Bern University of Applied Sciences (BFH)

Bachelor Thesis

CHVote Demonstrator

Authored by Kevin Häni <kevin.haeni@gmail.com>
Yannick Denzer <yannick@denzer.ch>
Supervised by Prof. Dr. Rolf Haenni <rolf.haenni@bfh.ch>
Prof. Dr. Philipp Locher <philipp.locher@bfh.ch>

Bern, November 30, 2017

Contents

1 Management Summary

2 Introduction

2.1 Electronic voting

Since 2015, it is possible for Swiss citizens registered in the cantons Geneva and Neuchâtel and living abroad, to vote electronically. However, these systems did not yet meet the requirements in terms of security and transparency, to be accepted as a secure E-Voting platform on a nationwide scale.

One of the requirements that is hardest to achieve, is that a good e-voting system must be transparent and allow an external person to verify, that every protocol participant has abided by the protocol and that only valid votes have been counted.

Another common problem is the **insecure platform problem**: If a voters computer is affected by malware, the vote casting process is no longer under the voters control and the candidate selection could be possibly manipulated without the voters notice. It must be **individually verifiable** to every voter that his intended vote has been received, while at the same time, the voters privacy must be ensured at all times. What sounds like a paradoxon, can be solved by modern cryptography.

A contract was formed between the state of Geneva and the Bern University of Applied Sciences to work out a new protocol which does meet the complex requirements set up by the government. In 2017, the resulting specification document has been officially published and a proof-of-concept / prototype has been successfully implemented by the State of Geneva.

2.2 The CHVote protocol

The protocol our project is based on, does not originate from us! The concept and specification has been created by Rolf Haenni, Philipp Locher and Reto E. Koenig of the Institute for Security in the Information Society (RISIS) of the Bern University of Applied Sciences. For this project, we have implemented the protocol according to their specification. In this section we summarize the most important aspects of the protocol and establish the terminology for better understanding this document and our application.

2.2.1 Actors

A **voter** is a person who is eligible to vote in his respective state. Every voter must possess a voting card that has been sent to him prior to an election event and that contains several codes such as the voting code and the confirmation code, which are used to identify the voter during the vote casting process. Every voter is assigned a **counting circle** which typically corresponds to the voters municipal. This information is required for statistical purposes.

A voter uses a **voting client** to participate in an election event, e.g. forming a ballot according to the protocol, by entering his selection, a voting code to cast, and later in the process, his confirmation code to confirm his vote. Verification codes and a finalization code are displayed by the voting client to ensure the voter that his vote has been cast as intended and hasn't been manipulated by a third party.

The **Election Administrator**, typically a person of the government, sets up the election event by providing the required information such as the candidates and the voters and initiates the generation of the cryptographic electorate data. He is also responsible for tallying the election and publishing the final results.

The **election authorities** can be seen as some kind of independent election observers. In a CHVote election event there are always multiple election authorities in order to avoid having a single authority that needs to be trusted. The authorities are involved in almost every step of the protocol, starting from generating their shares of the electorate data, shares of a jointly generated public key used for encryption, checking and responding to new ballots and vote confirmations, as well as in the mixing (a measure to ensure anonymity) and decryption phases.

The **Bulletin Board** acts as a central board over which most communication is done and where all the public data is stored.

The **Printing Authority** is responsible for printing the voting cards for all voters. Since the printing authority needs to be in possession of all the secret voting credentials, it is a very sensitive point in the protocol. The printed voting cards are handled over to a trusted mailing service for delivery.

2.2.2 Pre-Election & Voting cards

Before the actual election phase, the vote administrator sets up the election event by entering all required parameters, namely the voters, the elections with their corresponding candidates and the number of all. All election authorities jointly generate the cryptographic data for the whole electorate from which the voting card information is derived. The printing authority combines the information from all election authorities, prints the voting cards and delivers them to the voters by a trusted mailing service.

A voting card contains several codes, namely:

- a voting code
- a confirmation code
- a finalization code
- one verification code for every candidate

The **voting and confirmation codes** are authentication codes and used to authenticate the voter twice during the vote casting process; the first time with the voting code when he casts his vote, and a second time for confirming his vote. A second authentication code is required because otherwise an attacker who infected a voters computer with malware could just confirm a vote on behalf of the voter after he manipulated the candidate selection.

2.2.3 Individual verification with oblivious transfer

As mentioned earlier, one of the big challenges of an evoting-protocol is how it deals with the insecure platform problem: A voting platform that is infected with malware poses the risk that an attacker can manipulate the candidate selection on the vote-casting page after the voter has entered his voting code.

The CHVote specification suggests a "'Cast-as-intended verification"'-step to allow voters to detect this kind of manipulation: When a voter enters his voting code and the indices of his favored candidates, the voting client forms a **ballot** containing the voters selection encrypted with the authorities public key, his public voting credential derived from the voting code, and a **non-interactive zero knowledge proof** which proofs that the voter has formed the ballot according to the protocol and that he has been in knowledge of his voting code, without revealing any information about the voting code.

Every election authority has to check the voters public voting credential, the validity of the ballot proof and that the voter hasn't already cast a vote. The encrypted selection also serves as a query for an oblivious transfer. A **k-out-of-n oblivious transfer** allows a client to query a server for k messages, without the server knowing what messages the client requested, and without the client learning anything about the other $n - k$ messages. Adapted to the CHVote protocol: The voting client queries the authorities for the corresponding verification codes of the selected candidates, without the authorities learning which candidates the voter has selected, and without revealing any information about the other candidates.

The voter then checks if the returned verification codes match the codes of the candidates he has chosen on the printed voting card. If the selection was somehow manipulated by malware, the returned verification codes would not match the printed ones and the voter would have to abort the vote casting process. This way the integrity of the vote casting process can be guaranteed even in the presence of malware. In such a case, privacy on the other hand cannot be protected since the malware will learn the plaintext of the voters selection.

Another feature the protocol supports, is that an election event can consist of t multiple parallel elections. For example: An election event could consist of two elections in which 1 out of 3 possible candidates has to be selected. In such cases, the voter has to submit a single ballot, which contains his candidate selection for both parallel elections, so 2 in total. This raises the question, how the system can verify that a voter has chosen exactly one candidate in each election, and not zero in the first, and two candidates in the second election.

The specification suggests the following trick: Assuming an election consists of two parallel elections ($t = 2$) with 3 candidates each ($n_1 = 3, n_2 = 3$) candidates, of which a voter can select one candidate in each election ($k_1 = k_2 = 1$). The verification codes are derived from $n = \sum_{j=1}^{t} n_j$ random points on t polynomials (one for every election event j) of degree $k_j - 1$, that each election authority has chosen randomly prior to the election. By learning exactly $k = \sum_{n=1}^{t} k_j$ points on these polynomials, the voting client is able to reproduce these polynomials and therefore is able to calculate a particular point with $x = 0$ on these polynomials. The corresponding y values are incorporated into the second credential from which the confirmation code is derived. As a result, only if a voter has been able to reconstruct these polynomials with the returned points by submitting a valid candidate selection, he will be able to confirm the vote that he casted.

2.2.4 From homomorphic encryption to mix-networks

Since there is still a connection between the encrypted ballot and the voter at this point, the encrypted selections are extracted from the ballots as the first step of the post-election phase. In order to anonymize the list of encryptions, every authority performs a cryptographic shuffle with a random permutation and re-encrypts all ciphertexts in order to make it impossible to find out which voter has submitted which encrypted ballot. This is done by using the multiplicative homomorphic property of the ElGamal encryption scheme. A multiplicative homomorphic encryption scheme allows to perform multiplications on the ciphertext such that:

$$Enc(a) \cdot Enc(b) = Enc(a \cdot b)$$

The specification suggests multiplying the encryption with the encryption of the neutral element 1 since this yields a new ciphertext for the exact same plaintext.

2.2.5 Distributed trust

The public key that is used for encrypting the ballots has been jointly generated by all election authorities. Therefore in order to decrypt the result, all authorities must provide their share of the private key. The measure of multiple authorities participating in the whole e-voting process ensures the security of the whole election even if only one authority can be trusted.

2.3 Project task

Understanding such a complex protocol isn't easy and might be the reason why many people still do not trust electronic e-voting systems. In close consultation with the authors of the CHVote specification, we decided to develop an application that allows users to get a hands-on experience with the CHVote e-voting system and makes it possible to show and explain to an audience how the future of voting in Switzerland might possibly look like.

For this reason, we have implemented the protocol according to the specification and developed a web-based application on top of it, which allows to visualize every step of an election, from generating the electorate data, to casting and confirming ballots from a voters point of view, to the post-election processes like mixing, decryption and tallying. Opposed to a realistic prototype, the focus on our project is not to implement an e-voting-system that is totally realistic, but more on the visualization.

3 Project Plan

In this chapter we are going to describe several aspects regarding the planing of our project, such as our project methods, the requirements and use cases for our project and the concept of our application.

3.1 Project Method

Since this wasn't a project where the project scope and goals were strictly defined and clear from the beginning, it was very important to elaborate the requirements in close collaboration with our supervisors early on and work. This is why we decided our project would be very suitable for an agile kind of project method.

We have therefore set up regular meetings (usually once every 2 weeks) to discuss our ideas and get feedback about the current progress.

3.2 Requirements

As the first step, we have though about the goals and requirements for our application. We have structured the requirements into groups that correspond to the actors of the CHVote protocol and therefore the views our application is going to consist of.

Some requirements affect multiple or all actors and are therefore listed as "‘General Requirements"’.

3.2.1 General Requirements

	Description	Type	Prio.	Phase	Status
R1	The CHVote protocol is implemented as specified in the latest specification document. The only exclusion are the algorithms for channel security.	Must	High	1	Done
R2	The application is web-based shows updates within the same demo-election in real-time.	Must	High	1	Done
R3	The system supports 1-out-of-3 type of elections (e.g. elect 1 of 3 possible candidates)	Must	High	1	Done

R4	The system supports multiple parallel elections	Must	High	1	
R5	Users can create new elections	Must	High	1	Done
R6	The system supports internationalization. Providing more than one language is not required.	Must	Med.	1	
R7	The system can handle k-out-of-n type of elections	Can	Med.	1	

3.2.2 Election-Overview

	Description	Type	Prio.	Phase	Status
R8	The overview shows which phase the election is currently in	Must	High	2	
R9	A graphical scheme of the chVote protocol gives an overview of all participating parties	Must	Med.	2	

3.2.3 Election Administrator

	Description	Type	Prio.	Phase	Status
R10	An election can be set up by providing all required information such as the candidates, number of parallel voters, the number of voters and the number of selections (simplified JSON input)	Must	High	1	Done
R11	The election can be set up without entering the parameters in JSON format and allows easier set up of elections with multiple parallel election events	Must	Low	2	
R12	The election administrator view allows to perform the tallying and displays the final result of an election in numbers and a pie chart	Must	High	1	
R13	During election setup, the security parameters can be chosen from a set of predefined parameters	Can	Low	2	

3.2.4 Printing Authority

	Description	Type	Prio.	Phase	Status
R14	Users can generate and display voting cards for an election.	Must	High	1	Done
R15	Voting cards hide sensitive information behind a scratch card	Can	Med.	2	

3.2.5 Election Authority

	Description	Type	Prio.	Phase	Status

	Description	Type	Prio.	Phase	Status
R16	The election authority view shows all information known to an election authority	Must	High	1	Done
R17	After a voter has submitted a ballot, all election authorities can check and respond to the voters submission	Must	High	1	Done
R18	In the post-election phase, all election authorities can perform the mixing and decryption tasks	Must	High	2	
R19	Each authority can optionally processes all tasks automatically	Can	High	2	Partially done

3.2.6 Voter

	Description	Type	Prio.	Phase	Status
R20	Users are able to go through the whole vote-casting process for every voter	Must	High	1	Done
R21	The voting card of a voter is displayed on screen. The voting and confirmation codes can be copied into the input textfields by double clicking	Must	Med.	1	

3.2.7 Bulletin Board

	Description	Type	Prio.	Phase	Status
R22	The bulletin board view shows what information is publicly available	Must	High	1	Done
R23	The bulletin board view is extended with verification-functionality	Can	Low	2	

3.2.8 Out-of-scope

The following topics are considered out-of-scope for the duration of our project:

- The goal of our project isn't to build a realistic prototype. Therefore, the whole backend will run on a single server while in reality, there would be components running on distributed servers. Another difference between our implementation and a real implementation is that we generate the ballots on the server. In reality, the ballots would have to be generated on the client for security reasons. This however would require us to rewrite many of the already implemented algorithms in JavaScript.

