The Future of Democracy: Blockchain Voting

'Ethereum-based voting application proposal'

Matthew Flint

1247903 - mxf203@bham.ac.uk

Supervised by Dr. Ian Batten

This work was conducted as part of the requirements of the Module 06-26587 'Computer Science Project' of the Computer Science department at the University of Birmingham, UK, and is submitted in accordance with the regulations of the University's code of conduct.

13th April 2017

Contents

1 Abstract

An electronic voting protocol provides end-to-end verifiability if the voter can verify that their vote was correctly counted and any party can verify the results of the election. There have been several proposals outlining potential systems, however these have all been built on top of protocols primarily designed as transaction ledgers. In this paper I propose a voting solution, built on the Ethereum protocol, that uses the properties of smart contracts to enforce strict rules surrounding the ballots of an election. These ballots are both independently and universally verifiable and maintain all of the desirable properties of the blockchain (such as immutability). All of this is achieved without sacrificing voter privacy or ballot integrity. The resulting system shows clear potential for Blockchain technology to become a central part of applications wishing to provide transparency and security in public scenarios.

2 Introduction

Existing electronic voting systems all suffer from a serious design flaw; they are centralised by design, meaning there is a single supplier that controls the code base, the database and the system outputs while also supplying the monitoring tools to verify the result [1]. The lack of an independently verifiable system means that, once voters mark their ballot choice, they must place their trust in the organization that their vote is recorded and counted as intended. The lack of an independently verifiable output makes it difficult for these centralized systems to acquire the trustworthiness required by voters, which potentially limits voter participation, or cast doubt upon the published outcome of an election.

Despite the digitalisation of many aspects of modern life, elections are still being largely conducted offline, on paper [2], although the use of Electronic Voting Machines has been steadily growing over recent years. Paper ballots are still the traditional tool for voting and are typically marked by a human (voter) and then tallied by a machine. While costing less than most electronic systems to run, they rely on physical security and trust in polling stations to not manipulate and properly handle them [3]. Postal votes also utilize paper ballots and are used to allow voters to not have to physically attend a location in order to vote. These also suffer from the same flaws as traditional paper ballots while increasing the opportunities for attack during their passage through the postage system.

Voting systems comprise of five main components:

1. A registration service for verifying and registering legitimate voters.
2. Voter authentication stations with the task of determining a voters authorization to vote based on the completed work of the registration service.
3. Voting stations where the voter makes choices on a ballot.
4. A device called the ballot box where the ballot is collected.
5. A tallying service that counts the votes and announces the results.

Of the components listed above, only `(2)`, `(3)` and `(4)` are used during an election, with `(1)` being being required for an election to take place and `(5)` happening post election [4].

Current Electronic Voting (E-Voting) takes two forms; using a machine in a polling station, rather than a ballot paper and pencil, or casting a vote over the Internet. The former tends to refer to a Direct Recording Electronic system which typically displays ballot options on a screen that can be activated by the voter and then records that voting data in memory components to be processed later. However, as with many electronics, there is the inherent problem of the ability to modify software and potentially insert malicious code [3]. This has been an issue raised over several recent elections and a study from 2015 concluded that 43 American states would be using Electronic Voting Machines that are at least 10 years old during the last presidential election [5]. Voting over the Internet, while having to deal with issues such as privacy, fraud, voting under duress, and corruption, does nothing to improve a voter's trust. The voter must still assume that once they have cast their vote it will be recorded and counted honestly.

In order for our proposed E-Voting system to be a tangible challenger to more traditional voting methods, it must be able to provide the current systems' services, at least at the same level but preferably with improvements, while also providing substantial benefits. There are several standard requirements that a voting system should adhere to, each holding equal weight: security, functionality, privacy, usability, and accessibility [6]. A "secure" voting system means one that cannot be tampered with or manipulated in any way, ensuring that votes are accurately recorded as cast. It also ensures that additional votes cannot be cast after the polls have closed or tampered with at any stage of the process. System functionality can be broad but should include; The correct registering and recording of all votes cast, permitting a voter to vote for any candidate they have the right to vote for, allowing only eligible registered voters to vote and only allowing each voter to vote once. Voters have the right to a secret ballot and to cast their vote in private [6]. This is essential to protect voters from being coerced or bribed into voting a certain way, this means that our system should not provide a receipt or any way for another person to determine the contents of a voter's ballot. On top of this the system should be easy for voters to use, meaning it's as intuitive as possible, and maintain universal vote access. It should avoid introducing bias by selecting platforms that are less available to some groups than to others as the choice of the language, ballot format, or devices may seem innocuous, but it may actually prevent small factions of the voters to cast their vote [7].

Whilst maintaining these essential foundations already provided in traditional voting systems, there are several improvements and additions which I intend to explore. The first benefit in using a blockchain to log votes is in its decentralised nature. This means there is less need for trust to be placed in a centralised organization where votes are hidden behind closed doors. It also has the benefit of being significantly harder to tamper with as, once a transaction has been verified, an attacker would need to possess at least 51% of the computing power of the network to attempt to forge transactions. Any attempts to otherwise use a forged block will be noticed by the rest of the network and ignored. This decentralized system also brings in more transparency as anyone can view, transactions in the blockchain leading to higher levels of trust in the elections outcome. This is further strengthened by the independent verifiability which could be performed by anyone, therefore removing the need to trust the election organizers' declaration of the outcome. Once a transaction (vote) has been confirmed in the blockchain (and has further blocks built upon it) this vote for a candidate becomes immutable, meaning that the entire outcome of an election will be stored indefinitely, and is able to be accessed at any time in the future.

Verifiability properties of electronic voting are divided in two categories. "Individual verifiability", which involves auditing the processes of vote creation and vote storage by the voter; and "Universal verifiability", which ensures that only votes from eligible voters are stored in the ballot box and that all stored votes are properly tallied, which can be performed by anyone [8]. Systems providing both types of verifiability are known as "end-to-end verifiable systems".

One of the individually verifiable properties is cast-as-intended verification, which is focused on the audit of the vote creation process. Another property is recorded-as-cast verification, aimed at auditing the correct reception and storage of the vote in a remote voting server [8].

In this paper I outline the design of an "end-to-end verifiable system" built on the Ethereum Blockchain. In this system, a voter can register an Ethereum address which is then added to a list of *allowed addresses* inside a smart contract. Upon the ballot commencing, the voter will be able to modify the allocation of their vote inside this contract up to the point of the ballot ending. Due to the programmable nature of Ethereum contracts we can be sure that; no votes can be modified after the ballots end date, only the voter in control of the Ethereum address can change the vote associated with that address and only Ethereum addresses pre-registered to the contract are able to vote. Anyone can verify the number of votes for a candidate by querying the contract, and as they can be assured that only those addresses added to the contract are able to vote, we can be sure of the validity of each of these votes.

3 Background

3.1 Evaluation of Previous Research

There has been some research conducted in this area already and several protocols for blockchain based voting have been proposed. *Pierre Noizat* proposes a system [1] where each candidate provides a unique public key, KeyC, to each individual voter along with a singular bitcoin address, AddressC, which the final sum of the voting transactions will be sent to. Each voter is also assigned an individual public key, KeyA, by the election organizers and, either they generate a key pair themselves (this could be done by the voting software for better usability), KeyB, or be assigned the KeyB by the organization. From these three public keys, the voter can generate a `2-of-3` multi signature address which represents the vote of B in favour of C. This address is then funded with a bitcoin micropayment (around the price of a postage stamp), either by the voter or organisation, and this is the voters confirmation of their vote. After a few hours this ballot is securely logged on the blockchain and, as the multisignature address was funded, a voter can check that this address is represented in the blockchain and that their vote was registered. It should be noted that there is no way to guess neither the voter (B) nor the candidate (C) from a multisignature address without knowing all three public keys (KeyA, KeyB, and KeyC) and knowing to whom they belong. Once the election is concluded the organization is able to, via the `2-of-3` multi signature address, spend the coins the voter gave to the candidate to fund the address of the candidate, AddressC. This provides an unequivocal link between the vote and the candidate which can be seen and validated by anyone.

While the proposed method does provide a valuable alternative to current proprietary electronic voting systems and has the benefits of; protecting the secrecy of the ballots, allowing free, independent audits of the results and minimizing the trust level required from the organizers [1]. It does come with several drawbacks; the independent validation of votes before the organization funds a candidate's public address, AddressC, can only be done by the voting individual on their ballot choice. The protocols dependence on the Bitcoin blockchain could pose problems with subsequent elections as there is no definitive boundary between one election and the next. The currency units, although in very small denominations, could be transferred out of the election to private addresses for an individual's gain (though even if 100% of the vote currency was lost, the cost of an election would likely be less than if done using current methods).

Another proposal is that of Universal Cast-as-Intended Verifiability [8] which allows any party (not only the voter) to publicly verify that an encrypted cast vote really matches the selection of a voter. Their proposal allows a voter who's eligible to vote to register with a registrar, who then generates a pair of public-secret values for each voting option in the election. These secret values are sent to the voter, while the public ones are published, linked to the voting options they are related to. During the voting phase, the voter provides her selected voting options and a subset of the secret values she received during registration to the voting device. The voting device then encrypts the voters selections and creates a non interactive zero-knowledge proof, which will be valid only in the case that the voting device encrypted what the voter selected. Due to the zero-knowledge property of the proofs, they can be publicly verified while maintaining the voters privacy.

While this may seem like a vastly superior proposal on the surface, the additional complexity of the underlying system, that is the inclusion of zero-knowledge proofs, should not be underestimated. Furthermore, zero-knowledge protocols, despite being proposed in the late 1970s, are still in their relative infancy when compared to Blockchain technology (e.g. zCash whitepaper [9], 2014) and therefore have not been through the same level of scrutiny nor do they have the same level of development or adoption. The protocol also requires the voter to supply a secret value for each of the voting options they did not choose which, may require considerable effort if the ballot is large enough, and is counter intuitive to what would usually be expected to cast a vote.

3.2 Ethereum

3.2.1 What is Ethereum

In summary, Ethereum is an open software platform based on blockchain technology that enables developers to build and deploy decentralized applications. The block chain is a decentralized network of computers who, at the most basic level, all maintain a ledger in consensus with each other. One block is added at a time, each block contains a mathematical proof that verifies it's addition to the chain and the transactions within are protected by a strong cryptography.

Ethereum is poised to become the next greatest innovation based on block chain technology. So is Ethereum similar to Bitcoin? It does have similarities, but not really. Like Bitcoin, Ethereum is a distributed public blockchain network, however there are some significant technical differences between the two. The most important distinction to note is that Bitcoin and Ethereum differ substantially in purpose and capability. Bitcoin offers one particular application of blockchain technology, a peer to peer electronic cash system that enables online payments. While the bitcoin blockchain is used to track ownership of digital currency (bitcoins), the Ethereum blockchain focuses on running the programming code of decentralized application [10].

Ethereum enables developers to build and deploy decentralized applications. A decentralized application or 'Dapp' serves some particular purpose to its users, for example Bitcoin, is a Dapp that provides its users with a peer to peer payment system. Because decentralized applications are made up of code that runs on a blockchain network, they are not controlled by any individual or central entity. Running these Dapps on a decentralized platform, the blockchain, means they benefit from all of its properties [11]:

- Immutability, a third party cannot make any changes to data.
- Corruption and tamper proofing, as apps are based on a network formed around the principle of consensus, making censorship impossible.
- No central point of failure, as Dapps can be run on every node in the network.
- Secured using cryptography, applications are well protected against hacking attacks and fraudulent activities.
- Zero downtime, Dapps never go down and can never be switched off.

3.2.2 Ethereum Blockchain

The concept of the blockchain was originally outlined in a white paper [12] authored under the pseudonym Satoshi Nakamoto in November of 2008 and was quickly followed by an open source release of the Bitcoin proof-of-concept source code in January 2009 [13]. This is the distributed ledger which underpins the entirety of the Bitcoin and Ethereum systems. A distributed ledger is a consensus of replicated, shared, and synchronized digital data geographically spread across multiple sites, countries, and/or institutions [14]. This ledger is stored locally on every node in the network which is running the full version of the blockchain software [15] and records every transaction sent and confirmed on the network (the current size of the Ethereum Blockchain is around 21GB, March 2017[16]). This complete history, coupled with the fact that it is an open network, means that anyone can see what is happening in the network, not just now but during all periods in the past. This is extremely powerful as it allows an individual to fully audit the entire contents of the Blockchain without relying on external parties. This process is, in fact, what happens when you first download the full version of many blockchain reliant software [17].

While the Ethereum Blockchain is not the only most mature distributed ledger in existence, it does have several years of being a publicly proven method to achieve distributed consensus and does this via the 'proof-of-work mining' process [14]. This is how new information gets added to the blockchain, by nodes in the network running a special 'mining' variant of the Ethereum software which uses considerable computing resources to win the right to add another block to the Blockchain, which is accompanied by a reward for the winning user. The concept of 'proof-of-work' is a method of ensuring that the information being added to the Blockchain was difficult (in terms of cost and time) to be made, though is easy for others to validate that the requirements were met [18]. This means that the expenditure of computing power serves to secure the integrity of the Blockchain, while the miners themselves verify through public-private key cryptography the validity of each transaction they include in a block.

Blocks are chained together making it impossible to modify transactions included in any one block without modifying all following blocks; as a result, the cost to modify a particular block increases with every new block added to the block chain, magnifying the effect of the proof of work [17][19]. This is why, although a transaction is deemed clear upon its inclusion in a block on the Blockchain, best practices dictate that a user considers a transaction confirmed after its inclusion in a block and the addition of five subsequent blocks to the Blockchain [20].

The difficulty of the proof-of-work mining needs to be controlled, so that an average mining time of around 12 seconds per block is maintained. This time is somewhat arbitrary but is an attempt to find a balance between accepting transactions quickly and minimizing instability and waste in the network, as, while a new block is being distributed other miners will be working on an obsolete problem. As more miners join the network the block creation rate will increase due to the greater collective computational power. Therefore, every 2,016 blocks the difficulty of the mathematical challenge is recalculated so that the average mining time returns to normal [17][18].

Despite the media often suggesting that bitcoin (and the Blockchain technology behind it) is an anonymous payment system, the Blockchain is in fact a transparent record of all user transactions on the network. Blockchain transactions are in fact pseudonymous, and your transactions in the network are like writing under a pseudonym. If an author's identity is ever linked to their pseudonym then everything written under that pseudonym will be revealed [21]. This is particularly poignant when considering the Blockchain as every transaction is stored forever, therefore a compromised address could lead to all transactions being linked to a person. There are however ways to reduce the amount of statistical analysis which can be done on transactions that a person is a part of which help to achieve reasonable anonymity [22].

3.2.3 Mining and Ether

Ether is the fuel of the Ethereum system. It is the currency of the Ethereum network with which the payment of computation is achieved. Ethereum, like all blockchain technologies, uses an incentive-driven model of security where transaction consensus is based on a "proof-of-work" criterion of a given difficulty.

The block chain on which the Ethereum executes certain environment is known as the Ethereum Virtual Machine (EVM) [11]. Each participating node within the network runs the EVM and performs the proof of work algorithm called Ethash which involves finding a nonce input to the algorithm so that the result is below a certain threshold (depending on the difficulty) [23]. There is no better strategy to find such a nonce than enumerating the possibilities while verification of a solution is trivial and cheap. If outputs have a uniform distribution, then we can guarantee that on average the time needed to find a nonce depends on the difficulty threshold, making it possible to control the time of finding a new block just by manipulating difficulty [23].

This is the process of how transactions are validated, new transactions are forwarded around the network and placed into a pool of unconfirmed transactions. These are not considered 'accepted' yet but are available for all to see almost instantaneously. Miners draw from this pool to create a candidate next set of transactions to be officially accepted which will form the next block. The full text of all of these candidate transactions, along with the hash of the previous block and a nonce, are input into the the hash function (Ethash) and miners will try different values for the nonce until the resulting hash is below a certain value. Because it's a cryptographic hash, there's no way to find a nonce that satisfies the output hashes condition other than attempting to guess [17]. At this point, all of the miners are in a competition to find the hash first, each with a potentially different set of transactions to confirm. Once a miner succeeds they announce their solution to the rest of the network, their block becomes the next block in the Blockchain, and the transactions therein become confirmed. This strategy means that one miner will choose the next set of confirmed transactions, but the hash function effectively makes the miner a random one. All other mines then validate this new block, and the transactions held within, and can choose to accept it and start work on the next block. As the new block contains the hash of the previous block, this forms a chain of confirmed blocks securing the order of the transactions held within.

Occasionally, two miners may find a solution to the problem at the same time creating two potential next blocks in the chain. When miners produce simultaneous blocks at the end of the block chain, each node individually chooses which block to accept, this is usually the first block they see. Eventually another miner finds the solution to another block which attaches to only one of the competing blocks. This makes that side of the fork stronger and, as the general consensus is to use the strongest chain, other nodes will switch to this longer Blockchain [17]. While this is statistically unlikely to happen, it is even more unlikely for the subsequent blocks to be solved at the same time, meaning that one fork will grow quicker than the other and the fork will resolve itself quickly. Transactions that were in the fork that wasn't chosen are not lost and are placed back into the unconfirmed transactions pool [24]. The fact that the end of the chain can be forked and rearranged means you shouldn't trust transactions at the end of the chain as much as ones further back. In Ethereum, a transaction is not considered confirmed until it is part of a block in the longest fork, and at least five blocks follow it. In this case we say that the transaction has "5 confirmations". This gives the network time to come to an agreed-upon the ordering of the blocks [25].

