UNIVERSITÀ DI PISA

Peer to Peer Systems and Blockchains

Academic Year 2020/2021

The attempt of a democratic election

Luca Santarella

Contents

Chapter 1

Infrastructure and Tools

The project requested to modify the smart contract by implementing new functionalities such as offering the choice of choosing among different candidates and use souls as an ERC 20 (Ethereum Request for Comments) token. Also, it was requested to implement a GUI (Graphical User Interface) which allows to use the contract using a web page.

1.1 Smart Contract Implementation

The project uses two smart contracts: the **SOULToken** contract which implements only two functions, the constructor which creates the ERC-20 token SOUL with an initial supply provided and the function `fundVoters()` which is used to divide equally the total supply of SOUL among voters. This contract uses the implementation provided by the ERC-20 contract OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts) which follows the ERC-20 interface (https://eips.ethereum.org/EIPS/eip-20).

The **democraticElection** contract implements the actual election using the SOUL token and with a multiple candidates list option, providing the main functions `open_envelope()` which is used to open the envelope for the single voter and `mayor_or_sayonara()` which is used to establish whether there has been a tie (same amount of souls and votes) or

a new mayor has been elected. In order to keep track of the preferences during the election an extra mapping `preferenceCandidates (address => Preference)` was used to associate the address of the candidates with an object `Preference` which stores the number of votes and the amount of soul, this was done to ease the counting part at the end of the election. The constructor method now takes also the address of the instance of the SOUL contract as input, so it has to be deployed before the election contract. At the end of the main functions an event is emitted which is then used in the GUI to inform the end user about what is going with the smart contract.

1.2 Graphical User Interface Implementation

The GUI is structured as a single web page which was made using HTML (HyperText Markup Language) for the structure of the website, JavaScript (JS) for the interaction with the back end implemented by the smart contract and CSS (Cascading Style Sheets) for the style with the aid of Bootstrap (https://getbootstrap.com/).

Furthermore, the libraries web3.js (https://web3js.readthedocs.io/en/v1.3.4/) and truffle-contract.js (https://www.npmjs.com/package/@truffle/contract) were used to interact with a web3 provider such as Metamask (https://metamask.io/) and to have an Ethereum contract abstraction which made the usage of the functions from the smart contract easier.

Figure 1.1: The web page is made using HTML, JS and CSS

Although Metamask is used as web3 provider and Ethereum wallet, the accounts which are used during the election come from Ganache (https://www.trufflesuite.com/ganache) which is a personal Ethereum blockchain which can be used to run tests. The web page runs on the lite server provided by NodeJs (https://nodejs.org/it/), i.e. lite-server which is a node module which can be downloaded using npm (Node Package Manager https://www.npmjs.com/).

Figure 1.2: Metamask, Ganache and Truffle are main tools used to interact with smart contracts and the Ethereum blockchain

The Dapp is structured as a single Object with attributes and functions which rely on the functions provided by the smart contract. The flow of execution is one single function after the other and the creation of event listeners and function calls which had a blocking behaviour for the user by using the JavaScript Promises (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise), this allowed to wait the input from the user to go on with the execution and call the functions from the smart contract.

The accounts from Ganache are obtained parsing the JSON (JavaScript Object Notation) file `keys.json` which contains addresses and keys of accounts that can be used to transact, these accounts are displayed to the user who will need to import the keys in Metamask. The first account is used both as deployer account and escrow, so this account will be the one in charge to initialize the Dapp by creating an instance of the SOUL token contract, funding the voters and finally creating the instance of the election contract. Next the user will need to import the keys from every single voter and then proceed to compute, cast and finally open the envelope, a form is provided to the user where they can input the required data such as the sigil, the symbol (which is displayed as the name of the candidate) and the amount of soul. After each one of the voters had opened their envelopes the deployer account will click on a button which will call the function `mayor_or_sayonara()` which will declare the new mayor.