- The protocol takes into account that not all voters might be eligible to vote in all elections of a given election event (eligibility matrix). For simplicity, we assume that all voters are eligible to vote in every election for our application.

- Message level encryption and signature based integrity protection are very important in a real implementation of a evoting-system and are also described in the CHVote specification. However, as our system is only used for demonstration purposes and as we do not have a distributed infrastructure, there is no real need for channel security in our project.
- Providing more one language is also out-of-scope. If there is enough time, a second language might be provided optionally.

3.3 Time schedule & Implementation phases

As the next step we created a time schedule and structured our project into smaller units.

The last step before starting the actual implementation was to develop a proof-of-concept / prototype of our application, by implementing one usecase in a reduced extent with the envisaged frameworks to evaluate the technical feasibility.

The actual implementation phase has been broken down into two phases.

Phase 1 involves the implementation of all high priority must-requirements, so, basically bringing the application into a state that allows to visualize the whole CHVote election process. We have agreed that after phase 1, some areas of the user interface would still be in a rather primitive state (eg. user inputs are not validated and require a more technical type of input).

For phase 2 we planned on implementing the can-requirements and the must-requirements with lower priority.

This approach reduced the risk of technical limitations of our architecture not being revealed until later that would have resulted in time consuming architectural changes.

Project tasks		38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
Documentation																		
Update documentation and journal	soll	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦 5
	ist	🟩	🟩	🟩			🟩		🟩	🟩								
Concept																		
Working out goals / requirements	soll	🟦	🟦	🟦														
	ist	🟩	🟩	🟩	🟩	🟩												
Concept	soll		🟦	🟦														
	ist		🟩	🟩	🟩													
Implementation																		
Update CHVote cryptolibrary to the lastest specification	soll	🟦	🟦 1															
	ist	🟩	🟩															
Prototyping (proof of concept)	soll			🟦	🟦 2													
	ist			🟩	🟩													
Iteration 1																		
Implement backend API	soll					🟦	🟦	🟦	🟦	🟦	🟦							
	ist					🟩	🟩	🟩	🟩	🟩								
Implement frontend: Pre-election	soll							🟦	🟦									
	ist							🟩	🟩									
Implement frontend: Election	soll								🟦	🟦								
	ist								🟩	🟩								
Implement frontend: Post-election	soll									🟦	🟦 3							
	ist									🟩								
Iteration 2																		
Automation of election authority (R19)	soll											🟦						
	ist																	
Voting Card layout, Scratchcard (R15)	soll											🟦	🟦					
	ist																	
Election Overview (R8, R9)	soll												🟦	🟦				
	ist																	
k-out-of-n election types (R7)	soll													🟦				
	ist																	
Improving the input forms and layout (R11)	soll														🟦			
	ist																	
Flexible security level (R13)	soll															🟦		
	ist																	
Verifier functionality (R23) and reserve	soll															🟦	🟦 4	
	ist																	
Project completion																		
Poster, Final Day, Presentation	soll																🟦	🟦
	ist																	

Table 3.8: Project time schedule

From our requirements we have defined the following mile-stones:

- M1: Finishing the implementation of the CHVote cryptolibrary
- M2: Upon successful creation of a proof-of-concept / prototype for our application
- M3: After finishing implementation phase 1
- M4: Finishing implementation phase 2 including the can-requirements
- M5: After finishing our documentation

3.4 Use Cases

The next step was to create use cases. As an example we show one of the use cases, for the complete list of use cases we refer to the appendix.

Table 3.9: Use Case «Casting of a vote»

Use Case	**Casting of a vote**
Primary Actor	Voter
Description	The voter can cast a vote by selecting his favored candidate(s) and his voting code
Precondition	• The election has the status "‘Election Phase"’ • A voter is selected in the "‘Voter"’-view • The voter has the status "‘Vote Casting Phase"’
Postcondition	The first election authority receives a "‘Ballot-check task"’
Main path (M)	1. The voter visits the "‘Voter"’-view and select his voter object 2. The system demands a selection of the candidates and the voter's voting code 3. The voter clicks on "‘Cast ballot"’

3.5 Concept

In an essence, all the functionality of our applications revolves around visualizing all important steps and phases of a CHVote election event. We divided an election event into the following phases:

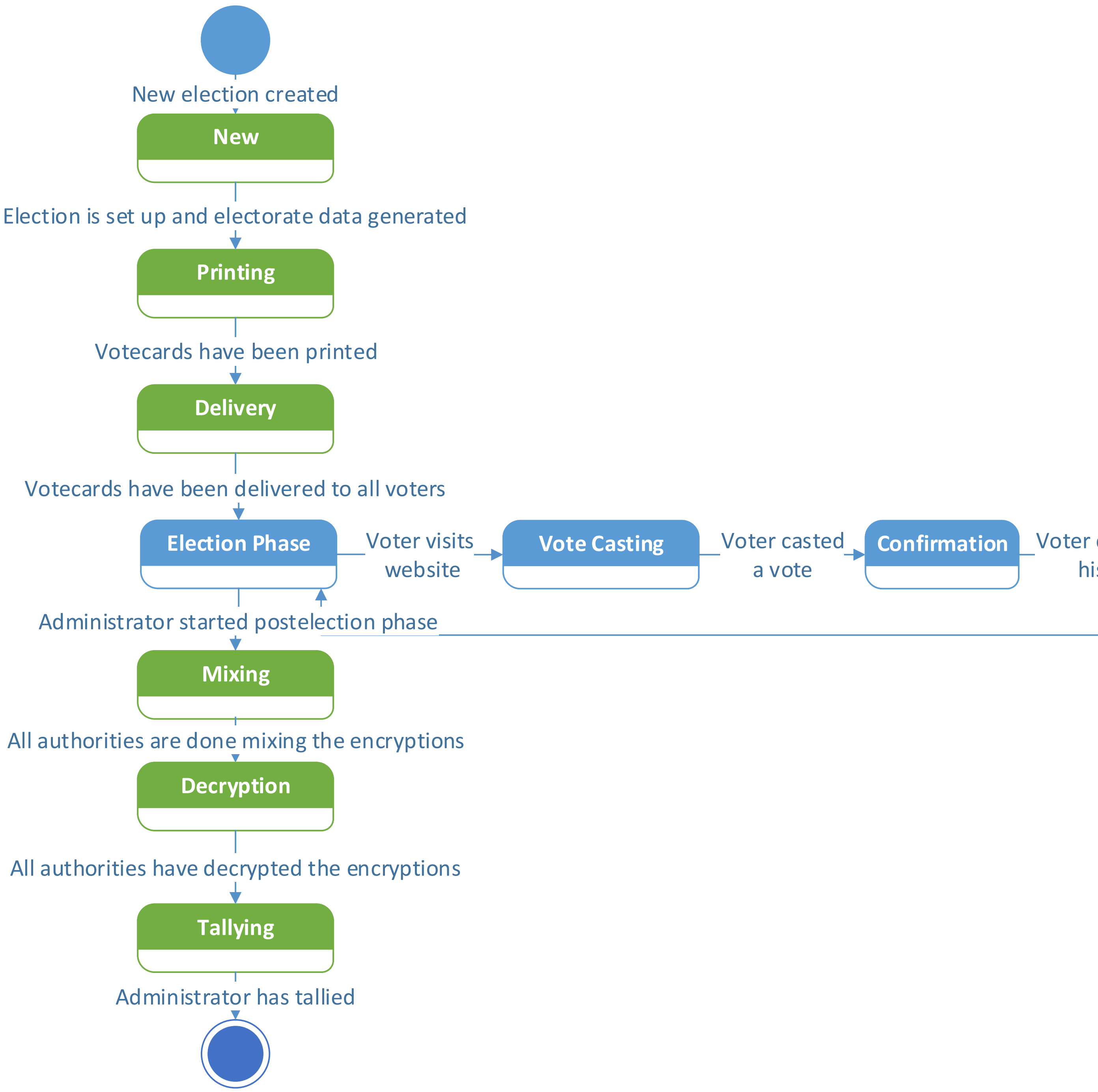

New election created
New
Election is set up and electorate data generated
Printing
Votecards have been printed
Delivery
Votecards have been delivered to all voters
Election Phase
Voter visits website
Vote Casting
Voter casted a vote
Confirmation
Voter
his
Administrator started postelection phase
Mixing
All authorities are done mixing the encryptions
Decryption
All authorities have decrypted the encryptions
Tallying
Administrator has tallied

Figure 3.1: System components

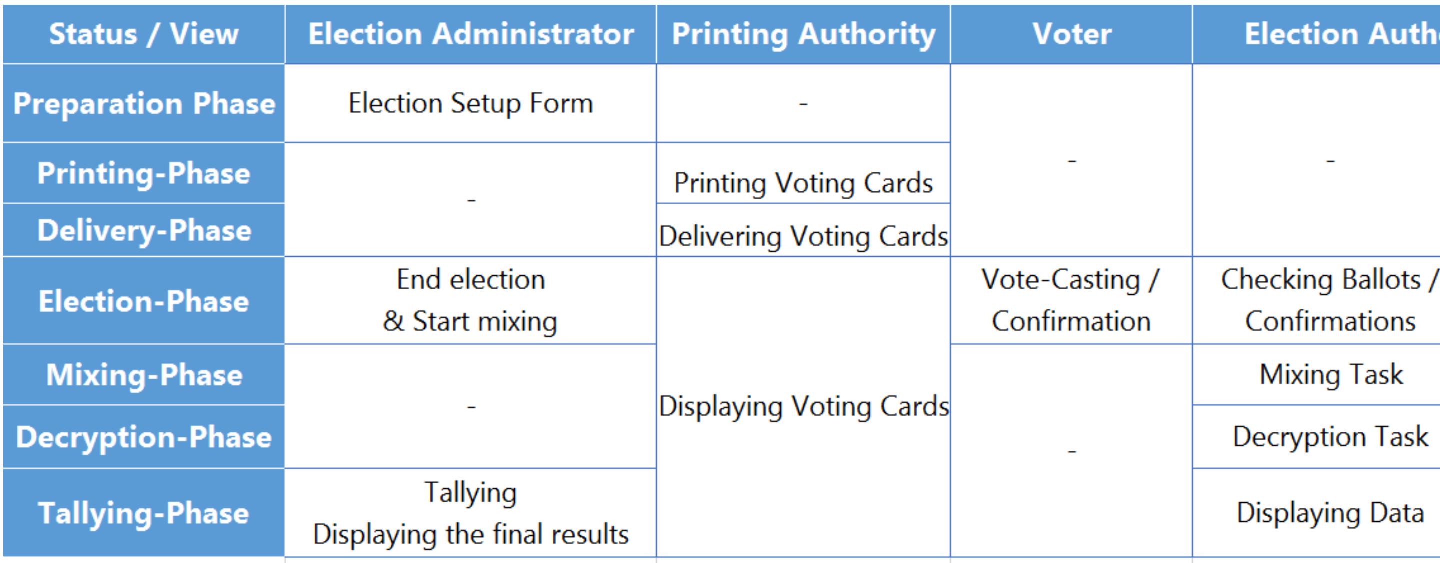
Each of theses phases consist of one or multiple steps within the CHVote protocol. The election preparation for example, involves the generation of the electorate data based on the input of the election administrator, the public credentials and the public keys. In the specification, these steps are listed as separate actions, however, to reduce the complexity we treat them as one single step in our application.

