Peer to Peer Systems and Blockchains
Final Project
The attempt of a democratic election system

gianmaria di rito - 544013

Academic Year 2020/2021

1 Structure of the project

The project is structured as a Distributed Application GUI. The Dapp was implemented thanks to web3 and truffle / contract libraries with which is possible to write an application in witch main logic is not executed by a single server but by a decentralized network like Ethereum. The Backend is (partially) composed by smart contracts while the Front end is any software application that use Web3. The project is therefore composed only of the Truffle folder **Final Project** in which are noteworthy:

- the folder **build** in which there is only the folder **contracts** inside which we can find the JSON files of the contracts that have already been compiled (using the truffle command **truffle compile**).
- the folder **contract** which contains the **.sol** files of the contracts that are part of the project.
- the **migrations** folder which contains the javaScript files to deploy the **.JSON** files of the compiled contracts.
- the folder **node_modules** created automatically after the initialization of a node project within the main folder (via the command **npm init**).
- the folder **src** which in turn contains:
  - a **css** folder containing the beauty.css style file for the html interface;
  - a folder **js** containing the javaScript file **app.js** that is going to implement the frontend and call the smart contract;
  - an html file **index.html** which will implement the graphical interface.
- the configuration file **bs-config.json** to tell lite-server the folders it needs to look at to serve the web application.

- the file **package.json** which contains the various dependencies and scripts to be run automatically with the **npm run dev** command

2 User Manual

1. install and log into **MetaMask** on a web browser (google chrome). If everything went well, MetaMask plugin has been added to the browser.
2. open the folder Final Project in an IDE like Visual Studio Code and open a terminal in this folder
3. run `npm install --save web3`
4. run `npm install --save @truffle/contract`
5. Make sure you have Node.js (>= v8) installed and run `npm install -g ganache-cli`
6. run `npm install --save lite-server`
7. run `ganache-cli`
8. open another terminal in witch:
   - run `truffle migrate --reset`
   - run `npm run dev`
9. a web page will open, in which to try the demo it will be necessary to login with MetaMask and import the accounts created by ganache-cli

3 Main decisions of implementation

The main elements of the Distributed Application I created are:

- the smart contract **Mayor.sol** which implements the rules followed by the Valadilène voting system;
- the JavaScript file **app.js** which, as mentioned previously, interacts with the smart contract and implements the front-end.
- the **index.html** file which implements the real graphical interface with which the end user will interact.
- the css file **beauty.css** which implements the graphical interface style.

3.1 Mayor.sol

First of all I start by saying that the extra propose, proposed by the election plan committee that I have chosen to implement is the **Pammerellum**. I have, therefore, foreseen the possibility for candidates in the elections to be part of coalitions that will naturally be valid if and only if they are composed of at least 2 people. This possibility has been implemented through a structure whereby each canidate, when presenting himself to the elections, provides 2 addresses: the first which, indicates his presentation as a single person and the second (possibly null) which indicates the coalition he belongs to.

```
struct Candidate {
    address payable cand;
    address payable coal;
}
```

There is therefore an array of `Candidate` for saving all candidates who have presented themselves: `Candidate [] public candidate_list`.

Then there are 2 other data structures, a map and an array, used to keep track of the various coalitions formed. In fact, inside the constructor, the first one will save, as a map key, the address of each new coalition and will keep as a value for each of them the number of people present who are part of the coalition. The second will contain only the address of coalitions with at least 2 members and will be useful in the smart contract to iterate on them.

```
mapping(address=> uint32) coalition_list;
address [] public  coalitions;
```

Worthy of note is the **number** structure created in order to manage the Pammerellum case in which several candidates have the same amount of souls.

```
struct number{
    uint32 numV;
    uint amount_souls;
}
```

It is exploited as a value in the **votes** map which keeps track of the various votes and souls received by each candidate and each coalition.
`mapping(address => number) votes`.

Finally there is the integer variable **tot_votes** which keeps track of the entire amount of soul received by the voters to handle the case of a coalition victory with soul `> = 1/3`.

The other variables and structures used do not differ from those present in the smart contract of the previous implementation of the Valadilène voting system.

