

Department of Computer Science and Engineering  
The Chinese University of Hong Kong

ESTR4999 Graduation Thesis Report

LYU1803

# **Opensource E-voting System for 8 million mobile devices**

Written by

Chan Maxwell Chun Sum (1155079378)

Supervised by

Prof. Lyu Rung Tsong Michael

12 April 2019

# Table of Contents

<b>Abstract .....</b>	<b>6</b>
<b>Acknowledgements .....</b>	<b>7</b>
<b>1. Introduction .....</b>	<b>8</b>
1.1. Motivation.....	8
1.2. Background .....	9
1.2.1. E-Voting consideration .....	9
End-to-end verifiability .....	9
Voter authentication & anonymity .....	10
1.2.2. Blockchain .....	11
1.3. Objective .....	13
<b>2. Related Work.....</b>	<b>15</b>
2.1. E-voting in Hong Kong .....	15
2.2. End-to-end verifiable voting .....	15
2.2.1. Helios.....	16
2.2.2. Pretty Good Democracy .....	17
2.2.3. In-person voting systems.....	18
2.3. E-voting using blockchain.....	19
2.3.1. A vote as a coin.....	19
2.3.2. Use blockchain for secure storage .....	20
<b>3. Design .....</b>	<b>21</b>
3.1. Overview .....	21
3.2. Algorithm related to the voting process .....	21
3.2.1. Helios cryptography .....	21
Create election .....	21
Ballot preparation .....	22
Tallying election.....	22
3.2.2. Helios zero-knowledge proof .....	23
Trustee knowledge on private key .....	24
Trustee honest decryption .....	25
Voter honest encryption .....	26

3.2.3.	Limitation on Helios and proposed modification.....	27
	Denial of service attack .....	27
	Slow tallying .....	28
	Coercion .....	30
	Authentication method.....	31
	Knowledge of who has voted .....	34
3.3.	Blockchain related protocol .....	34
3.3.1.	Type of blockchain.....	34
3.3.2.	Roles and permission .....	35
3.3.3.	Design a blockchain protocol for voting.....	37
3.3.4.	Block design.....	38
3.3.5.	Handshake protocol design .....	39
3.3.6.	Ballot submission protocol design .....	40
3.3.7.	Block generation process design.....	41
	Node selection protocol.....	41
	Consensuses protocol .....	42
	Block broadcasting protocol.....	44
<b>4.</b>	<b>Implementation.....</b>	<b>46</b>
4.1.	Overview .....	46
4.2.	Server-side.....	46
4.2.1.	Use of programming language .....	46
4.2.2.	System architecture.....	47
	Architecture design overview .....	47
	HTTP listener .....	48
	Router.....	49
	Controller .....	49
	Library.....	50
	Database model .....	51
	Package.....	57
4.2.3.	Detailed explanation of components.....	58
	Connect to the blockchain network.....	58
	Create an election or change election details.....	59
	Add a voter/trustee.....	59
	Voter/Trustee change key pair .....	60
	Freeze an election .....	60
	Processing a submitted ballot .....	61
	Generate new block .....	61
	Tally an election.....	62

4.3.	Client-side.....	63
4.3.1.	Choosing the deliverables .....	63
4.3.2.	Use of programming language .....	63
4.3.3.	User interface .....	64
	Create election and editing details .....	64
	Election management page .....	65
	Question list editing .....	66
	Server list editing.....	67
	Voters/Trustees management.....	67
	Change key pair by voters/trustees.....	69
	Election home page.....	69
	Voting page.....	70
	Voter & ballot list .....	71
	Start tally an election .....	72
	Decrypt an election .....	72
	Election result page.....	74
<b>5.</b>	<b>Testing.....</b>	<b>75</b>
5.1.	Overview .....	75
5.2.	Load testing – First round .....	76
5.2.1.	Block length test.....	76
5.2.2.	Arrival rate test.....	79
	1 node .....	79
	2 nodes.....	81
	3 nodes.....	82
	5 nodes.....	83
5.2.3.	Ballot aggregation test .....	83
5.2.4.	Conclusion .....	84
5.2.5.	System modification.....	85
	Forking child process.....	85
	Database indexing.....	86
5.3.	Load testing – Second round.....	86
5.3.1.	Block length test.....	86
5.3.2.	Arrival rate test.....	88
	1 node .....	88
	2 nodes.....	90
5.3.3.	Ballot aggregation test .....	91
5.3.4.	Conclusion .....	92
5.3.5.	System modification.....	92

5.4. Reliability testing.....	93
<b>6. Conclusion.....</b>	<b>95</b>
6.1. Summary .....	95
6.2. Future work.....	96
6.2.1. Improvement on scalability.....	96
Use more child processes.....	96
Possibility of partially broadcasting ballots.....	97
6.2.2. Improvement on reliability.....	97
Ballot re-broadcasting .....	97
Smarter blockchain synchronization protocol.....	97
Synchronize clock on different nodes .....	98
6.2.3. Full implementation of the proposed design.....	98
6.2.4. Enforce more security measure .....	99
6.2.5. Use newer communication protocols .....	99
6.2.6. Possibility of enabling “Voting-as-a-service” .....	100
<b>Bibliography.....</b>	<b>101</b>

# Abstract

In this project, we focus on designing and developing an application that integrates electronic voting with blockchain technology. So that the application can promote verifiability and anonymity of e-voting, which can be easily expendable to organize an election of any scale. The aim is to let the public to utilize and learn about the significance of using a secure e-voting system.

There are many pieces of researches on an end-to-end verifiable voting system and blockchain based voting system, but not many of them have proposed the combination of both, plus an implementation of an online remote voting system. Therefore, in this report, we take Helios, one of the popular end-to-end verifiable voting systems, as a reference, and proposed improvements on the system with the use of the blockchain technology.

We have actually built the opensource application to prove most of our proposals. The application has also been tested under different scenarios, so that we could further improve it and suggest the system requirements for using it in different scales.

# Acknowledgements

We would like to express our greatest appreciation to our supervisor Prof. Michael Lyu and advisor Mr. Edward Yau for guiding us on this interesting project. They have given a lot of precious advice and feedback from doing research, designing, implementation to the presentation. In addition, they have also assisted in setting up machines and environment for our system testing. This project will not be completed without their help.

Besides, we would also take this opportunity to thanks everyone who has supported us in the project or read this report.

# 1. Introduction

## 1.1. Motivation

Voting for many elections in Hong Kong is often paper-based, no matter it is in small scale like electing a student union (Figure 1-1), or in large scale like legislative council election. The problem is that the process uses a lot of time and resources, from marking to counting ballots, and even discourage voter intention to vote in person. In this digital era, almost everyone should have a mobile phone, digitalize the voting process and allowing people to vote on their own devices should be a trend in the future. Therefore, develop an e-voting application is important for Hong Kong, not only to improve the efficiency and accuracy of voting, but also aim to increase the voter turnout rate, which is important to democracy in this digital era.

According to a discussion in the legislative council [1], the reason why e-voting is not popular in Hong Kong is mainly due to the mistrust between people, the government, and computer network. Counting the electronic ballot inside a computer is just like a black box, people cannot easily monitor the process. So, they may not confident with the result, especially when the government is able to control the computer. Besides, with those report of incidents about network security threat and personal data leak, people may concern about the stability of the voting system.

On the other hand, the blockchain technology is getting much more attention recently. Despite the popular application on those cryptocurrencies such as Bitcoin and Ethereum, it is also used commercially to provide reliable and trusted data to the public, just like the flight data by Hong Kong International Airport [2]. Blockchain is popular because it can provide transparency, auditability, decentralization, etc [3]. If an e-voting system contains these characteristics, it will be more secure and reliable. Therefore, we would like to study the possibility of intergrading blockchain technology with an e-voting application, such that it can be used and trusted by the public.



*Figure 1-1: A traditional voting booth in CUHK for student union election  
(Source: www.facebook.com)*

## 1.2. Background

### 1.2.1. E-Voting consideration

#### End-to-end verifiability

An e-voting system should be end-to-end verifiable to promote the integrity of ballots [4], such that the result from a tallied election correctly reflect the intention of all voters. The end-to-end verifiability includes three parts (Figure 1-2).

#### Cast-as-intended verification

A voter should be able to verify the submitted ballot correctly record his intent. In the case of an encrypted ballot, the voter needs to verify the encryption system is working well so that it will be equal to the original ballot if decrypted. Also, if there is any unauthorized modification to the ballot after submission, the voter can notice the change.

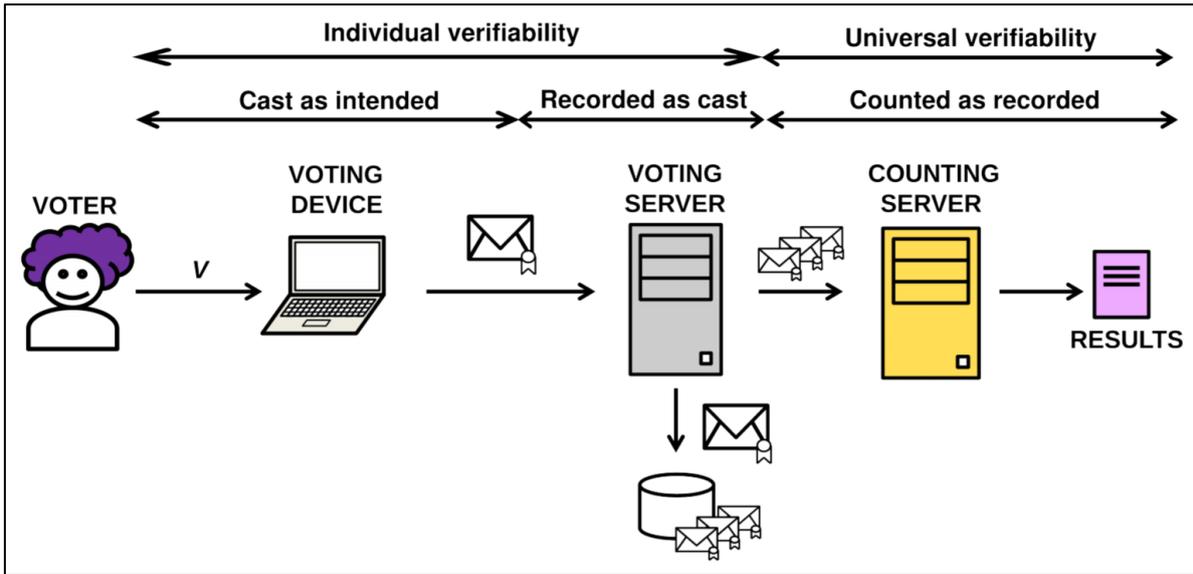


Figure 1-2: End-to-end verifiable voting (Source: fc16.ifca.ai)

### Record-as-cast verification

A voter can verify that the ballot saved on the server is the same ballot as he submitted. So that there is no error on ballot transmission, and the ballot has not been deleted any time before tallying.

### Tallied-as-recorded verification

A voter should be able to verify his vote is in the tally of an election result. More importantly, any voter can verify that a ballot is tallied if and only if it is valid, which means the tally should only include all ballots that voted by an eligible voter.

### **Voter authentication & anonymity**

There should be an authentication system so that only eligible voter for that election can enter the voting system. However, after authentication and submission, the ballot should be anonymous, so that nobody can tell the owner of the ballot and which options the voter has voted for.

## 1.2.2. Blockchain

Blockchain is a way of storing data in a system using a chain of blocks. A basic block should contain the sequence number, data, hash of this block, and the hash of the previous block (Figure 1-3). Modification to block data is hard by design, because a minor change in data will change the hash value, and thus affect all hash values in blocks after that. So, verification of the data can be easily done by checking the hash of each block, making the blockchain reliable.



Figure 1-3: An example of blockchain (Source: rubygarage.org)

Besides, blockchain is designed to be distributed and decentralized, so that everyone can become a node in the system and read the data. Therefore, this will require all nodes to agree on the generated block.

In a permission-less blockchain like Bitcoin, usually, a proof-of-work system is used [5] so that a nonce is needed to be calculated by the node that generates the block. The difficulty of nonce calculation can be adjusted (Figure 1-4) according to the number of nodes in the blockchain network. As a result, this calculation will be computationally intensive, so it is hard to have multiple nodes generates a block at the same time.

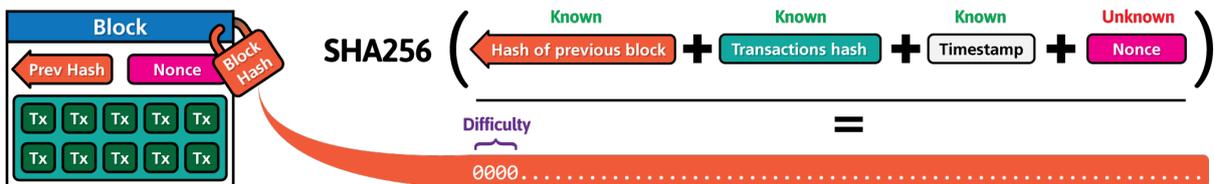


Figure 1-4: Block hash calculation using proof-of-work (Source: cryptographics.info)

In a permissioned blockchain, only some trusted server/node can process transactions and generate a new block. Therefore proof-of-work mechanism usually not

used, as the number of these nodes is relatively smaller [5]. Instead, a trusted node will be selected, may be randomly or in an order, to generate new block in a fixed time interval or after a certain amount of transactions arrived. The node's identity will also be stored into the new block for others to verify.

However, if there are some dishonest 'trusted nodes' in the permissioned blockchain, or the blockchain is not run in a trusted and reliable network, then a consensus algorithm is needed for all of them to agree on a node to generate new block every time.

Byzantine Fault Tolerance is one of the algorithms that usually used for this purpose [6], it is proven that having at least two-third honest nodes are enough for the consensus. Assume that there is a source that tells every node which node the block generator is, but the source is dishonest (Figure 1-5). According to the algorithm, every node will broadcast its piece of information to other nodes. After that, the decision in each node would be the majority of what it has received.

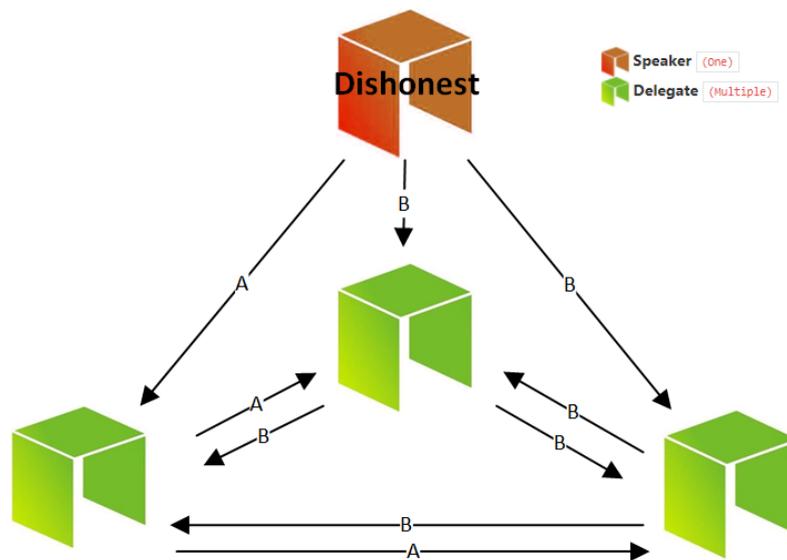


Figure 1-5: Situation when the Speaker is dishonest in this case (Source: steemit.com)

On the other hand, if the permissioned blockchain is running on a private or trusted environment, then there is no need for verifying the transaction or blocks multiple times across different nodes. Moreover, each node does not need to keep the whole replication of blockchain data. Although this kind of system is a bit more centralized than other blockchain systems, its transaction handling process can be more distributed and thus the overall throughput can be largely increased.

As the hash/nonce or the identity cannot be gotten easily, these consensus protocols make the modification of block data nearly impossible, unless more than half of the computers in the network become malicious, or there exists a polynomial time algorithm to crack the cryptography.

### **1.3. Objective**

The goal of the entire project is to design and build an opensource e-voting application that can be run by anyone to set up elections of any scale for others to cast their ballot and perform all kinds of verifications. The application should use blockchain and other technologies to enforce all consideration of e-voting, including end-to-end verifiable, voter authentication and anonymity. We hope, via the release of the application, the public can learn that e-voting can be more transparent and reliable than traditional paper-based voting.

The entire project is completed in 2 terms. For the previous term, we focused on research, design and brief implementation. Following were the tasks we have done for the first term:

- Researched on the implementation of end-to-end verifiable voting and blockchain voting technology that proposed by others
- Designed the cryptography algorithm to use and the flow of an election, from creating election to releasing result. Also, decided the type of blockchain and related protocol to be used for the e-voting application.
- Implemented the backbone of the application such as blockchain protocol and the basic part of the election, so that user can create, vote and tally an election.

For this term, we focused on full implementation and testing. Here are the objectives for this term:

- Apply zero-knowledge proof on key generation, ballot encryption and decryption, so as to further ensure the integrity of an election.
- Enforce full verification of all communication between servers, such as server signature on connection request, block and ballot, etc.

- Design a user-friendly interface so that the application is easy to deploy and use.
- Perform load tests on the system such that we can estimate the system performance and resource requirement needed for large scale election.

## 2. Related Work

### 2.1. E-voting in Hong Kong

While there is no large-scale e-voting used officially in Hong Kong, many tiny-scale e-voting is just done by setting up a simple website or using Google Form, which is obviously not fulfilling most of the e-voting consideration stated in Section 1.2.1.

The largest e-voting system in Hong Kong is PopVote which is for conducting civil referendums, such as the unofficial vote on issues related to constitutional reform proposals [7]. PopVote had been using website and mobile application to conduct the voting (Figure 2-1), but it changed to use Telegram Bot last year. No matter which medium is used, the design of voting is not end-to-end verifiable, voters can only trust the system.

The screenshot shows the registration interface of the PopVote mobile application. At the top, the status bar displays 'SMC HK', signal strength, Wi-Fi, time '20:05', and battery level '65%'. Below the status bar, there is a navigation bar with 'Back' and 'PopVote'. A progress indicator shows four steps: 'Please read', 'Input info', 'Validate identity', and 'Please vote', with 'Input info' being the active step. The main content area contains three input fields: 'HKID:' with a masked input field (one dot and a parenthesis), 'Cell phone:' with a masked input field (seven dots), and 'HK perm. resident aged 18 + ?' with 'Yes' and 'No' radio buttons. A 'Submit' button is located at the bottom of the screen.

Figure 2-1: Registration in PopVote mobile application (Source: [popvote.hk](http://popvote.hk))

Although it involves authenticating voter via HKID with the phone number and claims that the personal data won't be sent to the server, it can suffer from Denial of Service attack [8] and contains security loophole on disclosing personal data [9].

### 2.2. End-to-end verifiable voting

## 2.2.1. Helios

Helios is an opensource voting system with an online implementation (Figure 2-2) which invented by Adida [10]. It uses homomorphic encryption system for secure encrypt and decrypt the ballots, and uses a ballot bulletin board to enforce end-to-end verifiability.

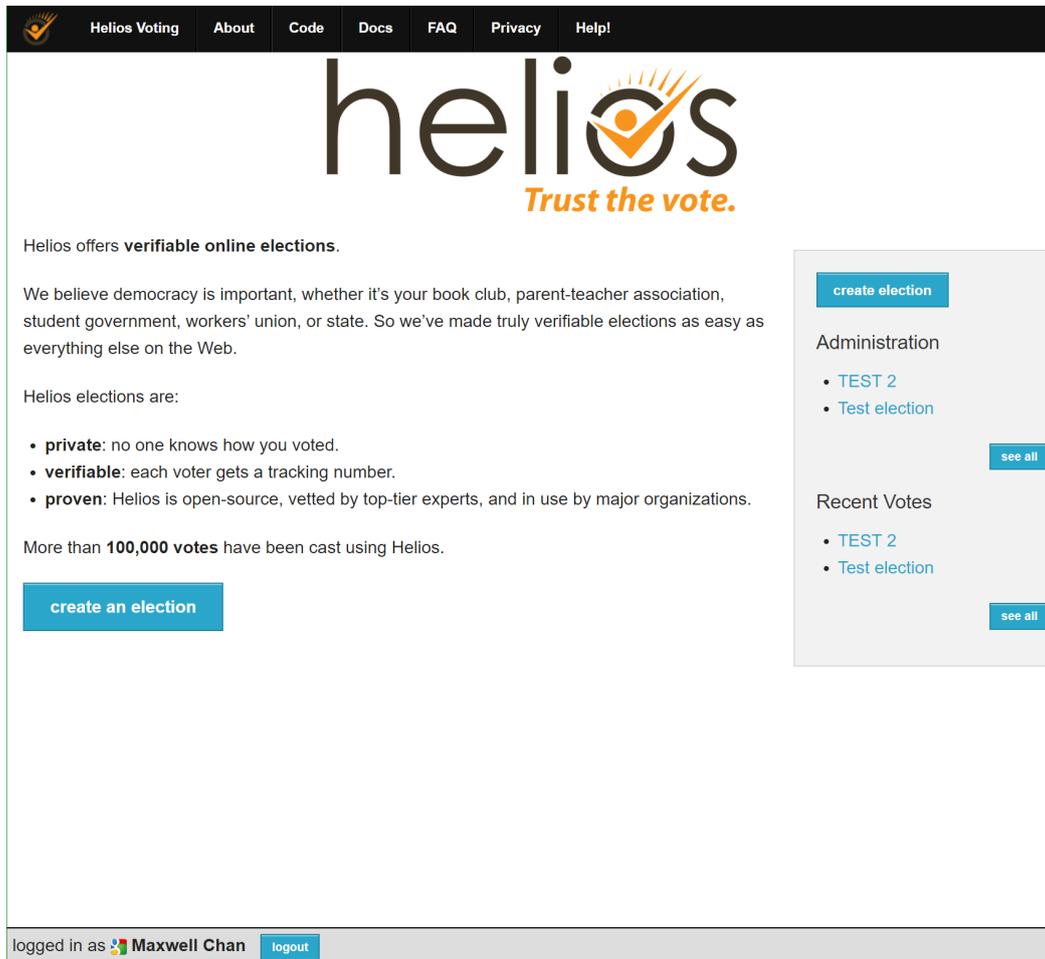


Figure 2-2: The online implementation of Helios

To create an election, besides setting the question and election details, a group of trustees is needed to be selected, so that each of them holds a different public and private key pair. All the public keys will be aggregated to become the election public key.

Voting in Helios is easy as clicking some radio button. The browser will encrypt the answer with some randomness and election public key, then a ballot fingerprint, i.e. the hash of the ballot, will be generated for the voter. A voter will be authenticated by Facebook or Google account, after that the ballot will be submitted with the voter ID only, which cannot recover the voter identity by others. On the other hand, the ballot bulletin board

(Figure 2-3), which accessible by the public, will post all submitted ballot fingerprint, so that voter can verify that his vote is counted in the tally correctly.

**Presidential Election for University — Bulletin Board** [\[back to election\]](#)

search:

**7 cast votes**

Voters 1 - 7 (of 7)

Name	Verification-Codes
Han Solo	25X5ChRhksV23kLq <a href="#">[view]</a>
Nicholai Hel	XCh9MnfN1Do83cBs <a href="#">[view]</a>
Ford Prefect	XC3YdNaLAsow439F <a href="#">[view]</a>
Zaphod Beeblebrox	qoJIfvEnoErFG45Q <a href="#">[view]</a>
Tricia McMillian	Cw7oPSr7dhFTE34W <a href="#">[view]</a>
Homer Simpson	AmJL1zHI198FGwsX <a href="#">[view]</a>

not logged in. [\[log in\]](#)  
[About Helios](#) | [Help!](#)

*Figure 2-3: Ballot bulletin board in Helios (Source: www.usenix.org)*

To compute the result, all ballots are aggregated homomorphically. Then, all trustees are required to provide his private key one by one to totally decrypt the aggregation. This process can be monitored by the public, as everyone can verify the aggregation. More importantly, no single ballot is decrypted in the whole process, unless all trustees are malicious. Therefore, even in the case of failure of the authentication system and voter identity is disclosed, his vote intention won't be leaked out.

### 2.2.2. Pretty Good Democracy

Pretty Good Democracy was proposed by Ryan and Teague [11]. This voting system uses special 'code voting' method to provide end-to-end verifiability, as well as proving voter and server identity

Prior to election begins, each eligible voter will receive a unique 'code sheet' (Figure 2-4), which provide a 'vote code' and a 'acknowledge code' for each option. All the codes are randomly assigned, so to act as a shared secret between all trustees and each voter.

Candidate	Vote Code	Acknowledgment Code
ALCHEMIST	5962	218931
ANARCHIST	2168	854269
BUDDHIST	3756	129853
MARXIST	1247	875391
NIHILIST	9881	039852

ID: 4896327

Figure 2-4: Code sheet in Preetly Good Democracy (Source: arxiv.org)

In order to vote, a voter needs to input the ‘vote code’ for the option he chooses into the system. Then, the system can the authenticate user via the code and it should respond the corresponding ‘acknowledge code’. The voter can compare it with the one on the ‘code sheet’, so as to ensure the server is not malicious. Notice that this mechanism can also prevent someone sniffing in the middle, as the sniffer can deduce nothing meaningful with only these codes.

Also, ballot bulletin board is used to show the ID of submitted ballots for voters to verify. To calculate the result, a mixnet is used to further anonymize the ballot, tallying can be done only after that.

### 2.2.3. In-person voting systems

There are more end-to-end verifiable voting systems that require perform voting in a voting booth. Usually in these systems, after voter marked his choice on the physical ballot, the ballot will be processed by a computer, while the voter will get a receipt for verification at a later stage. These systems include Prêt à Voter [12], Scantegrity [11], Punchscan [13], Voteegrity [11], etc.

Take Scantegrity as an example. In its physical ballot, a random code is assigned to each option, these codes are different for each ballot and each option. Moreover, the code is printed with invisible ink, which only becomes visible when voters mark the choice using a special pen (Figure 2-5). Therefore, voters can get the code together with the ballot ID as a receipt, which they can use to verify online if the submitted ballot is really in the database.

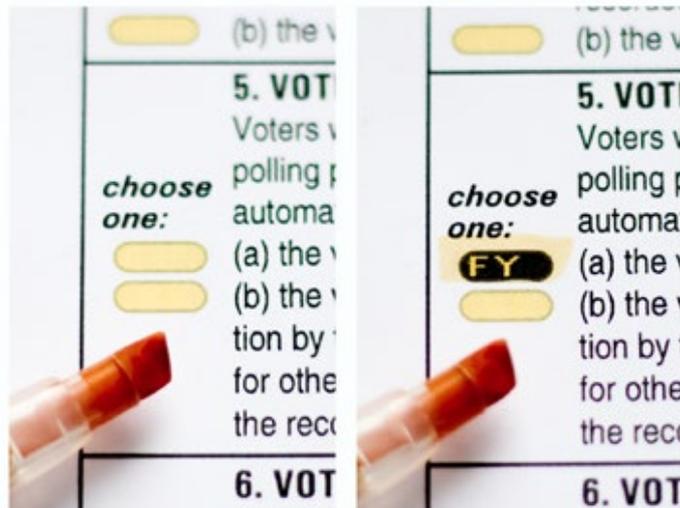


Figure 2-5: Using special pen for voting (Source: en.wikipedia.org)

## 2.3. E-voting using blockchain

### 2.3.1. A vote as a coin

There are proposals [14] [15] and implementation [16] of treating a vote as a coin in cryptocurrency, such that submitting ballot will be like making a transaction (Figure 2-6).

