Proof of Personhood
Project Report

Semester Project

Louis Bettens, Romain Birling, Diego Boros, Stefan Eric, Robin Goumaz, Nico Hauser, Anders Hominal, Zoé Marin, Johann Plüss, Ajkuna Seipi, Maxime Zammit

June 10, 2022

Supervisors: Prof. B. Ford, P. Borsò, J. Viaene
Advisors: G. Fleischer, N. Kocher, N. Raulin
K. Schneiter, L. Merino, H. Zhang

Decentralized Distributed Systems Laboratory, EPFL

Contents

Chapter 1

Introduction

This is a report describing the work done as part of the *Proof of Personhood* semester project at the DEDIS Lab at EPFL. We will start with a general introduction of important terms and concepts, then outline the project's ideas and goals and finish the introduction with the system architecture that was already determined before we started working on this project.

1.1 Proof of Personhood (PoP)

Online communication plays a big role in everyday life: People communicate with other people using messengers and read content posted by others in social networks. In contrast to conventional conversations in the real world, in the digital world it is simply not possible to determine what person you are talking to without having met them in real world. A user id or display name cannot be authenticated and reliably linked to a real person. Even for *verified* profiles you have to rely on the service provider's verification mechanism. And things are actually even worse: You do not even have the guarantee that the other party is a real person. In almost all online services there is a many-to-many relationship between online accounts and real people: Many real people can own none or a single online account and a single real person can own none to multiple online accounts.

For certain applications it is desirable, for others even of utmost importance, to be able to ensure that a single real person can own at most one online account. E-Voting is one of the most prominent examples where it is immediately clear that only real people should be allowed to vote. For a more comprehensive list of challenges, especially with respect to voting in democracies, whose solutions often prerequisite a proof of personhood, we refer to B. Ford [4].

Current "solutions" such as captchas may give evidence that a certain user

probably is a real person but even assuming it is impossible for machines to solve them, nothing prevents a single real person from solving multiple captchas. Thus strong guarantees, such as an actual *proof* of personhood (PoP), are required to prevent a single user from having multiple online identities.

One way to achieve this proof of personhood is by relying on the real world: Real people can only be at one place at a given point in time. Moreover robots are nowhere close to being indistinguishable from humans which means we can leverage this property to proof one's personhood in so-called *roll calls* which are introduced as *Pseudonym Parties* by B. Ford [7].

1.1.1 Roll Calls

A roll call for proving one's personhood is an event taking place in the real world. The goal of a roll call is that *attendees* can prove they are in possession of a real physical body. It assumes that exclusively people that have a real physical body are capable of attending such an event. Each attendee is supposed to generate a public/private key pair before attending the event and the output of a roll call is then a list of public keys where each key corresponds to at most one person. Each attendee has to be able to verify that this list is consistent, meaning their key is on the list and the list contains at most as many keys as there are people at the event. This makes sure that if fraud occurs, it can be detected.

One way of achieving such a roll call event is by having an organizer that is in charge of running the event and a delimited area. All people who want to attend the roll call can then enter this area until the event starts. After the event starts, it must be ensured that nobody can enter the area anymore. Attendees can then, one after another, leave this restricted area through the official exit where the organizer waits for them and takes note of their public key by, for example, scanning a qr code. After doing so, a big, publicly visible counter that was initialized to 0 increases by one. It keeps track of the number of real people attending the roll call. All attendees in the room have to be able to witness this process, i.e. be able to verify that the counter increases only if the organizer just scanned the public key of one attendee.

After the room is empty, the organizer publishes the list of public keys and all attendees can by themselves verify that the number displayed on the counter matches the number of public keys. If they do, then the length of the list of public keys corresponds to the number of real people who were scanned (Remember, each attendee was able to verify that the counter only increased if one real person left the room). Moreover each attendee can verify their public key is on the published list. Both together ensure that fraud is detectable if it happens. In any case, each attendee was able to verify that the length of the

published list corresponds to the number of real people without trusting the organizer of the event, thus satisfying the above described requirements.

For a more comprehensive description we refer to B. Ford [7].

1.2 The Project

The PoPStellar project was started by previous students in earlier semesters and includes sample use cases and applications leveraging a proof of personhood based on the example described in Section 1.1.1. In addition to roll calls, PoPStellar makes use of the concept of *local autonomous organizations* (LAOs). LAOs are run by an organizer (the same responsible for running the roll calls as previously described) and can be joined by people who want to attend a roll call. The system consists of a standard front- and back-end setup where the back-end is run by the already mentioned organizer. For an attendee to join a LAO, they need to know the the address of the back-end's server and the LAO's id (this allows a single server to host multiple LAOs).

After joining the LAO, attendees can attend roll calls to prove their personhood. Public keys of attendees (as described in Section 1.1.1), that are part of the published list after a successful roll call, are called *PoP Tokens*.

PoPStellar supports different features such as e-voting or social media that make use of these PoP tokens. In e-voting, attendees have to sign their votes using the PoP token to make sure each vote was cast by an actual person. Similarly in the social media feature, posts and reactions are signed using PoP tokens.

The project this report is about builds on the work of previous students. Each student working on the project was assigned a technology stack and a feature they are mainly supposed to work on. From the start, PoPStellar was developed using *n*-version programming where multiple version of the same software are implemented by different teams. This requires the interoperability of the systems and thus makes it necessary to have a clear specification of the communication protocols used. In PoPStellar there are four subsystems: two front-ends and two back-ends. One back-end is implemented in Go, one in Scala and for the front-ends there is a version written in Java (for Android) and one written in TypeScript using `react-native`.

Regarding the features there was a team, this semester, that worked on secret ballot elections (e-voting), a team working on a new feature, digital cash, and lastly a team working on the production-readiness of the whole system.

1.3 System Architecture

As hinted at in Section 1.2, the system is built on a client-server architecture with the clients being called *front-end*s and the servers *back-end*s. In the next two Sections we describe how they communicate and what also what kind of roles the users can have.

1.3.1 Communication

The front-ends communicate with back-ends using the websocket protocol which is a message based protocol abstraction on top of TCP. On top of the websocket protocol, the system uses JSON RPC to implement the publish-subscribe pattern for messages. This pattern groups messages into *channels* where messages are broadcasted to all *subscribers* of that channel. The back-ends keep track of what clients subscribed to what channels and of *broadcasting* the corresponding *published* messages in the respective channels. In addition to *publish*, *broadcast*, *subscribe* and *unsubscribe*, the system supports *catchup* messages that request the back-end to send the list of all messages previously sent in a given channel. This is very useful as it allows new subscribers to catch up on previous communication in a channel.

The just described communication layer is called the *low-level communication layer*. Built on top of this layer is the *mid-level communication layer* which ensures *authenticity* of messages by including the public key of the sender and a signature of the message content. Moreover all messages include an identifier that is the result of a hash function computed on the message content. As long as the used hash function is collision-resistant, this identifier is unique. The message content is the last layer, the *high-level communication layer*. Since signatures are computed on binary data and JSON representations of data are not unique, a trivial approach would result in signatures that are hard to verify. Thus the message content is encoded in Base64 and the resulting UTF-8 string is then signed. Another approach would have been to restrict the JSON syntax to a subset making the representation unique.

1.3.2 Organizers and Witnessing

The system requires attendees with different special roles to operate. First we have the *organizer* of the LAO that runs a back-end and provides the infrastructure. In roll calls, the organizer is in charge of creating and publishing the list of public keys as described in Section 1.1.1. In many other features such as e-voting, the organizer also has a special role such as setting up the election and requesting the back-end to tally the votes.

Second there are *witnesses* that are in charge of observing the whole process and help detecting fraud when it happens. In the beginning of the semester, the system did not have functional witnessing even though it is an integral

part of the system design. Thus the remainder of this section will not reflect what is implemented but rather describe how the system is supposed to function.

As just mentioned, witnesses should help detecting fraud. One type of fraud is censorship by the organizer: If all communication goes via the organizer, a malicious organizer can selectively not forward messages such as votes from certain attendees. Even though *authenticity* of the votes is guaranteed by the proof of personhood, e-voting will not be secure since *availability* can be broken. To reduce the likelihood of censorship, witnesses are supposed to run a back-end analogous to the one of the organizer. The back-ends of witnesses also keep track of all messages to keep the organizer in check. Attendees fearing censorship can send their vote to the organizer and to some witnesses that in turn then forward the message to the organizer. The organizer and witness are able to de-duplicate the message based on the unique message identifier. While this does not prevent censorship, the likelihood of it being detected increases with the number of witnesses.

In addition to forwarding messages, witnesses are supposed to sign all messages they see and agree with. The resulting signature can then be broadcasted. The organizer receiving the signature of witnesses is interested in forwarding it to connected clients as it will increase the trustworthiness of the message. In contrast, if an attendee connected to the organizer knows that some trusted witness is part of the LAO but never receives the signature of this witness, they have good reason to suspect fraud. Assuming the witness is trusted, they will sign the message when they receive it. Thus not receiving the signature of the witness indicates that the witness probably never obtained the message which is exactly what happens if the organizer censors the witness.

Chapter 2

E-Voting

2.1 Introduction

The E-Voting feature uses PoP tokens issued in a roll call (see Section 1.1.1) to facilitate elections where each real person can vote at most once following the principle *one person, one vote* [15].

2.2 Previous Work

This Section describes the work done by previous students working on the project. This will be later be used to highlight the changes made this semester.

2.2.1 Functionality

In previous work on the project, support for *open ballot* elections has been implemented. In open ballot elections votes are cast in public which means that all attendees can see what other attendees voted. To guarantee that a given vote was cast by a real person, attendees sign their vote using a PoP token they have previously received in a roll call. In this setup, each attendee can easily compute the outcome of the election by themselves: 1) Collect all casted votes, 2) Filter for votes signed by a valid PoP token, i.e. one that was part of the list published after a roll call, 3) Add up the votes. The result of this computation can then be compared to the officially published result in order to detect potential fraud.

2.2.2 Design

The functionality described in Section 2.2.1 is achieved by running a protocol after having performed a successful roll call. The protocol consists of four messages: `election#setup`, `election#cast_vote`, `election#end` and `election#result`.

All messages except for `election#result` are sent by a front-end then broadcast by a back-end as usual. The message `election#result` is sent by a back-end after having received an `election#end` message. A back-end then performs the computation of the election result as described in Section 2.2.1 and broadcasts it afterwards.

The other three messages, `election#setup`, `election#cast_vote`, `election#end`, accomplish what their name suggests: `election#setup` is sent by the LAO's organizer and creates a new election in the LAO, `election#cast_vote` is sent by any LAO attendee in possession of a valid PoP token to cast their vote and `election#end` is again sent by the LAO's organizer to terminate the election.

An election object had the following properties:

- `id` - The unique identifier of an election. It is computed by running a hash function on a constant, the unique identifier of the associated LAO and the properties `created_at` and `name`.
- `lao` - The unique identifier of LAO this election is associated to
- `version` - Set to `1.0.0`. Added to support versioning at a later point.
- `created_at` - Unix timestamp of the time the election is created
- `start_time` - Unix timestamp of the time the election starts
- `end_time` - Unix timestamp of the time the election ends
- `questions` - An array of questions, each having a unique identifier and a set of options. The unique message identifier is the result of computing the hash over a constant, the unique election identifier as well as the question string.

The `start_time` and `start_end` times are hard limits meaning only votes cast in between are valid. The different subsystems keep track of time individually and do *not* run a time synchronization protocol.

2.2.3 Security Considerations

E-Voting applications without security requirements are trivial to implement. To have an application usable in real word scenarios we at least require *authenticity* and *integrity* of the cast votes and the election result as well as *availability* guaranteeing to attendees the ability to cast votes. For open ballot elections, *confidentiality* is *not* required.

Authenticity and Integrity

Authenticity and integrity of the cast votes is ensured by requiring attendees to sign a hash of the message containing their vote using a previously

obtained PoP token as described in Section 1.1. This certifies that a given vote is cast by a real person and if the contents of the message were changed, it would invalidate the signature on the hash. If a given person casts multiple votes, only the last vote is considered valid.

Each vote in the `election#cast_vote` message possesses a unique identifier that is the result of computing a hash function over the election and question identifier as well as the selected option. Assuming the hash function is collision resistant, this hash binds the vote to a given election. Thus the computation of this identifier must be verified by all receiving parties in order to guarantee the binding and to prevent replay attacks.

The validity of the `election#result` message could only be checked by counting the cast votes by all attendees since the public key of the organizer's back-end is not known and thus the signature cannot be verified.

Availability

With the given architecture (see Section 1.3) it is hard to guarantee *availability* but it is possible to rely on witnesses as described in Section 1.3.2. While this does not guarantee *availability*, it makes fraud detectable as long as a given attendee sends its vote to at least one honest witness. Unfortunately the ability to connect to a LAO via a witness server was not implemented yet. This makes it possible for malicious organizers to prevent different attendees from voting without anybody noticing.

2.3 Goals

First the primary goal of this semester is described on a high level and afterwards the approach to achieve the goal will be outlined.

2.3.1 Primary goal

The primary goal of the E-Voting project this semester was to enable *secret ballot elections* where attendees can cast their votes in private meaning encrypted. This comes with new problems: First it is not clear who should be in charge of decrypting the votes and second it is no longer straight-forward how attendees can verify the published result of an election if the votes are encrypted. The security properties authenticity, integrity and availability as described in Section 2.2.3 should be maintained while additionally providing *confidentiality* of the cast vote.

It might not be straight-forward why *confidentiality* of the votes is necessary if the PoP tokens are not linked to an identity as this provides a form of *pseudonymity*. The problem is that this pseudonym does not change between votes which means votes from different elections as well as other actions

inside the LAO that make use of the PoP token are linkable. Depending on how long public keys are valid, this might allow attackers to uncover the identity of certain public keys.

There are alternative cryptographic primitives that provide unlinkable signatures. Unfortunately a trivial use of them would make it impossible to prevent attendees from voting twice since it is by definition not possible to link the two votes. What would be required is a scheme that has signatures that are linkable within an election but unlinkable between two different elections. A simple solution is to use a different public/private key pair for every election. Unfortunately this solution makes it harder to guarantee that the different keys owned by a single person are not linkable while at the same time guaranteeing that the number of key sets at a roll call matches the number of people or how do you prevent attendees from using two different keys in an election.

Solving these problems seems to be doable but would come with a high engineering effort whereas allowing attendees to cast their vote in private circumvents these problems and is in comparison easy to implement.

2.3.2 Approach

It was decided that in order to support secret ballot elections, ElGamal elliptic curve (Curve25519) public key cryptography is used. All organizers and witnesses derive a shared *election key* for a given secret ballot elections. Attendees can then use this public key to encrypt vote and send this as the vote value instead of the plaintext value. After the election ends, all organizers and witnesses must collaborate to decrypt the votes using Neff shuffles [10] which distributes the trust among the set of all organizers and witnesses.

Since at the beginning of the semester server-to-server communication was only supported in a limited way, there was not enough time to fully implement to described approach and in the current state of the projects there is only one organizer and no witnesses which means that the single organizer of the LAO can decrypt the messages by themselves. Security-wise this is far from ideal but unfortunately we did not manage to accomplish more during the semester. In the following sections we will restrict ourselves to this simplified version where there is only one organizer and no witnesses.

2.4 Design

The protocol design for secret ballot elections is simple. It consists of one additional message called `election#key` and some changes to the `election#setup` and `election#cast_vote` messages. In the following Sec-

tions the changes to the existing messages as well as the specification for the new `election#key` are described.

2.4.1 `election#setup`

The `election#setup` message needs to contain information to differentiate between open and secret ballot elections. Since the message already contained the unused `version` field (see Section 2.2.2), it was decided to change the type of this field to an enumeration of the values `OPEN_BALLOT` and `SECRET_BALLOT`.

2.4.2 `election#key`

As soon as the LAO organizer's back-end receives `election#setup` message for a secret ballot election, it generates an ephemeral public key and sends an `election#key` message telling attendees what keys they should use for the just created election. It is important to include the unique identifier of the election in this message as this is only way to bind the key to a given election.

2.4.3 `election#cast_vote`

After having received the public key for a given election, attendees can cast their vote in a confidential way. They convert the integer index of the ballot option to a 2 byte big-endian value and use ElGamal to encrypt this byte value using the provided election key. The resulting binary is encoded in Base64 and sent instead of the plaintext ballot option index.

2.4.4 Security Considerations

Assuming a trusted LAO organizer, the described implementation provides confidentiality of the votes since the organizer is the only party knowing the secret key and thus being able to decrypt the votes. As described in Section 2.3.2, the goal is to later extend this to a system where all organizers and witnesses only know a share of the key and no single party can decrypt the votes by themselves. In this setup, confidentiality will then be provided as long as at least one party is honest.

For availability, this implementation once again relies on witnesses as already described in Section 2.2.3. Different subsystems worked on providing support for witnesses this semester but unfortunately full support was not achieved yet.

One major issue the just described implementation has is that if in the future an attendee connects to a LAO via the server of a witness as described in Section 1.3.2, it is impossible for attendees to verify the authenticity of the `election#key` message. Thus a malicious witness could simply spoof an

`election#key` with a public key where the malicious witness knows the corresponding private key. Thus the need for verifying the authenticity of the `election#key` message arises.

The underlying issue is that for organizers and witnesses, there is no binding between their front- and back-ends. To solve this problem, the additional `lao#greet` message was introduced which is sent by a back-end right after the creation of the LAO. back-ends use this message to advertise their public key, the public key of the corresponding front-end. Of course this message alone does not suffice since this is only a one-sided claim and would allow back-ends to arbitrarily claim to be the back-end corresponding to a given front-end. To make this binding sound, we need a bidirectional claim. An easy way to achieve this without introducing more complexity is to rely on witness signatures: Attendees should only treat a `lao#greet` message as valid as soon as it is signed by the corresponding front-end (by using witness signatures). The high-level idea is that the back-end claims to be the back-end corresponding to some front-end and the front-end agrees to this binding by adding its witness signature.