Then there are the various functions of the smart contract:

-
  ```
  constructor(Candidate [] memory _candidate_list,
              address payable _escrow,
              uint32 _quorum) public {...}
  ```

  Which takes as parameters an array of candidates of type **Candidate**, an address **escrow** and a **quorum**. It initializes the local data structures of the contract concerning those who have presented themselves to the elections, and the variable **voting_condition** which keeps track of the conditions for accessing the vote, opening the envelopes containing the votes and the result of the vote.

-
  ```
  cast_envelope(bytes32 _envelope) canVote public{...}
  ```

  Which takes as a parameter an envelope encrypted by the **compute_envelope** function and only takes care of updating the count and data structures relating to the votes received.

-
  ```
  open_envelope(uint _sigil, address _symbol)
                                 canOpen public payable{...}
  ```

  Which takes as parameters the **sigil**, an integer that identifies the vote of the voter and the **symbol** which is the address of the candidate they voted for (it can also be a coalition). In practice, this function is called by any voter, who has previously sent their vote using the **cast_envelope** function, to concretely send the soul specified in its envelope to the smart contract. The main function of open_envelope is to update the entry of the address **symbol** in the **votes** structure by increasing the **amount_souls** field with the souls sent by the voter who called the function, update the **souls** and **voters** data structures relating to voter data, and update the conditions to allow the computation of the voting result.

-
  ```
  mayor_or_sayonara() canCheckOutcome public {...}
  ```

  This is the function that computes the result of the votes and returns the souls to the voters of the losing candidates if it is allowed. It includes all the outcomes specified in the Pammerellum. For a more in-depth analysis I refer the reader to the comments in the smart contract code.

-
  ```
  compute_envelope(uint _sigil, address _symbol, uint _soul)
                              public pure returns(bytes32){...}
  ```

  It is the function that deals with encrypting the voters' envelopes.

-
  ```
  getCandidates() public view returns(address[] memory){...}
  getCoalitions() public view returns(address[] memory){...}
  ```

  They are view type functions that return the list of single candidates and the list of coalitions participating in the election. They were created to make the contract interact with the front-end.

3.2 app.js

It provides in addition to the functions to initialize web3 and the contract:

- **listenForEvents** inside which there are listners for the various events that the Mayor.sol smart contract can launch:
  - **EnvelopeCast**: print "envelope has been sent" on the interface in case a vote has been sent.
  - **EnvelopeOpen**: prints "envelope has been opened" on the interface if a voter's vote has been opened.
  - **NewMayor**: print "new mayor" with the address of the Mayor on the interface in case a mayor has been elected.
  - **Sayonara**:print "sayonara" on the interface if no mayor has been elected and the election will have to be repeated.
- **render**: in this function all the various data necessary for the elements of the graphic interface are requested from the smart contract. Specifically, the list of candidates and coalitions necessary to build the selector through which the end user can select the possible candidates and coalitions to be able to vote.
- **pressSend**: in this function, first of all, the data necessary for sending the vote by the end user are collected from the various input elements of the graphic interface. Having decided to hide from it the two steps that make up the sending of the vote (compute_envelope and cast_envelope), in the function the **compute_envelope** function will be called first, to which the data previously collected will be passed as parameters from the interface and then **cast_envelope** to which the result of the first one will be passed as a parameter and an additional field **from** indicating the caller of the function that we will set with the address of the user who is interacting with the interface.
- **pressOpen**: in this function the data is collected by the graphic interface to call the Mayor.sol open-envelope function. Then call open_envelope by inserting in addition to the expected parameters of the function:
  - the account of the user interacting with the interface as the caller of the function (**from** field).
  - the value entered in the soul field of the interface as additional ethers for the transaction that will be created by this call (**value** field).
- **pressMayorOrSayonara**: is the last function implemented by the front-end and takes care of calling the mayor_or_sayonara function of the smart contract.

3.3 index.html

Implement the html web interface that we can divide into two phases:

- the initial one in which a loader will be visible until a user log in MetaMask with an address.
- the next one in which the graphical interface for the voting system will be visible which includes:
  - the address of the user connected with Metamask
  - the address of the counter escrow
  - an input box for inserting the **sigil** field
  - an input box for inserting the **soul** field
  - a selector where the addresses of the canidates and coalitions are visible from which the user can choose who to vote
  - the "send envelope" button which will call the **pressSend** function of **app.js**.
  - the "open envelope" button which will call the **pressOpen** function of **app.js**.
  - the "MayorOrSayonara?" which will call the **pressMayorOrSayonara** function of **app.js**.