Basically, the idea is that every voter and candidate have their own wallet. Initially, the number of coin in the voter's wallet will be equal to the number of choices they can vote. In order to vote, a voter needs to transfer a coin into the candidate wallet. It is not allowed to transfer coin into another voter's wallet or transfer more than one coin to the same candidate from the same voter. In this design, tallying is easy, as the number of coin in a candidate wallet directly reflect the number of votes he receives.

Because of the transparency characteristics of blockchain, everyone can easily monitor and verify all activities in the blockchain network, so it can achieve more than end-to-end verification. Just like cryptocurrency, every wallet is identified by an address, only one with the corresponding private key can spend coin in the wallet. Therefore, voter authentication and anonymity also achieved.

However, the problem with this design is the disclosure of intermediate result. In the blockchain, everyone knows how many coins voters and candidates have. So, one can easily

get a real-time result of the election. This is not appropriate because it can affect the vote intention of those voters that not yet voted, making the election unfair. Another problem to this blockchain is that voter can prove to others which candidate he votes for, just by telling others his wallet address. This can make vote-buying easily available, causing corruption in the election.

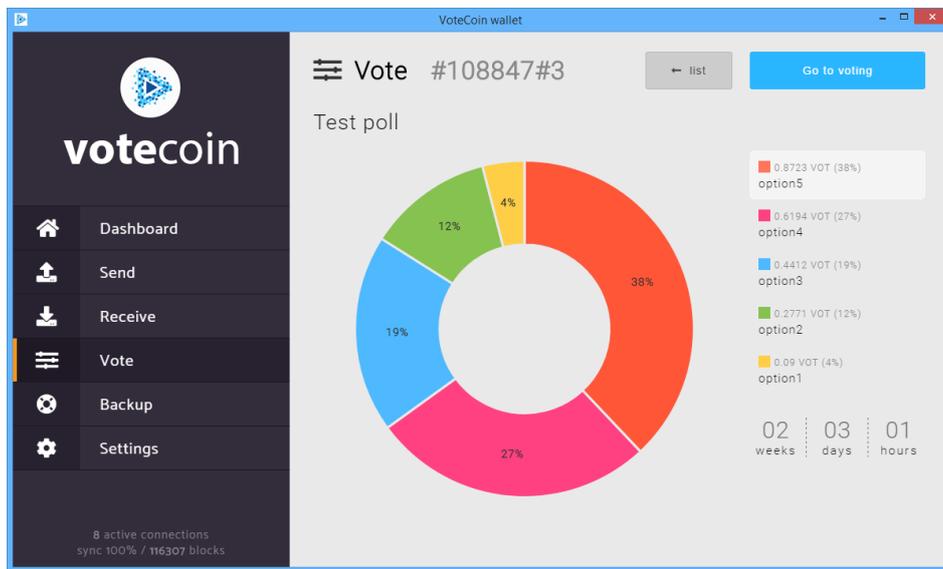


Figure 2-6: VoteCoin application (Source: [www.votecoin.site](http://www.votecoin.site))

### 2.3.2. Use blockchain for secure storage

Regardless of the encryption and decryption algorithm used for an election, other proposals [17] [18] [19] are trying to use blockchain mainly for a non-modifiable storage to support their election algorithm.

Use the research by Panja [20] as an example, it uses two blockchains. One blockchain will be used to store voter authentication details, this will be a private blockchain that only accessible by trusted parties. The hash value of the last block will be shared among them to verify the blockchain during the election. Another blockchain is a public one for storing ballot, it uses proof-of-work for consensus algorithm. The first block in this blockchain will store the election public key and signed by the administrator, for public verification purpose.

## 3. Design

### 3.1. Overview

After studying different end-to-end verifiable voting systems, we decided to use Helios' algorithm as the reference of our design. The main reason is that Helios is opensource and has a workable online implementation, which can demonstrate its merit and loophole easily, and also able to make obvious comparison with our final product. Besides, it uses homomorphic encryption to ensure no decryption of a single ballot, which is crucial to anonymity. More importantly, Helios has made use of zero-knowledge proof on several actions that involve applying the private key, which promote further confidence to the system on top of anonymity. While Helios has its limitations, we will discuss in this chapter and propose possible modifications.

For the blockchain part, we will use it as a storage for election details and encrypted ballots in Helios. More importantly, we allow everyone to run as a node and join the network for verification and monitoring the entire election. In this chapter, we will discuss in details the protocol design of the blockchain that we use.

### 3.2. Algorithm related to the voting process

#### 3.2.1. Helios cryptography

Helios uses homomorphic El Gamal encryption scheme, which is a variant of El Gamal encryption [21].

##### Create election

To set up a new election, one needs to generate the following parameters, they will be used consistently throughout the election:

$p$ : a prime number

$g$ : a primitive root<sup>1</sup> of  $p$

For every trustee  $i$ , he need to generate:

$x_i$ : trustee private key, where  $0 < x_i < p - 1$

$y_i$ : trustee public key, where  $y_i = g^{x_i} \bmod p$

All trustees' public key will be submitted to calculate the election public key  $y$ :

$y = y_1 y_2 \dots y_n \bmod p$ , where  $n$  is the number of trustee

### **Ballot preparation**

For each option in each question, the following are needed to be calculated:

$r$ : a random number, where  $0 < r < p - 1$

$$m = g^i \bmod p,$$

where  $i = 1$  if voter choose this option,  $i = 0$  if otherwise

$$c_1 = g^r \bmod p$$

$$c_2 = y^r m \bmod p$$

Then, the encrypt answer  $(c_1, c_2)$  will be submitted for each option.

The random number is important, as we do not want the voters that vote to the same option get the same encrypt answer, thus violating voter privacy. Therefore, the random number will not be shown to voter for security reason. However, if a voter is concerned about the trustworthiness of their browser, Helios allows a voter to view it for cast-as-intended verification, but then the answer will be re-encrypted using another set of random number.

### **Tallying election**

---

<sup>1</sup> If  $g$  is a primitive root of  $n$ , then for every integer  $a$  coprime to  $n$ , there is an integer  $k$  such that  $g^k = a \bmod n$ ;  $a$  is coprime to  $b$  if the greatest common divisor of them is 1

The following result computation process is repeated for each option in each question. As the trustees' private key will not be sent to the server, so firstly the server need to compute:

$$c_1 = c_{1,1}c_{1,2} \dots c_{1,a} \text{ mod } p$$

$$c_2 = c_{2,1}c_{2,2} \dots c_{2,a} \text{ mod } p$$

where  $a$  is the total number of voter,

$(c_{1,1}, c_{2,1})$  is the encrypt answer for voter #1

While on the client side:

$$g^m = c_2(c_1^{x_1} c_1^{x_2} \dots c_1^{x_n})^{-1} \text{ mod } p$$

where  $n$  is the number of trustee<sup>2</sup>,

$x_1$  is the private key for trustee #1

$m$  on the above equation is the actual number of voter that vote for this option. In order to get  $m$  from  $g^m(\text{mod } p)$ , we need to perform a discrete logarithm of  $g^m$  with base  $g$ . While there is no polytime algorithm to do discrete logarithm,  $m$  must be smaller than or equal to the total number of voters. So, what Helios use is just trying all possibilities, this will not be very time consuming when the total number of voters is not too huge.

The focus point of the entire encryption and decryption flow is on the  $g^0$  and  $g^1$  encoding in  $c_2$  when encrypting voter choice. As  $g^a g^b = g^{a+b}$ , when we do the aggregation of  $c_2$  for all voters, it will be able to produce  $g^m$ , so that we don't need to decrypt individual ballot one by one.

### 3.2.2. Helios zero-knowledge proof

Helios uses sigma protocol for all zero-knowledge proofs [21], which works with 3 steps as follow:

- i. Prover, who usually is the client, send a commitment message  $a$  to the Verifier,

---

<sup>2</sup> The <sup>-1</sup> is the equation means modular multiplicative inverse, so that  $a^{-1} = x$  when  $ax \text{ mod } p = 1$

who usually is the server

- ii. Verifier sends a challenge message  $e$  to the Prover
- iii. Prover sends the response  $f$  to the Verifier

It is important that the challenge should not be guessable by the Prover, so as to prevent the Prover pre-calculating the commitment message for a specific response message, which will violate the proof.

However, it is not a good idea to use an interactive proof for two reasons. First, this only allows a single Verifier to prove the validity, which is not ideal because in the system we want the details of an election to be able to prove to everyone. Second, if the server becomes malicious and controlled by the Prover, then it is easy to generate a “challenge” message that is guessable by the Prover.

Therefore, Helios use a non-interactive mode of the sigma protocol. This is done by generating the challenge by the hash of some parameters such as the commitment message, so that it can be generated on the Prover side. Since the output of a hash function is usually not guessable, so it is very hard, if not impossible, to find a commitment message that can hash to a designed value. As a result, the proof will be submitted as  $\{a, f\}$  since  $e$  can be calculated from the hash function, and it will still be considered as legitimate proof.

### **Trustee knowledge on private key**

When a trustee submits his/her public key, for two reasons the others need to know that he/she actually has the corresponding private key. First, if there exists at least one trustee that does not know the private key, then the entire election cannot be decrypted. Second, if a trustee submits the public key as  $y_i/(y_1y_2 \dots y_n)$ , then the election public key  $y = (y_1y_2 \dots y_n) \times y_i/(y_1y_2 \dots y_n) = y_i$ , which make the trustee possible to decrypt the whole election by his/her private key  $x_i$  only.

A trustee needs to compute the following to generate the proof:

$$a = g^s \text{ mod } p, \text{ where } s \text{ is a random number such that } s < p$$

$$e = H(g, a, y_i) \text{ mod } p, \text{ where } H \text{ is a hash function}$$

$$f = s + ex_i \text{ mod } (p - 1)$$

Others can verify the proof by:

$$g^f \equiv ay_i^e \pmod{p}$$

The proof can be verified without knowing the trustee private key because:

$$g^f = g^{s+ex_i} = g^s g^{x_i e} = a(g^{x_i})^e = ay_i^e \pmod{p}$$

### **Trustee honest decryption**

Another similar zero-knowledge proof related to trustees is to prove that trustees honestly contribute to the decryption process. This is required because a dishonest trustee can do more operation on the partial tally than just applying his private key. For example, the trustee can multiply the tally with  $g^1$  and  $g^{-1}$  to “add vote” or “subtract vote” from the tally respectively. Therefore, including this proof will improve the integrity of the election.

The trustee needs to perform the following calculation for the proof, notice that there are 2 commitment messages  $a_1$  and  $a_2$  for proving trustee knowledge on private key  $x_i$  and partial decrypted tally  $d_i = c_1^{x_i}$  respectively:

$$a_1 = g^s \pmod{p}$$

$$a_2 = c_1^s \pmod{p}$$

$$e = H(g, a_1, a_2, y_i, c_1, d_i) \pmod{p}$$

$$f = s + ex_i \pmod{p-1}$$

Others can verify the proof by:

$$g^f \equiv a_1 y_i^e \pmod{p}$$

$$c_1^f \equiv a_2 d_i^e \pmod{p}$$

The proof can be verified without having the trustee private key to perform decryption because:

$$c_1^f = c_1^{s+ex_i} = c_1^s c_1^{x_i e} = a_2 (c_1^{x_i})^e = a_2 d_i^e \pmod{p}$$

### Voter honest encryption

As stated in Section 3.2.1, voter choice on an option is encoded with 0 or 1, then perform the encryption before submitting the ballot. However, it is possible that voter encodes something else like a negative number. Since no one can validate this as the ballot has been encrypted and no individual ballot should be decrypted for checking, so we need a zero-knowledge proof to validate voter honesty.

Furthermore, every question has a limit on the minimum and the maximum number of choices. The only way for a server to verify this is by using zero-knowledge proof again, checking that the sum of all 0/1 choice is within the limited range of the question.

The above two proofs are both verifying a specific value is within a range, which is different from previous proofs that verifying a specific value exist. So, Helios solve this by generating a proof for each value in the range. For the value that actually reflects the voter intention, a “real proof” will be generated. While for all other values in the range, a “simulated proof” will be generated instead. However, other people cannot differentiate between a real and a simulated proof.

A simulated proof is generated as follow:

$e$ : a random number, where  $0 < e < p - 1$

$f$ : a random number, where  $0 < f < p - 1$

$$a_1 = g^f (c_1^e)^{-1} \text{ mod } p$$

$$a_2 = y^f (c_2 (g^i)^{-1})^{-1} \text{ mod } p, \text{ where } i \text{ is the value to be simulated}$$

While real proof should be generated like this:

$$a_1 = g^s \text{ mod } p$$

$$a_2 = y^s \text{ mod } p$$

$$e = H(\{a_1, a_2\} \text{ for all proof, } c_1, c_2) - (\sum e, \text{ for all simulated proofs}) \text{ mod } p$$

$$f = s + er \text{ mod } (p - 1)$$

Notice that, in the case of generating a proof for a question choice,  $r$  is the same random number as stated in Section 3.2.1 - Ballot preparation. While for the proof of the

number of choices in a question,  $r$  is the sum of  $r_i$  for all choice  $i \pmod{p-1}$ . This is also important for a decryptable election because it indicates the voter knowledge of  $r$ , so that he/she is not submitting a random number as  $c_1$  and  $c_2$ .

No matter the proof is real or simulated, it is verified by:

$$g^f \equiv a_1 c_1^e \pmod{p}$$

$$y^f \equiv a_2 (c_2 (g^i)^{-1})^e \pmod{p}$$

The following verification also required for every set of range proof:

$$\sum e \equiv H(\{a_1, a_2\} \text{ for all proof, } c_1, c_2) \pmod{p}$$

This is crucial to verify that there exists one and only one real proof in the set. It's because, for the real proof,  $e$  must be calculated after  $a_1$  and  $a_2$ , which cannot be simulated.

### 3.2.3. Limitation on Helios and proposed modification

#### Denial of service attack

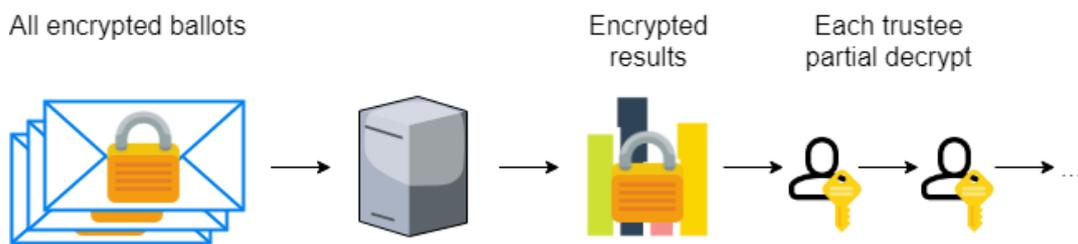
Currently, Helios implementation is not designed to be run on distributed servers and databases. So, there is a risk that the server may under attack and become unresponsive or even making changes to the database. Thus, Helios also claim that it is not suitable for large-scale election.

That's why we would like to implement Helios on a blockchain, so that the signal point of failure is removed from the system [22]. Everyone can make a copy of the database so that in case of all servers fail, the ballots won't be lost or modified. Besides, if any server is under attack, one can easily set up another server by just joining the blockchain network with a computer. Therefore, overall availability and integrity will be improved.

Moreover, using blockchain make tracing error easier. Previously in Helios, if a voter suspects that there is inconsistency in the ballot bulletin board, it is hard for a voter to tell where the error is come from, as he does not know what is going on in the server. With distributed server and database, one can have more information to check the record of all data.

### Slow tallying

As described in Section 3.2.1, computing result in Helios needs to perform discrete logarithm. While the speed is acceptable when the number of voters is small to medium, it can be slow in a large-scale election, since this action needed to be done for every option in every question. Another bottleneck will be in aggregating the encrypted answer on the server side, as it will involve lots of multiplication. (Figure 3-1)



*Figure 3-1: The tallying process in Helios*

So, we suggested allowing decrypting ballots in batch. A naïve approach to achieve this is, when tallying, divide all ballots into several batches (Figure 3-2). As we have distributed servers, we can assign different servers to be responsible for different batches, such that the decryption of all ballots can really be done in parallel, thus reducing the overall time of tallying. After that, each trustee will receive the partial decryption batch by batch, he/she will apply their private key on it and pass it to other trustees.

However, this approach has two problems. First, we cannot simply group the ballots in batch according to the arrival order. It's because we allow re-voting and only the last ballot submitted by the voter counts. Therefore, we need to group the ballots in a way that all ballots submitted by the same voter will be in the same batch, so that the server responsible for that batch will examine the timestamp of the ballots and process only valid ones. Second, when there are lots of batches, then each trustee will receive a lot of decryption request at a different time. More importantly, trustees may need to wait a long time for each batch to pass through each trustee, which is not convenient for them. As applying trustee private key should not be a bottleneck in the process, we should group all batches and send to the trustee only once for an election (Figure 3-3).

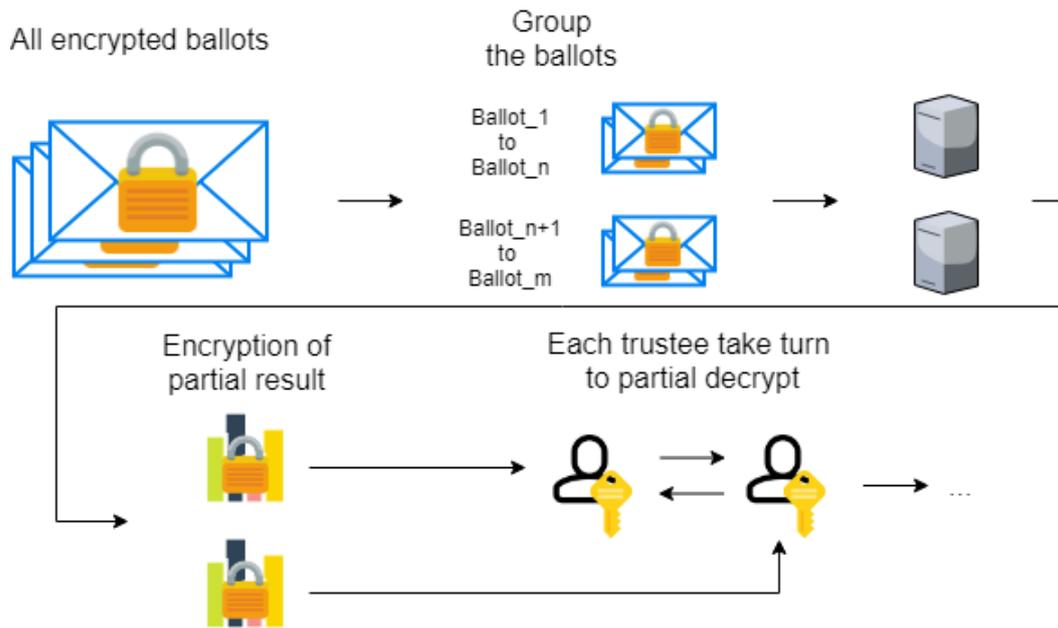


Figure 3-2: Our first proposal for the tallying process

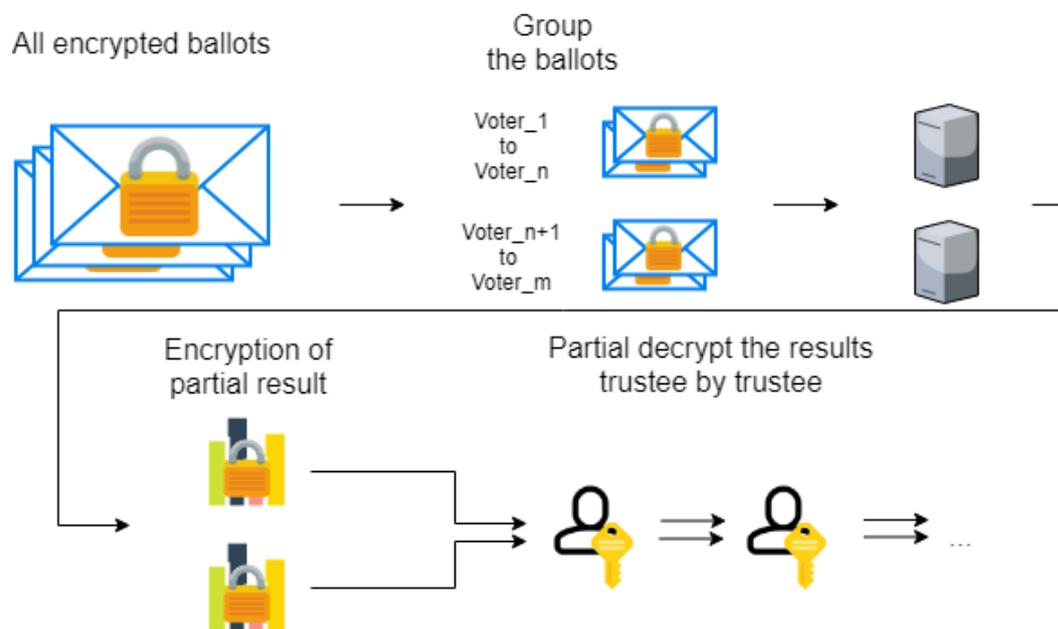


Figure 3-3: Our final proposal for the tallying process

Decrypting ballot in batches will not violate ballot anonymity as long as the group size is not too small to make guessing of individual ballot intention possible. Also, everyone in the blockchain network can verify the aggregation done by the servers, to make sure they won't give a single ballot to trustees for decryption.

## **Coercion**

Coercion in voting means that a voter is forced to vote for some candidate. This can be done in several ways: voter prove to coercer that what he had voted, coercer sits next to voter to force him on voting, and voter gives his credentials to the coercer. The first way is not possible in Helios, as no one can deduce the selected options from the encrypted ballot, even with the help of the voter.

However, the second and the third way is still possible in Helios. This is actually a common problem in a remote voting system, the server never knows who is actually sit next to the voter. Helios try to reduce the possibilities of coercion by allowing re-voting [21]. A voter can vote as many times as he wants in an election, only the last submitted ballot will be tallied. So that if a voter is being coerced previously, he can vote again when the coercer has left. Nonetheless, the coercer can still notice the change in ballot fingerprint if he knows the ID the voter.

While we will keep this re-voting mechanism, we think it is not enough. So, we proposed to add an option for in-person kiosk voting when setting up an election. If election organizer uses this option, he needs to provide an extra RSA public key, while keeping private key secret, when creating election. Election organizer needs to set up some kiosks, just like voting booths in paper-based voting, for a voter to vote in-person. Inside the kiosk, voter just needs to perform voting on a computer as normal. The only difference is that this computer contains the private key, which will sign the ballot and then send to the blockchain network (Figure 3-4). Therefore, the voter cannot vote again on their device after they vote in the kiosk.

We thought that the risk of coercion increases with election scale. For example, coercion risk is low for student union election, while it is high for district council election. Also, in a large-scale election, even for e-voting, some kiosk will be needed to set up for voters not having electronic devices or having technical issues. So, what we have just proposed simply add a functionality to these kiosks, which allow voters who are really concern about the risk of coercion to vote in the kiosk.

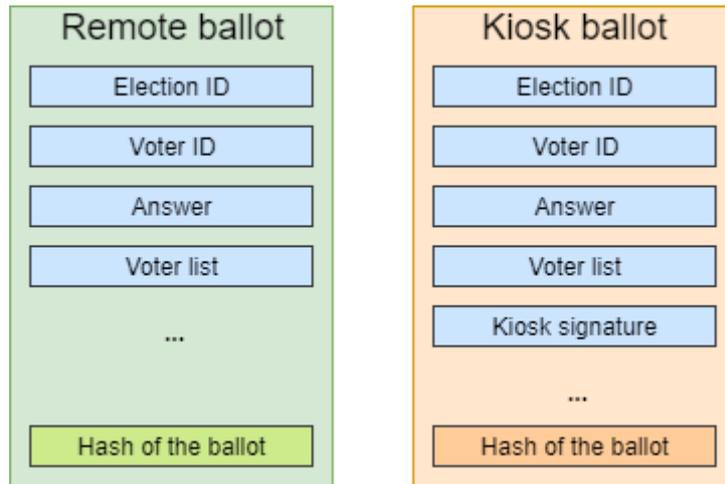


Figure 3-4: Difference between a ballot encrypted by kiosk to that by personal devices

### **Authentication method**

As stated in Section 2.2.1, Helios authenticate voter via Facebook or Google account. After authentication, the corresponding voter id and hash is put into the vote. The problem here is that the public can only trust the authentication by Helios, they cannot further verify.

Therefore, we suggested having ballot signature bound with the submitted ballot. Before starting an election, every voter needed to own an RSA private and public key pair. The public keys of all voters are stored in the blockchain together with their voter ID. So, when preparing the ballot, each voter is required to provide their private key to sign on the ballot before submitting. Such that everyone can verify the hash by using the corresponding public key.

Nevertheless, letting each voter get his/her key pair for the first time is like a chicken and egg problem. If the server does not contain any voter information beforehand, then it cannot authenticate the voters when they submit the public key. But if the server does contain voter credential or public key in advance, then others may query on the correctness and trustworthiness of the data. Anyway, we have thought of three different ways of enrolling voters into an election.

First, election administrator inputs a list of voter's emails. The server will generate a key pair for each voter, then send it to the voter via email but only save the public key on the blockchain database. While this seems heavily rely on server honestly to not save the

private key, but we think the server needs to be trusted in any case, because all the contents on the webpage come from the server. The concern with this method is the security when sending email, as the channel or the email client may not be secure, so the private key may leak out. While we can send out a unique link to each email for voters to generate their private key themselves so that the system looks more trustworthy, this requires additional security requirement on the database, and the security problem on email still not solved. In fact, if there exists a system that voters can verify and change their key pair via the system, then the security problem on email is not that important.

The second way is enabling voters' self-enrollment, so that people will register an account on the server and submit their key pair, then election administrator selects a subset of them to be the voters of the election. The problem here is that the server will be storing voter personal details or their credential, which may provide a clear link between voter ID and the identity of the account owner. Besides, it still requires a way to verify that the account is not registered by some malicious people. One of the solutions to this problem may be using an organization email address which cannot be obtained easily.

The third way is allowing election administrator to directly upload the voters' public key to the server, then voters can also verify that via the server. This is applicable if the key pair is generated in advance. For example, the government may give each citizen an electronic identity that consists a private key; or an organization distribute hardware private key (Figure 3-5) to the voters. The problem here is the voter needs to trust the election administrator that he/she did not make a copy of the private key.

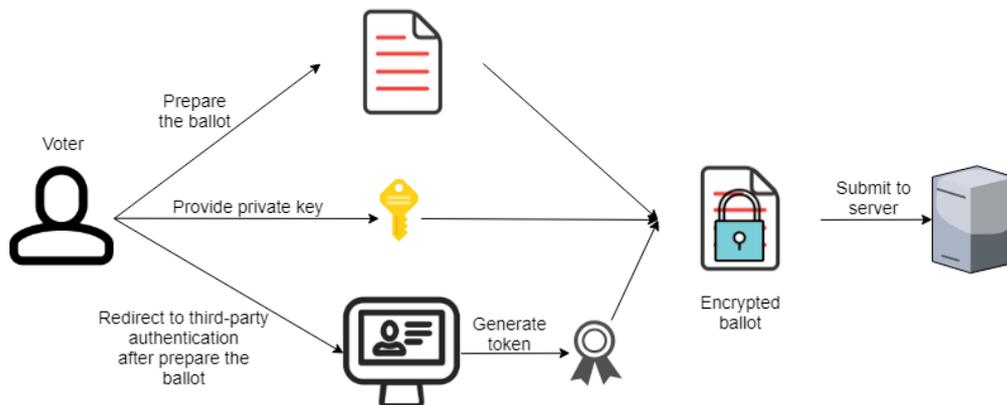


*Figure 3-5: Example of a hardware private key (Source: [www.yubico.com](http://www.yubico.com))*

Each of these three ways has its pros and cons. For different elections which may have different trust and security concerns, they may prefer a different solution. Hence, the election application may include all three of them or even more, in order to be more generalized.