The successful miner of a block receives a reward for the 'winning' block, consisting of exactly 5.0 Ether along with all of the gas expended within the block, that is, all the gas consumed by the execution of all the transactions in the block. Over time, it's expected the gas reward will dwarf the block reward and become the main incentive for miners to continue working [23].

3.2.4 Transaction Costs and Gas

Ethereum does have a small transaction fee, just like Bitcoin, where users pay a relatively small amount to the executor of your transaction. The sender has to pay the fees at each and every step of the activated program, this includes the memory, storage and computation [11]. The size of the fee paid is equivalent to the complexity of the transaction, i.e. the more complex the commands you wish to execute, the more gas (and Ether) you have to pay. For example if "Alice" wants to send "Bob" 1 Ether unit, there would be a total cost of 1.00001 Ether to be paid by Alice. However if A wanted to deploy a contract or run a contract function, there would be more lines of code executable, therefore more energy consumption placed on the distributed Ether network and she would have to pay more than the 1 Gas done in the first transaction [26]. Some computational steps cost more than others, either because they are more computationally expensive or because they increase the amount of data that has to be stored in the state.

Gas is the internal pricing for running a transaction or contract in Ethereum. The gas system is not very different from the use of kilowatts in measuring electricity except that the originator of the transaction sets the price of gas, to which the miner can or not accept [26]. Ether and Gas are inversely related say for instance if the Ether price increases, than Gas price should decrease to maintain the concept of real cost [11]. There is also a blocksize limit, so the more space your transaction takes up the more you have to pay to get it validated. With Bitcoin, miners prioritise transaction with the highest mining fees. The same is true of Ethereum where miners are free to ignore transactions whose gas price limit is too low.

The reason for the inclusion of a gas price per transaction or contract is to deal with the Turing Complete nature of Ethereum and its EVM essentially to guarantee that code running in the network will terminate. So for example, 0.00001 Ether or 1 Gas can execute a line of code or some command. If there is not enough Ether in the account to perform the transaction or message then it is considered invalid. This aims to stop denial of service attacks from infinite loops, encourage efficiency in the code and make any potential attacker pay for the resources they use (whether that be bandwidth, CPU cycles or storage) [26].

3.2.5 Smart Contracts

Smart contracts are the key element of Ethereum. In them, any algorithm can be encoded, they can carry arbitrary state and can perform any arbitrary computations even being able to call other smart contracts. This gives the scripting capabilities of Ethereum tremendous flexibility [27]. When run a smart contract becomes like a self-operating computer program that automatically executes when specific conditions are met and because they run on the blockchain, they run exactly as programmed without any possibility of censorship, downtime, fraud or third party interference. While all blockchains have the ability to process code, most are severely limited. Ethereum is different in this respect as rather than giving a set of limited operations, Ethereum allows developers to create whatever operations they want allowing developers to build thousands of different applications that go far beyond anything seen previously [10].

The Ethereum Virtual Machine is where smart contracts are run. It provides a more expressive and complete language than bitcoin for scripting and is Turing Complete. A good metaphor is that the EVM is a distributed global computer where all smart contracts are executed [28]. There are several higher level languages used to program smart contracts, but Solidity is the most mature and widly adopted. Its syntax is similar to that of JavaScript, its statically typed, supports inheritance, libraries and complex user-defined types among other features.

Smart contracts are run by each node as part of the block creation process and, just like in Bitcoin, this is the moment where transactions actually take place. An important part of how smart contracts work in Ethereum is that they have their own unique address in the blockchain. In other words, contract code is not carried inside each transaction that makes use of it. Instead contracts are "deployed" to the blockchain in a special transaction that assigns an address to a contract. This transaction can also run code at the moment of creation. After this initial transaction, the contract becomes forever a part of the blockchain and its address never changes. Whenever a node wants to call any of the methods defined by the contract, it can send a message to the address of the contract, specifying data as input and the method that must be called. The contract will then run as part of the creation of newer blocks up (subject to the gas limit or completion) and can return a value or store data [27].

4 Design

4.1 Overview

In this paper, I present an “end-to-end verifiable system” built on the Ethereum Blockchain, i.e., a system where a voter can be assured their vote has been fairly counted, only eligible voters are allowed to vote and the tallied results of the election are publicly verifiable.

Although this system has been designed and developed with the idea of a national general election in mind, the protocols and ideas involved could be applied to smaller scale ballots which wish to provide transparency in their audit. Although we wish to minimize trust in a central authority, due to the nature of these type of elections (where there needs to be some degree of voter eligibility verification), we cannot fully decentralize this system as we need to only allow those eligible the right to vote. Despite needing to verify an individual, we still need to ensure that their votes are publicly anonymous, especially given the public transactions underpinning the blockchain concept while providing the ability for an individual to verify that their vote was correctly counted.

I do not see this system as a direct “replace all” for national election voting. I believe there will still be a need for traditional voting implementations in certain situations; for example, maintaining postal vote for the elderly who may not have the technical capability or equipment for online voting. However I do think that this could be phased in along side traditional voting, eventually replacing the pre-existing e-voting systems and ultimately becoming the main way for the majority of people to choose their government.

The designed schema for this protocol is the following:

1. Ballot creation:
    - The available ballots in the election are designed and decided upon.
    - A smart contract is created and pushed to the blockchain for each ballot containing all of the voting options.
2. Pre-election voter verification:
    - Voter registers with an external voter registrar after providing a valid ID (this could be accomplished using pre-existing government electoral registration protocols).
    - This external registrar generates a *user_id* and *nonce* which can be used by the voter to log in to the system.
    - This *user_id* is then tied to any ballots the voter is eligible for.
3. Voter registration:

- The voter logs into the system using the received *user_id* and nonce at which time they are immediately required to change their login credentials.
- The voter can then register to vote through the online system for each of the ballots they are eligible for.
- A unique Ethereum address, *voter_address*, is generated and validated (while not being linked to the *user_id*)
- The *user_address* is added to the ballots smart contract which entitles this address to vote in that ballot.
- The address is funded with enough Ether for the voter to cast their vote.

4. Voting:
    - When the voter decides to cast their vote, they are presented with an interface mirroring the options in the ballot smart contract.
    - Upon the voter selecting their options, the contract is funded with the voters selected options.
    - At this point the voters choice is immutably entered into the blockchain and the tally is verifiable by all.
5. Election result:
    - Once the election is over, due to the nature of the smart contract design, no more votes can be added for any candidate.
    - The tally for each candidate is publicly verifiable by anyone along with all of the funded transactions casting votes.

4.2 Docker

Over the past few years, container technology has become increasingly promising as a means to seamlessly make software available across a wider range of platforms and allow developers to worry less about the eventual runtime environment (as this can be standardized). Docker containers provide a way to "wrap up a piece of software in a complete filesystem that contains everything it needs to run" [29].

There are several benefits to the use of Docker containers; this could substantially reduce the effort required to create and validate a new software releases, since Docker containers create their own dedicated environment and testing on one OS means that the application will run the same on any OS capable of running Docker. In addition, Docker containers provide a quick and easy way to install and use a software release, for our application this could mean faster patches if needed as you would simply swap out the image being used. Other benefits include faster bootup of containers (compared to virtualization), closer development to production parity, immutable infrastructure and improved scaling (on a per-container basis) [30]. There is also a security argument to be made for using containers, as they offer a degree of isolation for each enclosed application and only expose those services which you explicitly specify. The previous point about patching also add to security, as legacy applications often forgo patches in sensitive environments due to the possibility of breakages. When using containers, changes can be fully tested in the container which can then be swapped into the production environment [31].

All of my development for this project was conducted inside of Docker containers. I decided on this because there are very distinct separations between the applications which make up my system (this is described in the section on System Design 4.6), so running each one inside a Docker container seemed the logical thing to do. It also meant that I could control the startup of the system as a whole and expose (between containers) only those services necessary to communicate.

4.2.1 Ethereum

Ultimately the decision to use Ethereum for this project was an easy one. Ethereum is not just a digital currency, it is a blockchain based platform with many aspects desirable when designing and creating distributed applications. There simply is no other technology (at the time of writing) that can offer the same level of customization of decentralized programming and has a similarly substantial user base.

I initially investigated using the Bitcoin protocol as a method to store data (votes) immutably and reviewed several papers proposing voting solutions [32]. All of these proposals were however 'clunky' in design due to the inescapable fact that it is not what Bitcoin was designed for. Bitcoin was written in a stack based language that isn't Turing Complete as it was designed as a distributed value transfer ledger.

Ethereum, on the other hand, has contracts written in a Turing Complete Language meaning that anything can be done with it given enough time and enough computing power. This means that Ethereum was built specifically to handle smart contracts over simple currency transactions and although Bitcoin could be built upon to allow the functionality that Ethereum has it would seem unnecessary and be likely cause more problems when programming the application.

Ethereum's block confirmation time is also much shorter than Bitcoins. Bitcoin is currently at around 10 minutes whereas Ethereum is around 12 seconds. So consequently, while bitcoin transactions normally take a few minutes to be cleared, Ethereum transactions are cleared almost instantly and at most in a matter of seconds.

The choice to use Solidity as the programming language for my Smart Contracts was mostly governed by maturity. The simple fact is, that there is no other competing languages that have sufficient levels of publicly tested development to justify their use. The closest competitor looks to be Viper [33] though this is still in the very early stages of development and lacks many features I would require to be able to use it for this application.

4.3 Internal node networking

The design of the system requires services to be split up into separate nodes each with a specific set of functionalities. Many of these nodes require services from other nodes to operate (for example, the “Application Server” querying the “Ballot Regulator” for the list of ballots available to a user) and so efficient inter-node communication is vital. I ultimately decided to use the Python networking toolkit Twisted [34] for all of the networking of this system. The main draw of Twisted was that it offers asynchronous messaging through its AMP protocol.

I chose to use asynchronous messaging for a few reasons. First, is that ultimately there will be a human being operating this system eventually and people are inherently impatient. By interleaving the tasks together the system can remain responsive to user input while still performing other work in the background (e.g. user can still interact with the front end while the backend is registering the user to the ballot contract). While the background tasks may not execute any faster, the system will be more responsive for the person using it.

The second relates to the fundamental idea behind the asynchronous model; that it is an asynchronous program. When we hit a task that would normally block in a synchronous program, we can instead execute some other task that can still make progress. Therefore, an asynchronous program should only block when no task can continue. This is highly applicable to this application as, due to the standard block confirmation time of 12 seconds, we could sometimes be waiting a much longer length of time than would be acceptable to block for.

Twisted handles these asynchronous messages through the ‘Deferred’ interface. Deferred is a way to define what will be done to the object once it does exist after the result of an asynchronous call. Each ‘Deferred’ has a callback and an errback where you define what code you wish to execute once the object exists. When the object finally does exist, it is passed to the callback and similarly, if an error occurs the error is passed along to the errback. In my system, each node defines its own AMP methods, which can be called remotely and return a ‘Deferred’, inside the *network_request.py* class (e.g. OnlineBallotRegulator/network_request.py).

Twisted also offers the ability to secure connections using TLS which provides some necessary security benefits. Each party can be required to initially present a certificate verifying their identity, this would ensure that an attacker cannot ‘man in the middle’ attack this system and splice in their own node to intercept traffic. It also provides confidentiality as communication between nodes is encrypted and some degree of integrity as the TLS protocol checks to ensure encrypted messages actually came from the party you originally authenticated to.

4.4 Blind Signatures

A “Blind signature” is a signing scheme where the signer doesn’t know the content of the message he/she is signing but the resulting blind signature can be verified against the original unblinded message just like a regular digital signature [35].

This can be analogized to an individual, Alice, placing a letter inside a carbon paper lined envelope. This is then handed to a trusted third part, Bob, who (without opening the letter) signs the outside of the document and hands it back to Alice. Due to the carbon paper inside the envelope, Bobs signature is also transferred to the letter within. Alice can then extract the letter which now contains Bobs signature despite him never having seen the letter contents.

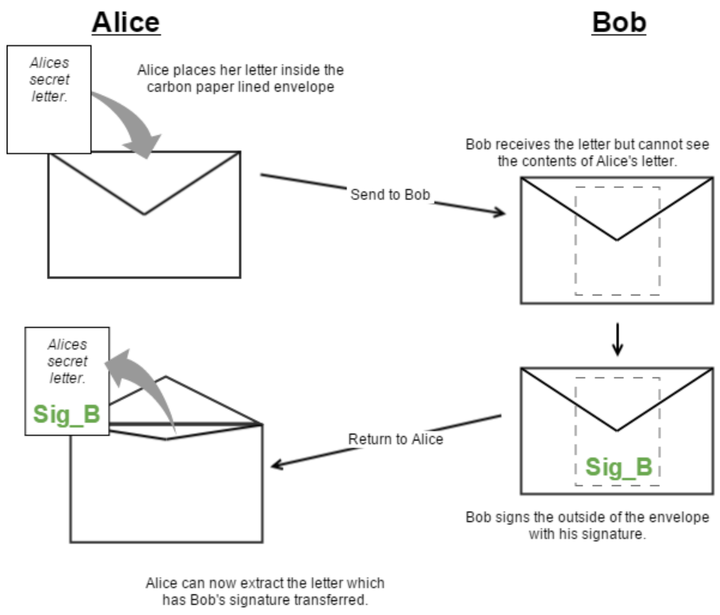

Alice
Bob
Alices secret letter.
Alice places her letter inside the carbon paper lined envelope
Send to Bob
Bob receives the letter but cannot see the contents of Alice's letter.
Alices secret letter.
Sig_B
Return to Alice
Sig_B
Bob signs the outside of the envelope with his signature.
Alice can now extract the letter which has Bob's signature transferred.

Figure 1: Blind signature analogy showing how Bob never sees the contents of Alice’s message despite being able to sign it.

Now we can try to translate this to the language of cryptography. Suppose Alice has a message m that she wishes to have signed by Bob, and she does not want Bob to learn anything about m. Let (n, e) be Bob’s public key and (n, d) be his private key. Alice generates a random value r such that $gcd(r, n) = 1$ and sends $x = (r^e m) \bmod n$ to Bob. The value x is “blinded” by the random value r; hence Bob can derive no useful information from it. Bob returns the signed value $t = x^d \bmod n$ to Alice. Since $x^d \equiv (r^e m)^d \equiv rm^d \bmod n$, Alice can obtain the true signature s of m by computing $s = r^{-1}t \bmod n$. Now Alice’s message has a signature she could not have obtained on her own [36].

4.5 Testing

As my project is written in Python and I am heavily using Django to present the frontened to the user, I have access to a wealth of test management functionality built into the platform. I have written a mixture of unit and integration tests which I believe provide good coverage of the features and actions of the system. As all interactions initially start from the "Application Server", my testing reflects this and tests actions as if they were being called by an end user.

4.5.1 Tests

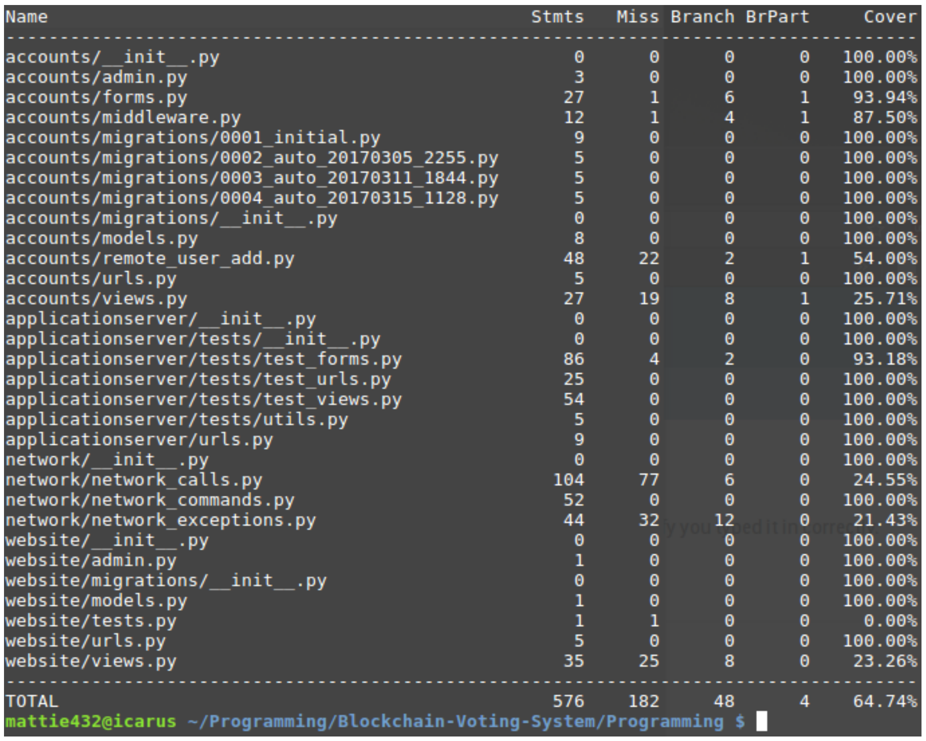
My system uses the NoseTests framework which makes the process of testing the project easier due to its test discovery and automated running. In nose, each test is a function whose name begins with 'test_'. We can group tests together in files whose names also begin 'test_'. To execute our tests we run the command *nosetests* which recursively searches the current directory and its subdirectories for test files, and runs the tests they contain.

