The attempt of a democratic election system

Mariagiovanna Rotundo - 560765

Peer to Peer Systems and Blockchains. A.A 2020/2021

Structure of the project

The project is composed of the contract, the files for the configuration of truffle, and the java program that implements the front-end and that interacts with the smart contract:

- The contract is written in solidity and is defined in the ***Mayor.sol*** file.
- The file to deploy the contract is the ***1_initial_migration.js*** file and here also the parameters used by the constructor are indicated. The configuration for truffle is defined in the file ***truffle-config.js***, where, for example, it is specified the version of the compiler to use to compile the contract.
- From the compilation of the contract, it is generated the ***Mayor.json*** file that describes the contract as a JSON object.
- The java files of the project are six:
    - The files ***Demo.java*** and ***FakeButton.java*** are used for the automatic demo. In the *Demo.java* is defined the sequence of commands to execute for the demo, while in the *FakeButton.java* file there are methods that simulate the behavior of the buttons of the GUI.
    - ***Mayor.java*** is a java wrapper and it is generated by the *Mayor.json* file. It is used to interact with the Mayor smart contract.
    - The ***Logic.java*** file is the file where the java wrapper is used, and here it is defined the logic of the java program.
    - The ***Gui.java*** file is the file where the GUI is defined and managed and where the behavior of the buttons and the panels is defined using the methods defined in the Logic class. The logic in this class is as simple as possible, the complexity is in the Logic class.
    - The ***Main.java*** file contains the main of the java project.

Set up the project

To use the project, install the truffle suite, in particular truffle and ganache (https://www.trufflesuite.com/), and the web3j library (http://docs.web3j.io/latest/quickstart/).

All the files needed for the project are in the zip but some of them can be generated again automatically. To avoid problems, after the command ***truffle init***, use the given file for the configuration of truffle (after this command the generated configuration can be different).

Then compile the contract with the command ***truffle compile.*** This command generates the file ***Mayor.json***. If in this file there is not the field “payable” for the payable functions of the contract, this can result in a wrong java wrapper (https://github.com/web3j/web3j/issues/1268). Start ganache with the command ***ganache-cli*** and deploy the contract on its network with the command ***truffle migrate*** (use the given js file for the migration). Now, if the compile command executed before has not generated the field “payable” in the JSON file, try to execute again the

command ***truffle compile***. Now the contract has been deployed so, we only need to set up and execute the java project.

To execute the java project, it is possible to set up it as described below, or it is possible to use the jar provided. The jar is runnable, so you can execute it by just clicking twice on the jar file. It also uses the other files in the same directory where is the jar, so do not move the jar file out of the directory.

For the java project, two external libraries had been used: the json-simple and the web3j libraries. The two jars of these libraries are in the zip. So, **create the java project**, **import the java files, and import the two jar in the project**.

The *Mayor.java* file is automatically generated using truffle and web3j, so you can generate it again (but is not needed because it is in the zip). To generate it, from the same directory of the truffle project, execute the command:

web3j generate truffle -t ./build/contracts/Mayor.json -p "" -o .

This command must be executed after the compilation of the contract because it uses the JSON file generated by the compilation (as we can see in the command).

Import the generated *Mayor.java* file in the java project. Remember that if in the *Mayor.json* file there is not the field “payable” the *Mayor.java* has some errors. The simplest way to correct these errors is to add manually to its code the highlighted java code in the following pictures.

```
public RemoteFunctionCall<TransactionReceipt> open_envelope(BigInteger _sigil, String _symbol, BigInteger weiValue)
    final org.web3j.abi.datatypes.Function function = new org.web3j.abi.datatypes.Function(
            FUNC_OPEN_ENVELOPE,
            Arrays.<Type>asList(new org.web3j.abi.datatypes.generated.Uint256(_sigil),
            new org.web3j.abi.datatypes.Address(_symbol)),
            Collections.<TypeReference<?>>emptyList());
    return executeRemoteCallTransaction(function, weiValue);
}
```

```
public RemoteFunctionCall<TransactionReceipt> deposit(BigInteger weiValue) {
    final org.web3j.abi.datatypes.Function function = new org.web3j.abi.datatypes.Function(
            FUNC_DEPOSIT,
            Arrays.<Type>asList(),
            Collections.<TypeReference<?>>emptyList());
    return executeRemoteCallTransaction(function, weiValue);
}
```

In the file Gui.java in the attribute “path” (the first one in the class) it is indicated the path where to find the images provided with the zip. Update it with the new path.