On the other hand, during the voting stage, there is another security problem. The one using the private key may not be the one who owns the private key, then the coercion problem may come again. So, we have thought of one possible way to tackle this, which is encouraging the election administrator to use valuable credentials for further authentication besides the private key.

Valuable credentials mean that the credentials cannot be created easily, and the voter may use these credentials for other important services' authentication, so that they are less willing to give it to others. For example, if it is an election for students in CUHK, then CUHK OnePass credentials will be valuable, as students use it for all IT services in CUHK like email. Another example would be biometric information like fingerprint and facial data, as people nowadays may use it for unlocking the mobile phone and even online banking. A counterexample would be Facebook and Google account, as creating a new account is very easy.



*Figure 3-6: Overview of steps for using third-party authentication*

As the voting application should be more generic to support elections of any scale, it may not enforce the authentication method stated above. A possible solution is to offer an API<sup>3</sup> for the election organizer to provide their add-on authentication method. Before a voter submits the ballot, he will need to pass through the third-party authentication first. In case of successful authentication, it will sign on the voter ID and submitted with the ballot (Figure 3-6).

---

<sup>3</sup> Application Programming Interface

### **Knowledge of who has voted**

In order to achieve record-as-cast verification and tallied-as-recorded verification, a ballot bulletin board is used in Helios, showing all election fingerprints with the corresponding voter ID or voter alias<sup>4</sup>. While no one can deduce voters' intention, one can still know whether a voter has been voted or not, if he knows the owner of a voter ID.

We believe that this is not a significant problem, as in any traditional paper-based voting system, it is always possible to know whether a voter has been voted or not by monitoring the entrance of voting booth. This may become a problem only when the voting intention of a voter is obvious, and his voter ID is known by a coercer which has opposite voting intention, then the coercer may coerce him not to vote.

Thus, we will not make changes to the mechanism of Helios here. Instead, we will encourage the election organizer to implement three measures. Firstly, voter ID should not be guessable and do not reuse the voter ID from one election to another election, so it is less likely for one's voter ID and his voting intention to be known by others. Secondly, always add an 'Abstention' option in an election, so that a voter intention is more difficult to guess by whether he has voted or not. Thirdly, educating voter not to disclose their voter ID, as this may infringe their own privacy.

## **3.3. Blockchain related protocol**

### **3.3.1. Type of blockchain**

Since the blockchain will be used for storing ballots and all election details, we want it to be publicly verifiable. It cannot be a totally private blockchain because an election is likely to involve multiple parties, so that usually the application is not running on a fully trusted network and computer. Thus, we need to choose between permissioned and permission-less blockchain.

Permission-less blockchain works well in cryptocurrency and smart contract. However, when it comes to voting, the 51% attack [23] may be easily possible. Notice that

---

<sup>4</sup> Election organizer can assign alias to voters instead of generating voter ID

everyone in a permission-less blockchain have a right to verify a ballot and put it into a block. Although the longest chain will be chosen as the right one, if there is a fork in the blockchain of the same length, the one that majority nodes have will be chosen. Consider a situation that there are only two candidates in an election, one of them has more computational power and own more than half of the nodes. These nodes can just approve the ballots from ‘trusted voters’, who is the voters of their side, and disapprove all other ballots. As a result, the election will not be a fair one.

Another reason that permission-less blockchain is not quite suitable is that usually the consensus algorithm they use is proof-of-work, which is quite computationally intensive. As we want our blockchain to be suitable for an election of any scale and easy to use, it is not a good idea to use proof-of-work or purchasing this kind of service.

Therefore, we decided to use a permissioned blockchain. The only trust here is on the trustees, just like the Helios’ algorithm rely the trust on all trustees. As long as not more than half of the trustees are malicious, it is still a fair election.

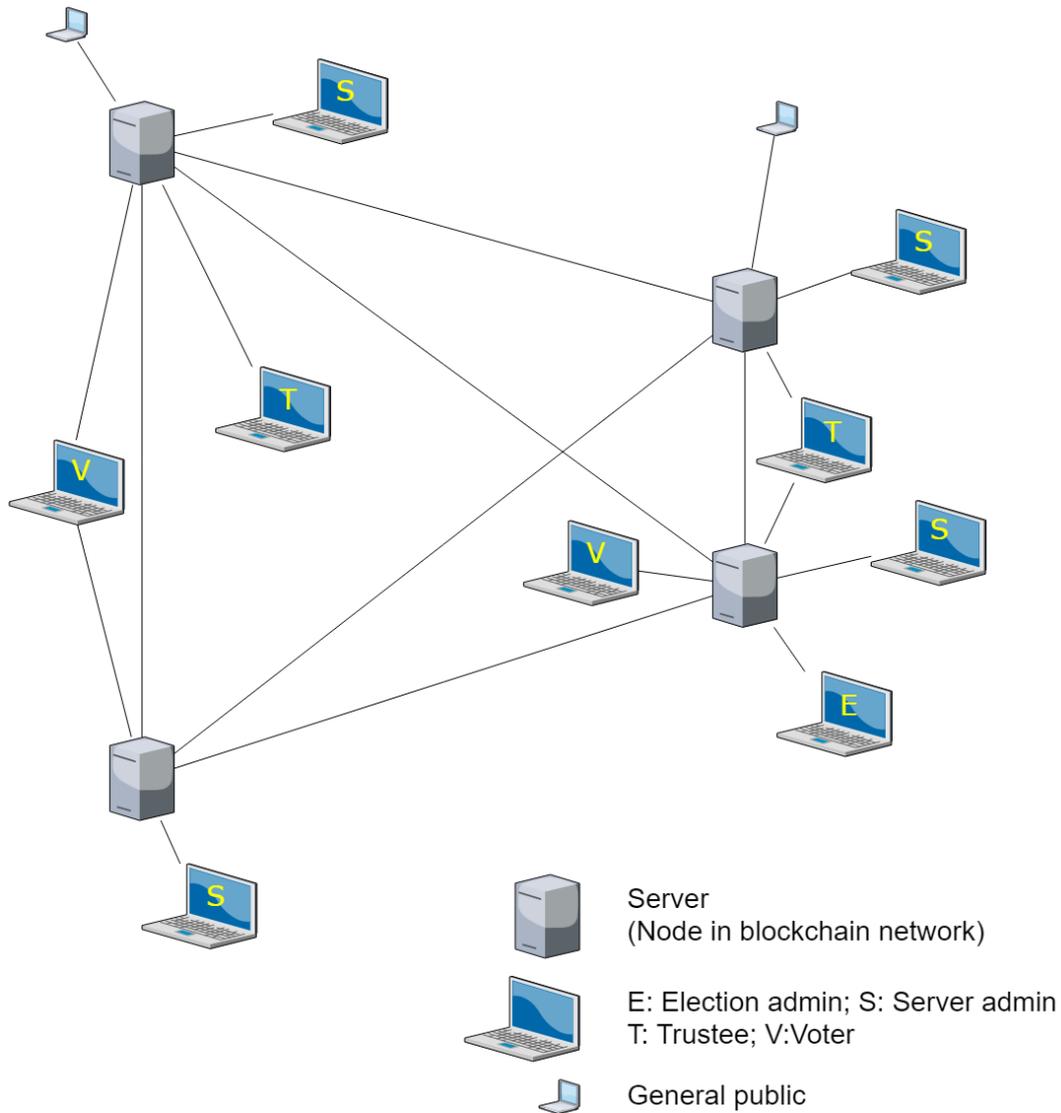
### **3.3.2. Roles and permission**

There are 5 different stakeholders here: election administrator, server administrator, trustee, voter, and the general public.

Election administrator is responsible for configuring an election, which includes assigning servers to participate in the election. He/she will have the right to write related information on the blockchain only before an election start and after it ends.

A server administrator is responsible for maintaining the server (node) and keep the connection with other servers. While he/she has the full right to change anything on the database that stores the blockchain data, this will be detected by others when performing verification.

Trustees will have the right on decryption via servers to write the related result on the blockchain, while voter can only read the blockchain and submit their vote via servers. The general public may have the right to read the blockchain for an election, depends on whether the election organizer wants the election to be known by everyone.



*Figure 3-7: An example of connections among different computers*

It is possible that trustees and server administrator can merge with each other, such that voters can be more confident with the trustworthiness of the servers. However, we separated them for two reasons. First, in reality, trustees may not be the one who has time and ability to maintain and configure a node. Second, there is a fundamental difference between these two roles. For servers, we only require more than half of them to be honest. For trustees, if any one of them is not cooperative, the election can't be decrypted successfully. Therefore, these two roles should be separated by using different key pairs.

Servers (nodes) should connect directly with each other, so that the consensus will make quicker. For trustees, voters or the general public, usually they should connect to one of these servers to read the blockchain and perform the action required, but they can also

connect to any other servers to do verifications. (Figure 3-7)

### 3.3.3. Design a blockchain protocol for voting

Unlike public blockchain like Bitcoin and Ethereum, which has many different implementation and library available online, there are not many opensource libraries for permissioned blockchain.

One of them is Hyperledger Fabric [24] (Figure 3-8), which is contributed by IBM. However, its first release is only one year ago, and it is still constantly developing. Although for our application we can use the Hyperledger Fabric with suitable modification, there may be some security loopholes that we didn't notice, as it contains more functionality than we actually need. Moreover, as we have tried, using Hyperledger Fabric is not that user-friendly. It requires some dependencies to be installed separately. Adding or removing nodes from the blockchain network will require additional configuration on every node. Since we aim at making the deployment of application easier and user-friendly, so we believe that Hyperledger Fabric is not the best choice for us.

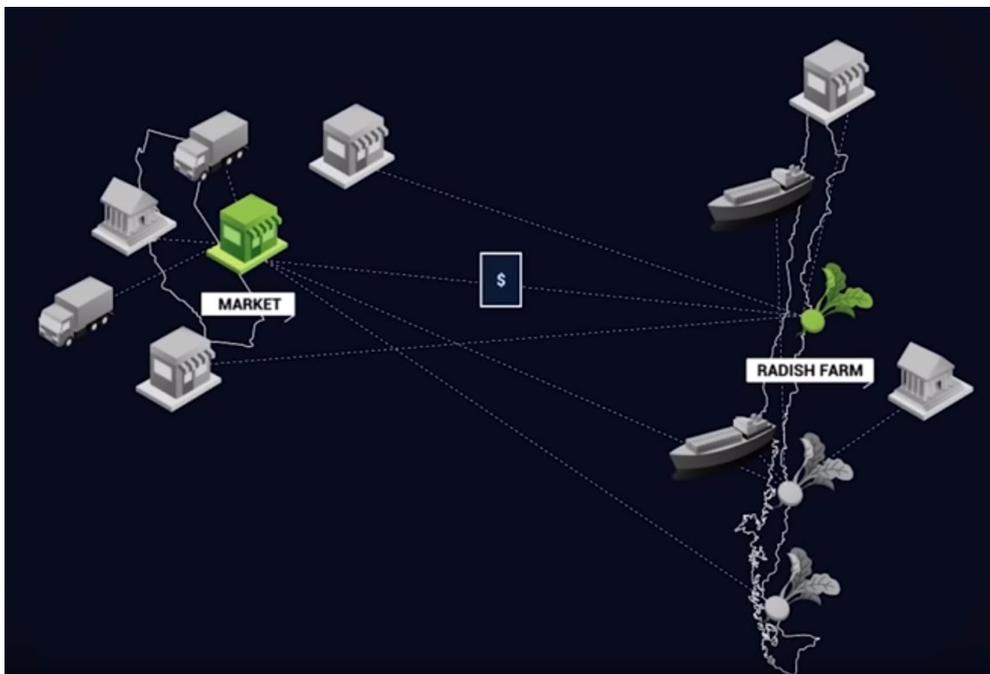


Figure 3-8 Screenshot from Hyperledger Fabric's introduction video

Hence, we would like to define our blockchain protocol for the voting application. In

fact, this will bring us more advantages. First, it will be more lightweight, as we will just design what we need. Second, it will be easier to run the functions related to voting, such as vote aggregation, since we can simply define the block structure as we want.

Although making new protocol may impose new vulnerabilities, we will try our best to search for these and take relative measures. More importantly, our application will be opensource and will not rely on security by obscurity, such that others can also prove the security.

### **3.3.4. Block design**

Every election will have its own blockchain since the trustees involved may be different and some elections may not be publicly available.

The first few blocks will contain all details required for an election, such as election ID, questions, election public key, list of voter ID, voters' public key, etc. Election organizer can make changes to these details by including new data into a new block, so that the latest piece of information will be used.

Until a block contains a key-value pair with the key of "Frozen At" and value of current time, election organizer may no longer make changes to the election details. The hash value of this block will become the hash value of the election details, for later verification purpose. After that, every block will contain a set of ballots that are submitted and verified before that block is generated. A block with type "Ballots" will be generated in a regular time interval, with a value preset in a configuration file. (Figure 3-9)

After the election has ended, blocks added to the blockchain will have the type "Election Tally". First, a block contains "End At" with the timestamp of election ends will be added. Second, when election organizer decided to start the tally process, another block contains "Tally At" with the timestamp of tallying starts will be generated.

Then, each server will perform the ballot aggregation for that assigned batch of ballots. Each server will write the result of the partially aggregated tally to the blockchain when it completes. After that, each trustee will perform the partial decryption, and record the result into the blockchain via servers. Finally, the election result will be also logged into the last block of the blockchain for this election.

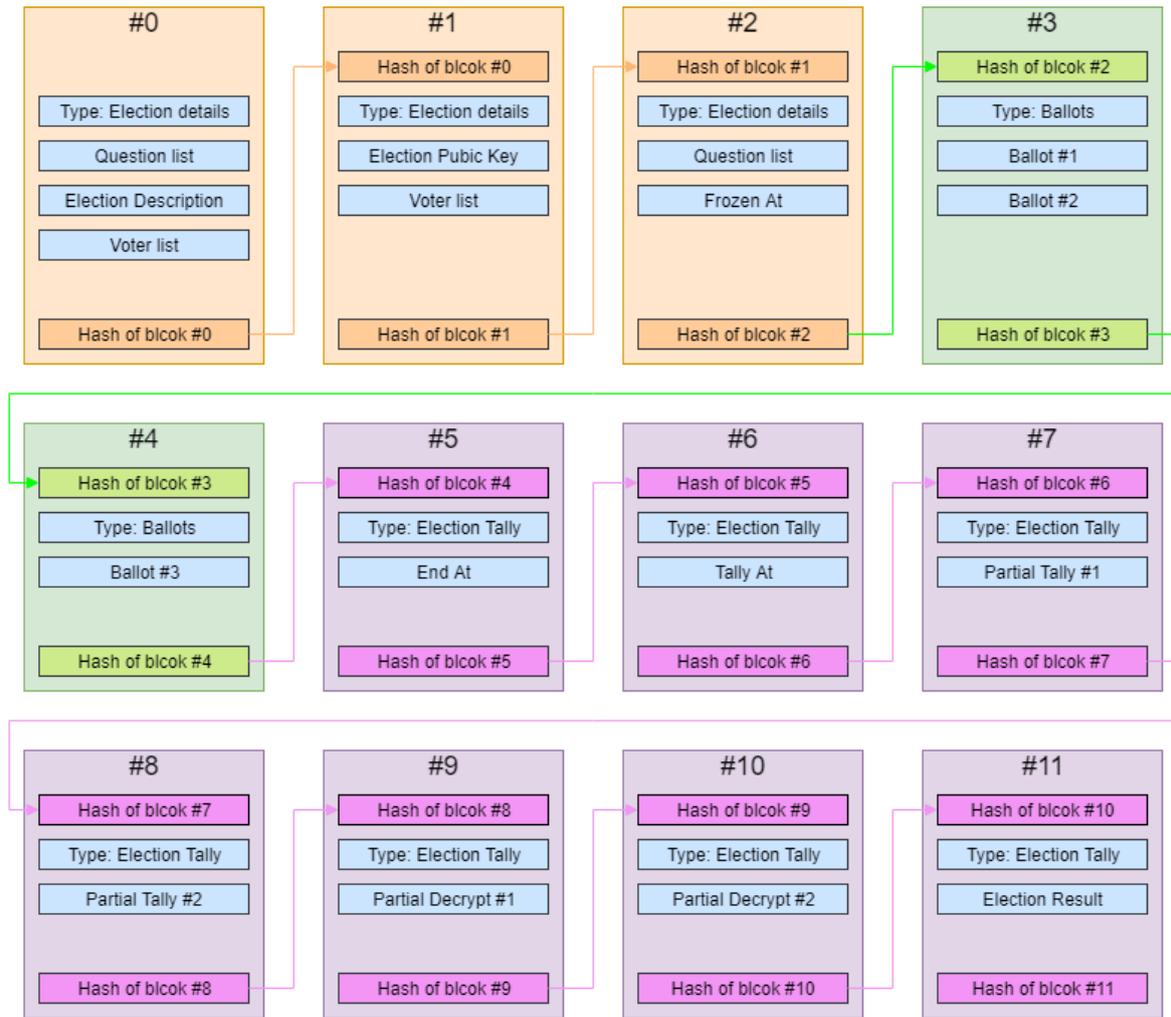


Figure 3-9: An example of blockchain in an election

### 3.3.5. Handshake protocol design

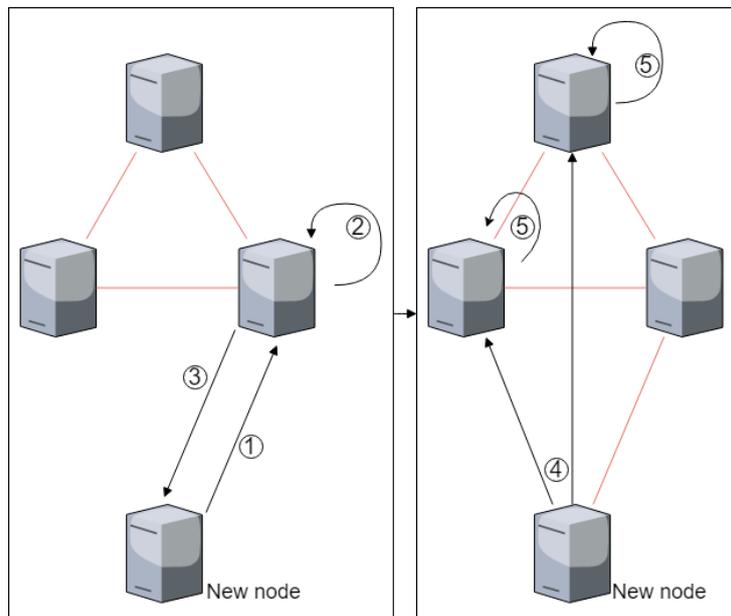
Here is the procedure of how a new node X join an existing network (Figure 3-10):

- iv. X get the address of any node Y in the network and send a connect request, which contains X's address, public key and instance ID.
- v. Y verify the request from X and save the details of X into the database. Y will also check if the same instance ID appear in existing records, if yes then Y will overwrite that record.
- vi. Y send back addresses of all nodes in the network to X

- vii. X save all the address into its database and ping all other nodes
- viii. Other nodes save the address of X when receiving its ping request

Besides handshaking, X should also make a separate request to query Y for the latest blockchain status, in order to keep X synchronize. If X found there are missing blocks in its database, X should get a copy of them as soon as possible.

All nodes in the network will ping each other periodically to check their liveness, a node will be deleted from the database if it does not response from ping.



*Figure 3-10: Steps of the handshake protocol*

### 3.3.6. Ballot submission protocol design

Here is the procedure when a node X in the blockchain network receive a ballot from a voter (Figure 3-11):

- i. X verify the ballot using the voter public key and save it into the local database
- ii. X broadcast the ballot in the network
- iii. All other nodes will verify the ballot using the voter public key and save it into its own database
- iv. If the ballot is valid, all nodes include X will sign on the ballot hash and

broadcast the signature in the network

A ballot is considered verified when more than half of the servers have signed on it.

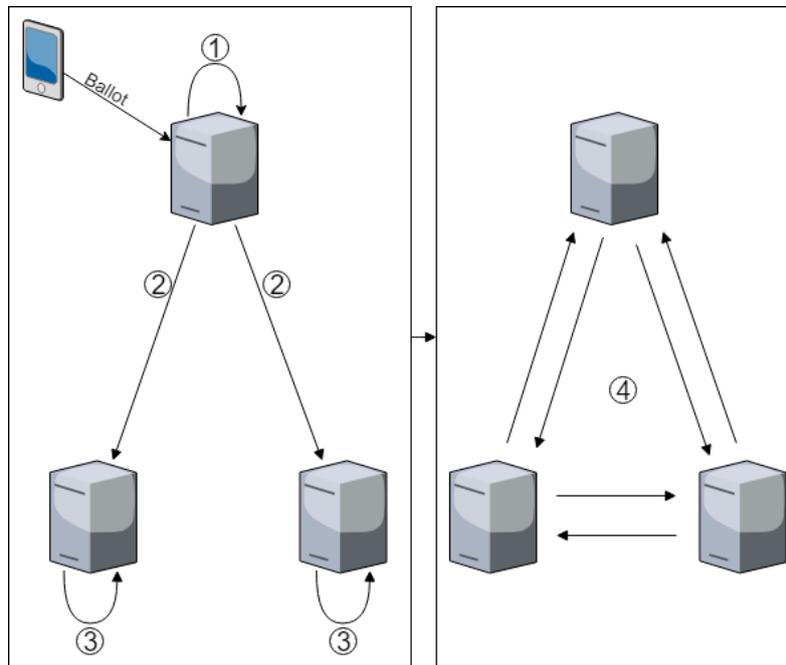


Figure 3-11: Steps of the ballot submission protocol

### 3.3.7. Block generation process design

#### Node selection protocol

As a block is generated in a regular time interval, all nodes need to select a node to be the generator every time. While always generating block from the same node will work, assigning the job to different nodes every time will make the blockchain more trustable, as no single trustee is controlling the blockchain. The selection process is as follow (Figure 3-12):

- i. Pick all nodes that are participating in this election, no matter they are online or not, sort them by its server ID, which is a function of its server public key
- ii. Get the hash  $h$  from the last verified ballot in this time interval
- iii. Let  $n$  be the total number of nodes, and  $i = h \bmod n$
- iv. The node on the  $(i + 1)$  row of the sorted address table will be selected

A ballot may get verified just at the time of node selection, causing inconsistency in different nodes. Therefore, we suggested that there will be a little time buffer when defining 'last verified ballot'.

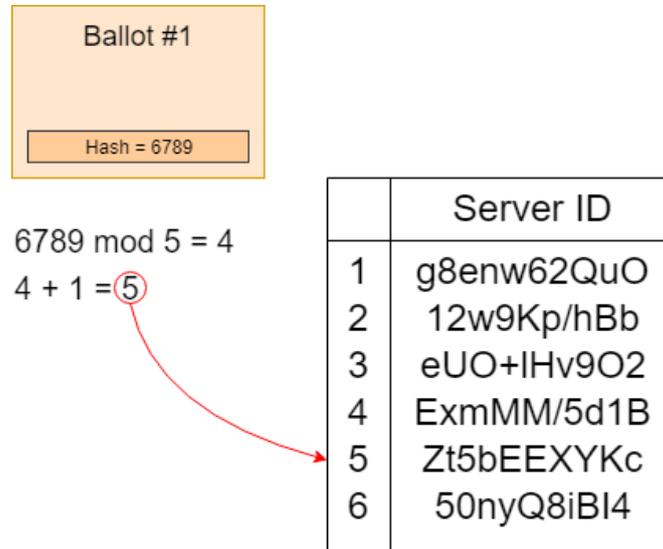


Figure 3-12: An example of selecting a node by the hash of the last verified ballot

We have thought about using only online nodes and their address for sorting in step 1, but it turns out that is not a good idea for two reasons. First, the signal of a node going offline may not be spread to all other nodes at the same time, so different nodes may end up with a different selection. Second, when there are half or less than half nodes go online, then it is meaningless to generate a new block because the new block will never get signed by more than half of the nodes.

It is possible that the selected node is offline, when it happens, just skip this round and wait for the next round of block generation.

### **Consensuses protocol**

Usually, the selection from all nodes will be the same unless there is serious network delay when broadcasting ballots, or there exist some malicious nodes in the network. Therefore, it is possible to results in different nodes being selected, so that a consensuses protocol is needed to select the majority of them to be the one who generates the new block.

We use the Byzantine Fault Tolerance algorithm for the consensus, which should be work as follows (Figure 3-13):

- i. Every node broadcast its result of the node selection protocol, the message will be signed by the node's private key.
- ii. When receiving node selection messages from other nodes, verify them by the corresponding node's public key.
- iii. Determine the majority of them by every node itself. In case of a tie, the smallest one will be chosen.

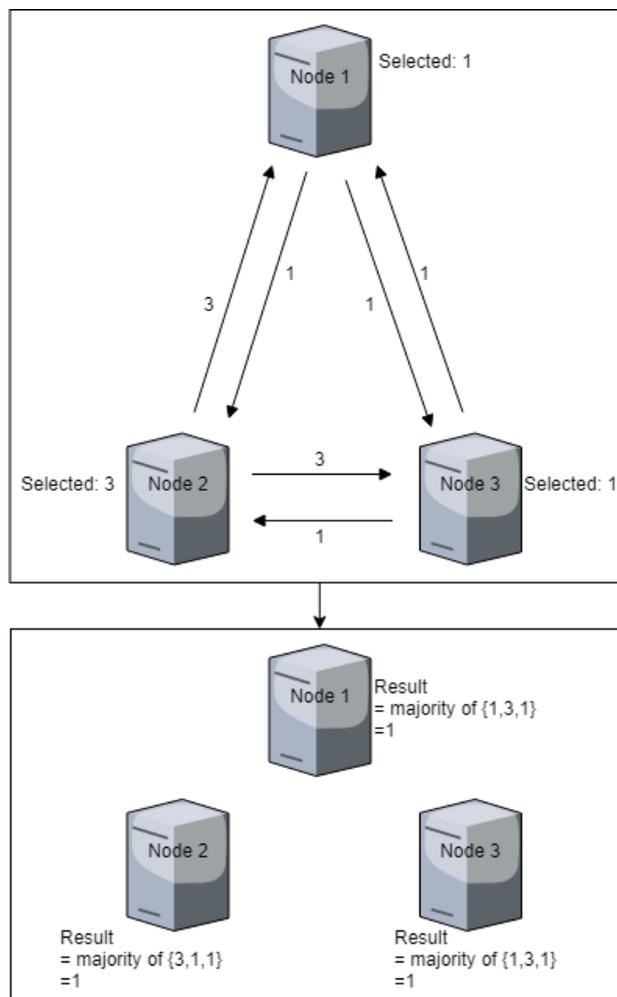


Figure 3-13: An example of the consensus protocol

A node does not need to wait for the broadcast message from all nodes, as some of the nodes may go offline and the message may get lost. The final result can be determined after receiving the same result from more than or equal to half of the total number of

nodes (Figure 3-14).