My written tests cover two areas, the first being Django specific interactions. This includes tests to ensure that defined urls remain accessible (such as the login and registration pages) along with form entry validation checks (such as registering with an invalid email address) ensuring that only validated data enters the system. The second area includes user interaction, there are checks to ensure that actions started by the user complete successfully. This includes initial setup such as entering the 'initial information request' data through to being able to successfully register for a ballot (along with all of the networking calls involved).

4.5.2 Coverage

The NoseTests framework also offers the ability to produce a 'code coverage' report. This is useful to ensure that the written tests are actually testing your code as the report returns how much of your code is exercised by running the tests. While this does not guarantee the effectiveness of the testing, it can be useful to identify areas of weakness for further improvement of test coverage.

While I have not achieved total code coverage, I believe the main areas which provide heavy service to the application are adequately covered.



```
Name                                                  Stmts   Miss Branch BrPart   Cover
-----------------------------------------------------------------------------------------
accounts/__init__.py                                      0      0      0      0  100.00%
accounts/admin.py                                         3      0      0      0  100.00%
accounts/forms.py                                        27      1      6      1   93.94%
accounts/middleware.py                                   12      1      4      1   87.50%
accounts/migrations/0001_initial.py                       9      0      0      0  100.00%
accounts/migrations/0002_auto_20170305_2255.py            5      0      0      0  100.00%
accounts/migrations/0003_auto_20170311_1844.py            5      0      0      0  100.00%
accounts/migrations/0004_auto_20170315_1128.py            5      0      0      0  100.00%
accounts/migrations/__init__.py                           0      0      0      0  100.00%
accounts/models.py                                        8      0      0      0  100.00%
accounts/remote_user_add.py                              48     22      2      1   54.00%
accounts/urls.py                                          5      0      0      0  100.00%
accounts/views.py                                        27     19      8      1   25.71%
applicationserver/__init__.py                             0      0      0      0  100.00%
applicationserver/tests/__init__.py                       0      0      0      0  100.00%
applicationserver/tests/test_forms.py                    86      4      2      0   93.18%
applicationserver/tests/test_urls.py                     25      0      0      0  100.00%
applicationserver/tests/test_views.py                    54      0      0      0  100.00%
applicationserver/tests/utils.py                          5      0      0      0  100.00%
applicationserver/urls.py                                 9      0      0      0  100.00%
network/__init__.py                                       0      0      0      0  100.00%
network/network_calls.py                                104     77      6      0   24.55%
network/network_commands.py                              52      0      0      0  100.00%
network/network_exceptions.py                            44     32     12      0   21.43%
website/__init__.py                                       0      0      0      0  100.00%
website/admin.py                                          1      0      0      0  100.00%
website/migrations/__init__.py                            0      0      0      0  100.00%
website/models.py                                         1      0      0      0  100.00%
website/tests.py                                          1      1      0      0    0.00%
website/urls.py                                           5      0      0      0  100.00%
website/views.py                                         35     25      8      0   23.26%
-----------------------------------------------------------------------------------------
TOTAL                                                   576    182     48      4   64.74%
mattie432@icarus ~/Programming/Blockchain-Voting-System/Programming $
```

Figure 2: Test coverage results from the "Application Server"

4.5.3 Pylint

Pylint is a Python tool that checks a module for coding standards with a range of checks run from Python errors, missing docstrings, unused imports, unintended redefinition of built-ins, to bad naming and more. By default Pylint offers an overwhelming amount of information and errors relating to extremely minor, often stylistic, differences (e.g. *W:675, 0: Class has no __init__ method (no-init)*). This error is not only useless, it masks any potential larger problems due to the scale of output Pylint produces (initially for my project over 5,000 lines). To make Pylint more usable I had to configure Pylint to only check areas I specified, this included only displaying detected errors (rather than information and warnings) along with certain warnings which I switched on (such as 'too many arguments'). I started with a very basic configuration and steadily expanded it over the duration of my project.

4.6 System Design

There were several points to consider when designing the high level plan of this voting system. The most important of which being the need to ensure separation of a voters account (tied to an individual) and the address they use to vote in the ballot contracts. This directly affected how I designed the system and lead to me splitting the system into distinct sections to take on specific roles; the *Application server* (voter interaction), the *Online Account Verifier* (verify the legitimacy of an account to vote) and the *Online Ballot Regulator* (manages the ballot contracts).

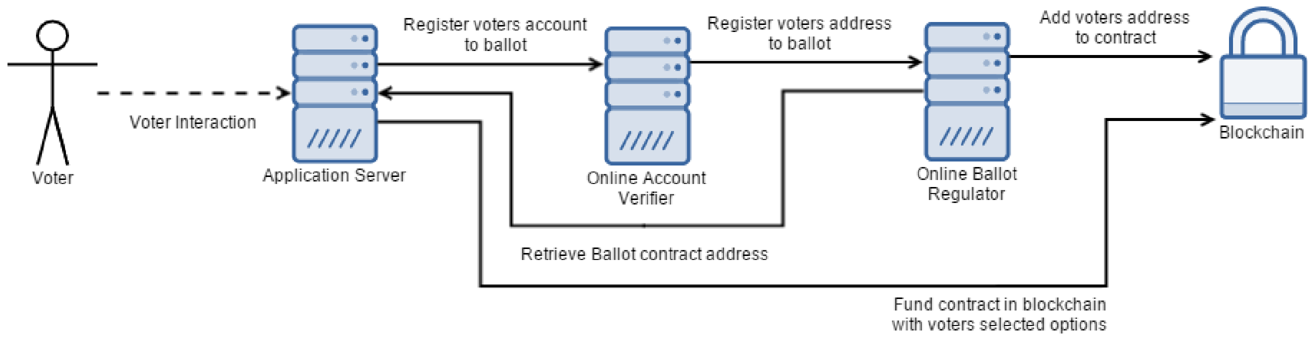

Voter
Voter Interaction
Application Server
Register voters account to ballot
Online Account Verifier
Register voters address to ballot
Online Ballot Regulator
Add voters address to contract
Blockchain
Retrieve Ballot contract address
Fund contract in blockchain with voters selected options

Figure 3: Outline of system showing the basic interactions between nodes.

Splitting the system like this adds both a layer of security and increased scalability. As all of the data is not centralized on one node (or in one database) this would make it more difficult for a potential attacker to breach the system as they would have to get past the security of at least two nodes to obtain anything useful. The big benefit here is scalability, as if this system was used for a general election, more traffic would be seen on some nodes than others (e.g. more logins through the *Application Server* than ballot queries in the *Ballot Regulator*). We could then independently scale each node accordingly.

As each of these nodes are effectively self contained, they only need to expose a select number of services to allow inter-node further decreasing the possibilities for attack. As many of these services only need to be called from other nodes, we can add authentication to these connections to ensure that is upheld. In fact, there are only two external points of contact with this system, the web interface for the voter to use and the blockchain interface to contact other Ethereum nodes, meaning we could black box our system from the outside world fairly completely (see Internal node networking 4.3 sections).

Finally, I chose Python as the main language for this system as it has strong frameworks for building web applications (Django) which I heavily utilized these for the *Application Server* and *External Voter Registration* nodes. There is also strong development of Web3.py [37], a Python implementation of web3.js, which is heavily used when interacting with Ethereum through the Geth client. I also discovered the Twisted Python networking package [34] which includes an Asynchronous Messaging Protocol (AMP) implementation for calling remote methods and the Pycrypto library which allowed me to perform the RSA blind signature verification which is crucial to separating a user from their Ethereum address.

4.6.1 External Voter Registration

The "External Voter Registration" node is meant to represent *some external registrar* who is not directly a part of the designed voting system, but plays a crucial role in the verification of a voters validity (e.g. the UK governments register to vote system). Their role should be purely during the "pre-election registration" stage where the voting populous registers their intent to vote in the upcoming election.

I envision this as being very similar to registering to vote with GOV.UK, where you send your uniquely identifying information (e.g. date of birth, national insurance number, etc) and are then, if verified as a valid voter, registered for the appropriate ballots for your area. As such there would be no need to write this application from scratch as it would make sense to utilize the existing verification systems already in place and modify them to interact with the Ethereum voting system as appropriate.

For the purpose of demonstrating my application I have written the software for the "pre-election registration" node so that I can register a new voter in the system easily (as there needs to be multiple database entries created) and so I can easily deploy a new ballot contract to the blockchain from a specific Ethereum address (necessary so that we have permission to register a voter to a ballot contract). The "External Voter Registration" application runs on a Django base, this is because I needed to present a web interface for interaction when registering a new ballot and user.

4.6.2 Application Server

The "Application Server" node is the main place for voter interaction with the system. Here, the Django backend provides the web interface for users to log in, register to a ballot and ultimately vote.

User interaction is centred around a dashboard page which displays key information to the user (ballots they are eligible for, whether they have voted, etc). From here the user can invoke all voting operations available to them such as registering for a ballot and voting, to account management options such as changing passwords. When displaying the voting page after a user requests to vote in a ballot, the application server will query the blockchain contract for the voting options and also display current statistics about the ballot (such as number of registered voters, the current votes per candidate and voter turnout).

4.6.3 Online Account Verifier

The “Online Account Verifier” holds the role of authenticating that a user is eligible for a ballot while retaining the anonymity of the final Ethereum address the user casts their vote with. This is accomplished with the use of blind token signing, the process of signing a message without seeing the contents. This means we can verify a voter and return them a signed token which can be sent in the future alongside an Ethereum address to verify that the address is legitimate.

Database table holding token requests.

token_request_id	blind_token_hash	user_id	ballot_id	created_on
1	54baa883f28a768d6bd352d12c730 7d1592d394acea4337a018ee2d0f1 43642a	68401	1234	2017-04-08 18:08:49.527817
2	4451458fe4d479387fdca5dde405a 16fd2fca2c4f92c516fea78b2f0d27 27f7c	67173	1234	2017-04-08 18:27:25.984678

Database table holding token registrations.

register_vote_id	signed_token_hash	voter_address	ballot_id	created_on
1	19a91bc3d91b5873e5b0e4687a988 a08b362bd357a85ce3bc109cba4ed 984407	0x79957083494aa13895ae6 bad9f04f2bb99f0ad39	4321	2017-04-08 18:30:25.736455
1	13jad238sjkgfn39n3asd882nd2d8 77ad327dnk3lq9afv4b234akd3nd8 98hd82	0x925A8e765F9563D979b5 76A68210903a9968B8Be	5432	2017-04-09 12:23:41.162341

Figure 4: Database tables used to store the requests to register an Ethereum address to a ballot, for a registered user.

The first table is used to verify that a user account has not requested to register for a particular ballot previously with a blind token. The second table is used to keep track of registered Ethereum addresses and which ballot they are registered to. Both of these tables are only used for internal state storage, they are used to determine if a user has already registered for a ballot and show the ‘register’ or ‘vote’ options accordingly. The data stored here is not used by the application server to retrieve any voter credentials. These tables store the token hashes so that we can ensure that tokens cannot be reused by an attacker.

4.6.4 Online Ballot Regulator

The "Online Ballot Regulator" manages everything to do with the voting ballots. This includes all created ballots in the system and their corresponding Blockchain addresses through to which user accounts are registered to which ballots.

Database table showing which ballots a userID is allowed to vote in.

ballot_register_id	user_id	ballot_id	created_on
1	1234	1234	2017-04-08 18:26:23.440678
2	1234	6543	2017-04-08 18:26:23.440678
3	2345	6543	2017-04-08 18:26:23.440678
4	67173	1234	2017-04-08 18:26:44.067895
5	67173	4321	2017-04-08 18:26:44.134928

Database table holding information about each ballot in the system.

ballot_id	ballot_name	ballot_address	created_on	ballot_interface	ballot_end_date
1234	Election of the Member of Parliament for the Harborne Constituency	0x127c73Af1F9E0efF8226 Db6bdf04310fDEe674F6	2017-04-08 18:26:23.440678	x800358071002e	1603238400
4321	Election of Police and Crime Commissioner for Edgbaston area	0x8C872c720DF854a058C3 D1DD54e4CEE51d798B6A	2017-04-08 18:26:23.440678	x800358071002e	1603238400
6543	Referendum on the United Kingdoms membership of the European Union	0x7654EC4067e8fA04184D 68fF08169A29A3B20F19	2017-04-08 18:26:23.440678	x800358071002e	1603238400

Figure 5: Database table used to store the requests to register an Ethereum address to a ballot for a registered user.

The "Online Ballot Regulator" is where the "External Voter Registrar" sends the available ballots for a user account to be stored. This node is the most queried in the entire system as it holds information about ballots the user account is tied to along with the address of ballots in the blockchain.

4.6.5 Blockchain Ballot Contract

At the heart of this system is an Ethereum Smart Contract, from which the entire system design is centred around. The smart contract is how we interact with and store data in the blockchain and, due to its turing complete programming language, we are able to express complex properties which are guaranteed to be upheld. A more detailed explanation of the inner working of smart contracts can be found in the Ethereum Smart Contracts 3.2.5 section.

For this Ethereum voting system I have designed a single smart contract that can be modified for each ballot that the system uses. This contract, upon creation, allows a ballot name and set of candidates to be added so it is re-usable for any number of ballots (note that this still creates distinct and unique contracts in the blockchain, but their underlying code and available functions will be the same). I believe that this contract would be suitable for the majority of ballots currently in use however, if the need arose to write different types of ballot contract (maybe a ballot where you can give candidates a percentage of your vote), these could be easily integrated into the system. The full code for the ballot is available in the "Online Ballot Regulator" node at ethereum/ETHVoteBallot.sol which was written in the Solidity smart contract language and a description of the systems steps for deploying a ballot contract is available in the Ballot Creation 4.7.1 section.

The smart contract is broken down into four distinct sections, the first of which is the global settings and contract constructor. The constructor is called only on the initial deployment transaction when the contract is first entered into the blockchain. Inside the constructor we set the ballots name and end date (this is in seconds since epoch) both of which are no longer editable after this point. The constructor also sets the 'owner' variable of the contract to the Ethereum address that funds the deploy transaction (in our system this address is under the control of the "Ballot Regulator") this owner variable will be used to limit the access to some internal functions further into the contract.

```
// ~~~~~~~~~~~~~~~~~~ Contract Constructor ~~~~~~~~~~~~~~~~~~~~~ //
address owner;                  // The address of the owner.
bool    optionsFinalized;       // If we can still add voting options
string  ballotName;             // The ballot name.
uint    registeredVoterCount;   // Count of registered addresses.
uint    ballotEndTime;          // seconds since 1970-01-01

// Modifier to only allow the owner to call a function.
modifier onlyOwner {
    if( msg.sender != owner ) throw; _;
}

// This function is called *once* at first initialization into the
    blockchain.
function ETHVoteBallot(string _ballotName, uint _ballotEndTime)
{
    if( now > _ballotEndTime)
        throw;

    // Set the owner to the address creating the contract.
    owner = msg.sender;
    optionsFinalized = false;
    ballotName = _ballotName;
    registeredVoterCount = 0;
    ballotEndTime = _ballotEndTime;
}
```

Listing 1: Contract constructor called when deploying the contract.

The next section sets the ballot options for the contract. First we define a structure for each voting option containing the candidate name and their vote tally, these are stored in a dynamically sized array called *votingOptions*. The *addVotingOption* function is used to add new candidates to the contract, note the use of 'throw' here will terminate the contracts running and refund spent ether. The final function, *finalizeVotingOptions*, sets an internal flag which stops any further modifications to the contracts candidates and opens up voting to those addresses added to the contract. We have added a modifier to these functions, *onlyOwner* (defined above), which will only allow the address which created the contract to call these functions.

```
// ~~~~~~~~~~~~~~~~~~~~~~~ Ballot Options ~~~~~~~~~~~~~~~~~~~~~~ //
// Structure which represents a single voting option for this
    ballot.
struct VotingOption
{
    string name;    // Name of this voting option
    uint voteCount; // Number of accumulated votes.
}

// dynamically sized array of 'VotingOptions'
VotingOption[] public votingOptions;

/*
*  Add a new voting option for this ballot.
*  NOTE: this can only be called by the ballot owner.
*/
function addVotingOption(string _votingOptionName) onlyOwner
{
    if( now > ballotEndTime) throw;
    // Check we are allowed to add options.
    if(optionsFinalized == true)
        throw;

    votingOptions.push(VotingOption({
        name: _votingOptionName,
        voteCount: 0
    }));
}

/*
*  Call this once all options have been added, this will stop
    further changes and allow votes to be cast.
*  NOTE: this can only be called by the ballot owner.
*/
function finalizeVotingOptions() onlyOwner
{
    if(now > ballotEndTime) throw;

    if(votingOptions.length < 2) throw;

    optionsFinalized = true; // Stop the addition of more options.
}
```

Listing 2: Functions used for modifying the ballot during setup.