Start the java program (the main of the project is in the *Main.java* file)

Try the project

So, having all the needed files, if we have already the network on and the contract deployed we can directly execute the java program. Suppose we do not have the network ready (but we have the project already compiled and configured), the steps to execute are the following:

1. Start ganache with the command ***ganache-cli***. It generates the private keys and the accounts in the same format shown in the following picture. The private keys are needed to use the graphical interface because they are used to authenticate the user.

```
Available Accounts
==================
(0) 0x1d305DB520ffbD47C85f8Ac7aA1E565C686A114c (100 ETH)
(1) 0xd89Bf8AE5ecBbC7916eAac2aEd13227C1Ae575B9 (100 ETH)
(2) 0x79E55cF900C7106Fed48ec96EFE802731CFC778C (100 ETH)
(3) 0x713eF517c543309e9FB52B3179aE75e491c8F065 (100 ETH)
(4) 0x922cABBfb44cEa6a1d9F568296C257f6e035c852 (100 ETH)
(5) 0xaF5D38E949A82e61F64889d27bbDF227B20be30b (100 ETH)
(6) 0x99E1de1163077926D4a794643c83F1a024Bc1ABF (100 ETH)
(7) 0x5394578dBa44f64c35AafD8607626bF7259c6AB3 (100 ETH)
(8) 0x13824e91087a2e4e761B016A364a2b82c2eaF72D (100 ETH)
(9) 0x552A557560c52a747f5FEda839CE805597DAdCA1 (100 ETH)

Private Keys
==================
(0) 0x509b449e461f5728b23bca0aceb8b5d7d971aa040c2b08ea2a4b9ab6c3ca4fb9
(1) 0x66d5bd4a47bee192c231420f7c951c0bcf0ec77bfbad29cc89ab5890e2635fc9
(2) 0xde28473842514866a458456072f25dd0c2ff2b4295ed6caa73c2ade5fa9cc435
(3) 0xf7f7d7316be0b3b14709cf5dbb7558d1710c72524e28ce26ade98005dbfdb333
(4) 0xf3fbc21ae1da7894746ab0be92e373d2155102e73d05a00f1b6fb629a848a365
(5) 0xc00e668291800dac166e79e9f60450a21adb85c54c030383411f3a7d4a7b411e
(6) 0xc59b7dc63d558a226891c07209c6204997d09e739056d64d3f04aa6db41fee53
(7) 0xbf2ff7e91b664b3c68ff9158b4e0de10781119475bd6102cc7c4c7aabdf34739
(8) 0x4a950bd2324b2ff78e6f84b0bec853aa91f3fac9b379ea96543c6865d03edf11
(9) 0xbf7255c0e6da1157dc55d2b2c38b9ac91e82474eab4b16620e4b0a3a0c7930cc
```

2. Deploy the contract on the network with the command ***truffle migrate.*** This command prints some information, like the following ones:

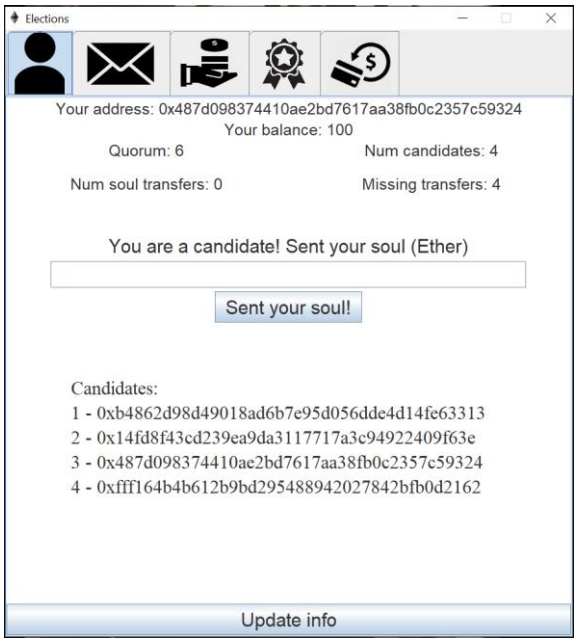
```
1_initial_migration.js
======================

   Deploying 'Mayor'
   -----------------
   > transaction hash:    0x69b17c9d1fe3ca037d71d7a335a897aa5a4232c19e4ba0c0021452941444ad9a
   > Blocks: 0            Seconds: 0
   > contract address:    0x3a7dC638A19802a706bD388fde9F6E0A624eA262
   > block number:        1
   > block timestamp:     1623359364
   > account:             0x1d305DB520ffbD47C85f8Ac7aA1E565C686A114c
   > balance:             99.94268842
   > gas used:            2865579 (0x2bb9ab)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.05731158 ETH

   > Saving artifacts
   -------------------------------------
   > Total cost:          0.05731158 ETH
```

