Università di Pisa
Dipartimento di Informatica

Peer to Peer Systems and Blockchain

Student: **Samuele Intaschi**
ID: **523864**

Democratic Election System Attempt

1 Introduction

The goal of this project is to implement a democratic electoral system for the imaginary city of Valadilene, through a decentralized application (DApp).
This application has a front-end, implemented in *Javascript* and accessible via a web browser, and a distributed back-end, implemented by means of *Ethereum* smart contracts.

2 Project Structure

The project folder is divided in subfolders:

- `build`: contains the contracts compiled, which are in *JSON* format, in this case only `Mayor.json`
- `contracts`: contains the source code of the contracts, which are written in the *solidity* programming language, in this case only `Mayor.sol`
- `migrations`: contains the code to migrate the contracts on the network, in this case on the *Ganache* test network
- `src`: contains the source code of the front-end, the behavior is implemented in *Javascript*, the style is molded with *CSS* and the frame is specified in *HTML*
- `test`: contains tests, written in *Javascript*, which ensure that the behavior of the smart contract is correct in almost all situations

Then, there also other files that specify some configurations for *lite-server* and for *truffle*, for example in one of this files the network in which the contracts are to be deployed is specified.
The back-end part of this project was developed with the help of the *truffle* platform, that provides useful functionalities for the management of smart contracts, in particular a *solidity* compiler, a migration tool to deploy contracts on the *Ethereum* network and an internal test environment.

Moreover, *truffle* offers also a local simulated test blockchain named *Ganache*, in which it is possible to migrate our contracts to test them without the need to have real *Ethereum* accounts. The front-end of the application is implemented in *HTML* and *CSS*, for graphics and the style, and in *Javascript* for logic. It is a web page, accessible by means of a web browser, which displays various forms, one for creating a coalition, one for casting an envelope and finally one for opening an envelope.
Thanks to *NodeJS* , we can have *lite-server* , a local server serving this application on localhost. The front-end can interact with the smart contracts by means of *Web3.js*, which is a library to interact with the nodes of *Ethereum* network with the *Remote Procedure Call* (*RPC*) protocol, from *Javascript*.
To interact to a smart contract, deployed on the *Ethereum* network, a *Web3* provider is required to manage the *Ethereum* accounts and interact with websites like this one that use *Web3* to interact with the blockchain. For this reason there is *metamask*, implemented mainly as a browser extension available for most web browsers. We can connect *Metamask* to *Ganache* to use the test accounts created by *Ganache* to send transactions to the back-end (smart contract) of the application, to test it. In the figure 1 it is shown the structure of the application, where *Ganache* locally implements an *Ethereum Virtual Machine* with its blockchain.