The third section refers to ‘voting options’, here we define another structure for voters but, rather than use an array, we invoke a mapping from the callee address. This means that we can cover the entire address space with the default values of the structure and only need to change the addresses we wish to give voting eligibility (seen in the *giveRightToVote()* function). The *vote()* method can also be seen which amounts to checking if the voter is eligible and incrementing the counter of their chosen candidates vote count.

```
// ~~~~~~~~~~~~~~~~~~~~~~~ Voting Options ~~~~~~~~~~~~~~~~~~~~~~ //
// Structure which represents a single voter.
struct Voter {
    bool eligableToVote;    // Is this voter allowed to vote?
    bool voted;             // State of whether this voter has
        voted.
    uint votedFor;          // Index of 'votingOptions' this voter
        voted for.
}
// State variable which maps any address to a 'Voter' struct.
mapping(address => Voter) public voters;

/*
*  Allow an address (voter) to vote on this ballot.
*  NOTE: this can only be called by the ballot owner.
*/
function giveRightToVote(address _voter) onlyOwner {
    if(now > ballotEndTime) throw;
    voters[_voter].eligableToVote = true;
    registeredVoterCount += 1;      // Increment registered voters.
}

/*
*  Allow an eligable voter to vote for their chosen votingOption.
    If they have already voted, then remove their vote from the
    previous 'votingOption' and assign it to the new one.
*
*  NOTE: if anything fails during this call we will throw and
    automatically  revert all changes.
*/
function vote(uint _votingOptionIndex) {
    if(now > ballotEndTime) throw;
    if(optionsFinalized == false) throw; //Not finalized, cant vote
    Voter voter = voters[msg.sender];   // Get senders Voter struct

    if(voter.eligableToVote == false) throw;

    // If the voter has already voted then we need to remove their
        prev vote choice.
    if(voter.voted == true)
        votingOptions[voter.votedFor].voteCount -= 1;

    voter.voted = true;
    voter.votedFor = _votingOptionIndex;
    votingOptions[_votingOptionIndex].voteCount += 1;
}
```

Listing 3: Contract code relating to voting.

Finally, we include a set of getter functions for various states of the contract. These functions can be used internally by the contract and called remotely by anyone with the contract address. This is how we allow external verifiability of our ballots.

```
// ~~~~~~~~~~~~~~~~~~~~~ Getter Functions ~~~~~~~~~~~~~~~~~~~~~~ //

// Returns the ballots name string.
function getBallotName() returns (string){ ... }

// Returns the number of voting options.
function getVotingOptionsLength() returns (uint) { ... }

// Returns the count of registered voter addresses.
function getRegisteredVoterCount() returns (uint) { ... }

// Returns the name of a voting option at a specific index. Throws
    if index out of bounds.
function getVotingOptionsName(uint _index) returns (string) { ... }

// Returns the number of votes for a voting option at the specified
     index. Throws if index out of bounds.
function getVotingOptionsVoteCount(uint _index) returns (uint){...}

// Returns if the voting options have been finalized.
function getOptionsFinalized() returns (bool) { ... }

// Returns the end time of the ballot in seconds since epoch.
function getBallotEndTime() returns (uint) { ... }
```

Listing 4: Summary of getter functions.

4.7 Pre-election setup

4.7.1 Creating a new ballot contract

The first thing which needs to be done before any other aspect of the election can take place, is publish the smart contract for each ballot in the election to the blockchain (a deeper analysis of the contract is presented in the Blockchain Ballot Contract 4.6.5 section). The smart contract I created can be seen as a 'template' of sorts, allowing different sets of voting options to be added to a similar core structure. The system requests the 'ballot name', 'ballots voting options' (provided as a comma separated list) and the 'end date' of the ballot. Because all of the Blockchain interactions are handled by the "Online Ballot Regulator" we wrap up this information and send it in a network call to the "Online Ballot Regulator".

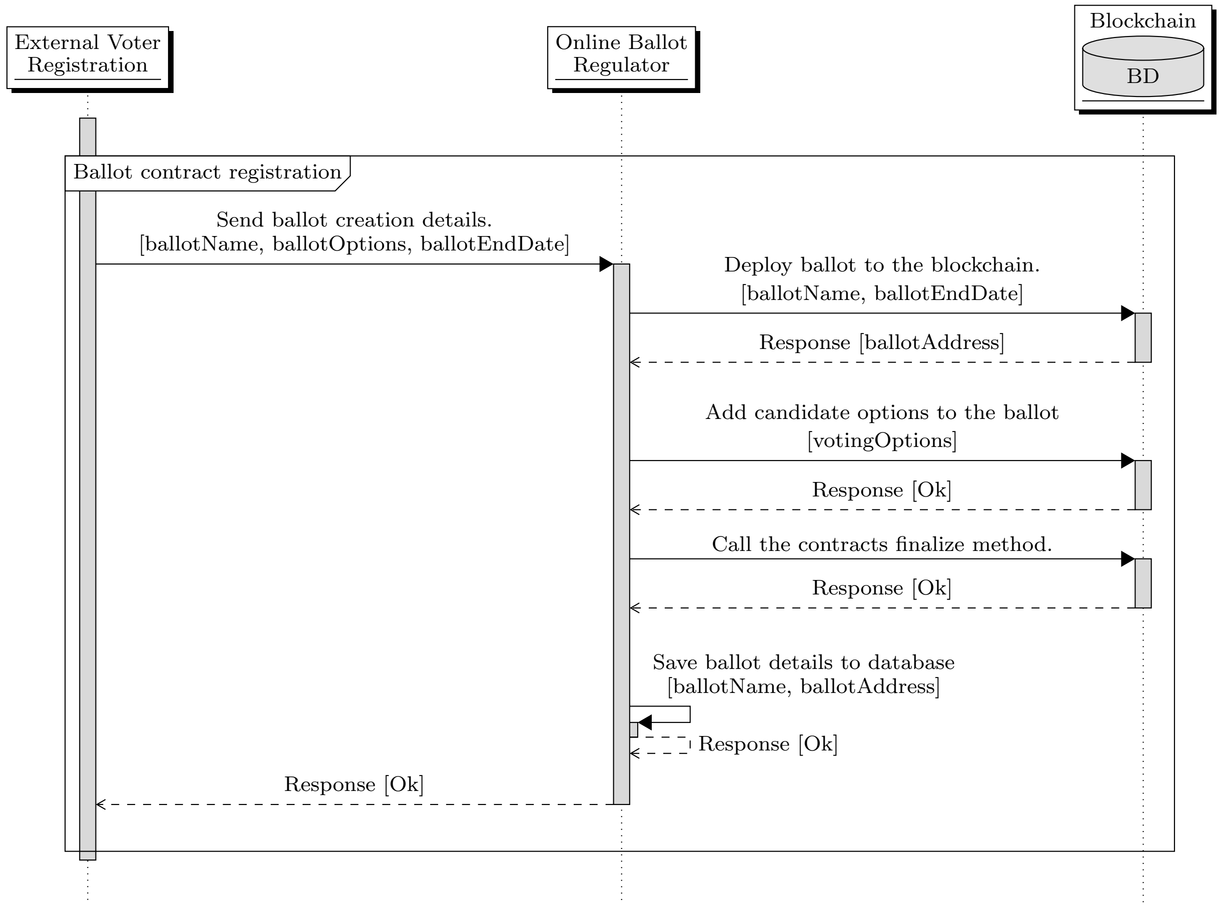

External Voter Registration
Online Ballot Regulator
Blockchain
BD
Ballot contract registration
Send ballot creation details. [ballotName, ballotOptions, ballotEndDate]
Deploy ballot to the blockchain. [ballotName, ballotEndDate]
Response [ballotAddress]
Add candidate options to the ballot [votingOptions]
Response [Ok]
Call the contracts finalize method.
Response [Ok]
Save ballot details to database [ballotName, ballotAddress]
Response [Ok]
Response [Ok]

Figure 6: Registering a new ballot in the blockchain.

The *Online Ballot Regulator* then registers the ballot contract into the Blockchain with the information received from the remote call. This is funded (and therefore deployed) by the *Ballot Regulators* private key and corresponding Ethereum address which, due to the programming of the ballot contract, means that the ballot regulator has exclusive rights to modify the contract. Deploying the contract happens in three stages in the ethereum/ethereum.py class of the *Ballot Regulator*. First, the contract 'template' is deployed to the Blockchain via the Ethereum software run on the server. This is done by sending the compiled contracts bytecode in an Ethereum transaction along with the contract parameters needed to initially setup the contract (the ballot name and ballot end time). Once the contract is deployed and confirmed into the blockchain we can access the contract at a specific address which we will use from here on out to interact with the contract (e.g. 0x127c73af1f9e0eff8226db6bdf04310fdee674f6).

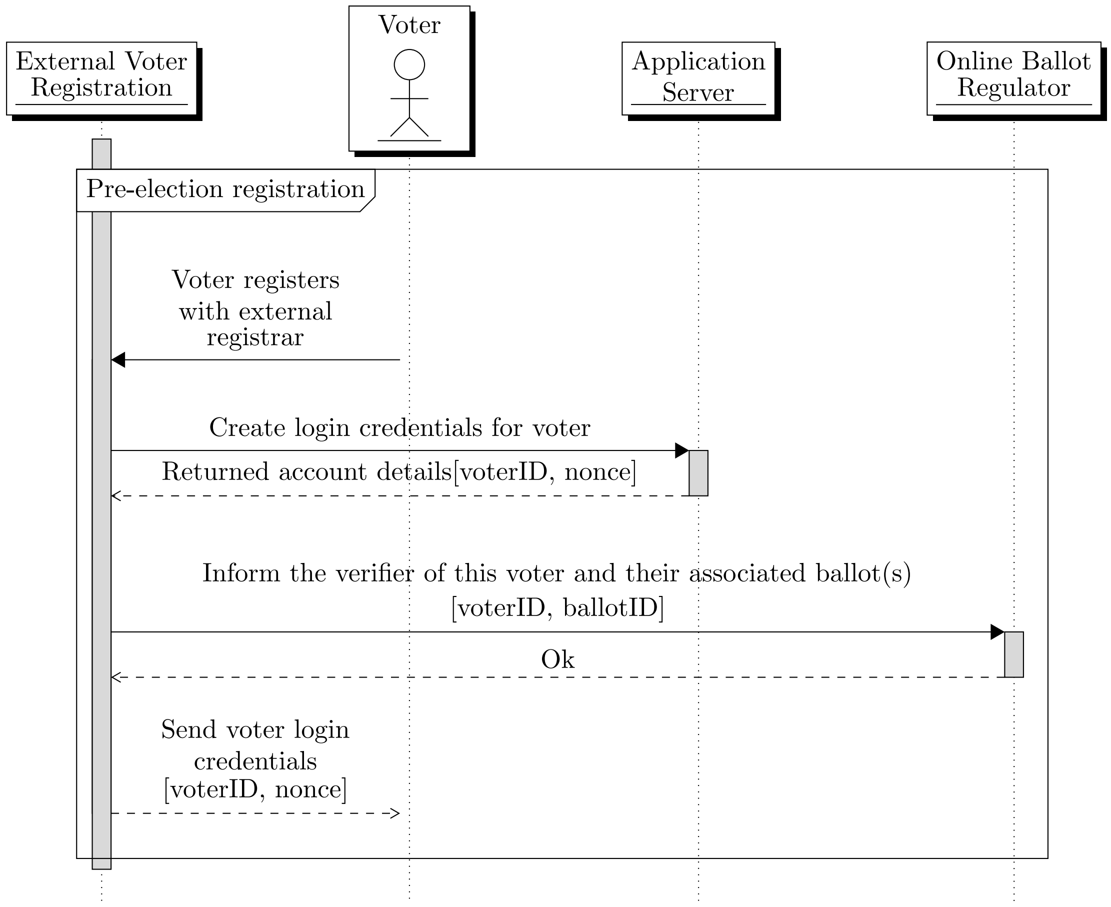
Next we send another transaction for each ballot option, calling an internal method of the contract, to add each of the options to the deployed contract (you can see examples as the 2nd and 3rd transactions in the above link). These options are then immutably added as choices of the ballot.

The final transaction to the contract is to the internal 'finalize()' method. After which, no more ballot options can be added and any registered voters are able to cast their votes.

Once the ballot has been deployed to the blockchain the *Ballot Regulator* confirms its validity and then stores internally the ballots name and Blockchain address. This allows us to query the *Ballot Regulator* later to obtain the correct address for a specific ballot.

4.7.2 Registering voters in our system

Once we have a voter who wishes to register, and is eligible to vote in a specific ballot (or set of ballots) we need to add this user to our system so they can log in and vote.



External Voter Registration
Voter
Application Server
Online Ballot Regulator
Pre-election registration
Voter registers with external registrar
Create login credentials for voter
Returned account details[voterID, nonce]
Inform the verifier of this voter and their associated ballot(s) [voterID, ballotID]
Ok
Send voter login credentials [voterID, nonce]

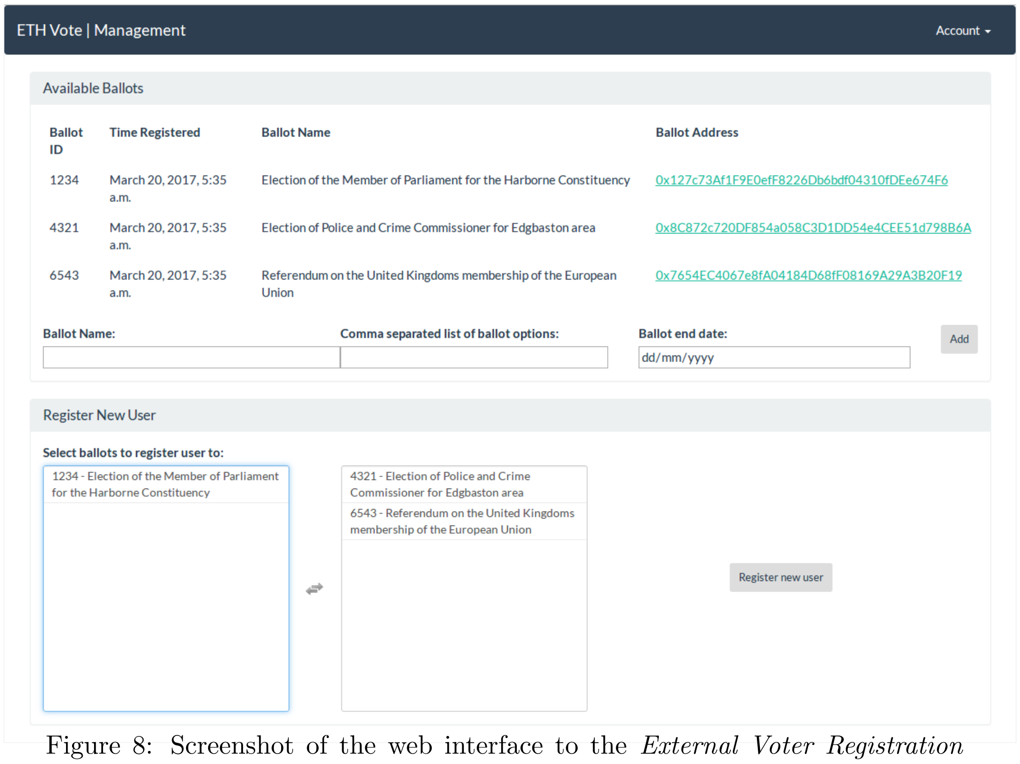
Figure 7: Sequence diagram showing order of calls when a voter is registered with our system.

The first step is validating the user requesting to register is eligible to vote. The validation process is out of the scope of this project but you could imagine this being a similar process to current election registration schemes. Therefore our system has no verification built in and allows anyone to sign up for any ballot of their choosing.

Next we request a new user account is created in the *Application Server* for our voter. A network request is sent and handled by the accounts/remote_user_add.py class which generates a new user_id and random secure password which are then passed back to the caller.

We now register the user_id for any ballots they are eligible for. This is done in another network call to the *Online Ballot Regulator* and is handled in the onlineballotregulator/network_request.py class. A database entry is created linking the userID to a ballotID which is used later to verify which ballots a logged in user is eligible for.

Finally the login credentials are sent securely to the user using an applicable method. For a more traditional registration system, this could be sent in the post similar to how you receive a credit card and pin number (separate letters). It would be possible to encode this information into a QR code format so that the end user need simply scan their received credentials to first log into the system. If the voter validation process was online based, i.e. allowing users to upload their identity documents for automatic processing, we could respond with the users login details almost instantly just like signing up to any secure website (the risks here are reduced as the user is required to change their password on first login anyway).