2.5 Other Changes

In addition to secret ballot elections, there are a few other things in the e-voting project that changed during this semester. First and foremost the election state progression was changed by removing reliance on time for starting and closing it. Instead of elections automatically starting at their start time and requiring them to stay open at least until their end time, the opening and ending of elections now exclusively relies on messages. For ending elections, the `election#end` message already existed and only the comparison between the election end time and the current time had to be removed. For opening an election, an `election#open` message analogous to `election#end` was introduced. Both can only be sent by the LAO organizer.

By changing this, election start and end time become more of a "this is when an election is supposed to start and end" rather than a strict timing. This removes the necessity of time synchronization that would otherwise be required to make the system robust. Instead it introduced a critical problem: organizers could behave arbitrarily and close elections as soon as their favorite option is winning. This should in the future be solved by adding consensus to these messages. (Students last semester implemented consensus for a subset of messages)

Furthermore we simplified the e-voting protocol by disabling and changing certain parameters. For instance we changed to protocol to require the write-in option to be set to false since support for it was never implemented.

Should this feature be implemented in a future semester, the option is still in the protocol and can be changed back to any boolean value.

Analogous to this, multiple-choice voting was never supported either. Given that it additionally would have caused issues with the implementation of secret ballot elections, we decided to remove this feature from the protocol completely. In order to support multiple-choice voting in secret ballot elections, we would have been required to decide between 1) encrypting each option individually, 2) encrypting all options together. Option 1) would have leaked the number of selected options and option 2) would have resulted in problems with the encryption algorithm since without a limit on the number of options the input size becomes unbounded but the used encryption algorithm only supports messages up to certain size.

2.6 Future Work

This section describes the work that needs to be achieved in the near future concerning the E-voting part, especially the secret-ballots implementation.

2.6.1 Shuffling and Partial Decryption

The initial goal was to implement secret ballot elections in two steps. The first step consists of implementing encryption and decryption function with the help of the DEDIS library *kyber*, and simply let the LAO's organizer decrypt the votes. As described above this step has been implemented, there are just a few bugs that need to be fixed before the presentation. The second step would decentralize the trust from the organizer to the set of all organizers and witnesses but unfortunately this has not been implemented yet. More concretely, the second step would have been to implement Neff Shuffles [10]. Each shuffling step that is performed by each organizer and witness consists of reordering the list of votes and then multiplying the encrypted votes with some random, hidden secret to unlink the in- and output ciphertexts while maintaining integrity of the votes. After that, the votes must be decrypted to determine the winner. Since now each party that participated in the shuffle knows a secret they used to unlink the ciphertext, they must also all participate in the decryption process.

2.6.2 Consensus

Following what is stated above, Consensus is an important part for the E-voting project. It enables the organizer and witnesses of a LAO to agree on a shared viewpoint, such as the deadline of casting votes. In the near future, it should be a priority to implement consensus for opening and closing elections to prevent the organizer from interfering with the result.

2.6.3 Multiple Voting Methods

As stated above, we decided to focus on the *plurality voting method*. The voters can only vote one time per question. If desired, future work should be able to add other voting methods without any major issues. The `version` field for elections probably has to be changed once more and the en- and decryption methods probably have to be adjusted to, for example, account for allowing the selection of multiple options.

Chapter 3

Digital Cash

3.1 Introduction

The Digital Cash feature allows LAOs to create their own virtual currency to enable members to participate in a local autonomous economy built around the LAO. Proof-of-personhood can be used to implement a form of so-called basic income, where every attendee receives a set allowance from the LAO, anonymously.

3.2 Previous Work

The digital cash functionality is new for this semester. Hence, this section will briefly outline the already existing generic features of the system that are used to implement digital cash.

As mentioned previously, PoP tokens are public keys where the owner knows the corresponding private key. This allows LAO members to produce digital signatures that can be attributed to the PoP token. This forms the basis of how members hold and spend coins. They can sign a message to show their intent to send someone money.

To communicate, we use the existing publish-subscribe (pubsub) channel infrastructure. All transactions are published on a new dedicated channel so that attendees can keep up with them.

Finally, to allow controlled issuance of new currency, we use the personhood property. The organizer knows that each token represents a distinct person, hence, if the policy is that each person receives a certain amount of currency, it suffices to send that amount to all valid PoP tokens.

3.3 Goals

The main goal of the semester is to enable the organizer to issue money, and LAO members to send this money around using transactions. The second goal is to allow for transactions in an environment where the sender or both parties, the sender and the receiver, do not have access to the organizer's server through the network. The third goal is to enable organizers and witnesses to bundle together past transactions in such a way that participants can catch up with the system without going through every message individually. In parallel, the security aspects of the system should also be taken into account.

3.3.1 Approach

The LAOCoin design is based on b-money[6], although we dispense of the requirement that money creation be backed by solving computational problems and omit contract functionality. Instead, money is issued by the organizer according to LAO by-laws. The accounting model is based on an implementation of b-money called Bitcoin[9].

In a LAO where digital cash is used, each PoP token can hold a balance of LAOCoins. As is the case in b-money and Bitcoin, LAO members can transfer money by signing a message describing the transfer with their PoP token. Such a message is called a *transaction*. Transactions must be broadcast on a designated LAO pubsub channel. This allows participants to reconstruct the ledger by processing all messages on this channel in order. The transaction is a complex object containing several pieces of information about the receiver and the sender. It is attached to a notion of partial order since it points to transactions that have occurred before.

Figure 3.1 shows how the representation of money through transaction. At time 0, the organizer issues 10 coin to 3 members, A, B and C. The transaction output correspond to the amount of LAO coins A, B and C have. At time 1, we see that A and B are sending money. Member A sends 4 coins to B and C. Additionally, they send the 2 remaining coins to themselves (the sender is also the receiver). Similarly, B sends 6 coins to A and the remaining 4 to themselves. Finally, at time 3, A sends money to C. To send money, they have to refer to all the last transaction where he was in the output.

3.4 Design

Multiple design choices had to be made, the first and most important part being the transaction object to be sent across subsystems. When designing it, we kept in mind the fact that we wanted the digital cash system to stay as compatible as possible with the Bitcoin network.

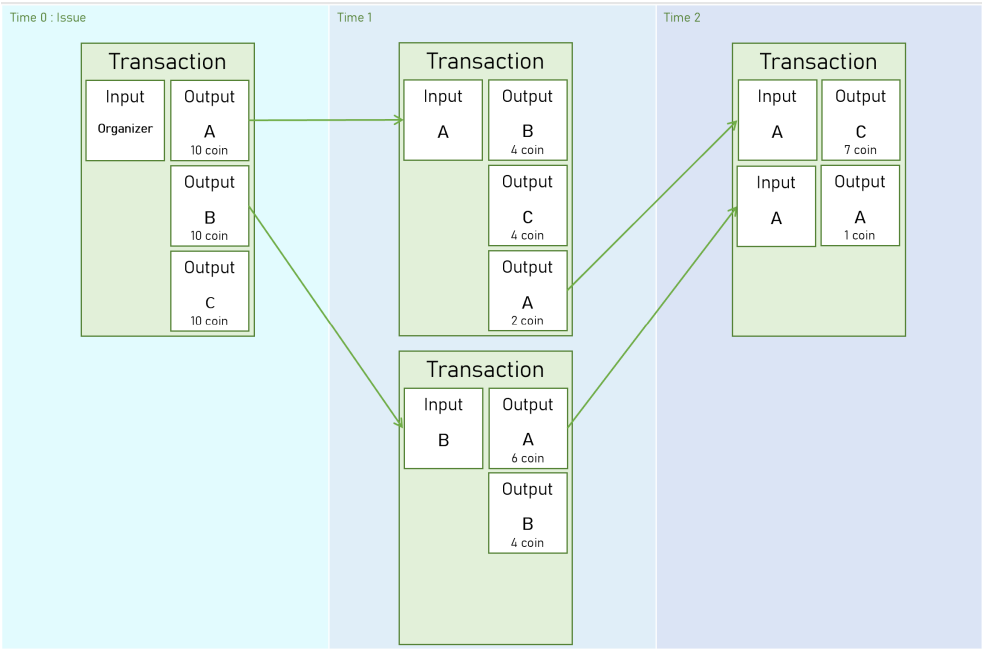


Figure 3.1: Transaction lifetime

As a matter of fact, the Bitcoin network uses the UTXO (unspent transaction output) model to account for account balances.[9, section 9] This means that each transaction is linked to other past and unspent transactions that are used as its inputs. The spending of those coins is represented by outputs, also present in the transaction.

3.4.1 Basic Transaction Design

We will now describe more specifically the design of a transaction according to the defined digital cash scheme. A transaction object will have the following properties:

- `inputs`: An array of transaction inputs, each input will contain:
  - `tx_out_hash`: the id of the transaction we are using as input
  - `tx_out_index`: the index of the output we are spending within the transaction
  - `script`: *unlocks* the output by responding to its script
    - `type`: A defined type that specifies the interpretation of this script, in this case we are using *Pay-to-pubkey-hash*
    - `pubkey`: The owner 's public key of the output we are referring to
    - `sig`: The signature over the inputs and outputs by the owner

- `outputs`: An array of transaction outputs, each output will contain:
  - `value`: The value of this output
  - `script`: *locks* this output by specifying the conditions under which it can be unlocked
    * `type`: *Pay-to-pubkey-hash*
    * `pubkey_hash`: The destination's public key hash
- `version`: The version of the transaction inputs, not used for now
- `locktime`: A timestamp, not used for now

The message object that we will be sending over the network will also contain the `transaction_id`, which is defined as the hash over all leaf values of the transaction object, with all values being concatenated in lexicographic order and using the PoPStellar HashLen[8] hash function. Assuming the function is collision-resistant, the id will be unique for each transaction and we can, without ambiguity, refer to a transaction by mentioning its id.

In our Bitcoin-like odyssey, we also decided to hash and truncate the destination's public key in our outputs, this way we are reducing the message length. The specification for inputs signature was also inspired from the Bitcoin convention: we concatenate all `tx_out_hash` and `tx_out_index` from each inputs, to which we add the concatenated string of all `value`, `script.type` and `script.pubkey_hash` fields of each outputs, and we sign over this entire concatenated string.

Finally, the `value` field is an integer that is counted as miniLAO, with 10^8 miniLAO = 1 LAOCoin. The maximum supply is $2^{53} - 1$ miniLAO, which reflects the maximum value for which we are able to use native integer types in all of our sub-systems. We also enforce that, for any given transaction, the total sum of money across all inputs does not exceed the maximum supply. Also, the sum of the money in the outputs must be exactly equal to that in the inputs.

3.4.2 Coinbase Transaction Design

Also, there is the need to find a way for an organizer of a LAO to emit some LAOCoins to attendees. This is implemented as a special case of the digital cash transactions called a *coinbase transaction*.

As you can imagine, a coinbase transaction does not need any inputs as its goal is to create coins out of thin air. To ease its integration into the existing model, we designed a specific input format that indicates that a transaction is a coinbase transaction and that respects the initial specification.

Here we describe the input format for a coinbase transaction:

- `tx_out_hash`: an all-zero transaction id, which has to be encoded in base64url
- `tx_out_index`: 0
- `script`: The usual input script, except that the private key to sign should be the LAO organizer's

When receiving a coinbase transaction, each sub-system will then need to verify that the signer is the organizer of the LAO related to the received message and only then accept the coin issuance.

3.4.3 Specific Implementation Details

In our implementation, the back-ends remain stateless, this means that most logic and security checks will happen in the front-ends.

We also decided that when using an unspent transaction output with a given PoP token, every output available will be spent in the transaction. This means that when sending less than the entire balance available, the rest of the balance will be sent back to the PoP token.

3.4.4 Security Considerations

Security Model

To be considered secure, a cash system like the LAOCoin system must not allow an adversary to spend others' money, freeze or destroy others' money, or create money outside of the agreed-upon issuance process.

For the adversarial model, we support the organizer being a covert adversary — an adversary that will only do bad things if they can avoid being exposed as the culprit —, and all other attendees being active adversaries to each other. This particular model of covert adversary fits within the LAO framework since we can use accountability of the organizers to ensure good circumstances.

In terms of adversarial capabilities, We assume that adversaries cannot break our cryptographic schemes or compromise attendees' or organizers' devices remotely. It is beyond the scope of our work to design cryptographic primitives or discuss software security. Instead, we rely on commonly available existing solutions referenced throughout this document. We also assume that attendees are not being coerced or threatened into revealing or using their private keys. This is of course something that can happen in real life scenarios, but it cannot be mitigated in the general case with system design.

Since we will require that certain transactions are deemed invalid and discarded by the system, we should define where this policy should be enforced.

The front-end component of each participant is responsible for performing these checks. Since not every participant runs their own back-end, this ensures that they don't rely on another participant being honest and instead have their device perform these checks to protect them. Back-ends may perform some or all of these checks, but it is not strictly necessary.

Discussion

To ensure only the owner of the funds can control them, we rely on the security of the signature scheme. Since transactions are only considered valid if unlock scripts provide authentic signatures from the rightful owners, and since the signature scheme guarantees that signatures cannot be forged, no adversary can create a valid transaction that spends other people's funds. Signatures sign over the core characteristics of transaction inputs and outputs, so reusing signatures is not possible. Replay attacks are also ineffective since transaction outputs cannot be spent twice.

There is no message that an attacker could publish that would affect funds they do not control. Even the organizer does not have the power to seize funds in the current system. In the current model however, the organizer controls the pubsub channels. Hence, they could filter certain transactions out, effectively freezing funds. This should be addressed with the planned censorship resistance features described in Section 1.3.2, which defend against this as long as at least one witness is honest. Under the assumption that the organizer cannot filter the channels, the system would be secure.

What about funds that an attacker used to control? It is possible to sign two contradicting transactions that spend the same output. In the current architecture, this would simply result in one of the being clearly invalid since the pubsub channel enforces an order. But when offline payments are supported or when the channels become decentralized, care should be taken to evaluate the consequences for the security model. In particular, thanks to the UTXO model, double-spending can be attributed clearly to a specific PoP token, so it might make sense to use accountability in this case.

We prevent unauthorized creation of money by discarding invalid coinbase transactions, and by checking that the sum of input values of any transaction is equal to the sum of output values. We also guard against arithmetic tricks that could be used to craft transactions that bypass this check by strictly defining the range and overflow rules for transaction amounts in section 3.4.1. This will stop any attendee from unduly creating money since they do not have the keys allowed to do that. As for the organizer, any creation is attributable to them due to the non-repudiation property of the signature scheme, hence, if they do not follow the agreed upon rules, this will be apparent. Since the adversarial organizer is covert, they are not able to do that.

3.5 Future Work

3.5.1 Offline Payments

Having seen the current state of the project, we can now think of future implementation. The first following step will be to implement offline payments. Indeed the infrastructure will be already done for the back-end, the only thing that should be updated will be the front-end. The idea behind offline payment is that when you don't have any internet connection, the message is signed by the sender and scanned by the receiver. Then the first among those that sent it on the network will publicise the transaction on the network. There is no repudiation from the sender possible as he signed the message with its private key.

3.5.2 Censorship Resistance

Another step could be censorship resistance with the help of multi-server and witnessing. Currently the organiser has complete power. Indeed there is no way to control coin issuance, therefore the organizer can issue himself infinitely many coins. To have a more trustworthy system, there should be witnessing implemented to control organizer power.

3.5.3 Addresses

We also briefly discussed human-friendly references to PoP tokens. The current de facto practice across PoPStellar is to use base64url-encoded public keys. This results in a string of 44 characters that has mixed case, dashes and underscores, and always ends in an equal sign. This is difficult to say out loud, write down, compare, recognize, or remember.

One idea that could be tested within the digital cash project and outside of it is, once again, based on Bitcoin. In Bitcoin, addresses are encoded in bech32[12]. This produces a case insensitive string that avoids visually similar characters and has an integrated checksum to catch human errors. Addresses derived from a Bitcoin public key hash are 42 characters long. It is also easy to adopt since LAO public key hashes are the same length. Compared to the current base64 method, bech32 strings would have comparable length, but feature checksums and a much nicer character set. We should note that it is not possible to derive the true public key from the address since it is hashed, but if this limitation is acceptable, this could be beneficial.

Another idea could be to use a randomly-generated image with the PoP token as a seed. Indeed we are already using a similar implementation in the Chirp project where we use the Ethereum Blockies[5], which look like figure 3.2. We could imagine generalizing that to the whole project to help the users verifying if they are communicating with the desired person. Working with

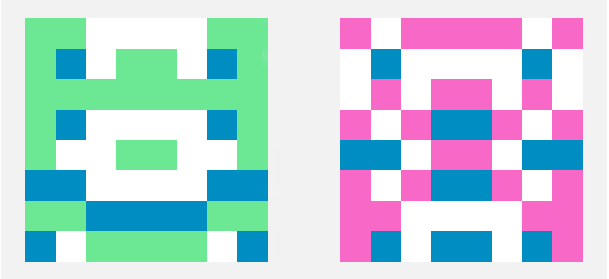


Figure 3.2: Blockies Example in Front-End 1

strings of arbitrary characters may be very error prone to communicate, two addresses may look more or less the same. Having these image may help to check by giving a color or a shape that may totally differ even if the key is mostly the same. It should be used as a verification tool.

3.5.4 Privacy Enhancements

In its current form, all transactions are revealed to all participants. This allows for users to validate these transactions and ensure no one is cheating, but as a side effect, users have to reveal information about their spending habits and balance. It might be desirable to prevent this leak and provide users more privacy for various reasons, including the risk for users to be targeted for robbery, fear of judgement based on spending habits, etc.

Fortunately, the existing cryptocurrency industry also tries to create schemes for private payments. One such scheme is RingCT[11] which is used by the popular Monero currency. It allows obfuscating the amounts, origins, and destinations of transactions, while still allowing participants to know that they are well-formed. This protocol works with a transaction structure that is similar to the one in Bitcoin and LAOCoin, with the difference that it isn't compatible with our current address scheme. Instead, users must publish two Ed25519 points as their payment address.[13, p. 6] They must be in plain: public key hashes are not possible.