As every actor involved in a CHVote election event has it's own set of data to be displayed and specific tasks it has to perform, we decided to create a separate view for every actor, accessible from a tab view displayed on the top of the page.

From the use-cases, we tried to figure out all commonalities between the views:

- The views typically display the information known by the respective actor. Especially the bulletin board and election authorities will have a lot of information to be displayed
- Almost all views have distinct tasks to be executed by the respective actor, such as casting a ballot in the voters view or confirming ballots from an election authorities view.
- The content within each view typically depends on the status an election event is currently in: During the pre-election phase, the vote administrator needs to able to set up the election while in the tallying phase, he must be able to tally and determine the final result.

The following matrix shows all the possible combinations: Each of the cell values corresponding to one or multiple use-cases.



Status / View	Election Administrator	Printing Authority	Voter	Election Auth
Preparation Phase	Election Setup Form	-	-	-
Printing-Phase	-	Printing Voting Cards		
Delivery-Phase		Delivering Voting Cards		
Election-Phase	End election & Start mixing	Displaying Voting Cards	Vote-Casting / Confirmation	Checking Ballots / Confirmations
Mixing-Phase	-		-	Mixing Task
Decryption-Phase				Decryption Task
Tallying-Phase	Tallying Displaying the final results			Displaying Data

Figure 3.2: System components

3.6 Design

Given the rather large amount of complex data to be displayed, the main challenge of the project is a well designed user interface that allows to display all important information while maintaining a clear overview.

To achieve this goal we tried to keep our design very minimalistic and follow the Google material design guidelines as much as possible by choosing an appropriate UI component framework. Even though mobile compatibility hasn't been a requirement, it was nevertheless our goal to make the layout as responsive as possible.

Throughout our application we tried to establish common concepts regarding the look and feel and on how to display our data. One popular layout-concept of the material guidelines is the card layout. Cards can be easily integrated in a responsive grid system, look quite modern and allow to visually group data. In addition, we used pushover menus, tooltips and popups a lot as they made it possible to hide lesser relevant information by default and display it only on demand of the user.

The following screenshots are an extract of the mockups in which we tried to visualize how we imagined the resulting application to look like during the conceptual phase.

3.7 Organization

TODO: Github, Docker, etc.

4 Implementation

In this chapter, we will describe how we implemented the application. We will start explaining the architecture from a high-level perspective with each component being a black-box. Later sections of this chapter will then further describe the internals of each component.

From a high-level perspective, our application consists of three components which themselves may consist of multiple sub components: The **CHVote crypto-library** is the result of our "'Project 2"' course project which we have finished before our bachelor thesis. This library contains all the algorithms that we have implemented according to the specification.

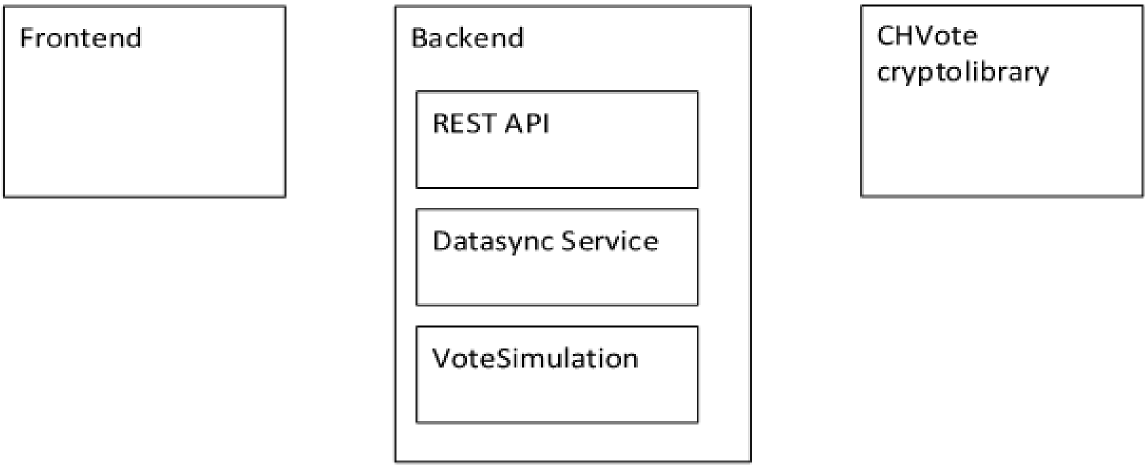


Figure 4.1: System components

The back-end consists of several components that make use of the CHVote crypto-library to build an actual e-voting ecosystem and providing an API to manipulate its state as well as a data sync service to synchronize data from the back-end to the web-clients. All the sub-components of the backend run on a single server.

The front-end basically is the web-app where all the functionality of our back-end is consumed and the resulting election state is visualized.

4.1 Technology & Language Decisions

When we implemented the CHVote crypto-library, we have evaluated and decided to use Python. Since Java has already been used by the team in Geneva, we wanted to use a different language as it was desirable to prove that the CHVote specification can be implemented in any language. Python seemed like a rather suitable language due to the following reasons:

- mature language with lots of libraries
- python's syntax enables programs to be written in a compact and readable style
- as the protocol wasn't completely specified at that point and has still been undergoing some changes, we wanted to use a language in which we can adapt changes quickly and easily
- native support for large integers (*BigInts*) and bindings for the GMP[1] library
- supports a lot of platforms
- many popular web development frameworks are available

Throughout the project not all of the reason above turned out to be true or ideal. The drawbacks that we have experienced during the implementation of this project will be discussed at the end of this document.

Since we used the crypto-library in our back-end, Python was also the obvious language for the back-end. Python offers a wide variety of frameworks for building web-services. Since we planed on building a single-page-application for the client, we chose the lightweight micro-framework flask for building a restful web-service.

For our **front-end** (web application) we evaluated several single-page application (SPA) frameworks. VueJS is a new, modern and lightweight SPA framework that in contrast to Angular has a much flatter learning curve but still offered all the functionality that we needed. The VueJS addon Vuex enabled us to establish a data-store pattern in our front-end, which makes it possible to have a copy of the back-end data-store in our web application which is synchronized in real-time through web-sockets.

Socket.io simplifies the usage of web-sockets and offers fallback technologies such as long-polling in case web-socket is not supported by either the browser or the web-server. Both Flask as well as VueJS have plug-ins that support and integrate socket.io.

For persisting the state of an election, we decided to use mongoDB. The reason behind this choice will be described in more details later on.

4.2 Architecture

The core of our application is the VoteService component in our backend which implements the e-voting protocol by utilizing all algorithms of the CHVote crypto-library according to the

[1] GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers, see `https://gmplib.org/`.

CHVote specification. The VoteService component internally holds the state of a whole CHVote election and exposes functions to manipulate this state at a granularity required by our web application to implement all use cases. For example: The VoteService contains a list of ballots and exposes functions to cast a new ballot, which will generate a new ballot according to the protocol, by calling the CHVote crypto-library, and adds the ballot to the list.

On top of the VoteService we have implemented a REST service that acts a facade to the VoteService component and makes its functionality available as an API for our web-clients. The REST service also has to initialize the VoteService by loading and persisting it's state from and to the database between each API call.

One of the requirements is that all clients must be informed in real-time about mutations of the election state made by other participants. To achieve this, we have implemented a data sync service which allows to push the state or parts of the state of the VoteService to the web-clients by using the websocket protocol. This service is triggered by the REST service after every API call to push the delta between the old and the new state to the clients.

To establish a proper separation of concerns, the state of the VoteService is always sent to the client via the data sync service. The REST API only returns success or error codes or information that is required in response to some particular API call, and never state objects. On the other hand, the data sync service does never manipulate the state of the VoteService and is solely responsible for sending s to the client.

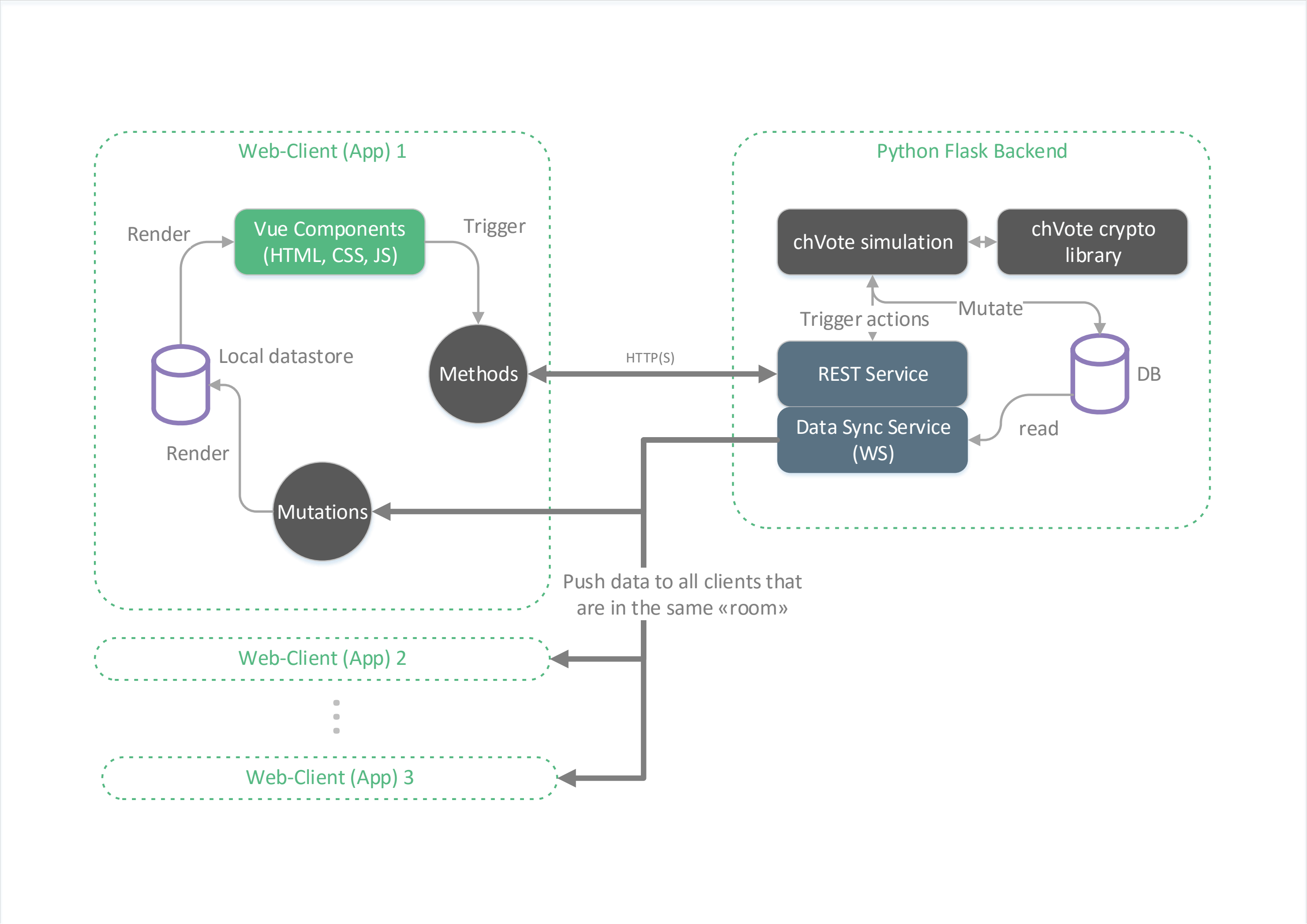


Figure 4.2: Architecture

From the clients point of view, the web-client contains a copy of the whole VoteService state in a local data-store. This store is initially populated when the web application is initialized with an election. Whenever the state of the VoteService changes, the data sync service pushes the new data to the web application. A local mutation handler within the web application handles those messages and writes the new data into the local data-store.

