UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Sergei Kuštšenko

Implementation of election bulletin board using HyperLedger Fabric

Bachelor's Thesis (9 ECTS)

Supervisor: Ivo Kubjas

Tartu 2020

Implementation of election bulletin board using HyperLedger Fabric

Abstract:

This thesis describes the architecture of an online voting system based on Gennaro Avitabile's [1] work. We consider technical aspects for implementing the voting system and based on the refined description we implement it as a microservice platform.

The implemented solution consists of different parts like voting application, key management application, vote collector service, election management service and bulletin board. The bulletin board is implemented as a permissioned blockchain by using HyperLedger Fabric.

Different testing methods were used to validate implemented solution against requirements. The system testing indicated that it is functional and practically usable. However, the authentication step in web applications should be moved from network to application layer to improve overall user experience. Performance testing showed that if an election has more than 8 choices, the voting process takes more than 10 seconds, which results in unsatisfying user experience.

As for the future, the system will be modified and stacked with different features to improve the overall experience and performance.

Keywords:

Permissioned blockchain, HyperLedger Fabric, internet voting

CERCS: T120 Systems engineering, computer technology

Valimiste teadetetahvli implementeerimine kasutades HyperLedger Fabric raamistikku

Lühikokkuvõte:

Käesolevas töös kirjeldatakse veebipõhise hääletussüsteemi arhitektuuri, mis põhineb Gennaro Avitabile [1] tööl. Me kaalume tehnilisi aspekte hääletussüsteemi rakendamiseks ja tuginedes täpsustatud kirjeldusele implementeerime seda mikroteenuste platvormina.

Implementeeritud lahendus koosneb erinevatest osadest nagu valijarakendus, kogumisteenus, võtmerakendus, valimise haldusteenus ja teadetetahvel. Teadetetahvel on HyperLedger Fabricu abiga rakendatud loalise plokiahela võrguna.

Rakendatud lahenduse valideerimiseks viisime läbi testimise kasutades erinevaid testimismetoodikaid. Süsteemne testimine näitas, et lahendus täidab funktsionaalseid nõudeid ning on praktiliselt kasutatav. Kuid veebirakenduste autentimise sammu võib tõsta võrgukihist rakenduskihti, et parendada üldist kasutajakogemust. Jõudlustestimine tuvastas, et kui valimisel on rohkem kui 8 valikut siis hääle andmine võtab rohkem kui

10 sekundit, luues kehva kasutajakogemuse.

Tulevikus süsteemi täiustatakse erineva funktsionaalsusega ning modifitseeritakse, selleks et muuta süsteemi jõudlust ja üldist kasutamist.

Võtmesõnad:

Loaline plokiahel, HyperLedger Fabric, netihääletus.

CERCS: T120 Süsteemitehnoloogia, arvutitehnoloogia

Contents

Acknowledgements

Development of the prototype was funded by European Union H2020 project PRIViLEDGE, grant number 780477.

1 Introduction

Internet voting (also called i-voting) is a system that gives a possibility to vote from a smart device or a computer via the Internet. Nowadays it becomes more popular as more countries, companies and other facilities are interested in using it as only or as an addition to traditional paper voting. There are several reasons for i-voting to be more preferable against paper voting:

1. **Accessibility:** More people can be part of the elections. For example, people who have some health issues or disabilities which restricts them from going out from home, this is an ideal option for them to vote from home with minimal physical stress or require them to vote by proxy.

2. **Availability:** To cast a vote voters don't have to go to the polling station, but rather can do it from home at the time which is suitable for them.

3. **Faster:** In paper voting, the voter must to go to the polling station and there he can encounter a waiting line to the voting booth. Online casting is much faster and can take minutes to finish because voters must only authenticate and select the desired choice from their computer or smart device.

4. **Cheaper:** Need fewer resources:

    - **People:** Fewer people must be involved in different election processes (organization, preparation, maintenance, audition).
    - **Space:** Does not require to reserve or rent different places for polling stations exists an only necessity of having a server room for the system.

5. **Automation:** Vote tallying works automatically which eliminates human error in the counting process.

However, because elections are typically used to give power to the candidate or an important decision making (for example referendum), i-voting system must be secure and transparent enough so it can provide trust. Creating such system has been challenging for companies and academic.

The goal of this thesis is to create a working prototype based on Gennaro Avitabile's [1] work which implements an End-to-End Verifiable i-voting system using permissioned blockchain platform such as HyperLedger Fabric. This would give voters and auditors a possibility to verify that vote registration and counting process were correctly performed.

The thesis consists of seven sections. Firstly, we define different functional and non-functional requirements for the system. The third section describes the high-level view of the cryptographic protocol. System architecture and used technologies are described in the fourth section. Usage is described in the fifth section. The sixth section contains the testing results of the implemented system. We conclude the thesis in the seventh section.

2 Requirements

2.1 Functional requirements

We describe different requirements for the desired system for different users:

- **Election Organizer** is a person who is responsible for election management and system initialization.
- **Voter** is a person who participates in elections by casting votes.
- **Auditor** is a person who audits election processes and participates in system initialization.

Table 1. Election Organizer requirements.

ID	Requirement
EA-1	The system must have a possibility to generate keys for vote encryption and decryption.
EA-2	The system must have a bulletin board.
EA-3	The system must have a possibility to publish votes to the bulletin board.
EA-4	The system must have a possibility to publish election configuration and results to the bulletin board.
EA-5	The system must have a possibility to validate published votes and election results.

The bulletin board necessity comes from the goal to create an End-to-End Verifiable system, therefore election-related parts (votes, results and election configurations) must be published somewhere for election audit.

Table 2. Voter requirements.

ID	Requirement
VOT-1	The system must have a possibility to cast vote.
VOT-2	The system must have a possibility for Voters to access the bulletin board.

Giving access to the bulletin board for Voters will increase the system transparency and improves trust.

Table 3. Auditor requirements.

ID	Requirement
AUD-1	The system must be set up with Auditor participation.
AUD-2	The system must have a possibility for votes, election configuration and election results verification.

Auditor participation must be not only procedural, meaning not only being able to view how the system is initialized and how different processes work, but he must actively participate in the system initialization. This will be considered as an additional trust level for the Voters and any observers of the election processes.

2.2 Non-functional requirements

We define non-functional requirements in Table 4 which should be implemented.

Table 4. Non-functional requirements for the system.

ID	Requirement
NF-1	The applications web interface should be simple and intuitive to use.
NF-2	The applications web interface should change its dimensions according to the screen resolution.
NF-3	The applications web interface should be mobile friendly.
NF-4	The vote casting process for voter should take no more than 5 seconds.

Most requirements are connected with the user interface because it is necessary to provide a good experience of web application usage and support different devices from which voters can cast vote. For example, if the voting application does not properly display the candidate list on a mobile device in the casting process, then voter could probably make a false choice or not make any.

As for the casting process time, if it takes a lot of time then voters may interrupt or cancel the process thinking the application is not working correctly or they are lacking time. This is important because for some voters online voting can be the only way to cast a vote. Therefore we should minimize the number of voters who would cancel the process.

3 End-to-End voting scheme

As the protocol is cryptographically involved we first describe the cryptographic protocol. The cryptographic protocol for casting the ballot is a slightly modified Commitment Consistent Encryption scheme (CCE) which was introduced by Oliver Pereira [2]. CCE scheme gives the possibility to implement End-to-End Verifiable elections by publishing some parts of encryption to a bulletin board where anyone can verify the election process and results by using these published parts.

There are different ways how to encode voter choice. Generally in voting schemes voter selects a candidate from the candidate list and creates one ciphertext which contains the name or ID of the candidate, but CCE uses vector-based approach. This means that vote is represented as a vector of zeroes and ones. For example, in the election with 5 candidates, where voter votes for a fourth candidate, the choice vector C is defined as:

$$C = (0,\ 0,\ 0,\ 1,\ 0)$$

where number 1 represents the chosen candidate and 0 is not. Given the choice vector C and c_i is a i-th candidate choice, the encryption is:

$$E = (Enc(c_1),\ Enc(c_2), Enc(c_3),\ Enc(c_4),\ Enc(c_5))$$
$$\Updownarrow$$
$$E = (Enc(0),\ Enc(0), Enc(0),\ Enc(1),\ Enc(0))$$

and E is encrypted vote vector.

Additionally, it is possible to extract a perfectly hiding commitment and an opening from the ciphertext. Commitment is essentially a voter confirmation to a specific choice which is used for vote verification and decryption. The opening is an arbitrary value which combined with commitment and vote can verify that commitment opens to this particular vote.

The vote being a vector of numbers brings up problems like a possibility to put 100 or -10 for the candidate. To prevent it the voter needs to prove that vote contains only zeroes and ones. This can be done by constructing zero-knowledge proofs. In zero-knowledge proofs, a prover P proves to verifier V that he knows a value x without revealing x content to the verifier. Zero-Knowledge proofs must satisfy three properties [1, 3]:

- **Completeness**: proof should convince the V that P knows what they say they know.
- **Soundness**: if proof is false, V does not accept P.
- **Zero-knowledge**: if proof is correct, then V learns only that it is correct and nothing more.

In our case voter uses zero-knowledge proofs to prove to Election Organizer that in vote vector E each commitment e_i contains 0 or 1:

$$\text{Zero-One proof} = (e_1 \text{ is encryption of 0 or 1},\ e_2 \text{ is encryption of 0 or 1}, \dots)$$

and Election Organizer should only know that this statement is true or not.

Unfortunately, there is a possibility of voter giving a 0 for each candidate or 1 for each candidate:

$$C = (0,\ 0,\ 0,\ 0,\ 0) \qquad \text{or} \qquad C = (1,\ 1,\ 1,\ 1,\ 1)$$

To counter that voter must provide additional proof that the vote vector sum is up to 1:

$$E = (e_1,\ e_2, e_3,\ e_4,\ e_5)$$
$$\text{Sum proof} = e_1\ +\ e_2 + e_3\ +\ e_4\ +\ e_5 = 1$$

where e_i is a encryption of i-th choice.

Another benefit holding vote as a binary vector is that in the tallying process it is possible to aggregate all votes by combining ciphertexts for each candidate without decrypting them individually which in the end produces one vector with counted votes for each candidate and Election Organizer needs to decrypt only this one vector to receive election results (see Figure 1).

$$\begin{array}{ccc}
E_1 = (Enc(0),\ Enc(0), Enc(1),\ Enc(0),\ Enc(0)) & & V_1 = (0,\ 0, 1,\ 0,\ 0) \\
+ & & + \\
E_2 = (Enc(0),\ Enc(1), Enc(0),\ Enc(0),\ Enc(0)) & & V_2 = (0,\ 1, 0,\ 0,\ 0) \\
+ & \Leftrightarrow & + \\
E_3 = (Enc(0),\ Enc(1), Enc(0),\ Enc(0),\ Enc(0)) & & V_3 = (0,\ 1, 0,\ 0,\ 0) \\
= & & = \\
E_{result} = (Enc(0),\ Enc(2), Enc(1),\ Enc(0),\ Enc(0)) & & V_{result} = (0,\ 2, 1,\ 0,\ 0)
\end{array}$$

Figure 1. Demonstration of votes aggregation.

On the left side (see Figure 1) encrypted votes are provided and the right side represents underlying data behind the ciphertext. The aggregation process that is shown in Figure 1 is possible due to CCE being a homomorphic scheme. From $E_{results}$ vector Election Organizer decrypts results and extracts an opening for later election results verification.

As mentioned before the CCE scheme gives a possibility to create End-to-End Verifiable elections. To do that the Voter and Election Organizer publish some parts of election processes to the bulletin board. Voter public part on bulletin board consists of:

- voter certificate;
- commitments for each candidate;
- commitments signature;
- proofs that commitments are encryption of 0 or 1;
- proof that commitments are sum of encrypted values which sum up to 1.

Election Organizer public part on bulletin board consists of [1]:

- election configuration;
- election results;
- results openings.

Bulletin board gives a possibility to anyone make different verifications [1, 4]:

- **Individual verification** means that voter can verify that vote was:
  - **Cast as intended**: ballot represents a vote for the candidate whom he or she intended to give the vote.
  - **Recorded as cast**: ballot is recorded as he or she cast it.
- **Universal verification** means that anyone can verify that:
  - Every vote was cast by an eligible voter.
  - Tally process is done correctly by verifying the results.

