

1 Design

1.1 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 block-chain 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.

1.2 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” [1].

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) [2]. 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 [3].

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 1.6](#)), 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.

1.2.1 Why Choose 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 [23]. 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 [24] 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.

1.3 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 [4] 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.

1.4 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 [26].

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.

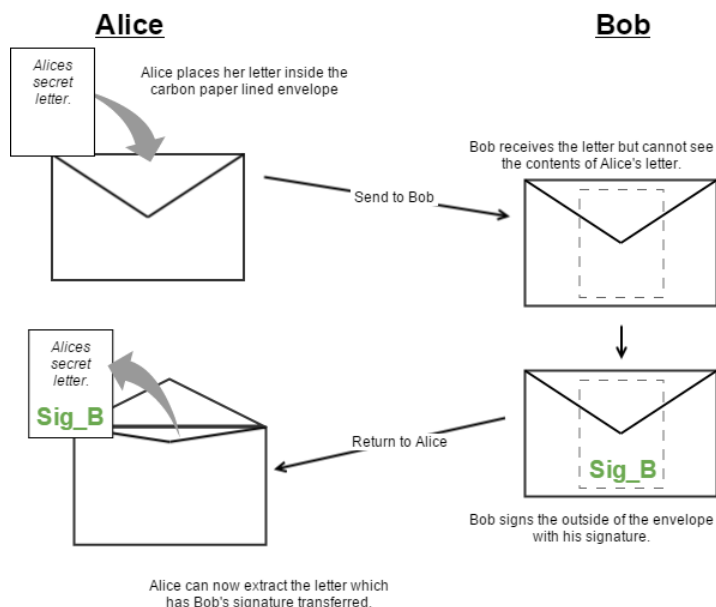


Figure 1: Blind signature analogy showing how Bob never sees the contents of Alice’s message despite being able to sign it.

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 $x^d \equiv (r^e m)^d \equiv r m^d \bmod n$, 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 [27].

1.5 Testing

As my project is written in Python and I am heavily using Django to present the fronted 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.

1.5.1 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).

1.5.2 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/utlis.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%

Figure 2: Test coverage results from the “Application Server”

1.5.3 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.

1.6 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).

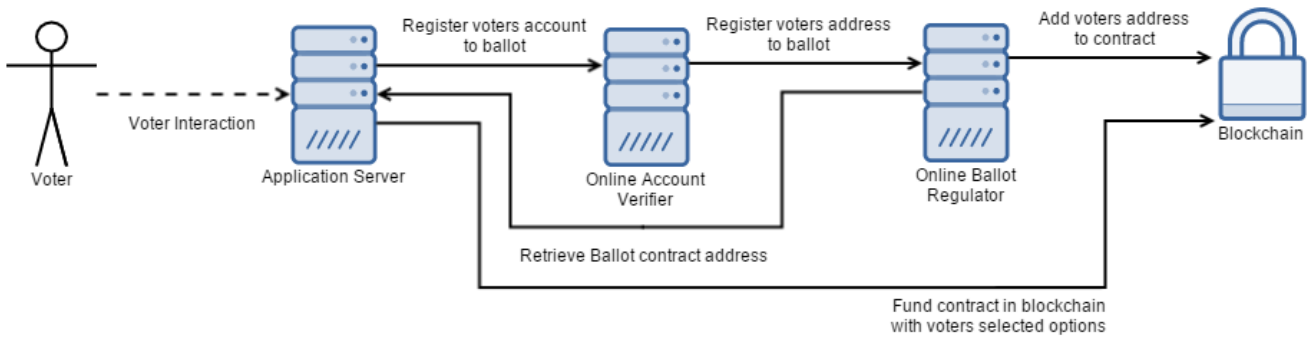


Figure 3: Outline of system showing the basic interactions between nodes.

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 1.3](#) 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 [25], 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 [4] 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.

1.6.1 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.

1.6.2 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).

1.6.3 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	54baa883f28a768d6bd352d12c7307d1592d394acea4337a018ee2d0f143642a	68401	1234	2017-04-08 18:08:49.527817
2	4451458fe4d479387fdca5dde405a16fd2fca2c4f92c516fea78b2f0d2727f7c	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	19a91bc3d91b5873e5b0e4687a988a08b362bd357a85ce3bc109cba4ed984407	0x79957083494aa13895ae6bad9f04f2bb99f0ad39	4321	2017-04-08 18:30:25.736455
1	13jad238sjkgfn39n3asd882nd2d877ad327dnk3lq9afv4b234akd3nd898hd82	0x925A8e765F9563D979b576A68210903a9968B8Be	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.