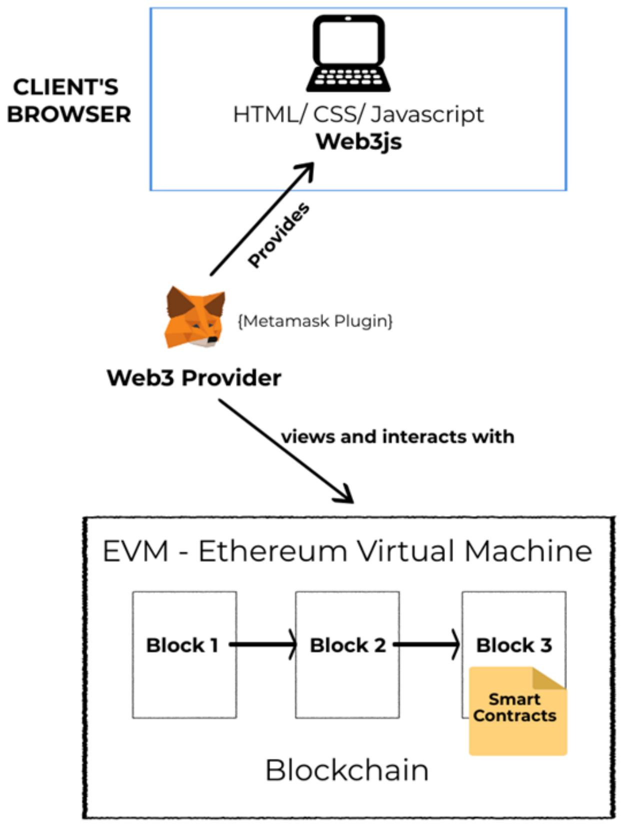


Figure 1: This picture is taken from `https://medium.com/@filzatariq92/build-your-ethereum-dapp-on-windows-with-truffle-ganache-and-metamask-beginners-guide-8c62b55ef556` and shows the way in which *Metamask* is used in this application, where the EVM is implemented by means of *Ganache*.

When the contract is deployed on the network and *Metamask* is connected to *Ganache*, it is possible to start and use the application, via a browser. The application starts by initializing *Web3* and the contracts (in this case only `Mayor.sol`), setting a listener for some of the events generated by the contracts and finally rendering the *HTML* page, loading information about

candidates and coalitions from the contract. Subsequently, the user can request to create a coalition, obviously if the voting quorum has not been reached yet, or to cast or open an envelope representing a vote. To make these request he has to compile the respective *HTML* form, from which the necessary information is taken and used to invoke the functions of the smart contracts, using *Web3* . For example if an account wants to create a coalition, it has to select through the checkbox which candidates will form the coalition and then press the *Submit* button to submit the request. The application takes the address of the account executing the request, which will be the address of the coalition, and the list of the candidates that make up the coalition and forwards the request to the smart contract. When a new coalition is created an event is triggered, so any web application connected to the deployed contract has an event listener that adds the coalition to the list of eligible candidates.
Once the voting quorum has been reached it is possible for the user to open the cast envelopes and when all the envelopes have been opened, the contract fires another event, the front-end listens to it and communicates the outcome of the elections. Other events generated by the contracts communicate the outcome of the operations requested at the contract and are used to ensure that operations have been completed successfully. At the end of the elections, when a new mayor is elected or the escrow takes the whole *soul* of the voters, the contract self-destructs, and a new contract, with other or the same candidates, can then be deployed on the network.
The tests contained in the `test` folder are used to verify that the general behavior of the back-end is correct. In particular a normal situation is simulated in which the *soul* paid by the voters is randomly generated and a candidate wins the elections, a situation in which a coalition is created and it wins the elections and a situation in which the total sum of *soul* is sent to the escrow because there are not winners, with or without the coalitions. This way not all the possible situations are tested, so the test coverage is not complete, but at least most of them are generated.

3 Design Choices

This section illustrates the main design choices made during the development of this application, for the technologies used and for the implementation.

3.1 Technologies

First, it was chosen to implement the application as a web application accessible from a normal browser because it was easier to use for an user and because in this way it was possible to take advantage of *Metamask*, that comes mainly as a browser extension that injects its *Web3* provider in the browser, as mentioned in section 2. In particular the application is implemented as a single page application (SPA), where the elements are molded by *CSS* and *Javascript*, and it was chosen *lite-server* (for *NodeJS*) to serve this page on the browser, because it is a lightweight node development-only server, providing the minimum functionalities to develop a web application, such as showing a fallback page when a route it is not found and automatically updating the page when there are changes in the *Javascript*, *CSS* and *HTML* files, which is very handy during the development.
To simulate an *Ethereum* network it was used *Ganache* , because it comes with *truffle* and is very easy to connect to *Metamask*.
Finally, to ensure that the behavior of the contract is correct, the test tool provided by *truffle* is used.