The commitments are rerandomized by the Election Organizer before they are published to the bulletin board. Rerandomization means that we change commitments ciphertext without changing the voting intention inside it. Reason for rerandomization is to achieve receipt-freeness in this scheme. This removes the possibility of vote-buying or coercion because after these commitments are rerandomized voter can not show to coercer or someone else how he or she voted.

Subsections below will give more detailed information about different parts of the scheme.

3.1 Commitment Consistent Encryption

We use asymmetric bilinear groups and assume the existence of a group generator `GrpGen` which on input 1^λ outputs

$$\mathcal{G} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, h_1, h_2, e, p)$$

where [1]:

- p is a prime of length λ;
- $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are cyclic groups of order p;
- g_1 is a generator of $\mathbb{G}_1$;
- h_1, h_2 are generators of $\mathbb{G}_2$ and there is no known y such that $h_2 = h_1^y$;
- e is a bilinear map (pairing) $\mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ such that $e(g_1, h_1)$ generates $\mathbb{G}_T$. Furthermore, e is a Type-3 pairing, meaning that $\mathbb{G}_1 \neq \mathbb{G}_2$ and there does not exist an efficient isomorphism $\psi(\mathbb{G}_1) \rightarrow \mathbb{G}_2$.

These cyclic groups are based on elliptic curves. In our implementation, we use a pairing-friendly curve BLS12-461. Initially, Gennaro [1] used BLS12-381, but unfortunately recent articles from Yonezawa S. et *al.* [5] claims that BLS12-381 does not achieve 128-bit security, therefore based on the article of Aurore Guillevic et *al.* [6] it was decided to choose a BLS12-461 which achieves about 134-bit security.
CCE scheme description follows as [1, 2]:

- *GenerateKeys*$(\mathcal{G})$: Generate an ElGamal encryption key $g_2 = g_1^x$, with $x \leftarrow \mathbb{Z}_p$ Return $(pk = g_2, sk = x)$, where:
  - pk is a public key;
  - sk is a private key (or secret key).
- *EncryptVote*(v, pk): Return $E = (c_1, c_2, d, \sigma_{cc}) = (g_1^s, g_1^r g_2^s, h_1^r h_2^v, \sigma_{cc})$ with $(r, s) \leftarrow \mathbb{Z}_p^2$ and σ_{cc} as consistency proof;
- *DecryptVote*(E, sk): Extract the discrete logarithm of $e(\frac{c_1^x}{c_2}, h_1)\ e(g_1, d)$ in basis $e(g_1, h_2)$;
- *ExtractCommitment*(E): Return d;
- *ExtractOpening*(E, sk): Return $a = \frac{c_2}{c_1^x}$;

- *VerifyOpening*(d, a, v): Check that $e(a, h_1) = e(g_1, \frac{d}{h_2^v})$.

To be sure that vote contains a valid commitment, consistency proof σ_{cc} is provided as well, so for the triple $(c_1, c_2, d) = (g_1^s, g_1^r g_2^s, h_1^r h_2^v)$ computation of proof is done as follows [1, 2]:

1. *Commitment computation*: $c' = (c'_1, c'_2, d') = (g_1^{s'}, g_1^{r'} g_2^{s'}, h_1^{r'} h_2^{v'})$ with $r', s', v' \leftarrow \mathbb{Z}_p^3$.
2. *Challenge computation*: $e = H(c, c', \textit{electionID})$ where H is a hash function.
3. *Response computation*: $f_r = r' + er$, $f_s = s' + es$, $f_v = v' + ev$.

Therefore consistency proof defined as $\sigma_{cc} = (e, f_r, f_s, f_v)$ and verification process of the proof σ_{cc} follows as [1, 2]:

1. *Commitment reconstruction*: compute $c' = (c'_1, c'_2, d')$ with $c'_1 = \frac{g_1^{f_s}}{c_1^e}$, $c'_2 = \frac{g_1^{f_r} g_2^{f_s}}{c_2^e}$, $d' = \frac{h_1^{f_r} h_2^{f_v}}{d^e}$
2. *Validity verification*: check that $e = H(c, c', \textit{electionID})$

3.2 Commitment rerandomization

As it was mentioned, commitments are rerandomized before they are published to the bulletin board to achieve receipt-freeness. Rerandomization is done by Election Organizer and it consists of adding additional randomness to the commitment [1].

Initially, voter creates commitments as:

$$d = h_1^r h_2^v, \text{ with } r \leftarrow \mathbb{Z}_p \text{ and } v \in \{0, 1\}$$

where r is voter randomness and v is voter choice. Election Organizer modifies it by adding his own randomness s [1]:

$$d' = d h_1^s = h_1^{r+s} h_2^v, \text{ with } s \leftarrow \mathbb{Z}_p \text{ and } v \in \{0, 1\}$$

Rerandomized commitment still contains the same voting intention but the voter can not prove it to coercer as he does not know $r + s$ [1, 2].

Additionally Election Organizer should provide to the voter a rerandomization proof σ_{rand}, that the rerandomization is done honestly. The creation of σ_{rand} is done as [1, 2]:

- *Commitment computation*: with $l_1, l_2, w, z_1, z_2 \leftarrow \mathbb{Z}_p$, compute $a_1 = h_1^{z_1} d^{-l_1}$, $a_2 = h_1^{z_2} (\frac{d}{h_2})^{-l_2}$, $a_3 = h_1^w$.

- *Challenge computation*: compute $l = H(d, d', a_1, a_2, a_3)$ and $l_3 = l - l_1 - l_2$, where H is a hash function.
- *Response computation*: compute $z_3 = w + sl_3$.

Then rerandomization proof provided to voter as $\sigma_{rand} = (l_1, l_2, l_3, z_1, z_2, z_3)$. For voter to verify that rerandomization is honest he needs to do following steps [1]:

1. *Reconstruct the commitment*: compute $a_1 = h_1^{z_1} d^{-l_1}$, $a_2 = h_1^{z_2} (\frac{d}{h_2})^{-l_2}$ and $a_3 = h_1^{z_3} (\frac{d'}{d})^{-l_3}$.
2. *Recreate challenge*: $l = H(d, d', a_1, a_2, a_3)$.
3. *Verify that*: $l = l_1 + l_2 + l_3$.

As a remark, during the development of the system rerandomization part was modified by the protocol author, but unfortunately, it was not possible to change it inside the code because of some technical problems. The modification was related with the inclusion of voter signing key pair to the rerandomization proof. Therefore to prove to the voter that rerandomization was done correctly Election Organizer prove the knowledge of the discrete logarithm of:

$$v_{pk} = g^{v_{sk}} \text{ or } \frac{d'}{d}$$

where v_{pk}, v_{sk} are voter public and private ECDSA signature keys and g is a generator of ECDSA signature scheme. This modification is more preferred because the Election Organizer does not know voter private key v_{sk} which means that this side $g^{v_{sk}}$ of OR operation will always fail. However, as the voter knows the private key, he could construct the proof to the coercer without knowing $r + s$. Therefore Election Organizer can only provide a valid proof for another side of the OR operation which is a proof of honest rerandomization. Modified rerandomization process follows as:

- *Commitment computation*: $a_1 = g^{z_1} v_{pk}^{-l_1}$, $a_2 = h_1^w$ with $l_1, w, z_1 \leftarrow \mathbb{Z}_p$.
- *Challenge computation*: $l = H(d, d', v_{pk}, a_1, a_2, \textit{electionID}),\ l_2 = l \oplus l_1$.
- *Response computation*: $z_2 = w + sl_2$.

The rerandomization proof is then defined as $\sigma_{rand} = (l_1, l_2, z_1, z_2)$ and voter can verify it by:

- *Commitment reconstruction*: $a_1 = g^{z_1} v_{pk}^{-l_1}$, $a_2 = h_1^{z_2} (\frac{d'}{d})^{-l_2}$.
- *Challenge computation*: $l = H(d, d', a_1, a_2, \textit{electionID})$.
- *Verify that*: $l = l_1 \oplus l_2$.

3.3 Well-formedness proofs

3.3.1 Proving that a commitment is to either 0 or 1

The proof itself is a disjunctive proof of knowledge of the discrete logarithm of d' or $\frac{d'}{h_2}$. Therefore for each commitment $d = h_1^r h_2^v$ Zero-One proof $\sigma_{0/1}$ is provided. The proof creation starts from proof commitments $a^1_{0/1}, a^2_{0/1}$ computation by voter:

$$\begin{array}{cc} \text{If } v = 0 & \text{If } v = 1 \\ (z_2, k_2, w_1) \leftarrow \mathbb{Z}_p^3 & (z_1, k_1, w_2) \leftarrow \mathbb{Z}_p^3 \\ a^1_{0/1} = h_1^{w_1},\ a^2_{0/1} = h_1^{z_2}(\frac{d}{h_2})^{-k_2} & a^1_{0/1} = h_1^{z_1}\ d^{-k_1},\ a^2_{0/1} = h_1^{w_2} \end{array}$$

then commitments $a^1_{0/1}, a^2_{0/1}$ are sent to Election organizer for rerandomization with his randomness s:

$$\begin{gathered} (s, s_1, s_2) \leftarrow \mathbb{Z}_p^3 \\ d' = dh_1^s \\ a^{1'}_{0/1} = h_1^{s_1}\ a^1_{0/1},\ a^{2'}_{0/1} = h_1^{s_2} a^2_{0/1} \end{gathered}$$

Rerandomized commitments are then sent back to voter for challenge k and response z creation:

$$k = H(d', a^{1'}_{0/1}, a^{2'}_{0/1}, \textit{electionID})$$

$$\begin{array}{cc} \text{If } v = 0 & \text{If } v = 1 \\ k_1 = k \oplus k_2,\ z_1 = w_1 + rk_1 & k_2 = k \oplus k_1,\ z_2 = w_2 + rk_2 \end{array}$$

Voter then sends $\sigma_{0/1} = (k_1, k_2, z_1, z_2)$ to Election Organizer for finalization. Election Organizer finalizes Zero-One proof $\sigma'_{0/1}$ creation by adding randomness s to respone z [1]:

$$\begin{gathered} z'_1 = z_1 + s_1 + sk_1 \\ z'_2 = z_2 + s_2 + sk_2 \\ \sigma'_{0/1} = (k_1, k_2, z'_1, z'_2) \end{gathered}$$

Given vote commitment d' and Zero-One proof $\sigma'_{0/1}$ verification can be done by anyone and it comes as follows [1]:

- *Commitment reconstruction*: $a^{1'}_{0/1} = h^{z'_1} d'^{-k_1},\ a^{2'}_{0/1} = h_1^{z'_2}(\frac{d'}{h_2})^{-k_2}$
- *Challenge computation*: $k = H(d', a^{1'}_{0/1}, a^{2'}_{0/1}, \textit{electionID})$
- *Verify that*: $k = k_1 \oplus k_2,$ where $k_1, k_2 \in \sigma'_{0/1}$

3.3.2 Proving that the sum of committed values is equal to 1

Given that exists n commitments d', the proof is a proof of knowledge of the discrete logarithm of:

$$D = \prod_{i=1}^{n} d'_i h_2^{-1}$$

Initially, voter creates commitments d for each choice i:

$$d_i = h_1^{r_i} h_2^{v_i}, \text{ with } r \leftarrow \mathbb{Z}_p, \ v \in \{0, 1\}$$

and sum proof commitment a_Σ:

$$a_\Sigma = h_1^{w}, \text{ with } w \leftarrow \mathbb{Z}_p$$

Then sends (d, a_Σ) to Election Organizer for further rerandomization. Rerandomization is done with randomness s addition:

$$d'_i = h_1^{s_i} d_i \text{ with } s_i \leftarrow \mathbb{Z}_p \text{ and } a'_\Sigma = h_i^{s} a_\Sigma \text{ with } s \leftarrow \mathbb{Z}_p$$

Commitments (d', a'_Σ) are then sent back to Voter for challenge k and response z computation:

$$\begin{aligned} D &= \prod_{i=1}^{n} d'_i h_2^{-1} \\ k &= H(D, a'_\Sigma, \textit{electionID}) \\ z &= w + k \sum_{i=1}^{n} r_i \end{aligned}$$

The sum proof is defined as $\sigma_\Sigma = (k, z)$ and sent to Election Organizer for process finalization. The final step is adding Election Organizer randomness to response z:

$$\begin{aligned} z' &= z + s + k \sum_{i=1}^{n} s_i \\ \sigma'_\Sigma &= (k, z') \end{aligned}$$

The verification process can be done by anyone and it follows as:

- *Commitment reconstruction*: $D = \prod_{i=1}^{n} d'_i h_2^{-1}, \ a'_\Sigma = h_1^{z} D^{-k}$
- *Verify that*: $k = H(D, a'_\Sigma, \textit{electionID})$

3.4 Tally

The tally computation consists of different steps which are made by Election Organizer. The tally process consists of different operations [1]:

1. Element wise multiplication of all CCE vectors $E_1, E_2, E_3, \ldots, E_i, \ldots, E_n$, obtaining one vector single CCE vector $E_{results}$ with encrypted results.
2. Element wise multiplication of all rerandomization vectors b, which consists of rerandomization factors $b_i = g_1^{s_i}$, where s_i is the randomization value used in commitment d rerandomization, obtaining one single vector $b_{combined}$.
3. Openings extraction from $E_{results}$ as $o_{combined}$.
4. Decryption of election results from $E_{results}$ obtaining vector results $v_{results}$, where v_i represents results for i-th choice.
5. Element wise multiplication of $b_{combined}$ and $o_{combined}$, obtaining opening vector $o_{results}$.