1.6.4 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	0x127c73Af1F9E0efF8226Db6bdf04310fDEe674F6	2017-04-08 18:26:23.440678	x800358071002e	1603238400
4321	Election of Police and Crime Commissioner for Edgbaston area	0x8C872c720DF854a058C3D1DD54e4CEE51d798B6A	2017-04-08 18:26:23.440678	x800358071002e	1603238400
6543	Referendum on the United Kingdoms membership of the European Union	0x7654EC4067e8fA04184D68ff08169A29A3B20F19	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.

1.6.5 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 ?? 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](https://github.com/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 1.7.1](#) 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.

```

1 // ~~~~~ Contract Constructor ~~~~~ //
2 address owner; // The address of the owner.
3 bool optionsFinalized; // If we can still add voting options
4 string ballotName; // The ballot name.
5 uint registeredVoterCount; // Count of registered addresses.
6 uint ballotEndTime; // seconds since 1970-01-01
7
8 // Modifier to only allow the owner to call a function.
9 modifier onlyOwner {
10     if( msg.sender != owner ) throw; _;
11 }
12
13 // This function is called *once* at first initialization into the
14 // blockchain.
15 function ETHVoteBallot(string _ballotName, uint _ballotEndTime)
16 {
17     if( now > _ballotEndTime )
18         throw;
19
20     // Set the owner to the address creating the contract.
21     owner = msg.sender;
22     optionsFinalized = false;
23     ballotName = _ballotName;
24     registeredVoterCount = 0;
25     ballotEndTime = _ballotEndTime;
26 }

```

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.

```

1 // ~~~~~ Ballot Options ~~~~~ //
2 // Structure which represents a single voting option for this
   ballot.
3 struct VotingOption
4 {
5     string name;    // Name of this voting option
6     uint voteCount; // Number of accumulated votes.
7 }
8
9 // dynamically sized array of 'VotingOptions'
10 VotingOption[] public votingOptions;
11
12 /*
13 * Add a new voting option for this ballot.
14 * NOTE: this can only be called by the ballot owner.
15 */
16 function addVotingOption(string _votingOptionName) onlyOwner
17 {
18     if( now > ballotEndTime) throw;
19     // Check we are allowed to add options.
20     if(optionsFinalized == true)
21         throw;
22
23     votingOptions.push(VotingOption({
24         name: _votingOptionName,
25         voteCount: 0
26     }));
27 }
28
29 /*
30 * Call this once all options have been added, this will stop
   further changes and allow votes to be cast.
31 * NOTE: this can only be called by the ballot owner.
32 */
33 function finalizeVotingOptions() onlyOwner
34 {
35     if(now > ballotEndTime) throw;
36
37     if(votingOptions.length < 2) throw;
38
39     optionsFinalized = true; // Stop the addition of more options.
40 }

```

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.

```

1 // ~~~~~ Voting Options ~~~~~ //
2 // Structure which represents a single voter.
3 struct Voter {
4     bool eligibleToVote;    // Is this voter allowed to vote?
5     bool voted;            // State of whether this voter has
6                             // voted.
7     uint votedFor;          // Index of 'votingOptions' this voter
8                             // voted for.
9 }
10 // State variable which maps any address to a 'Voter' struct.
11 mapping(address => Voter) public voters;
12
13 /*
14 * Allow an address (voter) to vote on this ballot.
15 * NOTE: this can only be called by the ballot owner.
16 */
17 function giveRightToVote(address _voter) onlyOwner {
18     if(now > ballotEndTime) throw;
19     voters[_voter].eligibleToVote = true;
20     registeredVoterCount += 1;    // Increment registered voters.
21 }
22
23 /*
24 * Allow an eligible voter to vote for their chosen votingOption.
25 * If they have already voted, then remove their vote from the
26 * previous 'votingOption' and assign it to the new one.
27 * NOTE: if anything fails during this call we will throw and
28 * automatically revert all changes.
29 */
30 function vote(uint _votingOptionIndex) {
31     if(now > ballotEndTime) throw;
32     if(optionsFinalized == false) throw; //Not finalized, cant vote
33     Voter voter = voters[msg.sender];    // Get senders Voter struct
34
35     if(voter.eligibleToVote == false) throw;
36
37     // If the voter has already voted then we need to remove their
38     // prev vote choice.
39     if(voter.voted == true)
40         votingOptions[voter.votedFor].voteCount -= 1;
41
42     voter.voted = true;
43     voter.votedFor = _votingOptionIndex;
44     votingOptions[_votingOptionIndex].voteCount += 1;
45 }