3.2 Implementation

The public information of the contract, in particular the list of votable candidates and coalitions, is taken when rendering the *HTML* page. There is no way to know in advance how many candidates or coalitions are, because they are returned one by one, so the procedure for retrieving this information ends when a contract exception is caught. This information is then shown to the user by means of *HTML* components. In this phase it is also checked if the quorum is

reached, because in this case users can open their own envelopes.
This smart contract generalizes an existing contract that allowed to cast only a positive or negative vote for a single candidate, adding the possibility of having a list of candidates that could be voted on. So, instead of a single candidate, now a list of candidates is passed to the contract constructor and saved by the contract.
The next steps are the same as in the previous contract, with the exception that now when a user casts a vote, he has to specify a *symbol*, which is the *Ethereum* address of the candidate who wants to vote, rather than a simple positive or negative vote (the previous *doblon*). The *soul* the user pays for a candidate are then associated to the candidate and used to compute the results of the elections at the end. With this modification it is very difficult for the elections to end without a result, because even if two or more candidates get the same amount of soul, the one with more votes wins.
There is another novelty compared to the previous smart contract, which consists in the addition of an event and a function used by the front-end to inform the user when the quorum has been reached, so that he can open his envelope. Without this information the user should try repeatedly to open the envelope (and send useless transactions) until the quorum has not been reached, and this is a bad and not much usable solution. In this way, on the other hand, when the last envelope is cast, the event emitted is not *EnvelopeCast* as in the previous cases, but is *QuorumReached*, that informs each front-end connected to the contract that from now on users can open their envelopes. If instead a front-end is opened later, when the quorum has already been reached (it does not receive the event), there is a function that returns true if the quorum is already been reached and false otherwise, which is called when the page is rendered. This information is communicated to the user who knows he can open his envelope. This function exposes the internal state of the function, so it does not cost any gas to run, and only says whether or not the quorum is reached, not the number of votes needed to reach it, so it should not pose a security problem.
After that the contract has been generalized, the extra proposal that has been decided to implement is *The Pammerellum*, i.e. the possibility of grouping together more than one candidate in coalitions. A coalition in the contract is treated as a structure that has its component address list and its own *Ethereum* address. The possibility to create a coalition is allowed to each *Ethereum* account connected, with the exception of accounts that have their own addresses already used by the candidates or by already existing coalitions. A coalition can be created until quorum is reached. When an user requests to create a coalition he specifies which are the candidates he wants to group, then the address of the coalition becomes the address of this user. Other possibilities were to specify coalitions at the time of contract creation, as is the case for candidates, or to limit the possibility of creating a coalition to candidates.
It was decided not to specify coalitions when creating the contract to make contract more dynamic, and its initialization faster and simpler. Additionally, in this way it is possible to exploit the events generated by the contract and the web application event listeners to dynamically add the newly created coalition to the front-end, as explained below. Instead it was decided not to allow candidates to create coalitions, to ensure that there are no situations in which a coalition and the candidate who creates it have the same address, creating confusion in the voters (and in the system). So, a candidate who wants to create a coalition can do so, but using a different *Ethereum* account. When a new coalition is created an event is fired by the contract and listened by the front-end, which in this way can update the list of the votable candidates/coalitions shown to the user adding the newly created coalition.
It is added also another function (`get_coalition`) that the front-end uses to retrieve the data structure representing a coalition by the front-end, as *Web3* does not allow to retrieve compound structures such as these. This is a *view* function, because it only exposes the internal state of the contract, so it does not consume gas to run.
In the `mayor_or_sayonara` function, now the first check is to look for a coalition that has at least one third of the total *soul* and more *soul* of all other coalitions, if it exists it wins the elections. If there are two or three coalitions that has at least one third of the total *soul* , there are no winners and all the *soul* goes to the escrow, if instead there is not even a coalition with more than one third of the total *soul* the candidate with more soul (or votes), if any, will be the winner of the elections.