Chapter 2

Dapp

2.1 Initialization of the environment

The application has been tested in the Windows Subsystem for Linux (WSL) using a Ubuntu 20.04 distribution with NodeJS 16.3 and npm 7.15.1. To begin with the setup it is necessary to open the Command Line Interface (CLI) and then download and install NodeJS (https://nodejs.org/en/download/) and the Node Package Manager, then you will need to run the command `(sudo) npm install -g truffle` which will install Truffle and then the command `(sudo) npm install -g ganache-cli` to install Ganache. Next, you will need to create a new directory named for instance ”p2p_prj” and the initialize the project with the command `truffle init`, then copy the contents of the submission of the final project inside the new directory ”p2p_prj”. Install the OpenZeppelin node module for the ERC20 contract implementation with `npm install @openzeppelin/contracts` and compile the smart contracts with the command `truffle compile`.

Start a local Ethereum development network with Ganache with the following command `ganache-cli --account_keys_path keys.json -a 20 2>&1 > gn.log &`. The option `--account_keys_path keys.json` will create a JSON file which contains keys and addresses of the new generated accounts, `-a 20` will create a total of

twenty accounts and `2>&1 > gn.log &` will allow to have a redirect of the output to the file "gn.log" and to run Ganache in background. To install lite-server, which is a lightweight development only node server type the command `npm install --save lite-server`.

2.2 Run the Dapp

In order to run the Dapp you will need to start lite-server with `npm run dev` and make sure that Ganache is running in background and that Metamask is installed and set in the default browser. Lite-server will serve and open a single page at the localhost:3000 address by default, showing as in figure 2.1.

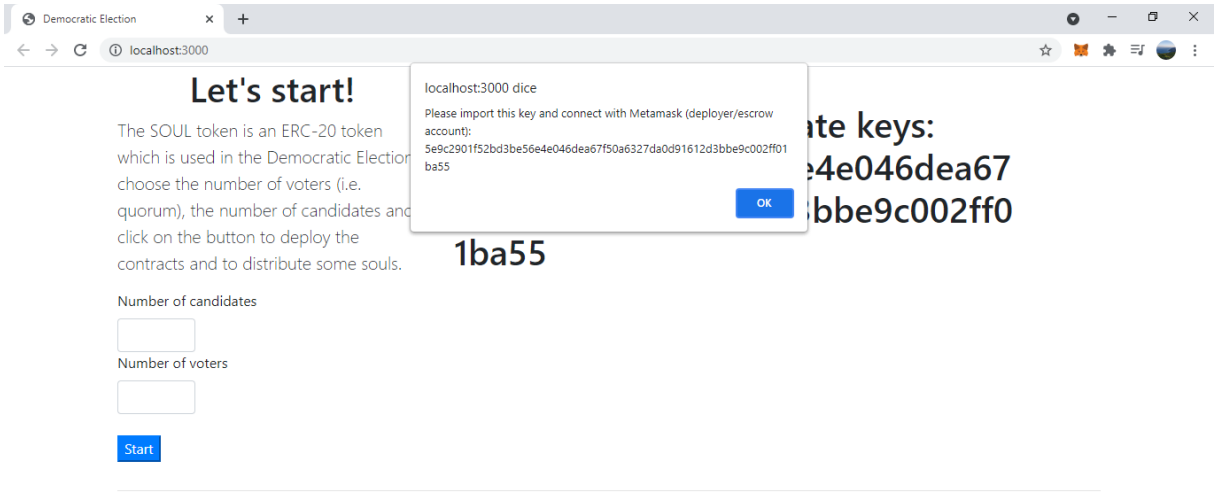
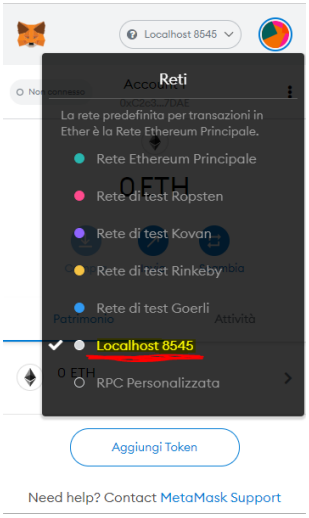
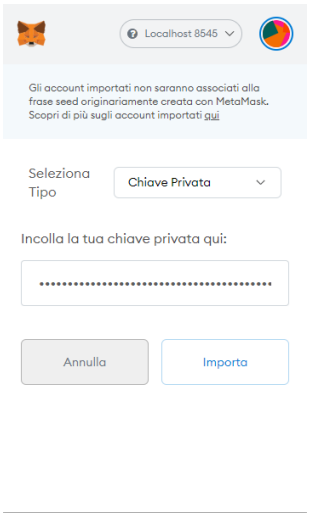