Ballot ID	Time Registered	Ballot Name	Ballot Address
1234	March 20, 2017, 5:35 a.m.	Election of the Member of Parliament for the Harborne Constituency	0x127c73Af1F9E0efF8226Db6bdf04310fDEe674F6
4321	March 20, 2017, 5:35 a.m.	Election of Police and Crime Commissioner for Edgbaston area	0x8C872c720DF854a058C3D1DD54e4CEE51d798B6A
6543	March 20, 2017, 5:35 a.m.	Referendum on the United Kingdoms membership of the European Union	0x7654EC4067e8fA04184D68fF08169A29A3B20F19

Ballot Name:
Comma separated list of ballot options:
Ballot end date: dd/mm/yyyy

Select ballots to register user to:

1234 - Election of the Member of Parliament for the Harborne Constituency

4321 - Election of Police and Crime Commissioner for Edgbaston area
6543 - Referendum on the United Kingdoms membership of the European Union

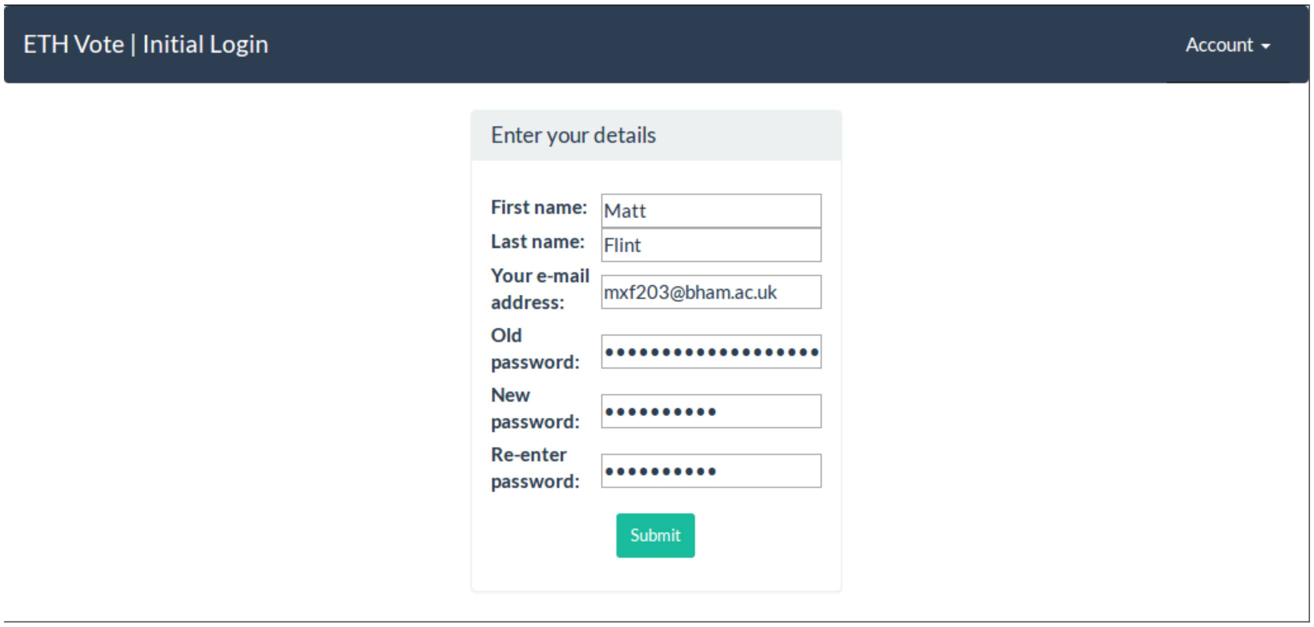
Figure 8: Screenshot of the web interface to the *External Voter Registration* showing the previously created ballots at the top and the ability to register a new user (to a set of ballots) at the bottom.

4.8 During the Election

4.8.1 First login

Once a user has requested to be registered into the system, and received their initial login credentials, they are able to access the service and log in. Before they are able to access any site content, we enforce a password change and basic user information to be entered. This is for security reasons and is good practice for web applications.

This enforcement is handled by some extra Django middleware in the "Application Server" (accounts/middleware.py) which will not allow access to any internal pages unless the user has already entered their initial information. It does this by checking, before the display of any internal page, if there is a 'requires initial information' flag set in the user database and if so refuses to display the requested page and redirects to the initial information request page.



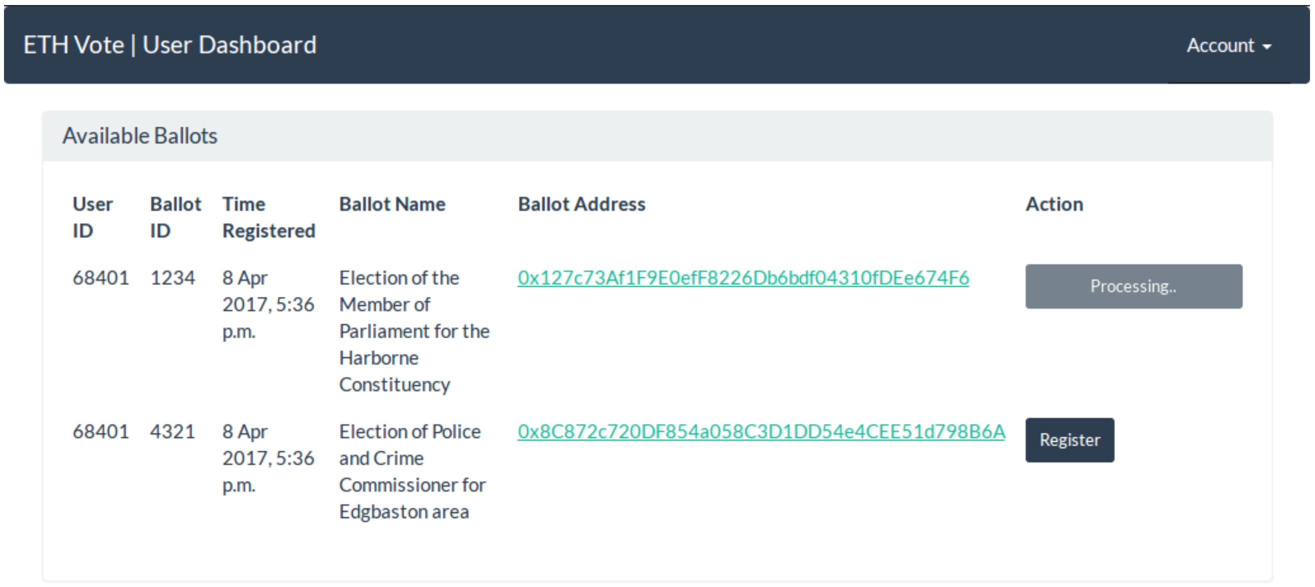
First name:	Matt
Last name:	Flint
Your e-mail address:	mxf203@bham.ac.uk
Old password:	•••••••••••••••••••
New password:	••••••••••
Re-enter password:	••••••••••

Figure 9: Initial login information entry page.

As the system I produced is only proof-of-concept I have not enforced entry of much information besides an email address and the password reset. If this system was deployed in an actual election you could request for more options to be set here such as contact preferences, display preferences (visually impaired mode) or further security options (maybe setting up two-factor authentication). Once the user has entered the required information they are able to proceed further into the system and access the user dashboard.

4.8.2 Online Registration

The user dashboard page is the starting place for any user interaction with the system. Here, a list of ballots the user has been approved to vote in is shown along with their corresponding Ethereum address and associated information such as the current user registration state. Users are presented with a link to an external blockchain viewer showing the transactions of each contract that they can use to independently verify a contracts validity if they wish.



Available Ballots

User ID	Ballot ID	Time Registered	Ballot Name	Ballot Address	Action
68401	1234	8 Apr 2017, 5:36 p.m.	Election of the Member of Parliament for the Harborne Constituency	0x127c73Af1F9E0efF8226Db6bdf04310fDEe674F6	Processing..
68401	4321	8 Apr 2017, 5:36 p.m.	Election of Police and Crime Commissioner for Edgbaston area	0x8C872c720DF854a058C3D1DD54e4CEE51d798B6A	Register

Figure 10: User dashboard showing ballot information along with the users registration state for each ballot.

In order for the user to engage in voting on a ballot we require them to 'register' with that ballot. This will start the process of generating a new Ethereum address that will be allowed to interact with the blockchain contract. Note that we could do this automatically without user interaction (possibly after the initial login information has been entered) but I have chosen to offer this as a distinct step which must be invoked in this proof-of-concept system. This is because its easier to demonstrate the separate process of registering a user to a system account to that of registering an address to a deployed ballot contract. In a real world system there would be no need to show this step to the user, as it could cause confusion about what is being registered, and it could easily be abstracted away.

When the user clicks the button to register on a ballot contract we begin the process of generating an Ethereum address for the voter to use, giving it some Ether and allowing it to vote on the selected contract.

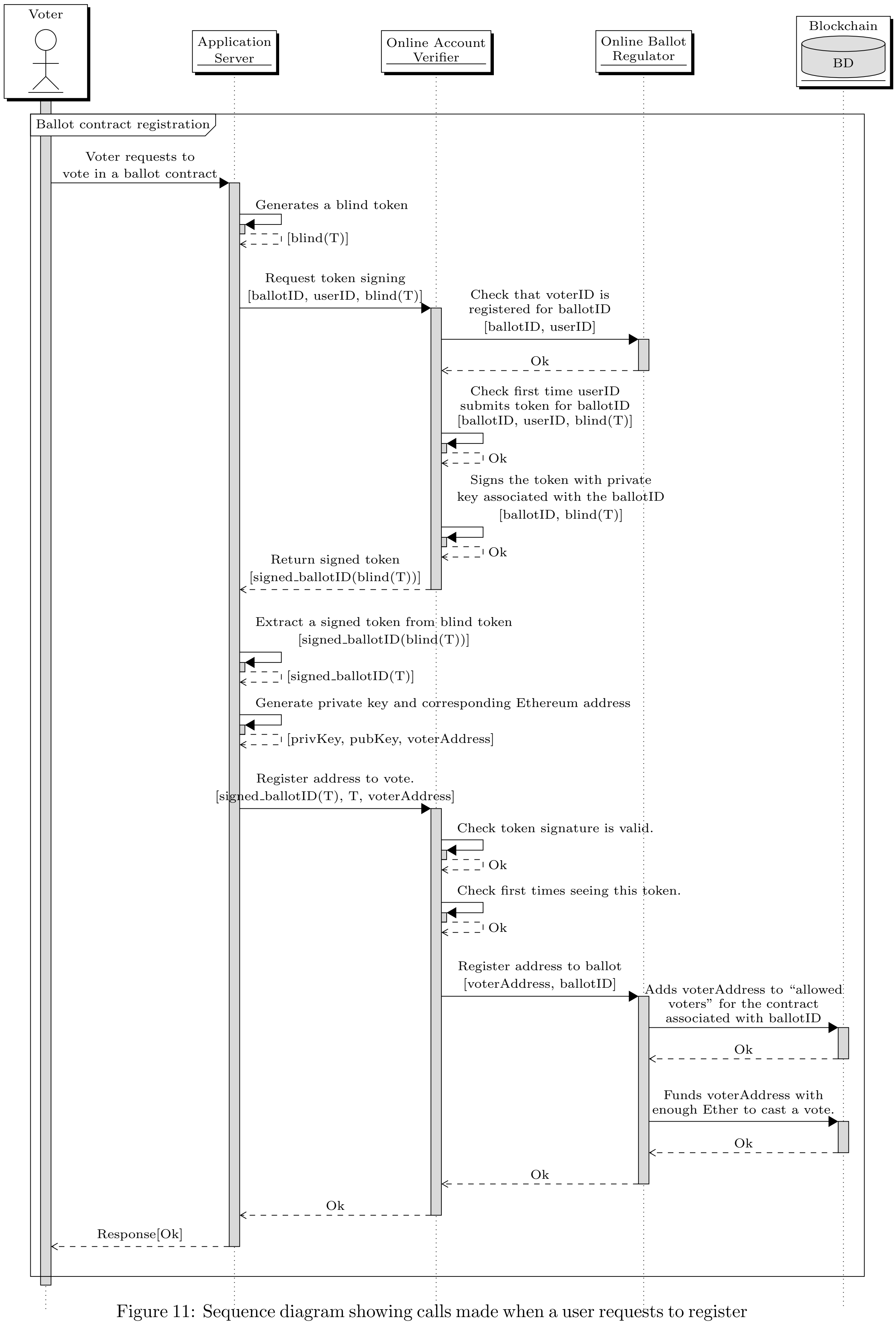

Voter
Application Server
Online Account Verifier
Online Ballot Regulator
Blockchain
BD
Ballot contract registration
Voter requests to vote in a ballot contract
Generates a blind token
[blind(T)]
Request token signing [ballotID, userID, blind(T)]
Check that voterID is registered for ballotID [ballotID, userID]
Ok
Check first time userID submits token for ballotID [ballotID, userID, blind(T)]
Ok
Signs the token with private key associated with the ballotID [ballotID, blind(T)]
Ok
Return signed token [signed_ballotID(blind(T))]
Extract a signed token from blind token [signed_ballotID(blind(T))]
[signed_ballotID(T)]
Generate private key and corresponding Ethereum address
[privKey, pubKey, voterAddress]
Register address to vote. [signed_ballotID(T), T, voterAddress]
Check token signature is valid.
Ok
Check first times seeing this token.
Ok
Register address to ballot [voterAddress, ballotID]
Adds voterAddress to “allowed voters” for the contract associated with ballotID
Ok
Funds voterAddress with enough Ether to cast a vote.
Ok
Ok
Ok
Response[Ok]

Figure 11: Sequence diagram showing calls made when a user requests to register with the ballot contract in the blockchain.

The process of allowing a voter to interact with a deployed ballot contract is quite involved and is designed to anonymize (to the system and external parties) a voterAddress while still being able to verify that the address is being supplied by a user who is allowed to vote. This verified yet anonymous status is achieved through the use of blind signatures on tokens.

I've used the concept of blind signatures in this system to send a randomized Token to the "Online Account Verifier" for them to sign. The singing in my system is accomplished using RSA keys and the PyCrypto library [38] which abstracts away most of the mathematics and allows the generation of a blind message (see client implementation in "Application Server" user_ballot_registration/views.py and the corresponding server code at "Online Account Verifier" at signatures/token_request.py). The "Online Account verifier" has a unique key pair for each ballot that is registered in the system, this means that any tokens signed are valid only as identification for the specific ballot they were requested for.

The order of events for a user to register an Ethereum address to vote is as follows: firstly, the "Application Server" generates a random token to be used in the interaction. This is then blinded with a randomly generated number (as explained in the Blind Signatures 4.4 section) and the public key of the ballot we are requesting to add the address to. Next we send this blinded token across to the "Online Account Verifier" to be signed.

When the request to sign the blind token is received by the "Online Account Verifier" we initially run a few checks. First, we check to see if the voter requesting to be registered for a ballot is indeed eligible to vote. Secondly, we check that this is the first time we are seeing this user request a token signature for this particular ballot. These checks ensure that users can only register for ballots they are eligible for and each address can only register one address per ballot. If all of our checks pass then we sign this blinded token with the associated ballots private key and return it to the "Application Server".

The "Application Server" now has a signed, blinded token (the contents of which have not yet been seen by the "Online Ballot Regulator") which we can unblind to reveal a signature of the raw token by the "Online Ballot Regulator". We now generate and store an ECDSA keypair [39] which we can use to derive the Ethereum address the voter will use to interact with the ballot in the Blockchain. The token, token signature and voterAddress are then sent back to the "Online Account Verifier" to be verified before being added to the ballot contract.

This is now the first time the "Online Account Verifier" is seeing the token and, as the message is not accompanied by any user_id, is unable to link this request to register into the ballot contract to a user. The system can however verify that this request is legitimate by checking the signature of the token, as only a valid voter is able to obtain a signature through the system we can verify that this request is from a genuine voter and should be allowed to proceed. First we check that this is the first time we are seeing this token and signature (so that the same token cannot be used multiple times) before sending a request to the "Online Ballot Regulator" to insert the voterAddress into the ballot contracts list of eligible voters.

As the "Online Ballot Regulator" is in control of the Ethereum address that created the ballot contract the node is the only one with the ability to add new voters to the contact (see Blockchain Ballot Contract 4.6.5 section). We create a new transaction calling the *giveRightToVote()* method of the ballot contract with the voterAddress as a parameter. Once this transaction is confirmed the state change in the ballot contract will mean that, when the voter with the keys to the voterAddress sends a transaction to call the *vote()* method in the ballot, they will be allowed to add their cast vote to the contract. At the same time we also fund the voterAddress with enough Ether to be able to fund their voting transactions (see ethereum/ethereum.py for the relevant code).

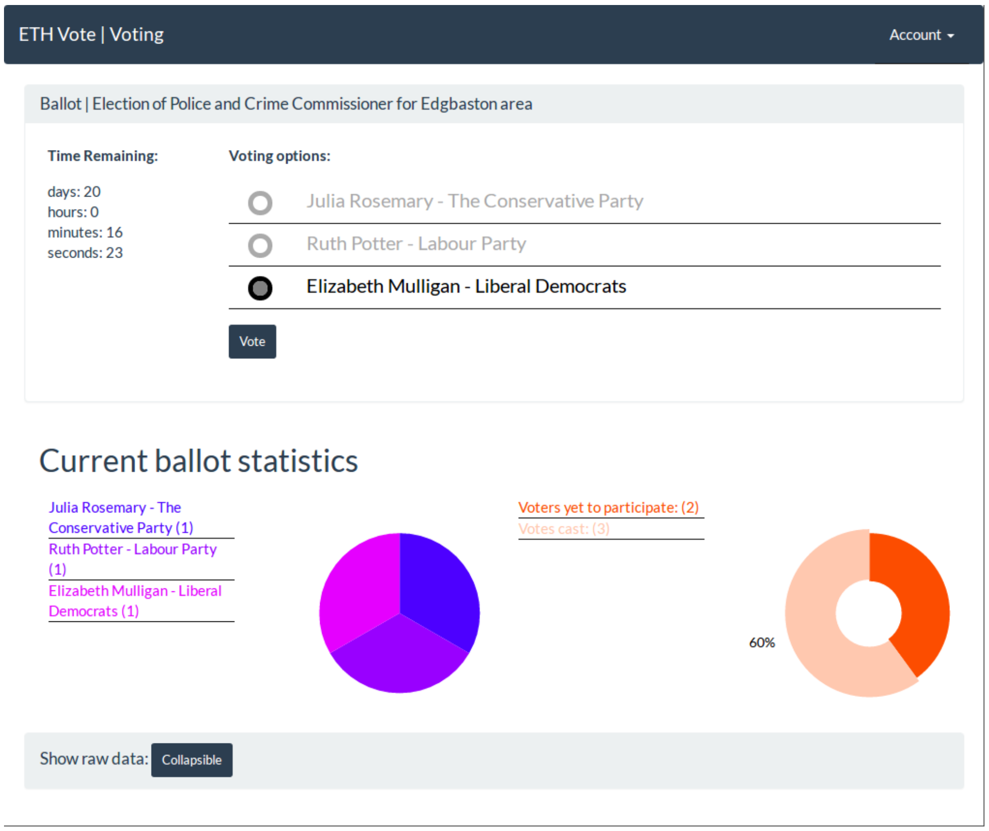
In short, this is how we are able to verify that an Ethereum address is eligible to vote without revealing the user behind the private key.