```

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.

```
1 // ~~~~~ Getter Functions ~~~~~ //
2
3 // Returns the ballots name string.
4 function getBallotName() returns (string){ ... }
5
6 // Returns the number of voting options.
7 function getVotingOptionsLength() returns (uint) { ... }
8
9 // Returns the count of registered voter addresses.
10 function getRegisteredVoterCount() returns (uint) { ... }
11
12 // Returns the name of a voting option at a specific index. Throws
    if index out of bounds.
13 function getVotingOptionsName(uint _index) returns (string) { ... }
14
15 // Returns the number of votes for a voting option at the specified
    index. Throws if index out of bounds.
16 function getVotingOptionsVoteCount(uint _index) returns (uint){...}
17
18 // Returns if the voting options have been finalized.
19 function getOptionsFinalized() returns (bool) { ... }
20
21 // Returns the end time of the ballot in seconds since epoch.
22 function getBallotEndTime() returns (uint) { ... }
```

Listing 4: Summary of getter functions.

1.7 Pre-election setup

1.7.1 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 1.6.5](#) 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”.

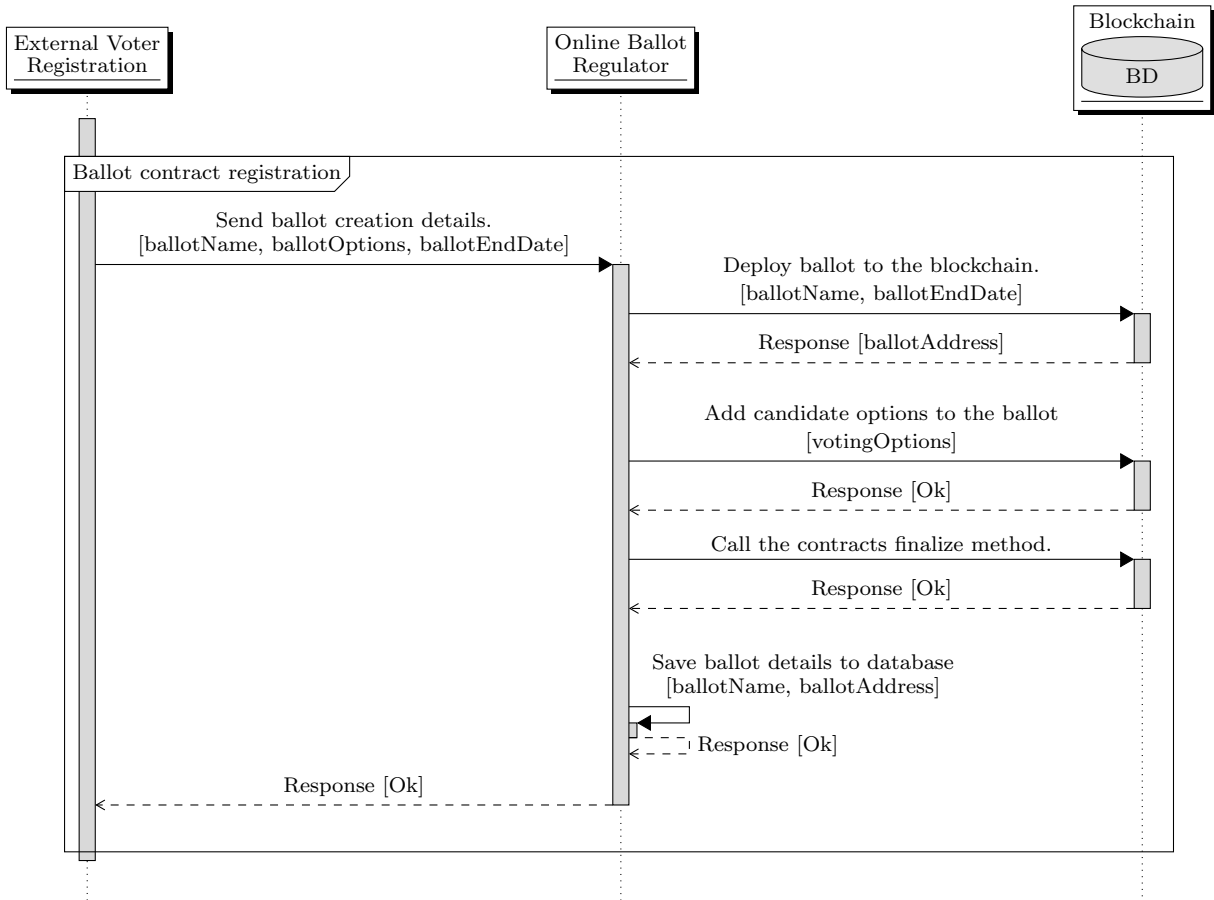


Figure 6: Registering a new ballot in the blockchain.

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.

1.7.2 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.

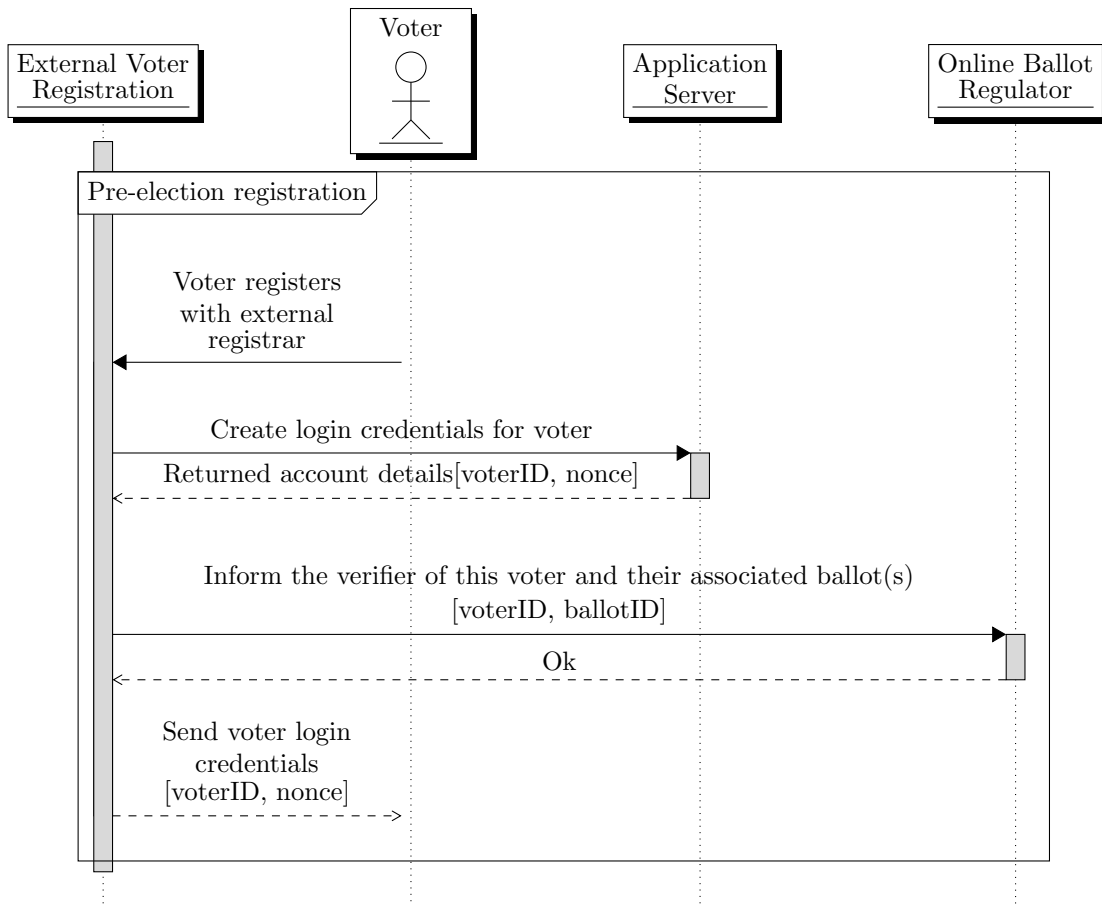


Figure 7: Sequence diagram showing order of calls when a voter is registered with our system.