Figure 2.1: This is how the web page should appear the first time

Firstly make sure to switch to the development network provided by Ganache indicated as "localhost:8545" in Metamask as shown in 2.2a and to import the key of the deployer account as shown in 2.2b. Next, after entering the number of candidates and voters by pressing the "Start" button a Metamask notification will appear asking if the user wants to connect to the web page (see 2.3a), go ahead pressing the confirm button. After connecting another Metamask notification will pop up asking the permission to create a new instance of the SOULToken smart contract (see 2.3b, press the confirmation button to accept.

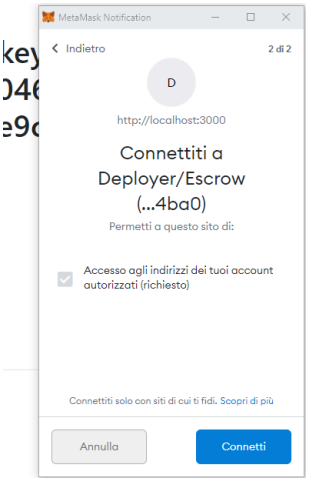


(a) Development network provided by Ganache

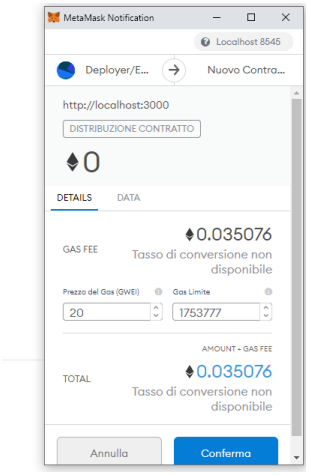


(b) Import the account of the deployer

Figure 2.2: First setup



(a) Connect the account of the Deployer with Dapp

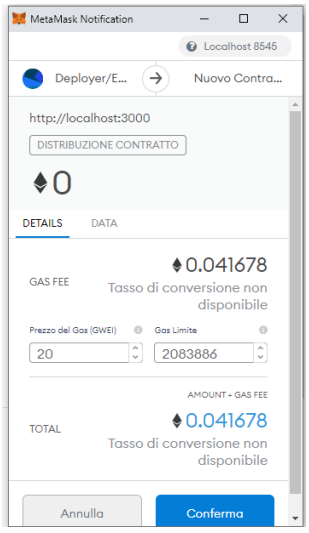


(b) Creation of the SOUL token

Figure 2.3: Starting the Dapp

Next, Metamask will appear once again to get the permission for the execution of the function `fundVoters()` which divides the total supply of soul among all the voters (see 2.4a) and lastly Metamask will ask if the user wants to confirm the creation of the election smart contract (which requires that the SOUL contract has been already deployed) as shown in figure 2.4b.

(a) The function called will fund the voters with souls



(b) Creation of the election smart contract

Figure 2.4: Last step of the initial setup

After the initial setup, a new interface will appear which shows a form which the user needs to fill with sigil, symbol (displayed as names instead of addresses) and amount of souls to compute the envelope. At the bottom of the page there is the list of candidates with the association between name and address and the list of voters which shows their private keys (that need to be imported in Metamask), their addresses and their balances in souls. On the right side of the page a text message shows the current step by indicating the action which is pending, e.g. in figure 2.5 the text shows that the user needs to compute the envelope with the account of voter 1.