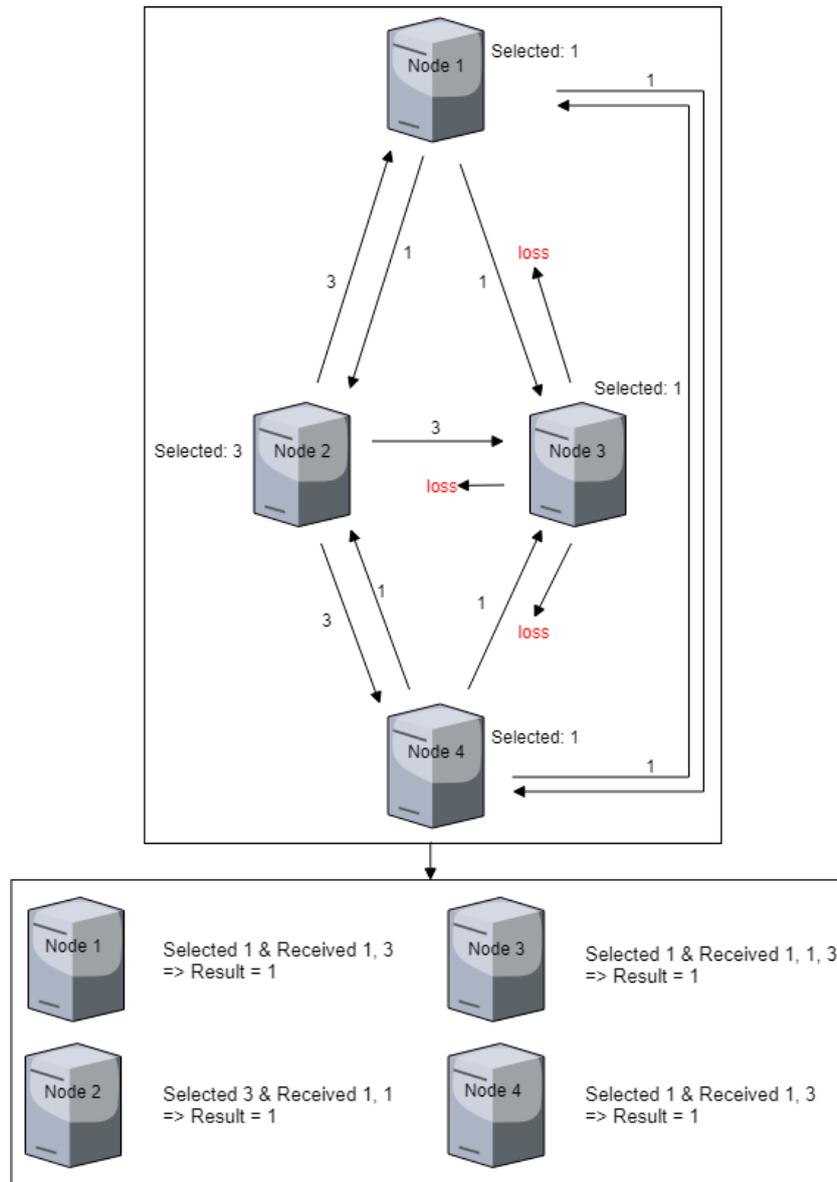


Figure 3-14: An example when some messages are lost

In case of timeout or the block generating node go offline just after the consensus protocol, then just ignore and leave it for the next round.

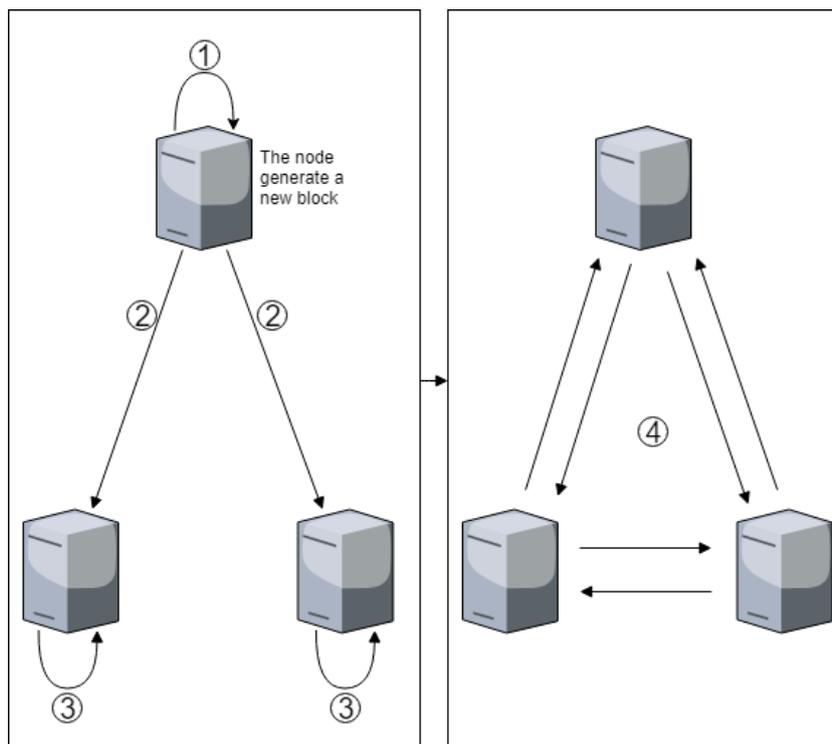
### **Block broadcasting protocol**

After a node X is selected, the following procedure will be performed (Figure 3-15):

- i. X generates a new block, including all verified ballots in the last time slot. The block will contain a pointer (previous hash) that link to the most recent block

that is valid.

- ii. X broadcast the block in the network.
- iii. All other nodes will save it first and then verify the block, including the block sequence number, block hash, etc.
- iv. If the block is valid, all nodes will sign on the block hash and broadcast the signature in the network.



*Figure 3-15: Steps of the block broadcasting protocol*

A block will be considered valid when more than half of the servers agree and sign on it. That's why we require at least half of 'honest' node participating in the blockchain network, otherwise, it is possible that the blockchain contains unvalidated data, although these unvalidated data may be detected and reported by the public. If a block is proven invalid, all the ballots within this block will be considered putting into the block for the next round.

# 4. Implementation

## 4.1. Overview

From a client-side viewpoint, i.e. election administrator, trustee or voter, we have implemented the end-to-end voting process with a user-friendly interface, which accessible using browser. While most of the encryption, decryption, and keys related computation are done on the client-side, we also provide some key pair generation function so that they can use the application without the need of pre-generating keys. Therefore, one can easily create an election, add voters/trustees, vote, tally and view the result by just follow the steps on the web pages.

From the server-side viewpoint, i.e. a node in the blockchain network, we have implemented almost everything to run an election using blockchain as described in Section 3. From nodes connection, required verification on data, to all blockchain related communication. Also, it can listen to client requests to perform different operations, such as modifying election details and voters/trustees key pair.

## 4.2. Server-side

### 4.2.1. Use of programming language

We use Node.js [25], which is a kind of JavaScript, as the programming language. Node.js first release in 2009, and become a popular language nowadays for developing a server-side application to process HTTP requests. Because of the popularity, it makes us easier for developing as there are many discussions and solutions online for problems encountered. Also, using a scripting language like JavaScript make debugging simpler. Moreover, Node.js has a huge amount of well-developed and popular modules available online, which we can reuse them to avoid writing a lot of tedious code, thus improving the reusability of our program.

For the database, we decided to use NoSQL<sup>5</sup> type instead of a relational database. The main reason is that NoSQL does not require a predefined structure, which can provide more flexibility. For example, a block in the blockchain may contain ballot or election details, thus not all fields are needed in each block. Also, the tables needed in our design do not require strong relation to each other, such as address table and block table. More importantly, using NoSQL can directly eliminate the risk of SQL injection security attack.

In all NoSQL type of database, we chose MongoDB [26]. It is the most popular NoSQL database nowadays<sup>6</sup>, which first released in 2009. Furthermore, MongoDB store documents in JSON format, which is the exact format that JavaScript represent objects, so it should be more efficient when using with Node.js.

## 4.2.2. System architecture

### Architecture design overview

Here is the overall design of a node in the blockchain network (Figure 4-1), i.e. trustee's computer. When there are multiple nodes in the network, every node will share the same design.

---

<sup>5</sup> Non-relational databases that do not use Structured Query Language

<sup>6</sup> Rank first in November 2018 for NoSQL database model, according to <https://db-engines.com/en/ranking/document+store>

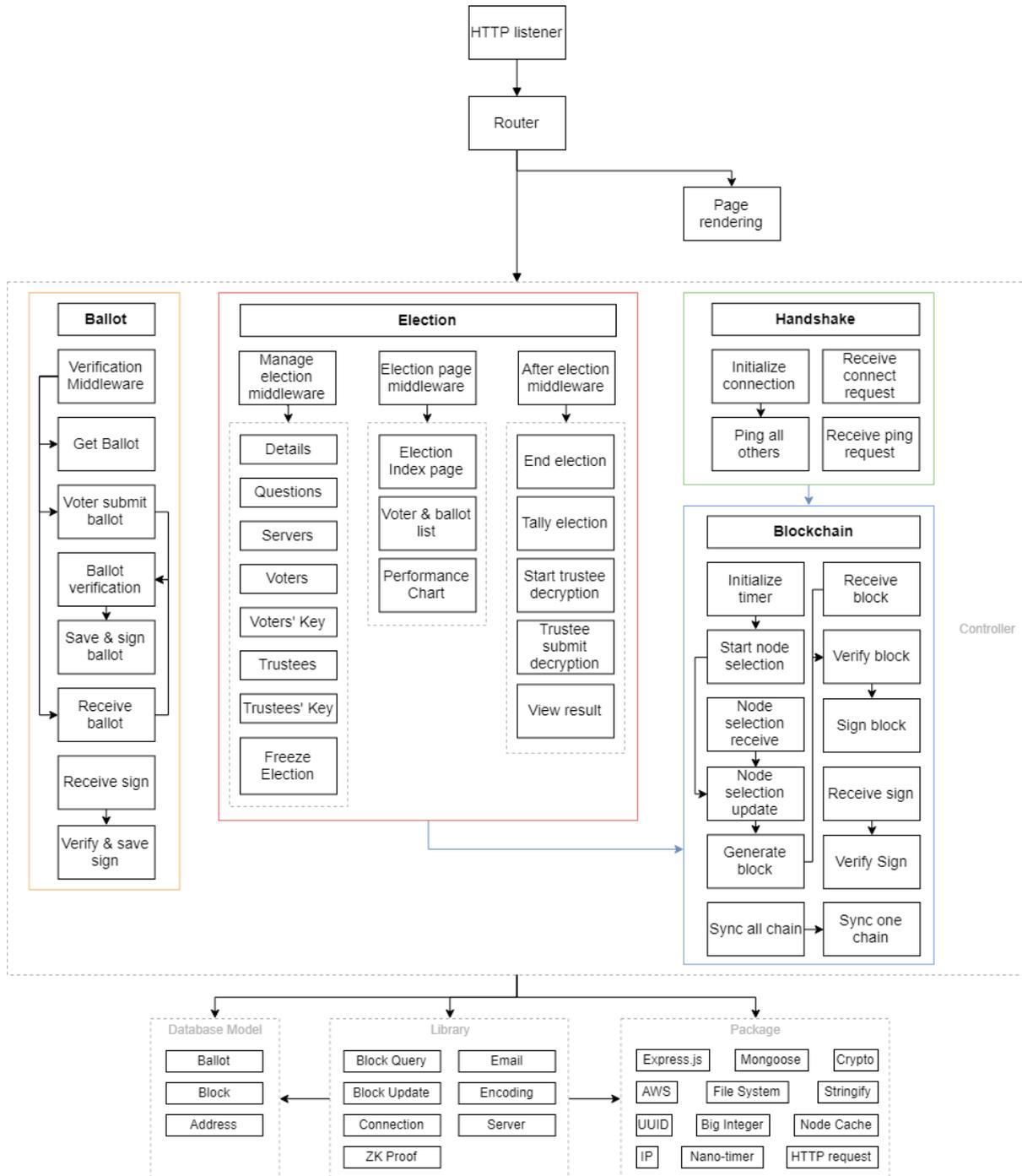


Figure 4-1: Architecture of a node in blockchain

### HTTP listener

This is a predefined module that set up the server and listen to HTTP requests. This is the beginning of the system, all request received will proceed to router for processing.

## **Router**

If the request is just a page view, then the router will pass it to page rendering engine, which will generate the HTML document required from templates and return the request. Otherwise, the request will be sent to corresponding controllers for further processing.

## **Controller**

Each system's component will have its own controller, which responsible for handling the related request. This will be independent with the front-end code, so a controller cannot invoke by users directly, only via routers.

## **Handshake**

When the system starts, this controller will prompt the server administrator for the address that this node should connect to, then follow the protocol defined in Section 3.3.5 to join the blockchain network. After that, it mainly handles ping requests or new connection request from other nodes.

This controller is connected to the Blockchain controller only for performing chain synchronization after setting up the connection.

## **Election**

This controller is mainly used by election administrator and trustees to configure an election. The 3 middleware will first inspect the request before passing to others function, in order to check if the user has the right initialize the request. For example, election administrator can only access management pages before an election freeze; election administrator can only start tally an election after it ends, etc.

This controller is also connected to the Blockchain controller to initialize a timer for block generation after an election freeze.

## Ballot

This controller mainly handles the request from voters to prepare and cast a ballot. When a ballot arrives, it follows the protocol stated in Section 3.3.6 to verify and broadcast the ballot.

The verification middleware will check if the election has frozen, started and not yet ended before passing the request to other functions. Notice that this is also applicable to ballot received from broadcasting. It's because we cannot fully trust each node, so we cannot fully trust their timestamp also. Therefore, each node will use its own judgement on whether the ballot is cast in a valid time period. This is usually not a problem when the network delay is tiny. Otherwise, election administrator needs to adjust the election end time respectively.

Ballot controller is not connected to the Blockchain controller because the ballot will be saved to the database, so they interact through the database.

## Blockchain

In the beginning, this controller will start a timer for block generation for all ongoing election. Every time when the timer expires, it will start the node selection and block generation process as described in Section 3.3.7. Besides, it also responsible for listening to chain synchronization request and return the requested blocks.

## Library

The library is a collection of handy functions that used multiple times across different controllers, so as to further improve system modularity. Below are some brief descriptions of these library function.

**Block Query:** Includes functions that query from the Block collection from the database.

**Block Update:** Save new blocks and signature to the Block collection. It also includes

functions that execute when receiving enough amount of signature for a block, such as marking ballots as “in the blockchain”.

Connection: Make use of the “HTTP Request” package to broadcast or send data to other nodes.

Server: Update or query documents for the Address collection in the database.

Encoding: Change the encoding of a string between base64<sup>7</sup> and Hexadecimal. Hexadecimal is usually used for calculation of big integer, while Base64 is usually used when transferring or saving data in order to save space.

ZK Proof: Perform zero-knowledge proof verification as stated in Section 3.2.2.

Email: Sending out emails, as a notification for voters and trustees. While we make use of Amazon Web Service (AWS) to send email, other developers can make use of other emailing services by modifying the code here only.

### **Database model**

A model for a collection defines its fields and related data types. Notice that creating a new collection in MongoDB do not require fields and data types to be predefined, one can always add a document with fields and data types different in the defined database model. However, using a database model can improve the efficiency and consistency in the collections. When adding or editing data in a collection, the system will always match with the defined fields first, unless it could not find one.

Below are the details of the database models that we defined.

#### **Address**

To store the network address of all nodes in the blockchain network. (Figure 4-2)

---

<sup>7</sup> Base64 encode binary data in text, such that every character represents 6 bits. There are 64 choices of character.

IP: IP address of the node, in IPv4<sup>8</sup> format.

Port: Port number to access the node from the corresponding IP address.

instanceID: The ID of the running instance of the node, in UUID<sup>9</sup> format. This ID should change every time the application restart.

serverID: The ID of the node, in base64 format. This is the hash of server public key.

serverKey: RSA public key of the node, in PKCS#8<sup>10</sup> format

updatedAt: The UTC time of the latest ping request from the node.

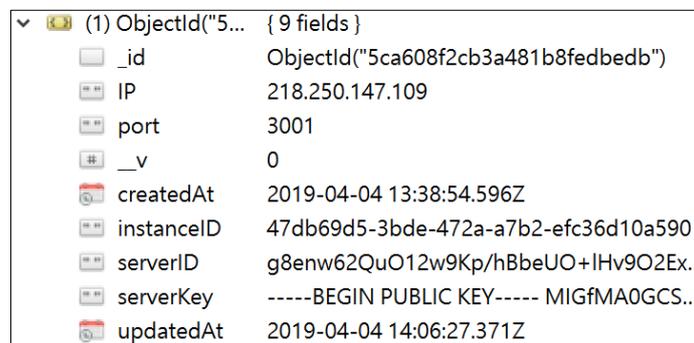


Figure 4-2: An example document in the Address collection

## Ballot

To store all ballots received either from voter or from other nodes, this mainly used for storing the ballots before adding them into a block. (Figure 4-3)

electionID: The election ID that this ballot belongs to, in UUID format.

voterID: ID of the voter in this election that submit the ballot.

answers: Array of {<choice>, <overall\_proof>} with size equal to the no. of questions in the election.

<choice>: Array of object in the form of {c<sub>1</sub>, c<sub>2</sub>, <choice\_proof>}, number

---

<sup>8</sup> Internet Protocol version 4, the format is x.x.x.x where x is integer from 0 to 255

<sup>9</sup> Universally Unique Identifier, the format is xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx where x can be any number or lower-case character

<sup>10</sup> Public Key Cryptography Standards #8, correspond to Private-Key Information Syntax Standard

of elements should equal to number of options in that question.  $c_1$  and  $c_2$  are the encrypted answer as stated in Section 3.2.1. They are encoded using base64 format.

<choice\_proof>: Array of <ZK\_proof>, should have only 2 elements.

<overall\_proof>: Array of <ZK\_proof> with size equal to maximum minus minimum number of choice plus 1.

<ZK\_proof>: Object with the form of  $\{a_1, a_2, e, f\}$ , as stated in Section 3.2.2  
- Voter honest encryption.

voterTimestamp: The UTC time when the voter submits the ballot. This mainly used for the calculation of application performance.

voterSign: Signature of voter on the ballot encoded in base64 format. This means the voter use his private key to encrypt the hash of  $\{\text{electionID}, \text{voterID}, \text{answers}, \text{voterTimestamp}\}$  using SHA256 hash function. One can verify the hash via voter's public key.

receiveTime: The UTC time when the first node receives the ballot from voter. This mainly used for the calculation of application performance.

sign: Array of <trustee\_sign\_ballot> with size equal to the no. of unique signatures received.

<trustee\_sign\_ballot>: Object with the form of  $\{\text{serverID}, \text{ballotSign}\}$ . serverID is the ID of the node that belongs to this signature. ballotSign is the signature in base64 format. This means the server use its private key to encrypt the hash of  $\{\text{electionID}, \text{voterID}, \text{answers}, \text{voterSign}, \text{receiveTime}, \text{voterTimestamp}\}$  using SHA256 hash function.

inBlock: A boolean value to indicate whether the ballot is in the blockchain or not.

Notice that the sign array may be different among different nodes, as it is possible that a signature may get lost and not received by some nodes. So, the node that generates the block for this ballot will use its own version of sign array. Also, the inBlock only reflect the local situation. If there is a fork in blockchain among the network, the inBlock value may be different for each node.

<ul style="list-style-type: none"> <li>▼ (1) ObjectId("5c9f341a5e5916af7fb8c31c")</li> <li>  _id</li> <li>  electionID</li> <li>  voterSign</li> <li>  __v</li> <li>  ▼ answers</li> <li>    ▼ [0]</li> <li>      ▼ choices</li> <li>        ▼ [0]</li> <li>          c1</li> <li>          c2</li> <li>          &gt; proof</li> <li>          &gt; [1]</li> <li>          &gt; [2]</li> <li>        ▼ overall_proof</li> <li>          ▼ [0]</li> <li>            a1</li> <li>            a2</li> <li>            e</li> <li>            f</li> <li>            &gt; [1]</li> <li>            &gt; [2]</li> <li>  createdAt</li> <li>  receiveTime</li> <li>  ▼ sign</li> <li>    ▼ [0]</li> <li>      _id</li> <li>      serverID</li> <li>      ballotSign</li> <li>  updatedAt</li> <li>  voterID</li> <li>  voterTimestamp</li> </ul>	<ul style="list-style-type: none"> <li>{ 11 fields }</li> <li>ObjectId("5c9f341a5e5916af7fb8c31c")</li> <li>dca7eec0-acf1-4936-879c-6fd19721016e</li> <li>ZwJWP0IIOSQgj072hjp1jv4+vw+hr73qxRgR16Rm...</li> <li>0</li> <li>[ 1 element ]</li> <li>{ 2 fields }</li> <li>[ 3 elements ]</li> <li>{ 3 fields }</li> <li>W+e4Ku2X1uBN/JCWFV3YP3KgMHHS+vP4Gz9i4Lfv...</li> <li>iw8UVcqFs5TEywi1HDv3s4sEGCkIQ+mRDHRjRbDV...</li> <li>[ 2 elements ]</li> <li>{ 3 fields }</li> <li>{ 3 fields }</li> <li>[ 3 elements ]</li> <li>{ 4 fields }</li> <li>LU5fCx8E8ktMLw+lJxC6nguDc8NrlslygqMguB6HI6PQ=</li> <li>Tpllv+upxTUHiffsFyHWImP0itEiUxZgoDbnWTsoX7k=</li> <li>nXu94KA85IJJFumWPMaw+KfMwGqO9tHj25He9V...</li> <li>N1VN+S62gwNJ52kbRIPdUqy31yrDoHLlpeJzY/22y3...</li> <li>{ 4 fields }</li> <li>{ 4 fields }</li> <li>2019-03-30 09:17:14.265Z</li> <li>2019-03-30 09:17:13.845Z</li> <li>[ 1 element ]</li> <li>{ 3 fields }</li> <li>ObjectId("5c9f341a94d2c249fcd47ba4")</li> <li>g8enw62QuO12w9Kp/hBbeUO+IHv9O2ExmMM/5d...</li> <li>pK0I0jmQN0aamP35IIP1pm51yefyKuFiXTBuq0lcFW...</li> <li>2019-03-30 09:17:14.437Z</li> <li>voterGen_0</li> <li>2019-03-30 09:17:13.499Z</li> </ul>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 4-3: An example document in Ballot collection

## Block

To store blocks in all election that the node is responsible for. (Figure 4-4)

blockUUID: ID of the block, in UUID format. This uniquely identify a block, no matter which blockchain it belongs to.

electionID: The election ID in UUID format. This identify which blockchain does the block belongs to.

blockSeq: An integer that represent the position of the block in the chain. The first block is 0.

previousHash: The hash value of previous block, encoded in base64 format.

blockType: Represent the type of data that this block used to store. Should be 'Election Details', 'Ballot' or 'Election Tally'.

data: An array, the actual structure depends on the block's type, details would be described below.

hash: SHA256 hash value of {blockUUID, electionID, blockSeq, previousHash, blockType, data}, encoding using base64 format.

sign: Array of <sign\_block> with size equal to the no. of unique signatures received.  
<sign\_block>: Object with the form of {serverID, blockHashSign}. serverID is the ID of the node that belongs to this signature. blockHashSign is the signature in base64 format. This means the node use its private key to encrypt the hash of the block using SHA256 hash function.

Similar to ballot, the sign array can be different in different nodes.

For a 'Ballot' block, data will be array of ballot object, which is the same as described in Ballot section, except it will not include the inBlock field.

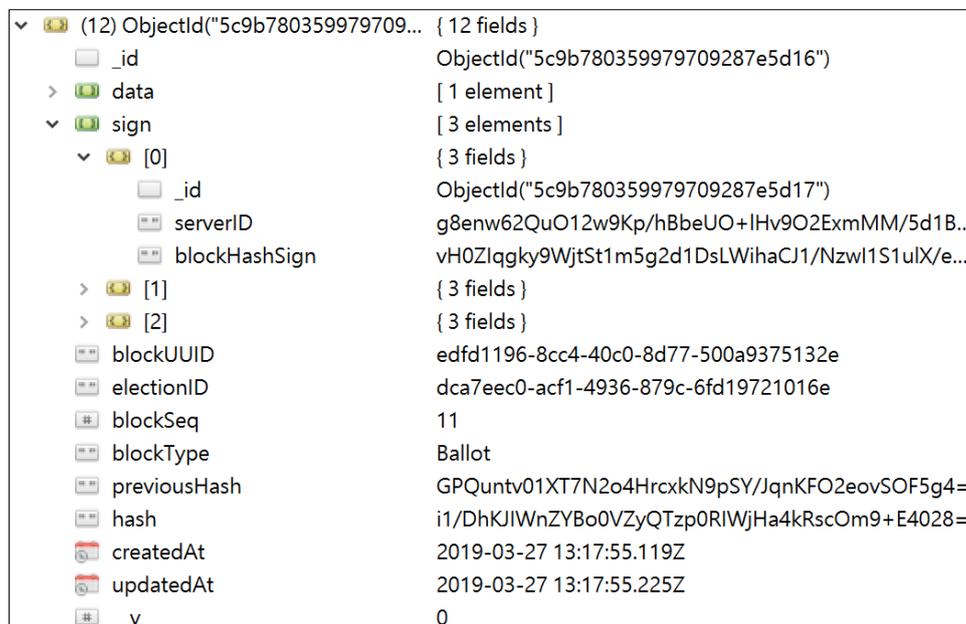


Figure 4-4: An example document in Block collection

Data may have the following fields if it is a 'Election Details' block:

name: Name of the election

description: Election description, such as purpose of the election

start: The UTC time that indicates election start.

end: The UTC time that indicates election end.

questions: Array of objects in the form of {question, <answers>, min, max}

question: Text of the question

<answers>: Array of choice in text for the question

min: Minimum number of choice a voter can choose

max: Maximum number of choice a voter can choose

key: An object in form of {p, g, y}, which represent the election public key as stated in Section 3.2.1. They are encoded in base64 format

admin: An object of the form {pubKey}

pubKey: RSA public key of the election administrator, in PKCS#8 format

servers: Array of objects in the form of {serverID}

serverID: ID of the server that participate in the election

voters: Array of objects in form of {id, public\_key}

id: ID of the voter

public\_key: RSA public key of the voter, in PKCS#8 format. This will be empty if the voter is removed from the election

voterSign: Signature of the voter using his/her old private key. Only applicable when voter is changing his/her RSA key pair

trustees: Array of objects in form of {trusteeID, email, y}

trusteeID: ID of the trustee

email: email of the trustee

y: trustee public key, as stated in Section 3.2.1

frozenAt: The UTC time when all election details are locked

adminSign: Signature of admin on the data that is changed

Data may have the following fields if it is a 'Election Tally' block:

endAt: The UTC time when election ends

tallyAt: The UTC time when election start tallying, so that servers will start aggregating ballots

tallyInfo: Array of objects in the form of {serverID, start, end} with size equal to the no. of batches or no. of servers that assigned to perform the aggregation

serverID: The ID of the node for this assignment

start: The starting index of voter, sort by voter ID.

end: The ending index of voter.

partialTally: Array of <question\_tally> with size equal to the no. of questions

<question\_tally>: Array of object in the form of {c1, c2, c1x} with size as the no. of choices. c1 and c2 are the aggregation of each ballot's  $c_1$  and  $c_2$  respectively. c1x is the current product of trustee decryption, refer to Section 3.2.1 - Tallying election

decryptAt: The UTC time when it starts to send the tally to trustees for decryption

partialDecrypt: Array of partialTally as described above, size should be the no. of batches divided.

proof: Array of <batch\_proof> with size equal to the no. of batches divided.

