Università di Pisa
Dipartimento di Informatica

Peer to Peer Systems and Blockchain

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

Final Term Report

1 Part 1: Ethereum and Smart Contracts

1.1 Implement the `open_envelope()` and `mayor_or_sayonara()` functions. Feel free to make any change to the smart contract state attributes

The task was done with the *truffle* suite. The *truffle* project is stored in the `/src` folder, attached to this document in the delivery package.
The subfolders in `/src` are as follows:

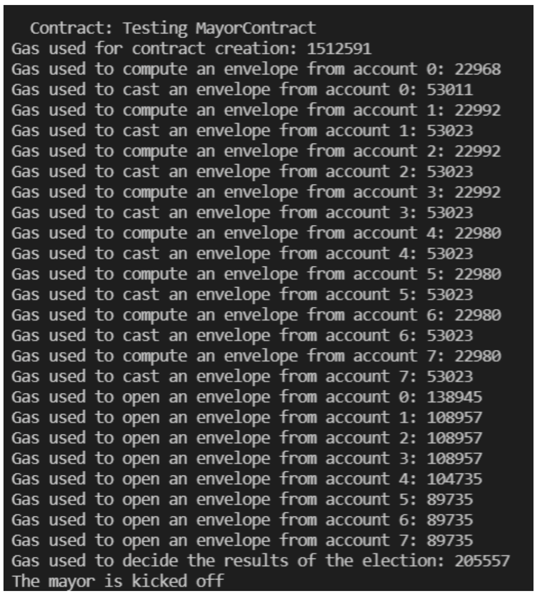
- `/build`: useless in this case, it should contains the compiled contracts
- `/contracts`: it containts the smart contract code produced, with the implementation of the required functions
- `/migrations`: it contains the code to deploy the contract to a target network
- `/test`: it contains the tests carried out with *truffle* (to execute them there is the `truffle test` command). In particular, in the same file, there are:
  - A test for the general contract behavior, where all events emitted by the contract and the balance of each account before and after transactions are checked,
  - A test to measure the gas consumption of `mayor_or_sayonara` with a high number of losing voters,
  - A test to measure the gas consumption of `mayor_or_sayonara` with a low number of losing voters,
  - A test to measure the gas consumption of `mayor_or_sayonara` with a small quorum value.

The gas cost measurements performed by these tests are visible in figure 1 and 2.

1.2 Evaluate the cost in gas of the functions provided by the smart contract

The evaluation of the gas cost of the functions provided by the smart contract was carried out using the `estimateGas()` function provided by the *truffle* suite, and is as follows (figure 1).

- `constructor`: 1512591 gas
- `compute_envelope`: 22980 (false *doblon*) and 22992 (true *doblon*) gas estimated but not really consumed if called directly, because it is pure
- `cast_envelope`: 53023 gas
- `open_envelope`: this function requires different quantities of gas if performed with different *doblon* values and whether or not performed for the first time[1], as it is visible in figure 1.



```
  Contract: Testing MayorContract
Gas used for contract creation: 1512591
Gas used to compute an envelope from account 0: 22968
Gas used to cast an envelope from account 0: 53011
Gas used to compute an envelope from account 1: 22992
Gas used to cast an envelope from account 1: 53023
Gas used to compute an envelope from account 2: 22992
Gas used to cast an envelope from account 2: 53023
Gas used to compute an envelope from account 3: 22992
Gas used to cast an envelope from account 3: 53023
Gas used to compute an envelope from account 4: 22980
Gas used to cast an envelope from account 4: 53023
Gas used to compute an envelope from account 5: 22980
Gas used to cast an envelope from account 5: 53023
Gas used to compute an envelope from account 6: 22980
Gas used to cast an envelope from account 6: 53023
Gas used to compute an envelope from account 7: 22980
Gas used to cast an envelope from account 7: 53023
Gas used to open an envelope from account 0: 138945
Gas used to open an envelope from account 1: 108957
Gas used to open an envelope from account 2: 108957
Gas used to open an envelope from account 3: 108957
Gas used to open an envelope from account 4: 104735
Gas used to open an envelope from account 5: 89735
Gas used to open an envelope from account 6: 89735
Gas used to open an envelope from account 7: 89735
Gas used to decide the results of the election: 205557
The mayor is kicked off
```

Figure 1: This picture represents the values obtained for the gas consumption of the functions provided by the contract, where the first four accounts give a true *doblon*, while the second four a false *doblon*.

  The very first time this is executed, it requires additional 15000 gas, probably to initialize the `voters` array and the `souls` mapping, then:

  - the first time it is executed with true *doblon* it requires 123957 gas
  - the first time it is executed with false *doblon* it requires 104735 gas
  - the next times, when executed with true *doblon* , it requires 108957 gas
  - the next times, when executed with false *doblon* , it requires 89735 gas

  So, in addition to the 15000 gas consumed for the first time ever, the function takes another 15000 gas, probably to initialize the `yaySoul` variable, used for the first time when we have the first true *doblon* , and `naySoul` variable, used for the first time when we have the first false *doblon* .

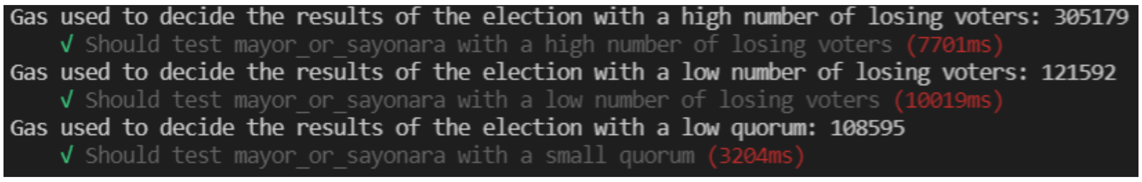
- `mayor_or_sayonara`: the quantity of gas consumed by this function strictly depends on the number of votes needed to reach the quorum and on the number of losing voters, because it must send their *soul* back to them.