Since the components that the web-pages of the web application are built of, are directly bound to the local data-store, all mutations are automatically reflected to the user. From those pages, the REST API can again be called to trigger some CHVote specific action on the back end. The resulting state change is again being pushed to all clients while the responsible client that performed the HTTPS request will additionally receive a success code, or an error message in case of an error.

We have decided on this VoteService centric architecture mainly because we wanted to keep all CHVote specific implementation details within a central module and avoid having protocol-specific logic both in the client as well as in the back-end. If the protocol is undergoing any changes or if the demonstrator application should be adapted to another protocol in future, only the VoteService component (and of course its dependencies) will be affected or must be replaced. It also allowed us to implement the CHVote protocol as similar to the specification as possible. Opposed to a real implementation, where all the protocol parties would be running a separate application, it was no problem for our use-cases to have the whole protocol running on one single server and within one process. Because of that, we had access to every actors state and functions from within our VoteService object and could pass data from one actor to another as simple as setting an object property to some value.

4.3 Back-end

In this section we describe the internals of the back-end services.

4.3.1 VoteService

The VoteService represents the state of a whole election and holds instances of all the actors that participate in a CHVote election. We have divided the state of a CHVote election into the following classes:

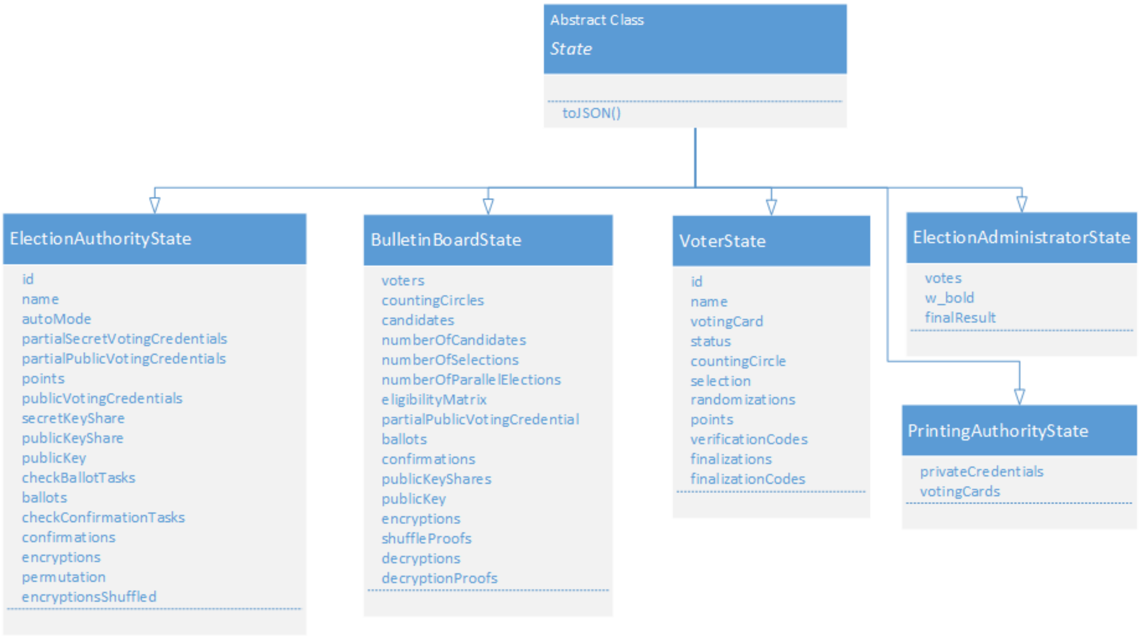


Figure 4.3: State classes

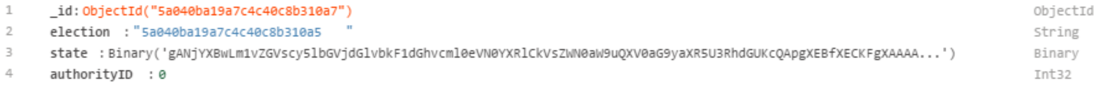
- **BulletinBoardState**: Holds all data that is publicly available on the bulletin board (the number of candidates, the tallied result)
- **ElectionAuthorityState**: Holds all data that an election authority knows (e.g. the list of ballots, the secret key of an election authority)
- **VoterState**: Since there is no distinction made between a voter and a voting client in our application, the VoterState contains the data of both the voter (e.g. the voting card) and

the data typically known to the voting client (e.g. the points returned by the oblivious transfer)

- **PrintingAuthorityState**: Holds the data known to the printing authority (e.g. the list of all voters private credentials and the voting cards)
- **ElectionAdministratorState**: Holds all data known to the election administrator

Since our application supports that multiple users work on different election events concurrently, the state of an election event cannot be kept in RAM, but needs to be persisted between every single request . For this reason we evaluated different database systems and concepts. We decided not to use a relational database system that requires us to define a database schema as we want our state objects to be the only place where the schema is defined. This "‘code-first"’ approach makes it easier to apply changes to the protocol in future.

For our purpose, mongoDB seemed like a good choice. Since we do not need the ability to access and filter our data with arbitrary queries, but only need to be able to save and load a state object of a particular election, we simply store the whole state as a binary string in a mongoDB collection. The only additional attribute that is saved to the database alongside with the serialized state is the electionId which denotes which election a particular state belongs to. An election contains multiple VoterStates and ElectionAuthorityStates. Therefore, these two states additionally require an electionAuthorityId and a voterId.



```
_id: ObjectId("5a040ba19a7c4c40c8b310a7")                                                               ObjectId
election  : "5a040ba19a7c4c40c8b310a5    "                                                              String
state  : Binary('gANjYXBwLm1vZGVscy5lbGVjdGlvbkF1dGhvcml0eVN0YXRlCkVsZWN0aW9uQXV0aG9yaXR5U3RhdGUKcQApgXEBfXECKFgXAAAA...')   Binary
authorityID  : 0                                                                                        Int32
```

Figure 4.4: Example: ElectionAuthority document collection

The only common functionality between every state object, is the ability to serialize the object to a JSON string. For this reason we had to write a custom transformer which tells the JSON parser how to serialize data-types such as mpz, bytearrays and custom classes. Luckily, python offers a way to easily serialize any custom object. By calling `object.__dict__` we can convert an object into a dictionary, as long as the transformer is able to serialize all properties of the object.

We described how the state classes are used to divide the data of the VoteService into smaller units. Similarly, the functionality of the VoteService class is separated into classes, one for every actor in the protocol.

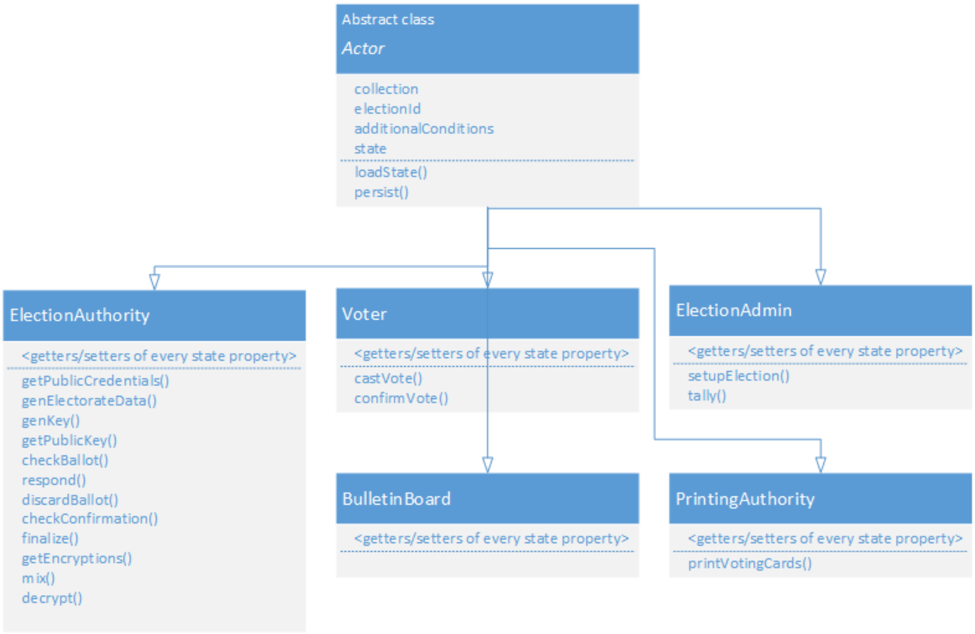


Figure 4.5: Actor classes

The common functionality, namely, a function for loading the corresponding state from the database and one for persisting the state to the database, are contained in an abstract base class.

The distinction between the actors and their states allows us to easily serialize the state of every actor and provide methods for loading and persisting the state to the database.