4.8.3 Voting

Once a user has been registered with a deployed ballot contract they are then able to cast their vote into the blockchain. Their registered status will be reflected in the dashboard as the 'register' button will change to a 'vote' button allowing them to participate in the ballot.

The process of casting a vote is very simple, we call the *vote()* method of the ballot contract with a funded Ethereum transaction and the voting selection. Because the Ethereum address used to call this method is the one belonging to the user (and has already gone through the 'online registration' process) we are allowed to submit our vote to the contract and have it counted.

The process of a user casting a vote in the system is as follows; first, the user selects which ballot they wish to cast a vote in (this is from the set of available ballots shown in the dashboard). The "Application Server" sends a network request to the "Online Ballot Regulator" requesting the blockchain address for the ballot contract.


ETH Vote | Voting

Ballot | Election of Police and Crime Commissioner for Edgbaston area

Time Remaining:

days: 20
hours: 0
minutes: 16
seconds: 23

Voting options:

- Julia Rosemary - The Conservative Party
- Ruth Potter - Labour Party
- [✓ Elizabeth Mulligan - Liberal Democrats]

Current ballot statistics

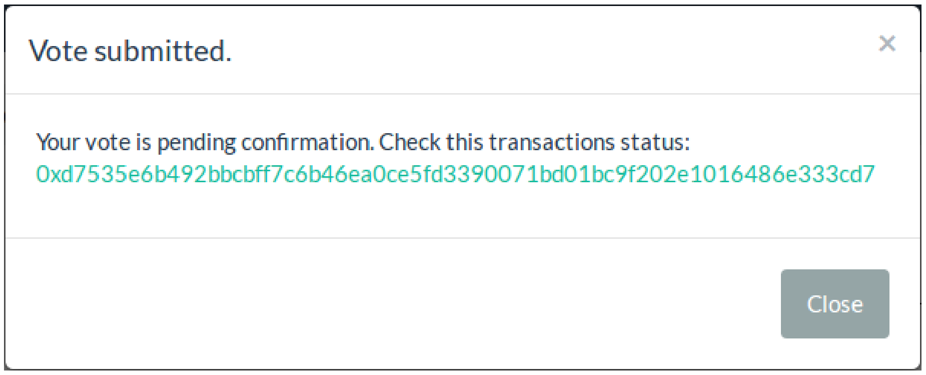
Julia Rosemary - The Conservative Party (1)
Ruth Potter - Labour Party (1)
Elizabeth Mulligan - Liberal Democrats (1)

Voters yet to participate: (2)
Votes cast: (3)

Figure 12: Ballot screen presented to the user.

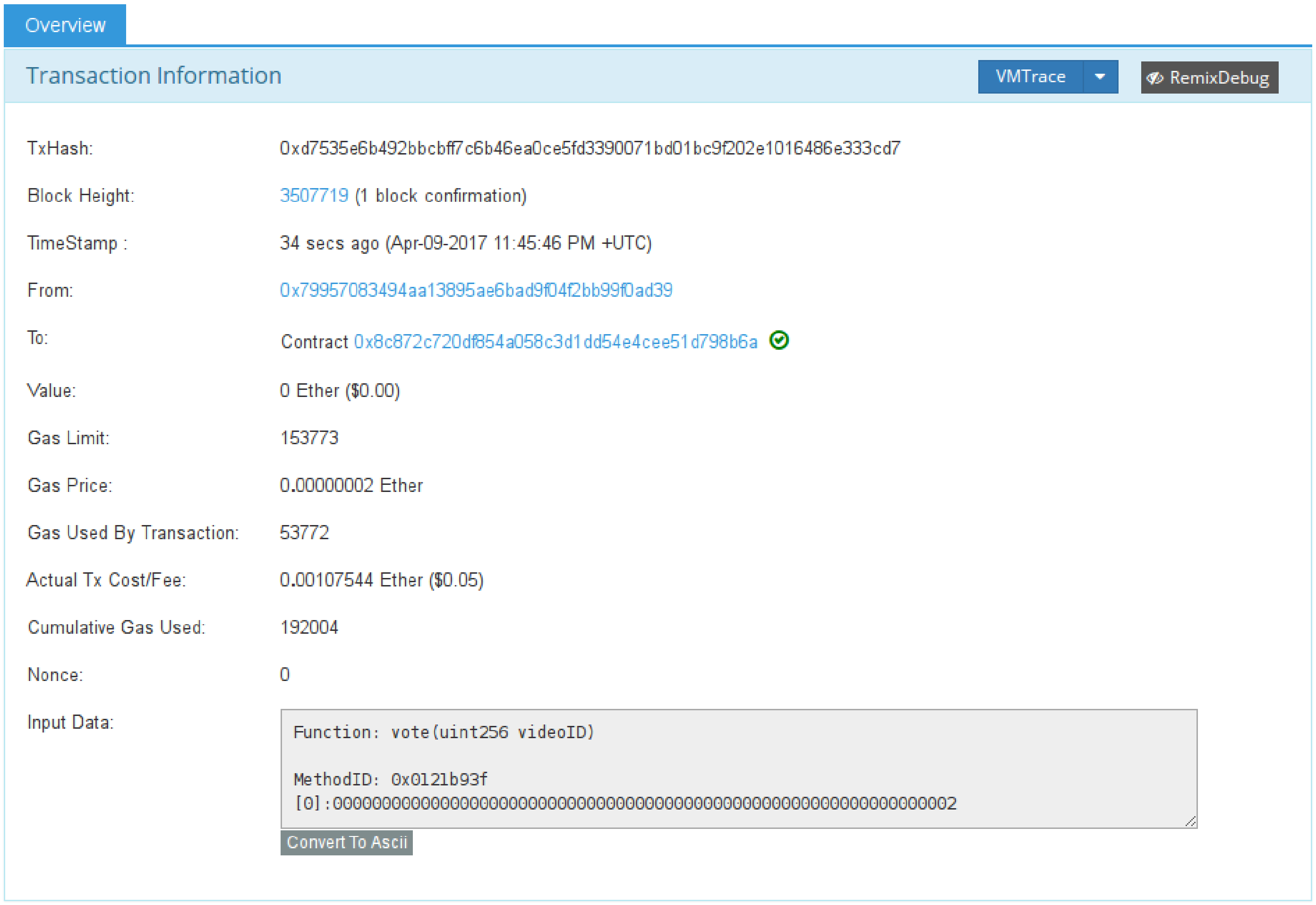
From now on all interactions with the contract are handled by the "Application Server". As we now have the contract address we can query the blockchain for this contract which returns the contracts definition along with the current 'state' of the contract. This means we can retrieve all of the latest information about the ballot directly from the blockchain, information such as 'Ballot Name', the list of candidates and the current votes for each candidate. The information available is directly related to the contract design (talked about in Blockchain Ballot Contract 4.6.5 section) but in my designed system this extends to the current votes for each candidate and the total number of registered voters. This means that we are able to display this information to the user (this can in fact be queried by anyone at any time) and use this to present relevant graphs directly onto the ballot page. At the top of the ballot we present the available candidates, as obtained from the ballot contract, as a radio button selection.

The voter now has all the available information to make their decision, once they wish to cast their vote they choose their voting options and click "submit". This generates a Ethereum transaction, funded by the users Ethereum address, which calls the *vote()* function in the ballot contract with the users voting choice as a parameter. The resulting transaction hash acts as a receipt for the voter, allowing them to verify that their vote was successfully accepted and confirmed into the blockchain. This can be done on any blockchain explorer almost instantly due to the propagation properties of the Ethereum network, the system does however provide a link to view the transaction in a trusted explorer for user convenience (e.g. 0xd7535e6b492bbcbff7c6b46ea0ce5fd3390071bd01bc9f202e1016486e333cd7).


Vote submitted.

Your vote is pending confirmation. Check this transactions status:
0xd7535e6b492bbcbff7c6b46ea0ce5fd3390071bd01bc9f202e1016486e333cd7

Figure 13: Pop-up box on submission of a vote giving the voter the transaction hash.



Transaction Information

TxHash:	0xd7535e6b492bbcbff7c6b46ea0ce5fd3390071bd01bc9f202e1016486e333cd7
Block Height:	3507719 (1 block confirmation)
TimeStamp :	34 secs ago (Apr-09-2017 11:45:46 PM +UTC)
From:	0x79957083494aa13895ae6bad9f04f2bb99f0ad39
To:	Contract 0x8c872c720df854a058c3d1dd54e4cee51d798b6a
Value:	0 Ether ($0.00)
Gas Limit:	153773
Gas Price:	0.00000002 Ether
Gas Used By Transaction:	53772
Actual Tx Cost/Fee:	0.00107544 Ether ($0.05)
Cumulative Gas Used:	192004
Nonce:	0

Input Data:

```
Function: vote(uint256 videoID)

MethodID: 0x0121b93f
[0]:0000000000000000000000000000000000000000000000000000000000000002
```

Figure 14: Example of a block explorer showing a confirmed vote transaction.

4.8.3.1 Real time results

As discussed in Blockchain Ballot Contract 4.6.5, the contract design means that up-to-date election results can be queried from the blockchain contract by anyone at any time. This has huge potential impacts on the way elections could be fought and would produces a more informed voting populous.

Say, for example, that our our system was opened for voting around the same time as postal votes (assuming that our system is the most common way for people to vote). Candidates would be able to follow the results of their last minute campaign trail, seeing how effective their campaigning was in a certain area by the real time increase in votes.

It also removes a lot of ambiguity around the speculated outcome of the election as the need to rely on opinion polls decreases and the speculated outcome can be based on 'real' votes. This could decrease the likelihood of an election wrongly being declared a 'safe outcome' which could potentially cause voters to not worry about submitting their vote (e.g. the results of the UK's EU Referendum [40]).

4.8.3.2 Changing a cast vote

Another innovation in the way elections could be held is the ability allow voters to change their cast votes in our smart contract. This is, in fact, allowed by my designed ballot contract. A voter can call the *Vote()* method as many times as they like and if they have previously voted, their vote is removed from the previous tally and added to the new one.

If deployed for a general election, this would revolutionize the way voters are able to interact with their government. Say for example, if new information relating to a candidate came to light late into the election, voters would not be locked into their uninformed choices. Under the current system, voters who submitted their vote before election day (e.g postal vote) could be misrepresented in their vote for a candidate which they cannot change.

It would also be statistically interesting to be able to analyse the swing of votes as new information came to light. This would be possible as the transaction history of previous votes would still be accessible in the blockchain, therefore we could track how a voter (or more accurately, a voterAddress) changed their opinion over time.

Alternatively this idea could be eliminated completely, we could simply have our ballot contract accept the first vote from each voter as final.

4.8.3.3 Tentative voting

If we were to mix the two innovations outlined above, we could have a brand new kind of election. This would be where voters can post tentative votes, which they reserve the right to change later, but that would allow them to express their opinion for a candidate. This would allow them to see how the general populous feels about a certain ballot without locking them down to that choice.

This would mean that decisions people might not have been willing to stand behind permanently, could be put out tentatively and then change their minds later based on the sentiments of the public. This ultimately leads to a more informed choice about who has a chance of winning an election, and voters can afford to initially choose candidates based on their merits rather than because they feel they have to vote for someone.

There are potential drawbacks with this idea; firstly, tentative votes could simply be forgotten about leading ultimately to an incorrect vote being cast by the voter (if they later on changed their mind). This is, however, no different from the currently employed system where once a vote is cast it is unchangeable.

This could also lead to last minute 'information leaks' about a certain candidate in an attempt to discredit them and cause voter swing. The legitimacy of this information need not even be accurate as, whether legitimate or illegitimate it would some voters to change their stance in favour of another candidate.

4.9 Post Election results

Assuming that you are running a full Ethereum node (that is, have all of the blockchain data available to you) querying any of these getter functions of a ballot contract will not cost you Ether. This is because the state of all of the variables in the contract can be calculated from all of the transactions (which you have locally) therefore the results can be computed without remote calls. The only transactions which you need to pay Ether on are those which perform operations in the blockchain, e.g. saving data or remote computation.

This increases the viability for external parties to audit the ballots in their entirety as if they had an attached cost, due to the size of the election being held, it could have been prohibitively expensive.

In a general election scenario, public verification would require the publishing of the address of each ballot contract prior to the election beginning. Due to the programming of the contracts this would pose no security risks while increasing transparency during the election. It would also have the benefit of allowing voters to verify that they are interacting with the correct ballot, thus increasing trust throughout the system.

In this scenario I would envisage multiple third parties creating verification systems checking the results of the election in real time through querying these ballot contracts. As the output of the election is now verifiable by anyone, we have fulfilled the "independent verification of the output" criterion which we set out to complete.

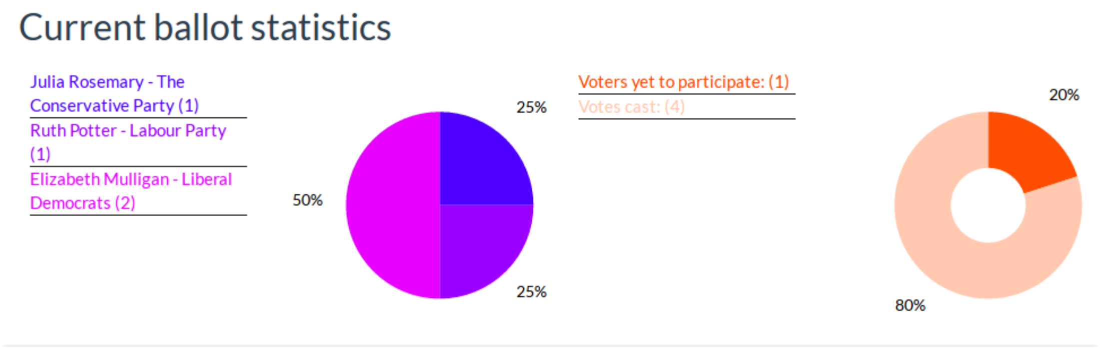


Figure 15: Example of an infographic generated from one ballot deployed to the blockchain.

5 Project Analysis

5.1 Overview

With this project I intended to create a proof-of-concept system for an 'end-to-end' verifiable voting platform. While previous sections discussed the rationale, design and implementation, I now intend to analyse the projects success and viability.

The project succeeds in presenting the protocol for an 'end-to-end verifiable system' through use of Blockchain technology. The system is *individually verifiable*, voters can check that their vote was successfully included in the Blockchain and can ensure that their vote was cast as intended (the stored result matches what the voter wanted). It is also *universally verifiable* as anyone, whether they are participating in the election or not, can verify the results of the election &, due to the systems design, be assured that each voter was only able to vote once per ballot.

The associated costs with deploying a similar system on the scale of a general election look, not only feasible, but substantially less than current means. As this estimate is tied to the traded price of Ethereum we can expect some fluctuations in the actual cost but even with an exponential increase the costs should be less than a traditional election.

Finally the scalability of the system, while not currently being able to process the vast number of transactions a general election would generate, does look like it will acquire the necessary throughput in the near future.

5.2 Financial Feasibility

Using the Ethereum network as our distributed database comes at a cost. Obviously this requires computing power as the network ensures the integrity of the Blockchain along with processing new transactions into it and these computers need to be incentivized to continue providing this service.

Is it important to understand what actually costs money in the system. You only pay for remote computation or data entry transactions so, in this system the only places we have an external cost are during poll creation and voting. Whenever we create a poll or a user votes we have to send Ether to the Ethereum network that collectively verifies your vote/poll and inserts it into the Blockchain. These are the two areas I have chosen to analyse and both cost money because we are writing data to the Blockchain.

Each transaction has a fixed cost of gas which depends on several factors, such as the amount of computational steps it requires, this number cannot be adjusted as it depends on the transaction code being executed. Each transaction can also set a 'gas limit' and a 'gas price'. The 'gas limit' is used to specify the upper bound of gas you are willing to pay for a transaction (unused gas is returned to the sender anyway) but is essentially used to ensure 'buggy code' does not deplete your Ethereum account balance [23]. The gas cost of the transaction will be bought by the ether you have in your account at a price you specified with 'gas price' Higher prices for each unit of gas means miners are more likely to run your contract or that you may overpay.