Another option is to adopt the ZeroCash[2] protocol used by Zcash. It was originally designed to operate on top of Bitcoin, which suggests it is suitable for LAOCoin. Unlike RingCT, it requires using a *trusted setup*, which is an one-time process where a trusted party generates public parameters for the whole system. Ideally, the trusted party should exist through multi-party computation so that multiple parties can be involved. The system remains secure as long as one of these parties is honest[3]. In the LAO framework, it would be acceptable to run such a setup either solely on the organizer's machine, or with multi-party computation in collaboration with witnesses and/or attendees. Noether[11, p.6] also notes that in Zerocash, the power to create money comes from the trusted party, whereas in RingCT money

creation is independent from the privacy features and can be performed via proof-of-work. Since we do not use proof-of-work, this is not a concern for LAOCoin.

Chapter 4

Engineering a Production-Ready System

4.1 Introduction

The purpose of this sub-project is to bring the system closer to a state where it is said to be production ready meaning that it can be out in the field. There are several reasons why this is not the case already during development. It is totally different paradigm to have developers run software they built in a safe, isolated demo environment and to have it running in the field where many unpredictable things can happen. For instance, unfamiliar users will use the app in a different fashion than the developers if the user experience (UX) and user interface (UI) are not carefully designed.

The PoPStellar app for all of its existence has been very far from being production ready. Nevertheless we did not at all start from scratch because last semester's students had already work on the same sub-project. We will detail it in Section 4.2.

Of course bringing the whole system to actual production readiness was a wholly unrealistic objective and we therefore set our goal to make true improvements about the robustness and quality of the software. This of course is not straightforward as many factors need to be weigh in to make engineering decisions about the direction of this sub-project. This is detailed in Section 4.3.

4.2 Previous Work

Previous students did great work to improve the overall quality of the system. The documentation was greatly improved, as well as the consistency with which protocols are defined and implemented.

Moreover, at the start of the semester the Unit Test code coverage ranged from good to great :

- Be1 - Go : 70.3%
- Be2 - Scala : 42.2%
- Fe1 - Web : 53.5%
- Fe2 - Android : 52.4%

A CI/CD run of those unit tests with coverage feedback and SonarCloud static analysis were already in place. Finally, a great tool called *Karate* was incorporated which enables integration testing. Integration tests for LAO creation were already added for both back-ends and front-ends.

4.3 Strategy

Given the resource constraints, both in time and in number of people involved in this sub-project, to make true improvements to the system quality, we had to devise a strategy as to what we would do during the semester. We decided that we would work on four fronts. First we would be collecting and solving bugs that arise during weekly PoP parties which is akin to end to end manual testing. On top of that, we would provide integration tests to have guarantees as to how large blocks of our subsystems behave. Then, with the sole exception of the social media part, the user interface of both front-end were problematic. Not only did they not provide a nice experience to the user but were the source of avoidable problems. We therefore set ourselves to make improvements in that regard as well. Lastly, we wanted to increase coverage with unit-tests. Advantages are multiple including early detection of bugs and prevention of regression bugs.

Our rationale behind this strategy was to build on the previous semester's work. In particular, we wanted the project to truly benefit from the integration testing framework they set up to have good confidence about the quality of some of the more important features for this semester. Indeed, by lack of time almost no integration tests were present at the start of the semester, so we identified their addition as crucial and as a central part of our code quality strategy.

Moreover, efforts we already made last semester to have a UI/UX following best practices last semester for the social media feature. It was restricted to this sole feature by lack of resources.

4.3.1 PoP Parties

PoP parties take place during the weekly global meeting and are testing events where most if not all people present take part. They amount to

manual end-to-end tests. Each week we would test a mix of android and web front-ends with one of the back-ends, both if time permitted, alternating between back-ends each week. One of the people would be the organizer and others attendees. Most of the tests were connecting all people to a LAO and conducting roll-calls and elections. We took note of bugs and interface design flaw with significant adverse impact. As much as possible we would fix bugs before the next PoP party such that new bugs could be found. Though unfortunately sometimes due to the severity of the problem, it took several weeks to fix it. There were a lot of design problems ranking from benign to harmful. Given the limited resources, we had to decide on whether to handle them given the cost to fix (the complexity) and what risk the flaw incurred for our system.

We note that this method of testing is extremely costly in term of developers' time. PoP party duration would be between 20 to 40 minutes, which for 10 students, amount to 200 to 400 minutes each week, i.e from 3h20 to 6h40 which is considerable. Of course, automating it would thus be time sparing but several concerns make it difficult to fit for this project. First, it would not be trivial to make multiple front-ends and back-end work together over an internet connection in an automated fashion. This requires infrastructure, maintenance. This implies problems since most students start the project with no previous experience with anything remotely similar in term of scale. Getting into the project is already challenging enough without having complex end-to-end tests to handle.

4.3.2 Integration Tests

Integration testing is the practice to test components together and verify their compliance to the specifications[14]. This is the intermediary ground between testing components individually i.e. unit testing and the whole system i.e. end to end testing. Conceptually, what is done is that we consider each subsystem, be it a back-end or a front-end, as a monolithic block, a black box. We gain insurances about the compliance of the system by putting in action or messages in the sub-system and checking that its output is what we expect.

As previously stated, we use Karate to that end. As mentioned in Section 4.2, the framework was already set up in the project by previous students. It allows us to test each subsystem separately by mocking either the front-end or back-end and then testing the respective counterpart by sending mocked messages. This allows us to craft mock network messages which makes it possible to test many cases that were completely untouched before such as invalid ids for example. Considering that the PoP project is at its 4^{th} iteration there are now many features. Therefore we had to make once again decision as to what should be done and what should be left for future work.

Provided that LAO creation and all roll-call features are used by each and every other feature of our system, we decided that without any doubt they should have integration tests by the end of the semester. Moreover we also considered that election features tests were an appropriate addition. Indeed they had the particularity of being already implemented at the beginning of the semester but being extended during the semester by a sub-project.

4.3.3 Unit Testing

Thanks to the advice and impulse of Mr. Borsò, the code coverage minimum for new code was raised to 55% at the beginning of the semester. We were very fortunate with the receptiveness and testing ethics of other students because the vast majority of the PRs had coverage on new code over 65% even regularly hitting over 80%. We decided that having a coverage on all code of at least 65% was a realistic goal for the systems that at the time were below that.

At the beginning of May, we decided to raise the coverage limits to 60% on new code to be merged. We made the decision because out of the open PRs, none was affected so we would not disrupt the immediate development flow. Furthermore, students had time to get familiar with their sub-systems and projects and the vast majority of them had already merged at least one PR. This decision was to ensure that efforts to raise the coverage would not be too affected by less tested new features. Of course, exceptions were possible and have been granted by the teaching staff when it was justified.

Unfortunately we have not reached our coverage goal at the time of the writing of the report but we fully expect to achieve the goal of 65% by the end of the project. The current code coverage is as follow :

- Be1 - Go : 78.1%
- Be2 - Scala : 56.0%
- Fe1 - Web : 76.3%
- Fe2 - Android : 62.2%

4.3.4 UI/UX Refactoring

Most of the features developed on both front-ends were done so with little regard for UI/UX principles. Moreover it let the user bring the application in a state that can cause errors whose source would be hard to understand.

4.4 Execution

4.4.1 Integration Tests

Front-End

Last semester the framework was set up for both web and android and a simple test was implemented, probably as a proof of concept.

Beyond a Simple Use Case The first challenge was to decide on a way to appropriately to add mocked behaviour from the back-end. Indeed by reacting accordingly based only on messages sent by our front-end under test, we would end up creating a mock back-end of ever increasing complexity - i.e. a back-end. This is obviously a very bad idea as this would not scale, would be error prone and would also offer lesser guarantees about our front-end under tests. To remedy that we extended the *Reply Producer* set up last semester. During each test we specify what behaviour from the mock back-end we expect for example LAO creation. This indicates to the mock back-end which behaviour to adopt. This is a major improvement to the aforementioned as we only need to care about the correctness of a few lines of code for each feature instead of the complexity of a mock ever closer to a real back-end.

Current State Unfortunately, at the time of writing this report, only the previously mentioned mocked behaviour pattern and a basic test for roll call creation from the organizer are implemented. Nevertheless we fully expect - as a must have - to have a set of tests covering roll calls and election features by the end of the project.

Back-End

Last semester we had some tests but the environment for back-end integration testing was not fully complete. Thus we needed to apply changes to it's implementation. Karate already has it's own implementation of web-sockets that was the perfect fit last semester to test `lao#create` message since we only needed a valid or invalid sample of such a message to be sent to the back-end and wait for it's reply. Then all we had to do is to test whether the answer was the expected one. However, to test more complex features as the creation of roll calls we needed the ability to send multiple messages to the back-end and receive all the answers for each of the messages. Unfortunately this was not offered by the Karate API. Last semester, some work was done to create a multi message web socket that would be able to store all the answers in a buffer. To get those efforts to a working state, only a few tweaks were needed. With this we had the tool for creating a mock front-end that could send multiple messages that we crafted and receive responses for each of

them. For each feature, we test the handling of valid messages and invalid messages. The latter are detailed in their respective entries.

Simple Scenarios To avoid complex issues with interdependence of tests, we restart the back-end at the beginning of each test. Therefore in order to test LAO related features, we always first need to send a valid `lao#create` message to the back-end, followed by a subscribe and a catch up message. To avoid code duplication, we introduce a black box abstraction that will put the back-end in a desired state and test only the feature that should be tested on top of it. Those we call "Simple Scenarios". For each feature, once tested we add it to the simple scenarios. For example once LAO creation is tested, we add a Simple Scenario for LAO creation.

Roll Call The roll call implementations are tested for different properties they must have. First, valid ones should be accepted without unexpected behaviour in the form of additional messages sent. Moreover for different kind of invalid messages, back-ends must respond with the appropriate error code and description.

`roll_call#create` The invalid fields we tested are: having a non organizer as sender, empty name for the roll call, sending a valid request on the root channel, having proposed start later than the proposed end, negative creation time, a creation time before the proposed start and invalid roll call ids.

`roll_call#open` The invalid messages we tested are: invalid roll call update id, sending a valid roll call open without creating a roll call before but having a LAO setup.

`roll_call#close` As for the invalid messages we tested the following conditions: a message containing an invalid roll call close id, valid roll call close request but without a previously opened roll call.

Election Having added a complete and valid roll call process to Simple Scenarios, we were ready to test election features.

`election#setup` The invalid conditions we tested are: invalid election ids, invalid question ids, an election containing a question with empty ballot options and an election containing some unsupported voting method.

`election#open` Invalid messages tested: sending an election#open before an election#setup, sending an election#open that has opening time before election#setup creation time, non-organizer sending a legitimate election#open, message with wrong election_id

`election#cast_vote` The election protocol was slightly changed during the semester so testing a cast vote needed to adapt to certain changes. At first,

we could cast a vote within time bounds set by election#setup. We therefore tested if a cast_vote cast before start time or after end time was properly handled as invalid. However by the introduction of election_open, start and end fields where no longer required and we only needed to cast a vote after a valid election#open message. The invalid messages we tested: sending a valid cast_vote on a LAO channel, sending a cast_vote on a non-existent election, casting a vote with a wrong vote_id, non-attendee casting a vote.

`election#end` The invalid election_end messages as well: wrong election_id, election ends containing different invalid registered_votes.

With this message we conclude the testing phase of the open election features for the back-end.

All the invalid messages contain mostly invalid message fields so we check if the error code for these messages is -4, and a few messages that have a non-organizer as sender are considered as messages with access denied and should contain an error code -5.

Self Contained Tests

We initially followed how the tests where written last semester and adapted them to our needs. The recurring structure was 1) putting the back-end in a certain state (that we modularized by using Simple Scenarios), 2) reading a prepared file that contains the base64 representation of some message data with all the fields filled in manually, 3) sending this message with the mock front-end and finally 4) checking if the response received is the correct one. This approach could be improved. It raised a lot of questions on the programmer's side as to what is the data sent. Manually filling base64 data has many flaws, such as poor maintainability, unreadability, inflexibility and it is time consuming to debug. For example to reuse a message but adapt one field, we would have to manually decode the data to see message data fields, change it, encode it once again to base64 and place it in a file. This is a very tedious approach. This approach would also not scale to a bigger project. Indeed, the person writing the test should be able to use utility functions to obtain value that are the results of complex computations such as hashes. Otherwise each tester has to have a perfect understanding of how each and every field is computed. It is a waste of time and is thus costly. In order to address this problem, we decided to add one layer of abstraction between karate tests and the functionality. In order to achieve this, we decided that karate tests should be self contained, meaning that someone who did not write a karate test before should be able to write one easily without needing to read a ton of exterior files and documentation. Rather it should be possible to construct the data of a message and fill in the fields without needing to know how to compute them. We therefore added utility functions that form

an API for the computation of every field. This way of writing tests makes them more maintainable and easier to read.

Nonetheless, we did find some slight limitations in the way we implemented this abstraction. We therefore extended our API with standard valid non-computed values for example name or specified time that are user input. We added standard valid and invalid values for each field so that they do not have to be written by the programmer for each test.

4.4.2 UI

Front-End 1 - Web

Given the bad user experience (UX) the user interface (UI) of the previous semester (see Figures A.1 to A.15) provided, we decided that we must do something about this.

First and foremost, it happened numerous times during PoP parties that roll calls did not work because somebody did not setup their wallet. And this is totally understandable, the UI never told the user that they are required to first setup their wallet. The wallet setup screen was not even the initial screen of the user interface. Moreover the screen offered two options, one for generating a new seed for the wallet and one for entering an old seed for restoring the wallet.

In the new UI these three screens were condensed down to two: One welcoming the user, showing them information about the application and some newly generated seed. If they use the application for the first time, they can simply press one button and can start using the application without ever worrying again about setting up the wallet.

If they want to restore the wallet using a previous seed, they can do so by pressing the corresponding button in the new UI. This new UI forces the user to setup the wallet and has a similar setup to standard login / register screens with a prioritization for new users since they do not know the application. A side-by-side comparison can be found in the appendix in Figure A.1c.

Next, throughout the whole application, centered texts were replaced by ones aligned to the left of the screen. Moreover all screens were wrapped in a common component to ensure every screen has the same padding to all sides and allows scrolling if there is too much content (visible in all Figures A.1c to A.15b).

Input fields were standardized throughout the application and we now properly use labels and placeholders (see Figures A.7b, A.10b, A.11b, A.12b.

A consistent color scheme was defined and applied to the whole application: All buttons have the accent color blue (see Figures A.6b, A.12b, A.14b) if

they can be clicked and gray if they are disabled (see Figures A.10b, A.11b, A.12b. The navigation bar was switched from a top to a more standard bottom navigation bar and now also uses this same accent color (see Figures A.10b, A.11b, A.12b). The accent color is also used as the background for the introductory screens for setting up a wallet (see Figures A.1c, A.2b). This differentiates these two setup screens from the rest of the application. In these two screens, the buttons and input fields have the accent color as their background and a white border surrounding them.

Spacing was standardized by defining a basic spacing unit and then using different multiples for all spacing throughout the screens. A general list style was defined and used throughout the application making many views look similar and *boring* which is exactly what a good user experience should partially result in: The user should not be required to think too much when using the application (See Figures A.5b, A.8b, A.14b, A.15b).

The many buttons that were previously scattered throughout many screens were collected in an *action sheet* menu (see Figure A.9b) that shows up when pressing the either the three horizontal dots in top right of the navigation bar (see Figures A.3b, A.14b) or the pen on a paper icon on the event screen that is often used to denote *create* or *add* (see Figure A.8).

Modals were also standardized by having the same spacing to the side, a modal header and a button to close it (see Figures A.6b, A.6c).

With the changes to the user interface, we would expect an iOS user to feel right at home since the styling of the UI elements is in a large part inspired by iOS. Given that front-end 2 already provides a native android user experience, it was a natural decision to make front-end 1 look like an iOS application. That said, `react-native` supports platform-dependent styling which means it is totally possible to extend the UI further in the future and make Android users feel as welcome as iOS users.

Front-End 2 - Android

Having noticed during a PoP party that we had to heavily insist that people initialize their wallet before participating in a roll call, we decided that it should be impossible for the user to access a roll call before said initialization. We further considered that all LAO based feature needs a valid roll call token. Therefore, we decided to extend the restriction on users who have no uninitialised wallet. They now can only access the wallet and the settings.

The second aspect covered was navigation. At the start, there were top buttons, aesthetically unpleasing, and they lacked robustness to screen size, with buttons easily only displaying part of text. We replaced them with a bottom navigation bar (see Figure A.16).

The work with navigation continued on the event lists. The same buttons with the same downsides were used (see Figure A.16). The show/hide properties button was replaced with a QR code icon next to the title that expand a layout to show the LAO QR code (see Figure A.19). Navigation slots are precious, especially with the addition of digital cash, and this allowed to free one.

Though witnessing is not supported system wide, some previous implementations exist. Though it was badly designed as the witness list and witness addition were deeply buried within the event list view screen (see Figure A.20a). Moreover the messages for witnesses to sign were in a different view thus giving a very counter-intuitive behaviour (see Figure A.21). We refactored by creating a fragment for witnessing with a top tab for easy and intuitive navigation between first the addition and display of witnesses, and second the witness messages to sign. By lack of time, and since this part is low priority for the project at this state, the message list for signing is not refactored at the time of writing of the report and suffer from many flaws common of the previous UI/UX. It may be by the final presentation though it is considered a stretch goal once again because of low priority. See a whole side by side comparison with Figures A.20 and A.21.

The third aspect was the list of elements displayed such as LAOs or events, be it roll calls or elections. The android widget used, ListView, is old (since API 1) and is considered legacy on Android Studio. We therefore replaced it with RecyclerView because per Android developers[1] is a more modern and flexible widget. Moreover, we updated the element of list to be displayed in a more airy fashion, the previous cranked up version being unpleasant to select (see FigureA.16).