Finally v_{result} and o_{result} are published to the bulletin board by Election Organizer where anyone can verify the decryption process.

The verification process of the published results follows as:

1. Published commitments $d'_1, d'_2, d'_3, \ldots, d'_i, \ldots, d'_n$ are aggregated into $d'_{results}$.
2. Verify that commitments $d'_{resutls}$ are opening to $v_{results}$ with openings $o_{results}$.

4 Implementation

This section consists of two subsections. Used technologies briefly describe what technologies were used and why they were chosen in a prototype implementation. Architecture describes different components of the system and how they interact with each other.

4.1 Used technologies and libraries

Before we proceed to the description of selected tools it is good to note, that the stack of used technologies in this thesis can be changed, but not so much.

For front-end development, Angular framework was selected, but it can be changed to other libraries or frameworks based on JavaScript such as React or Vue.js. Reason being that in the i-voting example it is necessary to create a single-page application (SPA) for client applications. SPA has a lot of benefits [7]:

- **Faster:** The application needs to be loaded only once from the server with necessary HTML, CSS and script files.
- **Smooth experience:** It creates a feeling of a desktop application to the user, because of the smooth flow of processes.
- **Transparency:** SPA gives the possibility to inspect the application logic.

For back-end development the number of languages is a little bit smaller, the reason being the HyperLedger Fabric. To interact with it we need to use a Fabric SDK in the client application and the languages that support it are Java, Go and Node.js.

4.1.1 Go

Go is an open-source programming language, which was firstly designed at Google in 2007 for internal needs only but then it was decided to release new language to the public in 2009. The main reason why authors started developing was motivated by a shared dislike of C++ and its complexity. As well they wished for an easier language for development [8, 9]. The main goal while creating Go was to create it as simple as possible and without extra features which are not gonna be used [8].

Go language is used mainly in server-side applications because its ideology is created around packages and services. It gives a possibility to split one big application into multiple microservices because of simple concurrency.

Reasons why it was selected for this project [10]:

- **garbage collector** - Go can manage memory by itself: removing old pointers, prevents memory leaks;

- **strict code style** - Go has integrated code formatter which makes code looks almost the same across different projects, at the same time it will throw an error on compilation if some variable is not used because efficiency is the high priority;
- **compiling** - Go compiles itself into machine code.

4.1.2 Angular

Angular is a front-end open-source framework which uses HTML and Typescript as main programming languages. Originally released as an AngularJS in 2010 by Google to fulfil the purpose of developing a single-page application (SPA). Initially, AngularJS used JavaScript as a programming language but starting from Angular version 2 (released in 2016) language choice shifted to TypeScript which introduced more object-oriented programming to a front-end application [11].

Reasons why it was selected:

- **ease of development** – different tools inside the framework makes development smooth;
- **compiling** – because TypeScript compiles itself into a JavaScript we can catch some errors before we start using the application.

4.1.3 MIRACL Core Cryptographic Library

This library is an extended and re-released version of Apache Milagro Cryptographic Library (AMCL) which supports elliptic curve and pairing-friendly curve cryptography as well RSA, AES symmetric encryption and hash functions [12]. It has implementations in different languages including the ones that are necessary for this project - Go and JavaScript.

4.1.4 Docker

Docker is a tool which gives a possibility to create, deploy and run applications by using containers. These containers allow to package an application with all necessary parts (libraries, tools, dependencies files). In some way, Docker is like a virtual machine but it does not create a new operating system, it uses a host Linux kernel, which gives higher performance than virtual machine [13]. Another benefit is that it eliminates a known problem where an application runs on one computer but on other, it can exit with error (also known as *It works on my machine*).

4.1.5 HyperLedger Fabric

HyperLedger Fabric is an open-source permissioned distributed ledger technology platform, designed to provide trust between different organizations. Because the platform is permissioned it means, that all participants inside the network know each other and agree on who, how and what can do inside the ledger [14]. Figure 2 shows the general structure of HyperLedger Fabric:

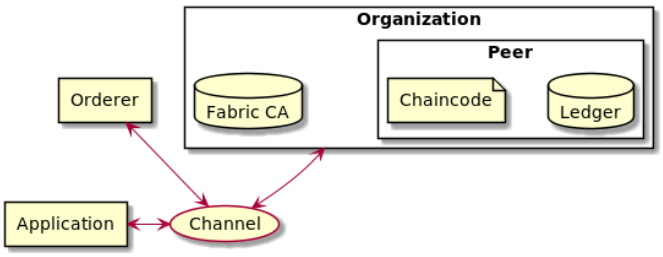


Figure 2. HyperLedger Fabric main components.

where [1, 14]:

- **Organization** is a member of the blockchain network.
- **Application** is a client that uses Fabric SDK to submit transactions to peers and transaction proposals to orderers, also it queries ledger state.
- **Peer** is a node that hosts ledgers and chaincode. The number of peers inside the organization is not limited only to one, it can be as many as the organization wants. Running the organization with only one peer is not a good choice, because if this peer not functioning correctly, then the whole organization is not functional.
- **Ledger** consists of two parts:
  - **World state** is a database that holds the current state of set of ledger values.
  - **Blockchain** is a transaction log that contains all records and changes that resulted in the world state. Blockchain gives immutability to history, means that once something was changed in the ledger, which resulted in the current world state, it is impossible to somehow remove or delete log about this modification.
- **Chaincode** or smart contract is a specific code invoked by a client application which manages access and modifications to a set of key-value pairs in the world state.

- **Orderer** is a node that creates transaction blocks for ledger modification and maintains the list of organizations that are allowed to create channels.
- **Channel** is a private "subnet" for communication between different organizations, peers, orderers and applications. Each channel holds own ledger and all channel members can see all transactions that are made in this channel.
- **Fabric CA** is an optional Certificate Authority for HyperLedger Fabric. Which is responsible for identity creation with different roles like peer, orderer, client etc.

Fabric uses new architecture for transactions that is called execute-order-validate [15]:

- **Execute:** Transaction is executed on peer using chaincode, which returns *Read* and *Write* sets, where *Read* set consists of key-value pairs before the modification and *Write* set consists of key-value pairs after the modification.
- **Order:** When enough peers (the amount of peers that must execute the chaincode is defined in the endorsement policy on chaincode instantiation) agree on execution, meaning they have similar results (*Read* and *Write* sets) transaction is being ordered.
- **Validate:** Each peer validates transaction before applying it to the ledger.

This architecture has different benefits, but the most important ones:

- It removes Double-Spending problem, where for example the same digital currency could be spent more than once [16].
- Eliminates any non-determinism because all logic (chaincode execution) is done before ordering.

More detailed transaction flow can be seen in Figure 3.

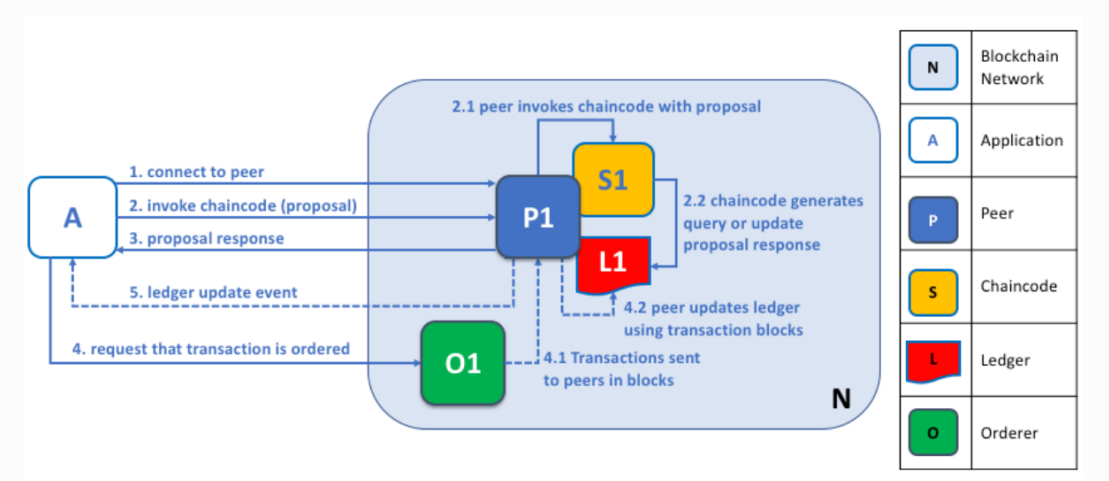


Figure 3. Transaction process diagram [17].

HyperLedger Fabric is a very complex platform, where individual pieces are modular - for example, the orderer can be implemented on different managing methods, chaincode as Go, JavaScript, Java code etc. For complete description including the modules, we refer the reading to HyperLedger Fabric homepage[1].

4.2 Architecture

This section describes the whole system with details. The system is split into different parts where each part is responsible for specific actions. Figure 4 provides system general structure with actors.

[1] https://www.hyperledger.org/projects/fabric

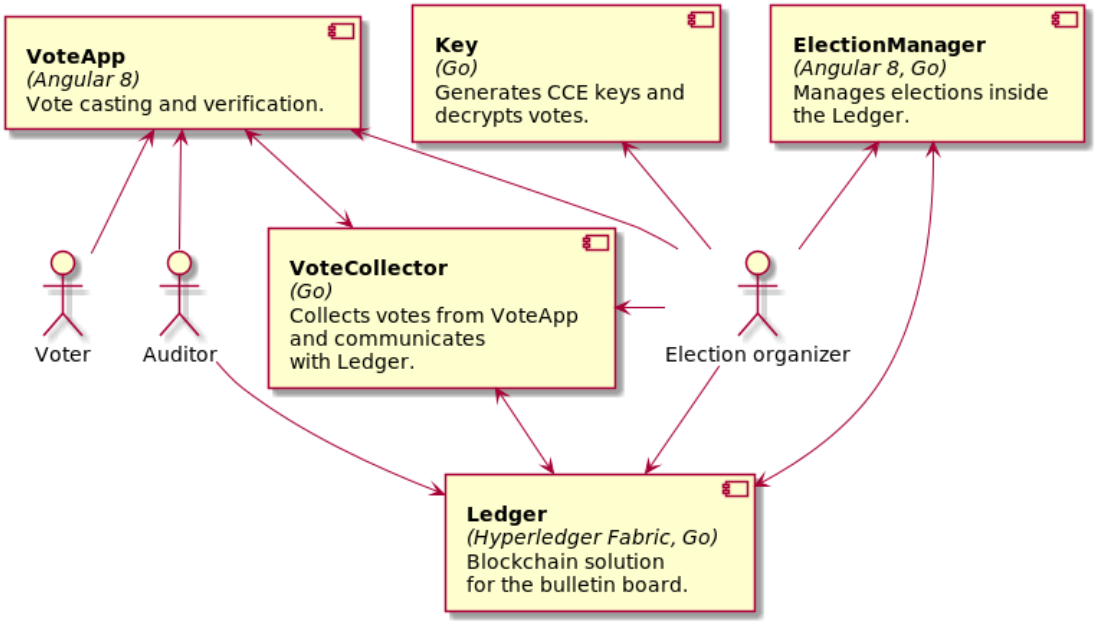


Figure 4. System general structure.

Next subsections will give more information about different parts of the system, they are divided into client and server-side applications. Client-side means that application is run on users local machine (computer, mobile, tablet). Whereas server-side means that action and logic are run on a web server.

