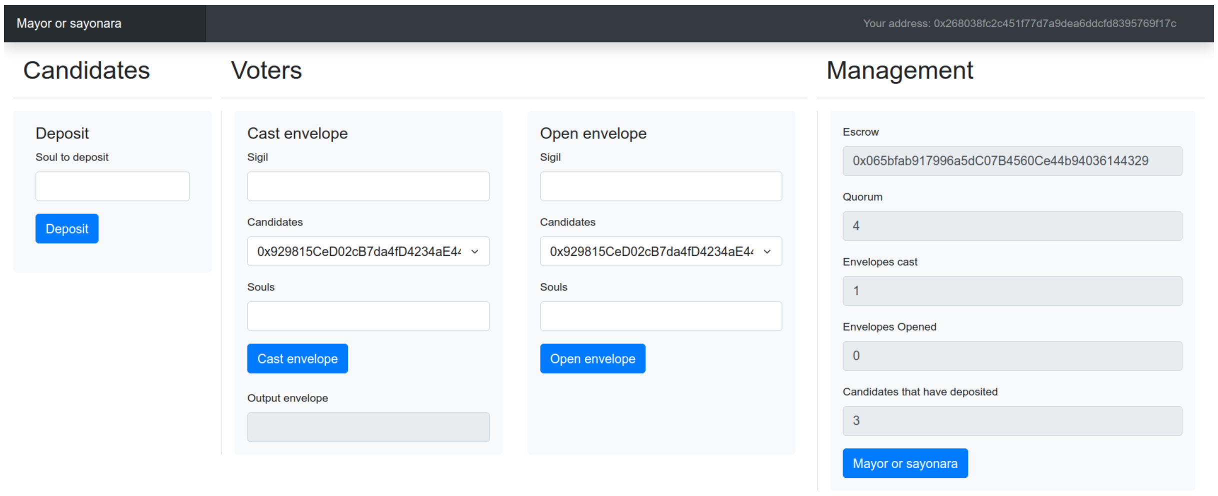
Peer to Peer Systems and Blockchains - Project

Leonardo Vona
545042

August 24, 2021

1 Overview

The project generalizes the contract of the final term extending the concept of voting for a single candidate to become mayor or not to the possibility of choosing from a list of candidates and voting for the preferred one. The extra implemented into the contract is *'Join my side, I give you cookies'*, and then the candidates have the possibility to deposit some soul that will be given in equal parts to all her / his electors in case of victory. The project was developed using the following tools and frameworks: Truffle, Ganache, Node.js, web3.js, lite-server, Bootstrap.



Your address: 0x268038fc2c451f77d7a9dea6ddcfd8395769f17c

Candidates: 0x929815CeD02cB7da4fD4234aE4

Escrow: 0x065bfab917996a5dC07B4560Ce44b94036144329

Quorum: 4

Envelopes cast: 1

Envelopes Opened: 0

Candidates that have deposited: 3

Figure 1: DApp interface.

2 Project structure

The project root folder mixes the structure given by Truffle and Node.js:

- `contracts/`: Contains the Solidity smart contracts. In particular it includes `Mayor.sol` which implements the *Mayor* smart contract.
- `migrations/`: Contains the code that Truffle executes during the deployment. The file `2_deploy.js` is used to deploy the *Mayor* smart contract with the following default configuration:
  - Candidates: accounts 0, 1 and 2.
  - Escrow: account 3.
  - Quorum: 4.

  It is possible to edit the configuration in order to try different combinations of the smart contract parameters.
- `src/`: It includes:
  - `css/`: Contains the `.css` files used to describe the presentation of `index.html`. In particular, it includes a minimized version of the Bootstrap framework.

  - `js/`: Contains `app.js`, which implements the front end code of the project and uses web3.js to communicate with the Solidity smart contract.
  - `index.html`: Implements the interface used by the user to interact with the smart contract.

- `test/`: Includes `test_contract.js` which contains the code used to test the *Mayor* smart contract.
- `bs-config.json`: lite-server configuration file. Includes the list of folders that lite-server needs to look at to serve the web application.
- `package.json`: Contains all the requirements for the npm project.
- `truffle-config.js`: Truffle configuration file. Includes the specification of the development Ethereum network.