To make sure that the behavior of the contract is correct, also if the users are wrong or malicious, some checks are carried out by the contract through modifiers, for example to check if the quorum is or is not reached or if all the cast envelopes have been opened. There are some other checks also in the `create_coalition` function, to make sure that there are more than one candidates in a coalition, that all the components of the requested coalition are really candidates, that the address requesting the creation is not the address of a candidate and finally that there are no duplicates in the components that will make up the coalition.
Apart from these, the only checks carried out by the contract are to make sure that there is only one envelope cast for an address, and that the user opens his envelope and not another
The first checks relating to the format or correctness of the information entered by the user on the front-end are carried out by the front-end itself. The front-end carries out, as well as the contract, also the checks relating to the achievement of the quorum to avoid the sending of useless transaction. Anyway, if there are other exceptions thrown by the contract due to other errors, the front-end catches them and shows an error to the user.

4 Usage

First, to use this application the user must have a browser, such as *Google Chrome* or *Mozilla Firefox*, which is essential for accessing the web page. Moreover he need *NodeJS* [1] and its packet manager *npm*[2] to install all packages and serve the web page. With *npm* installed, the user can install *truffle* with the command:

```
npm install -g truffle
```

With *truffle* installed the smart contract can be compiled with the following command:

```
truffle compile
```

Now a network is needed in which deploying the compiled contract, for this purpose the user can install *Ganache* , downloading it from the page `https://www.trufflesuite.com/ganache`, or he can install the command line interface version[3] of the tool with the command:

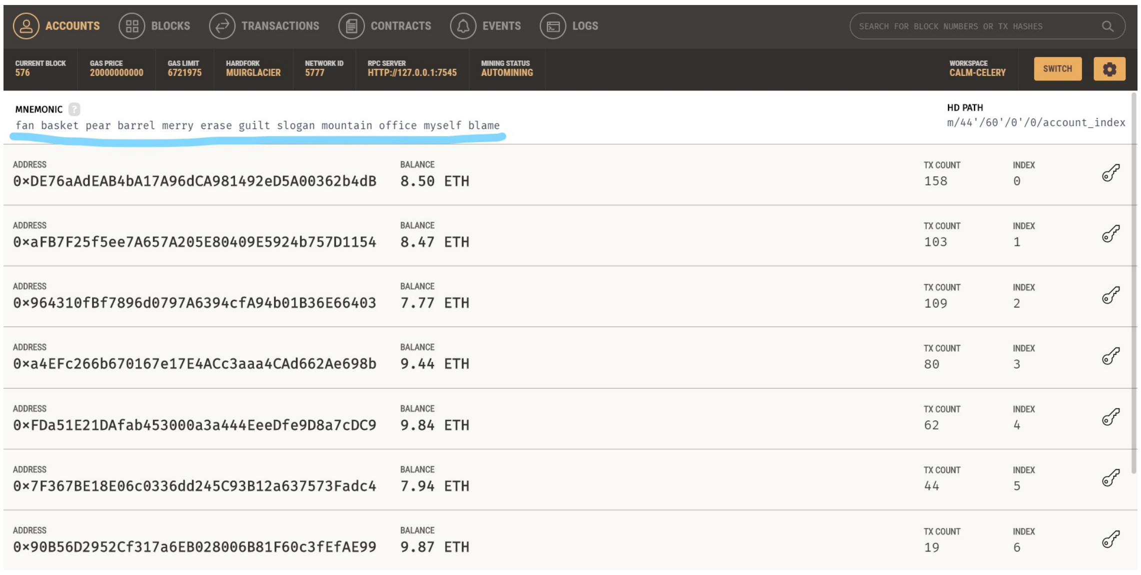
```
npm install -g ganache-cli
```

At this point the user can start *Ganache* and then deploy the previously compiled contract, with the command:

```
truffle migrate
```