4.2.1 Client-side

4.2.1.1 Voter application

Voter application or **VoteApp** gives a possibility to cast vote and verify election processes, where main possibilities for verification are:

- election configuration check;
- voter certificate check;
- voter signature check;
- Zero-One and Sum proof check of published commitments;
- election results verification.

VoteApp is a single-page application (SPA) which gives the possibility for a voter to see how the vote casting and verification logic is executed.

4.2.1.2 Election manager

Election manager (ElectionManager) is an application which is responsible for election management. It gives a possibility to add and delete elections as well publish election results. ElectionManager consists of two applications:

- client - single-page application (SPA) for Election Organizer interaction;
- server - which is a proxy between ElectionManager client application and Ledger.

Initially, it was not decided how complex ElectionManager will be, therefore it was decided to split ElectionManager to a client (SPA) and server-side applications, as one of the possible future features would be client-side election configuration signing without sending the signing private keys to the server. This would allow to separate the duties of technical and administrative parts, where the technical administrators are not able to modify election configuration.

4.2.2 Server-side

4.2.2.1 Vote collector

Vote collector or **VoteCollector** serves as a REST API. It holds different responsibilities like:

- communication between VoteApp and Ledger network;
- vote casting (ballot formation with VoteApp);
- vote parts publication to a bulletin board inside the Ledger;
- saving votes locally;

After votes are cast they are saved locally in VoteCollector folder (see Figure 5).

```
votecollector
votes
    home2020
        10000000000
            vote~2020-04-07T08:12:43Z.json
        10000000001
            vote~2020-04-07T08:47:40Z.json
        10000000002
            vote~2020-04-07T08:07:11Z.json
```

Figure 5. VoteCollector folder structure with saved votes from the casting process.

4.2.2.2 Blockchain network

Blockchain network (**Ledger**) serves as a bulletin board where public records of votes, election configuration and results are being published. Network topology can be seen in Figure 6.

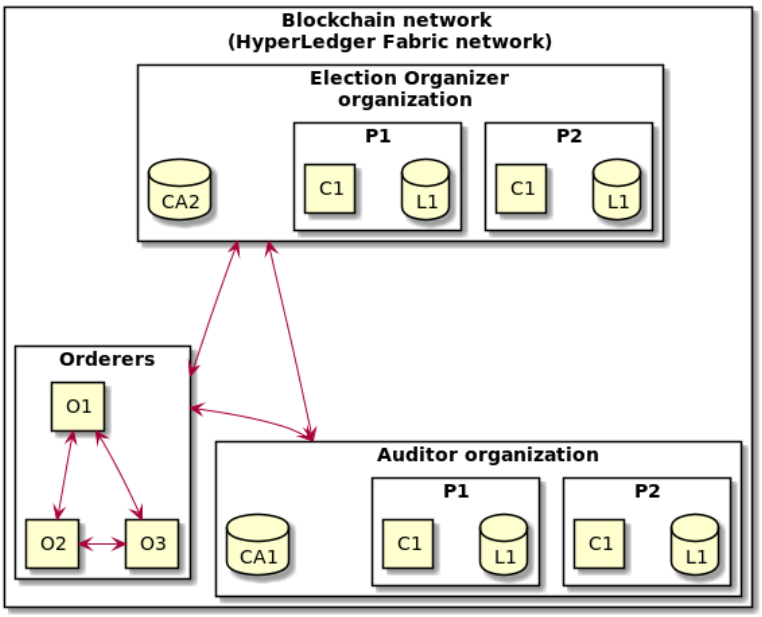


Figure 6. Blockchain network topology.

Figure 6 shows that both organizations have almost the same structure. This means that they have the same chaincode (business logic) C1 and ledger state L1. The difference comes from the fact that each organization controls it's own Fabric Certificate Authority which is shown as CA1 and CA2 for identity creation. Election Organizer uses CA2 to create an identity for VoteCollector and ElectionManager to give a possibility for these applications to interact with Ledger. Auditor organization in the current implementation is not using CA1 for own purposes, but this may be changed in future.

Both organizations have two peers (shown as P1 and P2 in Figure 6) for additional redundancy, which means that if one of two peers goes down inside an organization, the organization will still be functional.

Additionally, the network has a *MAJORITY* (greater than half) rule. This means that in our case both organizations should accept this action to add or modify the ledger, but this rule can be changed. For example, it can be configured in such a way that only Election Organizer or Auditor organization should accept these actions or set that all organizations should accept it.

Orderers that are shown in Figure 6 are running in Raft protocol which gives the possibility for making the transaction even if one of the orderers goes down.

4.2.3 Command-line interface applications

4.2.3.1 Key application

A Key application is responsible for CCE keys generation and votes decryption from VoteCollector.

For CCE key pair generation it takes as an argument election ID, where election ID defines in what folder generated keys will be saved. The folder structure can be seen in Figure 7.

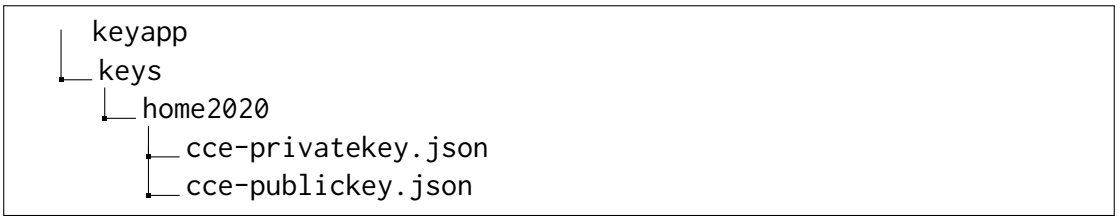


Figure 7. Key application folder structure after a key pair is generated.

To get results for election Key application takes as an input election ID, path to key pair and path to votes folder. Firstly, it finds the latest vote for each voter and then aggregates them to create one vote vector. When the aggregation process is finished it decrypts and extracts openings from result vector. Then results are saved to JSON file with corresponding election ID. Folder structure after the decryption process can be seen in Figure 8.

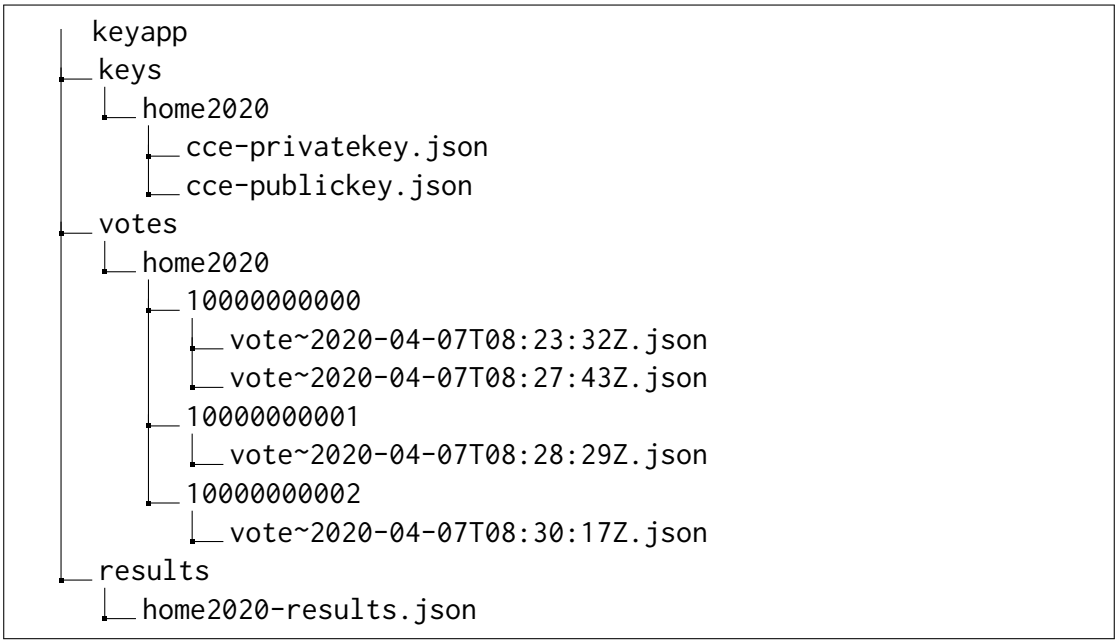


Figure 8. Key application folder structure after decryption for election with "home2020" ID.

4.2.4 System communication

This subsection will tell more about how different components interact with each other. It will contain different sequence diagrams with a high-level description.

4.2.4.1 VoteApp with VoteCollector

Authentication.
Figure 9 describes the accepting flow of the authentication process.

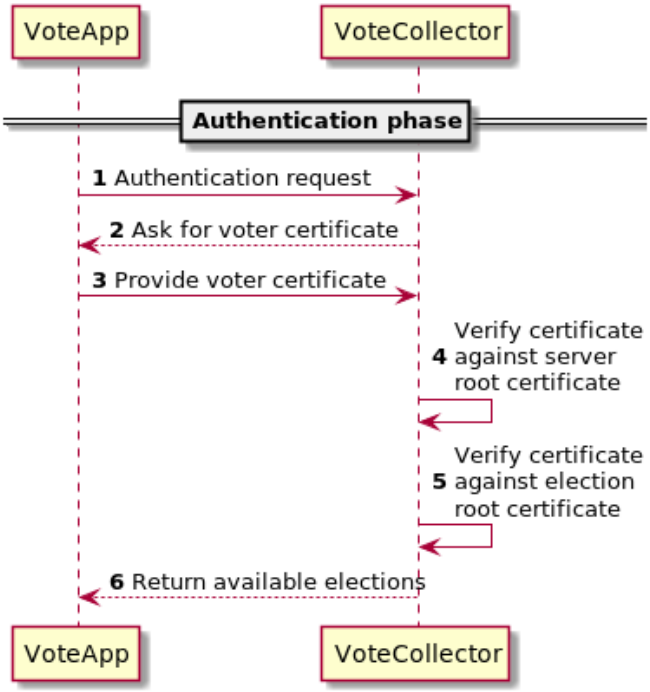


Figure 9. Sequence diagram of authentication phase between VoteApp and VoteCollector.

Authentication phase may fail in step 4 with different scenarios:

- voter did not provide a certificate and VoteCollector will close the connection with error;
- voter provided a certificate but it does not verify with VoteCollector configured root certificate.

On step 5 VoteCollector iterates through all cached election from Ledger and checks if this election is available for this voter. Availability means that voter certificate was issued from root certificate in election configuration.

Vote casting.
After successful authentication voter can cast a vote. Figure 10 shows an interactive casting process between VoteApp and VoteCollector.

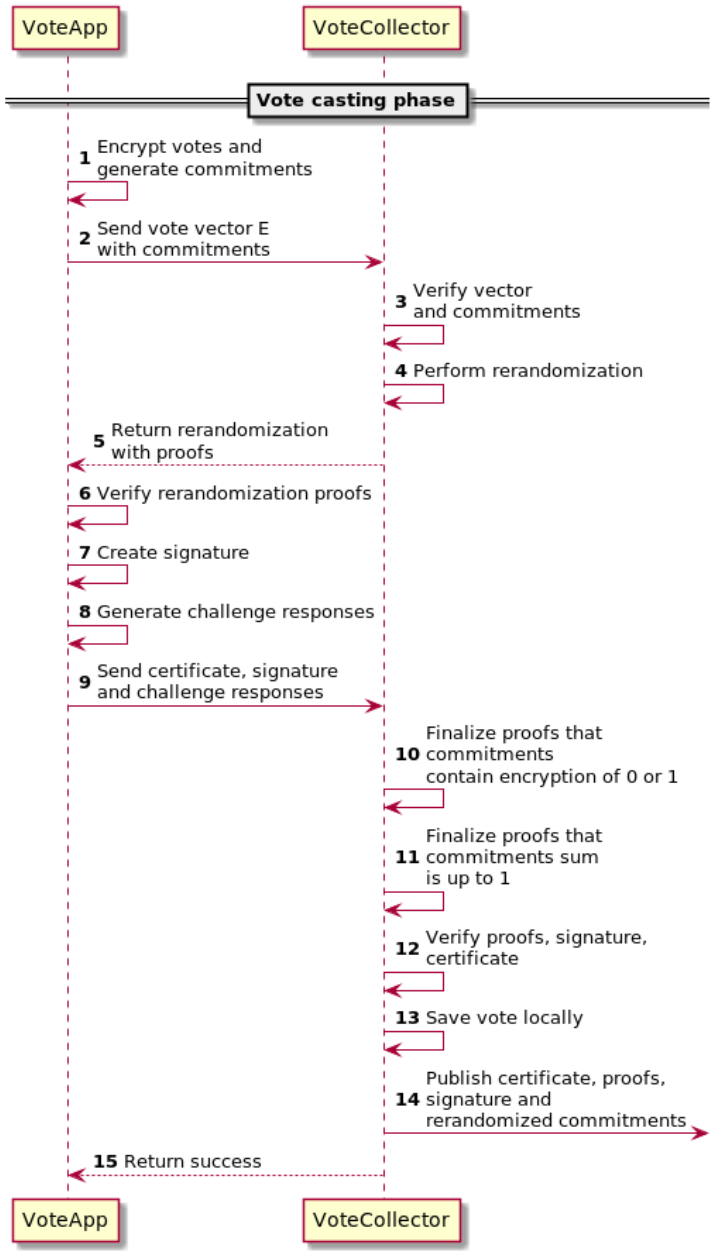


Figure 10. Sequence diagram of vote casting phase between VoteApp and VoteCollector.