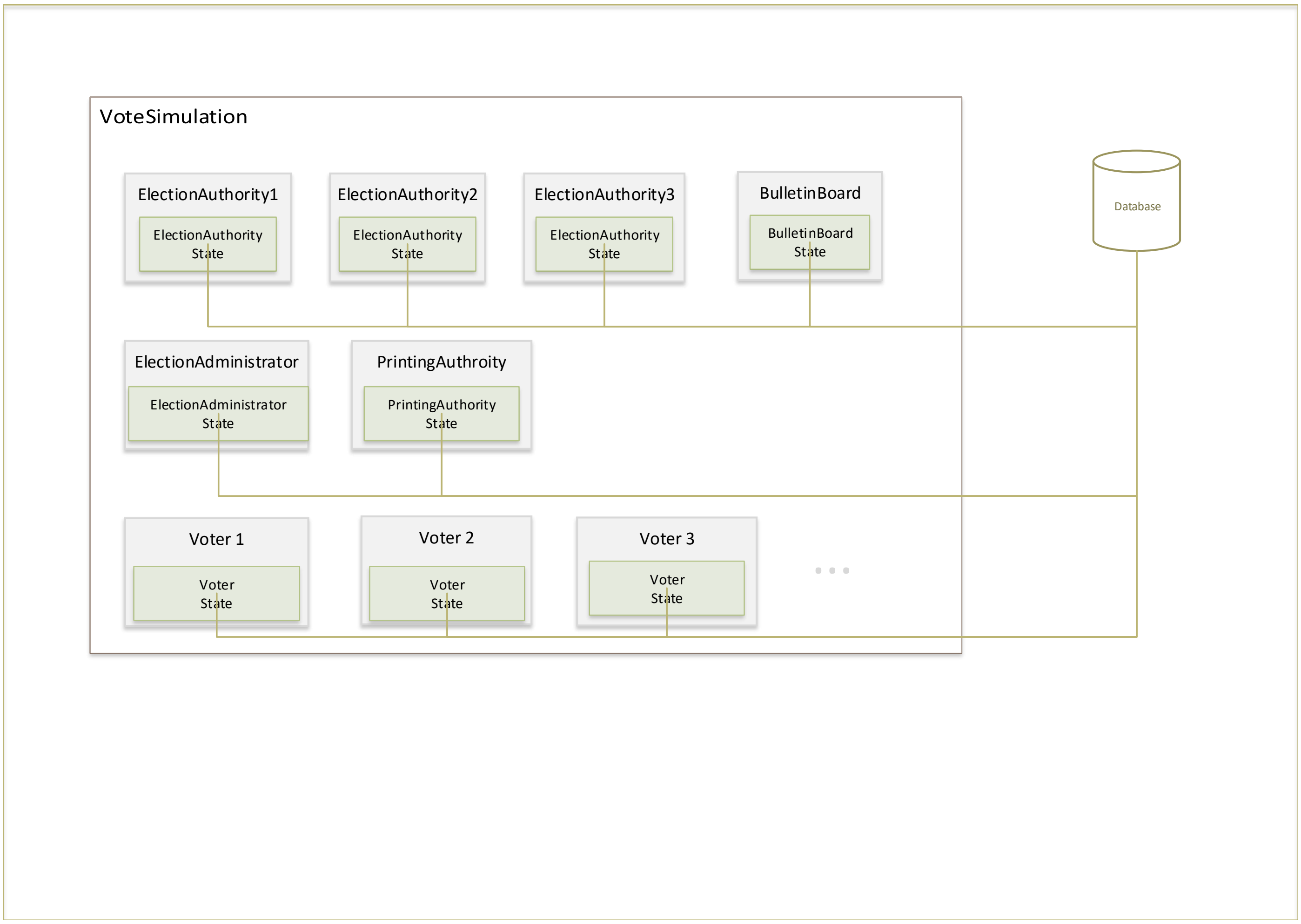


Figure 4.6: State classes

This approach was also necessary because we have implemented the data synchronization between the clients and the back-end using the JSON Patch approach, for which the delta between the original state and the modified state can be automatically determined and operations generated which will patch the state object on the client such that it equals the new, modified state of the back-end. The original state is simply attached as another property to the actor-object.

4.3.2 Data-Sync Service

One of the big challenges of our application has been the synchronization of the election state from the back-end to the clients local data store. As mentioned earlier, we wanted to achieve real time data synchronization such that every web client observing a particular election, is informed of any change of this elections state. For this purpose we have used web-sockets.

Additionally, we had to keep an eye on the performance of the data transfers since especially the bulletin board and the election authorities grow big in size when they contain a lot of ballots. We observed that the size of the whole state of an election with 6 candidates and 10 voters, of which each has submitted at least one ballot and a confirmation can easily reach

600 kilobytes already. Admittedly, we haven't noticed any performance issues even with rather large elections, however, transferring the whole state of an election after every single mutation didn't seem like a proper solution.

Nevertheless, when a client connects to the data sync service for the first time, it needs to fetch the JSON representation of every state object of the VoteService. For this purpose we have implemented a "‘FullSync"’ method which will populate the clients local data store with the full state of an election.

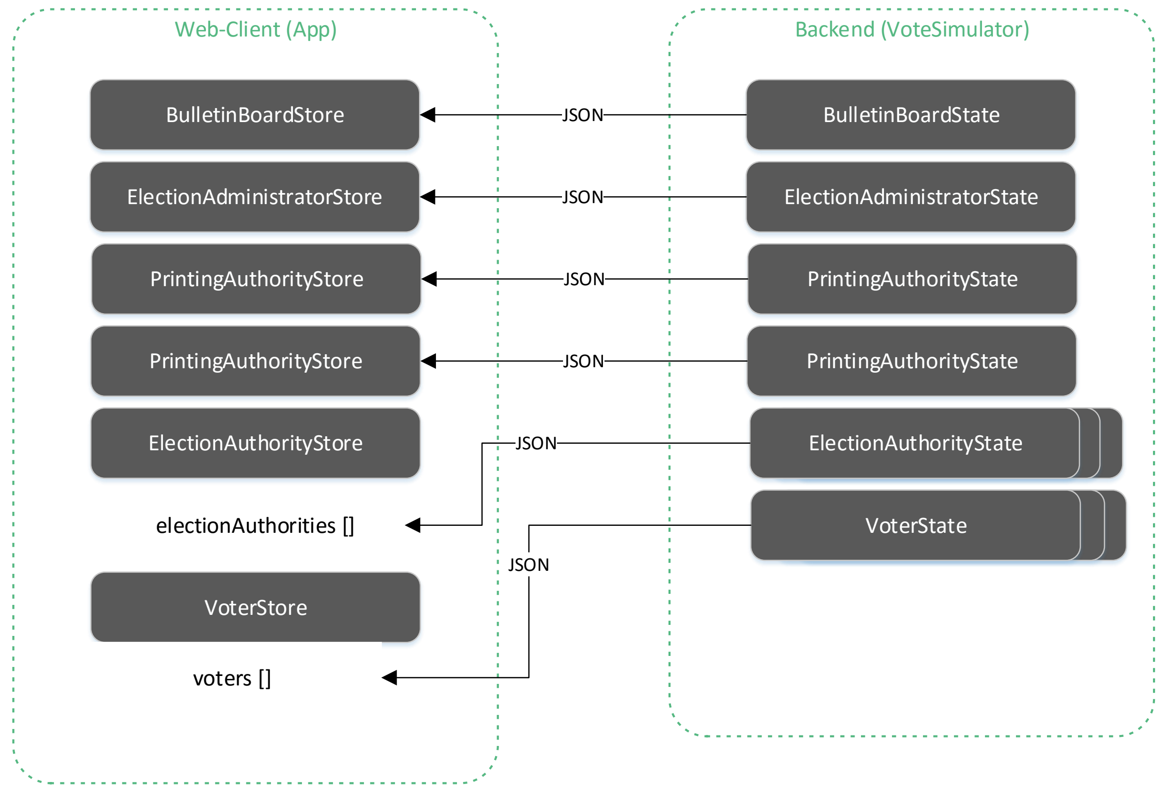


Figure 4.7: Datastores

After a client has pupulated his local data store, future manipulations on the backend are synchronized using the so called JSON Patch operations, which only contain the delta between the previous and the current state.

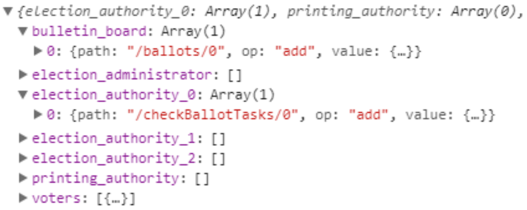
JSON Patch is a structure for describing how a JSON document can be modified / "‘patched"’. The procedure is standardized and described in the RFC 6902 of the Internet Engineering Task Force (IETF). There exist JSON Patch implementations for many languages, including Python and JavaScript. We used JSON Patch to realize our incremental data synchronization as follows:

When the VoteService loads the state of its actor objects, it sets the `originalState` property of the actor to a deep copy of the loaded state object. Mutations are always done only to the `state` property. Before calling the `persist()` method on an actor object, we use the Python JSON Patch library to create a set of JSON Patch operations that describes how to patch the `originalState` such that it becomes identical to the manipulated `state` object, by calling `make_patch(json.loads(self.originalState.toJSON()), json.loads(self.state.toJSON()))`

The result is a JSON array of operations that contains:

- The path of the manipulation
- The type of operation (replace, add, remove, ...)
- The new value (if required)

For example, after casting a ballot, we might receive the following JSON patch:



```
▼ {election_authority_0: Array(1), printing_authority: Array(0),
  ▼ bulletin_board: Array(1)
    ▶ 0: {path: "/ballots/0", op: "add", value: {…}}
  ▶ election_administrator: []
  ▼ election_authority_0: Array(1)
    ▶ 0: {path: "/checkBallotTasks/0", op: "add", value: {…}}
  ▶ election_authority_1: []
  ▶ election_authority_2: []
  ▶ printing_authority: []
  ▶ voters: [{…}]
```

Figure 4.8: JSON Patch example

These JSON patches are pushed to all the clients that need to receive the mutations and are applied to the local datastore which contains the original state. After applying the JSON patch, the data store of all clients contains the same state of an election as the backend.

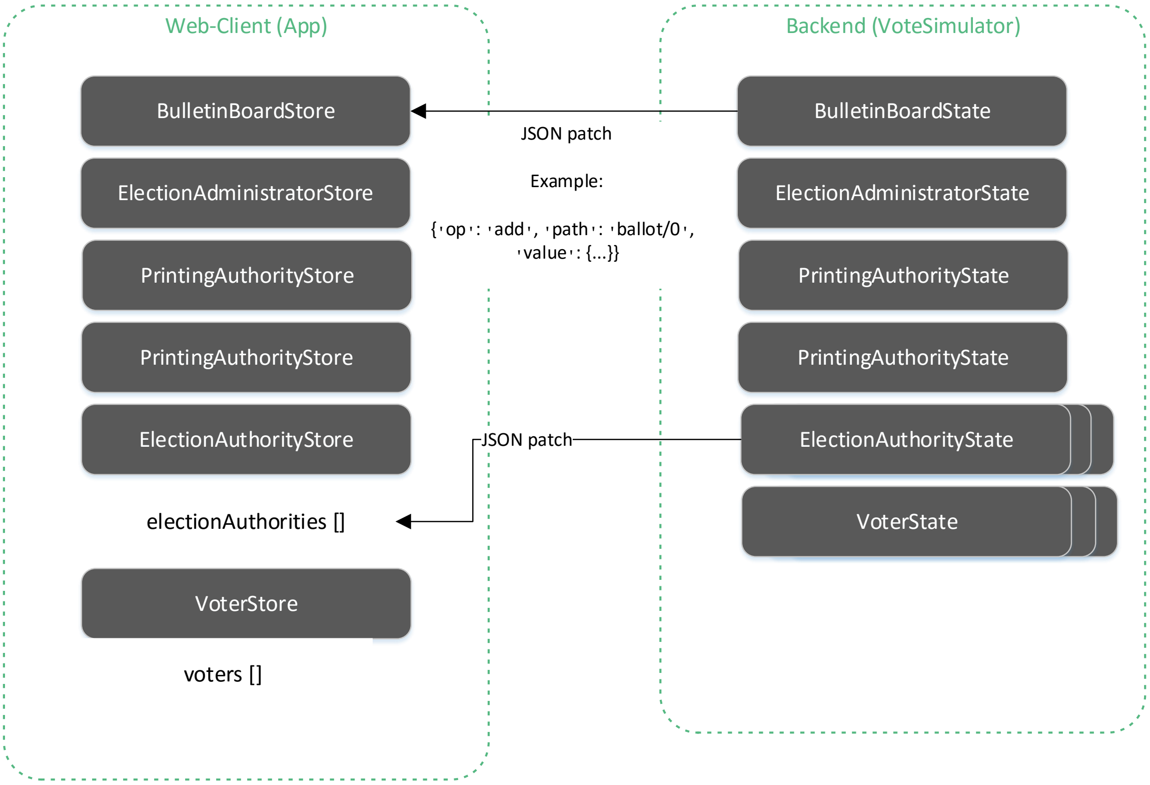


Figure 4.9: Data-Sync with JSON Patches

We decided not to use JSON patches when populating the data store for the first time, since generating JSON patches to patch an empty object to the full state of an election could result in thousands of operations that would almost certainly perform worse than sending the whole object over a fast websocket connection.