3.4 beauty.css

As mentioned, this file implements the style of the graphical interface and includes:

- the statement for inserting an image as the background of the interface
- the loader style statement that appears when opening the html page.

4 Instructions to execute the demo

Before starting with the steps for the demo I show the situation in which we find ourselves. We have 10 accounts offered by ganache-cli that we can imagine as an array of 10 addresses that we will use both for the deployment of the contract, which will then be passed to the contract constructor to act as candidates, and as voter addresses that we will import later in MetaMask . I decided to use 6 of them as candidate and coalition addresses plus one for the escrow address. They are passed to the contract being deployed in the **2_deploy.js** file located in the **migrations** folder. The situation of the candidates imagined is therefore the following:

Candidate	Candidate Address	Coalition Address
1	Address[0]	Address[2]
2	Address[1]	Address[2]
3	Address[3]	Address[5]
4	Address[4]	Address[5]

The other parameters passed to the constructor are:

- **escrow** = Address[6]
- **quorum** = 3

The 3 remaining addresses offered by ganache-cli will be used as voters of the Valadilène voting system. I decided to set up the demo with only 3 voters and 6 addresses as single candidates and coalitions because this starting situation gives the possibility to test all the cases of voting results provided for by the Pammerellum. The demo will include the testing of a single case of the Pammerellum; in particular the one in which a coalition wins because it receives souls `> = 1/3` of the total souls and is the coalition with the most souls;

4.1 Account Setting

1. enter in the project folder within an IDE such as Visual Studio Code, open a terminal and run the command `<ganache-cli>`
2. open another terminal and run `<truffle migrate --reset>`
3. in the same terminal run `<npm run dev>`
4. at this point an html page will open with the Valadilène voting system which is being loaded and waits for you to log into MetaMask and import an appropriate account
5. go back to the terminal where ganache-cli was launched where you can find the 10 addresses it created. Copy the key marked with the number "(7)" under "Private Key"

6. return to the html page and open the previously created MetaMask plugin (if it is not present, I refer you to the User Manual section)

7. log in with your credentials and click on the circle at the top right. This will open a menu where you have to select the "Import Account" item. Paste under "Paste your private key here:" the private key copied in step 5. Click on the "Import" button.

8. The new account imported from ganache-cli will then open with 100 ETH available but must be connected by clicking on the "Not connected" item located to the left of the account name.

9. repeat steps 5-6-7-8 for ganache-cli accounts "(8)" and "(9)"

10. At this point we are ready to access the Valadilène voting system.

4.2 Demo

Since we imported 3 accounts we would go to make 3 votes by impersonating the various voters. The following table summarizes the data to be entered in the voting system interface.

Account	Sigil	Soul(wei)	Candidate
2	1	2000000000000000000	5° in the list
3	2	1000000000000000000	1° in the list
4	3	1000000000000000000	1° in the list

4.2.1 Table legend

- Since Metamask provides a default account indicated as "Account1" those imported by us will be called "Account2", "Account3", "Account4"

- the souls to be entered are in wei so the figures of the souls in the table are respectively 2 Ether, 1 Ether and 1 Ether.

- The list of candidates in the interface first includes all single candidates and then all coalitions. To stick to the case we want to experiment, the vote of the first voter will be destined for a coalition while the other two to the same single candidate. In particular, according to the Pammerellum, in this case the coalition voted by the first voter will be elected mayor as it has received an amount of soul `> = 1/3` of the total soul paid by all voters.

4.2.2 Steps of the Demo

1. Access the first imported account (”Account2”) by scrolling them from the circle at the top right of the MetaMask interface and reload the html page twice

2. By impersonating the voter who owns the account, we will enter the data in the relative fields of the voting system interface in the row relating to ”Account2” of the table previously shown (in particular the ”Sigil”, ”Soul” and ”Candidate”)

3. press the ”send envelope” button; a MetamMask page will open with the recalled transaction; click on ”Confirm” in it

4. repeat steps 1-2-3 for ”Account3” and ”Account4”.

5. Moving now to the ”Open Envelope” phase, essentially you will need to repeat steps 1-2-3-4, paying attention in step 3 to press the ”Open envelope” button instead of the ”Send envelope” button.

6. After the ”Open Envelope” phase, just press the ”MayorOrSayonara?” impersonating any of the voters to activate the transaction, which if accepted, will lead to the result of the vote.