Casting phase contains a lot of verifications between requests which means it can be stopped by VoteApp or VoteCollector. Here is the list of checks and possible points of failure:

- on step 3 VoteCollector finds out that commitments are not in the expected format

or contains something different which triggers VoteCollector to return an error;

- on step 6 verification of rerandomization proof fails and VoteApp stops vote casting process. This failure means that VoteCollector made such rerandomization that he changed the content of initially provided commitments;
- on step 12 verification fails which resolves in casting process failure. Verifications that are done:
  - check that received signature verifies with received commitments;
  - check that voter certificate was issued from election configuration root certificate;
  - check that each commitment contains encryption of 0 or 1;
  - check that commitment sum is up to 1.
- on step 13 if VoteCollector can not save vote locally it will stop casting process with failure;
- on step 14 if public record addition to Ledger fails, the casting process is interrupted and saved local vote on step 13 is deleted as well.

Audit data query.
Voting verification processes can be made by anyone so the authentication step is skipped.

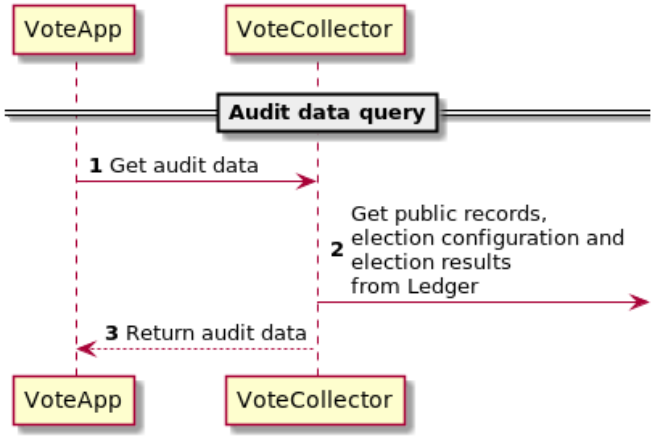


Figure 11. Sequence diagram of the audit data query between VoteApp and VoteCollector.

In this phase, failure can occur only if VoteCollector or Ledger are not working. In case if there are no elections inside the Ledger response will be “null”. Actual verifications are done inside the VoteApp on user demand (see Section 5.2).

4.2.4.2 VoteCollector with Ledger

VoteCollector authentication
Authentication inside VoteCollector is added for identifying if the current VoteCollector can send any transactions proposals to the Ledger.

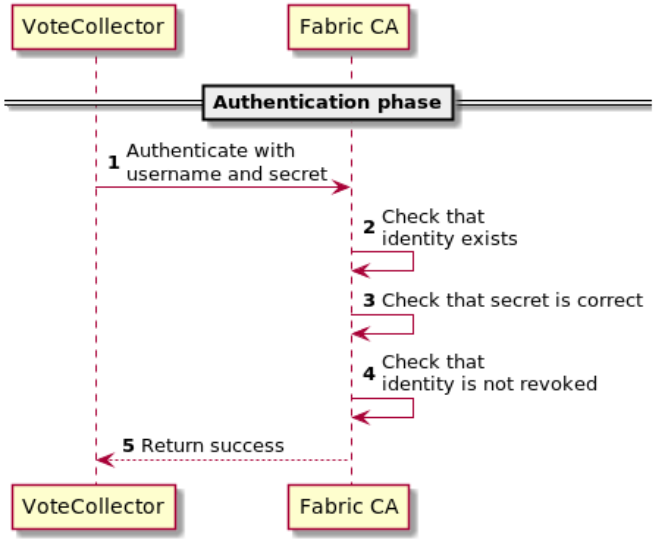


Figure 12. Sequence diagram of the authentication phase between VoteCollector and Fabric CA.

In current implementation VoteCollector is controlled by Election Organizer organization, therefore authentication is done to it's CA.

VoteCollector initialization
Figure 13 provides VoteCollector initialization flow.

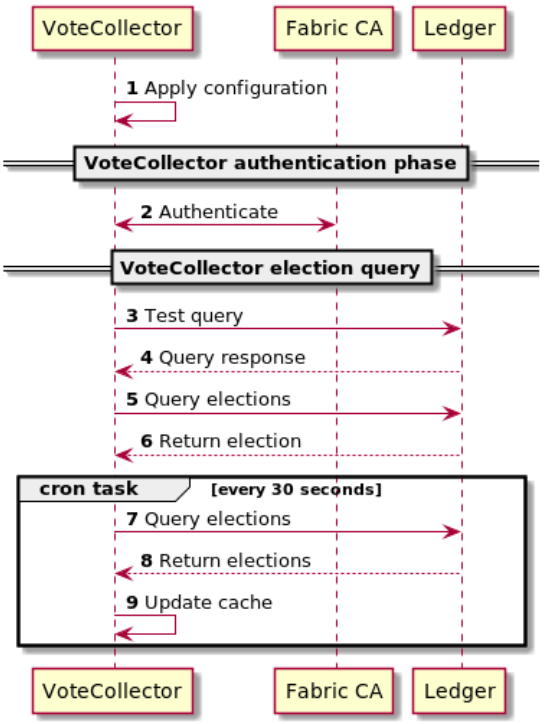


Figure 13. Sequence diagram of VoteCollector initialization.

Configuration initialization is shown as a separate step because at that time Fabric SDK (specific library for HyperLedger Fabric network interaction) initialization occurs which makes some additional requests to Ledger for connectivity check. On step 4 we make our test query which is a query of “TEST” key inside the Ledger which should return an “OK” string. This query is done to make sure that it is possible to receive data from Ledger. Cron task is added to cache election configurations locally. Caching gives a possibility to provide available elections to voter faster, rather querying elections from Ledger each time voter authenticates to the system.

Public record addition

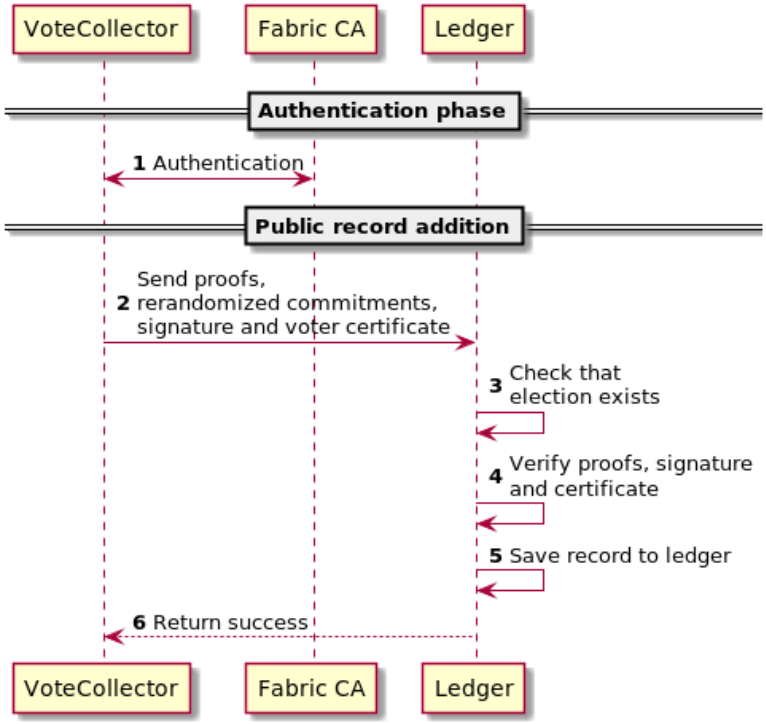


Figure 14. Sequence diagram of public record addition to the Ledger.

Possible points of failure:

- on step 1 authentication fails which restricts access to ledger modification;
- on step 3 if election with provided election ID does not exist, Ledger will return an error;
- on step 4 verifications fail, which means that public record is invalid. Verifications that are made are the same as in the VoteCollector:
  - check that received signature verifies with received commitments;
  - check that voter certificate was issued from election configuration root certificate;
  - check that each commitment contains encryption of 0 or 1;
  - check that commitments sum is up to 1.
- on step 5 public record saving process fails.

Audit data query

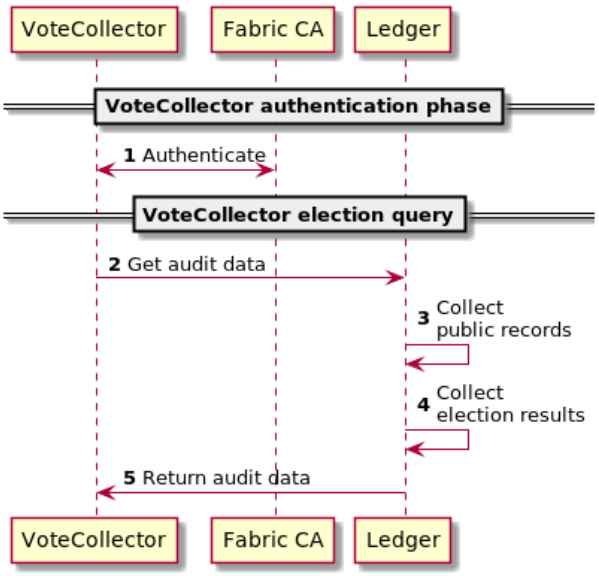


Figure 15. Sequence diagram of audit data query from the Ledger.

Audit data query from the bulletin board can fail if identity with which VoteCollector authenticates to the Fabric CA is invalid or revoked.

4.2.4.3 ElectionManager with Ledger

Authentication.
Authentication phase is added to the application because we need to restrict the number of users that can manage elections. It contains a two-layer authentication (see Figure 16):

1. Check inside the ElectionManager that the user who tries to use the application is a Election Organizer.

2. Check that ElectionManager server identity is valid and was issued from Election Organizer CA.

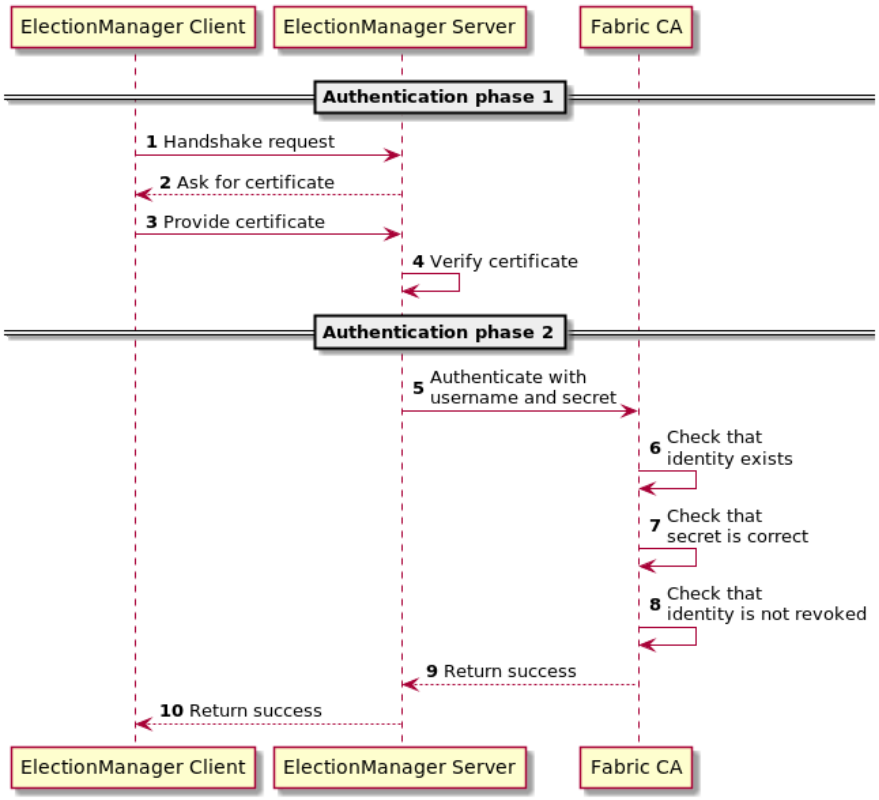


Figure 16. Sequence diagram of authentication in ElectionManager.