<batch\_proof>: Array of <question\_proof> with size as no. of questions.

<question\_proof>: Array of object with the form of {a1, a2, f, d} refer to Section 3.2.2 - Trustee honest decryption, the size is the no. of choices.

result: Array of <question\_result> with size equal to the no. of questions.

<question\_result>: Array of <choice\_result> with size as the no. of choices.

<choice\_result>: Array of integer with size as the no. of batch divided. The integer indicates the no. of vote to this choice in this batch. Sum of this array will be the final count for the choice.

adminSign: Signature of admin on the action such as end and tally election

## **Package**

A package is an opensource Node.js module that allows us to add specific functionality to our program to reduce our amount of work. We always download package

carefully by finding out its popularity, stability, and related online discussions, in order to avoid our system being affected by possible vulnerabilities in the package.

### 4.2.3. Detailed explanation of components

In this section, we use sequence diagram to describe how each component interact with each other for some main functionality, including connect a node to the blockchain network (Figure 4-5), create an election (Figure 4-6), add a voter/trustee to an election (Figure 4-7), change the key pair of a voter/trustee (Figure 4-8), freeze an election (Figure 4-9), voter submits a ballot (Figure 4-10), select node with block generation, and end an election from tally to decrypt (Figure 4-12).

#### Connect to the blockchain network

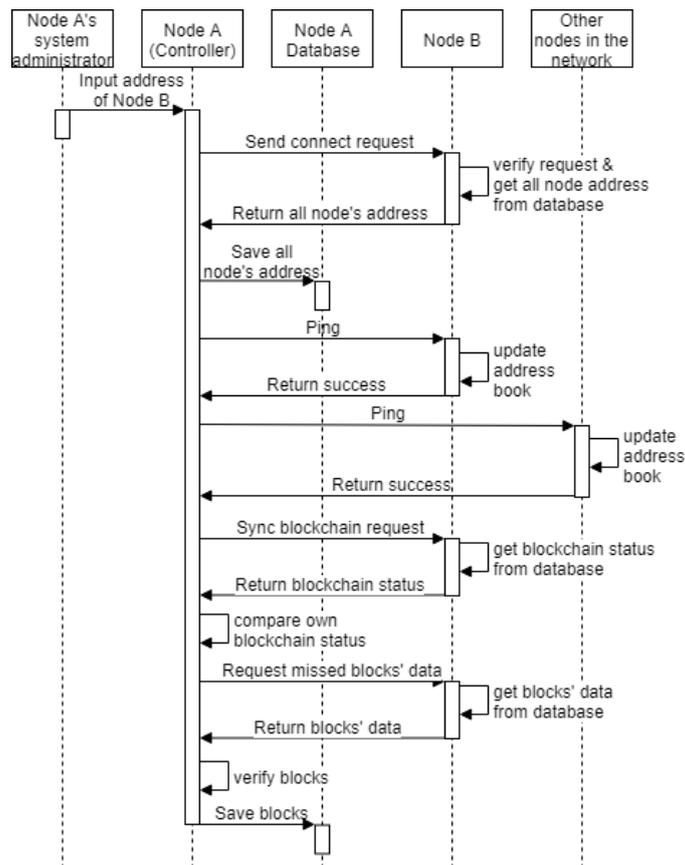


Figure 4-5: Sequence diagram of connecting a node to the blockchain network

### Create an election or change election details

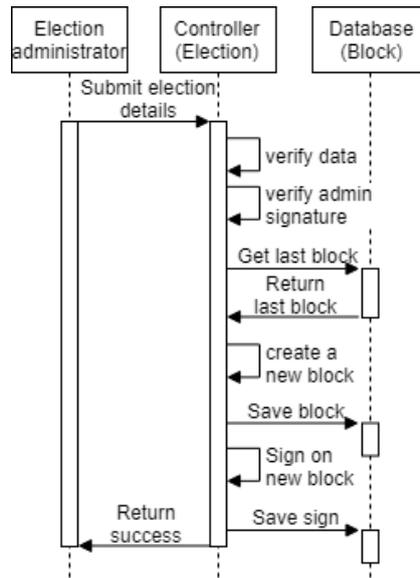


Figure 4-6: Sequence diagram of creating an election or update details like questions

### Add a voter/trustee

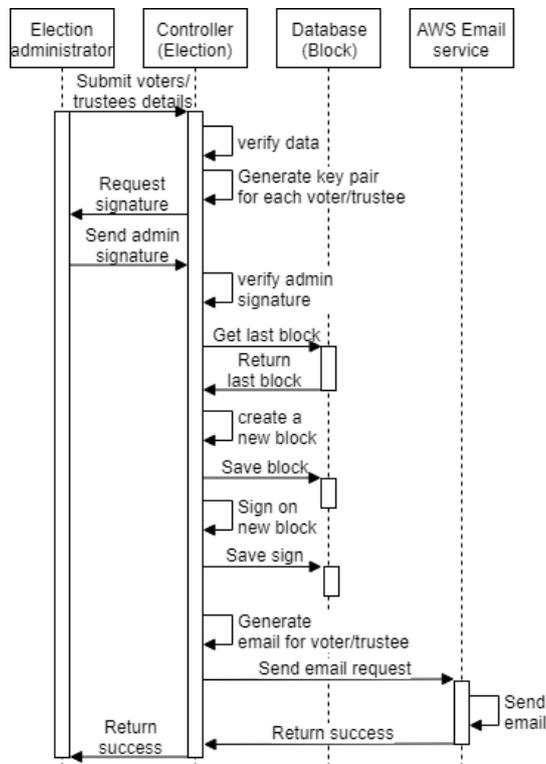


Figure 4-7: Sequence diagram of adding voters/trustees

### Voter/Trustee change key pair

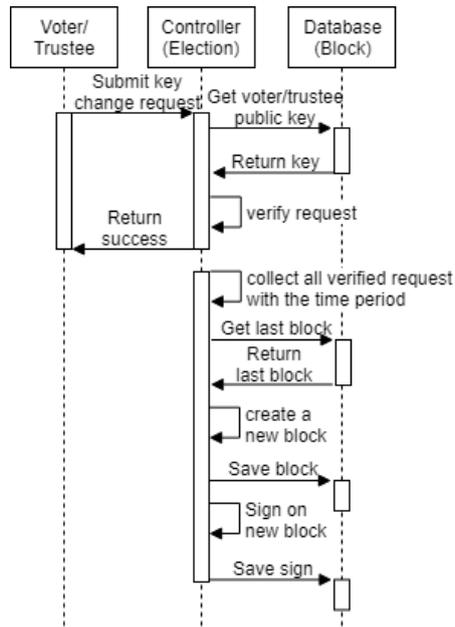


Figure 4-8: Sequence diagram of changing the key pair by a voter/trustee

### Freeze an election

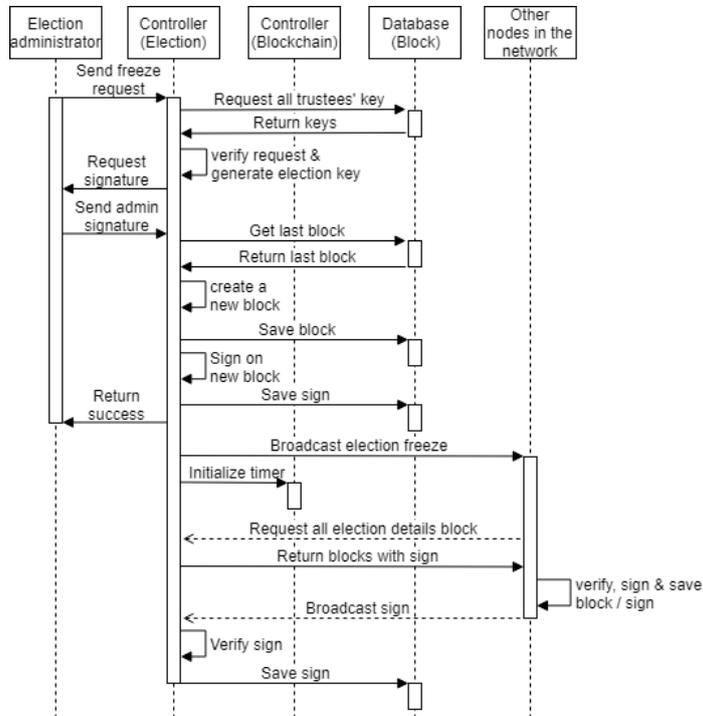


Figure 4-9: Sequence diagram of freezing an election with broadcasting blocks

## Processing a submitted ballot

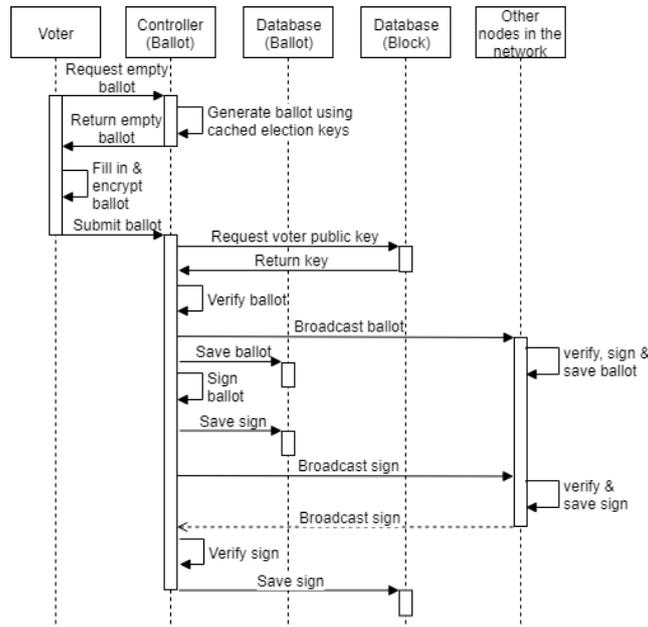


Figure 4-10: Sequence diagram of when a ballot submitted by a voter

## Generate new block

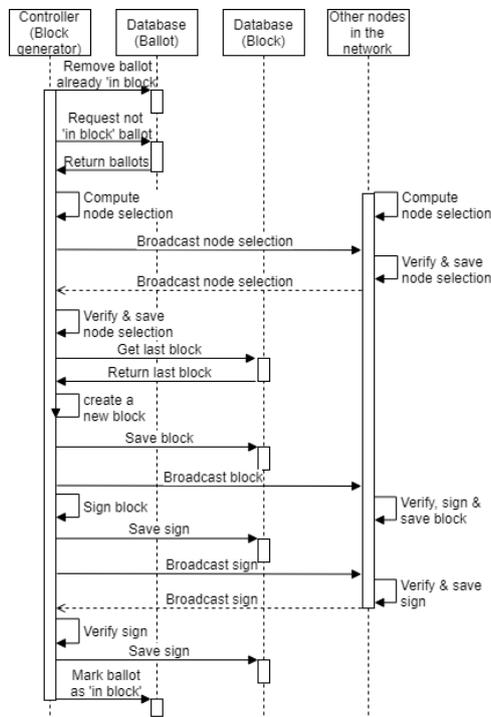


Figure 4-11: Sequence diagram of the block generation process

## Tally an election

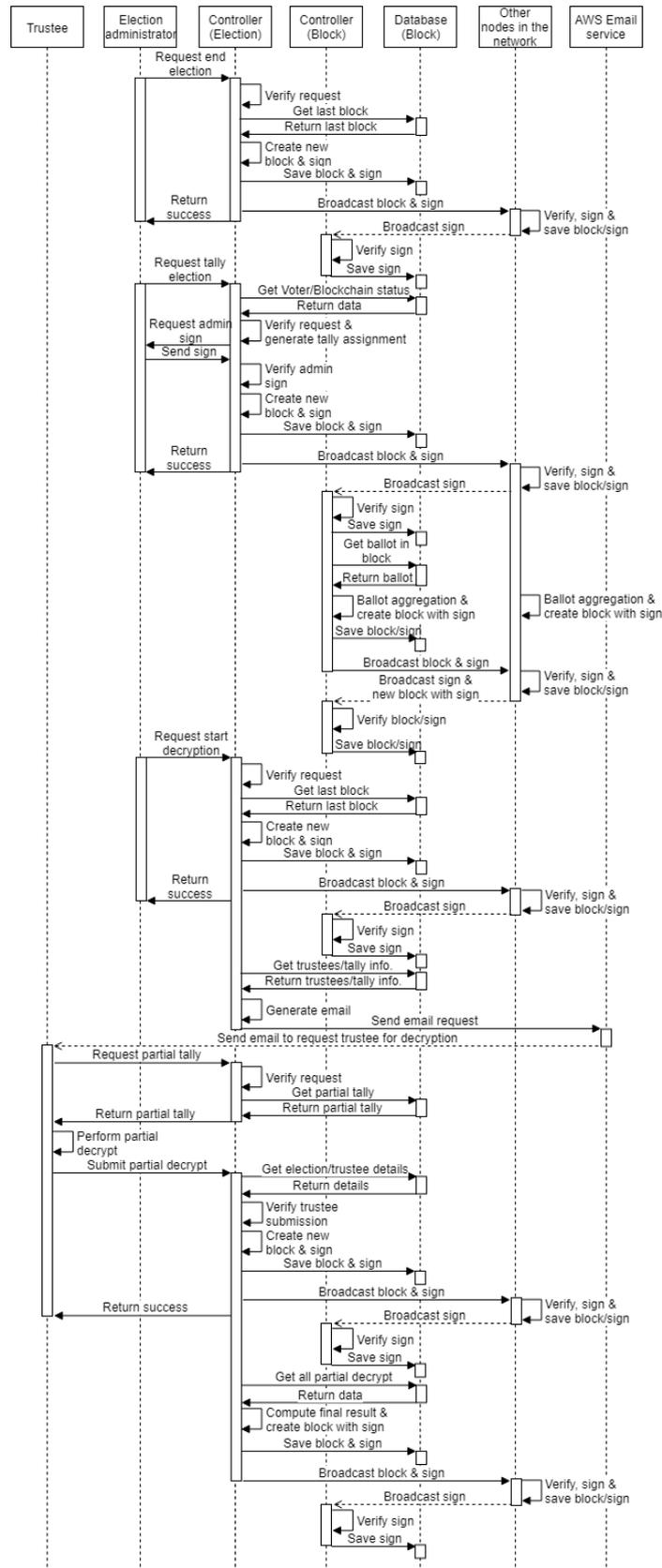


Figure 4-12: Sequence diagram from ending an election to computing result

## 4.3. Client-side

### 4.3.1. Choosing the deliverables

While the topic of our project is about voting on mobile devices, we decided to use a website as our deliverable for the following reasons:

- A web application is more portable, either a computer or a mobile phone can access the website as long as it has a network connection. Thus, the application may be able to reach more people easily.
- People tend not to install many applications on their smartphone, especially if the application is not used frequently.
- Developing a web application is simpler when compared to a mobile application, so we can spend more time on testing.

There are people saying that a mobile application is better for achieving security [27] and privacy [28]. However, we have much more experience in developing web application than mobile, thus we have more experience in tackling the security threat on a web application, such as Cross-Site Scripting (XSS) attack.

### 4.3.2. Use of programming language

The languages used here are HTML, CSS and JavaScript, which are just a standard set of languages for the client-side of a website. For CSS, we have used the Materialize framework, since it is popular, responsive and it's design align with many mobile applications. Also, it includes some useful plugin such as date/time picker which allows users to fill in a form easily. For JavaScript, we have included the jQuery JavaScript library, which is the most popular one. It provides handy functions such as sending AJAX<sup>11</sup> request and accessing elements on a webpage. Besides, Big Integer library also used for the same reason as stated in Section 4.2.2.

---

<sup>11</sup> Asynchronous JavaScript and XML. A way of sending HTTP request to reduce data size.

### 4.3.3. User interface

#### Create election and editing details

eVoting Create

Create election

#### About the election

Election name  
Test election

---

Election description  
an election

---

Start date	Start time
Apr 10, 2019	00:00

---

End date	End time
Apr 30, 2019	00:00

---

#### Election public key

Prime number - p (Base64 Encode) GENERATE

qUN9TEqk5T2fepMIYeRiF0NPIHwwFml3VjwADg4scs=

---

Generator - g (Base64 Encode)

Aw = =

---

#### Election administrator key

Public key GENERATE

-----BEGIN PUBLIC KEY-----  
MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAKSgMaEkzRcQbVvqTopEbfqAR4rq6cdV  
56js5aJSN+hLNq6dApOdFnBmX3IqGZe3CZBi+WGH0LgzaOjQ4AGbsCawEAAQ==  
-----END PUBLIC KEY-----

---

Private key (for signing only, wont send to server)

-----BEGIN PRIVATE KEY-----  
MIIBWwIBADANBakghkiG9w0BAQEFAASCAT4wgdE6AgEAAKEApKAwoSTNFxBTW+pO  
ikR5WoBHiurpx1XnomxJoll36Es2rp0CK50WcHEzHeKoZl7cJkGL5YYlQuDNo6NB  
DgAZuwIDAQABAKAccBPCq3HGly9E9ne0yNcS1yAerkmOBffqjdyebad7t+LB/S6O  
Ul0goy5TVPIY98Bj20/uugdCmDqzNqizG6ZZAIEA7T6mUkz+k86pg4chG/AA3M5E  
fEqm2cE7ndWbpej+BY8CJCQx0+JTH1DNd84luM3kKAXSZ7KHNPmM2q3j3putSQuL  
FQIqYeRx+1+wI4i4RX9ImAaGNjYqCDA1Z78W0Z+sY98pwo0CICpPbmBLP0sEt+Q  
HseCM8caJkGvNqjFMI5Wt2lU9YVAIEA64E956VRxyym5HE13AzRJRCSwUdyNwS  
xRL8afyCk3Q=  
-----END PRIVATE KEY-----

---

CREATE

© 2019 eVoting Server ID: +D11XQq4KIOpOvu87qxMeh151jhWeqvZ4ZkRgARE4=

Figure 4-13: A filled in election create form

Figure 4-13 shows the interface for creating an election. Election administrator needs to fill in all the field to create an election. When choosing date and time, we provide pickers (Figure 4-14) for convenience, as the user does not need to worry on the date/time format. For the keys, users can get them elsewhere and paste into the form, or by just clicking the “Generate” button then keys will be generated locally. Notice that the private key here is only for signing purpose and will not send it outside.

We use a similar page for editing election details, except that those key related fields will be disabled as it is not allowed to edit keys.

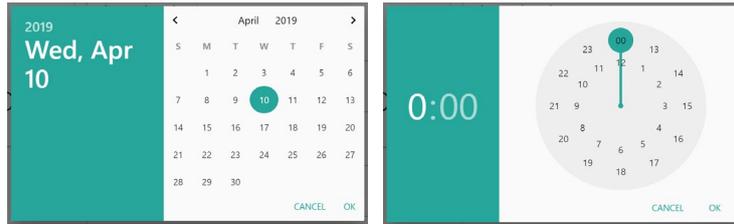


Figure 4-14: Date-picker (left) and Time-picker (right)

By submitting this form, a request will be sent to the server. The browser will redirect to Election management page if success.

### Election management page

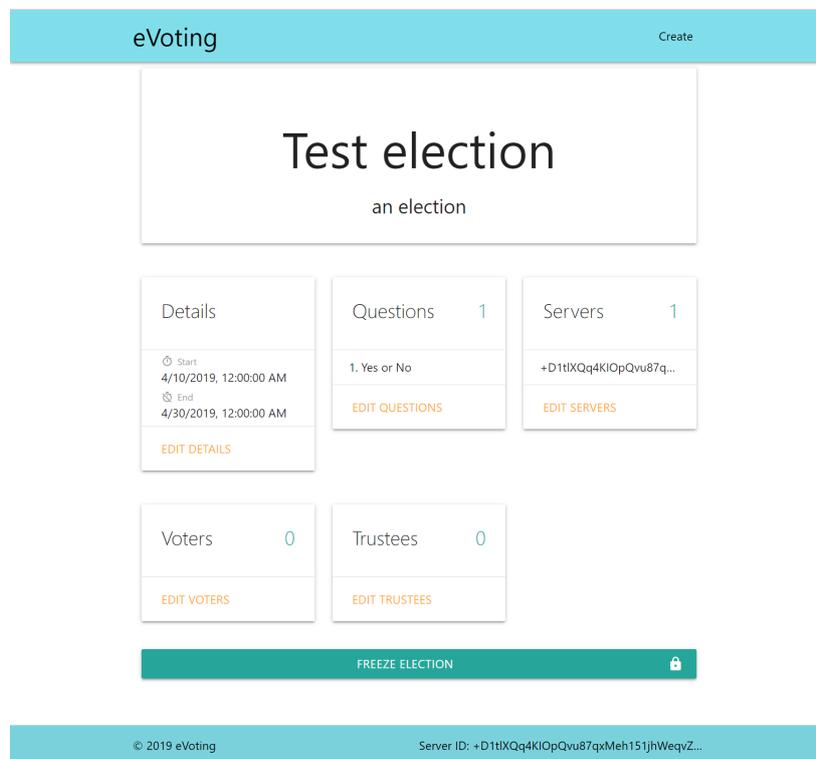
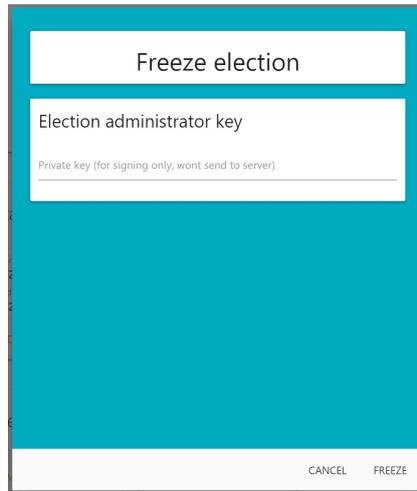


Figure 4-15: Management home page of an election

Election management page (Figure 4-15) gives a quick overview of all settings in an election, he/she can edit them via clicking the link at the bottom of each small box. Notice that all users can view these management pages, but only election administrator which has the private key can update the settings.

By clicking the “Freeze Election” button, a modal (Figure 4-16) will be popped out. Election administrator should provide the private key to sign on the request before sending

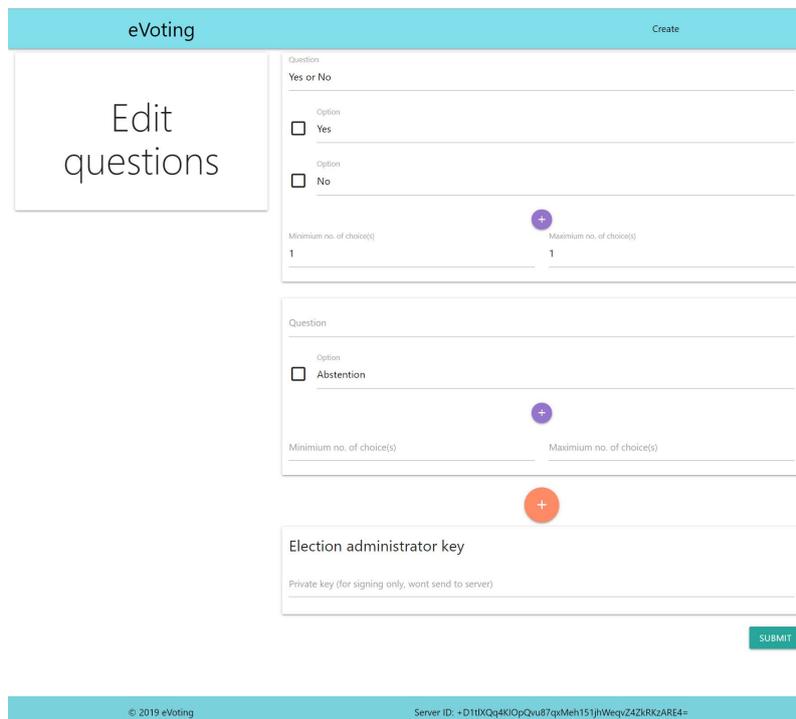
to the server. After success freezing, the browser will redirect to Election home page.



A modal window titled "Freeze election" with a teal background. It contains a text input field labeled "Election administrator key" with a placeholder text "Private key (for signing only, wont send to server)". At the bottom, there are two buttons: "CANCEL" and "FREEZE".

Figure 4-16: Modal for freezing election

### Question list editing



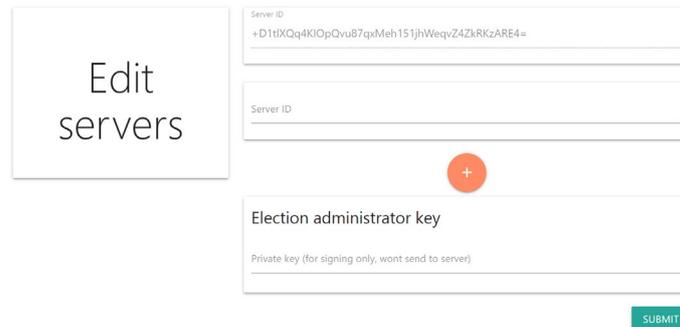
The "eVoting" interface for editing questions. It features a teal header with "eVoting" on the left and "Create" on the right. A large box on the left says "Edit questions". The main area contains three question boxes. The first is titled "Yes or No" and has two options: "Yes" and "No", each with a checkbox. Below it are input fields for "Minimum no. of choice(s)" (value 1) and "Maximum no. of choice(s)" (value 1), with a blue "+" button between them. The second question box is titled "Abstention" and has one option: "Abstention" with a checkbox. Below it are input fields for "Minimum no. of choice(s)" and "Maximum no. of choice(s)", with a blue "+" button between them. The third question box is titled "Election administrator key" and has a placeholder text "Private key (for signing only, wont send to server)". Below it is a green "SUBMIT" button. At the bottom, there is a teal footer with "© 2019 eVoting" on the left and "Server ID: +D10XQq4KI0pQvu87qMeh15JhWepqZ4ZkRkzARE4=" on the right.

Figure 4-17: The user interface for question editing

Figure 4-17 shows the view when editing questions. To add a question, just click the orange button, then a new 'question box' will be shown. By default, the first option is 'Abstention' for the reason stated in Section 3.2.3 - Knowledge of who has voted. Users can

add options by clicking the purple button. On the other hand, emptying the input fields can delete the options or questions.

### **Server list editing**



*Figure 4-18: Form for editing servers*

This page (Figure 4-18) is for editing the server (node) list that will be participating in the blockchain network of this election. Similar to editing questions, adding a server can be done by clicking the orange button and input the server ID, while deleting a server can be done by emptying the input field.

### **Voters/Trustees management**



*Figure 4-19: The user interface of voters' management page*

The interface for adding voters and trustees are similar, except that the system will save and display the email for trustees but not voters. Figure 4-19 shows the page for managing voters, which contains the voter list and a delete button for each voter. By clicking the delete button, it will show a modal similar to Figure 4-16 for election

administrator to provide the private key and sign on this action.

To add voters, click the “+” button on the bottom-right corner, then it will pop up a modal (Figure 4-20) to input the voter ID and email address of the new voters. By default, the voter ID is randomly generated automatically for the reason stated in Section 3.2.3 - Knowledge of who has voted.