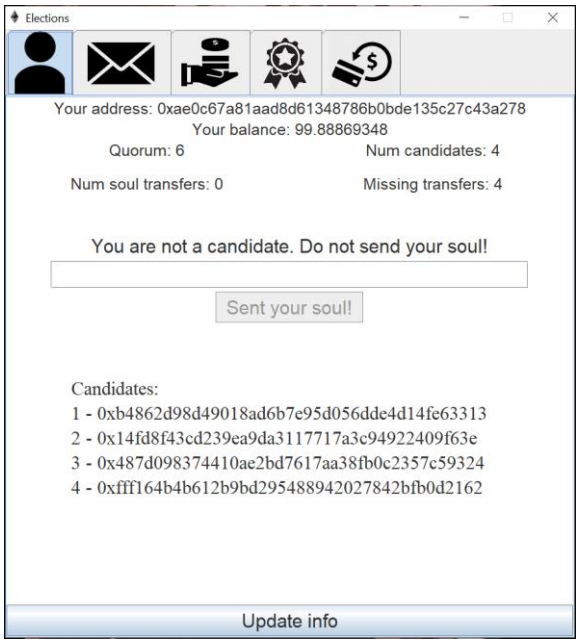
   To interact with the contract, the contract address indicated above will be needed. It is possible to read it also in the information printed by ganache about the transaction to deploy the contract (as shown in the following picture).

```
Transaction: 0x69b17c9d1fe3ca037d71d7a335a897aa5a4232c19e4ba0c0021452941444ad9a
Contract created: 0x3a7dc638a19802a706bd388fde9f6e0a624ea262
Gas usage: 2865579
Block Number: 1
Block Time: Thu Jun 10 2021 23:09:24 GMT+0200 (Central European Summer Time)
```

3. Start the java program using the main in *Main.java* file (not the one in *Demo.java*). After this, the following frame appears:

Here, insert, in the first field, one of the private keys generated, and, in the second field, the contract address seen above. Then click on the “Confirm” button.





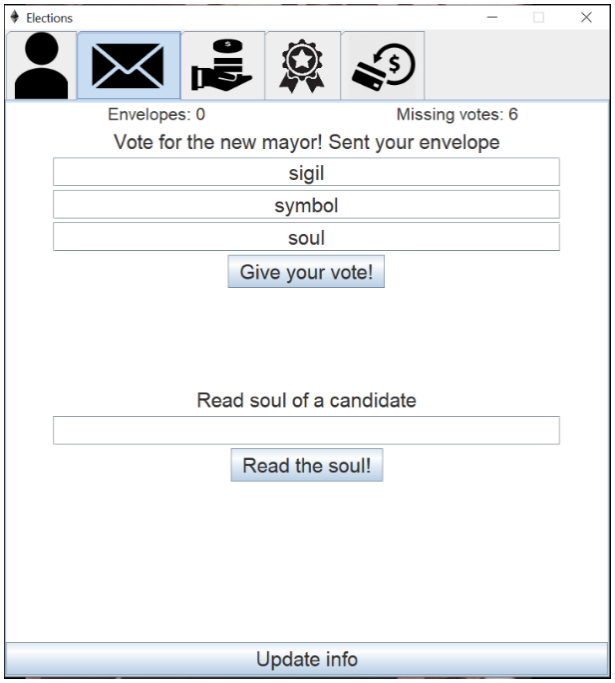
Now if the private key is of a candidate, the frame shown on the left of the picture above appears, otherwise appears the frame on the right. Here you can see:

- the address associated with the private key inserted.

- the balance associated with that account.
- the quorum.
- the number of candidates.
- the number of candidates that have already sent their soul.
- the number of candidates that have not sent their soul yet.
- the list with the candidates (if the candidates are many you can see the whole list scrolling down on the list).

Here there are 2 buttons: the button “Sent your soul!” that reads how many soul you have indicated in the box, and sends these soul to the contract, and the button “Update info” that updates the info shown in this tab. The information is automatically updated every time the tab is shown or when the soul are sent. For accounts that are not candidates the button to send the souls is not enabled.