All the information needed to deploy the contract on the *Ethereum* network selected is specified in the configuration file `truffle-config.js`, so the user has to adapt this information if using a different configurations for *Ganache.*
In the file `1_initial_migration.js` (contained in the `migrations` folder) information relating to the construction of the contract at the time of its deploying on the network is specified, i.e. quorum value, candidates list and escrow address, so by changing the default values of these parameters before migration it is possible to customize the contract.
To make requests from the browser to the smart contract deployed on the *Ethereum* network the user has to install *Metamask* browser extension, following the steps on `https://metamask.io/`. With *Metamask* installed, the user has to connect to the network, using the seed phrase generated by *Ganache* for our test network. If the graphical version of *Ganache* is used, the seed phrase is the one underlined in the figure 2.

[1]`https://nodejs.org/it/`
[2]`https://www.npmjs.com/`
[3]This application was developed entirely using the graphical version of *Ganache* , it is not guaranteed that works with *ganache-cli*



CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE
576	20000000000	6721975	MUIRGLACIER	5777	HTTP://127.0.0.1:7545	AUTOMINING	CALM-CELERY

MNEMONIC
fan basket pear barrel merry erase guilt slogan mountain office myself blame

HD PATH
m/44'/60'/0'/0/account_index

ADDRESS	BALANCE	TX COUNT	INDEX
0×DE76aAdEAB4bA17A96dCA981492eD5A00362b4dB	8.50 ETH	158	0
0×aFB7F25f5ee7A657A205E80409E5924b757D1154	8.47 ETH	103	1
0×964310fBf7896d0797A6394cfA94b01B36E66403	7.77 ETH	109	2
0×a4EFc266b670167e17E4ACc3aaa4CAd662Ae698b	9.44 ETH	80	3
0×FDa51E21DAfab453000a3a444EeeDfe9D8a7cDC9	9.84 ETH	62	4
0×7F367BE18E06c0336dd245C93B12a637573Fadc4	7.94 ETH	44	5
0×90B56D2952Cf317a6EB028006B81F60c3fEfAE99	9.87 ETH	19	6

Figure 2: Screenshot of the graphical version of *Ganache*.

To successfully use *Ganache* test environment from *Metamask*, the user has to create a custom RPC (remote procedure call) on the *Metamask* browser extension panel, setting a name, an URL (typically it is `http://localhost:7545` for *Ganache*) and the chain ID, which is 1337 for *Ganache*. Then the user has to select on *Metamask* panel the newly created RPC from the selection menu He can now see the accounts generated by *Ganache* and select one of them to connect it, seeing also its balance and general information. At this point everything is ready to go with a terminal in the application folder and start it with the command:

```
npm run dev
```

The application, after the execution of this command, is listening on `http://localhost:3000` so, opening a web browser and going to this URL, the user should finally see the web page visible in figure 3.