Points of failure are almost the same as in VoteApp authentication (see Figure 9), except that on step 4 certificate verification is valid only if Election Organizer certificate was provided.

Election publication.

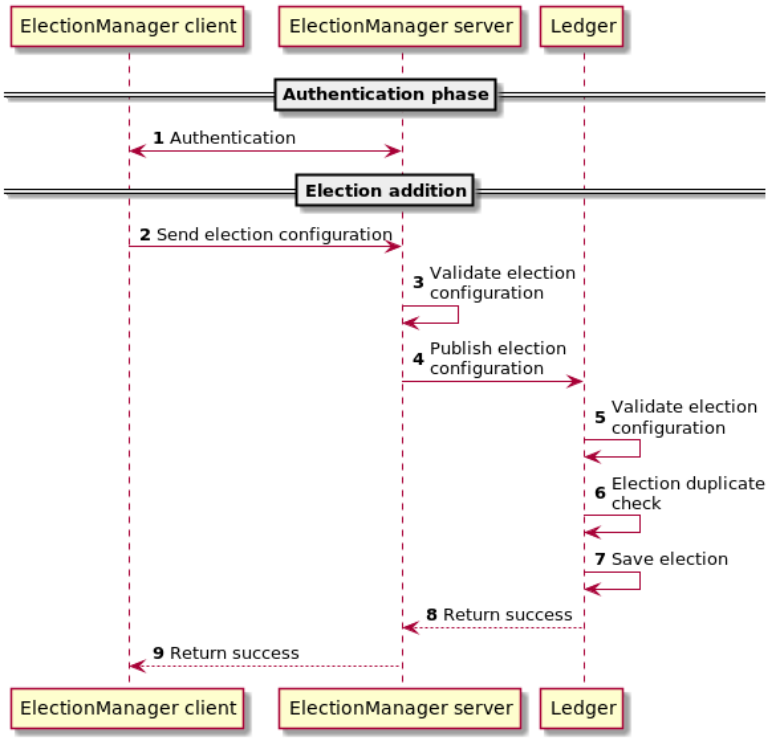


Figure 17. Sequence diagram of election configuration publication.

Configuration publication failure points:

- authentication process fails on step 1;
- on step 3 or step 5 election configuration validation failed because of file invalid structure or one of the necessary fields are empty;
- on step 6 Ledger detects that exists election with the same ID.

Election deletion.
With CCE scheme we forbid election configuration modification inside the ledger. The reason being that there is a possibility of choices list modification which can affect voting results. Therefore it is possible only to delete the whole election with results, public records and configuration.

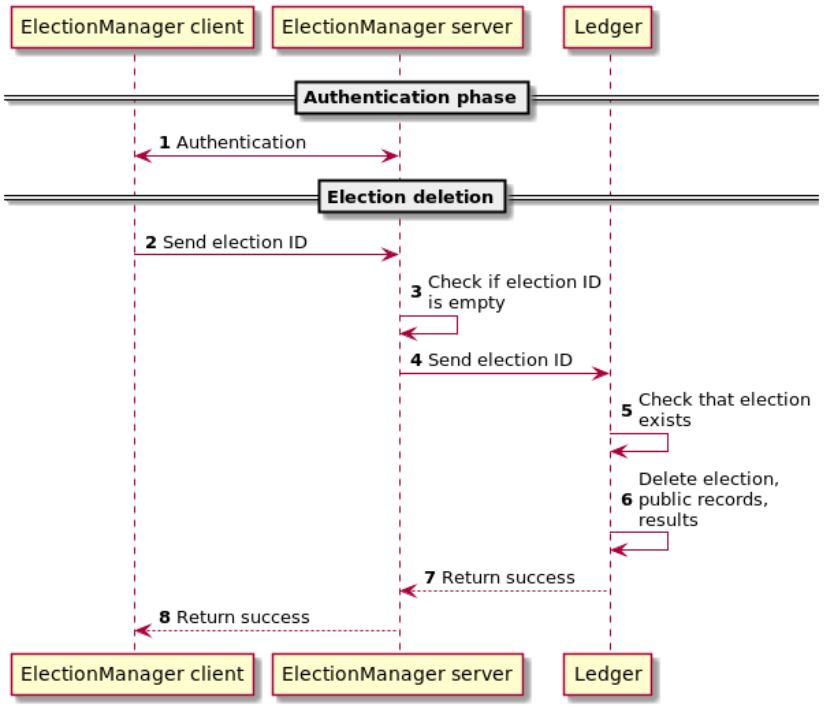


Figure 18. Election deletion sequence diagram.

Possible points of failure:

- authentication fails and the connection is dropped;
- on step 3 received election ID is empty;
- step 5 election ID does not exist inside the Ledger.

Election results publication.

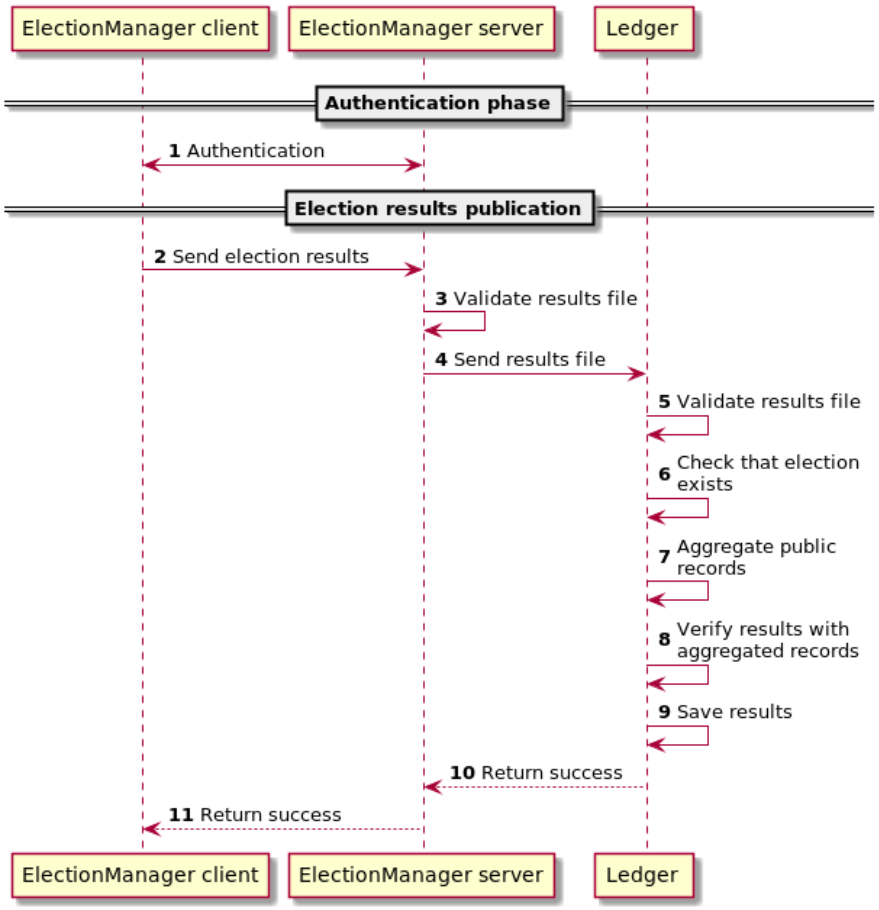


Figure 19. Sequence diagram of election results publication process.

Possible points of failure:

- authentication fails and the connection is dropped;
- on step 3 file validation fails because of the invalid file structure or some necessary fields are empty;
- on step 6 provided election ID in results file does not exist in Ledger;
- on step 7 while public record aggregation is in the process some of public record verifications fails:
  - voter certificate was not issued from election configuration root certificate authority;
  - voter signature does not correspond to provided commitments;

  - commitments do not contain encryption of 0 or 1 (zero-one proof);
  - commitments sum does not equal to 1 (sum proof);
  - choices IDs are not the same as in election configuration.

- on step 8 verification with aggregated commitments, provided results and openings fails which ends in publication failure.

5 System usage

This section will describe who and what does on each phase. This is also can be considered as a user manual for this system.

5.1 Setup phase

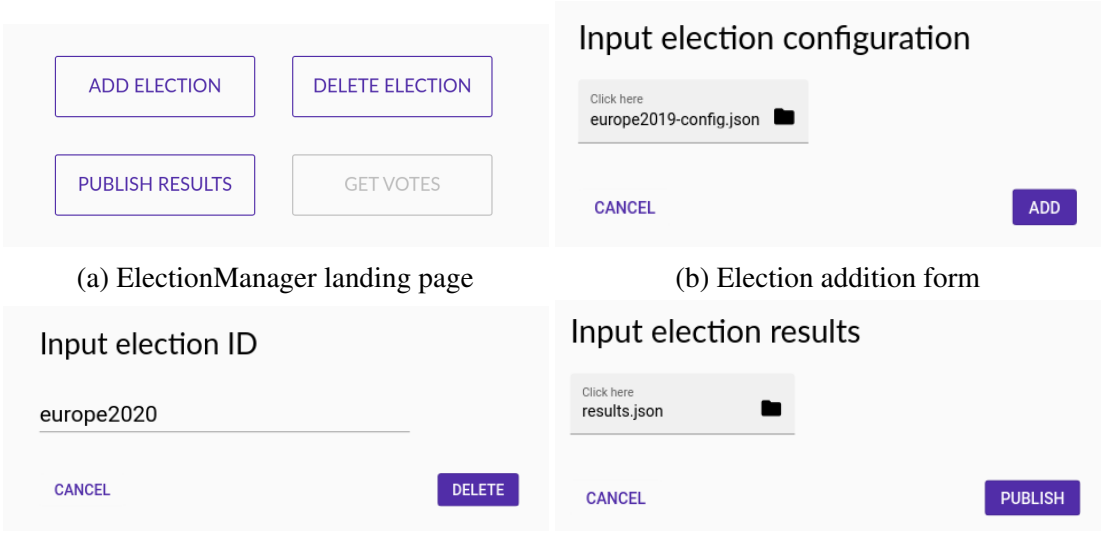
Firstly, Election Organizer with Auditor set up a Ledger with two organizations, they both create a consortium where they agree on who can add, read and modify information inside the Ledger and on what conditions. Then Election Organizer makes different operations to finalize the system initialization:

1. Creates a Public Key Infrastructure (PKI) to give authentication and signing possibility to voters.
2. Generates identities for VoteCollector and ElectionManager by using Fabric CA which belongs to his organization.
3. Creates configuration files for VoteApp, VoteCollector and ElectionManager.
4. Compiles Go binaries (Key application, VoteCollector, ElectionManager server).
5. Creates application Docker images.
6. Initializes VoteApp, VoteCollector and ElectionManager as containers.

Because this is a complex process these operations are automated. Automation can be done by one general `tiviledge/src/Makefile` script or by executing different parts of the system individually:

- `hlf/hlf.sh` for Ledger initialization and election management;
- `testdata/world.py` for PKI, voters and TLS certificates generation;
- `key/Makefile` for Key application;
- `collector/Makefile` for VoteCollector;
- `voteapp/Makefile` for VoteApp;
- `election-manager/client/Makefile` for ElectionManager client;
- `election-manager/server/Makefile` for ElectionManager server;
- `hlf/fabric-ca.sh` for identity creation and revocation in Fabric CA.

After the system is initialized Election Organizer starts with election configuration creation and CCE key pair generation. Appendix B shows an example of election configuration. Then Election Organizer proceeds with election management (see Figure 20).



(a) ElectionManager landing page

(b) Election addition form

(c) Election deletion form

(d) Election results publication form

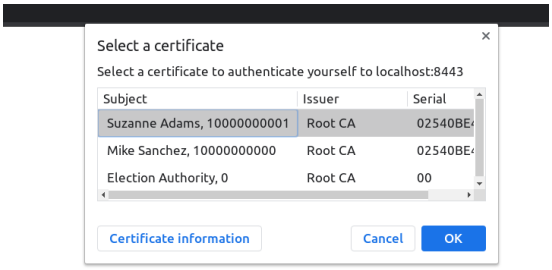
Figure 20. Election management demonstration in ElectionManager.