Therefore the user needs to import the account of the first voter and remember to **connect the account to Metamask** by clicking the dedicated button in the top left corner of the Metamask interface and compute the envelope. After clicking on the compute button the interface will show the envelope and a button ”Cast envelope” which when clicked will start the contract function `cast_envelope()` as shown in 2.6.

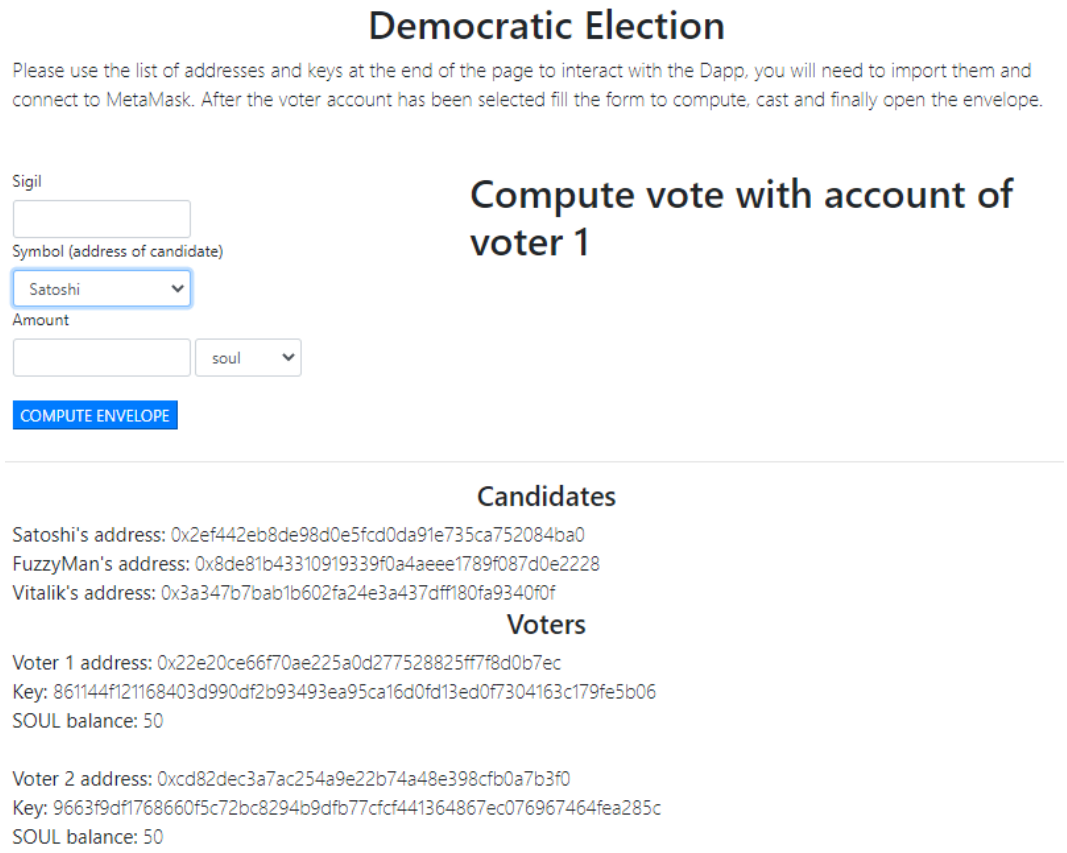


Figure 2.5: The compute envelope interface also shows list of candidates and voters