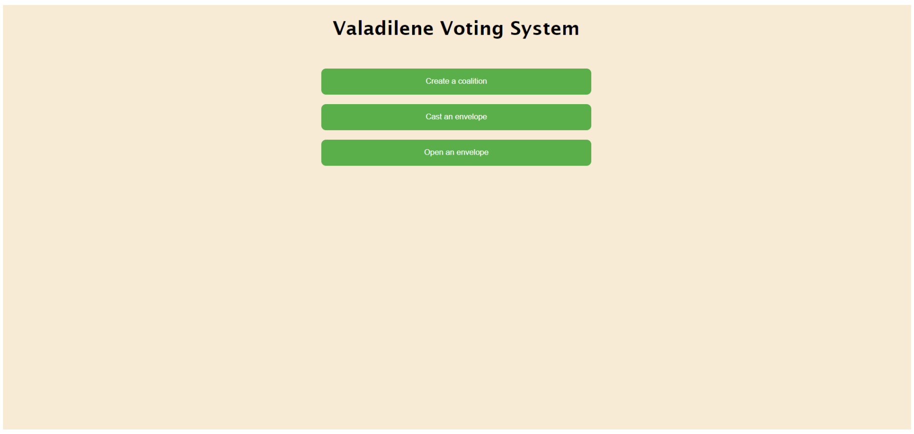
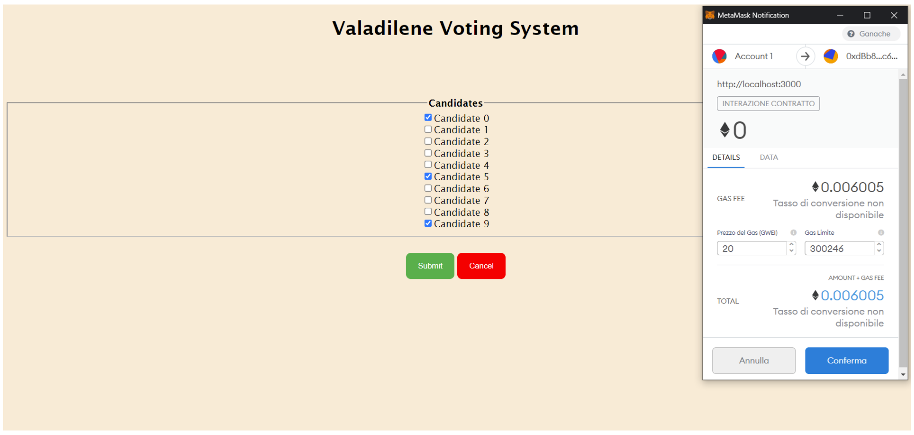


Figure 3: Home page of the application.

In this page the user can perform three actions which are to create a new coalition, that is to give his address to a coalition, to cast an envelope, which represents a vote, and to open a previously sent envelope.
To create a coalition the user has to press the *Create a coalition* button, then fill in the checkbox corresponding to the candidate he would like to include in the coalition, press *Submit* and finally

confirm the transaction in *Metamask* , that shows him a window to do this, as it is represented in figure 4.

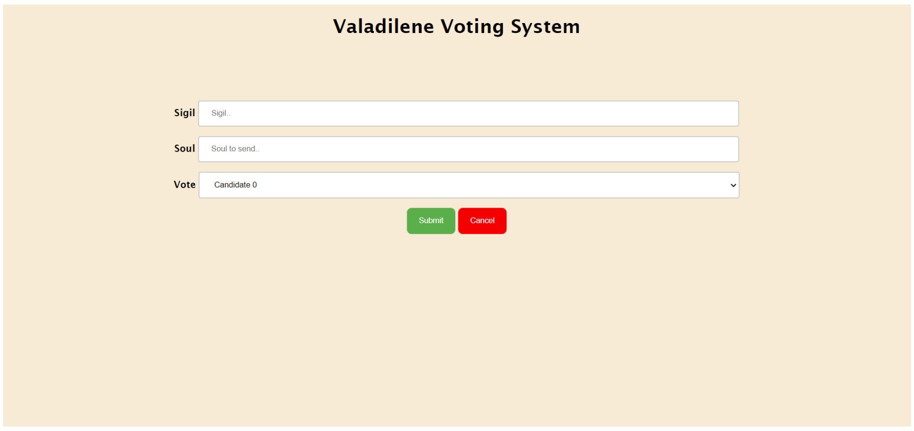

Valadilene Voting System

Candidates
- [✓] Candidate 0
- [ ] Candidate 1
- [ ] Candidate 2
- [ ] Candidate 3
- [ ] Candidate 4
- [✓] Candidate 5
- [ ] Candidate 6
- [ ] Candidate 7
- [ ] Candidate 8
- [✓] Candidate 9

MetaMask Notification
Ganache
Account 1 → 0xdBb8...c6...
http://localhost:3000
INTERAZIONE CONTRATTO
0
DETAILS DATA
GAS FEE 0.006005 Tasso di conversione non disponibile
Prezzo del Gas (GWEI): 20
Gas Limite: 300246
AMOUNT + GAS FEE
TOTAL 0.006005 Tasso di conversione non disponibile

Figure 4: Creation of a coalition that groups together the candidates 0, 5 and 9.