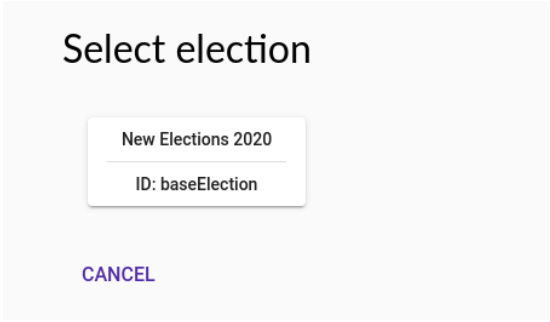
5.2 Voting phase

In this phase, voters are using VoteApp to cast and verify votes. Voter interaction with VoteApp to cast a vote:

1. Selects “CAST VOTE” button (Figure 21a);
2. Provides his certificate from the browser certificate store (Figure 21b);
3. Selects available election (Figure 21c);
4. Selects the desired choice (Figure 21d);
5. Receives success message of casting process (Figure 21e).

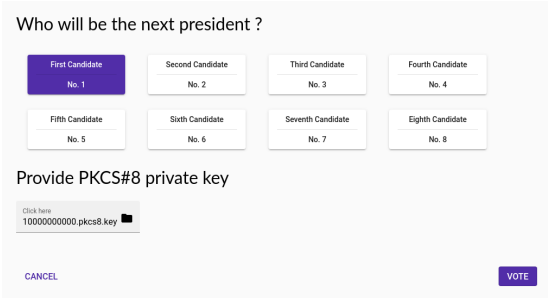
(a) VoteApp landing page



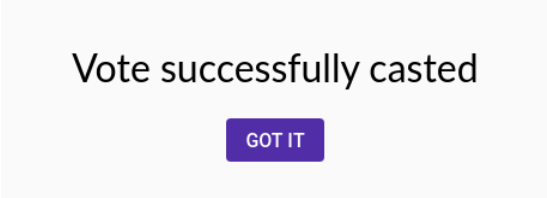


(b) Authentication step

(c) Election selection step



(d) Choice selection step



(e) Cast success message

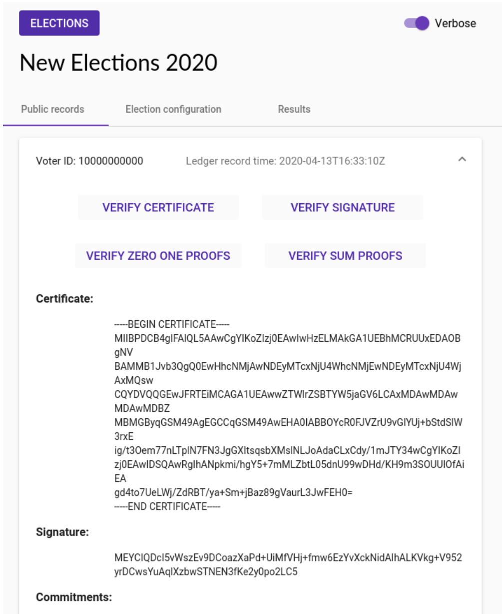
Figure 21. VoteApp casting process demonstration.

When voter finished casting process his public record is published on bulletin board and anyone can verify that this record is correct. Verifier interaction with VoteApp:

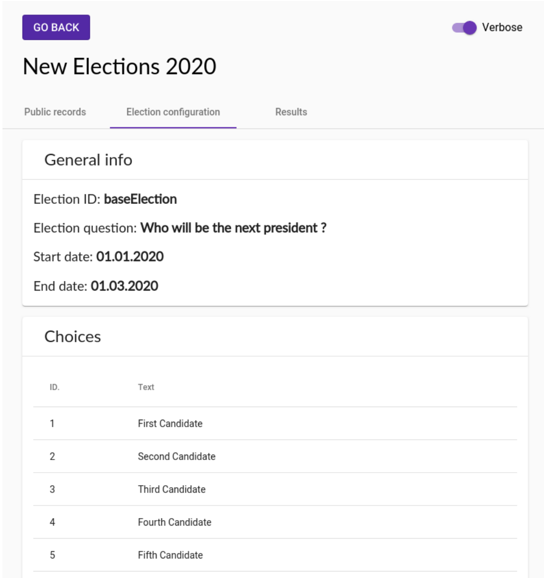
1. Selects “VERIFY VOTE” button (Figure 21a);
2. Selects available election (Figure 21c);
3. Verifier can now check any public record individually (if there are any) by making different verifications (Figure 22a):

- VERIFY CERTIFICATE – checks that voter certificate was issued from root certificate in election configuration;
- VERIFY SIGNATURE – verifies that signature corresponds to commitments plus election ID;
- VERIFY ZERO ONE PROOFS – verifies that commitments contain only value zero or one;
- VERIFY SUM PROOF – verifies that commitment sum is equal to one.

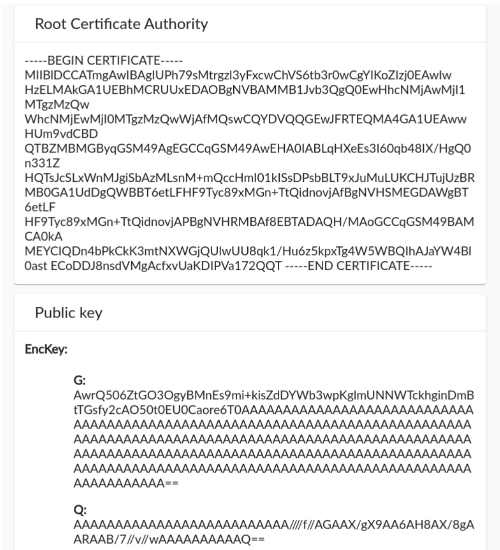
4. Verifier can check election configuration (Figures 22b, 22c);



(a) Public record on bulletin board



(b) Election configuration part 1



(c) Election configuration part 2

Figure 22. Public record and election configuration demonstration in VoteApp.

5.3 Tallying phase

Election Organizer will gather all votes from VoteCollector container and by using Key application performs decryption with additional opening extraction from aggregated commitments, example of decryption process can be seen in Apendix C. Then decryption

results with openings can be published on bulletin board where anyone can verify the results. The publication can be done in two ways:

1. By using `hlf/hlf.sh` script for publication.
2. By interacting with ElectionManager (Figure 20a, 20d).

When election results are published to the bulletin board, anyone can verify election results via VoteApp (Figure 21a, 23).

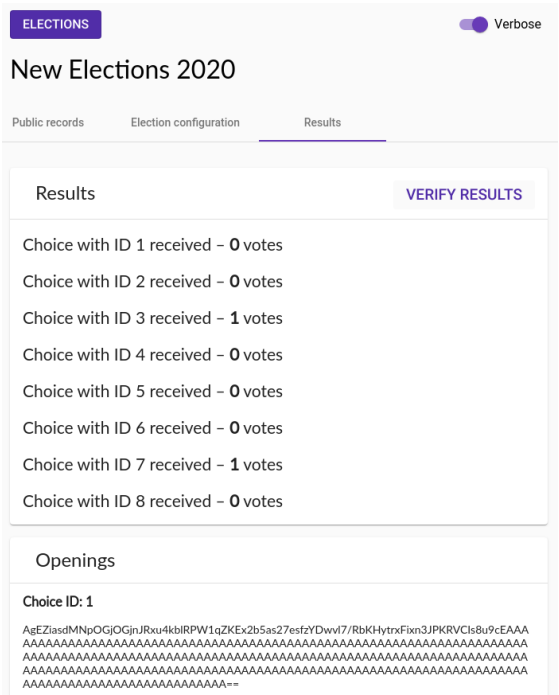


Figure 23. Election deletion sequence diagram.

By clicking on "VERIFY RESULTS" button on Figure 23, which triggers a tally process verification. VoteApp aggregates public records into one vector, additionally, it makes the same checks as shown on Figure 22a. When aggregation is finished commitments are being verified with published openings and results. Verifier will be notified if this process succeeded or failed.

6 Testing and results

This section describes testing methods used to verify that implemented online voting system fulfils desired functional and non-functional requirements. Most of the testing processes were done on a local machine, see Table 5 for machine configuration.

Table 5. Testing machine hardware and operating system info.

Operating system	Ubuntu 18.04.4 LTS
Operating system type	64-bit
Linux kernel version	4.15.0-88-generic
CPU	Intel i5-7260U
RAM	16 GB

6.1 System testing

In system testing, the whole system was set up locally with components described in Section 4.2. System initialization consisted of the same procedures described in Section 5.1. Election configuration publication and vote casting were performed manually by interacting with ElectionManager and VoteApp via the browser. ElectionManager and VoteApp were tested with two browsers:

- Google Chrome (Version 80.0.3987.163 (Official Build) (64-bit));
- Firefox (Version 74.0.1 (64-bit)).

While Chrome browser worked without any problem, Firefox for some reason on the authentication phase (see Figures 9, 16) did not provide any possibility to choose a certificate from its certificate store. Unfortunately, the real cause was not found. A probable reason is in how Firefox handles TLS authentications.

It is possible to automate vote casting and election management by using different automation tools like Selenium for user interaction simulation in the browser or by sending different HTTP requests directly to VoteCollector and ElectionManager server with Postman but this was more time consuming and is out of thesis scope.

For testing purpose four different elections were created where the number of choices represents a different type of elections:

- 2 - as a referendum where voters need to choose between “YES” and “NO”;
- 8 - as small local elections;
- 70 - as 2019 European Parliamentary elections in Estonia;

- 1100 - as 2019 Estonian Parliamentary elections.

It was decided that 5 votes for each election would be enough to show how the number of choices affects computation time in the system.

6.2 Performance testing

Performance testing consisted of computation time recordings of different processes inside components. Measured processes where:

- vote casting (Table 6);
- public record verification (Table 8);
- votes decryption (Table 7);
- election results publication to Ledger (Table 9);
- election results verification in VoteApp (Table 9).

Table 6. Vote casting time pure computation and total time with HTTP requests. All times are provided in seconds.

Number of choices	**2**	**8**	**70**	**1100**
Cast message creation - VoteApp	0.87	2.22	17.73	281.52
Cast message process - VoteCollector	0.28	1.58	9.58	147.88
Final message creation - VoteApp	0.53	1.65	13.87	222.73
Final message process - VoteCollector	0.12	0.41	3.57	68.11
Public record addition - Ledger chaincode	0.14	0.45	4.00	64.27
Total casting time	1.94	6.31	48.75	784.51
Total casting time with HTTP requests	**3.82**	**8.44**	**51.02**	**799.65**

Table 7. Votes tallying time results in Key application (with 5 votes). All times are provided in seconds.

Number of choices	**2**	**8**	**70**	**1100**
Vote aggregation time	0.11	0.16	1.25	20.77
Decryption with opening extraction	0.39	0.64	5.47	99.5
Total time	**0.50**	**0.80**	**6.72**	**120.27**

Table 8. Vote individual verification in VoteApp. All times are provided in seconds.

Number of choices	**2**	**8**	**70**	**1100**
Certificate check	0.17	0.10	0.18	0.12
Signature check	0.20	0.13	0.14	0.18
Zero One proof check	0.63	1.28	10.36	157.94
Sum proof check	0.18	0.24	0.91	12.04

Table 9. Vote universal verification in VoteApp and Ledger chaincode (5 votes). All times are provided in seconds.

Number of choices	**2**	**8**	**70**	**1100**
Public records aggregation with individual verification (s)	2.78	7.42	52.42	–
Opening verification	0.59	2.18	17.66	–
Total time VoteApp	**3.37**	**9.6**	**70.08**	**–**
Results verification inside Ledger chaincode	0.86	2.99	25.11	–

In Table 9 there are no time results for election with 1100 candidates. The reason is that the election results verification takes a lot of time (longer than 300 seconds), which triggers Orderer (see Section 4.1.5) inside the Ledger to drop the connection. To resolve that some changes to Ledger network configuration must be done but at the time of writing and testing the right place where this modification must be done is not found.

Unfortunately with a high number of choices (greater than 8), vote casting takes a lot of time and as we see from the tables that by increasing the number of choices computation time increases linearly. Therefore some modifications to the system are preferred to improve the performance of the system.