4.3.3 REST API

The third component of the backend is the REST API. Its responsibility is to serve as an API and to provide all the functionality of the VoteService to the webclients. Every function, such as the `castVote()` has a corresponding endpoint in the REST API service. Since almost every API endpoint looks almost the same, we take a look at one of them:

```
@main.route('/castVote', methods=['POST'])
@cross_origin(origin='*')
def castVote():
    data = request.json
    electionId = data["election"]
    selection = data["selection"]
    voterId = data["voterId"]
    votingCode = data["votingCode"]

    if len(selection) == 0:
        return make_error(400, "Empty selection")

    try:
        sim = VoteSimulator(electionId) # prepare voteSimulator

        sim.castVote(voterId, selection, votingCode) # perform action

        patches = sim.persist()  # persist modified state and retrieve JSON
         ↪  patches

        syncPatches(electionId, SyncType.ROOM, patches)  # send the JSON
         ↪  patches to all clients

    except Exception as ex:
        return make_error(500, str(ex))

    return json.dumps({'result': 'success'})
```

The API can be reached by sending a HTTP(S) POST request to our webserver hosting the backend. The URL defines what function will be executed. For example: A POST request to https://<server>:5000/castVote/ would call the above function. The required parameters are added in the POST body.

As a first step, parameters are extracted from the POST request and validated if necessary. As the next step , a VoteSimulator object is instantiated by passing the electionId to the constructor. The VoteSimulator will internally load the states of the corresponding election from the database and instantiate the actor objects such as the election authorities.

Now the VoteSimulator can execute the function which the user intended to call, for example "‘CastVote"’. By calling the function `persist()`, the new state is written to the database and the JSON Patches of all mutations are determined, returned and can be sent to all clients by calling the Data-Sync service.

The following sequence diagram shows how the vote casting use case is implemented within the backend and how all the components work together.

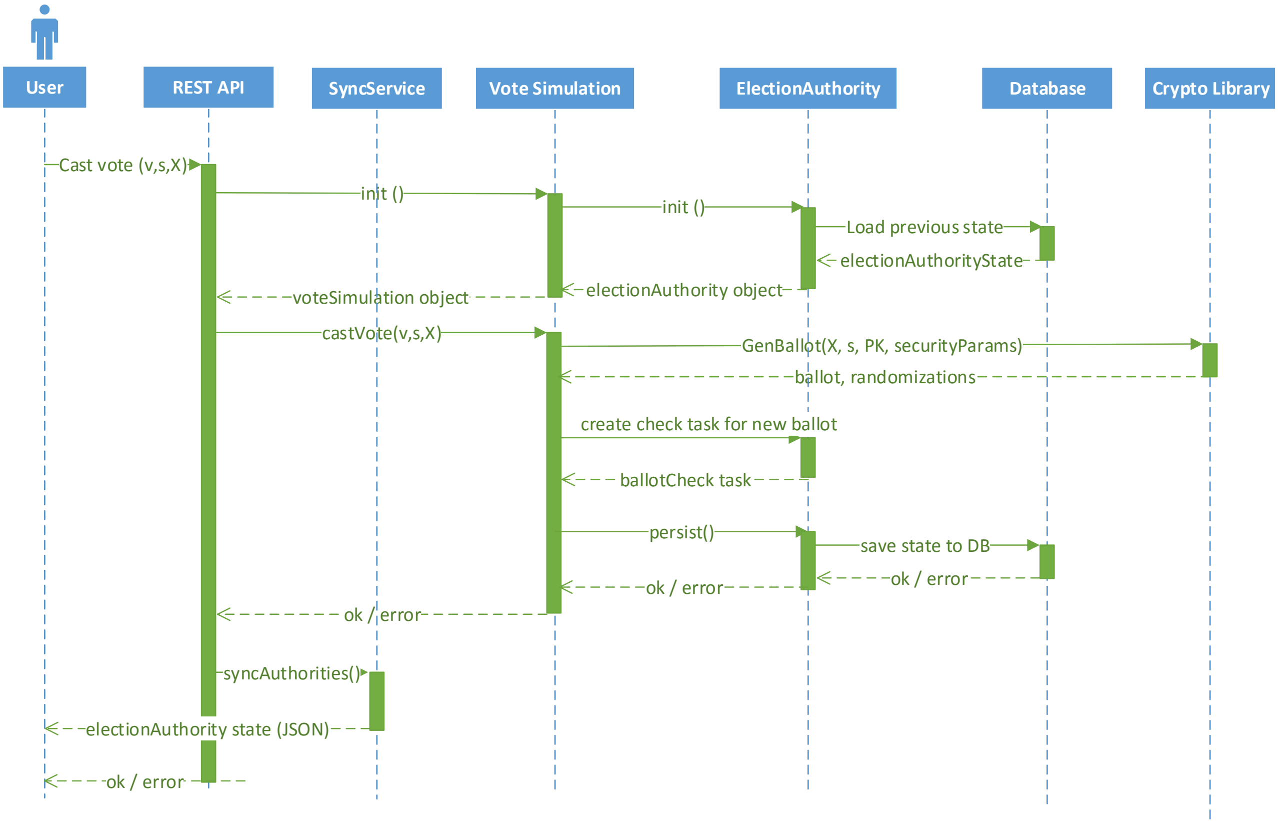


Figure 4.10: Vote casting sequence diagram

4.4 Crypto-library

4.4.1 File structure

We decided to put every algorithm of the specification in its own file together with related unit tests. The files are structured according to the actors of the protocol, for example:

- **Common**: contains common cryptopgraphic algorithms and the security parameters used by multiple algorithms

- **ElectionAuthority**: contains all the algorithms used by the election authority
- **PrintingAuthority**: contains all the algorithms used by the printing authority
- **VotingClient**: contains all the algorithms used by the voting client
- **ElectionAdministration**: contains all the algorithms used by the election administrator
- **Utils**: contains helper classes and miscellaneous utility functions

4.4.2 Public parameters

There exist two types of public parameters:

The **security relevant parameters**, e.g:

- The order of the prime groups: p, ${}'p$, $\hat{p}$
- The length of the voting, confirmation, return and finalization codes
- The number of authorities: s

and **public election parameters**, e.g.:

- The size of the electorate: N_E
- The number of candidates: n
- The list of candidate descriptions: c

The security parameters are typically used within the algorithms and remain unchanged for a longer time period, whereas the public election parameters are only used by the protocol implementations and change with every election.

The object `SecurityParams` holds all security relevant parameters and is injected as an additional function argument to all algorithms. Several different `SecurityParams` objects are created initially, which contain all the parameters according to the recommendations in the CHVote specification document ("level 0" for testing purposes and "level 1" through "level 3" for actual use of the protocol). This approach allows us to use different levels of security during development of the algorithms and protocols. For simple unit testing we used "level 0" in order to inject the security parameters recommended for testing puposes. For actual test runs of the project the security parameters from "level 2" were used.

The public election parameters on the other hand are directly passed to the algorithms by the calling party. If an algorithm needs to know certain election parameters (like the size of the electorate N_E), these values are typically derived from vectors that they have access to, so they do not require specific knowledge of these parameters.

4.4.3 Coding style

The following source code sample shows a typical implemation of an algorithm (in this exmaple, algorithm 7.18 according to the CHVote specification).

```
import unittest
import os, sys
from gmpy2 import mpz
import gmpy2

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from Utils.Utils                    import AssertMpz, AssertList, AssertClass,
 ↪  AssertString
from Crypto.SecurityParams          import SecurityParams, secparams_l0
from Utils.ToInteger                import ToInteger
from VotingClient.GetSelectedPrimes import GetSelectedPrimes
from VotingClient.GenQuery          import GenQuery
from VotingClient.GenBallotProof    import GenBallotProof
from UnitTestParams                 import unittestparams
from Types                          import Ballot
from Utils.StringToInteger          import StringToInteger

def GenBallot(X_bold, s, pk, secparams):
    """
    Algorithm 7.18: Generates a ballot based on the selection s and the voting
 ↪  code X. The
    ballot includes an OT query a and a proof pi. The algorithm also returns
 ↪  the random
    values used to generate the OT query. These random values are required in
 ↪  Alg. 7.27
    to derive the transferred messages from the OT response, which itself is
 ↪  generated by Alg. 7.25.

    Args:
        X_bold (str):                       Voting Code X ∈ A_X^l_X
        s (list of int):                    Selection s = (s_1, ... , s_k), 1
 ↪  <= s_1 < ... < s_k
        pk (mpz):                           ElGamal key pk ∈ G_p \ {1}
        secparams (SecurityParams):         Collection of public security
 ↪  parameters

    Returns:
        tuple:                              alpha = (r, Ballot) = (r, (x_hat,
 ↪  a, b, pi))
    """

    AssertMpz(pk)
    AssertList(s)
    AssertClass(secparams, SecurityParams)

    x = mpz(StringToInteger(X_bold, secparams.A_X))
    x_hat = gmpy2.powmod(secparams.g_hat, x, secparams.p_hat)

    q_bold = GetSelectedPrimes(s, secparams)                     # q = (q_1,
    ↪   ... , q_k)
    m = mpz(1)

    for i in range(len(q_bold)):
        m = m * q_bold[i]

    if m >= secparams.p:
        return None

    (a_bold, r_bold) = GenQuery(q_bold, pk, secparams)
    a = mpz(1)
    r = mpz(0)

    for i in range(len(a_bold)):
        a = (a * a_bold[i]) % secparams.p
        r = (r + r_bold[i]) % secparams.q

    b = gmpy2.powmod(secparams.g,r, secparams.p)
    pi = GenBallotProof(x,m,r,x_hat,a,b,pk, secparams)
    alpha = Ballot(x_hat,a_bold,b,pi)

    return (alpha, r_bold)

class GenBallotTest(unittest.TestCase):
    def testGenBallot(self):
        selection = [1,4]       # select candidates with indices 1,4
        (ballot, r) = GenBallot(unittestparams.X, selection,
         ↪  unittestparams.pk, secparams_l0)
        print(ballot)
        print(r)

if __name__ == '__main__':
    unittest.main()
```

All algorithms contain a short description, which was taken as-is from the specification document, as well as a comment (Google-style documentation string), which can be used to automatically generate code documentation. The algorithm itself is implemented as close to the specification as possible, using the same variable names and (as far as the language supports it) similar control structures:

- The suffix `_bold` for emphasized (bold) variables, e.g. `p_bold` for $\mathbf{p}$
- The suffix `_hat` for variables with a hat, e.g. `a_hat` for $\hat{a}$
- The suffix `_prime` for variables with a prime, e.g. `a_prime` for a'
- etc.