To cast an envelope the user has to click on the *Cast an envelope* button, then he has to specify his own *sigil*, i.e. an integer, the amount of *soul* he intends to send to the candidate and the *symbol*, i.e. the candidate he would vote, on the page shown in figure 5.


Valadilene Voting System

Sigil: Sigil..
Soul: Soul to send..
Vote: Candidate 0

Figure 5: Page used to cast an envelope.

After specifying this information, he has to press the *Submit* button and confirm the transaction generated by *Metamask*. In this phase *soul* transactions are not generated, the envelope is cast into the smart contract to be opened during the opening phase. The envelope representing the vote can only be cast until the quorum is reached, otherwise the front-end reports an error to the user. Candidates are shown on this page as *Candidate 0*, *Candidate 1* and so on, because the *Ethereum* address is not so convenient to show especially to distinguish a coalition from a normal candidate, but perhaps it is better to use the candidate's name if the project is carried forward.
Once the quorum is reached, the front-end communicates it to the user, who can open the previously cast envelope, specifying the same *sigil*, amount of *soul* and *symbol* selected at the moment of casting, with a page identical to that shown in figure 5. Then the user has to press the *Open envelope* button and confirm the transaction in the *Metamask* panel.
The *soul* sent corresponds to *wei*, a unit so small that the *Metamask* panel does not show it unless at least 1000000000000 *soul* (a *microether*) are specified to send.
When the last envelope is opened the result of the elections is computed and finally the symbol of any winner is displayed on the front-end of all connected users.

After this the contract has finished its work and therefore self-destructs.
So in summary, the full correct use with all dependencies installed and if the user is not interested in creating a coalition, it is:

1. Open a browser and go to `http://localhost:3000`,
2. Cast an envelope,
3. Wait for the front-end to communicate that the quorum has been reached (in this time interval the user can change his vote),
4. Open the envelope sent earlier,
5. Wait for all envelopes to be opened and the results computed.

To run all tests, the user should edit the *truffle-config.js* file, commenting out the lines relative for the network where the contract will be deployed, to make sure the tests do not interfere with the network, then run the following command:

```
truffle test
```

or, if the user also wants to see the events generated by the contract:

```
truffle test --show-events
```

All the tests are run and the report is printed on the terminal.

5 Demo

The demo provided is very simple and it is shown in the video `demo.mp4`, contained in the project folder. Unfortunately the interactions with *Metamask*, especially the account switch, and the drop-down menu are not recorded, but it has been considered a good way to explain how to reproduce these operations on any normal browser.
A normal situation is shown where there are three candidates and the quorum is three, so three accounts express their vote and in the end one of these candidates wins. It is the same if a coalition is selected instead of a candidate.
In details, the steps recorded in the demo are:

1. Creating of a coalition by *Ganache* account 1
2. Cast of an envelope with *sigil* 1 and 100 *soul* for the candidate 1, with address 0xF076E015a43B8d4510b64CD7302B7a7d127874F, by account 1
3. Cast of an envelope with *sigil* 2 and 200 *soul* for the candidate 2, with address 0x48A9f868446130A896D73BC84d0447eb3b17bD07, by account 2
4. Cast of an envelope with *sigil* 3 and 400 *soul* for candidate 3, with address 0xf22F913131989234020AfaAb74bE4434F8D996EC, by account 3
5. Accounts open their envelopes in turn
6. The results of the election is shown, as expected the winner is candidate 3 (0xf22F913131989234020AfaAb74bE4434F8D996EC), that has received 400 *soul*

However, as explained in section 4, the behaviour of the contract, even if not of the front-end, can be verified in more detail by running the file `test_mayor.js`, in the way explained at the end of section 4.