6.3 Requirements validation

Another important part of testing is that it is necessary to validate that the implemented system satisfies defined requirements in Section 2.

Verification results of functional and non-functional requirements are provided in Table 10 and Table 11 respectively.

Table 10. Functional requirements verification.

ID	Requirement	Status	Reasoning
EA-1	The system must have a possibility to generate keys for elections.	Satisfied completely.	Election Organizer can use Key application for key pair generation and vote decryption.
EA-2	The system must have an election bulletin board.	Satisfied completely.	The election bulletin board is being held inside the Ledger with Election Organizer and Auditor organizations.
EA-3	The system must have a possibility to publish votes to the bulletin board.	Satisfied completely.	Votes are saved locally and rerandomized commitments are published to the bulletin board by VoteCollector.
EA-4	The system must have a possibility to publish election configuration and results to the bulletin board.	Satisfied completely	These actions can be made with ElectionManager and their appearance on the bulletin board can be checked with VoteApp.
EA-5	The system must have a possibility to validate published votes and election results.	Satisfied completely.	Verification of public records and election results are made in VoteCollector and Ledger. Whereas same verifications can be made by anyone in VoteApp.
VOT-1	The system must have a possibility to cast vote.	Satisfied completely.	Voters are using VoteApp for vote casting.
VOT-2	The system must have a possibility for Voters to access the bulletin board.	Satisfied completely.	Voters can access the bulletin board via VoteApp and make different verification to public records, election configuration and election results.
AUD-1	The system must be set up with Auditor participation.	Satisfied completely.	Auditor participates in Ledger creation by representing his organization in the network.
AUD-2	The system must have a possibility for votes, election configuration and election results verification.	Satisfied completely.	The same as in the EA-5 verifications are made with VoteCollector, Ledger and VoteApp.

Table 11. Non-functional requirements verification.

ID	Requirement	Status	Reasoning
NF-1	The applications web interface should be simple and intuitive to use.	Satisfied partially.	The thesis author opinion is that this requirement is satisfied completely, but to be more objective different users should test implemented interface for honest opinion.
NF-2	The applications web interface should change it's dimensions according to the screen resolution.	Satisfied completely.	VoteApp and ElectionManager change its user interface according to the screen size and additional testing was made with Google Mobile-Friendly test which concluded that the page is mobile-friendly (see Figure 26).
NF-3	The applications web interface should be mobile friendly.	Satisfied completely.	The same as with NF-2 requirement, Figure 26 shows that both VoteApp and ElectionManager are mobile-friendly.
NF-4	The vote casting process for voter should take no more than 5 seconds.	Satisfied partially.	With a high number of choices casting time takes longer than 5 seconds, therefore this requirement is satisfied only with a small number of choices.

7 Conclusion

This section finalizes thesis work and provides different proposals for the system modifications and improvements.

7.1 Future

In future, this prototype will be improved and stacked with new features. Possible changes to the system:

- Authentication in VoteApp and ElectionManager can be moved to the application level so that users don't have to import their certificates to the browser certificate store. Another possibility is to change authentication to username and password combination.
- Change BLS12-461 elliptic curve to a more efficient one.
- Randomnesses generation for each commitment can be initialized right after candidate list reception. Additionally, VoteApp connection with VoteCollector can be done with WebSockets for parallel computation, which can improve performance.

Additional new features to the system:

- When Election Organizer publishes election configuration or results he signs them with his private key.
- Make election management possible only for the same person who initially published election configuration to the ledger.
- CCE key pair generation in a threshold manner.
- Add possibility to query specific keys in Ledger (bulletin board) to anyone.
- Add possibility to get votes via ElectionManager.
- Use Fabric CA to generate necessary crypto material for peers and orderers.

7.2 Summary

The goal of this thesis was to develop a working prototype of an End-to-End Verifiable i-voting system with permissioned blockchain network, which gives a possibility to anyone verify that cast votes were formed correctly by eligible voters, as well to verify the tally process.

The thesis contains a description of the voting scheme used for End-to-End Verifiable protocol and system architecture with additional manual for its usage.

Testing proved that the implemented prototype is functional and fulfilled almost all requirements described at the beginning of the thesis and pointed out some different weak points in it. The primary one is that because authentication is done with client certificate from the browser certificate store, the client certificate needs to be imported manually by the user into the browser before using the voting application (VoteApp), which is not good for user experience. Another point is that the casting process with more than 8 candidates takes a lot of time (more than 9 seconds), which as well affects user experience and makes the system less appealing for the masses. These two points considered as the most important ones to be resolved for the production live version.

HyperLedger Fabric proved to be functional as the bulletin board and blockchain feature provides trust that any data inside the bulletin board can not be lost. Besides the possibility to hold the same copy of bulletin board among different organizations is very appealing for honest elections.

References

[1] Gennaro Avitabile. "End-To-End Verifiable Internet Voting on Permissioned Blockchains". University of Salerno, 2019.

[2] Oliver Pereira. *Verifiable Elections with Commitment Consistent Encryption A Primer*. https://arxiv.org/pdf/1412.7358.pdf . (01.04.2020).

[3] Shaan Ray. *What are Zero Knowledge Proofs?* https://towardsdatascience.com/what-are-zero-knowledge-proofs-7ef6aab955fc . (08.04.2020).

[4] Sven Heiberg et al. *Improving the verifiability of the Estonian Internet Voting scheme*. https://research.cyber.ee/~janwil/publ/ivxv-evoteid.pdf . (21.04.2020).

[5] Yonezawa S. et al. *Pairing-Friendly Curves*. https://tools.ietf.org/id/draft-yonezawa-pairing-friendly-curves-00.html#rfc.section.4.2 . (19.01.2020).

[6] Aurore Guillevic, Simon Masson, and Emmanuel Thomé. *Cocks–Pinch curves of embedding degrees five to eight and optimal ate pairing computation*. https://eprint.iacr.org/2019/431.pdf . (01.04.2020).

[7] Anastasia Z. *What's the Difference Between Single-Page and Multi-Page Apps*.

[8] QArea News Editor. *The Evolution of Go: A History of Success*. https://qarea.com/blog/the-evolution-of-go-a-history-of-success/ . (03.12.2019).

[9] Wikipedia. *Go (programming language)*. https://cutt.ly/Pe3hgpJ . (03.12.2019).

[10] Reema Oamkumar. *Advantagges And Disadvantages Of Golang (Go)*. https://www.software-developer-india.com/advantages-and-disadvantages-of-golang-go/ . (03.12.2019).

[11] angular.io. *Angular architecture*. https://angular.io/guide/architecture . (17.01.2020).

[12] MIRACL. *MIRACL Core Cryptographic Library*. https://github.com/miracl/core . (19.01.2020).

[13] opensource.com. *What is Docker?* https://opensource.com/resources/what-docker . (08.04.2020).

[14] Hyperledger. *Hyperledger Fabric*. https://hyperledger-fabric.readthedocs.io/en/latest/whatis.html#hyperledger-fabric . (19.01.2020).

[15] Kynan Rilee. *Understanding Hyperledger Fabric – Endorsing Transactions*. https://medium.com/kokster/hyperledger-fabric-endorsing-transactions-3c1b7251a709 . (14.04.2020).

[16] Bisade Asolo. *Double-Spending Explained*. https://www.mycryptopedia.com/double-spending-explained/ . (14.04.2020).

[17] Hyperledger. *Peers*. https://hyperledger-fabric.readthedocs.io/en/release-2.0/peers/peers.html . (14.04.2020).

Appendices

A Source code

The source code of the prototype will be published according to the publication schedule alongside PRIViLEDGE project deliverables at `https://priviledge-project.eu/publications/deliverables`. For preliminary access please contact the thesis author at **sergei@ivotingcentre.ee**.

B Election configuration example


```
{
    "electionID": "home2020",
    "electionName": "Go home 2020",
    "electionQuestion": "Do you want to go home ?",
    "startDate": "01.05.2020",
    "endDate": "12.05.2020",
    "choices": [
        {
            "id": "1",
            "text": "Yes"
        },
        {
            "id": "2",
            "text": "No"
        }
    ],
    "publicKey": {
        "EncKey": {
            "G": "AwrQ506ZtGO3OgyBMnEs9mi+kisZdDYWb3wpKglmU...",
            "Q": "AAAAAAAAAAAAAAAAAAAAAAAAAA////f//AGAAX/gX...",
            "Y": "AgGM/w01P14/y4O5uwJfMp6x2p6Ia3lzvPm1XOsDT..."
        },
        "H1": "AhDMVBOKBqUKmvZ5RTwnDIkvl8KQfteNRtWBpmgD8Ibb...",
        "H2": "AwVYEWU6UPTb3iZRmry/JHaQFWleJ9cW4aVOmhtdY4Oc..."
    },
    "rootCA": "-----BEGIN CERTIFICATE-----\nMIIBkzCCmgAwIBA..."
}
```

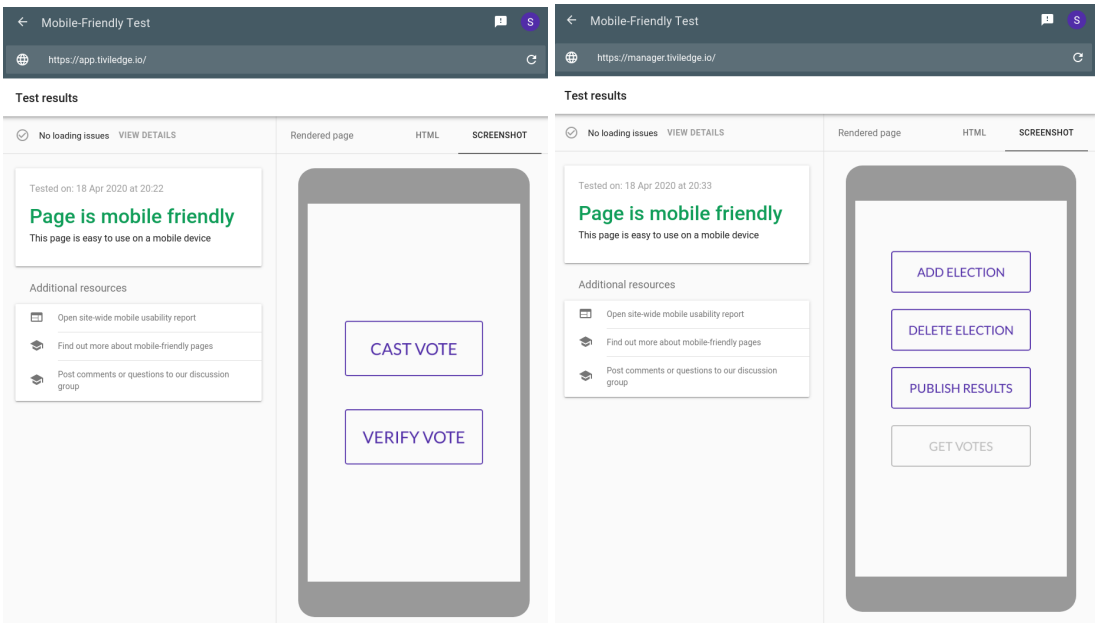

Figure 24. Election configuration example.

C Decryption process with Key application

```
$ ./key --decrypt --id home2020 --keys keys/home2020/ --votes
↪  votes/home2020/
Initiate vote decryption for 'home2020'...
Reading keys...
Aggregating votes...
Number of voters is: 3
Voter '10000000000' latest vote is 'vote~2020-04-07T08:27:43Z.json'
Voter '10000000001' latest vote is 'vote~2020-04-07T08:28:29Z.json'
Voter '10000000002' latest vote is 'vote~2020-04-07T08:30:17Z.json'
Votes are aggregated...
Aggregation took: 41.560649ms
Decrypting ciphertexts and extracting openings...
Decryption and openings extraction took: 170.552309ms
Election results are:
--- Choice ID - 1 got result - 2
--- Choice ID - 2 got result - 1
Election results with openings are saved to
↪  'results/home2020-results.json'
```

Figure 25. Key application decryption process.

D Mobile-friendly test results



(a) VoteApp (b) ElectionManager

Figure 26. Google Mobile-Friendly Test results.

E License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Sergei Kuštšenko**,
*(*author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Implementation of election bulletin board using HyperLedger Fabric,
   *(*title of thesis)

   supervised by Ivo Kubjas.
   *(*supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Sergei Kuštšenko
06/05/2020