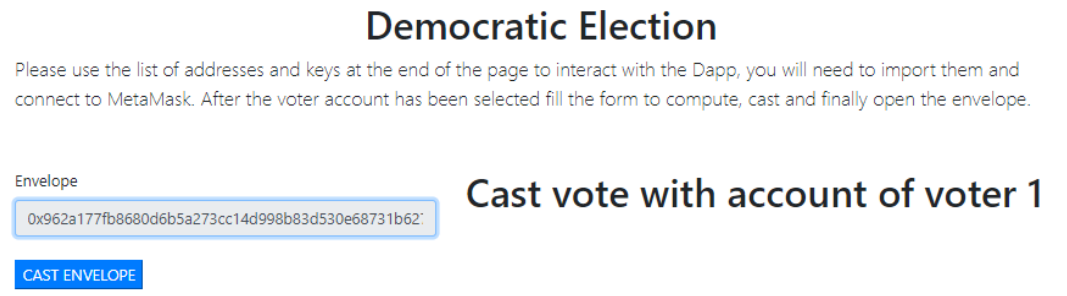


Figure 2.6: The computed envelope that will be casted

This step of switching and connecting accounts, computing and casting the envelopes must be repeated for each one of the voter. When every voter has casted their voter the web page will ask to switch to the first voter account and open the envelope. The parameters of the envelope in the form are already filled in by default since this Dapp is just a demo, so the user will just need to click on the open envelope button, approve with Metamask the allowance of the souls token to the election contract and finally confirm the transaction.

When all users have opened their envelopes the user will need to switch the deployer account (e.g. the first account that was imported) and click on the button elect (see 2.7) to call the contract function `mayor_or_sayonara()` to find out who is going to be the next candidate.

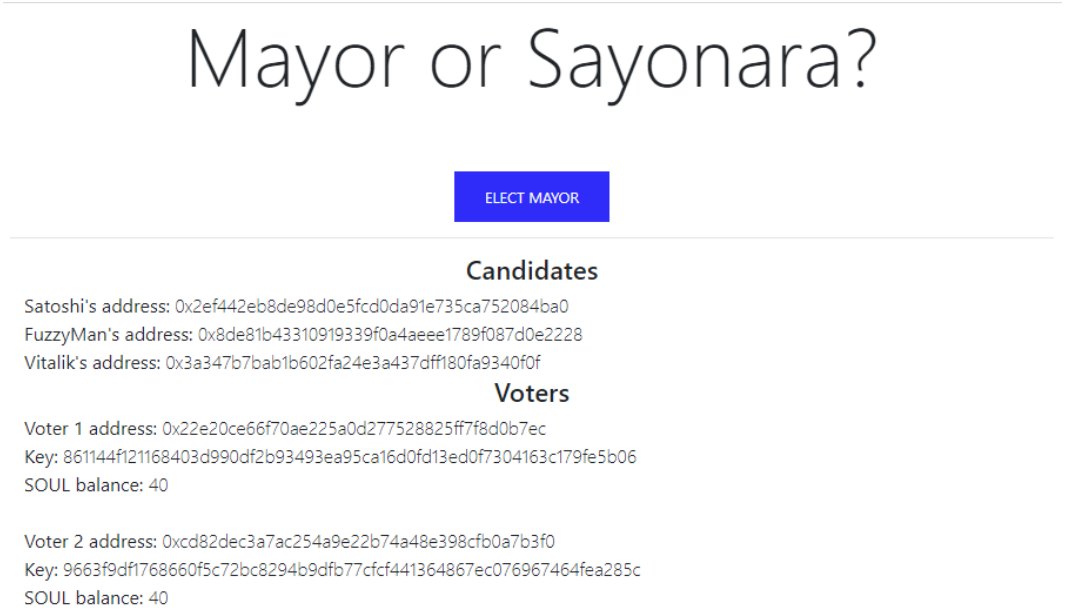


Figure 2.7: The deployer account will need to click on the button to elect the new mayor

The web page will show a different text whether there has been a tie or a new mayor has been elected, in case of victory the name and address of the candidate will be displayed on the screen with the amount of smolsoul (1e-18 soul) and votes that made him won (see 2.8). On the other hand in case of a tie a text warning the user will be displayed in addition of information about the amount of souls and the address of the escrow (i.e. the deployer account) where the souls have been sent (see 2.9). To restart the Dapp simply refresh the page.

