"jsonrpc": "2.0",
"method": "publish",
"params": {
    "channel": "/root/<lao_id>/consensus/", /* LAO Social Platform */
    "message": {
        "data": base64({ /* base64 representation of this object */
            "object": "consensus",
            "action": "learn",
            "message_id": "..base 64", /* message_id of the "elect" message
            "instance_id": "..base 64..", /* unique id of the consensus instance */
            "created_at": 1631280815, /* UNIX Timestamp in UTC */
            "value": {
                "decision": bool
            }
            "acceptor-signatures": ["..base64.."], /* Signatures of all the received
                Accept  messages*/
```

```
        }),
        "sender": "..base64..", /* Public key of sender */
        "signature": "..base64..", /* Signature by sender over "data" */
        "message_id": "..base64..", /* hash(data||signature) */
        "witness_signatures": [ /* Array of witness signatures */
            {
            witness: "...base64...", /* Public key of witness */
            signature: "...base64...",/* Signature of (message_id) */
            }, ...
        ]
    }
 },
 "id": 8 /* unique request identifier */
}
```

A.1.8 Consensus Failure

```
{
"jsonrpc": "2.0",
"method": "publish",
"params": {
    "channel": "/root/<lao_id>/consensus/", /* LAO Social Platform */
    "message": {
        "data": base64({ /* base64 representation of this object */
            "object": "consensus",
            "action": "learn",
            "message_id": "..base 64", /* message_id of the "elect" message
            "instance_id": "..base 64..", /* unique id of the consensus instance */
            "created_at": 1631280815, /* UNIX Timestamp in UTC */
        }),
        "sender": "..base64..", /* Public key of sender */
        "signature": "..base64..", /* Signature by sender over "data" */
        "message_id": "..base64..", /* hash(data||signature) */
        "witness_signatures": [ /* Array of witness signatures */
            {
            witness: "...base64...", /* Public key of witness */
            signature: "...base64...",/* Signature of (message_id) */
            }, ...
        ]
    }
 },
 "id": 8 /* unique request identifier */
}
```

A.2 Social Media Messages

A.2.1 Chirp Add

```
{
"jsonrpc": "2.0",
"method": "publish",
"params": {
    "channel": "/root/<lao_id>/social/<sender>/", /* LAO Social Platform + sender */
    "message": {
        "data": base64({ /* base64 representation of this object */
            "object": "chirp",
            "action": "add",
            "text": "My new chirp!", /* UTF-8 encoded chirp */
            "parent_id": message_id /* Either message_id of parent chirp, optional */
            "timestamp": 1631280815, /* UNIX Timestamp in UTC */
        }),
        "sender": "..base64..", /* Public key of sender */
        "signature": "..base64..", /* Signature by sender over "data" */
        "message_id": "..base64..", /* hash(data||signature) */
        "witness_signatures": [ /* Array of witness signatures */
            {
            witness: "...base64...", /* Public key of witness */
            signature: "...base64...",/* Signature of (message_id) */
            }, ...
        ]
    }
 },
 "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 */
}
```

A.2.2 Chirp Notify_Add

```
{
"jsonrpc": "2.0",
"method": "broadcast",
"params": {
    "channel": "/root/<lao_id>/social/chirps/", /* LAO Social Platform + Chirps */
    "message": {
        "data": base64({ /* base64 representation of this object */
```

```
            "object": "chirp",
            "action": "notify_add",
            "chirp_id": message_id, /* message_id of the chirp message above */
            "channel": <channel> /* The channel where the chirp is located */
            "timestamp": 1631280815, /* UNIX Timestamp in UTC of the original chirp */
        }),
        "sender": "..base64..", /* Public key of the server */
        "signature": "..base64..", /* Signature by server over "data" */
        "message_id": "..base64..", /* hash(data||signature) */
        "witness_signatures": [ /* Array of witness signatures */
            {
            witness: "...base64...", /* Public key of witness */
            signature: "...base64...",/* Signature of (message_id) */
            }, ...
        ]
    }
}
}
```

A.2.3 Chirp Delete

```
{
"jsonrpc": "2.0",
"method": "publish",
"params": {
    "channel": "/root/<lao_id>/social/<sender>/", /* LAO Social Platform + sender */
    "message": {
        "data": base64({ /* base64 representation of this object */
            "object": "chirp",
            "action": "delete",
            "chirp_id": message_id /* Message id of the chirp published */
            "timestamp": 1631280815, /* UNIX Timestamp in UTC */
        }),
        "sender": "..base64..", /* Public key of sender, must be identical to
           message_id sender */
        "signature": "..base64..", /* Signature by sender over "data" */
        "message_id": "..base64..", /* hash(data||signature) */
        "witness_signatures": [ /* Array of witness signatures */
            {
            witness: "...base64...", /* Public key of witness */
            signature: "...base64...",/* Signature of (message_id) */
            }, ...
        ]
    }
 },
 "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 */
}
```

A.2.4 Chirp Notify_Delete