Each file also contains unit test relevant to the specific algorithm (if unit testing was considered useful for the particular algorithm).

The following example shows the similarities between the algorithm pseudo code and the actual implmentation in Python:

```
Algorithm: GenBallot($X, \mathbf{s}, pk$)
Input: Voting code $X \in A_X^{\ell_X}$
       Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k$
       Encryption key $pk \in \mathbb{G}_q \backslash \{1\}$
$x \leftarrow \mathsf{ToInteger}(X)$
$\hat{x} \leftarrow \hat{g}^x \bmod \hat{p}$
$\mathbf{q} \leftarrow \mathsf{GetSelectedPrimes}(\mathbf{s})$
$m \leftarrow \prod_{i=1}^{k} q_i$
if $m \geqslant p$ then
    return $\perp$
$(\mathbf{a}, \mathbf{r}) \leftarrow \mathsf{GenQuery}(\mathbf{q}, pk)$
$a \leftarrow \prod_{i=1}^{k} a_i \bmod p$
$r \leftarrow \sum_{i=1}^{k} r_i \bmod q$
$b \leftarrow g^r \bmod p$
$\pi \leftarrow \mathsf{GenBallotProof}(x, m, r, \hat{x}, a, b, pk)$
$\alpha \leftarrow (\hat{x}, \mathbf{a}, b, \pi)$
return $(\alpha, \mathbf{r})$
```

```python
x = mpz(StringToInteger(X_bold, secparams.A_X))
x_hat = gmpy2.powmod(secparams.g_hat, x, secparams.p_hat)
q_bold = GetSelectedPrimes(s, secparams)

m = mpz(1)
for i in range(len(q_bold)):
    m = m * q_bold[i]

if m >= secparams.p:
    return None

(a_bold, r_bold) = GenQuery(q_bold, pk, secparams)
a = mpz(1)
r = mpz(0)

for i in range(len(a_bold)):
    a = (a * a_bold[i]) % secparams.p
    r = (r + r_bold[i]) % secparams.q

b = gmpy2.powmod(secparams.g,r, secparams.p)
pi = GenBallotProof(x,m,r,x_hat,a,b,pk, secparams)
alpha = Ballot(x_hat,a_bold,b,pi)

return (alpha, r_bold)
```

4.4.4 Return types

In most cases, when an algorithm returns more than a scalar datatype, tuples are used. Tuples allow to return multiple values from a function:

```python
def foo():
    return (1, 2)

def main():
    a, b = foo()
```

This way a lot of the source code looked very similar to the pseudo code in the CHVote specification. For more complex data types or return values that are used more often, named tuples were used. The data type "namedtuple" is like a lightweight class and allows access to named properties.

```python
Ballot = namedtuple("Ballot", "x_hat, a_bold, b, pi")

def main():
   Ballot b = getBallot()
   x_hat = b.x_hat
```

By following this approach we can avoid having lots of container classes only used to pass data structures between the algorithms.

4.5 Frontend

Most of the work during our project had to be done for the frontend. Displaying the rather large amount of voting specific data and large numbers required a clean and well structured layout and a modular component design. Luckily, the framework we had choosen, VueJS, did very well in supporting exactly those requirements. We tried to follow the design patterns and best practices proposed by the makers of VueJS wherever possible.

In this section we will explain what concepts of VueJS we used and how we adapted them to our needs.

4.5.1 Components

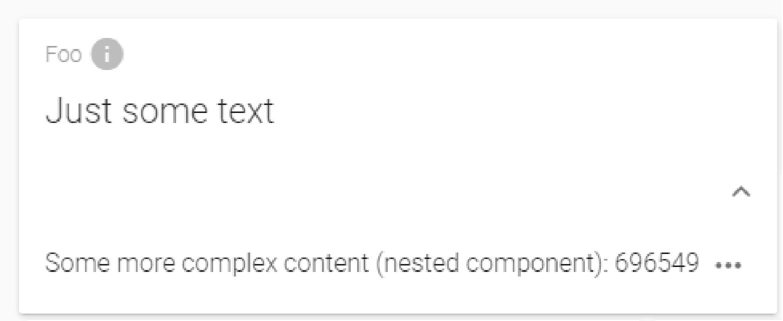
Components are the basic building blocks of the VueJS framework. The application itself is a component, every page of the application is a component and the pages typically contain lots of components, one for every control like form controls, buttons or custom controls such as the ballot-list etc., which themselves may again consist of multiple components. The concept of components encourages to create reusable modules, provides a way to structure the application and divide it into smaller units and makes the resulting HTML template more expressive and easier to read.

We have created our own VueJS components for every control that we used more than once. For example the ballot list that is shown both on the bulletin board as well as in the election authority view, the labels for displaying truncated large numbers or the cards used as our standard mean for displaying data have all been turned into a custom component. One of the beautiful features of VueJS components is the concept of slots. By defining one or multiple slots within a components template markup, it becomes possible from the parent of a component, to embed content into different locations (slots) within the components HTML template.

We have been using slots to create our card component, which can display information either as the main content of the card, or within an expandable area on the bottom of the card:

```
<DataCard title="Foo" :expandable=true>
    Just some text
    <p slot="expandContent">More complex content <BigIntLabel
     ↪  :mpzValue="publicKey"></BigIntLabel>
    </p>
</DataCard>
```

The first line "‘Just some text"’, which gets inserted into the default unnamed slot, could as well be passed as a string parameter to the data card component. However, as things are getting more complicated, one might like to place arbitrary HTML or even another VueJS component inside the expandable content of our datacard. In such cases, component parameters wont work as they only accept primitive data types. Slots on the other hand allow arbitrary content. The



Foo

Just some text

Some more complex content (nested component): 696549 •••

Figure 4.11: Vote casting sequence diagram

following code shows how the data-card component is implemented:

```
<template>
    <v-card class="dataCard">
        <ConfidentialityChip v-if=showConfidentiality"'
         ↪  :type="confidentiality" class="confidentialityChip" />
        <v-card-title primary-title class="dataCardTitle">
            <div><span class="label grey--text">{{title}}
              <v-tooltip top>
                <v-icon v-if="!disableTooltip" color="grey lighten-1"
                 ↪  slot="activator">info</v-icon><span>Programmatic
                 ↪  tooltip</span>
             </v-tooltip></span>
            </div>
        </v-card-title>
        <v-card-text class="dataCardContent">
                <slot></slot>
        </v-card-text>
        <v-card-actions v-show="expandable">
            <v-btn icon @click.native="showExpander = !showExpander">
                <v-icon>{{ !showExpander ? 'keyboard_arrow_down' :
                 ↪  'keyboard_arrow_up' }}</v-icon>
            </v-btn>
        </v-card-actions>
        <v-slide-y-transition v-show="expandable">
            <v-card-text v-show="showExpander">
                <slot name="expandContent">
                </slot>
            </v-card-text>
        </v-slide-y-transition>
    </v-card>
</template>
<script>
    import { mapState } from 'vuex'

    export default {
      data: function () {
        return {
          showExpander: false
        }
      },
      computed: {
        ...mapState({
          showConfidentiality: state => state.showConfidentiality
        })
      },
      props: {
        title: {
          type: String,
          required: true,
          default: 'Title'
        },
        expandable: {
          type: Boolean,
          required: true,
          default: false
        },
        confidentiality: {
          type: String,
          required: true,
          default: 'public'
        },
        disableTooltip: {
          type: Boolean,
          required: false,
          default: false
        }

      },

        mounted () {
        }
    }
</script>
```

The first part within the `<template>` tag describes the HTML markup as well as the slots that we have just mentioned.

The `<script>` tag contains the actual logic of the component. The `data` object contains variable which are defined and valid only locally within the component. The `computed` object maps variables from our central data-store to a local variable which is reactively bound to the data-store. Whenever the value of the given variable changes in the data-store, the computed property is automatically updated in this component. From the template, we can access both local data as well as computed properties. Computed properties can also be used whenever a local variable needs to be formatted, or in some way manipulated for displaying in the components template.

The third source of data are 'props', which are passed as arguments from the parent component. They are typically used to defined options for a component.

Additionally, components may contain `methods`, typically used for event-handlers like button clicks and event hooks like `mounted`, `beforeDestroy`, `beforeCreated` to influence the components creation/destruction at different times during the components lifecycle

4.5.2 Centralized Data-store & Flux pattern

One of the big challenges regarding the architecture of our front-end, was about how and where we would store all the data of an election. Clearly, since we have already divided the elections data into one state for every actor and given that every actor also has it's corresponding view in our front-end, simply storing the data to the respective component has been our first thought. Although a voter mainly needs to access his own data contained in the voters state, some data is to be shared between components, for example the information on the bulletin board.

Since we wanted to avoid having to keep data redundant in multiple components, we have decided to use the Flux design pattern for our front-end. The basic idea of the flux pattern is to have a single, central data source where all the data is stored and which all components have access to. This single data source is called a "'store"' by Flux terminology. VueJS has it's own implementation of the Flux pattern called Vuex. Another important concept is that components can freely access the data in the store, however, they are not allowed to change data, at least not directly. Instead, if a component wants to change data in the store, this has to be done by calling **mutation functions**. Not allowing direct manipulation of the store makes it much easier to keep track of where mutations came from.

Our web applications data store is divided into multiple modules, one data store module for each corresponding state of the back-end. In reality, all these data stores are part of one single data store, but having multiple modules allows us to structure the mutation and getter functions and help to avoid naming conflicts by having separate namespaces for every module.

The state of a data-store can be accessed directly from any component by defining a computed property. If the computed properties have to do perform some formatting, aggregation, filtering etc. on the state variables, it is recommended to write getter-functions in the data-store to avoid writing mu

4.5.3 Internationalization (i18n)

TODO: I18n

4.5.4 Development Environment

TODO: Webpack, ESlint etc. beschreiben

4.6 Challenges

In this section we describe some of the challenges that we encountered while developing our application.

4.6.1 Websocket subscription concept

Since our application allows that multiple elections exist at the same time, the question has arised how we could handle that only those clients who are actually observing a particular election receive web-socket messages when an action has been triggered in the respective election.

We have seen similarities between our problem and the one a typical chat has that consists of mutliple chat-rooms in which only those users should be notified of new posts that have actually joined the room. We have adapted this "‘chat-room"’ concept to our problem by defining a room to be equal to an election.

We can assume that all the pages that actually show data of an election require the elections id to be passed as part of the URL. For example: /BulletinBoard/1 is the route to reach the BulletinBoard view of the election with id 1.

Whenever a route is called that contains the argument ‘electionId‘, we need to make sure that the client has joined this elections room. We therefore set a global variable called python/joinedElectionId/ to match the id of the election a user has joined. We created a mixin that can be added to every election page, which makes sure that if the client hasn’t yet joined the corresponding election, it emits a "‘join"’ request to the socket.io server, passing along the electionId:

```
export default {
  created () {
    if (this.$store.getters.joinedElectionId !== this.$route.params.electionId) {
      this.$socket.emit('join', {election: this.$route.params['electionId']})
    }
  },
}
```

On the server side, we have defined a socket.io listener called "‘join"’ which will remove the calling client from every room before joining the room of the requested election. There is only exception: The client cannot leave the room that corresponds to the ‘sid‘ of the request, since this is basically the channel over which the ‘join‘ request is handled.

```
@socketio.on('join')
def on_join(data):
    from app.api.syncService import SyncType, fullSync

    electionID = data['election']
    for room in rooms():
        if room != request.sid:
            leave_room(room)
    join_room(electionID)

    from app.api.syncService import emitToClient, SyncType

    fullSync(electionID, SyncType.SENDER_ONLY)

    emitToClient('joinAck', electionID, SyncType.SENDER_ONLY)
```

The `joinAck` handler in the web application will then set the `joinedElectionId` variable to the just joined election id, such that the join will only be called once or until the user chooses a different election.