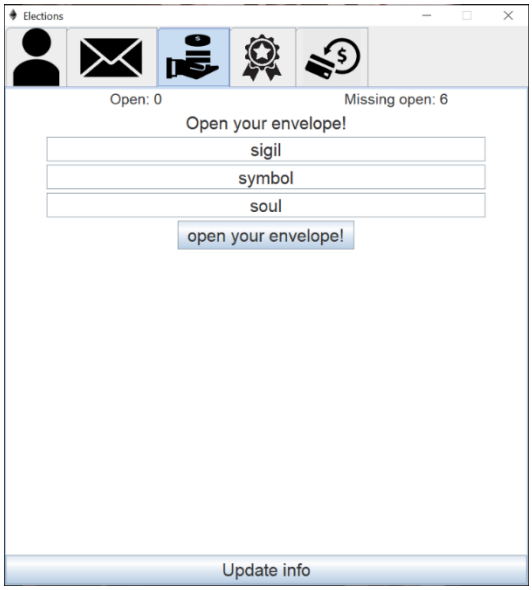
Clicking on the image of the letter, you change the tab and the frame is like the following one:

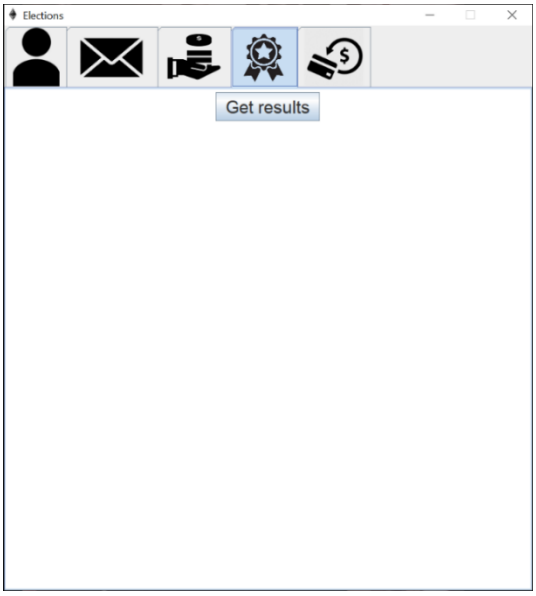


Here you can see the information about the voting phase or the soul sent by the candidates. The information about the voting is the number of envelopes received and the number of envelopes missing to reach the quorum. Then there are the fields to insert the information to vote: the sigil, the symbol, and the soul to use to compute the envelope and to send it to vote. After inserting this information, clicking on the button “Give your vote!” you send the envelope.

The part below is for getting information about the soul deposited by a candidate. Inserting the address of the candidate and clicking on the button “Read the soul!”, you obtain as response how many soul was deposited by the indicated candidate. When the tab is shown, when you vote, and when you click on the button “Update info” the information about the voting phase is updated.

Clicking on the 3rd image of the frame, you see the tab to open the envelopes sent before, and it is like the one in the below picture on the left. Here is shown information about the number of open envelopes and the number of envelopes to open (that has not been opened yet). To open an envelope, insert, where it is indicated, the information about it and click on the button “open your envelope!”. The information about open envelopes is updated when you open an envelope, when you click on the button “Update info”, and every time the tab is shown.





Clicking on the 4th image you access the tab to calculate and get the results of the voting process (it is the one shown in the picture above, on the right). Clicking on the button “Get results” the winner is calculated and shown and to the winner is also shown the bonus received (the bonus is equal to the sum of the deposits of the other candidates plus the soul received by the votes). It is indicated if there is no winner. It is also shown the list of the candidates with, for each candidate, the soul and the number of votes received.

The last tab is the one for asking for the refund and, like the one for the results, it has only one button (“Ask for refund”), as shown in the following picture. When the refund is available the number of souls that are given back is indicated.

The phases must be executed in order: first the deposit by the candidate, then the votes, the opening of the envelopes, the calculation of the results, and, in the end, the requests for refund. If you try to execute one of these functions before its phase, you will obtain an error, and it will be shown on the graphical interface.

The main decisions made during the implementation of the smart contracts and the front end.

The smart contract is composed of the following methods:

1. *deposit*: used by candidates to deposit the soul before the voting phase.
2. *cast_envelope*: used by the voters to vote.
3. *open_envelope*: used by the voters to open their envelope.
4. *mayor_or_not_mayor*: used to calculate the results of the voting.
5. *ask_refund*: used by voters to ask for the refund.
6. *compute_envelope*: to compute the envelope to send.
7. *getCandidates*: to obtain the list of the candidates.
8. *getDeposit*: to obtain the deposit of a specific candidate.
9. *getcandidateVote*: to obtain the information about the votes received by a candidate.