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).

ETH Vote | Management Account ▾

Available Ballots

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	0x127c73Af1f9E0efF8226Db6df04310fDEe674F6
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	0x7654FC4067e8fA04184D68fF08169A29A3B20F19

Ballot Name: Comma separated list of ballot options: Ballot end date: Add

Register New User

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

Register new user

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.

1.8 During the Election

1.8.1 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.

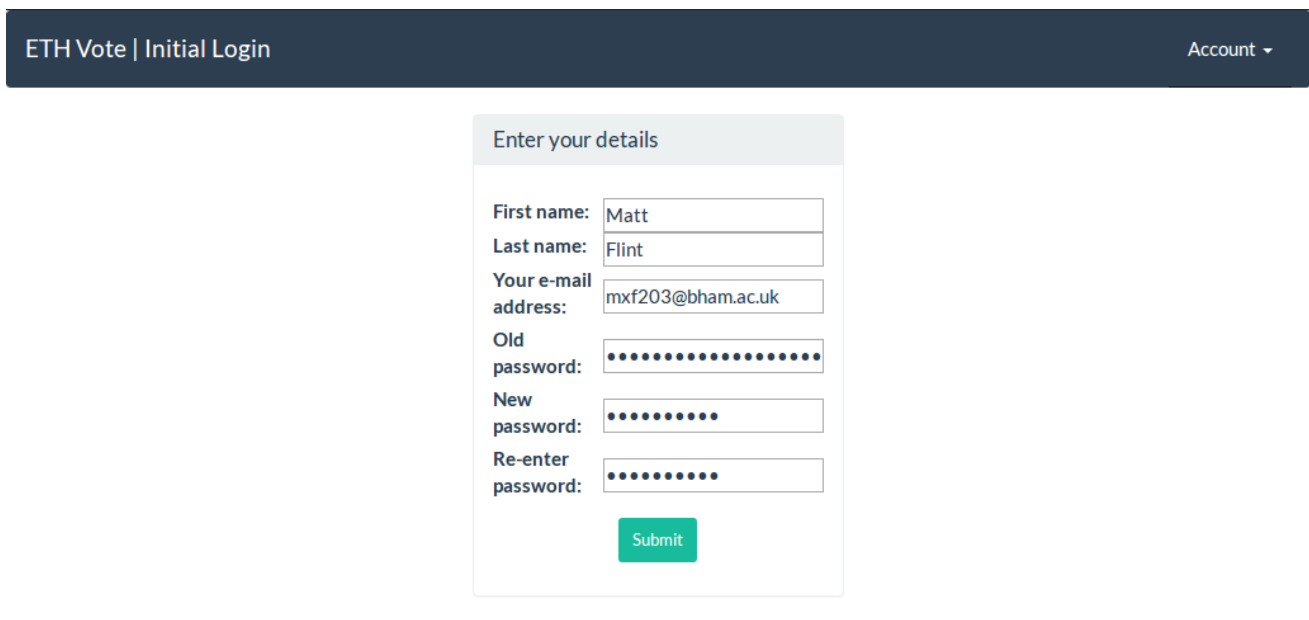


Figure 9: Initial login information entry page.

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.

1.8.2 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.

ETH Vote User Dashboard						Account ▾
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.