A subsequent major problem was that the action and management buttons of an event were displayed directly on each event entries. This lead to several problems:

- There was a lot of duplication of identical buttons on one of the main screens of the app
- The layout for event list elements was badly designed thus making only half a button out of two displayed on a ”normal” phone screen size
- The list entry was displaying all information such as start and end date, and name on top of the previously mentioned buttons. This resulted in a very crowded display that is contrary to UX/UI principles.
- The logic behind was extremely complex. Partly because of the use of ListView, but also because having a list of views with dynamically changed display is intrinsically complex and error prone. This resulted in multiple bugs such as non updated content and homogeneous change of display when only one element should have been affected.

Fortunately addressing exhaustively those issues proved to be convenient and efficient. We removed the buttons from the list elements and instead added a forward arrow indicating to users they may enter each event. There, both buttons are displayed and event information as well. The event detail can be seen in FiguresA.22 and A.23

4.5 Findings

Mostly during karate testing we found there where some badly implemented parts of features, whenever we found something we proceed in the following way: 1) try to reproduce the error to see if it is deterministic and easily reproducible, 2) note the error down, than try to identify to which subproject it can be related to, 3) notify the subsystem that we found a bug or inconsistency, 4) open an issue with detailed explanation of how it can be reproduced or describe what the problem is, and 5) try to solve it ourselves if possible or leave it to the respective team to deal with it otherwise.

4.5.1 Idempotency

Idempotency is the property of having the same behaviour for the same request no matter how many times it is submitted. Valid messages should always receive a valid response, not matter how many times they are submitted. Conversely, error messages are split into temporary and permanent categories. Permanent errors must be responded consistently on each identical request. For temporary errors response, they may become valid (or permanent error) if the cause of error is fixed. Permanent errors can be invalid message field for example. Temporary may be a lack of memory on the server that will be later fixed.

Having idempotency brings us a lot of benefits, making the system very robust and prevents some exploits that can be leveraged by an attacker. For instance, if the back-end does not consider a permanently bogus message as faulty, a form of replay attack would allow the attacker to eventually obtain a valid response to a buggy message which can be a major vulnerability. Nonetheless, achieving idempotency is not an easy task and requires work on all subsystems, especially the back-ends. Moreover it should be carefully tested. As an intermediary step, we decided that permanent errors should always result in an error message from the back-end. We fixed this by not storing the error messages on the server side so that even if a replay occurs the server will have to recheck the validity of the message on every request. The check will therefore always fail and the same error message will be sent as a response from the back-end.

4.5.2 Bugs

In this paragraph we lay out the bugs that PoP parties and integration testing detected. We omit unit testing because those bugs were solved before merging into the codebase. Due to the sheer quantity of bugs, many of them benign, we only share a relevant subset here. If not indicated otherwise, all those bug were fixed.

Fe1 - Web

Many smaller issues were fixed throughout the semester such as vague protocol specifications or incorrect implementation and of course UI bugs. As we consider none of those major, we will not go into detail into them

Fe2 - Android

The application we received at the start of the semester had several major bugs that rendered it unusable. The first and most severe one, was the impossibility to connect to another LAO.

Another was a buggy implementation of error utility functions that caused a crash whenever an error was to be logged and displayed to the user, as well as another when a generic error was handled.

Finally, handling of error message from the back-ends was inadequate which resulted in crashes in many different situations.

Back-ends

The addition of Karate tests lead us to the discovery of major bugs that completely undermined the security properties that our system is supposed to hold. These bugs were mostly due to the lack of checks of the validity of various id fields by the back-end. Some invalid requests were treated as valid which broke the integrity and authenticity properties.

The most serious vulnerability on both back-ends was that the signature of sender on data sent and the message id were never checked. The first is the most severe since it breaks both integrity and authenticity. Indeed the signature allows the sender to prove that the data field was filled with the sender's private key. The message id relies on the validity of the signature. It provides a uniqueness property to each message. Breaking it allows replay attacks.

Furthermore, we discovered some bugs that were not common for both back-ends but could still be the root for certain vulnerabilities. For starters, the Go back-end accepted that a non-organizer can create an LAO, create, open and close a roll call, setup, open and end an election. This defeats the

purpose of the organizer entirely since if everyone is free to do whatever in the system.

Next, we discovered that the timestamp present in election messages was sometimes taken for granted. It was never checked whether the election setup creation time is before the election open time.

We also discovered that the registered votes field was not checked for the election end message. it is a hash over all received valid votes by the end of the election. Not checking them could allow an attacker to finish an election earlier and have a biased outcome.

Finally there were a few smaller bugs like not checking the whether the roll call name is empty, sending an error message twice whenever the invalid message is sent on the root channel, some unit tests not being exhaustive, etc.

Like for Go, we had some bugs that were only present in the scala back-end. The major scala specific issue was allowing roll calls to open before they were created, allowing roll calls to be closed before they open. Moreover it was possible to close an already closed Roll Call and to open an already opened roll call. These bugs made us realize that the server lacked some form of roll call state which would allow to identify relationships between the messages more easily.

Some ids were also not checked properly like vote id and question id for election messages, which broke the integrity property as already explained before.

Scala also did not check that only the organizer should be allowed to perform certain actions such as creating, opening or closing roll calls which once again would defeat the purpose of the organizer. And some bugs with lesser impacts were also present like allowing empty LAO names.

4.6 Future Work

By the time of writing the report, not all goals that we had set for ourselves are met, but some remain in progress and we will give our best to achieve them in the upcoming weeks.

4.6.1 Karate

Currently the error codes sent by the 2 back-ends are more or less the same for all the invalid requests sent to them, however the description of the error is not the same, they remain similar but an attentive user could potentially distinguish on which back-end it is connected, which leaks some information and can cause some issues if one back-end is ore vulnerable to attacks.

Another crucial part of Karate tests is to ensure that the system remains correct even when adding new features or improving old ones, one way to achieve this is to run karate tests on the CI, this way feature implementers can check that they did not break preexisting functionalities.

Line coverage for Karate tests should be added.

Integration tests should be extended to all messages. The untreated ones are: LAO greet, LAO state, LAO update, Secret Ballot cast#vote message and the whole social media part.

The test set Digital cash features should be completed.

4.6.2 Network Resilience

Almost all functionalities of our systems need resilient communication in order to function. The PoP app is designed to withstand internet connection losses and reordering but no true testing has be done in that regard. We tried using a basic script early on that randomly rearranged and dropped packets. The app still worked the same and we soon realised that we were effectively testing TCP (spoiler alert: our very basic script did not find any flaw in TCP surprisingly). A possible idea would be to implement a pass through server for testing. It would receive messages first and subsequently drop and reorder messages before delivering them to the actual server. This is quite heavy a testing tool to develop.

Chapter 5

Subsytems

5.1 Back-End 1 - Go

In this section, the architecture and changes specific to the Go back-end are described. A detailed description of the subsystem specific implementation of the different features of the E-Voting (Chapter 2) and Digital Cash (Chapter 3) projects are largely omitted as the specification is already described in the respective chapters. Interesting implementation parts of specific features are highlighted.

We will first begin with a general description of the implementation of Back-End 1 , then we will focus on the changes done during the semester.

5.1.1 System Architecture

The back-end Go will accept web-sockets connections from front-ends and messages will be exchanged through these connections. The incoming messages are handled in the hub of the server which will then redirect them to their corresponding channel. Most of the features are in their own separate channel that handles the processing of their respective messages.

5.1.2 E-Voting

At the beginning of the semester, the Go back-end already supported the whole election pipeline, including the election setup, the ability to cast votes, to end elections and to compute and broadcast the results. All of this means that at the start of the semester we could run open ballot elections.

Addition to the election pipeline

During the semester we made a few additions to the election pipeline.

By symmetry to the `election#end` message, we added an `election#open` message to open the election. We allowed it to open elections regardless of start and end times set during setup. With the new registry it was a matter of creating the new message in `election_open.go`, verifying its contents in `election/verification.go` and change the state of the election from closed to open in `election/mod.go`.

To implement secret ballot elections we need a key pair that is unique to each election. The key pair is created at the setup of the election and then a message `election#key` is created by the server and broadcasted to the subscribers of the election channel as well as stored in the inbox for the new subscribers to find. The new message is defined in `election_key.go`, new fields were added to the `election.Channel` struct for the keys and the type of election and a function `createAndSendElectionKey()` was defined to perform the creation and broadcasting of the new message.

Encryption and Decryption

Before being able to decrypt the votes coming from the front-ends, we had to make changes to the `election#cast_vote` message in `vote_cast_vote.go` to comply with the new specifications. `Vote.Vote` was changed from `[]int` to `interface{}` so that we can accept votes both in `int` and `string` format. Due to this change we had to redefine the umarshalling function to ensure that `Vote.Vote` either contained an `int` or a `string`. The verification of the `election#cast_vote` messages was changed in `election/verification.go` for the same reason.

Given the above it was possible to implement the decryption algorithm in the function `decryptVote()` in `election/mod.go` based of the specification. Moreover the the encryption algorithm was also implemented so that we could properly test the decryption of the votes.

Other changes

To verify the authenticity of messages sent by the organizer's server or witnesses' servers, we added a new message called `lao#greet` that is sent when a LAO is created. The creation, broadcasting and inboxing of the new message is handled the same way as for the `election#key` message from before, this time in the function `createAndSendLAOGreet()`.

5.1.3 Digital Cash

At the beginning of the semester, as it was a new project, we had a lot of discussion concerning how to implement this huge JSON Schema in Go . Even though it contains multiple similar structures with only minor

differences, we decided to create a separate struct for each of them. We have separated them in order to make the code easier to understand.

Implementation of the channel

The implementation of the new channel was reached in multiple steps. We first implemented the generic channel interface which contains the `Subscribe`, `Unsubscribe`, `Publish`, `Catchup` and `Broadcast` method. Then we implemented the channel creation and the registry to handle the messages. Currently there is only one message for the digital cash project: `coin#post_transaction`, but for the sake of coherence, we stuck to the same structure used everywhere else. Finally we implemented the processing of the message which is achieved by first unmarshalling the received message, checking that it fits in the message structure, then verifying its message id, storing it into the inbox and finally broadcasting it to all the clients subscribed to the channel.

Testing the implementation

The final part concerning the Go back-end implementation of digital cash was the testing. We based our test on the protocol example messages that were created for this purpose. We reached a coverage of around 80%.

5.1.4 Other Changes

Implementation of Registry

At the beginning of the semester, we started with some channel refactoring. We improved the implementation by completing a registry to handle the different message in each channel. That registry allows us to remove multiple switch statements that were calling different actions according to the message. This helped to reduce the complexity of the overall codebase and significantly improved readability. It reduces the code duplication associated with the aforementioned switch statements as well.

Checking the Organizer Key

Despite the obligation to provide a public key at the command line when starting a go server, it was never used in any way. The public key of the organizer's front-end is sent with the LAO create message. In a first step, we added a check to ensure the public key contained in the LAO create message equals the key specified on the server's start and returned an error when it did not match. Subsequently, we decided to make the provision of a public key on the command line an option with the back-end only checking LAO create messages' sender when the option was used. This serves two purposes.

- The owner of the back-end should be able to decide if they want their server to accept only LAOs created by them or also LAOs created by others. The second case could arise in many occasions such as EPFL kindly providing a server for students to create LAOs for example. Of course, we would have to deal with spam possibly resulting in DoS but this is a concern for another time and another report or paper.
- The current implementation of the scala back-end does not ask a public key on start and therefore never checks the authenticity of the LAO create message. While having supplementary feature is okay, having divergent behaviour should be avoided as it contradicts the N-Version programming principle of this project (with N=2 in our case).

Refactoring and Testing

In addition to the refactoring done on the registries, further work was done on the structure of the channels. Due to the fact that multiple people worked on multiple channels they ended up being structured differently despite all of them being similar in many ways. We defined a structure and organized the channels accordingly. This change makes channels more readable and easier to understand.

The hub folder was also refactored, some comments were corrected and parts of the code that did not use the dedicated function for hashing now do. Also other small bugs found during the PoP parties were fixed.

A database feature was also in the process of being implemented before the semester, but it was never used and more complicated that necessary. With the refactoring of the channels, the feature was rendered useless and could be implemented in a more efficient way if we had to do it again. The feature was also not tested and thus we decided to remove it from the Go codebase.

At the start of the semester, the Go subsystem was the lead in test coverage across all subsystems. We increased the coverage with each of our PRs by testing features and new functions as thoroughly as we could. We encountered some problems with some parts of the codebase that weren't tested enough or at all. And that is why some of the future work will be spent improving the test coverage even more.

Error handling

Error propagation was not consistent across the back-end, the server did not always return the correct error code when an invalid request was received. To fix this we needed to take a step back and observe what happens when a bad request arrived. We found that in some cases an error was returned followed immediately by a valid response, which could be observed only with karate tests when checking if some additional messages were sent by the server. The

bug was not easily identifiable but easy to fix, it was a simple missing return. This problem could have been the root of a silent vulnerability since the first response was the one expected.

Furthermore, we tried to enforce a certain error core on a high level to be consistent with the other back-end. The system is easier to use if we can categorize the errors based on their type. The error categories were already defined at the start of the semester but the rule was not respected everywhere and it was not uncommon for the back-end to send the same error code for every error. We refactored the error handling so that all possible custom error codes can be returned and give more indication on the nature of the error.

More checks

The back-end had some checks missing which would allow some invalid messages to be accepted, and thanks to karate tests we identified a lot of checks that should be added as not to create more vulnerabilities. Like mentioned in **bugs** paragraph , we found some bugs which allowed us to locate where the lack of checks was present, and where we needed to add them and create some unit tests on top of the karate tests that helped identified the flaw and prevent it from occurring again.
We introduced the most important checks for signatures and for message id. The remaining fixes had not major impact or are self explanatory with the bug description since they were edge cases of some features, nonetheless we wanted to secure the application as much as possible so the fixes were applied and correspond to one of the test cases of the overall karate tests that we implemented.

5.1.5 Future Work

E-Voting

For the e-voting sub-project, the next steps will be to implement server-to-server communication, finalize and thoroughly test the witnessing feature, implementing the Neff-shuffling and finally partial decryption so that the secret ballot elections are complete.

Digital Cash

Concerning the Digital Cash sub-project, the next step will be the implementation of witnessing and consensus into the project making the system more trustworthy.

Other work

Our test coverage is already good but it could always be better and some parts of the codebase are still untested from previous semesters. So improving the test coverage is also one of the key points of future work.

The Go codebase also has the highest code duplication percentage mainly because the channels, after refactoring the registry, are very similar. Reducing these duplications is also something that the Go sub-system should strive for.

5.2 Back-End 2 - Scala

In this section, the architecture and changes specific to the Scala back-end are described. A description of the subsystem specific implementation of the features of the E-Voting (Chapter 2) and Digital Cash (Chapter 3) are briefly described. We focus on interesting implementation that are specific to the Scala subsystem. We will first begin with the interesting features of the e-voting part, notably the encryption and decryption process, and then we will focus on the improvements and code consolidation of the initial code.

5.2.1 E-Voting

Initial state

Initially, the state of the Back-End 2 only supported the `election#setup` message. Therefore, we could only display an election. The verification of the `SetupElection` messages was implemented in the validators.

Support for the other messages was not implemented.

Modified components

The main work on the Back-End 2 was to support `election#cast_vote` and `election#end` messages, and further, to send the result `election#result` to the front-ends.

Computing the results of an election requires a lot of interactions with the database. That is why we created a new helper class *ElectionChannel* containing shared functions to avoid code duplication. The class for instance contains the function *extractMessage* that helps extracting all messages of given types. Each channel can this way obtain the set of messages it is concerned with.

The steps to compute the results of an election can be summarized as:

1. Extract election information, such as question ids and available ballot options for each question. For this purpose we created the function *getSetupMessage* in *ElectionChannel.scala*.
2. Only keep the last `election#cast_vote` message per attendee. This is what the *getLastVotes* function does in *ElectionChannel.scala*.
3. For each ballot of each question, we count the number of cast votes with the function *createElectionQuestionResults* in *ElectionHandler.scala*.
4. Create the *ElectionQuestionResult* objects from the previous results also done in *createElectionQuestionResults* in *ElectionHandler.scala*.
5. Create the `election#result` data, convert it to a message and broadcast it in the function *handleEndElection*.

To support the `greet#lao` described in Chapter 2, we needed to find a way to store the address of the server. In the Back-End 2 , the address was stored in a simple value in the *Server.scala* file. The simplest way to get access to this value was to use the same value in the handlers of the LAO, and to store it in the object *LaoData*. Therefore, we added a new field to it, and updated the *updateWith* function. For the future it would be wise to find a better way to obtain access to the address of the server.

Encryption and Decryption

We included the whole DEDIS cothority library project in the Scala side as a dependency, because we decided to use the *Kyber* library and the cryptographic primitives it provides since there exist compatible implementations for all four subsystems.

We implemented a new class *KeyPair* containing a *PublicKey* and a *PrivateKey*. The two classes *PublicKey* and *PrivateKey* implement a encrypt and decrypt function, respectively. With a *KeyPair* object it is also possible to access the encrypt and decrypt functions.

After generating the keypairs and sending the `election#key` message to the front-end, we had to find a way to store the private key safely, so that we can use it again for the decryption. We decided to create a new object *ElectionData*, which is stored in a new private channel. The object stores simply the id of the election concerned and the corresponding private key of the pair of keys generated. We decided to create a completely new channel so that no Client gets access to the private key.

To store the *ElectionData* and, successively, the private key in the database, we had to implement two functions in the *DbActor.scala*, which are the following:

1. *createElectionData* that creates the object and stores the private key. It is only used in the `election#setup` message, when we need to send the

`election#key` message. It creates a new channel which has the path `root/private/electionId`.

2. *readElectionData* that is used when we compute the `election#result` message with encrypted votes. We retrieve the private key, and can decrypt the votes to compute the result.

The decryption of `election#cast_vote` messages become pretty easy once all this is done. We first changed the signature of the class *VoteElection* to take an `Option[Either[Int, Base64Data]]`, for open_ballot or secret_ballot versions. We obviously needed to modify the *MessageDataProtocol* to support those changes.