5.2.1 Ballot Deployment

Creating a new ballot requires three things (as discussed in Creating A New Ballot Contract 4.7.1) deploying the contract, adding the ballot options and finalizing the contract. More specifically, we are storing the following information into the Blockchain:

- **owner** - The address (20 bytes) of the ballot creator.
- **optionsFinalized** - Boolean value of whether the poll is set up.
- **ballotName** - Arbitrary length UTF-8 ballot name.
- **registeredVoterCount** - Integer count of registered voter addresses.
- **ballotEndTime** - Integer value of seconds since epoch.
- **votingOptions** - Dynamically sized array of the 'VotingOption' structure (this structure contains a string name and integer vote count).

As you can see we do quite a lot in a varying number of transactions (varying because we add each option in a separate transaction and we can have an unlimited number of options). I calculated the costs associated with all of the ballots I publicly deployed over the course of my testing (spreadsheet available here) and found that while the average cost of fully deploying a contract (including adding multiple ballot options and finalizing) is around £0.96 the individual cost vary considerably depending on the data being added (minimum cost was £0.79 for a 2 option ballot and maximum was £1.17 for a 6 option ballot). Changes in cost here seem to depend most on the number of options being added and the length of the string in each of those options.

Description	**Average Cost (£)**
Average cost of the initial deployment.	0.71
Average cost of adding a ballot option.	0.05
Average cost of finalizing the contract	0.02
Average cost to fully deploy a ballot (initial deployment, option addition and finalization).	0.96

Figure 16: Average cost of deployment actions (prices calculated April 2017).

5.2.2 Adding voters

The next cost incurred by the system is during the addition of voter addresses to the deployed ballot contract. Due to the fixed length of voter addresses (20 bytes) this type of transaction will always have the same amount of data being added to the blockchain; therefore we will always use the same amount of gas, 49100, for each transaction.

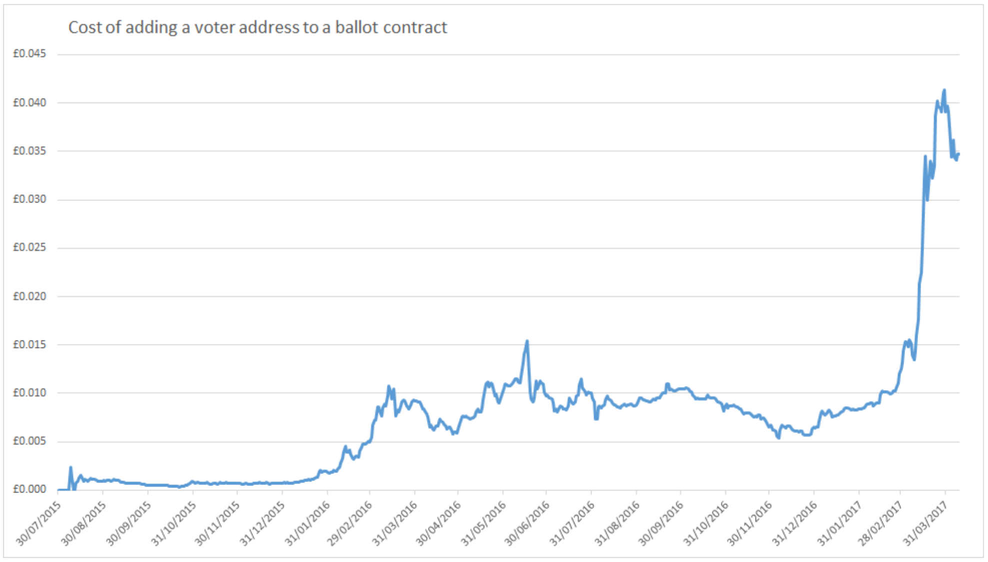


Figure 17: Cost of registering a voter address over time tracking the inflation of Ethereum.

Therefore the only cost variable here is the 'Gas Price' (the Ether price we buy a unit of gas for) which is tracked to the traded price of Ethereum. This means that the price we pay for adding a voter to a ballot contract will fluctuate over time with the price of Ethereum.

Currently (April 2017) the price of Ethereum is around £35.26 resulting in a price per address registered of £0.04. Data released under the public information act [41] states that, during the 2010 UK General Election, there were 45,597,461 registered voters on the UK Parliamentary Register. To register this many people on our system at a cost of £0.04 per voter, would equate to £1,823,898. Even if we assume that the increased rate of inflation of Ethereum continues, an estimate of more than double the current Ethereum costs (£0.10 per voter) would cost around £5 million. For perspective the cost to print, dispatch and return the postal votes for the 2010 election were around £10.6 million [42].

5.2.3 Voting

The final area of cost relating the the ballot contracts is that of voting. And additional consideration here is allowing user to change their vote, as the cost incurred each time a vote is cast but more computation needs to be done on subsequent casts (due to the removal of the previous vote allocation).

Description	**Average Cost (£)**
Average cost of the initial vote.	0.025
Average cost of re-voting.	0.045

Figure 18: Average cost of per user voting and re-voting (prices calculated April 2017).

As you can see from the average costs table, the cost of re-voting is a little under half of the cost of the initial vote meaning it's not prohibitively expensive in relation to the original voting cost.

If we calculate these prices with in relation to the registered number of voters in the 2010 election (45,597,461) [41] we find that cost for everyone to vote once would be around £1,139,936.53 and the cost of everyone submitting a subsequent vote being £2,051,885.75.

To represent this information in another form, these costs would mean (prices taken April 2017) that 1000 initial votes would cost the system £25, 1000 re-votes would cost £45, an individual casting their vote then changing their mind twice would cost £0.115 and 1000 people doing so would cost £115.

5.2.4 Cost Comparison

While initially looking at the above costs on a per-voter basis you might think that these prices are too high to be competitive with traditional voting techniques, however when broken down, we see a potentially different picture. As you might suspect, holding elections today is expensive and slow. The system is still based off of physical booths, paper ballots and requires humans to count the ballots (in some cases there are machines counting the ballots, but this is often more expensive than humans on a per voter cost basis [43].)

To compare this Ethereum-based voting system to the current voting system I'm using data from the 2010 UK General Election where the vote turnout was at 65.1%. Therefore, out of 45,597,461 registered voters 29,991,471 actually voted [44]. The estimated the cost of this general election was £113,255,271 [42] of which £28,655,271 was the cost of distributing candidates mailings and a further £84.6 million was for the conduct of the poll. This means that, on average, it cost about £3.77 to register and count a vote but if we were to only take the costs for conducting the poll (£84.6 million), it would cost £2.82 to cast a single vote.

Therefore we can compare the cost of using our Ethereum based voting system to the traditional system for this election. Remember that the average cost of the initial vote in our system was £0.025, which means that if all of the voters were to use our system it would cost a total of £749,786.75 to process these votes into the blockchain. Whilst this number might not be completely correct as we are presuming a lot (such as the price of Ethereum being the same, the fact that all of these votes would be using our system, and not counting the costs of running the backend of this system), it does put into perspective the potential cost savings when compared to traditional means.

To compare, the traditional voting system would be around 3800% (38x) more expensive than a blockchain voting system in this scenario. This breaks down to the cost of casting each vote being £2.82 in the traditional system to £0.025 in our system.

With this margin of cost savings we could afford to let every voter re-vote up to 21 times and still cost less that the traditional system. This is likely an underestimate of the actual number as its unlikely everyone would choose to re-vote.

5.2.5 Cost minimization strategy

In Ethereum, each instruction in the Turing complete scripting language has an associated cost when executed, along with additional costs for writes to the Blockchain. These costs are counted in gas and the user specifies the price per gas unit that they are willing to pay. Permanent storage to the Blockchain dominate transaction costs, it currently costs 3 gas to perform an addition but 20,000 gas to store a single 256-bit integer.

As each operation consumes a calculable amount of gas we can aim to minimize the gas used through careful contract design. Simplifying the contract as much as possible is a good way to start but we could also look into more subtle ways to reduce costs such as choosing storing the voters ballot choice as an index (integer) rather than a string.

We can also look at the contract at a higher level, for example the contract I have written for this system uses multiple transactions to setup the contract (one for the initial deployment, one for each ballot option and a one to finalize). I was unable (at the time of writing the contract but future language updates should fix this) to send an arbitrary number of strings in the constructor of the contract due to current limitations of the EVM. Therefore I settled on using multiple transactions to maintain this functionality rather than compromise the contracts usability. Once the EVM has been updated and is able to handle this sort of information, we could compress this into one singular call to deploy the contract with all of the ballot options minimizing the cost of setting up a ballot.

There is also the potential for analysis of the optimal gas price to set for each transaction. As miners can choose which transactions to accept based on the amount of computation and the gas price of the transaction, we could work out the most effective price to use at a given moment in time to minimize both the cost to us and the transaction confirmation time. Though the savings per transaction would likely be minimal, when scaled to the numbers involved in a general election the compounded savings could be of reasonable note.

5.3 Scalability

Throughout this paper I have been working on the assumption that the Ethereum network would be able to handle the amount of transaction volume a general election would bring.

At the current gas limit of 4,019,884, at the cost of 42182 gas per initial vote, we could only register some 95 votes every 12 seconds. That is about 684,000 in a day. Therefore, holding the General Elections on the Ethereum Blockchain would take around 43 days to complete. This is of course not acceptable and therefore can say that, at the current time, the Ethereum Network does not have the capacity to scale to this level. This problem is more general than the Ethereum network and is currently inherent to blockchain technologies, the same scalability problems currently exist across all other protocols.

However, Ethereum has some scaling properties already built in. While there is a 'Block Gas Limit' governing how many transactions can be confirmed in a single block, this limit (unlike other protocols such as Bitcoin) is not fixed and does in fact have the ability to adjust after every confirmed transaction. Miners can choose to change the gas limit $\pm\frac{1}{1024}$ depending on the networks gas consumption at a given time. Block gas limits scale indefinitely [45] so under excessive demand we can expect to see more transactions per block.

A more recent and promising development was that Vitalik Buterin, founder of Ethereum, has laid out an ambitious roadmap with "unlimited" transaction scalability within two years [46]. The changes have been outlines in a yellow-paper [45] and introduce the idea of 'sharding' to the Ethereum network which would ultimately mean that not every node needs to process every transaction.

"The long term goal for Ethereum 2.0 and 3.0 is for the protocol to quite literally be able to maintain a blockchain capable of processing VISA-scale transaction levels, or even several orders of magnitude higher, using a network consisting of nothing but a sufficiently large set of users running nodes on consumer laptops."
- Vitalik Buterin, 2016 [46].

The changes outlined in this paper would allow throughputs of around 10000+ tx/sec [47] which would more than cover the needs of our system. Currently, Ethereum is the only protocol who is actively working towards sharding as a solution to the scalability problems which gives further validity to the choice of using it for this system.

5.4 Privacy

Voter privacy (i.e. the inability to link a voter to a vote) is the most important property of a voting system because once it is compromised, coercion and collusion cannot be avoided and therefore no other requirement can be assured [4].

The design of the system ensures that, with the exception of the securely stored private key of each voter, there is no information retained which could link a user account (and the individual behind it) to an Ethereum address. Unfortunately, there is still a degree of trust which must be instilled in the central authority that they are running this system as described and maintaining this secrecy which many users may be unwilling to do. While open sourcing the codebase could alleviate some this there will always be some parts of the system behind closed doors though how to gain the public's trust in such a system is beyond the scope of this paper.

5.4.1 Voter Obfuscation

Due to the public nature of the votes being cast in a Blockchain, we need to be aware of the possibility for a voter to inadvertently leave a trail which could be followed and directly identify them. The main concerns here would be based around some form of statistical analysis being able to discern potentially identifying traits of a voters Ethereum address.

5.4.1.1 New voting address per ballot

One possible way for an address to become less anonymized would be if it was used for multiple ballots. For example, if you could see that an address voted in a particular regional ballot and that they were the only ones to vote for a particular candidate while being externally vocal about their opinions on this particular ballot. You could determine who that individual was and then, because they used the same address for other ballots, see other voting choices.

The system actually takes steps to eliminate this possibility. We generate a new, unique voting address for each ballot the voter registered to participate in. As address creation is very cheap and the address space is incredibly large, it makes sense to generate a new address each time while incurring almost no additional overhead. This means that there is no chance to link addresses across ballots, as each address is only used for a single ballot.

5.4.1.2 Transaction timings

Another possibility for statistical analysis could be found in the timing of transactions being submitted. For example while testing the system, as there was very few users actively voting concurrently, it was easy to determine which transactions were mine as they were the only ones being sent. Applying this more generally, maybe an ISP could see the times you accessed the voting web interface and if there were sufficiently few transactions during that time we could trivially link your voting choice to you.

While this problem is significantly reduced when the system is used by a larger number of people, we could also employ tactics such as modulating the 'gas price' of the transaction to increase the amount of time between it being submitted and confirmed. Whilst its desirable to be able to see your vote as 'confirmed' almost instantly, it is not strictly necessary. As long as you can see that is has been included in the pool of unconfirmed transactions you can assume with relative certainty that is will be accepted at some point in the future.

5.4.2 Voter coercion

Though illegal the removal the guaranteed secrecy of a polling booth does open the possibility for increased levels of voter coercion when using this system.

5.4.2.1 Vote buying

The transparency of this system undermines the basis of vote buying. To ensure that the purchased votes were not merely promised, the vote buyer would be compelled to collect vote receipts and to record the identities of the sellers. He would also have to ask the vote seller to forgo payment until after the election results had been finalized (as a voter can change their vote up to the election deadline). Therefore as the sellers would have no means of enforcing the completion of the transaction the low level of trust between buyer and seller would make this form of vote buying unlikely.

Even if a valid voting receipt, in the form of a transaction hash, was presented for payment after the election it would similarly be of no value. Since all votes are publicly broadcast, anyone could submit any transaction hash that was cast in favour of a particular candidate as this receipt is proof that a ballot has been cast, but gives no indication who cast it, a buyer is unlikely to pay in this scenario.

5.4.2.2 Forced voting

One concern unique to this way of voting, is the guaranteed return of a receipt for each vote cast in the form of the transaction hash. The returning of this hash is inescapable, as its a core part of the blockchain functionality and provides the only way for a voter to validate that their transaction was included into the blockchain.

The problems of community pressure, are mitigated by the fact that (as mentioned in the previous section), a voter can provide any transaction hash as a receipt and it is not individually identifiable. Therefore requesting a receipt to prove the way an individual voted does not work.

A more realistic problem however is, due to the lack of a polling booth, individuals can be forcibly made to vote in a particular way. I was unable to think of a direct way to combat this, however I propose a method for a voter to discreetly mark these forced ballots as such and for them to not be counted in the ballot tally.

As the user is required to enter a password to unlock the Ethereum account before a transaction can be authorized, I would introduce the idea of a 'panic password' at this stage. Panic passwords gained traction as a safety measure for ATM transactions. It is a special password or set of actions which the user can trigger to alert the server that the user is under duress. In this system it could be that the user enters their password in reverse to trigger the panic status.

Once this occurs, the system continues as usual processing the transaction and providing a receipt except with the addition of a flag as part of the transaction which ultimately invalidates this vote in the contract. This would mean the person forcing the vote would be none the wiser as externally it would appear the same as a regular vote.

5.5 System security

5.5.1 The Ethereum Network

With the scope of this project aiming to design a secure voting system, possibly usable in a general election, it's essential to confirm the level of security and to understand the risks involved with the Etherum network.

Ethereum while not branded as a cryptocurrency, is often traded on cryptocurrency exchanges and as the name suggests, cryptography is a central part of the protocol. Ethereum makes use of the KECCAK-256 cryptographic hashing function (essentially SHA-3 before standardization).

Signatures in Ethereum are performed exactly the same as in conventional public-key cryptography. We can verify that Alice is the owner of a transaction because she has signed it with her private key, which can then be verified with her public key. Ethereum uses the ECDSA elliptic curve for generating the key pairs which appears to have been chosen for possible speed optimizations in the future [48]. Attacks here could involve breaking the underlying elliptic curve cryptography either by solving the discrete logarithm problem (which could be possible with quantum computers, but there is currently no efficient non-quantum algorithm), or by finding vulnerabilities in the specific curve chosen.

As with most public-key cryptography, Ethereum does not sign the entirety of the transaction as this would be too expensive. Instead, it signs a hash which can then be checked against the transaction to verify its integrity. If we could break the hash function, it could be possible to generate a secondary input transaction to hash to the same value as an original. This could then be used to perform a signature replay attack to forge a transaction.

The likelihood of either of these two scenarios being immediate threats is however, relatively low. Ethereum uses industry standard technology and a breach in any of the underlying cryptography would undermine a large proportion of the security we enjoy on the internet.

Despite the underlying cryptography being secure, there are still a number of possible attacks which could be utilised against the network. While a dishonest miner cannot generate Ether (illegitimately), steal Ether from an account or make payments on your behalf (pretend to be you), they could delay or refuse the relaying of valid transactions to other nodes, attempt to create blocks which exclude specific transactions of their choosing or attempt to create a longer blockchain that would render previously accepted blocks invalid [49]. However all of the above require the attacker to have sufficient block creation power, effectively a 51% attack on the network. When an individual or group owns more than half of the network they could produce enough "computational work" to convince others that their blockchain is the best choice [48].