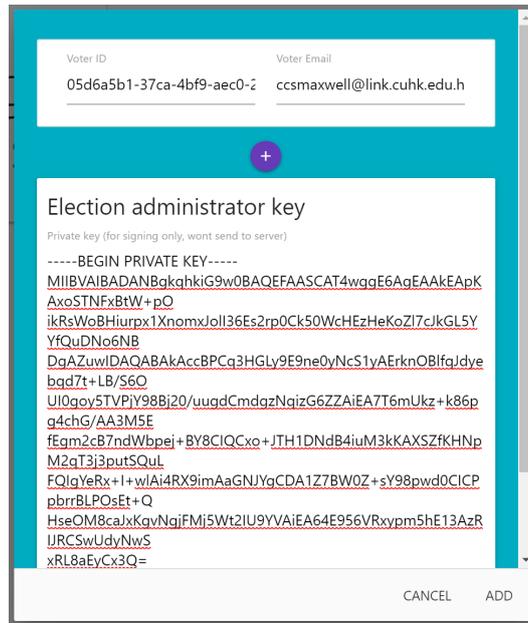


Figure 4-20: Modal for input new voters' information



Figure 4-21: An example email that sent to the new voter

After adding the voter, a unique email containing a private key (Figure 4-21) will be sent to each of the new voters, according to the first way stated in Section 3.2.3 -

Authentication method.

### **Change key pair by voters/trustees**

The screenshot shows a web interface titled "Manage trustee key". It is divided into three main sections:

- Election public key:** Contains two fields: "Prime" with the value "qUN9TEqkST2lepMYeRIFONPIHwwFml3VjwADg4scs=" and "Generator" with the value "Aw==".
- New trustee key:** Contains two input fields: "Public key" and "Private key (for creating proof only, wont send to server)". A green "GENERATE" button is located to the right of the "Public key" field.
- Trustee info:** Contains two input fields: "Trustee ID" and "Private key (for signing only, wont send to server)". A green "SUBMIT" button is located at the bottom right of this section.

*Figure 4-22: The user interface for changing trustees' key pair*

Again, the view of changing key pair is much the same, they both need to provide the new public key, voter/trustee ID and current private key for signing purpose. However, trustees will need to also input the new private key, this is used for creating a proof as stated in Section 3.2.2 - Trustee knowledge on private key, and will not send to the server. Besides, voters are using RSA key, while trustees are using ElGamal key encoded in base64 format.

Although voter/trustee can create the key pair using external programs, we also provide the key generation button which creates the key pair locally. Users can disconnect from the network during key generation to prove that our function will not send the private key out.

### **Election home page**

Figure 4-23 shows the home page of an election after freezing, which contains all information related to the election and links to the Voting page and Voter & ballot list. Notice that the link to the Voting page (green button) will only be enabled after the election

starts and before it ends.

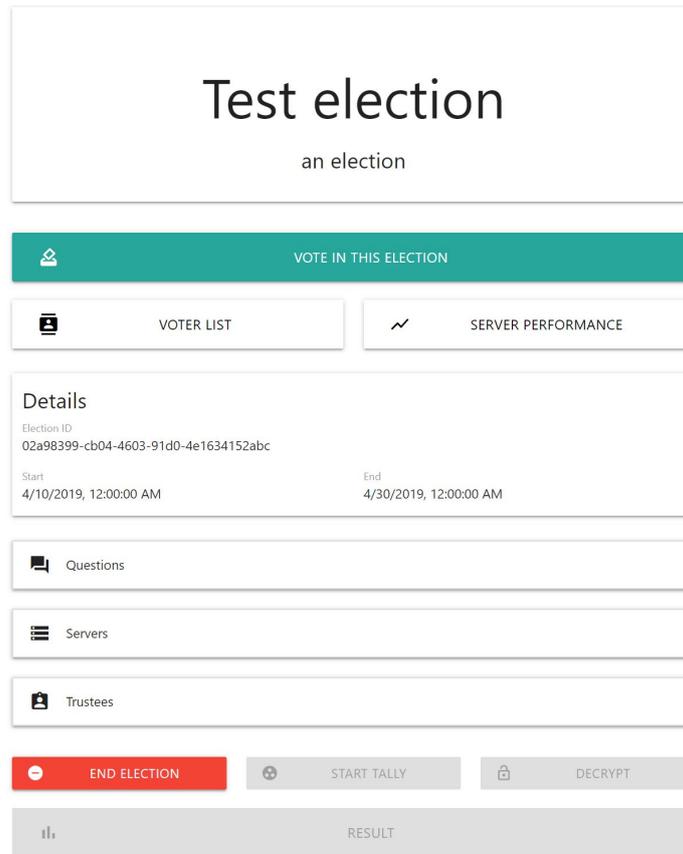


Figure 4-23: View of an election home page

At the bottom, it contains three action buttons for election administrator to use, from ending, tallying, to decrypting the election. These buttons will only be enabled when that action is valid to perform. For example, by clicking the “end election” button, it will pop up a modal like Figure 4-16 for election administrator to sign on the action, then the election will end before the preset ending time, so the button is enabled only before election ends.

Similarly, the “result” button that links to the Election result page is enabled when the final result is computed successfully.

### Voting page

All voters need to prepare their ballot via this voting page (Figure 4-24). It is very simple to perform voting, just select the choice and input voter ID and private key. After clicking “encrypt ballot” button, the browser will clear the choices as well as the private key, and perform ballot encryption (as stated in Section 3.2.1 - Ballot preparation) together

with creating proof (as stated in Section 3.2.2 - Voter honest encryption). Then, it will show a new box containing the encrypted ballot for the voter to review (Figure 4-25).

Vote.

Election details

Name	Description
Test election	an election
Start	End
4/10/2019, 12:00:00 AM	4/30/2019, 12:00:00 AM

Q1: Yes or No

Choose exactly 1 answer(s)

Yes  No

Voter info

Voter ID

05d6a5b1-37ca-4bf9-aec0-2ea8fec0756c

Private key (for signing only, wont send to server)

ENCRYPT BALLOT

Figure 4-24: The user interface for the voting page

Encrypt ballot

Encrypted answers

```
[{"choices":{"c1":"BviQB11NblcAbvni9Avu5EK/CsiE24CFnWtSeYPS+nk=","c2":"nov4xjH0JGX84IYnBQ...
```

Signature

```
YR3ozKG/UEkEKkca/9tOyUe6kpv5t0jkVeWAG8Bm2iG5GE5Zfv/uxAawRHcKRnHcd8wzSoAyXkOlnQV330tBNIBfPRGUgXfx4lz5+X0kmcq0F85Tk1vuC0U+Cv1EtfL23v+c46MPDK7B83dKjRqYqvmxlg5wHr5vncUWJO/8qoE=
```

SUBMIT BALLOT

Figure 4-25: Review the encrypted ballot before submission

After clicking the “submit ballot” button, the ballot will send to the server, and the browser will redirect to Voter & ballot list if success.

### Voter & ballot list

This will list out all voters with the signature of the last ballot they have submitted (Figure 4-26). Notice that if the submitted ballot is invalid or not yet be put into the

blockchain, then it should not appear on the list, so a voter may need to wait a while to see the list changes after ballot submission.



Figure 4-26: View all voters with their ballot signature

This list is important for end-to-end verification as stated in Section 1.2.1. it is because voters can verify that their ballot is recorded properly on the blockchain by comparing the ballot signature on the list and the one they submitted. More importantly, voters can notice via this list if their private key has been stolen by others to perform voting.

### Start tally an election

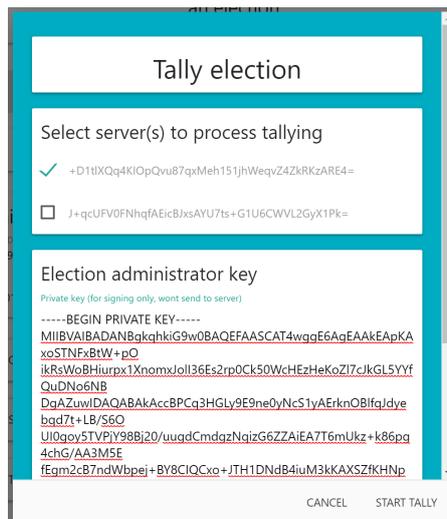


Figure 4-27: Modal for starting tally

After clicking the “start tally” button in the Election home page, a modal will be popped up (Figure 4-27) for election administrator to select servers for the tallying process, in order to achieve batch tallying as described in Section 3.2.3 - Slow tallying.

### Decrypt an election

Election administrator can start the decryption process by clicking on the “decrypt”

button in the Election home page, then it will show a modal similar to Figure 4-16 asking for a private key to sign on the action.

The server will send an email (Figure 4-28) to one of the trustees requesting he/she to partially decrypt the tally. The email consists of a link to direct the trustee to the decryption page (Figure 4-29). Trustees need to input the trustee ID and the public key, as well as the private key for signing purpose.

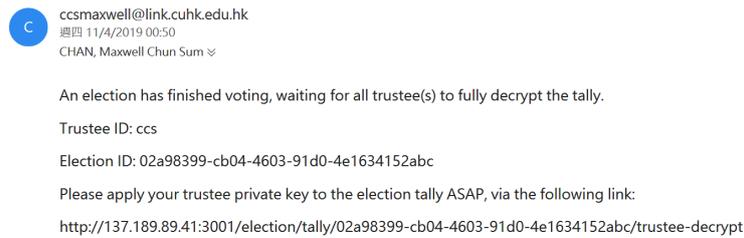


Figure 4-28: An example email sent to a trustee requesting decryption

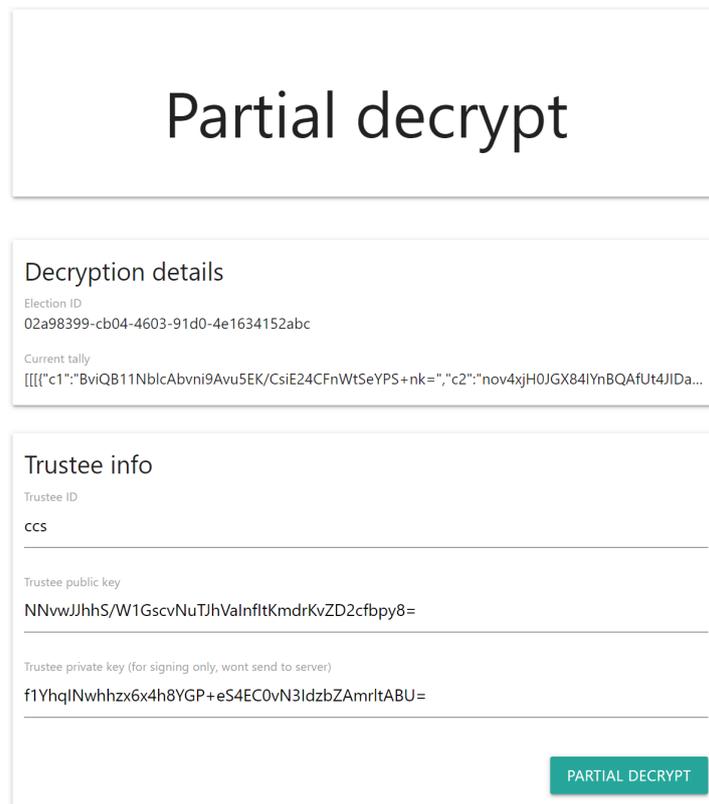


Figure 4-29: The user interface of the partial decryption page

After that, the browser will compute the partially decrypted tally (as stated in Section 3.2.1 - Tallying election) and the proof (as stated in Section 3.2.2 - Trustee honest decryption), show them on a box (Figure 4-30) for the trustee to review before submitting.

The server will repeat the process for all trustees by sending the email one by one until all trustees have submitted their decryption. Then, the final result will be stored into the blockchain.



Figure 4-30: Review the partially decrypted tally before submission

### Election result page

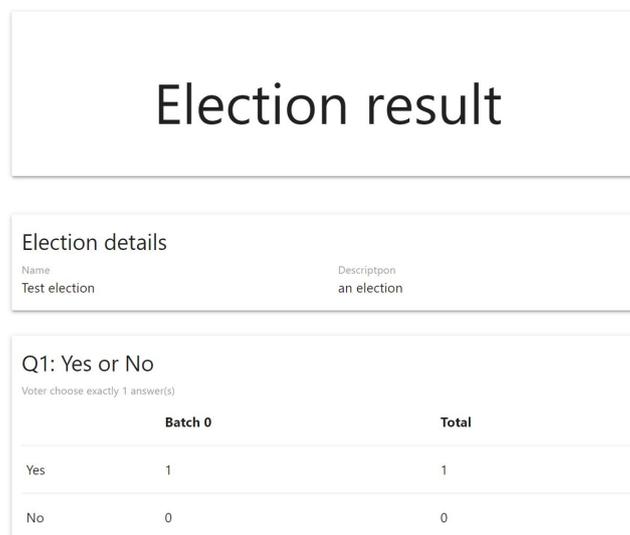


Figure 4-31: Viewing an election result

Figure 4-31 shows the page for users to view the final result of an election, which is got from the last block of the blockchain. The result is shown first by questions, then by options, and finally by ballot batches that divided in the tallying phase.

# 5. Testing

## 5.1. Overview

Other than functional testing which will be similar to the description of Section 4.3.3, we have done another two areas of testing, namely load testing and reliability testing. It is because one of our objectives of the application is to support an election of any scale. Of course, the actual performance will depend on how powerful the machine is. Therefore, the aim of the testing is to eliminate the bottlenecks that are related to programming or system design as much as possible, such that the system performance is able to scale up when using a more powerful machine for a larger scale election.

For load testing, it is divided into three parts:

- Block length testing, it aims to observe how system performance vary when the size of blockchain increase. So, the ballot generator will generate lots of ballots for the same voter, and the blockchain network will only have 1 node.
- Arrival rate testing, this is to test the maximum system limit on a constant arrival rate of ballots, and how system perform when the arrival rate is larger than the maximum. The testing time and the number of voters will keep constant at 20 minutes and 1 voter respectively.
- Ballot aggregation testing, it aims at measuring the time required to do the ballot aggregation as described in Section 3.2.3 - Slow tallying. The number of voters will be equal to the number of ballots we are testing on.

For reliability testing, we mainly test the system reaction when some nodes in the blockchain network go offline, which is to simulate the situation when the node is under attack and not able to respond to other nodes' request.

Notice that the machine we obtained for testing includes 3 "Local machines" inside the CUHK CSE network and 1 "Google Virtual Machine" on the Google Cloud Platform. Each "Local machine" has 4 CPU@2.8GHz and 8GB memory. While the "Google Virtual Machine"

has 8 CPU@2.5GHz and 56GB memory<sup>12</sup>.

We will mainly include 3 different types of graph to show the testing result, namely Cumulative Ballot Arrival (CBA), Ballot Latency per minute (BL) and Ballot Throughput per minute (BT). For CBA, the red line represents the time when ballot generated from the ballot generator, the yellow line represents the time when the first node receives the ballot, the green line represents the time when the ballot is put into blockchain. Usually, the green line should be in 'step form' as a block is generated every constant amount of time (15 seconds in our case). BL would be calculated as the time difference between the time ballot is generated and the time it is put into blockchain. BT is the amount of ballot put into blockchain in the time period (1 minute).

## 5.2. Load testing – First round

### 5.2.1. Block length test



Figure 5-1: CBA graph for 100, 1000, 10000 and 100000 ballots (left to right, top to bottom)

First, we have tried 100, 1000, 10000 and 100000 ballots (Figure 5-1) on Local machine. The system works fine for all of these. Although the ballot latency is gradually increasing for the case of 100000 ballots (Figure 5-2), it is still within the limit of 15 seconds,

<sup>12</sup> On “Google Virtual Machine”, 5% of CPU and 1GB of memory are constantly allocated for other background tasks.

which should be considered normal. Besides, in the BT graph (Figure 5-2), there is some fluctuation, which may be because the system is busying with some other tasks and delayed the block generation.



Figure 5-2: BL (left) and BT (right) graph for 100000 ballots

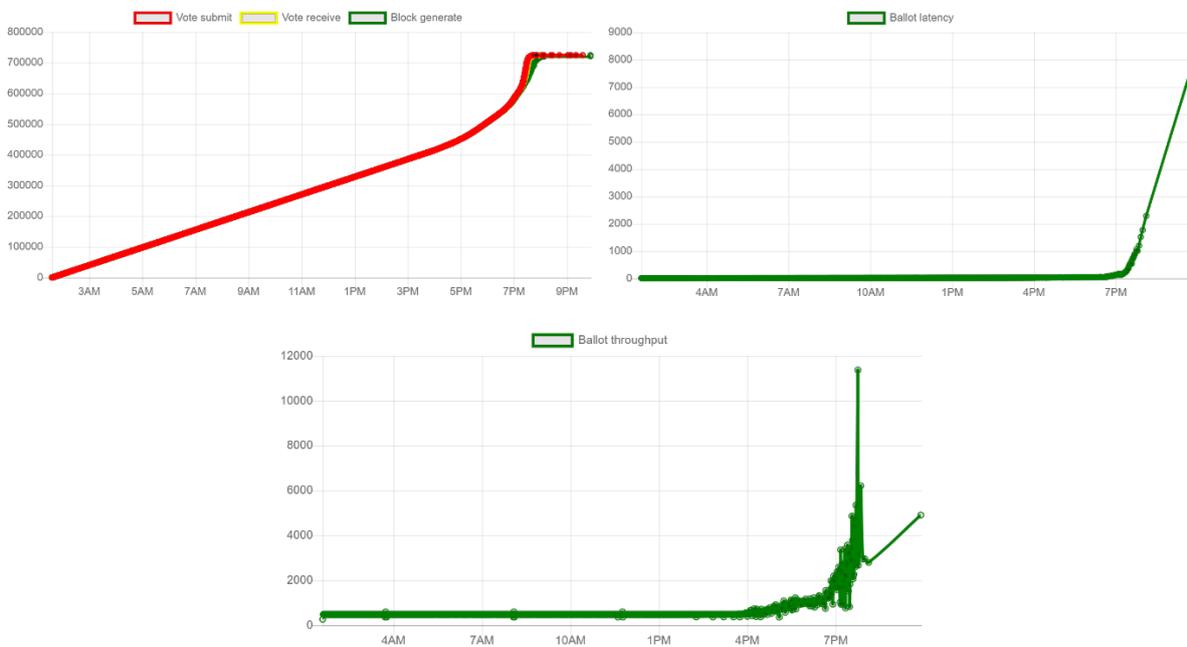


Figure 5-3: CBA (top-left), BL (top-right) and BT (bottom) graph for 1000000 ballots

However, when we want to test on 1000000 ballots, the application crashed after running for 20 hours (Figure 5-3). When we looked into the process viewer of the operating system, we observed that the MongoDB service uses a lot of memory and almost not responding, so that it used up the resource in Node.js when more ballot comes in and waiting for processing. Moreover, we noticed that the red line of the CBA graph is not constantly increasing after around 400000 ballots. This is an indication that some ballots are included in more than one blocks. It is because when ballot latency is longer than the double of block generation time (Figure 5-4), which is 30 seconds, the application did not

have enough time to mark the ballot as 'inBlock' before generating another block.

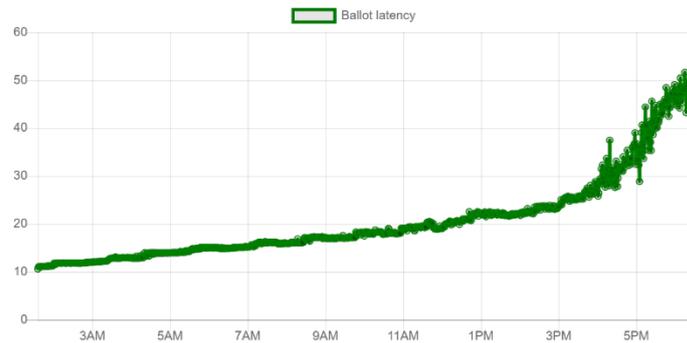


Figure 5-4: Zoom-in of BL graph for 1000000 ballots

We have also tried to cut the last few blocks and start the system again, to see if there is any possible memory leak. However, the system crashed again due to the same reason, and the ballot latency goes up very fast (Figure 5-5).

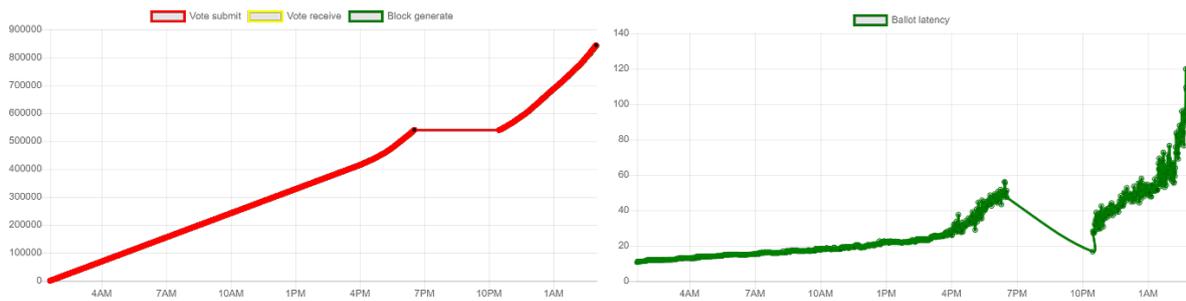


Figure 5-5: CBA (left) and BL (right) graph for the extended test of 1000000 ballots

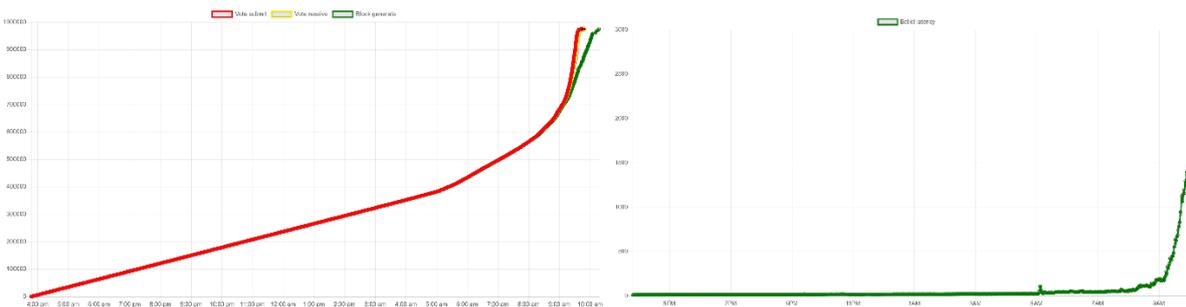


Figure 5-6: CBA (left) and BL (right) graph of 1000000 ballots test on Google VM

Since we thought that the problem may be on the memory, so we performed the same test on "Google Virtual Machine". Yet, a similar problem arises (Figure 5-6). While this time the Node.js application didn't totally crash and still sometimes responding to request, but the MongoDB is still not responding. According to the CPU usage graph (Figure

5-7), we thought that the reason may be the processing time increase when the block length increase, then it crashes up to a point that incoming request more than outgoing response.



Figure 5-7: CPU usage graph of 1000000 ballots test on Google VM

## 5.2.2. Arrival rate test

### 1 node

On the testing at “Local machine”, first we have tested with 10 and 11 ballots in a second. The system works well, and the latency is within the limit (Figure 5-8).

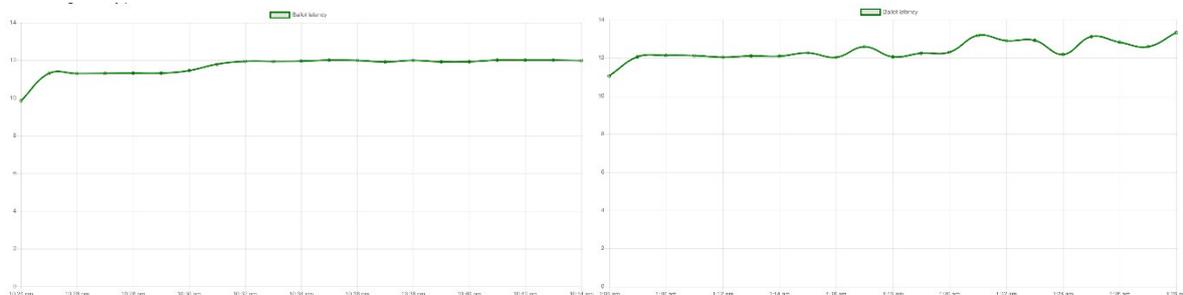


Figure 5-8: BL graph for the test of 10 (left) and 11 (right) ballots in a second

Nonetheless, when the arrival rate is 12 ballots per second, the latency increases a lot (Figure 5-9). From the CBA graph, we can also see that the lines started to divert, and the ‘step’ on the green line is not evenly distributed. This indicated that the system is too busy, and the queue length started to increase, so that it cannot perform the task of block generation on time. If we further increase the arrival rate to 14 ballots per second (Figure 5-10), then the latency just further increases up to 200 seconds.

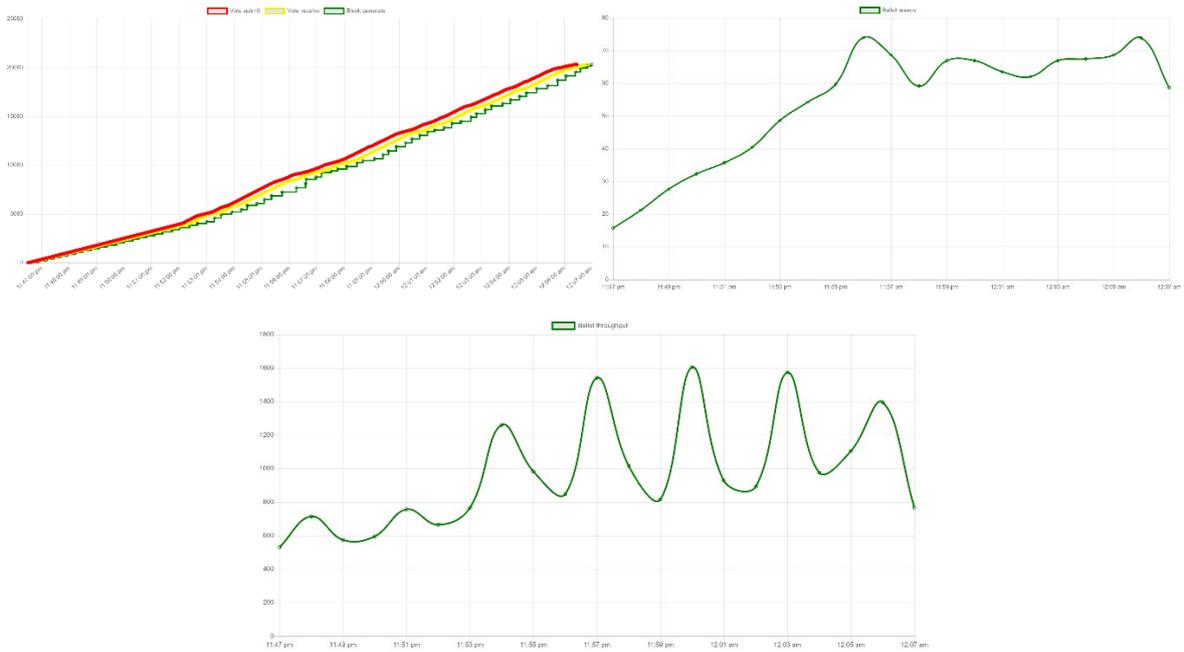


Figure 5-9: CBA (top-left), BL (top-right) and BT (bottom) graph for 12 ballots per second

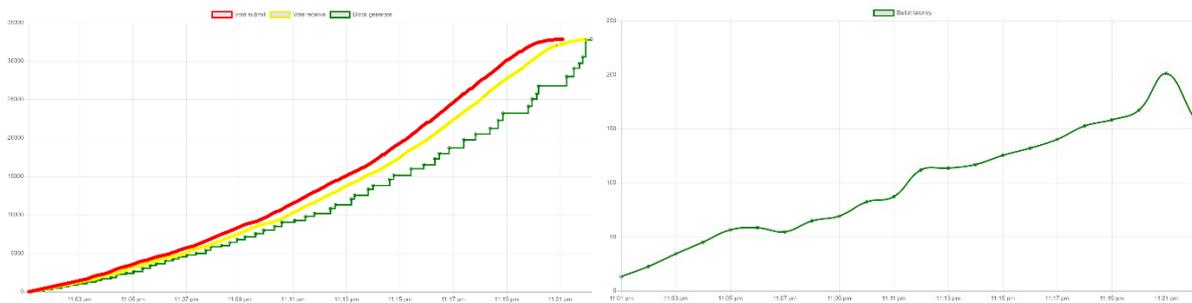


Figure 5-10: CBA (left) and BL (right) graph for 14 ballots per second

We have also tried to perform the 12 ballots per second test on the “Google Virtual Machine”, which has a more powerful CPU, but the result is similar. This shows that the application itself didn’t well schedule and use the CPU.