Verification

Currently, the validators of the `Election` are fully complete. We check thoroughly all the attributes of each messageData when receiving it from the front-end. We point out particularly the checks for the vote ids, question ids and the election id. Moreover to verify the validity of `election#open` and `election#end` messages we needed to use the *extractMessage* function to check its current state.

The *extractMessage* function is also very useful to compute the `registered_votes` field of the `EndElection` message. We need to retrieve all the votes of the previous `election#cast_vote` messages. We want to ensure the integrity of the message. We created a function *compareResults*, in the file *ElectionValidator.scala*, that computes the expected hash accordingly to the protocol and compares it with the received hash in the `registered_votes` field of the `election#end` message.

Apart from that we needed to apply also some refactoring, for instance some roll call messages were accepted when they shouldn't have been. By adding some roll call state and refactoring the validator made the server more aware of the relationships between different messages and thus less error prone.

Furthermore we added basic checks to complete the full verification of the message including checking the staleness of the timestamp of the messages as well as the signature and message id which as mentioned in `Bugs` introduced a massive vulnerability in the system.

5.2.2 Digital Cash

In the digital cash subsystem, back-ends are less involved than front-ends. They need to relay published transactions on the appropriate channel and optionally validate them. The only message type is `postTransaction`. The first step was to make sure it could be serialized and deserialized from and to JSON. This also required creating case classes for all the components of a

transaction. Since transaction amounts are specified as values of the range $[0; 2^{53}[$, we represent them with `scala.Long`, which is a built-in signed 64-bit integer type.

Additionally, we implemented many validation steps including checking the transaction hash, verifying the signatures and checking transactions for overflows. We also made sure this code was covered by tests.

5.2.3 Code consolidation

In this section, we focus on bugs found and fixed during this semester. We also describe new features added and explain why they are relevant.

Features and Fixes

As a first task, we needed to fix a JSON schema bug. The server disconnected when receiving a wrongly formatted JSON message. We modified it to gracefully handle the exception and reply with an error message instead of simply disconnecting.

During the semester, we particularly focused on the fixes in the `Rollcall` implementation. We needed it to fully work, since the E-voting part depended on the correct implementation of the `RollCall`. Here are the main fixes:

1. First, the update of the attendees in the `LaoData` was not correctly done. We modified the function *handleCloseRollCall* consequently, by adding an interaction function with the database by using the *WriteLaoData* function.
2. We needed to modify the handlers such that we cannot open and close a `RollCall` message when it was already opened or closed, respectively.
3. The creation of the same `RollCall` several times was also possible. We decided to create a dedicated channel for each `create#rollcall` message. If the channel already exists in the database, this means that the `RollCall` was already created and the back-end then returns an error indicating this.
4. Initially, only basic verifications were made in the function validators of the *RollCallValidators.scala* file. We focused on checking the ids contained in the `opens` and `closes` fields. Since those ids are linked to previous `RollCall` messages, we needed to retrieve or store the previous `open#rollcall`, `reopen#rollcall` or `close#rollcall`. A first idea was to use the *extractMessage* function from *ElectionChannel.scala* but this turned out to be a bad idea. All `RollCall` messages are stored in the main channel, so by trying to filter for these messages, all messages had to be checked. Therefore, we created a new object `RollCallData` in a

new channel which has the path `root/rollcall/laoId`. `RollCallData` has two attributes, the state of the previous message (i.e. CREATE, OPEN, REOPEN or CLOSE), and its id (the `update_id` field, the roll call ids change during the lifetime of a roll call) of the latest message. We also implemented two new functions in the *DbActor.scala*, respectively *writeRollCallData* and *readRollCallData*. For each message, we use the *writeRollCallData*, which updates the state and the `update_id` field to match the new message. The *readRollCallData* is used in the validators of the Finally, the *readRollCallData* is used in the`RollCall`. In fact, it helps checking the validity of the opens and closed ids, by retrieving the `RollCallData` with the updated fields. The consecutive updates were necessary, since we can close and open/reopen a roll call several times. For instance, a roll call can only be opened after a close or a creation action, and can be closed only after a open/reopen action.

5. The function *updateWith* in *LaoData.scala* was fixed. This function updates the corresponding fields after the `lao#create` and `rollcall#close` messages. After a `rollcall#close` message, the list of attendees did not include the PoP token of the LAO organizer. It threw an exception when a second `rollcall#close` message was sent after reopening the roll call.

Currently, the Scala back-end can handle the `witness#message` messages. To do so, we needed to implement the handler and validator of the object `WitnessMessage`. We also needed to implement the function *AddWitnessMessage* in the file *DbActor.scala*. This function adds the `witnessSignaturePair` to the message we want to witness. It takes the message id of the concerned message and the signature to add. Of course, those two fields need to be correct and consistent. This is checked in the validator.

Tests and coverage

We improved the coverage of the Back-End 2 back-end by adding several tests to the `Election` and `RollCall`. In fact, initially, the tests for the Election were only testing the `election#setup` messages. By implementing the handlers and validators of the entire `Election`, we added unit tests that were verifying the integrity of the messages, and also the corner cases. The big part concerned to test the handlers, notably the computation of the `election#result` message. Therefore, a lot of tests concerning the *handleEndElection* method were added. To do so, we used several `Mocked Databases` that reproduce the behavior we want to test.

We also added tests to the *RollCallHandlerSuite.scala* file, which test the `rollcall#open`, `rollcall#reopen` and `rollcall#close` messages. Previously, only the `rollcall#create` messages was tested. We also created a whole new file *RollCallValidatorSuite.scala* which tests the validators of the

`RollCall`. Consequently, we created tests in the *DbActorSuite.scala* for the new function created for `RollCallData` and for `ElectionData`.

After the implementation of the handler and validator of `witness#message`, we also created new files *HandleWitnessMessageSuite* and *ValidateWitnessMessageSuite* containing the tests for the handling and verifying of the corresponding messages. The verification of a `witness#message` should currently test fully the integrity of the message.

5.2.4 Future work

There are several things in Back-End 2 that need to be taken care of in the near future.

Test coverage

More tests for other, previously untested features should be added. The handlers and validators of `LAO` have very few tests. Moreover concerning the *DbActorSuite.scala* file, the written tests currently do not verify the corner cases. This would certainly increase the coverage of the Scala back end, which was, initially, the subsystem with the lowest coverage.

Server communication

As stated in the e-voting description, to achieve all goals of the secret-ballots elections, we need to have server to server communication but at the time of writing Back-End 2 only supports one server at a time. Even though the support for `witness#message` has been implemented, future work needs to be done to include support for witness servers as described in Section 1.3.2.

Code quality

There are a number of weaknesses in the maintainability and readability of the code that need to be taken care of.

Firstly, the message pipeline uses `scala.util.Either` backwards with regards to the convention. It puts error in the `Right` constructor for `Either` and results in `Left`. Although this is completely functional because `Either` is symmetric, this can be confusing for readers since it flips the convention. Taking a moment to reverse this would therefore do good.

The message pipeline also seems to prefer hiding `scala.concurrent.Future` objects inside methods, making sure to block the thread to wait for them before returning. This isn't the way futures are designed to be used. They can be returned from functions and passed around to represent the result of asynchronous operations, and waiting on them synchronously is discouraged.

Therefore, some work has been done to switch to the intended usage pattern, but it is not yet complete.

The project uses Scala 2.13. A new major version of Scala called Scala 3 is now available. Migrating the codebase can be mostly automated. It would be good to do so since future students will likely learn Scala 3: in particular, CS-210 "Functional Programming" at EPFL, which some future students will have taken, has already switched.

Adopting automated code formatting and styling could reduce friction. There is currently no such mechanism in Back-End 2.

5.3 Front-End 1- React/Typescript

In this section, the architecture and changes specific to the React/Typescript front-end are described. A detailed description of the subsystem specific implementation of the different features of the E-Voting (Chapter 2) and Digital Cash (Chapter 3) projects are largely omitted as the specification is already described in the respective chapters. Interesting implementation parts of specific features are highlighted.

The first section will describe the the architecture of the subsystem set up by other students in previous work and the remaining sections will highlight changes in the architecture and other interesting changes.

5.3.1 System Architecture

Framework and Programming Language

Front-End 1 is developed using `react-native` which is a user-interface (UI) framework on top of `react`, a Javascript library for building web UIs. It uses the same syntax as `react` but provides many general components which allows it to compile to either native iOS or Android applications as well as web applications. The main purpose is building mobile applications which means that while the web application will still run on a desktop machine, it is hard to make it responsive (adapt to different screen sizes) compared to standard CSS styling.

The main concept of `react` is that of so called *components* which make up the whole UI. There is one root component that then embeds all other components of the applications, possibly dependent on the application state and arguments passed to it. The UI consists of a tree of these components where a component can have zero to multiple children. A component is nothing more than a function that returns its children in the UI tree when it is being rendered. Components can accept a set of arguments called *properties* (*props* for short) that allows it to return a UI dependent on these properties.

A simple example is a button component that displays a certain text and performs a given action if the text is touched, can be defined as follows:

```
type Props = { text: string, onPress: () => void; };

const Button = ({ text, onPress }: Props) => {
  return (
    <TouchableOpacity onPress={onPress}>
      <Text>{text}</Text>
    </TouchableOpacity>
  );
}
```

In the above example, `TouchableOpacity` is another component that handles touches and shows visible feedback to the user.

Even though `react-native` and `react` are Javascript libraries, the majority of the code is written in Typescript, a superset of Javascript . Javascript is a weakly, dynamically typed language which allows the detection of type errors during compile time (Typescript is compiled to Javascript). In contrast to that, Typescript is statically typed which allows type-checks to be performed by the compiler (and also the linter). Moreover, `react` extends both, Javascript and Typescript with an XML-like syntax that allows an easier to read representation of the UI as visible in the above example, especially if the reader is already accustomed to HTML.

In the following we will only refer to Javascript even if the implementation is done in Typescript unless it is something very specific to Typescript.

State Management

For state management, Front-End 1 relies on another Javascript library called `redux`. It works by splitting state management functionality into two pieces: `actions` that are *dispatched* by the UI to request a state change and `reducers` that receive these `actions` and based on the `action` and the current state compute the updated state. The actions act as an abstract interface to the underlying state storage and state updates allowing reducers to change without requiring changes in all places actions are dispatched.

A reducer is simply a function accepting an action and the current state that then returns the new state. So a reducer can be as simple as in the following example.

```
const countReducer = (
  state = 0,
  action: AnyAction
) => {
```

```
  switch(action.type){
    case "INCREASE":
      return state + (action as IncAction).value;
    case "DECREASE":
      return state - (action as DecAction).value;
    default:
      return state;
  }
};
```

There are some subtleties that have to be taken into account when writing reducers, for example `redux` requires reducer states to be immutable. Fortunately there are libraries such as `@reduxjs/toolkit` that enforce this property and make writing reducers as well as actions even easier by removing a lot of the boilerplate code that is required otherwise.

Data Flow

Given the communication architecture described in Section 1.3.1, we know that every message in the system is broadcasted to all attendees, including messages sent by ourselves. This allows Front-End 1 to solely rely on incoming messages for state updates making it very trivial to keep the state consistent with the back-ends.

In a bit more detail, after connecting to a back-end, Front-End 1 maintains a websocket connection and listens for incoming messages. As soon as the network module receives a message, it dispatches a `redux` action resulting in it being stored in the corresponding reducer. The so-called *ingestion* module maintains a watcher on this reducer state and starts processing added messages by calling the corresponding message handlers. These in turn then again dispatch redux actions to change the corresponding application state. The user interface maintains watchers on the reducers relevant for the UI and triggers a re-render as soon as the data in the respective reducers changed.

This indirection via the redux store decouples the UI from the message ingestion and the message ingestion from the networking module making them easier to maintain and test in isolation of each other.

As mentioned in the beginning of this Section, attendees also receive a broadcast of messages they sent themselves. This means it suffices to send out the corresponding network message after an interaction with the UI and at some point this message will then be received by the network module and the previously described process starts. In short, the communication architecture allows the same UI update logic to be used for state updates caused by any attendee and no special cases are needed for the user that

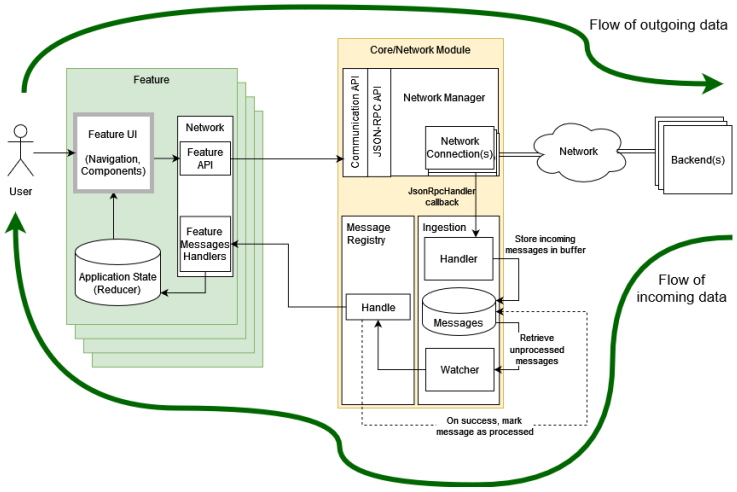


Figure 5.1: Data Flow Architecture of Front-End 1

interacted with the UI. Figure 5.1 graphically depicts the just described data flow architecture.

5.3.2 E-Voting

An interesting aspect during the implementation of the e-voting functionality was the realization that previously it was never checked by the front-end whether an `election#result` was sent by the organizer. This did not really matter since it was in any case only possible to connect to the organizer's back-end which means this check was performed implicitly. Now since the `lao#greet` message was introduced (see Chapter 2) it is finally possible to properly verify the authenticity of this message analogous to the `election#key` message.

Unfortunately there is the problem that the current implementation of the `lao#greet` message is still not fully sound since the back-ends do not both support witness signatures including the special case where the organizer front-end signs the `lao#greet` message. Thus at the time of writing, Front-End 1 accepts any `lao#greet` message as valid. This has to be fixed at latest before there is server-to-server communication and the front-end will be able to connect to a LAO via a witness or a second organizer's back-end. The corresponding check was already implemented in Front-End 1 but it is disabled with and marked with a `//FIXME` comment.

For the implementation of the vote encryption, the npm package `@dedis/-kyber` provided by the DEDIS lab was used as it is compatible with the kyber libraries used on the back-ends. Unfortunately the library does not directly export a key generation or en- and decryption functions which thus had to be implemented manually based on a provided Go code sample. The hardest

part in the implementation was getting the encoding of the encrypted data correctly aligned with the back-ends.

Other than these points, the implementation of the e-voting feature in Front-End 1 was rather straight-forward.

5.3.3 Inter-Feature Dependencies

Until the end of last semester, the project was structured by functionality and not by feature making it harder to realize which files belong to what feature and to see what the dependencies between features are. In between the two semesters, the DEDIS lab changed the structure to be feature based which made it much easier for us to get started. Now since the features previously were all intertwined and there was not a lot of time between the two semesters, simply restructuring where files are stored resulted in a lot of *inter-feature* dependencies such as the evoting feature importing functionality from the lao feature. Since it is in feature-based architectures preferable to have the features as independent as possible to, for example, allow disabling some of them, we put quite some effort into separating them. Now of course, completely separating them is simply impossible as for example most features simply depend on the lao feature. Thus we decided to go for dependency injection which is a design pattern in which the dependencies are explicitly passed at runtime. More specifically, we modeled each feature as having some in- and some output. The input consists of a *configuration* containing all dependencies and the output consisting of an *interface* that the feature exposes to other features. Each feature then exports a `configure` function accepting the configuration, initializing the feature and returning the exposed interface. The example below shows a simple example of how this could look like.

```
interface MyFeatureConfiguration {
  getCurrentLaoId: () => Hash;
}

interface MyFeatureInterface {
  getEvent: (eventId: Hash) => Event;
}

const configure = (
  config: MyFeatureConfiguration
): MyFeatureInterface => {
  initMyFeature(config);

  return {
    getEvent: (eventId: Hash) =>
```

```
      EventStore.get(
        config.getCurrentLaoId(),
        eventId
      );
  };
}
```

After having decoupled features this way, there is a point where each feature's `configure` function has to be called with the proper dependencies. This then puts everything together and initializes all desired features.

```
const configureFeatures = () => {
  const laoInterface = laoFeature.configure();
  const eventInterface = eventFeature.configure({
    getCurrentLaoId: laoInterface.getCurrentLaoId
  });
};
```

In case of circular dependencies, this gets slightly more complicated. One way to resolve them, is to add a second `configure` function, e.g. `configure2`, which then allows a feature to be initialized in multiple steps: First `configure` is called, the returned interface is used to initialize a second feature and the interface exported by this second feature is then in turn used to call `configure2` on the first.

To enable react components to access this feature configuration object, we leverage a react feature called *context*. A context allows passing properties from one component to all children component, no matter how deeply nested they are. By wrapping the whole application in a feature context where each feature adds its own configuration, react components of all features can access their respective configuration object and this ways access the injected dependencies.

At the time of writing all inter-feature dependencies have been replaced by dependency injection except with the exception of the social media feature.

5.3.4 Client to Multiple Servers Communication

In the future clients will need to be able to connect to multiple servers in order to enable the protection against censorship described in Section 1.3.2. The networking module already partially supported multiple connections by the front-end but the support for it was not implemented all the way. During this semester we changed the LAO state to use a list of server addresses rather than a single address, we added the possibility to manually add a second server while being connected to a LAO and we made sure that when writing the `lao#greet` specification, it included a `peers` field that contains

the list of other servers hosting the same LAO. This field in the message solves bootstrapping for the front-end in cases of honest back-ends.

5.3.5 Properly Typed Navigation

In the beginning of the semester, the type checking for navigation was disabled by using the Typescript `any` type which prevented the linter from checking whether all issued navigation events are correct. This issue was resolved by statically defining the list of all navigation screens and the arguments they accept. Unfortunately it was not possible to do this in the different features since this type needs to be statically known and dependency injection only happens at runtime. Thus the options we came up with were to either 1) have this types global, i.e. in the core of the application, 2) to have it in the features requiring inter-feature imports (which is something we wanted to remove, see Section 5.3.3) or 3) to keep it the way it was. Given this choice, we decided for option one which seemed to be the least bad option.