Furthermore, there are the getter methods for the *winner*, the *escrow,* and the *voting_condition* because these attributes are declared as public.

Each phase is well divided from the previous one: there is first the deposit phase, then the voting phase, the opening phase, the calculation of the results, and at the end, the phase for asking for the refund. To separate these phases the “require” are used, and they are defined in the modifiers, that verify that the conditions are satisfied.

The voting phase cannot start if there is even only one candidate that has not sent soul yet. This because the deposit by the candidate is seen as a fee to confirm the candidacy. So, a candidate cannot be voted if he does not deposit some soul. To do this a counter is used, and it is incremented every time a candidate deposits some soul. When this counter is equal to the number of the candidates, the voting phase starts. So, this counter is a condition regarding the voting, and for this reason, it has been added to the *Conditions* struct. The deposit of a candidate is stored in the *Vote* data structure. Here, there are stored the deposit of the candidate, the sum of the soul, and the number of votes received by the candidate. The use of this structure makes it simple to calculate the winner and divide the deposit between the voters of that candidate. The sum of the deposits is saved in an attribute.

After the deposit, the voters send the envelopes and then they open them. In the function for the open, it is controlled if an envelope is opened for the first time or not. This control is used to avoid that someone opens his envelope more and more times to reach the quorum of opened envelopes. In this way, everyone can open his envelope exactly once. When this function is called, if the calculated envelope coincides with the one sent previously, the soul sent (*msg.value*) is summed to the soul associated with the voted candidate, and his number of votes is increased by 1. Furthermore, the *msg.value* is added to the attribute previously used to sum the deposit, having, in the end, the sum of all the soul received by the contract in this attribute. The use of these accumulator variables allows reducing the need for gas from the following phases. For

example, if there is not a winner at the end, to calculate the soul (the ether) to send to the escrow account, without the use of this accumulator variable, we should use a for loop on the voters and the candidates to sum all the deposits and the soul sent from the votes. In this way, the gas needed grows up with the number of voters and candidates. Summing the soul to each calling of the function to vote or deposit soul, allows us to distribute the cost (in gas) of the sums between the calls, instead of having all the cost in one call. In this way for these functions, we can use a gas limit smaller than the one we should use otherwise, because the functions with for loops may require more easily more gas than the one indicated as the gas limit.

The soul and the symbol voted by a voter are saved in the *Refund* data structure because we must know who has voted for losing candidates and how many souls he has sent.

In the function *mayor_or_not_mayor*, the winner is calculated and the soul for him are transferred. If there is not a winner, all the soul received by the contract are transferred to the escrow account.

The refund of the voters is implemented in a specific function. All the voters can ask for a refund. If there is a winner and they have voted him, they receive a part of the deposit of the winner, while if there is a winner but they voted a losing candidate, they receive back all their soul. If someone, that is not a voter, calls the function, he does not receive any soul.

These functions emit an event to give more information about the execution, and these are useful especially for the front-end part. Every function is implemented looking also to security, try to do not make attacks possible.

The function to compute the envelope is in the contract because of the front end. The front end is written in java, but the functions, to compute the envelope, encode the envelope differently even if are used the same input parameters. So, to allow to open an envelope, the function in the contract is used.

About the getter functions, these are defined as “view” because they only read the attributes, but they modify nothing. In this way, if a user calls them, he does not consume gas.

The getter function *getcandidateVote* has a modifier. This cannot be used until all the envelopes are opened. This because the information about the received votes by the candidate must be secret until all the votes are known. It is a way to keep secret the votes until everyone has voted and has made public his vote, to avoid giving voters information about who is going to win. For example, a voter cannot see how many soul a candidate has received and give more soul to the candidate he wants to vote, because this information is not available.

In the code, for loops on the array of voters have been avoided, preferring to use functions that check if the *msg.sender* is a voter to execute a certain functionality. This is for the possible problem of the gas limit cited above. For the candidates, this problem is not considered because typically, in the elections, the number of candidates is limited, while the number of voters can be very high.

Front end implementation

The front end is implemented in Java, using the library web3j that allows working with smart contracts.

The Java project (not considering the files for the demo) is composed of 4 files: *Main.java*, *Gui.java*, *Logic.java*, *Mayor.java*, where are defined the respective classes.

In the *Main.java* file there is the method main and here is implemented the connection to the network. If the connection has success, it is created an object of the class Gui to show the graphical user interface. Here, for the testing, the network used is the ganache network ("http://localhost:8545"), but web3j supports also Infura, thanks to dedicated classes.