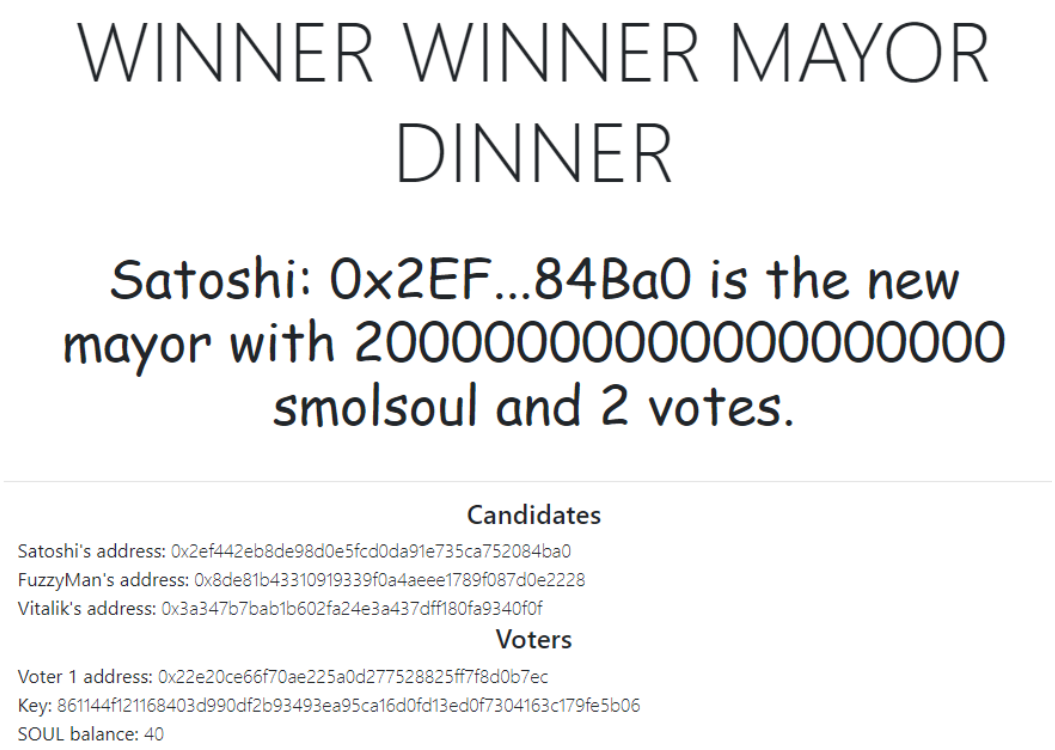


Figure 2.8: The first candidate has won with 20 souls and 2 votes

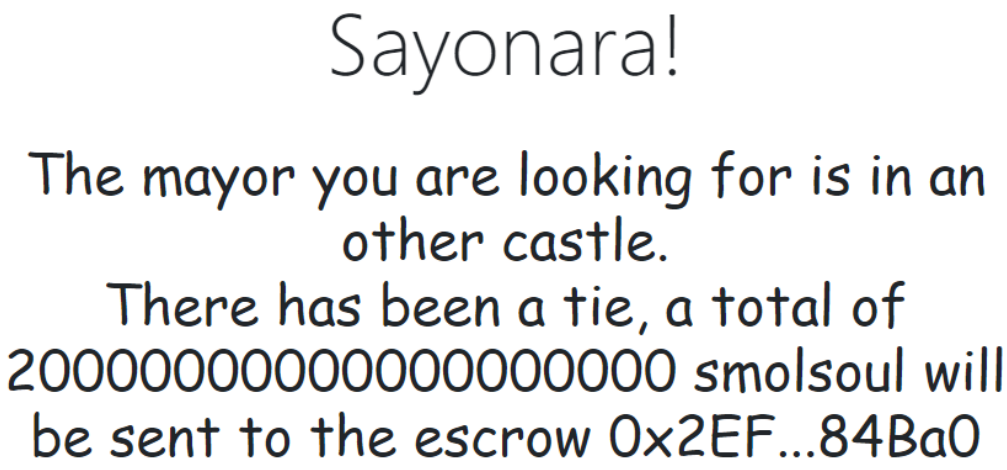


Figure 2.9: The election ended with a tie, 20 souls have been sent to the escrow