5.3.6 Code Conventions

In the process of working on the project, we realized that enforcing additional conventions on the code will be helpful. In the following we list the linter rules we added and changed.

Import Order

Previously imports were in no particular order. To keep things more readable we added a linter rule enforcing an order on the imports by grouping different types of imports, requiring a newline between the groups and requiring alphabetical order within the groups.

Color Literals

In general only a small set of different colors should be used throughout an application so that the user gets a consistent experience across features and screens. We try to force developers to use colors of the existing color palette by forbidding inline color literals. Unfortunately we did not find a linter rule enforcing the same rule for numeric literals as spacing, border with etc. should as well be consistent throughout the application.

Inline Styles

Inline styles were already discouraged before, i.e. the linter printed warnings. During this semester we were able to remove all inline styles and change the linter rule to report errors instead of just warnings.

5.3.7 Witnessing

Support for the basic `message#witness` message (see Section 1.3.2) was already implemented in Front-End 1 but there was no UI built for it. There are three different cases for messages: Either they 1) have to be witnesses manually, i.e. the content of the message has to be verified by the witness, 2) they can be witnessed automatically because the content of the message is not of our concern, for example cast votes and 3) they do not have to be witnessed at all.

Supporting 3) only required adding a map telling the application which messages have to be witnessed and which do not. In order to support 2) we also need to differentiate between *active* and *passive* witnessing but the same map can be used for this. Automatically sending the already implemented message in cases of messages that can be witnessed passively was straightforward to implement.

Supporting 1), i.e. messages whose content should be manually checked by the witness before approving, is the hardest to support as it is not trivial to design a user interface for this.

Showing a popup every time the front-end receives a message results in a very bad user experience (UX) and thus we decided for an implementation that allows for delayed choices. To do so, we implemented a general notification feature that can in the future be used by other features as well. If the front-end receives a message that should be witnessed manually, it adds a notification to the list allowing the user to decide on it as soon as they have time for it. Since there are a ton of different messages in the protocol and time during the semester was limited, we did not yet design a proper way to display messages to the user and the current screen displaying the decoded json message mostly acts as a proof-of-concept (PoC).

5.3.8 User Interface

The user interface of Front-End 1 has been completely overhauled, the details can be found in the *Engineering a Production-Ready System* chapter in Section 4.4.2.

5.3.9 Future Work

There are several things in Front-End 1 that in the (near) future should be taken care of:

- **Inter-Feature Dependencies**
  It would be great to remove the remaining inter-feature dependencies from the social media feature. Given the already done work this should not be a very time-consuming task and is planned for the time between

the report submission and the presentation. After doing so, `depcruise`, a tool for detecting inter-feature dependencies, should be set to report errors instead of just warnings.

- **Re-Subscription Bug**
  There is an issue where Front-End 1 is no longer subscribed to sub-channels of the LAO after it is reloaded or the connection breaks. This is due to the fact that both back-ends for each channel directly store the set of TCP sockets that have previously subscribed rather then the set of public keys of the corresponding users. At the time of writing it was not decided what the best option is to resolve this issue. Either the back-ends switch their implementation to public keys or Front-End 1 must add changes that make it re-subscribe to all relevant channels after a connection breaks.

- **Native Builds**
  At the time of writing building native applications is impossible. This is because several `react` dependencies are used that are only web-compatible and not with native applications. Among them are the date picker and the QR code scanner. For both, alternatives exists which support native applications, in the case of the QR code scanner it will require some additional work though since it either requires updating our dependencies (the newest version of expo has a QR code scanner supporting both, native applications and the web) or loading the old expo QR code scanner only for native applications and a web-specific scanner for the web since it only supports native applications. Obviously all css imports also need to be made conditional to the web as it is not supported in native applications.

- **Literal Strings and Constants**
  At the moment there are still quite a lot of literal strings in the UI code even though they should all be placed in `strings.ts` as by the current convention. The reason for it is supporting translation in the future. It might be worth considering `react-intl / Format.JS`, a library that takes care of translation and does not require storing them all in one central location. It can generate this global list of all strings automatically from the source code where messages are defined in their corresponding features. At the same time it should be taken care of separating UI strings from application constants, at the moment they are partially mixed.

- **UI and UX**
  Even though a lot of progress has been made on that front, ensuring user experience is a continuous task and never finished. Therefore it is almost certain that there are also issues with the new user interface. Moreover there are also parts of the UI that were mostly untouched

(due to the limited time) but would also need a brush up.

5.4 Front-End 2 - Android

In this section, the architecture and changes specific to the Java front-end are described. A description of the subsystem specific implementation of the features of the E-Voting (Chapter 2) and Digital Cash (Chapter 3) are briefly described. We focus on interesting implementation that are specific to the Java subsystem. We will first begin with the interesting features of the E-Voting part, notably the encryption and decryption process. Then we will discuss about all the changes that have been made regarding the digital cash team. Finally, we won't discuss here the code consolidation that has already been described in Production-Ready.

5.4.1 E-Voting

Initial state

The original state of the E-Voting was pretty complete: message handling was well written and most importantly, it was possible to do an election successfully (at least with the Back-End 1 back-end). Some tests were created, but it was not complete. Furthermore, the current implementation relied heavily on time events. Consensus was also implemented for the process of opening an election. On top of this, user interface design was pretty simple and not very user friendly.

Modified components

The main goal of Front-End 2 was firstly to remove time dependency for opening and ending an election.

The handler of `election#end` message was already implemented, we had to only enable a button for the organizer to end the election. For opening an election, the work to be done with `election#open` was more consistent, since the message had not been implemented yet. In the same way as for the `election#end`, a button was introduced for the organizer to open the election whenever he wanted. The appropriate handler *handleElectionOpen* had to be implemented too, so the election could actually begin upon receiving the `election#open` broadcast.

Concerning the `greet#LAO`, we needed to find a way to store the address of the server. The simplest way to do it was to introduce a new global repository *ServerRepository*. The new repository stores the address given a LAOId in a *HashMap*, after a successful handling of the `greet#LAO` by the *handleGreetLao* function.

Encryption and Decryption

Prior to actually encrypting the vote and casting it, a modification of the *Election* object had to be done to introduce two new fields. Firstly, *ElectionVersion* which indicates if the election should be encrypted or not, and *electionKey* parameter which stores the value of the public key sent by the back-end in the `election#key` message. Appropriate handling and tests for the new `election#key` message and the new version of the `election#setup` had to be written. Finally, the user could now choose, on the setup screen, choose the version of the election he wanted, respectively *open-ballot* or *secret-ballot*.

Concerning encryption, we included the whole DEDIS cothority library project in the Java side as a dependency, because we wanted the *Kyber* library and the cryptographic primitives.

The encryption was done by the *encrypt* function in the *ElectionPublicKey* class. *ElectionKeyPair* and *ElectionPrivateKey* classes were also added, so that we could encrypt and decrypt, and also generate public and private key sets.

The new version of the `election#cast_vote` required a bit more of work because of the *oneOf* field of the Json message. Thus, an appropriate *deserializer* had to be coded to correctly deserialize the vote object into the correct type (either *ElectionVote* or *ElectionEncryptedVote*). Furthermore, we made the appropriate changes so that only one vote could be selected. The appropriate type changes and signatures were done inside the code base.

Finally, some checks needed to be implemented to compute the *hash* of the *voteId* for the `election#cast_vote` and the *hash* of the voteId's for the `election#end` message.

5.4.2 Digital Cash

The Figure 5.2 shows the process behind the digital cash feature. In fact, before using this feature, it is necessary to create a LAO, then to create a roll call, to enter it, and finally close it. [1] The user can use the interface in two ways: by issuing first money when acting as the organizer of the LAO, or by sending money when acting as a member of the roll call. Users also have the possibility to see their last received money on the interface. [2] The sending information on the user interface (send/issue fragment) is passed to the view model which transforms it into a transaction message. Then, it publishes the json message according to the `postTransactioncoin.json` and sends it to the coin channel that had been set up with creation of the LAO. [3] The message is handled by the back ends.[4] The `TransactionCoinHandler` gets the message from the channel [5]. Then the `TransactionCoinHandler` calls the `LAO` which updates the state of the application. It keeps in memory the last transaction per user, that is a dictionary which indicates the current

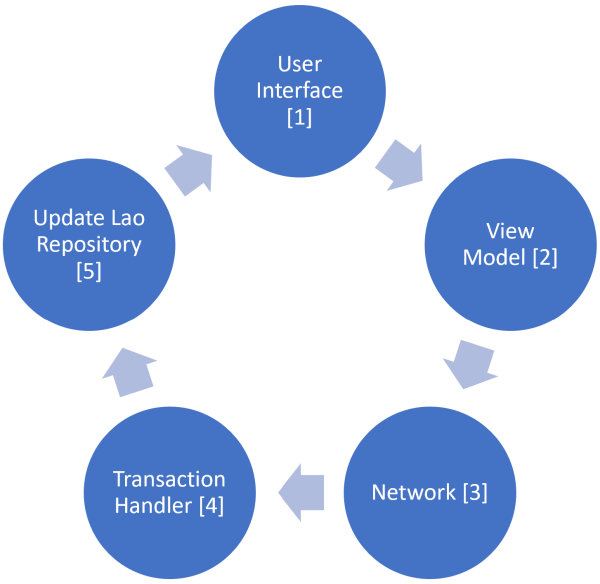


Figure 5.2: Steps for the digital cash feature

amount of money available per user public key. In the same way, there is an overall transaction history per user public key.

Further detail on the user interface

Let us now detail the topic of the [2] user interface of the application. Under the current Beta-Version, it shows the main activity `DigitalCashMain.java` and several fragments for the menu options such as ”home”, ”send”, ”receive”, ”history” and ”issue” which are all wired and handled by a view model.

As pointed out previously, the digital cash feature can only be used after creating a LAO and entering/closing a Roll Call. A button on the LAO menu authorizes the access to the digital cash feature. Figure 5.3 displays the user experience using the extension. First, we have a home environment which displays address and amount for the user of the current user device. Below, the menu displays each digital cash option. In case, the user is the LAO organizer, he/her can click the issue menu button in order to send money to any member of the last closed roll call or to all the members of LAO. For the moment, the witness option is not implemented yet. In case the user is just a member that has enough money and entered the roll call, he/she can use the send fragment to send an amount to another user. Once the send button is pressed, a receipt for the transaction is produced with on it the

sender and the amount. Finally, by using the receive option a user sees the last transaction that concerned him or her. Amount, sender ID, QR code and elapsed time are also displayed.

Further detail on the communication

The digital cash communication [3] follows the JSON schema agreed upon throughout the semester 3. To be able to communicate with the back ends, a java android front-end was required to create special classes which are linked to the communication objects and sub-objects. Therefore we have created the `PostTransactionCoin`, `Transaction`, `Input`, `Output`, `ScriptOutput` and `ScriptIntput` classes. The view model is the one which publishes the network message on the coin channel with the preexisting message handler and communication classes that were already available in the code.

The information about the transaction is then broadcasted to all devices and handled by the `TransactionCoinHandler` which creates a `TransactionObject`. Then the `LAO` of each device is updated with the state of transaction for the application according to public key mapping. The names for the dictionaries used are `transactionPerUser` and `transactionHistory` representing the last transactions per public key, giving the current state of money of the user with the output as well as the history of all transactions per user.

5.4.3 Production Ready

During the refactoring of the event list UI we implemented a significant change as to how event state are handled. Previously, there was a hybrid (and quite buggy) implementation of both event and (poorly handled) time. Handing time is messy and therefore from the start the PoPStellar app was never intended to be solely based on time for crucial elements like the closing of an event. Nevertheless the hybridism resided that the opening of election was time based, or rather was a poor attempt at that because the refresher was not working at all and the only option left was to leave the view and come back to it to have it updated. The closing was a mix of both where the closing button was enable only when the time of election was passed. It suffered of the same updating issues as the opening.

We fix most of the problem by having the state be purely event driven and time only be a suggestion and has therefore no impact whatsoever on state. It is nevertheless displayed as organising real life event would require indicating start and end times. There subsists a bug for elections where the event might jump to the right section of the list (i.e. previous, current and upcoming events) though we hope it will be fixed by the final presentation. We don't provide a higher confidence of fixing it because initial analysis of the bug seems to indicate a non-trivial issue.

5.4.4 Future work

Test coverage

Some tests still need to be added to fully cover all cases of the handling of the different *message* for the election. At this stage, even though a lot of tests have already been written and some UI tests still need to be added for the digital cash and E-votin as well.

Server communication

Future attempts to implement multiple server communication could use the newly added *ServerRepository* to connect to multiple back-ends. However, during the elapsed semester, nothing in particular has been implemented in this field.

Code maintainability

As for the code maintainability, we believe that the code will be a lot easier to maintain after the changes done by the production ready team. Indeed, new code pushed this semester had good coverage minimizing the need for additional testing.

Suggested front end development for the Digital Cash feature

A suggestion for further improvement of the front end would be to add a history fragment which should display the entire LAO transaction history per user as shown on the figure 5.4, some scanning possibilities for the QR code in the send/issue fragments as well as a witness option.

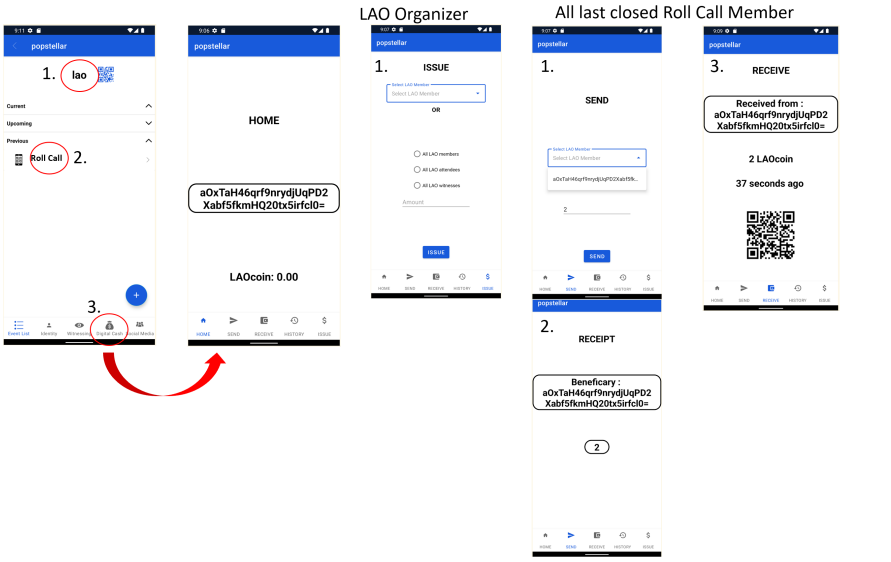


Figure 5.3: Digital Cash User Interface

Figure 5.4: History of Transaction. (currently filled with static value but we're confident that It will not be the case at the end of the semester)

Chapter 6

Conclusions

During most of the semester, we were behind the schedule across all three projects and subsystems, and it mostly was the case that the builds deployed for PoP parties broke because of trivial issues that could have been resolved before if somebody had tested it earlier. In general, communication during the first half of the semester was lacking and we did not do a lot of progress. What did not help this is the fact that the specification of the different features was not finished until quite late in the semester.

Unfortunately, at the time of writing, not all primary goals of each of the three projects, E-Voting, Digital Cash and Engineering a Production-Ready System are already met and the system is not bug-free. Given this state, the DEDIS team allows us to work for an additional four weeks on the project to finish what we started.

But not all is bad, especially given that the state of certain subsystems was certainly not ideal at the beginning of the semester: Front-end 2 (Android) simply crashed when trying to connect to a LAO, back-end 2 (Scala) did not support elections at all, back-end 1 (Go) did not check any hashes at all and front-end 1 had a questionable user interface with text and buttons all over the screen. And this is just the beginning.

So even though a lot could have gone better, especially in the beginning of the semester, still a lot has been achieved. The system as a whole is now much more stable, there are not 10 undocumented tricks you have to know to get the system running and the e-voting team even managed to merge the secret ballot elections into the master branch right before the last global meeting. There are some small bugs that have been discovered but nothing that cannot be fixed until the presentation.

The digital cash project has taken up pace since the team finished tweaking the specification and we are confident to be able to deliver a working prototype by the date of the presentation. At the time of writing, the back-ends

support the required messages and for the front-ends there are open PRs that just need some polishing before they can be merged.

And last but most definitely not least, there has been a lot of progress on the production-readiness of the system. As already mentioned before, the system is able to run in a much more stable state than in the beginning of the semester which is, in a big part, thanks to the production-ready system team that continuously kept pushing for a higher test coverage, convinced the whole team to document all discovered bugs to keep track of them, and kept pointing out outstanding bugs. And not only that, by writing integration tests, they were able to discover a big range of critical bugs which would have completely undermined many of the security guarantees in the system if they were not discovered. On the front-end side, a lot has improved for the users of the systems. Both front-ends overhauled the user interface to provide a better user experience and making the system easier to use.

All in all, the project definitely did not go flawlessly but still a lot of progress has been made and we think the project is on the right track for the future.