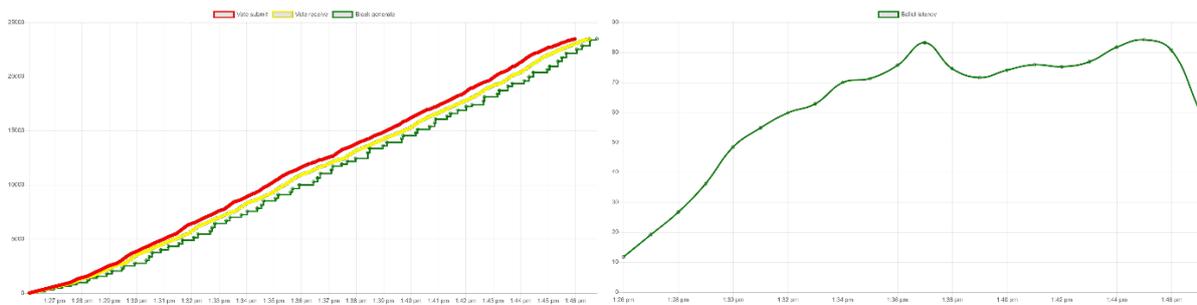


Figure 5-11: CBA (left) and BL (right) graph for 12 ballots per second test on Google VM

## 2 nodes

We run the 2 nodes on 2 separate “Local machines”. For the test of 6, 8, 9 and 10 ballots per second, the system and the blockchain network work well (Figure 5-12).

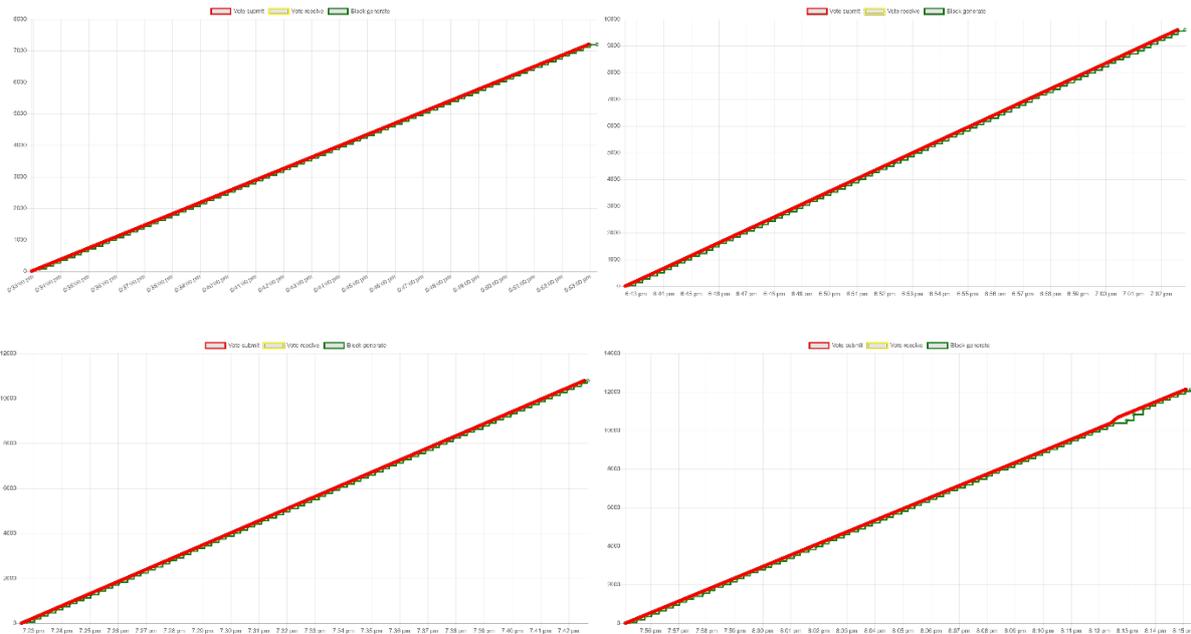


Figure 5-12: CBA graph for 6, 8, 9 and 10 ballots per second (left to right, top to bottom)

During the testing of 11 ballots per second, one of the 2 nodes starts to response slowly, thus their signature of ballot and block cannot arrive on time, causing a fork in the blockchain. As we require more than half of the nodes to agree on ballots and blocks, which is 2 in this case, the blockchain cannot grow from that point on (Figure 5-13).

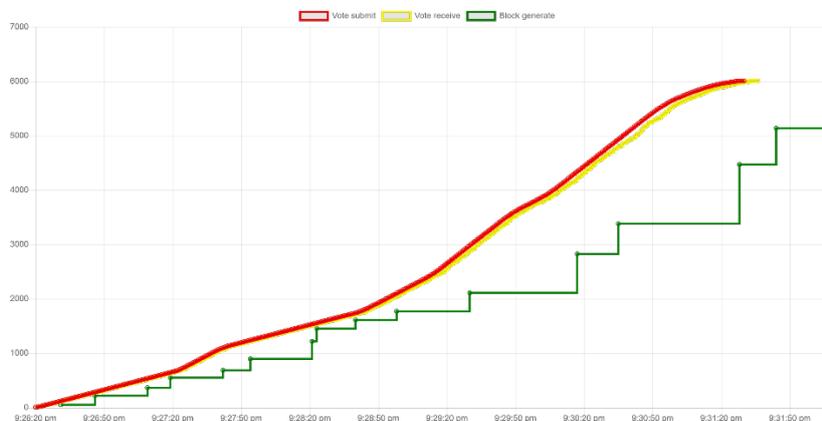


Figure 5-13: CBA graph for 11 ballots per second testing, when the blockchain not yet fork

This observation is as we expected. It's because more nodes in our blockchain network do not mean distributed processing. Instead, every node will process the same number of ballots, blocks and signature. When compare to the case of 1 node, where the maximum arrival rate is 11 ballots per second, here we get 10 ballots per second for the case of 2 nodes just because every node needs to process extra signatures.

### 3 nodes

Here we run 3 nodes on the same physical “Local machine”, as we need to reserve another “Local machine” as a ballot generator. So, the result should be only comparable to the case of 5 nodes.

First, we have tested 2, 4, 6 and 8 ballots per second, and the blockchain network run as expected (Figure 5-14). Notice that in the case of 8 ballots per second, there are occasions that the blocks are not generated on time, it is because 1 of the 3 nodes has become non-responsive. But the blockchain still grows because here we only require 2 out of 3 nodes work well.



Figure 5-14: CBA graph for 2, 4, 6 and 8 ballots per second (left to right, top to bottom)

When it comes to 9 ballots per second (Figure 5-15), very soon that 2 out of 3 nodes got non-responsive, thus the blockchain cannot grow anymore. Therefore, we can conclude

that 8 ballots per second are the maximum for this test case.

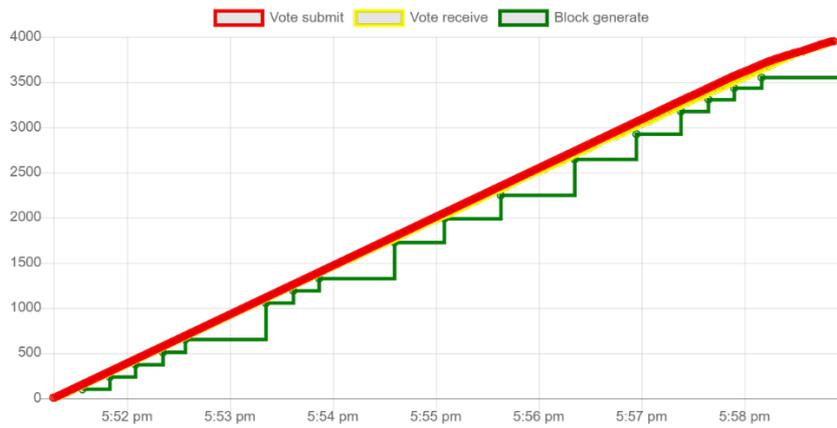


Figure 5-15: CBA graph for testing of 9 ballots per second, when the blockchain not yet fork

### 5 nodes

Here we run 3 nodes and 2 nodes on 2 separated “Local machine” respectively. For the test of 6 ballots per second, it works well. But for the test of 8 ballots per second, more than half of nodes start to respond request slowly after 10 minutes of testing, causing the blockchain to fork and refuse to grow. (Figure 5-16)

Therefore, the maximum ballot arrival rate for this test case is around 6 to 7, which is comparable to the case of 3 nodes, as this time every node needs to process more signatures.

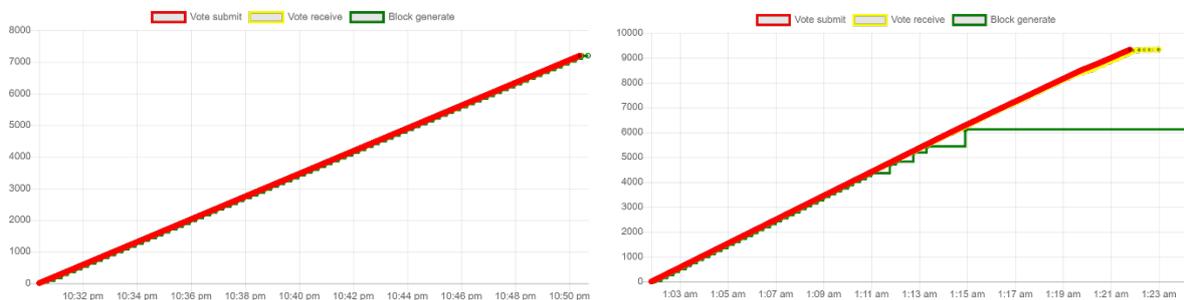


Figure 5-16: CBA graph for 6 (left) and 8 (right) ballots per second

### 5.2.3. Ballot aggregation test

On the test of 100 and 1000 ballots, if we distribute the aggregation assignment into 3 nodes that running on the same physical “Local machine”, then it requires 0.07 and 0.6 seconds respectively. (Figure 5-17)

If there are 10000 ballots, then distributing into 3 nodes requires 72 seconds, while distributing to only 1 node requires 201 seconds.

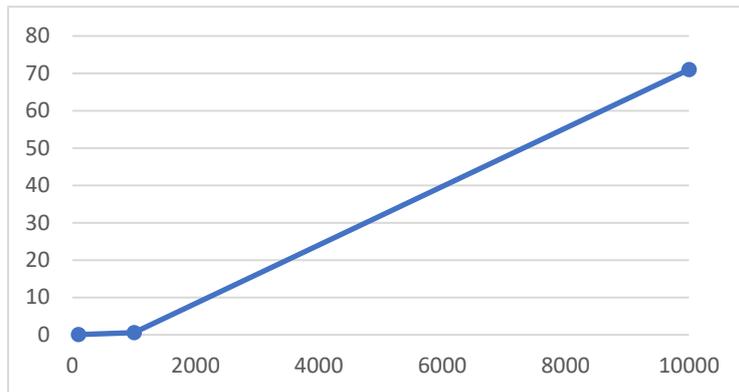


Figure 5-17: Graph showing no. of ballots versus aggregation time

We can see that the time required for aggregation is not linearly increasing with respect to the number of ballots. This may be because of the ballots are not divided by ‘cutting’ the blockchain, as stated in Section 3.2.3 - Slow tallying. Hence, every node needs to scan the whole blockchain and capture the related ballots, and this causes the bottleneck.

#### 5.2.4. Conclusion

While in this round of testing the system function most of the time, it clearly shows that the performance of our application cannot scale up with respect to computational power. Therefore, we need to find out the bottleneck and make improvements on system design.

The first problem we observed is that the application cannot well use the CPU resource. It’s because Node.js is a single threaded language [29]. This design is good to prevent race conditions from happening, but it limits the system throughput. That is why the ballot arrival rate will affect the on-time performance of block generation. Therefore, we will need to make some functions to run in parallel by creating more threads or

processes, and also take race condition into account.

Another problem we noticed is MongoDB uses lots of memory and affected its response time, especially when the blockchain is long. In fact, a block in the blockchain is just a normal document in a collection, MongoDB does not have a special algorithm to do searching in a blockchain. However, in reality, most of the time we only need to care about the last few blocks in the blockchain to generate a new block, or first few blocks to look for election details. Thus, we will need to find a way to reduce MongoDB meaningless searching on other blocks, in order to reduce the processing time.

### **5.2.5. System modification**

#### **Forking child process**

Within the 4 modules (controllers) shown in Section 4.2.2 (Figure 4-1), 'Ballot' is the busiest one. As 'Election' usually triggered before or after an election, 'Blockchain' usually triggered in a preset time interval, and 'Handshake' usually triggered when a node starts and ping request. Hence, we decided to make the Ballot controller as a child process.

When a node (master process) starts, it will automatically fork several Ballot child processes. By default, the number of processes is equal to the number of CPU minus 1, leaving the remaining CPU to handle the master process. Every time when the master process receives a request related to 'Ballot', it will pass it to one of the Ballot child processes in a round robin manner.

The only race condition that we need to take care of is, sometimes the signature of a ballot will arrive before the ballot itself. Previously, we just cache the signature in memory and check that cache every time the node receives a ballot. This cannot be implemented on the child process design because the ballot and its signature may be distributed to a different child process. So, we temporary save the signature in the Ballot database collection if the ballot not yet arrived. Later when the ballot arrives, we use 'upsert', i.e. update and insert, to atomically save the ballot and retrieve the signature for further verification.

## Database indexing

Currently, the database only index on the unique field. For example, in the Block collection, the only index is electionID and blockUUID (refer to Section 4.2.2 - Block). But this is not enough when we query the database using other fields, such as block sequence number. Thus, we added the following two indexes to improve searching by using up some storage space.

- electionID: ascending; blockType: ascending; blockSeq: descending; data.voters.id: ascending
- electionID: ascending; blockSeq: descending

The first index is useful for getting the election details, as well as voters' public key, which stored in the first few blocks in the blockchain with block type as "Election Details". The second index is useful for querying the last block in the blockchain, when a node needs to generate or verify new blocks.

## 5.3. Load testing – Second round

### 5.3.1. Block length test

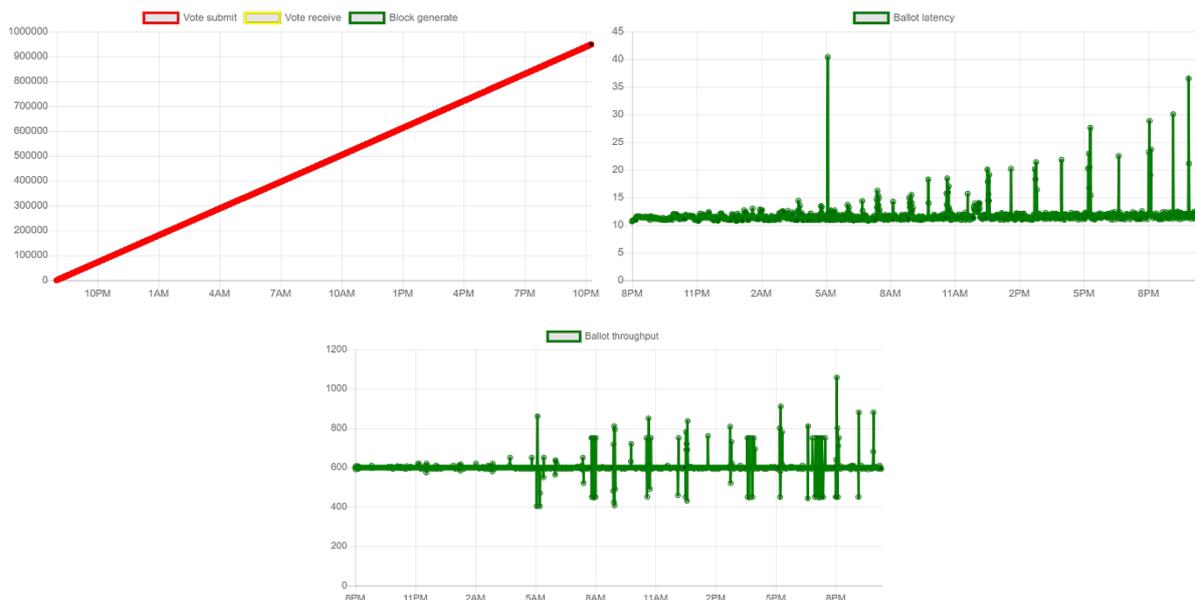


Figure 5-18: CBA (top-left), BL (top-right) and BT (bottom) graph for 1000000 ballots test

We tried the testing of 1000000 ballots again on “Google Virtual Machine” (Figure 5-18). Although this time the application still crashed at around 950000 ballots, the red line on the CBA graph is constantly increasing, and the ballot latency is mostly constant with only some spark occasionally. This shows that the processing time on Node.js and MongoDB did not increase with block length. It is a significant improvement when compared to Figure 5-6. Furthermore, the CPU usage graph (Figure 5-19) shows that the CPU usage did not increase much throughout the experiment, which is also a big difference when compared to Figure 5-7.



*Figure 5-19: CPU usage graph of 1000000 ballots test*

When investigating using the system process and resource viewer, we found that the reason for application crash was Node.js went out of memory. At 3 hours before the crash, our master process uses 16.9GB of memory, while each child process only uses 100MB of memory, and MongoDB only uses 5GB of memory. This indicates that there may be a memory leak on the master process. This can also explain why there are higher and higher sparks in the BL graph in Figure 5-18. It is because Node.js was performing garbage collection, which is to free memory that no longer reachable. When memory usage increase, garbage collection time also increase as it needs to scan the memory, thus causing latency in block generation.

In order to prove our hypothesis, we restart our application for another 2 hours without restarting MongoDB (Figure 5-20). We can see that it can go for 1000000 ballots and even more. Also, there is no spark in the ballot latency. Thus, our hypothesis is proven.

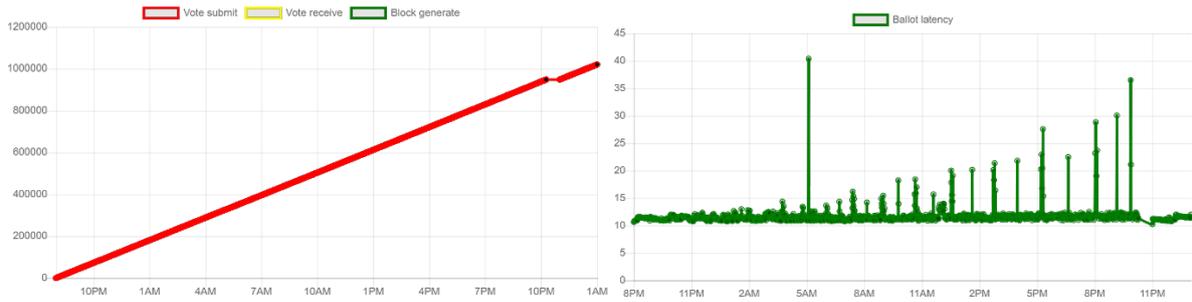


Figure 5-20: CBA (left) and BL (right) graph for the extended test of 1000000 ballots

### 5.3.2. Arrival rate test

#### 1 node

First, we performed testing on “Local machine”. It works well with 12, 16, 24 and 30 ballots per second as we can see the ballot latency is almost constant (Figure 5-21).

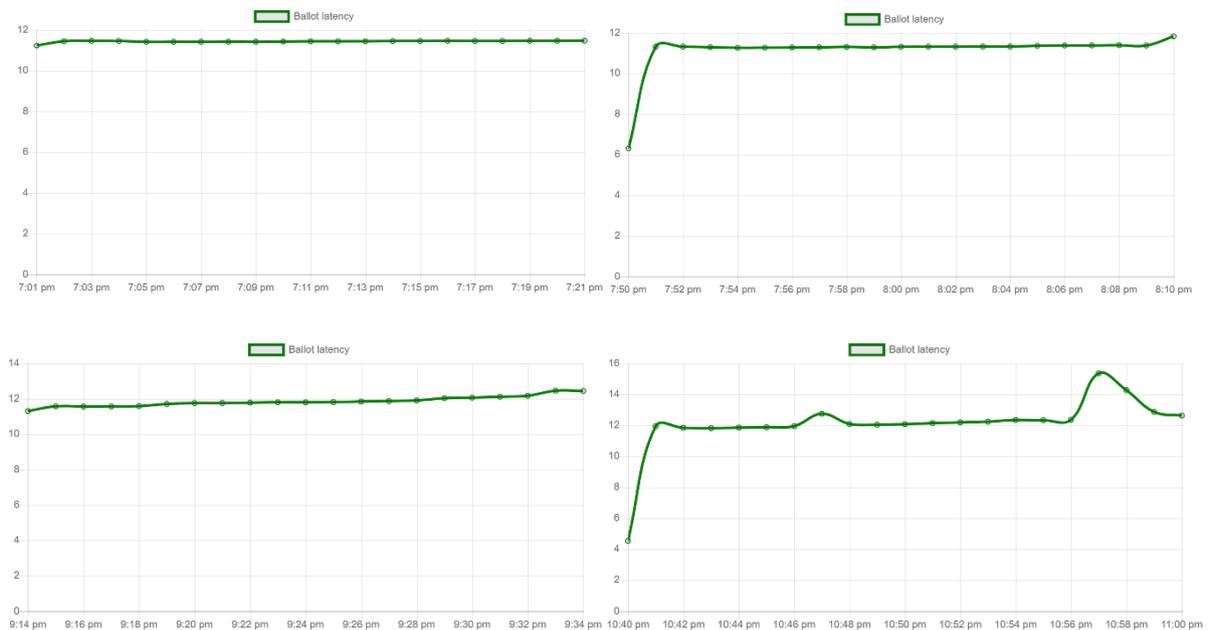


Figure 5-21: BL graph for testing 12, 16, 24 and 30 ballots per second (left to right, top to bottom)

For the test of 32 ballots per second, although the BL graph shows that the latency is constantly increasing (Figure 5-22), the lines in CBA graph only slightly diverted, and the ‘steps’ on the green line is evenly distributed. This shows that even if the arrival rate is over

the maximum, the system still behaves as expected, which is a significant improvement when compared to Figure 5-9 and Figure 5-10.

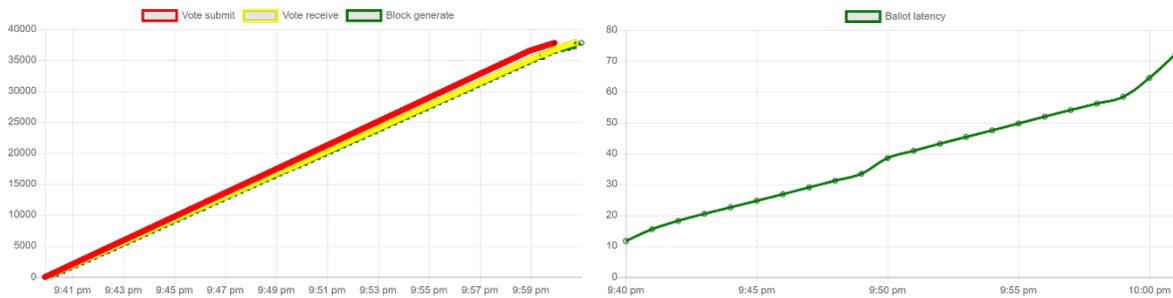


Figure 5-22: CBA (left) and BL (right) graph for the test of 32 ballots per second

We have also performed the testing with 7 Ballot child process on a “Local machine”<sup>13</sup>. It turns out that this time the application can afford up to 36 ballots per seconds, and only went a bit diverted when testing with 40 ballots per seconds (Figure 5-23). We thought this may be because a single child process cannot use up the limit of one CPU, and the CPU requirement for the master process is lower than the child. Therefore, forking a bit more child process can consume CPU resource more efficiently.

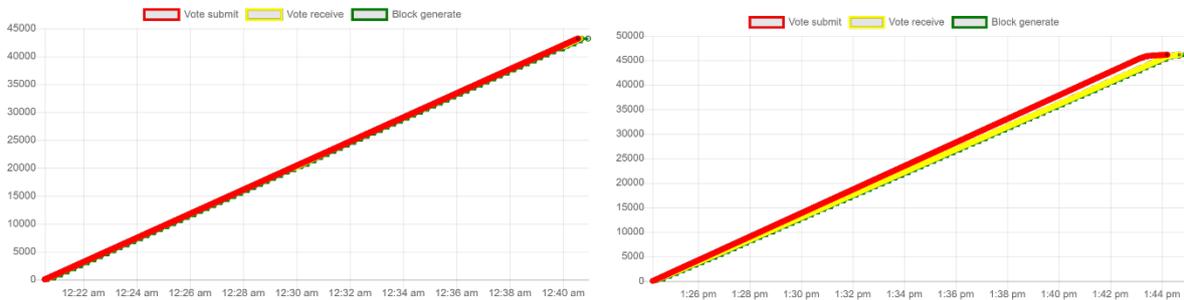


Figure 5-23: CBA graph for testing 36 (left) and 40 (right) ballots per second

On the other hand, we have done the testing on “Google Virtual Machine”. With the arrival rate of 30 or 40 ballots per second, it has no problem. When it comes to 48 ballots per second, the ballot latency only slightly goes over 15 seconds. (Figure 5-24) The ballot latency increases constantly if the arrival rate reaches 56 ballots per second (Figure 5-25). If we look at the CPU usage graph (Figure 5-26), we can see that 48 to 56 ballots per second are almost the system limit, as it uses up 80% to 90% of CPU in average.