In the GUI class is defined the graphical interface and the behavior associated with the elements of the GUI, in particular, to the buttons and the panels. All the behaviors are implemented using the methods defined in the Logic class. The complexity is not in the GUI class, but in the Logic class and the methods in the GUI simply call the methods of the Logic class. The GUI is as simple as possible because its role is to manage the interface, so the complexity is moved in the Logic class.

The Logic class is the only one to interact with the Mayor class, the java wrapper to call functions of the smart contract. Here are implemented all the operations, like the deposit from the candidates, the send of the envelopes, and their opening. In this class, there are controls to check that inserted parameters are valid, but these controls are also on the smart contract because also other interfaces may interact with it. In the Logic class, there is a method to compute the envelope, but this calls the specific function on the smart contract. Initially, the envelope was computed by the java methods, without interaction with the smart contract, using the methods of the classes to encode of the *org.web3j.abi* package, and the class *Keccak.Digest256* in the *org.bouncycastle.jcajce.provider.digest* package. The problem was that the envelope computed with these classes did not coincide with the one computed by the smart contract, so the envelope could not be opened. For this reason, the function *compute_envelope* of the smart contract is used. From the GUI the votes can be sent only to the candidates, while the contract does not control this. This control is added to the GUI because, inserting manually the addresses, the probability of vote a wrong user because a typo is high, and this check wants to help to notice the typo. This is useful especially in the demo, because in this way if there is a typo in the voting phase, the demo has not to start again. In the contract, this is not seen as a problem because a vote for an account that is not a candidate is seen as an empty vote, useful to reach the quorum.

In the Logic class, all the methods needed to the graphical interface to obtain information about the elections and to interact with the smart contract are implemented. To get information about the smart contract the getter functions defined in the contract are used, but to get information about the execution of the functions of the contract, we read the information by the transaction or by the emitted event. For example, when a candidate deposits some soul, the amount of

deposited soul is retrieved by the transaction. Even if this amount is the same as expressed in the graphical interface, it has been preferred to retrieve the information from the interaction with the contract (from the transaction) to show how it works and how we can retrieve the information from the blockchain.

It is also possible to get information about the execution of a function of the smart contract thanks to the events. An example of how to use them is in the function to ask for a refund. Here the amount of Ether received is obtained from the event *RefundObtained* of the smart contract that indicates the amount of Wei transferred.

In the Logic class the errors have been managed but they are printed on the console for having more information about them.

The Mayor class is generated automatically from the smart contract.

Instructions to execute the demo.

This project has a GUI to interact with the smart contract and, for this reason, the manual demo allows to test how the graphical interface works, but it is also provided an automatic demo. In particular, two demos are defined, to test the logic between the GUI and the contract, that simulate a case of win of a candidate and a case of a draw.

How to execute the automatic demo

The execution of the automatic demo can be done in two ways. First way:

1. Start ganache with the command: ***ganache-cli -a 20 --acctKeys keys.json***. This generates 20 accounts that will be used for the demo and saves information in a JSON file called keys.json.
2. Put the generated file in the current directory of the java project or insert the path to the file in the attribute "path" in the *Demo.java* file.

```
public static void main(String[] args) {

    String path = "./keys.json";

    Web3j web3 = Web3j.build(new HttpService("http://localhost:8545"));
```

3. Start the execution of the demo using the main defined in the *Demo.java* file.

The second way:

1. Start ganache with the command:
   ganache-cli -a 20 -m "voice lens curtain text vivid opinion glide together bicycle buzz valley erupt"
   In this way, the private keys will be derived by the phrase indicated in the command.
2. Start the execution of the demo using the runnable demo.jar, clicking on it twice (it uses the provided keys.json file).

At the end two files are generated in the current directory: *demo1.txt* and *demo2.txt* respectively with the data of the first and the second demo.

In the demo, the contract is deployed automatically by the java code.

Automatic demo

The first demo, or test, shows the following things in order:

- The balance of the accounts involved in the demo after the deployment of the smart contract.

- The errors raised if someone tries to vote, to open an envelope, to get the results, or to get the refund when these phases are not allowed, in particular before that all candidates send their soul (because this is a preliminary phase, a prerequisite to start the voting phases and, as a consequence, all the following phases).
- The soul (Ether) sent by all voters and the updated information about the number of candidates that have sent their soul, the number of candidates that have not sent them yet, and the balance (in Ether) of the voter. This is shown for each voter.
- Then it is done a similar thing than before for the opening of the envelopes.
- After that, all voters ask for results and it is shown the winner and the souls and votes received by each candidate in a list. The winner sees also the bonus received, that is the amount of ether that he gets from the other candidates and from the voters that have given their vote to him.
- After that, each candidate and voter ask for a refund and it is shown the amount of the refund for that address.