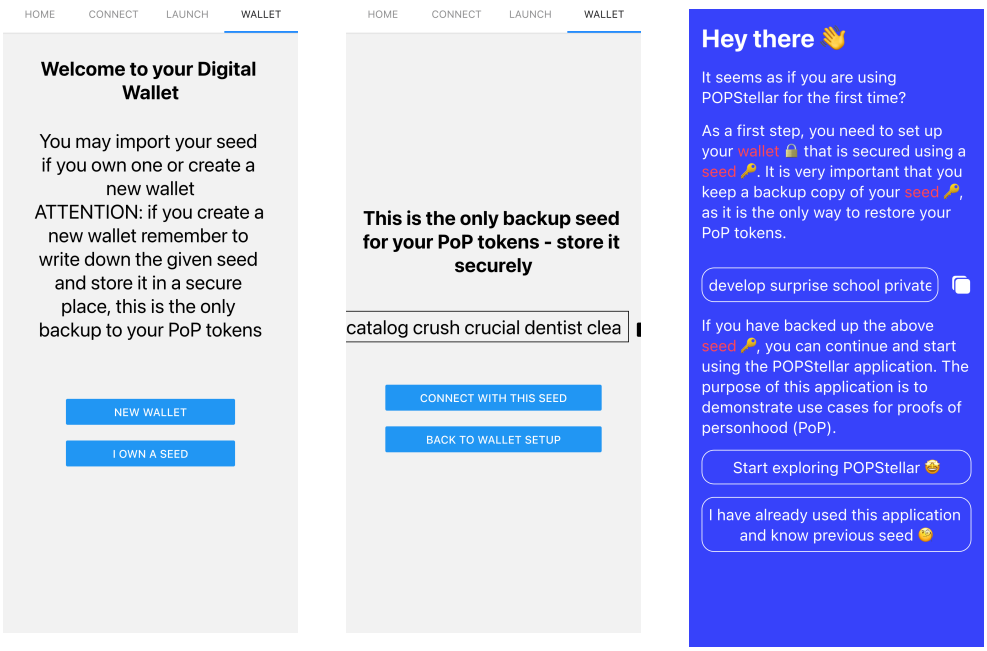
We would like to thank all our supervisors for their great assistance throughout the semester!

Appendix A

Appendix

A.1 Front-end 1 UI comparisons

A.2 Front-end 2 UI comparisons

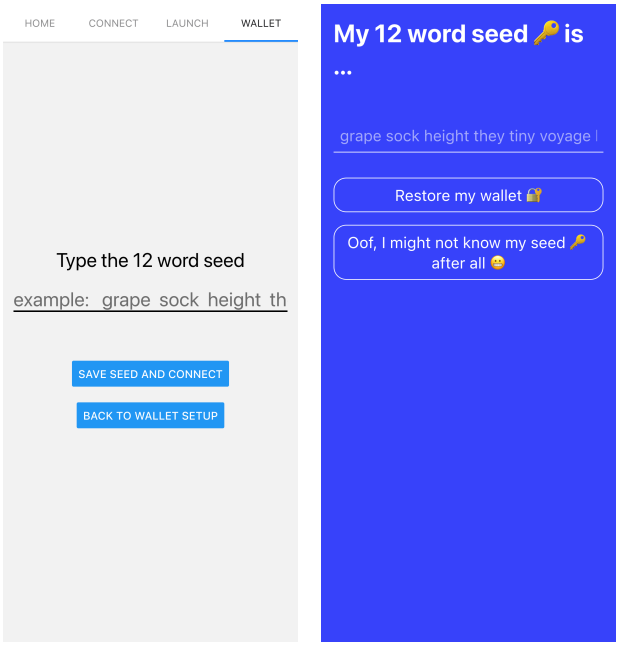


(a) Old UI for non-initialized wallets

(b) Old UI for setting up a new wallet

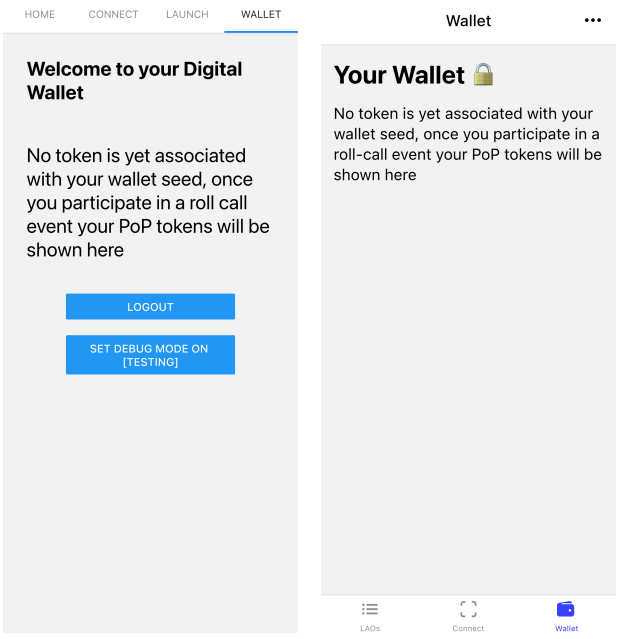
(c) New, combined UI

Figure A.1: A side-by-side comparison for the wallet setup screen in front-end 1



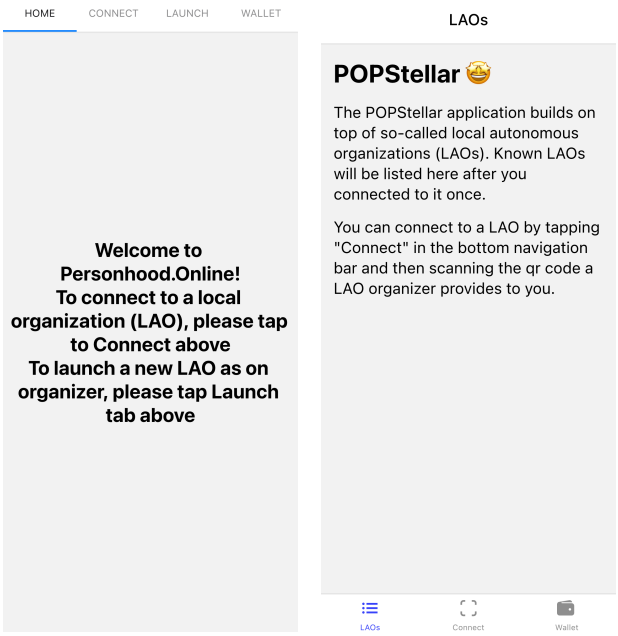
(a) Old UI **(b)** New UI

Figure A.2: A side-by-side comparison of the screen for restoring a wallet in front-end 1



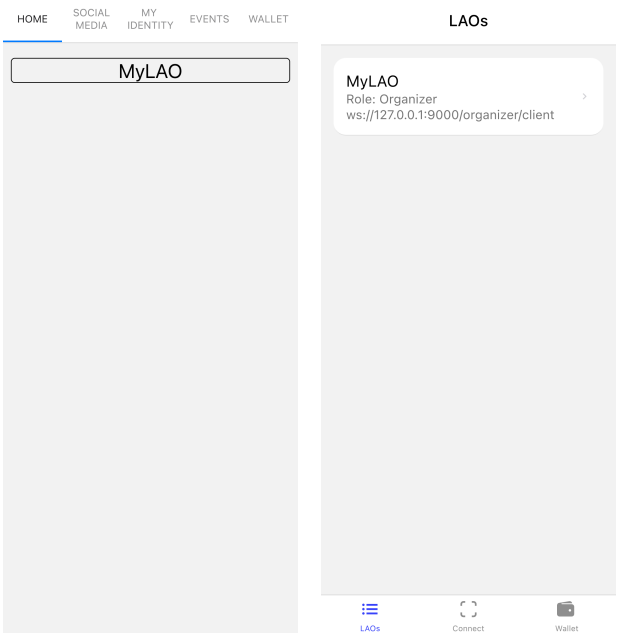
(a) Old UI **(b)** New UI

Figure A.3: A side-by-side comparison of the screen for a setup wallet in front-end 1



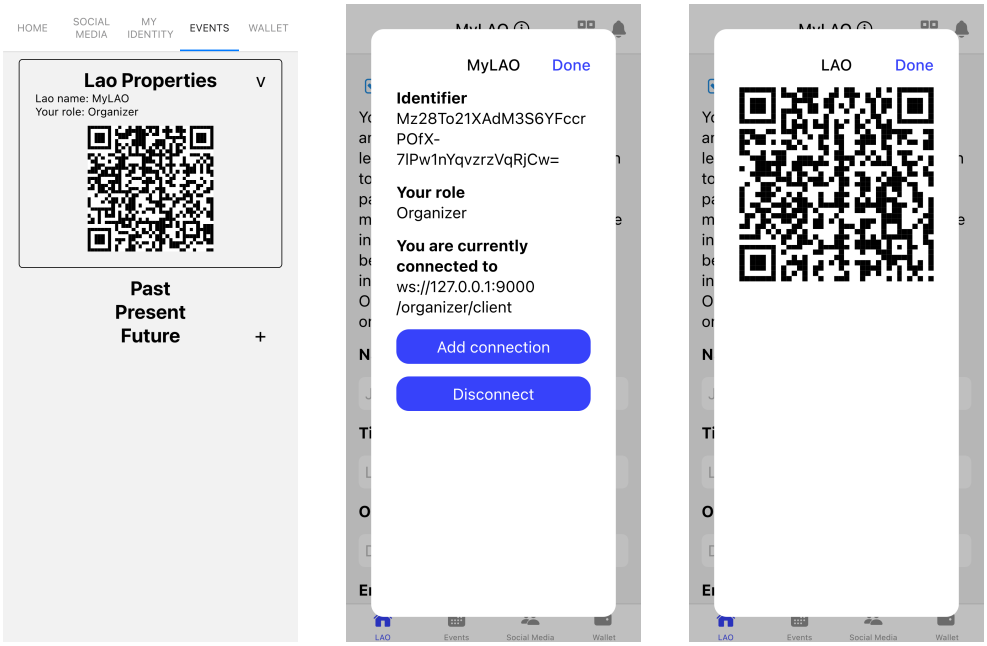
(a) Old UI **(b)** New UI

Figure A.4: A side-by-side comparison of the home screen in front-end 1



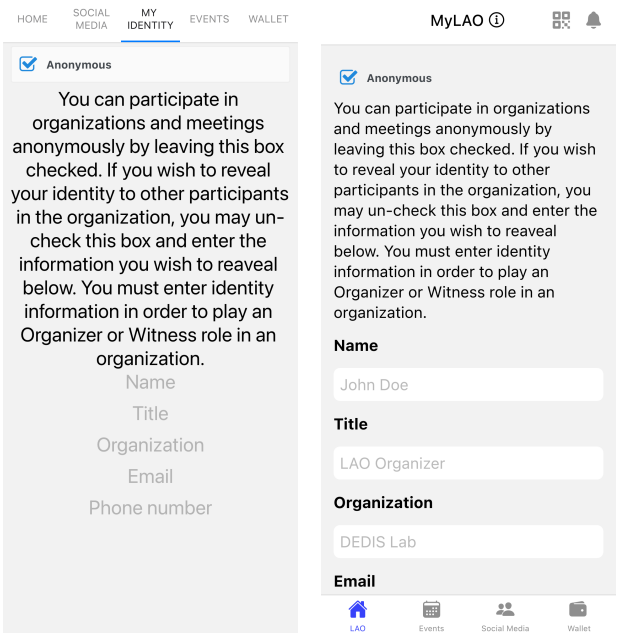
(a) Old UI **(b)** New UI

Figure A.5: A side-by-side comparison of the list of LAOs in front-end 1



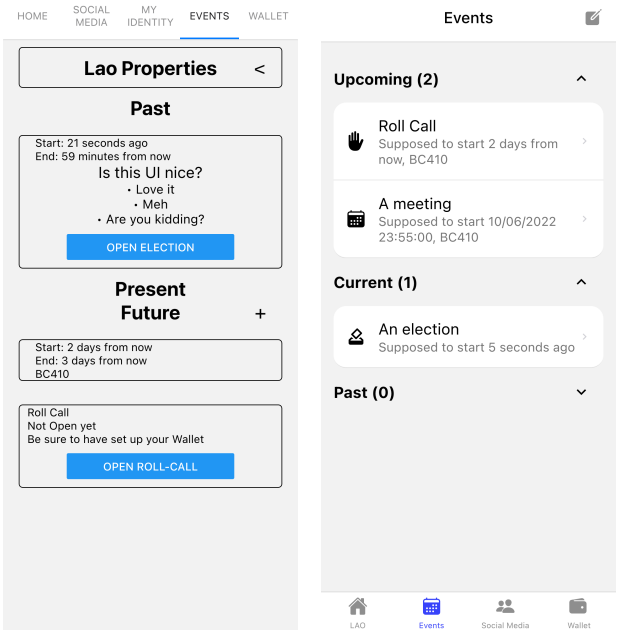
(a) Old UI **(b)** New UI **(c)** New UI

Figure A.6: A side-by-side comparison of the screens showing LAO properties in front-end 1



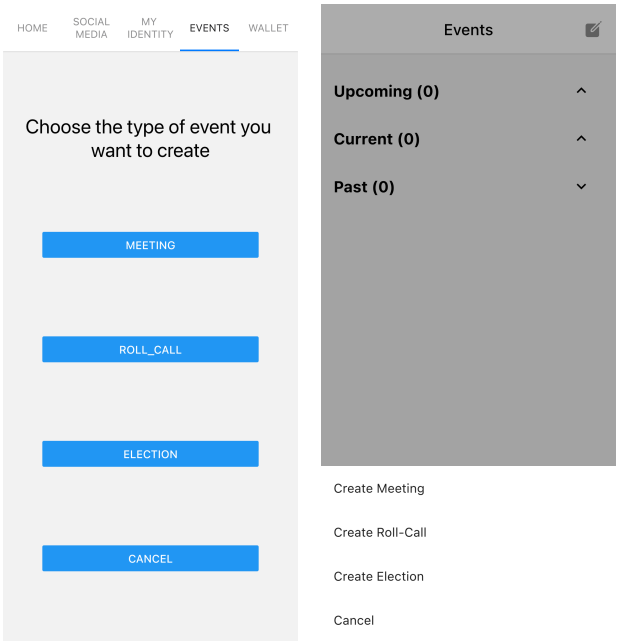
(a) Old UI **(b)** New UI

Figure A.7: A side-by-side comparison of the LAO identity screen in front-end 1



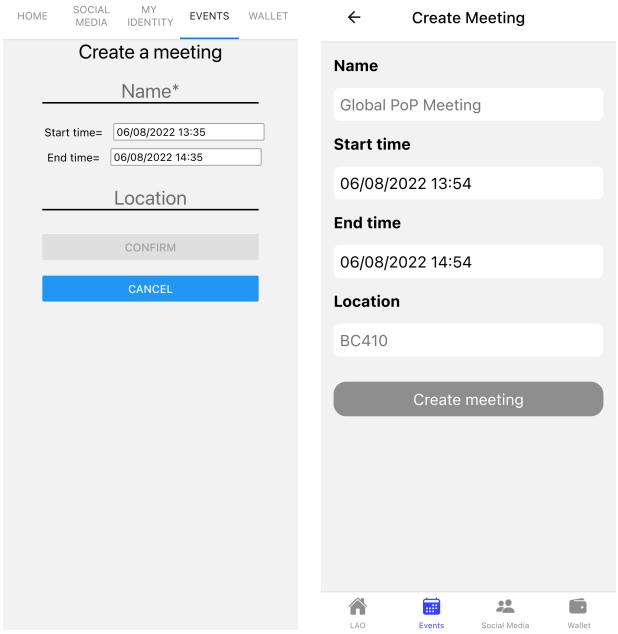
(a) Old UI **(b)** New UI

Figure A.8: A side-by-side comparison of the LAO event screen in front-end 1



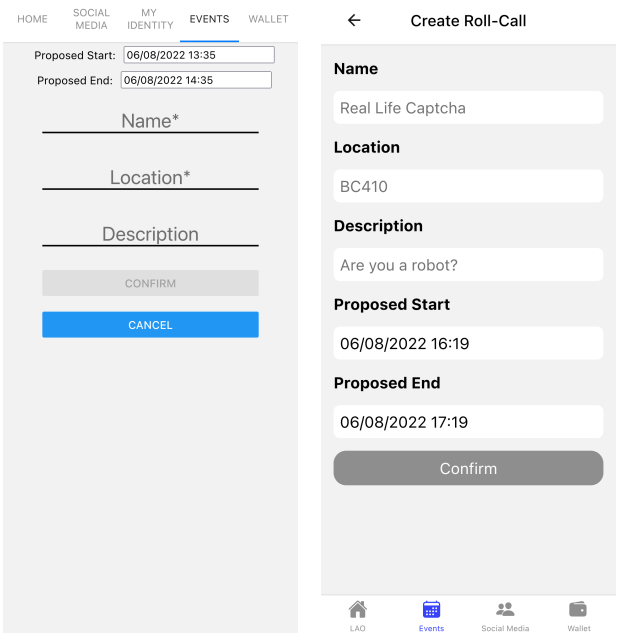
(a) Old UI **(b)** New UI

Figure A.9: A side-by-side comparison of the LAO screen for adding events in front-end 1



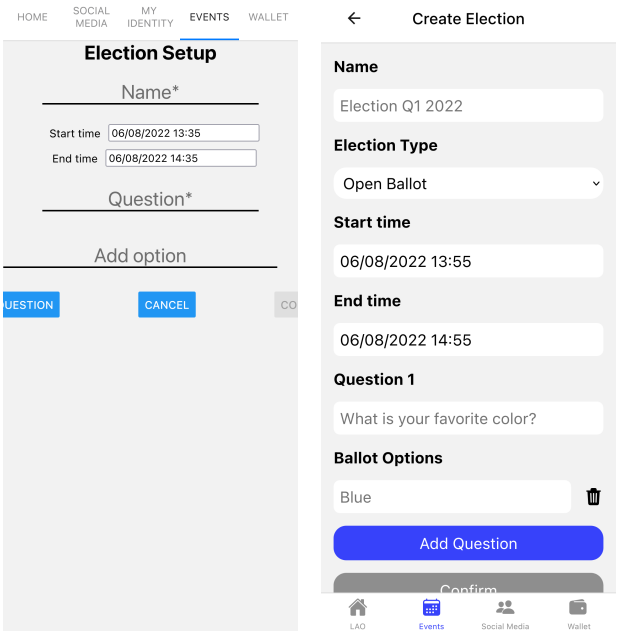
(a) Old UI **(b)** New UI

Figure A.10: A side-by-side comparison of the LAO screen for adding meetings in front-end 1