```
{
"jsonrpc": "2.0",
"method": "broadcast",
"params": {
    "channel": "/root/<lao_id>/social/chirps/", /* LAO Social Platform + Chirps */
    "message": {
        "data": base64({ /* base64 representation of this object */
            "object": "chirp",
            "action": "notify_delete",
            "chirp_id": message_id, /* message_id of the chirp message above */
            "channel": <channel> /* The channel where the chirp is located (starting
                from social/ inclusive) */
            "timestamp": 1631280815, /* UNIX Timestamp in UTC given by the message
                above */
        }),
        "sender": "..base64..", /* Public key of the server */
        "signature": "..base64..", /* Signature by server over "data" */
        "message_id": "..base64..", /* hash(data||signature) */
        "witness_signatures": [ /* Array of witness signatures */
            {
            witness: "...base64...", /* Public key of witness */
            signature: "...base64...",/* Signature of (message_id) */
            }, ...
        ]
    }
}
}
```

A.2.5 Reaction Add

```
{
"jsonrpc": "2.0",
"method": "publish",
"params": {
    "channel": "/root/<lao_id>/social/reactions/", /* Reactions Channel */
    "message": {
```

```
        "data": base64({ /* base64 representation of this object */
            "object": "reaction",
            "action": "add",
            "reaction_codepoint": <3, /* Emoji indicating a reaction */
            "chirp_id": message_id, /* message_id of the tweet message */
            "timestamp": timestamp /* UNIX timestamp in UTC  */
        }),
        "sender": "..base64..", /* Public key of sender */
        "signature": "..base64..", /* Signature by sender over "data" */
        "message_id": "..base64..", /* hash(data||signature) */
        "witness_signatures": [ /* Array of witness signatures */
            {
            witness: "...base64...", /* Public key of witness */
            signature: "...base64...",/* Signature of (message_id) */
            }, ...
        ]
    }
 },
 "id": 3 /* unique request identifier */
}
```

The "<3" should be an actual emoji, but with our current packages, LaTeX does not let us insert one.

A.2.6 Reaction Delete

```
{
"jsonrpc": "2.0",
"method": "publish",
"params": {
    "channel": "/root/<lao_id>/social/reactions/", /* Reactions Channel */
    "message": {
        "data": base64({ /* base64 representation of this object */
            "object": "reaction",
            "action": "delete",
            "reaction_id": message_id, /* message_id of the add reaction message */
            "timestamp": timestamp /* UNIX timestamp in UTC */
        }),
        "sender": "..base64..", /* Public key of sender */
        "signature": "..base64..", /* Signature by sender over "data" */
        "message_id": "..base64..", /* hash(data||signature) */
        "witness_signatures": [ /* Array of witness signatures */
            {
            witness: "...base64...", /* Public key of witness */
            signature: "...base64...",/* Signature of (message_id) */
            }, ...
        ]
    }
```