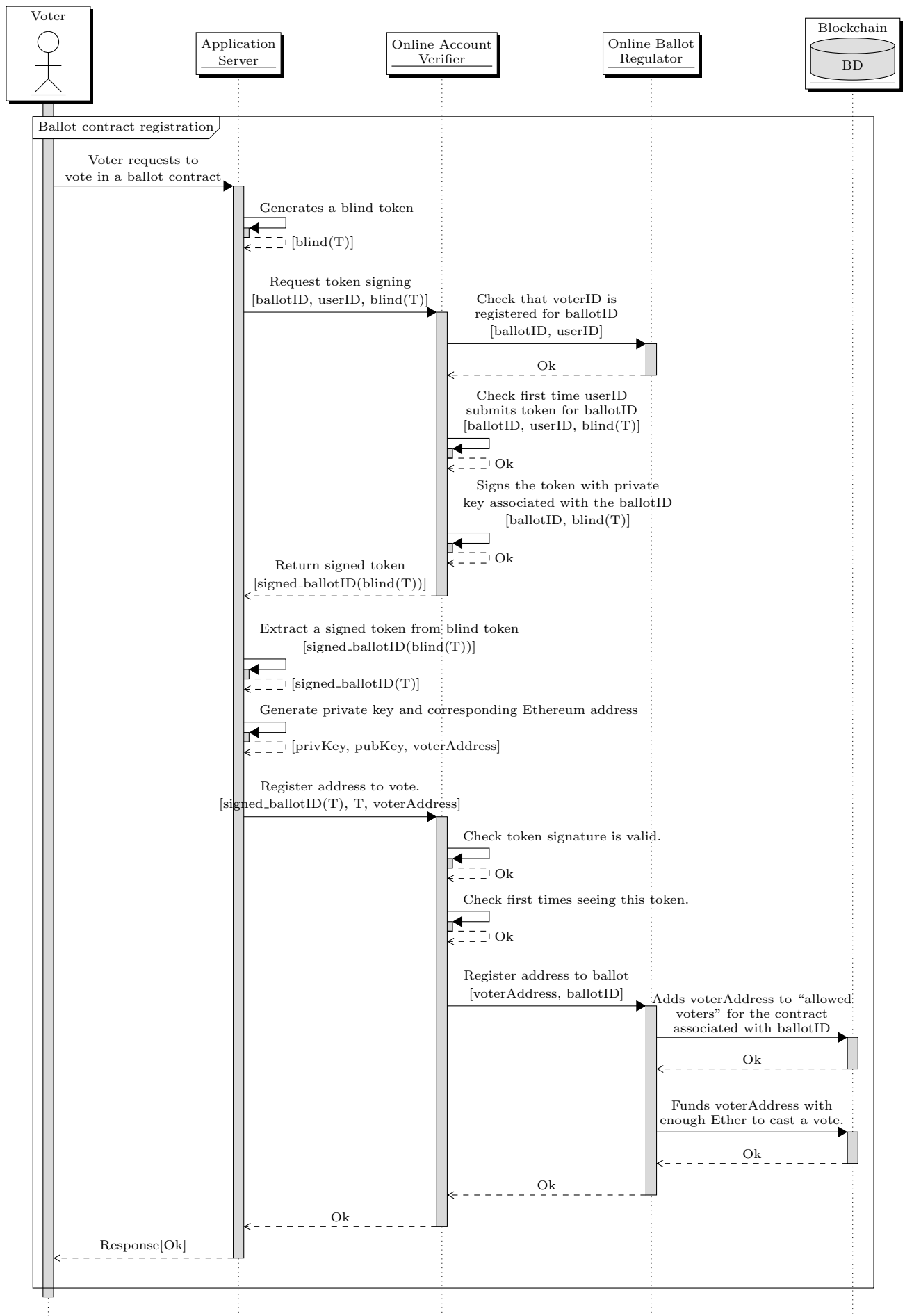


Figure 11: Sequence diagram showing calls made when a user requests to register with the ballot contract in the blockchain.

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 signing in my system is accomplished using RSA keys and the PyCrypto library [28] 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 1.4](#) 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 [29] 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 contract (see [Blockchain Ballot Contract 1.6.5](#) 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.

1.8.3 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.