In this first demo, 8 accounts are considered. The 8th account is the escrow and the 6th and 7th do not execute transactions showing that their balance does not change.

The first 4 are candidates (green in the tables) and the quorum is of 5 envelopes. For simplicity in this test, the souls sent by the candidates and by the voters are the ones shown in the following table (here the Ether are approximate, but in the results that we will see they are a little bit less because of the gas used for the transaction).

Candidates/voters	first	second	third	fourth	fifth
Initial soul	100	100	100	100	100
Soul sent (candidate)	2	3	4	5	
Soul sent (voters)	2	2	2	2	2
Remaining balance	96	95	94	93	98
refund	1	2	1	2	1
Bonus		17			
Final balance	97	114	95	95	99

Here the 1st, 3rd, and 5th voters vote for the second candidate, so he receives 6 souls by vote. The 2nd and the 4th vote the third candidate that receives 4 souls by vote. So, the winner is the second candidate that receives a bonus of 6 Ether by the votes, and 11 Ether (that is equal to 2 + 4 + 5) from the soul versed by the other candidates. So, the winner receives a bonus of 17 Ether. As a refund, the 2nd and the 4th candidates receive back their soul (that are 2) because they voted for a losing candidate. The other voters receive a part of the soul deposited by the winner, which are 3 souls. So, they receive 3 soul / 3 voters = 1 soul for each voter. In the end, the final balance is the one indicated in the upper table (approximated because of the gas).

The second demo, or test, shows the case of the draw, so all the ether goes to the escrow account. The quorum is of 6 voters and 3 voters vote for a candidate and the other 3 voters vote for another candidate. The following table shows the final results.

Candidates/voters	first	second	third	fourth	fifth	Sixth	escrow
Initial soul	100	100	100	100	100	100	100
Soul sent (candidate)	2	3	4	5			
Soul sent (voters)	2	2	2	2	2	2	
Remaining balance	96	95	94	93	98	98	100
refund	0	0	0	0	0	0	0
Bonus							26
Final balance	96	95	94	93	98	98	126

Manual demo

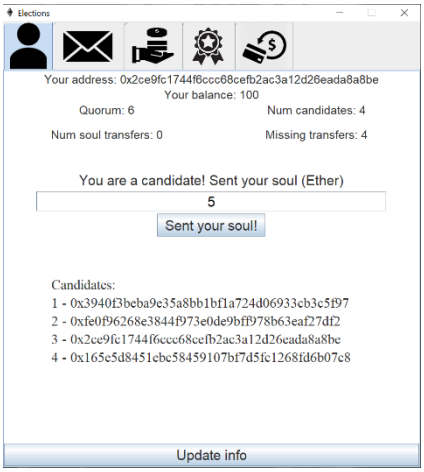
To execute the manual demo:

1. Compile the contract with the command: ***truffle compile.***
2. Start ganache: ***ganache-cli***. Remember what the generated private keys are. They are needed to use the graphical interface because they are used to authenticate the user.
3. Deploy the contract on the network: ***truffle migrate.*** Remember what the indicated contract's address is. For the demo, the candidates are the accounts with indexes 1, 2, 4, and 6 (green in the table. The indexes start from 0). The escrow is the account with index 5 and the **quorum is 6**.

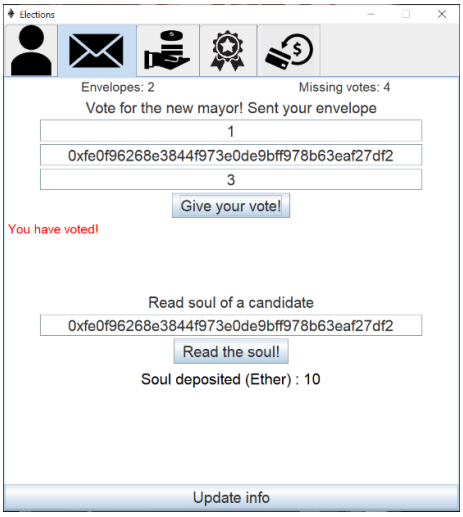
   The demo will use the parameters indicated in the following table and will show a win for the number of votes but with equals received souls.