<sup>13</sup> Default number of Ballot child process is 3 on “Local machine”

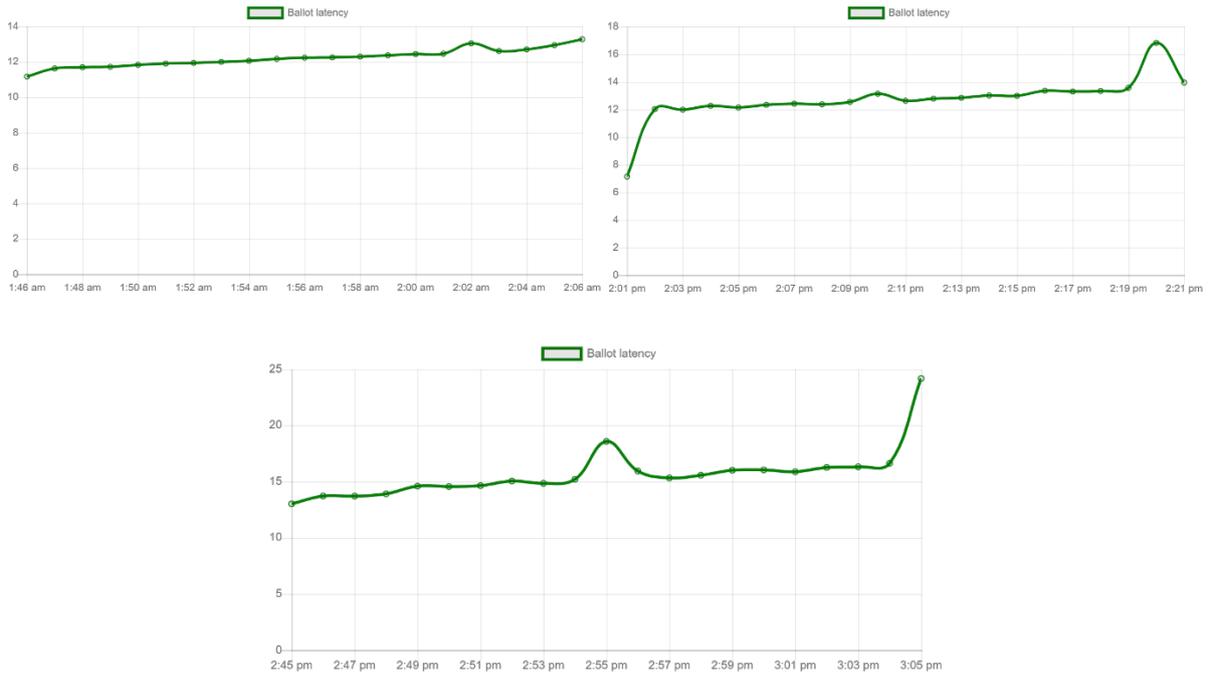


Figure 5-24: BL graph for testing 30 (top-left), 40 (top-right) and 48 (bottom) ballots per second on Google VM

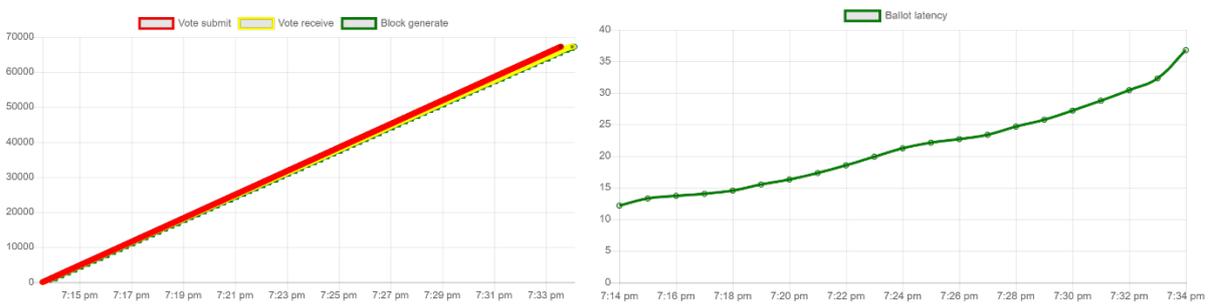


Figure 5-25: CBA (left) and BL (right) graph for testing 56 ballots per second on Google VM

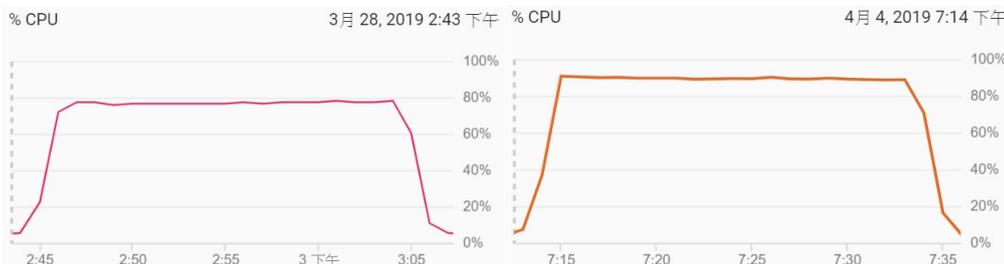


Figure 5-26: CPU usage graph for testing 48 (left) and 56 (right) ballots per second

**2 nodes**

Here we run the 2 nodes on 2 separate “Local machine”. The nodes and blockchain network behave as expected for an arrival rate of 20 and 28 ballots per second (Figure 5-27). When it comes to 30 ballots per second (Figure 5-28), although the ballot latency is increasing, blocks still generated and verified in a regular interval. This is improved when compared to Figure 5-13.

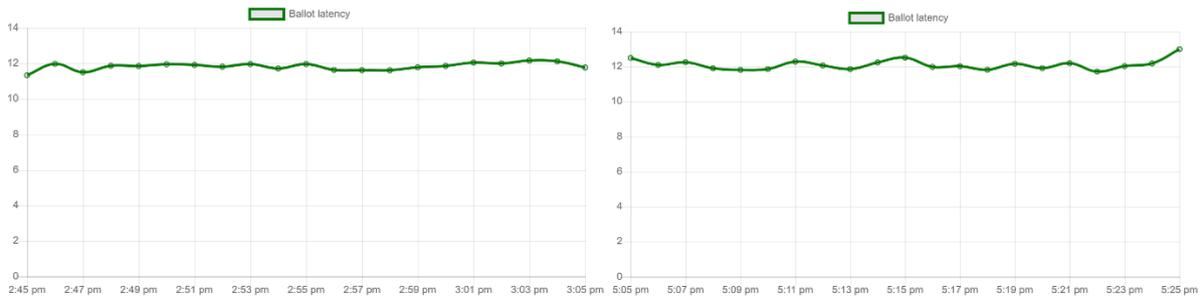


Figure 5-27: BL graph for testing 20 (left) and 28 (right) ballots per second on 2 nodes

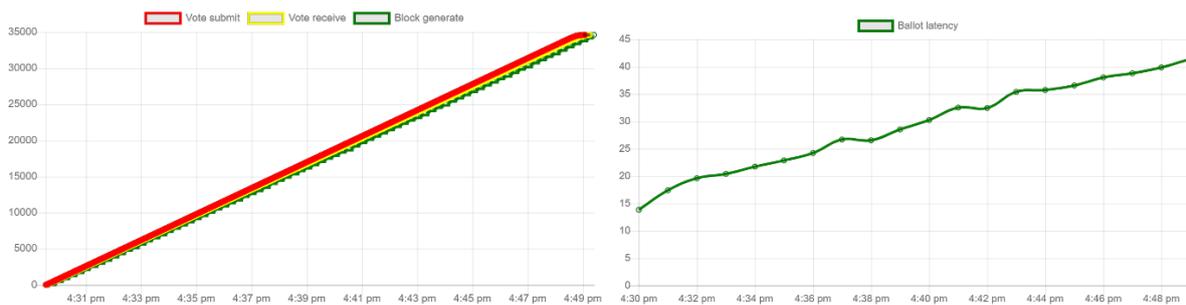


Figure 5-28: CBA (left) and BL (right) graph for testing 30 ballots per second on 2 nodes

The result is also as expected, which is the arrival rate slightly lower than the 1 node’s case, as this time each node and its child processes need to process more signature.

### 5.3.3. Ballot aggregation test

Different from the test on Section 5.2.3, 3 tests were run with 5000, 10000 and 20000 ballots respectively. For each test, we have tried to do aggregation on 1 or 2 nodes. Although the graph (Figure 5-29) showing that the relationship between aggregation time and the number of ballots is still not linear, it has a great improvement when compared with Figure 5-17, as the time required for 1 node aggregating 10000 ballots is only 26 seconds.

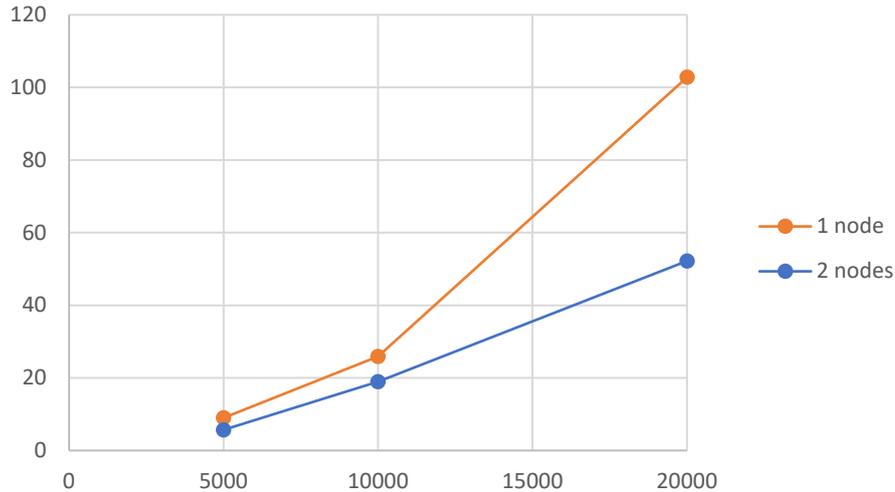


Figure 5-29: Ballots aggregation time with a different number of ballots/nodes

### 5.3.4. Conclusion

It is obvious that after the modification stated in Section 5.2.5, the system's performance enhanced a lot. Especially for the arrival rate test, it shows that the application is able to scale up with respect to the computational power<sup>14</sup>, which achieved our objective of the project. Besides, the ballot aggregation test gives reasonable aggregation time, which indicates the possibilities of decrypting a large number of ballots.

The main problem we discovered during this testing is the memory leak problem in the block length test, so that we will need to figure out where in our code causing the leak. But other than this, the block length test shows that the ballot latency almost did not increase with the block length, this implies the system's scalability is high.

### 5.3.5. System modification

In order to find out the memory leak, we have made use of the 'inspect' tool provided by Node.js, which allow us to record the data stored in memory and see how they are allocated and freed.

---

<sup>14</sup> "Google Virtual Machine" is around 1.7x the "Local machine" in specification, while the result from "Google Virtual Machine" is around 1.4-1.5x the "Local machine".

We noticed that every time during the block generation process, there are around 1MB of memory allocated but not freed after a long time. That piece of memory was allocated by the node-cache package. With further investigation, we found that the problem may be due to the node-cache package did some deep copy of the cached object but somehow did not free that when deleting the cache.

Previously, we cache the ballots at the node selection stage using node-cache package, and it will be used and deleted in the block generation stage. Now, we directly save the ballots in the memory without using cache, and delete that piece of memory after the block is generated.

After the modification, we inspect the application again and found that almost no allocated memory will not be freed after the block generation, so we believe the memory leakage problem has been solved.

## 5.4. Reliability testing

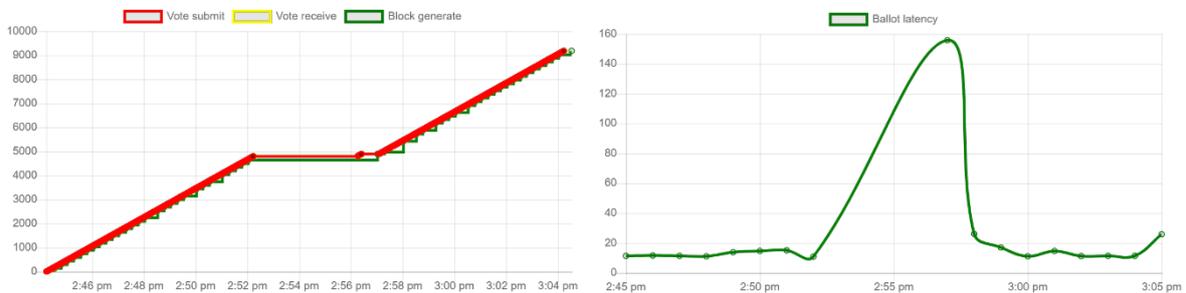


Figure 5-30: CBA (left) and BL (right) graph for the reliability test

The test is executed in a 3-nodes setting that running in a single physical “Local machine”. We generate ballots with an arrival rate of 10 per second to the first node for 20 minutes. To test the reliability, we manually shut down the third node on the 4<sup>th</sup> minute and the second node on the 8<sup>th</sup> minute. Then, we start the second node again on the 12<sup>th</sup> minute and the third node on the 16<sup>th</sup> minute.

From the CBA graph (Figure 5-30), we can see that the system looks good for the first four minutes. After the third node has been shut down, the green line occasionally did not grow. This is normal because according to the protocol stated in Section 3.3.7, when the

nodes agreed to select an offline node for block generation, then they will just skip it.

After the second node shut down on the 8<sup>th</sup> minute, all lines went flat because no ballot can get half nodes sign on it, so all ballots submitted at that time period will be ignored. When the second node came back at the 12<sup>th</sup> minute, although the system used some time to synchronize, it worked fine afterwards. It also behaved as normal when the third node came back.

If we look at the BL graph (Figure 5-30), it is also reasonable. The sudden jump at around the 12<sup>th</sup> minute is due to some ballots submitted just before the second node went offline, only put into the blockchain after the second node came back.

This test shows that the system is able to adapt to the situation when nodes go offline and online. So, when the server administrator aware that a node is not responding or under attack, then he/she can simply restart the node somewhere without affecting the whole system.

The only issue we observed here is that the synchronization time from 1 node to 2 nodes is a bit longer than expected, this may be due to the synchronization task is handled by the blockchain component (Figure 4-1) and it was busy. Therefore, we may use another child process to handle this.

## 6. Conclusion

### 6.1. Summary

To conclude for this term, we have done more than we planned in the objective. First, on the parts related to Helios, we have studied the mechanism of those zero-knowledge proofs. While the Helios' documentation about this part is not detailed enough, we still implemented it on our application successfully. On the other hand, we have also reviewed our proposals about authentication and list out more possibilities for different situations.

Secondly, on the blockchain part, we slightly adjusted the protocol we designed in the previous term, in order to better support functions like handshaking and tallying, as well as improve the stability of the network. Moreover, we have enforced all the verification necessary for communications between nodes. This includes the ballots, blocks, signatures and requests verification.

Thirdly, we designed the web-based user interface for our application, so that different roles of an election can make use of it from end to end. Besides, access control via private key has also be enforced for all functions in the backend.

Lastly, we have done quite a lot of testing to stress the system under different circumstances. Although the first-round test showed that the application could not perform as expected especially in the scalability, we figured out the bottleneck and upgraded the system. So, the second-round test's results are more satisfactory, as the performance of our application improved at least 4 times the previous round.

If this application is used for an election involving 8 million of voters, and assume that 10% of them will vote during the last hour, then the arrival rate requirement will be 223 ballots per second. This may be doable with some machines having very high computation power, but we have not tested this yet and the time for ballot aggregation may also be very long. Nonetheless, we believe that our application is sufficient for large scale election in Hong Kong like the Legislative Council Election. According to the data of the 2016 election [30], we only have around 4 million registered electors. Besides, the candidates are divided by 5 different geographical constituencies. Hence, it is possible to set up 5 groups of machines responsible

for different constituencies. Take the largest constituency, New Territories West, as an example, it has around 1 million registered electors. Let's assume again that 10% of the voters will vote in 1 hour, then the arrival rate requirement will be only 28 ballots per second. According to our testing result, the "Google Virtual Machine" will be sufficient and there is still room for having more nodes in the blockchain network.

We have already made the application opensource on GitHub<sup>15</sup>, as we believe that security cannot be enhanced through obscurity. By making it opensource, more developers can investigate the application and provide suggestions to improve it. Also, this is also a proof to others that our application is runnable, and also not malicious to do things like saving others' private key.

Voting system itself is not a hot topic in Hong Kong, not many people really concern about the current voting system on its anonymity and verifiability, even after the incident of losing the computer that contains all voter information [31]. Actually, we also learned about all these voting considerations only after we started working on this project. It is interesting to know that e-voting can be much more reliable than we thought. That's why our goal is to educate others about these things.

## 6.2. Future work

### 6.2.1. Improvement on scalability

#### Use more child processes

As stated in Section 5.2.5, currently we only make the 'Ballot' component separate from the master process. However, other components can sometimes become a bottleneck, such as the 'Blockchain' component. Therefore, we think that every component should become one or more child process. This not only to improve the performance of the node, but also make it more reliable since the failure of a component will not crash the entire application.

---

<sup>15</sup> Available on [https://github.com/ccsmaxwell/e2e\\_voting](https://github.com/ccsmaxwell/e2e_voting)

### **Possibility of partially broadcasting ballots**

System throughput cannot improve indefinitely just by having more child processes, because all requests still need to pass through the master process before arriving at the child process, so the queue at master process will be the bottleneck. To deal with this problem, we have thought of the possibility of not broadcasting ballots to all nodes, but only to just more than half of the nodes to get the required number of signatures.

However, this brings up another problem when generating blocks, as a node cannot put ballots that are not in its database into a new block. Therefore, we think the solution may be having more than one node, i.e. master process, connecting to the same database. All nodes responsible for the same database should have the same server ID, and only one of them will be responsible for block generation. There should be no communication required between these nodes, so that we can increase the overall throughput by having more this kind of nodes.

## **6.2.2. Improvement on reliability**

### **Ballot re-broadcasting**

As we can see from the test in Section 5.4, all ballots submitted during the time when less than half online nodes will be ignored. Although this did not affect the system performance, it worsened the experience of the voter, which may think that the system is not reliable. Hence, we think that when there is less than half of the nodes go online, these nodes should still store the ballot properly and re-broadcast it when the remaining nodes come back again.

### **Smarter blockchain synchronization protocol**

Currently, the blockchain synchronization process only triggered when a node joining the network. However, although happened very rarely, sometimes a node may not receive some signatures for a block, then the node may think that the block, which should be valid, should not be accepted. This will make the node deny all the following blocks, as it cannot verify the block hash and sequence number.

Therefore, we think that the blockchain synchronization process should also be triggered when a node received a sequence of 'invalid blocks', so that the node can self-synchronize automatically. When implementing this function, we should aware of a possible attack by a malicious node that sending a sequence of invalid blocks to all other nodes, causing a lot of synchronization tasks triggered at the same time and occupy many computer resources.

### **Synchronize clock on different nodes**

According to our block generation protocol in Section 3.3.7, a new block is generated in a regular time interval. So, the block generation time is the multiple of that interval plus the offset equal to the election freezing time stored in the blockchain. Currently, we assume that the clock on all the nodes is synchronized, so that they can use the local clock to calculate the block generation time.

But in reality, the local clock on different nodes may not be synchronized, this may lead to delay in block generation, or even cannot generate a new block if there exists a serious clock deviation between nodes. So, a clock synchronization function is needed to be activated when a node joins the blockchain network.

### **6.2.3. Full implementation of the proposed design**

In this term, we decided to focus more on the testing because we think that it is more important having the application available to support a larger election scale. So, mainly two proposed designs are not implemented in our application. One of them is 'kiosk voting' stated in the Coercion par.

Another one is some ways of performing authentication stated in the Authentication method part. We think more study should be done on this part before implementation. For example, the ways described for enrolling voters are based on using email as a communication channel, but ways involving other channels should also be considered, such as messaging services like Telegram and Whatsapp.

#### **6.2.4. Enforce more security measure**

For the ease of testing, we intentionally remove some security-related verification. One of them is related to the replay attack, which means a malicious person can cache a request from the user and send it again to the server sometime later, and the server will accept the request. To tackle this problem, every time before submitting a request to a server, the user needs to get a nonce from the server and attach to the request, so that the server will accept the request only if the nonce matches the one it generated and has not used by other requests.

Another one is about a secure connection between users and servers, i.e. using HTTPS, to tackle the man-in-the-middle attack by encrypting all communications from end to end. However, enforcing HTTPS will require a certificate and a domain name bound to a server, which makes the deployment of more servers not that convenient, so we did not implement this currently.

#### **6.2.5. Use newer communication protocols**

In our application, all communications between servers and communications between users and servers are using HTTP. It is because HTTP is easy to use and Node.js has many supports on it, so that we can generate or reply an HTTP request with simply a function call. However, HTTP is built on top of TCP<sup>16</sup>, which is relatively slow as it requires some handshakes for each individual request before sending data.

Recently, Google has created a new communication protocol called QUIC [32]. Although it is built on top of UDP<sup>17</sup>, QUIC also provides reliability and security. Most importantly, it only requires 1 round trip of handshake for a new connection and 0 for resuming previous connection. So, we should study the possibilities of applying QUIC into our application, so as to reduce the latency for all communications.

---

<sup>16</sup> Transmission Control Protocol

<sup>17</sup> User Datagram Protocol

### **6.2.6. Possibility of enabling “Voting-as-a-service”**

Although the application is opensource, we think that it is still possible to start a business as “Voting-as-a-service”. Basically, the idea is to charge the election organizer the amount of computation power used to handle requests related to that election. On the other hand, people can host a node and join the network, then they will get paid for the computation power they served. In this way, an election organizer can organize an election easily even if they do not know anything about installing the application.

# Bibliography

- [1] P. Tang, "Electronic voting," Legislative Council Secretariat, Hong Kong, 2017.
- [2] Digital Transaction Limited, "AIDB: Authoritative Information Distribution Blockchain," Hong Kong, 2018.
- [3] M. Hooper, "Top five blockchain benefits transforming your industry," IBM, 22 2 2018. [Online]. Available: <https://www.ibm.com/blogs/blockchain/2018/02/top-five-blockchain-benefits-transforming-your-industry/>. [Accessed 21 11 2018].
- [4] J. Benaloh, R. Rivest, P. Y. A. Ryan, P. Stark, V. Teague and P. Vora, "End-to-end verifiability," arXiv preprint arXiv:1504.03778, 2014.
- [5] T. Jenks, "Pros and Cons of Different Blockchain Consensus Protocols," Very, 8 3 2018. [Online]. Available: <https://www.verypossible.com/blog/pros-and-cons-of-different-blockchain-consensus-protocols>. [Accessed 21 11 2018].
- [6] G. Konstantopoulos, "Understanding Blockchain Fundamentals, Part 1: Byzantine Fault Tolerance," Medium, 1 12 2017. [Online]. Available: <https://medium.com/loom-network/understanding-blockchain-fundamentals-part-1-byzantine-fault-tolerance-245f46fe8419>. [Accessed 22 11 2018].
- [7] "6.20-29 Civil Referendum," PopVote, 16 2 2015. [Online]. Available: [https://popvote.hk/english/project/vote\\_622/](https://popvote.hk/english/project/vote_622/). [Accessed 21 11 2018].
- [8] J. Ma, W. Lee and R. Chung, "PopVote: A Revolution in Gathering Opinions in Hong Kong," RTHK, 12 8 2013. [Online]. Available: [http://rthk9.rthk.hk/mediadigest/20130812\\_76\\_123023.html](http://rthk9.rthk.hk/mediadigest/20130812_76_123023.html). [Accessed 21 11 2018].
- [9] Hong Kong Computer Emergency Response Team Coordination Centre, "Advices on the security concerns of the PopVote System," 10 2 2017. [Online]. Available: [https://www.hkcert.org/my\\_url/en/blog/17020901](https://www.hkcert.org/my_url/en/blog/17020901). [Accessed 21 11 2018].
- [10] B. Adida, "Helios: Web-based Open-Audit Voting," *USENIX security symposium*, vol. 17, pp. 335-348, 2008.
- [11] S. T. Ali and J. Murray, "An overview of end-to-end verifiable voting systems," in *Real-world electronic voting: Design, analysis and deployment*, 2016, pp. 171-218.
- [12] R. Peter Y. A., B. David, H. James, S. Steve and X. Zhe, "Prêt à Voter: a Voter-Verifiable Voting System," *IEEE transactions on information forensics and security*, vol. 4, no. 4, pp. 662-673, 2009.

- [13] S. Popoveniuc and B. Hosp, "An Introduction to PunchScan," *Towards trustworthy elections*, vol. 6000, pp. 242-259, 2010.
- [14] C. Meter, "Design of Distributed Voting Systems.," arXiv preprint arXiv:1702.02566, 2017.
- [15] N. Faour, "Transparent Voting Platform Based on Permissioned Blockchain.," arXiv preprint arXiv:1802.10134, 2018.
- [16] VoteCoin team, "Vote Coin: Anonymous Crypto Democracy," 2017.
- [17] Y. Liu and Q. Wang, "An e-voting protocol based on blockchain," IACR Cryptol. ePrint Arch., Santa Barbara, CA, USA, 2017.
- [18] F. Þ. Hjálmarsson and G. K. Hreiðarsson, "Blockchain-Based E-Voting System.," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018.
- [19] F. S. Hardwick, G. Apostolos, N. A. Raja and K. Markantonakis, "E-Voting with Blockchain: An E-Voting Protocol with Decentralisation and Voter Privacy.," arXiv preprint arXiv:1805.10258, 2018.
- [20] S. Panja and B. K. Roy, "A secure end-to-end verifiable e-voting system using zero knowledge based blockchain.," 2018.
- [21] O. Pereira, "Internet Voting with Helios," in *Real-World Electronic Voting: Design, Analysis and Deployment*, 2016.
- [22] R. Gupta, *Hands-on Cybersecurity with BlockChain : Implement DDoS Protection, PKI-Based Identity, 2FA, and DNS Security Using BlockChain*, Birmingham: Packt Publishing Ltd., 2018.
- [23] "51% Attack," Investopedia, [Online]. Available: <https://www.investopedia.com/terms/1/51-attack.asp>. [Accessed 21 11 2018].
- [24] "Hyperledger Fabric," Hyperledger, 2018. [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/latest/whatis.html>. [Accessed 21 11 2018].
- [25] S. Tilkov and S. Vinoski, "Node. js: Using JavaScript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80-83, 2010.
- [26] K. Chodorow, "MongoDB: The Definitive Guide: Powerful and Scalable Data Storage," O'Reilly Media, Inc., 2013.
- [27] D. Bressler, "4 Reasons a native mobile app can be more secure than a mobile browser based app," 8 3 2016. [Online]. Available: <http://davidbressler.com/2016/03/08/4-reasons-native-mobile-app-can-secure-mobile-browser-based-app/>. [Accessed 21 11 2018].
- [28] M. Kassner, "Apps vs. mobile websites: Which option offers users more privacy?," TechRepublic, 30 9 2016. [Online]. Available: <https://www.techrepublic.com/article/apps-vs-mobile-websites-which-option-offers->

users-more-privacy/. [Accessed 21 11 2018].

- [29] V. Tesanovic, "Multi threading and multiple process in Node.js," Itnext, 12 4 2018. [Online]. Available: <https://itnext.io/multi-threading-and-multi-process-in-node-js-ffa5bb5cde98>. [Accessed 7 4 2019].
- [30] "2016 Legislative Council Election," Registration and Electoral Office, 12 1 2017. [Online]. Available: [https://www.elections.gov.hk/legco2016/eng/tt\\_gc.html?1554980942176](https://www.elections.gov.hk/legco2016/eng/tt_gc.html?1554980942176). [Accessed 11 4 2019].
- [31] "Report of the Task Force on the Computer Theft Incident of the Registration and Electoral Office," Legislative Council Secretariat, Hong Kong, 2017.
- [32] "QUIC, a multiplexed stream transport over UDP," The Chromium Projects, [Online]. Available: <https://www.chromium.org/quic>. [Accessed 12 4 2019].