2.1 *Mayor* smart contract

The *Mayor* smart contract was developed on the basis of the one provided with the final term. The contract has been extended and edited to satisfy the new requirements, and the main differences with the final term smart contract are:

- The `doblon` concept has been replaced with the `symbol`, which corresponds to the address of a candidate.
- The candidates are managed with a `mapping(address => Candidate)` and a dynamic array containing the addresses of the candidates. `Candidate` is a struct including, among the others attributes, the number of votes and the amount of soul received by the electors; the latter values are increased inside the `open_envelope()` function and are used into the `mayor_or_sayonara()` function to decree a new mayor. In order to iterate through the mapping, a dynamic array of candidates addresses has been used.
- The deposit phase is managed with:
  - The struct representing a candidate (`struct Candidate`), which also contains attributes that indicate if the candidate has deposited and the amount of the deposit. Considering that the candidates are stored in a mapping, an additional boolean attribute `isCandidate` has been added to the struct in order to simplify the check that an account is effectively a candidate and then is able to deposit.
  - An event `Deposit(address _candidate, uint _soul)` emitted when a candidate deposits some soul.
  - A modifier `canDeposit()` that allows the deposit of soul only if not every candidate has deposited.
  - A modifier `depositComplete()` that allows a voter to cast an envelope only after the completion of the deposit phase.
  - A function `deposit() canDeposit public payable{...}` used to effectively allow a candidate to deposit some soul.
- An additional event `Tie(address _escrow)` has been added to handle the situation where two or more candidates receive the same amount of soul and votes, and then there is a tie.
- A new function `kill() external{...}` and a new attribute `address payable public owner` have been added to allow the owner of the contract to destroy it. This function is useful for the testing phase, in order to allow all the tests to be executed in a clean environment.

- The draw of the new mayor inside the `mayor_or_sayonara()` function has been modified to handle multiple candidates.
- An additional function `get_number_of_candidates() public view returns(uint){...}` has been added to permit the retrieval of the number of candidates from the front end code, which uses this information to iterate trough the candidates addresses dynamic array.

3 Project setup

In order to setup and try the DApp, it is necessary to have installed Node.js, Metamask and the ganache-cli tool[1]. To execute the DApp, you have to perform the following steps:

- Open a terminal and execute the command `ganache-cli`.
- Open another terminal, move to the project root directory and execute, in order:
  - `npm install`
  - `truffle migrate --reset`
  - `npm run dev`
- At this point, open the browser (if it does not open automatically) and browse to the page `http://localhost:3000/`. You will see the DApp interface.
- Log in to Metamask and import one of the accounts made available by Ganache, connect it to the DApp and reload the page.

After the execution of these steps, it is possible to interact with the DApp through the web page (you need to import a total of four accounts, including the candidates, for a complete interaction).

4 Main decisions

Smart contract attributes

- Some smart contract attributes are redundant. This means that some information that can be obtained elaborating one or more attributes are also stored in form of another attribute. For example the number of candidates that have deposited can be deduced looping the candidates data structure and counting the candidates that have deposited; however, this information is directly stored in the attribute `candidates_deposited` of the voting conditions data structure.

  This choice has been made in order to reduce the gas price required to execute the functions of the smart contract. Indeed, if we consider the example above, the cost of checking if all candidates have deposited is much lower if we use an additional attribute that stores the number of candidates that have already deposited.
- All the attributes of the smart contract are public; this means that they can be accessed (but not modified) through a getter function. The decision to make all the attributes public has been made to simplify the process of information retrieval from the front end code.
- The voters and candidates data are stored into the smart contract in form of two mappings - `mapping(address => Voter)` and `mapping(address => Candidate)` - and two dynamic arrays of addresses. The arrays of addresses are necessary to loop through the mappings.

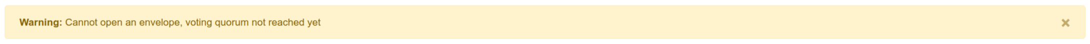
[1]The DApp tries to connect to the default ganache-cli port (8545), then if you have modified it or if you instead have installed the gui version of Ganache, you need to edit the network port into `truffle-config.js` configuration file and `src/js/app.js` file

Deposit

- The smart contract has been developed making the choice that the deposit phase must happen before the voters start casting their envelopes. This means that after the contract creation, all the candidates must deposit some soul (which can also be 0), and only after this phase the voters can cast their envelopes.
- It was decided that a candidate can deposit only once; otherwise, the function will fail and the transaction will be reverted.