This became a real threat on the Bitcoin network in January of 2014 as the mining group *Gash.io* started to approach 50% of the mining power of the entire network. At one point the group had collectively solved 42% of the blocks in a 24 hour period [50]. The situation was resolved without incident, due to miners leaving *Gash.io* for smaller pools, as well as the pools own decision to stop accepting new miners [48].

5.5.2 Voter key management

As with the various types of Bitcoin and Ethereum wallet software available there is always a trade-off between user managing their own keys and online services doing it for them. Storing private keys on behalf of users is a very dangerous thing to do, especially if we are talking about a general election which would be of great interest to hackers. However, the truth of the matter is that the vast majority of people lack the skills needed to properly protect such information by themselves.

For this reason the system should be designed to securely store the encrypted private keys of each user in its internal database. This encryption should be of sufficient strength, AES-256, and while being password based we should enforce a different password than the one a voter uses to log into the system (as this could present a risk that if the server becomes compromised an attacker would trivially be able to affect the votes in the blockchain).

This is a difficult problem to solve, companies such as LastPass get around this by only ever seeing an encrypted version of your private data (as encryption and decryption happen client side). This approach opens the user to the possibility of being compromised and also would also require sending the private key over the wire to authenticate a transaction (though this concern may be mitigated with secured connections).

A potential alternative could be to enforce the user to print out their private key (similarly to a paper wallet) in the form of a QR code. This could then be scanned in when necessary to unlock the account and authorize a transaction. Although this approach might reduce the chances of keys on a voters computer being compromised, it still places the security of the voters ballot in the voters hands.

5.5.3 Loss of Ether

Because we fund each users account with enough Ether to submit a transaction vote, there is potential for a large scale loss of Ether either to occur. This could be through over supplying ether to accounts or by users attempting to game the system.

For the proof-of-concept, the system currently supplied 0.005 Ether to each registered account so that they may perform around 4 to 5 transactions. This is obviously not the best way to implement this when scaling to the level of a general election as, say for example the user only votes once then the remaining Ether goes unspent at cost to the system. It also incentives people to attempt to game the system as they see this as free money (albeit a very small denomination) which they may attempt to transfer into a private account.

As a potential future improvement I suggest a system where we always fund the voter address with enough Ether to conduct one additional vote. This amount should be calculated based on predicted gas and Ether prices and should include a degree of leniency in favour of over funding. When a user submits a vote, their account is automatically topped up the to required level to be able to submit another one. In this way, we eliminate the over funding problem for those accounts who were never going to re-cast their votes while still allowing those who do wish to, unlimited voting opportunity.

This system however could be easily exploited due to the slight over funding to compensate for price fluctuations. A user could re-vote multiple times to accumulate an excess of Ether which could then be siphoned off. To compensate for this, some form or rate-limiting could be employed (maybe 3 re-votes per hour) along with a check of whether the voters account has enough to cast another vote before funding it again.

5.5.4 Individual computer security

Moving an election of any type away from the security of a polling booth and on to individuals computers, phones and laptops presents many security issues. The potential for an attacker to compromise an uneducated user's computer is relatively high and an event such as a general election would undoubtedly produce more attempts to do so. As voting options are conducted through a web interface, the possibility of XSS attacks or attempts to miss-represent one voting option as another must be considered.

Therefore alongside strong secure design, as a minimum voters should be educated on basic computer security that would allow them to increase the difficulty for an attacker to compromise an individual voter.

We could also enforce certain security requirements on the user, such as running an up-to-date OS version or requiring certain security software to be installed and active. But this must not raise the barrier to entry so much that we alienate a group of the voting populous (for example the elderly might not be able to easily conform to these requirements).

5.5.5 Direct contract interaction

For the most security conscious we could allow direct contract interaction from an individual. This would likely only be undertaken by a handful of people but it would still require them to use some parts of the system to submit an address which they wish to use and have it added to the ballot contract in the blockchain. Ultimately though this would still be heavily reliant on system interaction which would be what people wishing to conduct this interaction will be most critical of.

It may be possible to move more of the inner workings of the system on to the blockchain its self. Although some form of verification will still need to happen, users could interact with a smart contract in order to have their private addresses added to the list of allowed voters. Having more of the system become transparent would be beneficial not only to these more sceptical individuals but increase trust in the system as a whole.

5.6 Improvements

Reflecting upon the project, there are a number of design decisions which I would consider doing differently next time.

5.6.1 Implementation

Firstly, I would run Geth (the Ethereum software) in its own Docker container. Currently it is being run inside the "Online Ballot Regulator" and could be easily extracted for a more modular system. Running it in its own container would further segregate the various sections of the application providing increased security and decouple the files of the "Ballot Regulator" from any files generated by Geth.

I would also serve the web application over Apache (or a similarly established software). Currently the system is running off of Django's built in webserver which, while fine for development, it is a stripped-down, very basic, single-threaded server for the purposes of development and has not gone through the required security audits or performance tests for production deployment. Transferring the server over to Apache would be trivial to setup with the WSGI interface.

5.6.2 Protocol

There is a glaring omission in the design of the ballot contract, the opportunity for voters to practice their right to suffrage. Currently the ballot contract template does not add any option for a 'non-vote' which should be included to maintain voter rights.

This could be easily implemented with minimal change to the solidity contract. We could add an extra ballot option titled 'non-vote' or similar during the finalization state of ballot creation meaning it would be automatically applied to all ballots.

I would also re-implement the way voter accounts receive funding to allow them to vote (also discussed in the Loss of Ether 5.5.3 section). Essentially I would change it to a top-up system where more Ether is allocated to your account when you vote rather than giving it all to you in one go. This reduces unnecessary allocation of funds to voters who will only use their initial vote and reduces the likelihood of an individual attempting to game the system.

6 Conclusion

Although this system has been designed and developed with the idea of a national general election in mind, the protocols and ideas involved could be applied to smaller scale ballots which wish to provide transparency in their audit. Although we wish to minimize trust in a central authority, due to the nature of these type of elections (where there needs to be some degree of voter eligibility verification), we cannot fully decentralize this system as we need to only allow those eligible the rights to vote. Despite needing to verify an individual we still need to ensure that their votes are publicly anonymous, especially given the public transactions underpinning the blockchain concept while providing the ability for an individual to verify that their vote was correctly counted.

I do not see this system as a direct "replace all" for national election voting. I believe there will still be a need for traditional voting implementations in certain situations; for example, maintaining postal vote for the elderly who may not have the technical capability or equipment for online voting. However I do think that this could be phased in along side traditional voting, eventually replacing the pre-existing e-voting systems and ultimately becoming the main way for the majority of people to choose their government.

I started this project with the goal of producing an 'end-to-end' verifiable voting platform. This project succeeds in presenting the protocol for such a system through use of Blockchain technology.

The system provides *individually verifiability.* When voters submit their vote they receive a transaction hash which they can use to check that their vote was successfully included in the Blockchain. This 'cast-as-intender verification' can only be done by the voter, as they are the only one who knows which specific transaction relates to them. While some level of voter education would be necessary for the average voter to desire and be able to check their vote was counted (as the onus of this does still rest with the voter), at worst this is no different from the currently employed systems where uses must trust their vote was submitted and counted correctly while providing benefit to those who engage in it.

The system also provides *universal verifiability* as anyone, whether they are participating in the election or not, can verify the results of the election through querying the ballot contracts. These results hold all of the desirable properties of the Blockchain such as being immutable and instantly globally distributed. Due to the underlying contract design, be assured that each Ethereum address voted at most once per ballot.

The system also manages to protect voter privacy through the use of blinded tokens. This is how the system is able to register an Ethereum address to a ballot contract, without being able to link it to an individual, but they can be assured that it belongs to *some* verified voter. Voter privacy after the election is tied to the voter, as only they know the resulting transaction hash (voting receipt) for each vote cast it is kept private at their discretion.

While the proposed system does not solve all the issues associated with electronic voting, it does provide a valuable alternative to current proprietary electronic systems and has potential use in both governmental and private organizations wishing to conduct transparent ballots.

7 Appendix

7.1 Source Archive

Alongside this report is the accompanying code for this proof-of-concept system containing everything needed for deployment. Project layout closely follows the system design 4.6 with the code for each node being contained in its own folder.

7.1.1 Building

As I have used Python for each node in my project, I was able to create a *setup* file to create the correct Python environment which can be called using *'pip3 install .'* while in the nodes' root directory.

As I am using docker, building manually is not recommended as there is a lot of environmental setup which must be done. Each nodes' project contains a *Dockerfile* which is used to setup, build and run each application. Each node folder contains a *bin* directory inside which are various executables one of which, *docker_entrypoint*, is ultimately where we start the systems but the recommended way to start an individual node is to use the *build_and_run* executable which combines these steps for ease of use.

7.1.2 Running

You can independently start each node through Docker with the *'docker run'* command (conveniently you can run *build_and_run* with the parameter *runme*).

This is unnecessary however as Docker provides a tool, Compose, which is used for defining and running multi-container Docker applications. This means I can define a *docker-compose.yml* file in the root directory which will start all of the nodes, with the correct parameters, in the correct order.

This is the recommended way to run this application so calling *'docker-compose up'* while in the root directory should be the most hassle free.

7.1.3 Notes

- **Helper Files -** There are a number of helper scripts located in the *HelperScripts* folder which allow for easy interaction with the Docker nodes while running.
- **Blockchain download -** As this runs a full Ethereum node and must download the full blockchain before it can push transactions, an initial download of around 20GB will occur on first run.

7.2 Bibliography

[1] Pierre Noizat. *Blockchain Electronic Vote*. 2016. URL: http://www.the-blockchain.com/docs/blockchain-electronic-vote.pdf (visited on 17/11/2016).

[2] Adam Ernest. *The Key To Unlocking The Black Box: Why The World Needs A Transparent Voting DAC*. 2014. URL: https://followmyvote.com/wp-content/uploads/2014/08/The-Key-To-Unlocking-The-Black-Box-Follow-My-Vote.pdf (visited on 15/12/2016).

[3] Willem Wyndham, Spencer Chen and Saurav Das. *Proposal For Secure Electronic Voting*. 2016. URL: http://www.economist.com/sites/default/files/maryland_cyber_ctr.pdf (visited on 15/12/2016).

[4] Safevote. *Voting System Requirements*. 2001. URL: http://www.thebell.net/papers/vote-req.pdf (visited on 19/12/2016).

[5] Briony Holmes. *Blockchain voting systems offer transparency and security Brave New Coin*. 2016. URL: http://bravenewcoin.com/news/blockchain-voting-systems-offer-transparency-and-security/ (visited on 16/12/2016).

[6] 2016. URL: http://www.ncsl.org/research/elections-and-campaigns/voting-system-standards-testing-and-certification.aspx (visited on 17/12/2016).

[7] Sabina Petride. *Security Properties for Electronic Voting*. 2016. URL: http://www.cs.cornell.edu/courses/cs513/2002sp/proj.00.StuSolns/sp2580.htm (visited on 18/12/2016).

[8] Alex Escala et al. *Universal Cast-as-Intended Verifiability*. 2015. URL: https://fc16.ifca.ai/voting/papers/EGHM16.pdf (visited on 21/12/2016).

[9] Eli Ben-Sasson et al. *Zerocash: Decentralized Anonymous Payments from Bitcoin*. 2014. URL: http://zerocash-project.org/media/pdf/zerocash-oakland2014.pdf (visited on 21/12/2016).

[10] 2017. URL: https://blockgeeks.com/guides/what-is-ethereum/.

[11] 2017. URL: https://steemit.com/ethereum/@najoh/ethereum-explained-for-beginners.

[12] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: https://bitcoin.org/bitcoin.pdf (visited on 02/12/2016).

[13] Satoshi Nakamoto. *Bitcoin v0.1 released*. 2009. URL: http://www.metzdowd.com/pipermail/cryptography/2009-January/014994.html (visited on 02/12/2016).

[14] 2016. URL: http://www.blockchaintechnologies.com/blockchain-definition (visited on 03/12/2016).

[15] 2009. URL: https://github.com/bitcoin/bitcoin (visited on 02/12/2016).

[16] 2016. URL: https://blockchain.info/charts/blocks-size?timespan=all (visited on 03/12/2016).

[17] 2016. URL: https://bitcoin.org/en/developer-guide#proof-of-work (visited on 03/12/2016).

[18] 2016. URL: http://www.blockchaintechnologies.com/blockchain-mining (visited on 03/12/2016).

[19] 2016. URL: http://redpinata-development.com/bitcoin-academy/index.php/reader/items/proof-of-work.html (visited on 07/12/2016).

[20] 2016. URL: https://en.bitcoin.it/wiki/Confirmation (visited on 03/12/2016).

[21] 2016. URL: http://bitcoinsimplified.org/learn-more/anonymity/ (visited on 04/12/2016).

[22] Jeff Chan, Tanya Liu and Eddie Xeu. *Analyzing the Bitcoin Transaction Graph: A Look at Mixers and Traceability*. 2013. URL: https://css.csail.mit.edu/6.858/2013/projects/jeffchan-exue-tanyaliu.pdf.

[23] 2017. URL: https://ethereum.gitbooks.io/frontier-guide/content/.

[24] Scott Driscoll. *Introduction to Bitcoin and Decentralized Technology — Pluralsight*. 2016. URL: https://app.pluralsight.com/library/courses/bitcoin-decentralized-technology/table-of-contents (visited on 30/11/2016).

[25] Michael Nielsen. *How the Bitcoin protocol actually works*. 2013. URL: http://www.michaelnielsen.org/ddi/how-the-bitcoin-protocol-actually-works/ (visited on 06/12/2016).

[26] What Ethereum? *What is the Gas in Ethereum?* 2017. URL: https://www.cryptocompare.com/coins/guides/what-is-the-gas-in-ethereum/.

[27] Sebastian Peyrott and Thameera Senanayaka. *An Introduction to Ethereum and Smart Contracts: a Programmable Blockchain*. 2017. URL: https://auth0.com/blog/an-introduction-to-ethereum-and-smart-contracts-part-2/.

[28] 2017. URL: https://medium.com/zeppelin-blog/the-hitchhikers-guide-to-smart-contracts-in-ethereum-848f08001f05.

[29] Jim Kowalkowski. *Study of a Docker use-case for HEP*. 2017. URL: http://cd-docdb.fnal.gov/cgi-bin/RetrieveFile?docid=5693&filename=docker-study.pdf&version=2.

[30] Jef Scherensstraat. *Use Case: Running Docker Containers*. 2017. URL: http://www.in4it.io/usecases/use-case-docker-containers-production.pdf.

[31] 2017. URL: https://zeltser.com/security-risks-and-benefits-of-docker-application/.

[32] Matthew Flint. *Ballots and Bitcoins*. 2017. URL: https://github.com/Mattie432/Blockchain-Voting-System/raw/master/Paper/Scientific-Paper/Scientific-Paper.pdf.

[33] 2017. URL: https://github.com/ethereum/viper.

[34] 2017. URL: https://twistedmatrix.com/trac/.

[35] Mark Ryan. *Blind signatures*. 2017. URL: https://www.cs.bham.ac.uk/~mdr/teaching/modules06/netsec/lectures/blind_sigs.html.

[36] Rebecca Bellovin. *Cryptography: Authentication, Blind Signatures, and Digital Cash*. 2015. URL: http://wwwf.imperial.ac.uk/~rbellovi/writings/chaum.pdf.

[37] 2017. URL: https://github.com/pipermerriam/web3.py.
[38] 2017. URL: https://www.dlitz.net/software/pycrypto/.
[39] Vincent Kobel. *Create full Ethereum wallet, keypair and address*. 2017. URL: https://kobl.one/blog/create-full-ethereum-keypair-and-address/.
[40] Portes Curtice. *Brexit ? six months on: Public opinion ? UK in a changing Europe*. 2017. URL: http://ukandeu.ac.uk/public-opinion/.
[41] 2017. URL: https://www.whatdotheyknow.com/request/statistics_from_2010_election.
[42] 2017. URL: http://www.bbc.co.uk/news/uk-politics-24842147.
[43] Charles Arthur. *Counting the cost electronically*. 2017. URL: https://www.theguardian.com/technology/2009/sep/30/electronic-vote-counting.
[44] 2017. URL: http://www.ukpolitical.info/Turnout45.htm.
[45] Gavin Wood. *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER EIP-150 REVISION*. 2017. URL: http://gavwood.com/paper.pdf.
[46] 2017. URL: https://www.cryptocoinsnews.com/ethereum-announces-unlimited-scalability-roadmap/.
[47] 2017. URL: https://github.com/ethereum/EIPs/issues/53.
[48] Edward Z Yang. *The Cryptography of Bitcoin : Inside 736-131*. 2011. URL: http://blog.ezyang.com/2011/06/the-cryptography-of-bitcoin/ (visited on 07/12/2016).
[49] Brave New Coin. *A gentle introduction to blockchain*. 2016. URL: http://www.the-blockchain.com/docs/A-gentle-introduction-to-blockchain-technology-web.pdf (visited on 03/12/2016).
[50] Alec Liu. *Bitcoin's Fatal Flaw Was Nearly Exposed*. 2014. URL: http://motherboard.vice.com/blog/bitcoins-fatal-flaw-was-nearly-exposed (visited on 07/12/2016).