```
 },
 "id": 3 /* unique request identifier */
}
```

Appendix B

Production-Ready details

B.1 Production-Ready details

B.1.1 Project-Overview Notation

In this section, a detailed explanation of the entries and the notation is given.
This notation is used in 6.1 and 6.2.

B.1.2 Notation

- **Unimplemented** : The feature is not implemented.
- **No Data** : No Data was collected on this specific feature.
- **Not Working** : The feature is not working properly and cannot be used.
- **Unstable** : The feature is not working properly but can still be used in the main scenarios.
- **Operational** : The feature can be used in all tested scenarios. It does not mean that it is reliable. It only means that the current tests are passing and that one is able to use the functionality properly.
- **Reliable** : The feature has a sufficient number of tests covering all of its aspects. We can say confidently that it can be used properly in almost every situation.

Currently, no feature is defined as reliable. To truly define a feature as reliable, we would need a lot more tests covering much more of their aspects and edge cases.

B.1.3 Entries

Measures

The measures are taken from the SonarCloud analysis The measures of the initial state were taken after the pipeline got fixed, and the coverage was correctly computed. Therefore, on the 1st of November 2021.[1]
The measure of the current state of the system were taken on the 5th of January 2022.

- **Lines of code** : The number of code line that are not part of the test suite.
- **Coverage** : The percentage of lines that are covered by tests.
- **Technical debt** : The computed technical debt in hours of work to fix the issue.
- **Codesmell** : The number of issues automatically found in the codebase.

Features

Most features were tested manually, as automatic testing was not widely deployed, especially at the start of the semester.

- **LAO Creation** : An organizer can create a LAO.
- **Roll call** : A roll call can be created and used to retrieve the PoP Token of the attendees.
- **Re-open a Roll call** : An organizer can re-open a roll call after it has been closed, keeping the already scanned attendees.
- **Election** : An election can be created. And attendees possessing a valid PoP Token can cast a vote.
- **Message Validation** : The incoming messages are validated against the protocol. The JSON-Schema [23] is respected, and the verification fields are valid.
- **Digital Wallet** : The digital wallet produces valid PoP Tokens.
- **Cross-server Comm** : The server can communicate between each-others.
- **Consensus** : A consensus among the different peers can be achieved.
- **Chirp Creation** : A Chirp can be created, sent and displayed when a user's personhood is verified in a roll call.
- **Chirp Deletion** : A Chirp can be deleted by a user.

[1] The Android measurements did not take the XML lint issues into account until the middle of the semester. Therefore, they were recomputed manually using SonarQube [24] on an older commit.

- **Add Reaction** : A user can add a Reaction to a Chirp when their personhood is verified in a roll call.
- **Remove Reaction** : A user can remove one of its Reaction to a Chirp.

B.1.4 Justifications

This section contains the reasoning behind the attributions of the state in the tables 6.1 and 6.2.

No feature is currently defined as reliable, as the system would need a more edge cases tests to be defined as reliable.
For the Unimplemented, No Data and Operation states, no justification is needed as it their definition gives enough information about their state.

Initial state

Roll calls in Scala back-end were implemented but did not follow the protocol, hence rendering the feature unusable with both front-ends. Furthermore, validation of the messages with data was missing. But also another limitation to the functionality of roll calls was the absence of saving the attendees list in the database, due to its strict implementation.

Elections in the front-ends were using the device public key as the sender of the cast-vote message, which was not expected by the protocol.
Also, the Go back-end had not fully implemented the feature yet.

The digital wallet of the web front-end was producing wrong private-keys. Thus, the signing was not working properly.

Current state

The message validation of the Android front-end is not complete. The incoming messages are validated against the schema, but the different hashes and signatures are mostly not verified.

Bibliography

[1] JSON-RPC Working Group. *JSON-RPC 2.0 Specification*. 2010. URL: https://www.jsonrpc.org/specification. (accessed: 05.01.2022).

[2] Gaurav Narula. URL: https://github.com/dedis/student_21_pop/blob/master/be1-go/docs/images/flowchart.png. (accessed: 05.01.2022).

[3] *Akka*. URL: https://doc.akka.io/docs/akka/2.6/stream/stream-introduction.html. (accessed: 06.01.2022).

[4] *Akka-stream*. URL: https://doc.akka.io/docs/akka/current/stream/index.html. (accessed: 06.01.2022).

[5] Pierluca Borsò. *PoP Web Frontend*. 2021. URL: https://github.com/dedis/student_21_pop/blob/master/fe1-web/docs/README.md. (accessed: 04.01.2022).

[6] AirbnbEng. *Airbnb JavaScript Style Guide*. URL: https://github.com/airbnb/javascript. (accessed: 04.01.2022).

[7] ESLint. *ESLint: Find and fix problems in your JavaScript code*. URL: https://eslint.org. (accessed: 04.01.2022).

[8] SonarSource SA. *SonarCloud*. URL: https://sonarcloud.io. (accessed: 04.01.2022).

[9] Scarlet. *Scarlet: A Retrofit inspired WebSocket client for Kotlin, Java, and Android*. URL: https://github.com/Tinder/Scarlet. (accessed: 05.01.2022).

[10] Hilt. *Dependency injection with Hilt*. URL: https://developer.android.com/training/dependency-injection/hilt-android. (accessed: 05.01.2022).

[11] Google. *Google Java Style Guide*. URL: https://google.github.io/styleguide/javaguide.html. (accessed: 05.01.2022).

[12] Jakob Nielsen. *10 Usability Heuristics for User Interface Design*. 2020. URL: https://www.nngroup.com/articles/ten-usability-heuristics/. (accessed: 05.01.2022).

[13] *benbjohnson/clock*. URL: https://github.com/benbjohnson/clock. (accessed: 06.01.2022).

[14] Bobby Allyn. *Researchers: Nearly Half Of Accounts Tweeting About Coronavirus Are Likely Bots*. 2020. URL: https://www.npr.org/sections/coronavirus-live-updates/2020/05/20/859814085/researchers-nearly-half-of-accounts-tweeting-about-coronavirus-are-likely-bots. (accessed: 04.01.2022).

[15] *React Blockies*. 2019. URL: https://www.npmjs.com/package/react-blockies. (accessed: 04.01.2022).

[16] *Karate*. URL: https://github.com/karatelabs/karate. (accessed: 06.01.2022).

[17] *Appium*. URL: https://appium.io/. (accessed: 06.01.2022).

[18] *Gatling*. URL: https://gatling.io/. (accessed: 06.01.2022).

[19] *networknt/json-schema-validator*. URL: https://github.com/networknt/json-schema-validator. (accessed: 05.01.2022).

[20] *sbt*. URL: https://github.com/sbt/sbt. (accessed: 06.01.2022).

[21] Rush Stack (Microsoft). *TSDoc*. URL: https://tsdoc.org. (accessed: 05.01.2022).

[22] Callstack. *React Native Testing Library*. URL: https://github.com/callstack/react-native-testing-library. (accessed: 05.01.2022).

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

[24] SonarSource SA. *SonarQube*. URL: https://www.sonarqube.org/. (accessed: 07.01.2022).