Errors and events

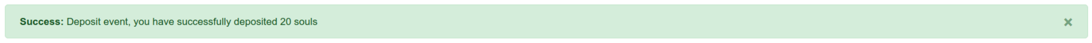
- In case of failure during the execution of a transaction (for example because a require statement inside a smart contract function fails), an error is propagated to the front end through Metamask. This error is handled through a promise object, that is parsed and elaborated in order to retrieve the revert reason, which is displayed to the user via a bootstrap alert. Metamask does not handle reverts gracefully, and then is necessary to elaborate the promise object in order to retrieve the original error reason.



Warning: Cannot open an envelope, voting quorum not reached yet ×

Figure 2: Error message example.

- The emission of a Solidity event is handled in the front end code through a promise object. This object is parsed to retrieve the event and its parameter(s), which are displayed to the user via a bootstrap alert.



Success: Deposit event, you have successfully deposited 20 souls ×

Figure 3: Event emission message example.

Miscellaneous

- The GUI has been developed as a single page including the various sections for all the roles into the DApp. This choice has been made in order to simplify the interaction with the smart contract.
- The value of the soul is intended in ether and not in wei (1 *soul* = 1 *ether*). This decision was made because Metamask considers only the first four decimals, and then if a transaction has a value less than 0.0001 *ether*, Metamask displays a value of 0 even if it is not.
- The `compute_envelope()` function from the smart contract has been 'masked' from the GUI: it is not possible to execute it directly, but only in combination with the `cast_envelope()` function.

  When a voter wants to cast an envelope, it chooses the sigil, the symbol (from a restricted list of available values) and the soul, and when he / she presses the *Cast envelope* button, the DApp will execute both the `compute_envelope()` and `cast_envelope()` functions. The result of the `compute_envelope()` function will be displayed into the *Output envelope* field.

  The decision to mask the `compute_envelope()` function has been made in order to avoid the casting of envelopes with invalid symbol, which therefore would never be opened.

- The task of checking the possibility to execute a function is demanded to the smart contract. This means that, for example, when a voter tries to open an envelope, it is up to the `open_envelope()` smart contract function to check if the casting envelope phase is concluded.
- When a new mayor is elected inside the `mayor_or_sayonara()` function, the soul she / he has deposited is given in equal parts to all her / his electors. In Solidity the division is implemented by the integer division; this means that if the division is not exact, some extra soul will be left from the operation. It has been decided to give the extra soul to the escrow account.
- To avoid a possible double spending attack, it has been chosen that the `mayor_or_sayonara()` function can be executed only once.

5 Demo execution

The demo of the DApp is manual and references to the default configuration of the contract (candidates: accounts 0, 1, 2; escrow: account 3, quorum: 4). The interaction with the application happens through the user interface provided by the view `index.html` and multiple accounts are needed (the accounts must be connected to the DApp and when you switch between them in Metamask, you need to reload the page to effectively change account into the DApp).

The demo is intended as a correct flow of execution from all the accounts that participate (e.g. a voter does not try to deposit some soul or an account does not press the *Mayor or sayonara* button before the previous phases are completed), but alternative sequences of steps can be easily made in order to try the DApp in conditions that bring to a revert of a transaction (e.g. try to deposit twice from the same candidate or open a wrong envelope).

In order to execute the demo, the following steps have to be performed (the authorization of the transaction into Metamask is omitted):

- Setup the DApp (section 3).
- Import the first three accounts (the candidates) and another of your choice from Ganache into Metamask using their private keys.
- *Deposit phase* - For each candidate:
  - Fill the *Soul to deposit* field with an amount of soul of your choice.
  - Press the *Deposit* button.
  - A message will appear informing you that the soul has been deposited.
- *Cast envelope phase* - For each account:
  - Into the *Cast envelope* form, fill the input elements with values of your choice.
  - Press the *Cast envelope* button.
  - A message will appear informing you that the envelope has been cast and the output envelope will appear inside the *Output envelope* field.
- *Open envelope phase* - For each account:
  - Into the *Open envelope* form, fill the input elements with the correct values of each voter (for each voter insert the same values of the cast envelope).
  - Press the *Open envelope* button.
  - A message will appear informing you that the envelope has been opened.
- *Mayor or sayonara* - With an account of your choice press the *Mayor or sayonara* button, then a message will appear at the bottom of the page informing you the election of a new mayor or a tie.