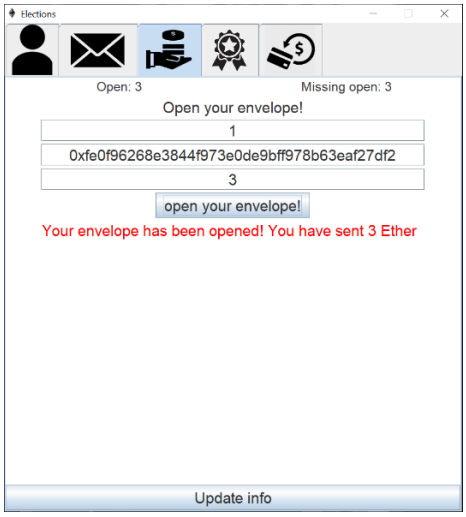
Candidates/voters index	0	1	2	3	4	5 (escrow)	6
Initial soul	100	100	100	100	100	100	100
Soul sent (candidate)		3	10		5		7
Soul sent (voters)	0.4	3.4	3	3	1.6		5
Remaining balance	99.6	93.6	87	97	93.4	100	88
Voted candidate	4	1	2	4	4		6
refund	1.6666	3.4	3	1.6666	1.66	0	5
Bonus					25		
Final balance	101.266	97	90	98.66	120.066	100	93

4. Start the java program using the main in *Main.java* file (not the one in *Demo.java*) for each candidate and voter.
5. For each candidate:
    - Insert the private key of the candidate and the contract address and click on “Confirm”.
    - Insert the soul indicated in the table above for that candidate and click on “Sent your soul”.

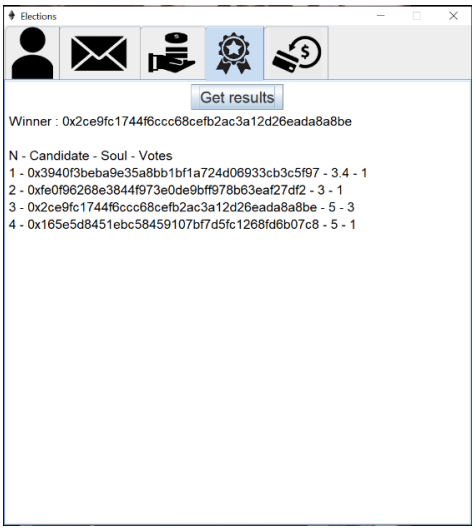


6. For each voter:
    - Insert the private key of the voter and the contract’s address and click on “Confirm”.
    - Click on the envelope image.
    - Insert, in this order, the sigil, the address of the candidate, and the soul as expressed in the table above. The sigil can be any positive number.
    - Click on the button “Give your vote!”. It will appear the phrase “You have voted!” as in the picture below on the left.
    - Here you can see the soul deposited from the candidates inserting one of their addresses and clicking on the button “Read the soul!” (as shown in the picture).
7. After sending all the envelopes, for each voter:
    - Click on the image of the hand with the coins.
    - Insert the same information used to vote in the same order and click on “open your envelope!”. It will appear the confirmation of the opening as shown in the picture below on the right.

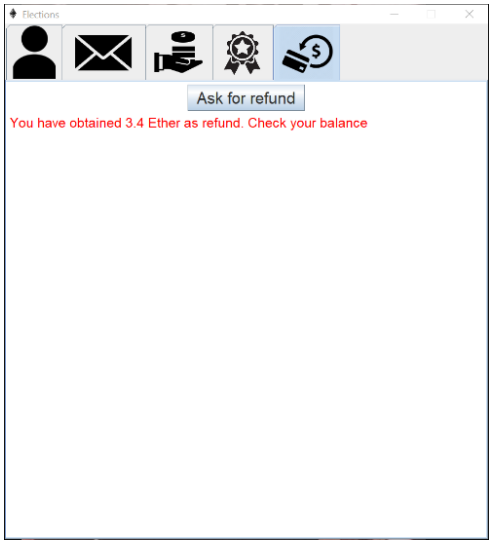




8. After the opening of all the envelopes, use some accounts to verify the results:
    - Click on the 4th image and click on "Get results". Here the list of candidates with the received souls and votes is shown. If it is the winner to ask for results, it is also indicated the received bonus (as you can see in the following pictures).



9. For each voter:
    - Click on the 5th image (the one with the credit card) and click on "Ask for refund". The voter is refunded, and it is shown a confirmation about the refund that indicates the amount of soul (Ether) transferred (as shown in the picture on the right).



10. Looking at the balance of the accounts that are been used, the balance is around the one indicated in the table above. It is a little bit less because of the usage of the gas to execute some functions of the smart contract.