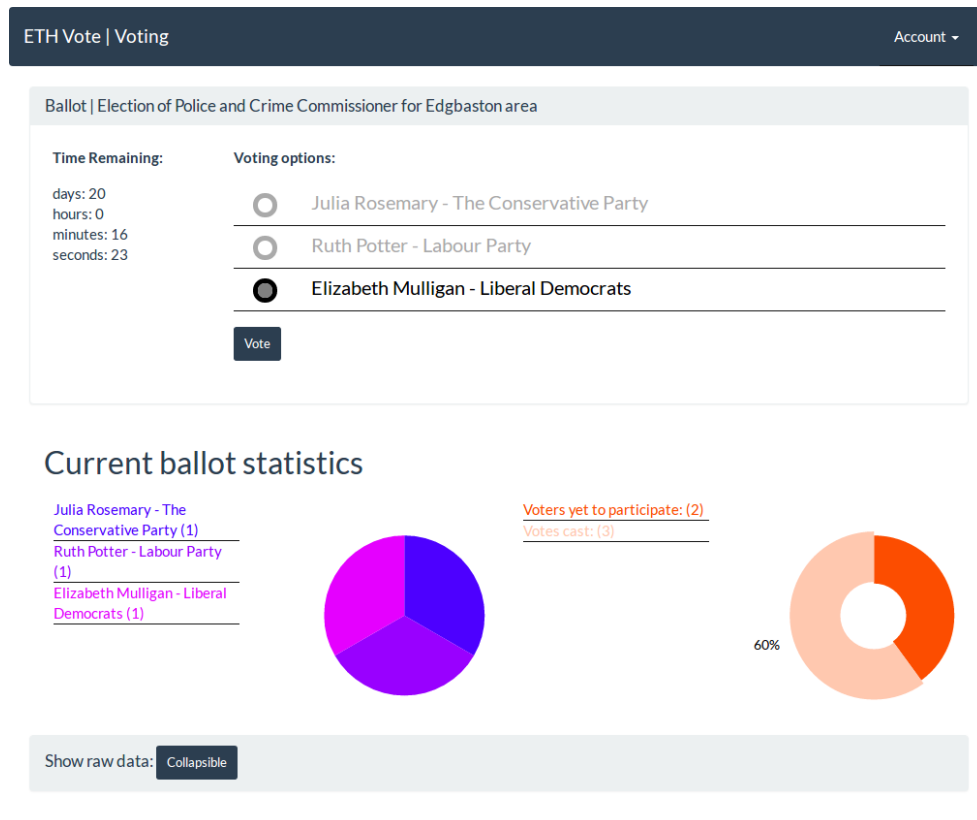


Figure 12: Ballot screen presented to the user.

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 1.6.5](#) 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](#)).

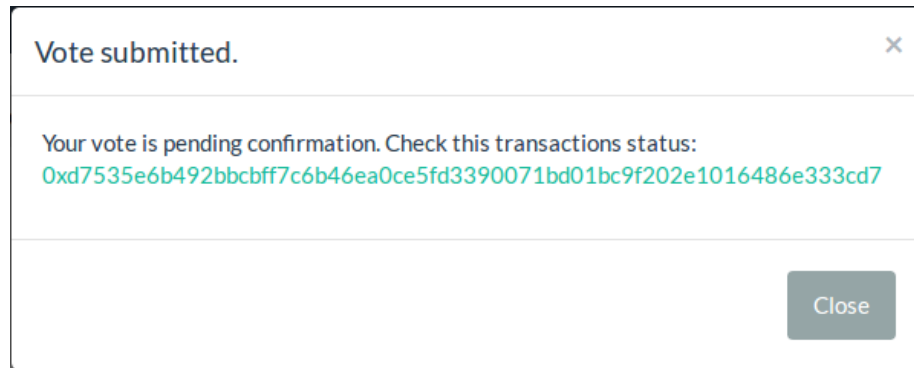


Figure 13: Pop-up box on submission of a vote giving the voter the transaction hash.

Overview

Transaction Information
VMTrace
RemixDebug

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] : 0002

Convert To Ascii

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

1.8.3.1 Real time results

As discussed in [Blockchain Ballot Contract 1.6.5](#), 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 produce a more informed voting populous.

Say, for example, that 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 [30]).

1.8.3.2 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.

1.8.3.3 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.

1.9 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.

Current ballot statistics

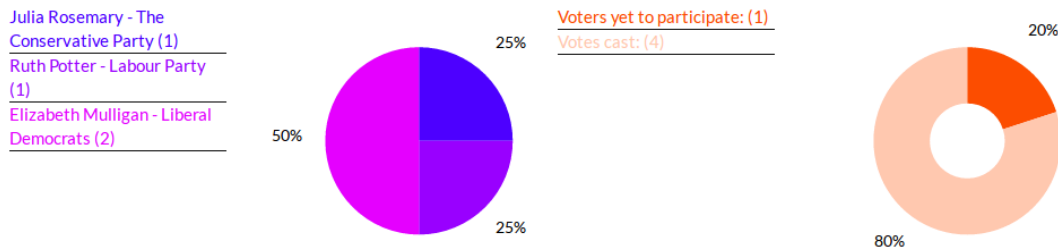


Figure 15: Example of an infographic generated from one ballot deployed to the blockchain.