4.7 Automatic Task Processing for election authorities

In our application, every election authority normally has to manually process all incoming tasks such as ballot checking, confirmation checking, mixing and decrypting. As this can become cumbersome and it might be desirable to perform these tasks manually only for the first election authority, we have implementation an auto-mode in which an election authority automatically performs all tasks.

There were several different ways how to implement this feature. One possibility would have been building a service which regularly checks for new tasks and processes them. Since every task requires a preceding action (for example: A Ballot-Check-Task requires a voter casting a vote), and since it is reasonable for our use-cases to assume that the authorities perform the tasks sequentially, we have chosen the most simple approach:

When a user casts a ballot and a ballot-check-task is created, we check if the first authority is set to automatic. If yes, the function for checking this ballot is automatically called. This function again checks, if the next election authority is set to automatic and recursively calls itself if that's the case, and so on until one authority has auto-mode disabled.

This means that if the first authority is in manual mode and the other two are set to automatic, they both wait with their execution until the first election authority has manually started executing the task. This strict sequential order is only required for the mixing task, all other tasks could be called in any order. If desired, this behavior could also be implemented with our approach, however, for the moment this is not required.

5 Conclusion

During the first few weeks we felt as if we have been thrown into cold water. Reading and understanding the protocol wasn't easy at first, because we had to get used to the notation and memorize a large amount of variables used by the many algorithms. While some of the cryptographic primitives were taught in previous courses, most of them were new and unknown to us. We focused on getting a good understanding of the protocol on a higher level rather than learning about each and every algorithm in detail, as this was sufficient for implementing and understanding the protocol.

Additionally, programming algorithms isn't something we are doing on a daily basis. Therefore, the first few algorithms took us quite some time to implement. After a few weeks, we could greatly increase our productivity and in the end, we could implement even the larger algorithms in not much more time than the simple ones in the beginning of the project.

From our perspective, the project has been extremely interesting and we are still impressed by the ideas presented and specified in the CHVote specification. From simply implementing the protocol we could learn a lot about the CHVote protocol and E-Voting in general and could improve both our knowledge of more advanced cryptographic topics and get practise in implementing cryptographic algorithms.

5.1 Python drawbacks

During the project we have experienced a few issues with the programming language that we used to implement the specification in, Python. In particular, we have observed the following issues:

- Performance issues due to Python being an interpreted language
- Function overhead: function calls in Python seem to be quite slow
- Strongly dynamic typing vs. static typing: the Python interpreter needs to inspect every single object during run time (be it an integer or a more complex object)
- The *BigInteger* library surprisingly isn't as fast as using directly the GMP library
- Larger projects tend to turn out messy
- Little to no standard documentation regarding project structure

- No real standard for unit testing, documentation generation etc.

For detailed information regarding the performance issues that we have experienced see [2] and [3]. Based on the reasons above we would not recommend to use Python for the use in similar or larger project. Python is indeed a very handy language to write quick prototypes and proof of concepts, but issues become more frequent in larger projects.

Bibliography

[1] "CHVote System Specification", by Rolf Haenni, Reto E. Koenig, Philipp Locher and Eric Dubuis, April 11, 2017.

[2] "Why Python is Slow: Looking Under the Hood", by Jake VanderPlas, see http://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/

[3] "Python speed: performance tips", from the official Python wiki, see https://wiki.python.org/moin/PythonSpeed/PerformanceTips

6 Appendix

6.1 Use Cases

Table 6.1: Use Case «Create new election»

Use Case	**Create new election**
Primary Actor	User
Description	The system allows to create new elections
Precondition	The system shows the available elections in a list
Main path (M)	1. User clicks on "‘create election"’ 2. System demands a name for the election 3. User is redirected to the election overview page

Table 6.2: Use Case «Set up election»

Use Case	**Set up election**
Primary Actor	Election Administrator
Description	The election administrator can set up the election. This involves the generation of the cryptographic electorate data in the backend
Precondition	An new election has been created
Postcondition	The election has the status "‘Printing"’
Main path (M)	1. Election Administrator visits the "‘Election Administrator"’-view of a new election. 2. The system demands the following information: • Number of parallel elections • Candidates per election • Number of possible selections per election event • Number of voters • Counting circles of the voters 3. User clicks on "‘Generate"’

Table 6.3: Use Case «Printing of voting cards»

Use Case	Printing of voting cards
Primary Actor	Printing Authority
Description	The printing authority generates voting cards
Precondition	The election has the status "‘Printing"’
Postcondition	The election has the status "‘Delivery"’
Main path (M)	1. The election administrator visits the "‘Printing Authority"’-view of an election. 2. The election administrator clicks on "‘Print Voting Cards"’ 3. A list of all voters is displayed 4. The election administrator can select a voter to see his voting card

Table 6.4: Use Case «Delivery of voting cards»

Use Case	Delivery of voting cards
Primary Actor	Printing Authority
Description	The printing authority can send the voting cards to the voters
Precondition	The election has the status "‘Delivery"’
Postcondition	The election has the status "‘Election Phase"’
Main path (M)	1. The election administrator visits the "‘Printing Authority"’-view of an election. 2. The election administrator clicks on "‘Deliver Voting Cards"’ 3. Within the voters-view, the voting card shows up for every voter

6.2 Journal

6.2.1 Week 1

6.2.2 Kickoff Meeting

During our kickoff meeting we discussed the possibilities of our bachelor thesis based on the spadework of the previous "project 2" module and broke them down into two options: A realistic prototype of the whole CHVote system which includes everything a real implementation would need, like signatures, channel security, distributed election authorities with docker etc., or to build a demonstrator tool (an application that allows to demonstrate the functionality of the chVote protocol in a more visual manner).

Table 6.5: Use Case «Confirmation of a vote»

Use Case	**Confirmation of a vote**
Primary Actor	Voter
Description	The voter can confirm his vote by verifying the verification codes and entering his confirmation code
Precondition	• The election has the status "'Election Phase"' • A voter is selected in the "'Voter"'-view • The voter has the status "'Confirmation Phase"'
Postcondition	The first election authority receives a "'Check-confirmation task"'
Main path (M)	1. The voter visits the "'Voter"'-view and select his voter object 2. The system displays the verification codes of the selected candidates 3. The voter must manually verify that the displayed codes match the verification codes of the selected candidates on his voting card 4. The system demands the confirmation code 5. The voter clicks on "'Confirm vote"'

6.2.3 Week 2

In week 2 we have made the decision to build the demonstrator mainly because the final product would potentially be more attractive visually than a prototype where the main work lies in the background which is not visible to an outsider. We have also started thinking about what technologies and frameworks to use and to build a few sketches and mockups to have some basis for discussion for our next meeting.

We have also started updating our implementation of the CHVote crypto-library to the latest specification. Since we ran into a few problems, this took us almost the whole week.

6.2.4 Reflexion

At this stage we have yet been very unsure about how the application should look like, what audience we should have as our main target and what functionality the application should offer.

Table 6.6: Use Case «Checking a ballot»

Use Case	**Checking a ballot**
Primary Actor	Election Authority
Description	The election authority can verify the validity of a ballot and respond to the voters query
Precondition	• The election has the status "‘Election Phase"’ • The currently selected election authority has a new "‘Check ballot task"’
Postcondition	• The next election authority receives a "‘Check ballot task"’ • If this election authority was the last one, and the ballot was valid, the voter now has the status "‘Confirmation Phase"’
Main path (M)	1. The user visits the "‘Election Authority"’-view and select one of the available election authorities that has new "‘Check ballot task"’ 2. The system displays the query, the ballot proof and the voting credential of the voter 3. The user click on "‘Check validity"’ 4. The system displays the result of the validity check 5. The user clicks on "‘Respond"’

6.2.5 Week 3

During our second meeting we discussed the further elaborated the goals, the structure of our project and talked about the audience.

- In essence, the application should allow to demonstrate a chVote election from the view of every party participating in the election process.
- The application should be a real-time webapp that updates the views automatically as soon as something changes and without having to reload the page

6.2.6 Week 4

In the fourth week we continued describing the goals and further worked on the system architecture. We also made some first experiences with the envisaged frameworks and technologies (VueJS, socket.io, Flask, MongoDB).

Table 6.7: Use Case «Checking a confirmation»

Use Case	**Checking a confirmation**
Primary Actor	Election Authority
Description	The election authority can verify the validity of a confirmation and respond to the voters query
Precondition	• The election has the status "‘Election Phase"’ • The currently selected election authority has a new "‘Check ballot task"’
Postcondition	• The next election authority receives a "‘Check confirmation task"’ • If this election authority was the last one, and the confirmation was valid, the voter now has the status "‘Finalization Phase"’
Main path (M)	1. The user visits the "‘Election Authority"’-view and select one of the available election authorities that has new "‘Check confirmation task"’ 2. The system displays information about the confirmation 3. The user click on "‘Check validity"’ 4. The system displays the result of the validity check 5. The user clicks on "‘Finalize"’

6.2.7 Reflexion

- working with socket.io and VueJS has been very intuitive and looked very promising and suitable for our project
- We were not yet sure whether or not mongoDB is the right technology for our needs.

During prototyping, we observed that our first architecture approach of doing everything over websockets, turned out to be a bad decision.

6.2.8 Week 5

In the fifth week we started with the real implementation.

Table 6.8: Use Case «Mixing»

Use Case	**Mixing**
Primary Actor	Election Authority
Description	Every election authority can perform the mixing on the extracted list of encryptions
Precondition	• The election has the status "‘Mixing"’ • The previous election authority has already performed the mixing
Postcondition	• The next election authority is able to mix
Main path (M)	1. The user visits the "‘Election Authority"’-view and select one of the available election authorities that hasn’t mixed before 2. The system displays the list of encryptions of the previous election authority (or the first one in case the first election authority is selected) 3. The user clicks on "‘Mix"’ 4. The new, mixed list of encryptions is added to the known data of this election authority

6.2.9 Reflexion

- working with socket.io and VueJS has been very intuitive and looked very promising and suitable for our project
- We were not yet sure whether or not mongoDB is the right technology for our needs.

During prototyping, we observed that our first architecture approach of doing everything over websockets, turned out to be a bad decision.

Table 6.9: Use Case «Decryption»

Use Case	**Decryption**
Primary Actor	Election Authority
Description	Every election authority can perform the (partial) decryption
Precondition	• The election has the status "'Decryption"' • The previous election authority has already performed the decryption
Postcondition	• The next election authority is able to decrypt
Main path (M)	1. The user visits the "'Election Authority"'-view and select one of the available election authorities that hasn't decrypted before 2. The system displays the list of encryptions 3. The user clicks on "'Decrypt"' 4. The list of partial decryptions is added to the known data of this election authority

Table 6.10: Use Case «Tallying»

Use Case	**Tallying**
Primary Actor	Election Administrator
Description	The election administrator can perform the tallying and view the final result
Precondition	The election has the status "'Tallying"'
Postcondition	The has the status "'Finished"'
Main path (M)	1. The user visits the "'Election Administrator"'-view 2. The user clicks on "'Tally"' 3. The final result is added to the known data of the election administrator