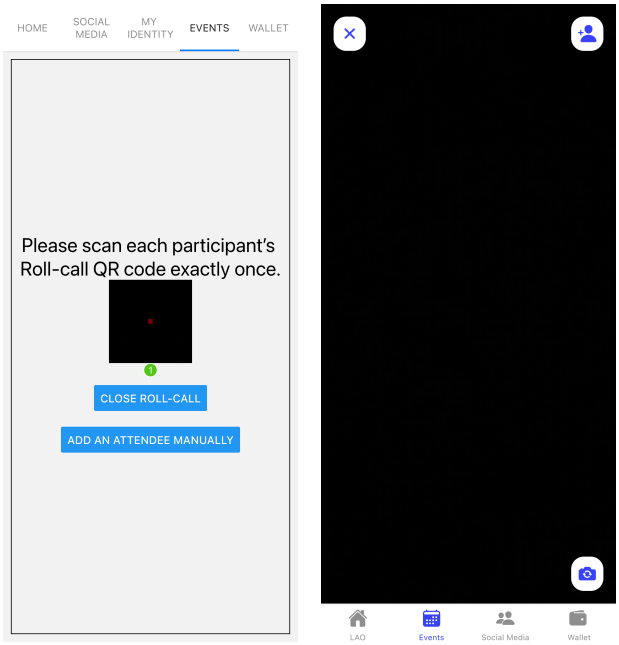
(a) Old UI **(b)** New UI

Figure A.11: A side-by-side comparison of the LAO screen for adding roll calls in front-end 1



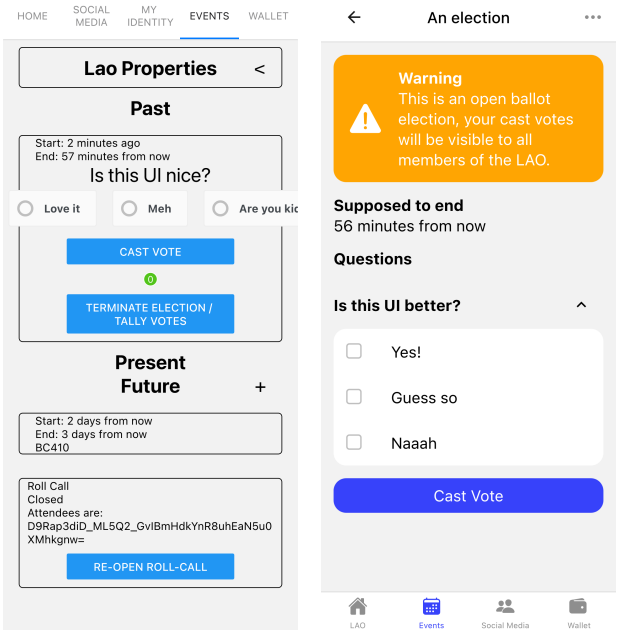
(a) Old UI **(b)** New UI

Figure A.12: A side-by-side comparison of the LAO screen for adding elections in front-end 1



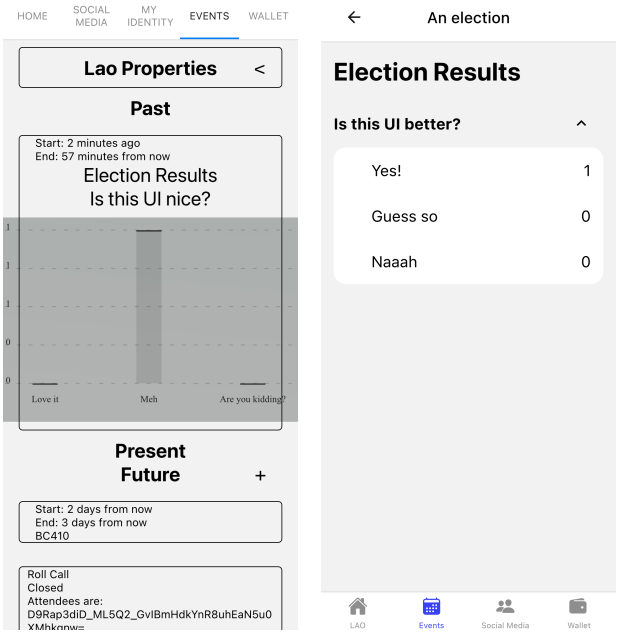
(a) Old UI **(b)** New UI

Figure A.13: A side-by-side comparison of the LAO screen for adding roll call attendees in front-end 1



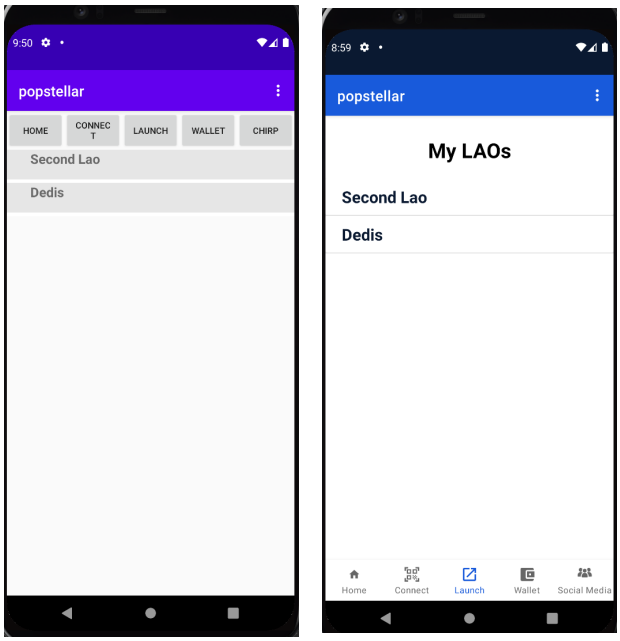
(a) Old UI **(b)** New UI

Figure A.14: A side-by-side comparison of the LAO screen for open elections in front-end 1



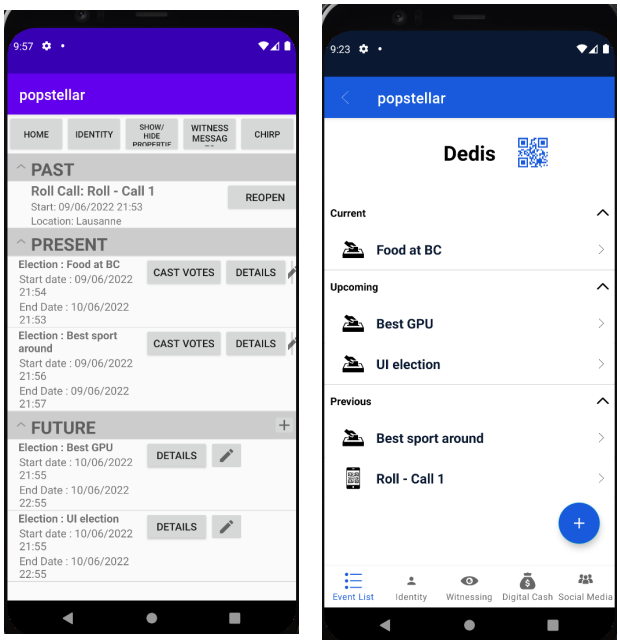
(a) Old UI **(b)** New UI

Figure A.15: A side-by-side comparison of the LAO screen for election results in front-end 1



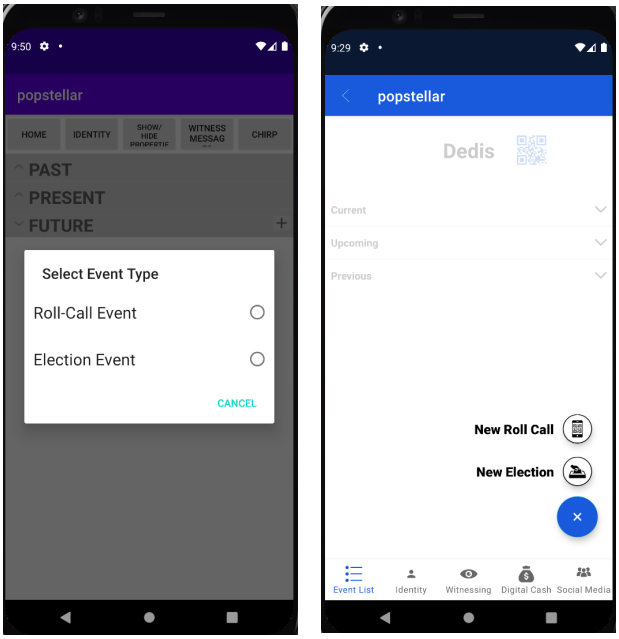
(a) Old UI **(b)** New UI

Figure A.16: A side-by-side comparison of the screen for LAO list on front-end 2



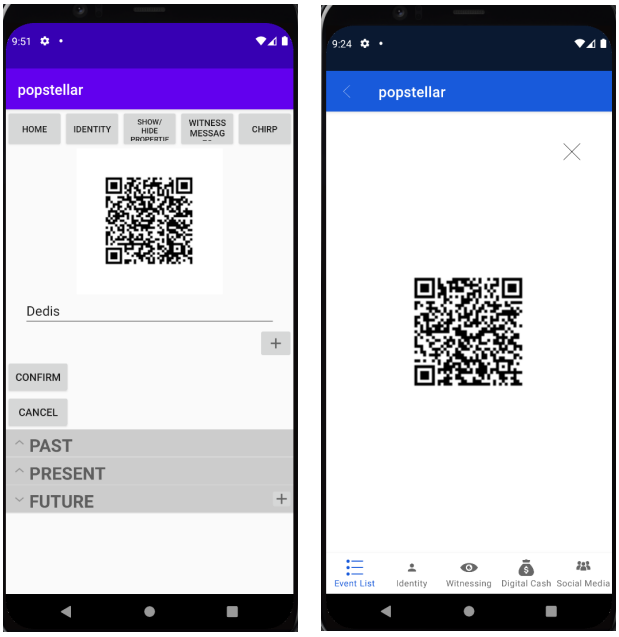
(a) Old UI **(b)** New UI

Figure A.17: A side-by-side comparison of the screen for event list on front-end 2



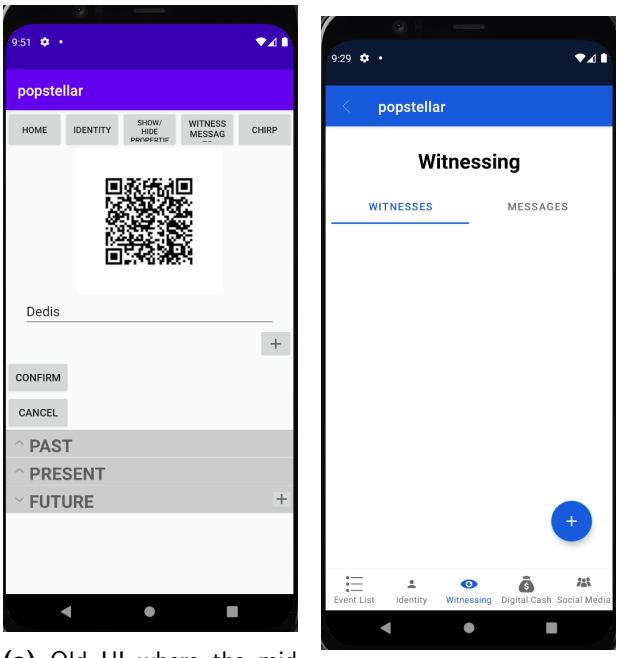
(a) Old UI **(b)** New UI

Figure A.18: A side-by-side comparison of the screen for event addition on front-end 2



(a) Old UI **(b)** New UI

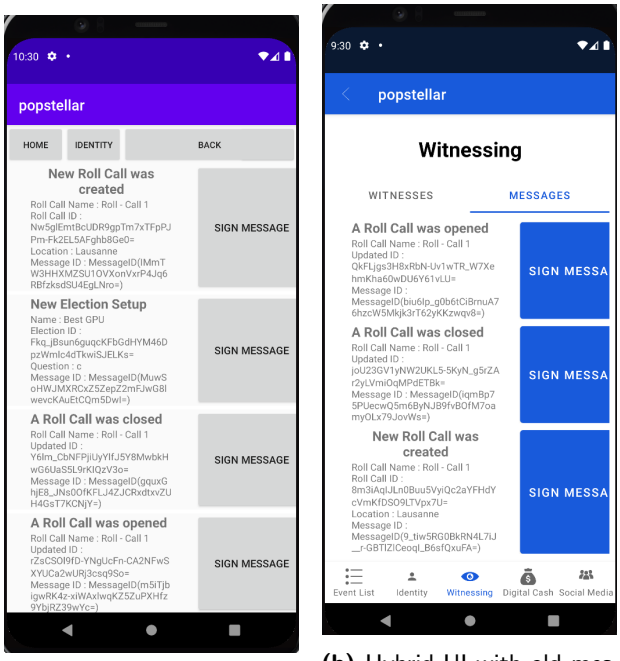
Figure A.19: A side-by-side comparison of the screen for LAO QR code display front-end 2



(a) Old UI where the mid screen + button is for adding witnesses

(b) New UI to display witnesses and add them

Figure A.20: A side-by-side comparison of the witness addition



(a) Old UI, on a separate screen

(b) Hybrid UI with old message layout and new containers

Figure A.21: A side-by-side comparison of the witness addition

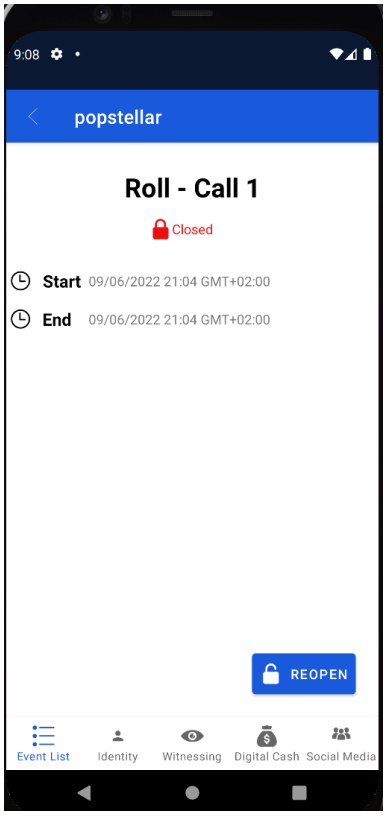
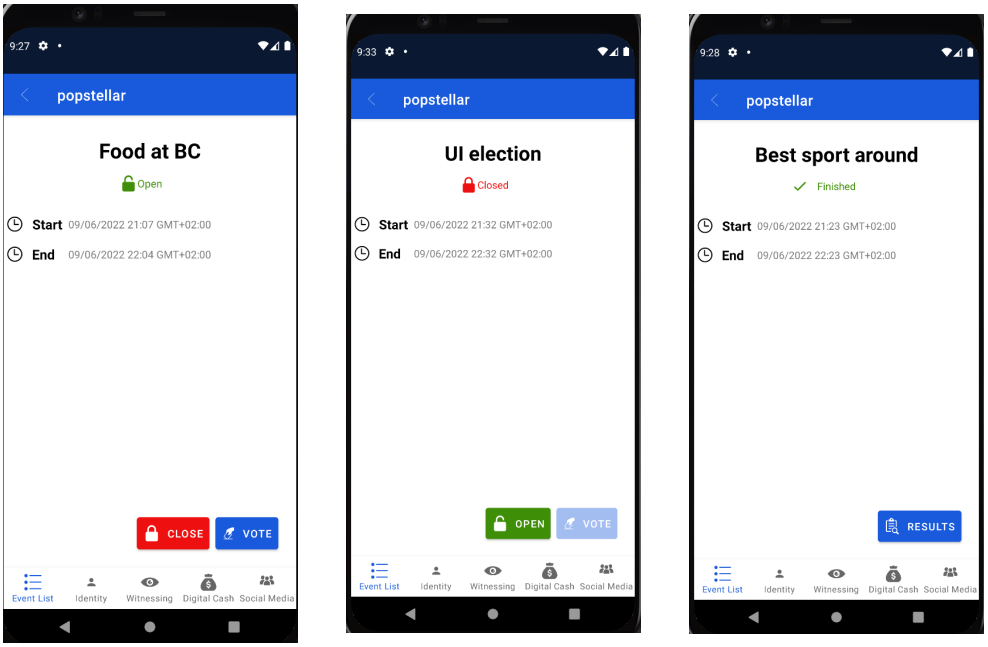


Figure A.22: New UI for roll call details



(a) Opened state **(b)** Closed state **(c)** Finished State

Figure A.23: New election detail UI

Bibliography

[1] Android Developers. Listview. https://developer.android.com/reference/android/widget/ListView, 2022. [Online; accessed 09-June-2022].

[2] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[3] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *2015 IEEE Symposium on Security and Privacy*, pages 287–304, 2015.

[4] Lucy Bernholz, Hélène Landemore, and Rob Reich. *Digital Technology and democratic theory*, chapter 2. University of Chicago Press, 2021.

[5] Erin Dachtler and Alex Van de Sande. Ethereum blockies. https://github.com/ethereum/blockies.

[6] Wei Dai. b-money. http://www.weidai.com/bmoney.txt, 1998.

[7] Bryan Ford. Identity and personhood in digital democracy: Evaluating inclusion, equality, security, and privacy in pseudonym parties and other proofs of personhood, 2020.

[8] PopStellar Lab. Hashlen function. https://github.com/dedis/popstellar/blob/master/docs/protocol.md#concatenation-for-hashing.

[9] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[10] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. *Proceedings of the 8th ACM conference on Computer and Communications Security - CCS '01*, 2001.

[11] Shen Noether. Ring signature confidential transactions for monero. Cryptology ePrint Archive, Paper 2015/1098, 2015. https://eprint.iacr.org/2015/1098.

[12] Pieter Wuille and Greg Maxwell. Bip 0173. https://en.bitcoin.it/wiki/BIP_0173. [Online; accessed 08-June-2022].

[13] Nicolas van Saberhagen. Cryptonote v 2.0. https://www.bytecoin.org/old/whitepaper.pdf, 2013.

[14] Wikipedia contributors. Integration testing — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Integration_testing, 2021. [Online; accessed 07-June-2022].

[15] Wikipedia contributors. One man, one vote — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=One_man,_one_vote&oldid=1082021501, 2022. [Online; accessed 20-May-2022].