[1]This evaluation was done with a quantity of *soul* sent between 100 and 1000.

1.2.1 Provide 2 or 3 cost variations of the `mayor_or_sayonara()` function, for example by fixing the quorum and varying the number of losing voters, or varying the quorum, or any other method

As it is explained above, the `mayor_or_sayonara()` function strictly depends on various factors, in fact the gas consumption increases as quorum increases, in particular because there will be more losing voters to refund. Some variations of the gas consumption are shown below[2] (and in figure 2):

- With a quorum value of 8 and 4 losing voters to be refunded (end of figure 1): 205557 gas
- With a high number of losing voters (8) out of a quorum of 9: 305179 gas
- With a low quorum value of 2 and only 1 losing voter to refund: 121592 gas
- With a low number of losing voters to refund (1) on a quorum of 9: 108595 gas



```
Gas used to decide the results of the election with a high number of losing voters: 305179
    √ Should test mayor_or_sayonara with a high number of losing voters (7701ms)
Gas used to decide the results of the election with a low number of losing voters: 121592
    √ Should test mayor_or_sayonara with a low number of losing voters (10019ms)
Gas used to decide the results of the election with a low quorum: 108595
    √ Should test mayor_or_sayonara with a small quorum (3204ms)
```

Figure 2: This picture represents the values for the gas consumption obtained by calling `mayor_or_sayonara` with different values of quorum and losing voters.

1.3 What are the security considerations and potential vulnerabilities related to the `mayor_or_sayonara()` function? List and explain them

The potential vulnerabilities of `mayor_or_sayonara()` are as follows:

- **Re-entrancy**
  A procedure is re-entrant if it can be interrupted in the middle and initiated over, and both run without errors. A contract non-recursive function is re-entrant, so it can be re-entered before its termination.
  In this function, when a transaction is sent to the candidate for mayor, escrow or a losing voter, the payable function of these accounts is called. If this payable function recall `mayor_or_sayonara()`, and the fund to be sent is updated after the transaction, the transaction can be sent multiple times until the contract balance is completely drained off or, if `transfer` or *send* are used to send back *soul* , until the gas limit of 2300 is reached. In this implementation this thing is avoided by updating the balances to be refunded before the transaction, so if the function is re-entered the balance to be sent is zero.

- **`send` and `transfer` functions**
  If in this function they are used `send` or `transfer` to send *soul* to the candidate for mayor, escrow or losing voters, the contract may no longer work in the future because these functions comes with a gas limit of 2300.
  Every OPCODE has a specific gas cost, that reflects the underlying resources consumed by each operation on the nodes that make up Ethereum. For this reason these gas costs change over time, so in the future the gas limit of 2300, which already allows only a few simple operations (write on storage requires more than 2300 gas), may not be sufficient to support the required operations, and the contract should not depend on the cost of gas. Moreover, `transfer` throws an exception if the transaction fails and all the changes are reverted, while `send` simply return false in this case, so it does not even handle the exceptions.

[2]These gas consumption are estimated with the `estimateGas()` function provided by *truffle* .

- **The Replay Attack**
  When the contract transfers *soul* to a candidate for mayor, escrow or losing voter, if the recipient is an attacker, he can copy the transaction, that is correctly signed, and send it back to the network to receive the *soul* again.
  Fortunately, in Ethereum there is the nonce (transaction number) to avoid this, because two transactions with the same nonce cannot be performed, and the attacker cannot modify the nonce without invalidating the transaction signature.

- **Voting impartiality**
  Due to the fact that the gas price is decided by the caller of a function and that this determines the priority with which a transaction is mined, one faction of voters may decide to use a higher gas price than the other faction to cast the envelopes. Then, the envelopes casted for this faction will be considered first and can fill the quorum before the others are casted. This problem is not directly related to `mayor_or_sayonara()` function, but it can distort the elections.

1.4 The `compute_envelope()` function is an helper to compute an envelope. Why is its presence an issue if the smart contract would be deployed to the Ethereum network?

The presence of `compute_envelope()` when the smart contract is deployed to the Ethereum network can be an issue if the function is called inside a transaction from another contract.
This function is pure, because it does not modify the state of the blockchain, so if it is called directly or within a pure function it is not executed by the miners and by all the nodes of the network, and in fact it does not consume gas. If, on the other hand, it is called within a transaction, for example when it is called by the `open_envelope()` function to check if the envelope corresponds to the one casted, it consumes gas because it is executed by the miners and by the nodes which then verify this transaction, so its parameters and its result are visible to the entire Ethereum network.
For example, there may be a contract that implements a service that allows the user to vote under a fee. In this case when the user sends the payment, the contract sends the vote using also `compute_envelope()`, and, as this is a transaction, all nodes on the network execute `compute_envelope()` and so they can see the *sigil* , the *doblon* and the *souls* sent. Then, all nodes on the network could see the user's vote and use this information to manipulate the elections, for example by casting a vote against with the same amount of *soul* from another account, to make the previous vote useless.

2 Part 2: BITCOIN and the Lightning network

2.1 Question 1: the Double Spending Attack in Bitcoin

Double spending essentially occurs when an account tries to spend the same bitcoins twice, and it is a problem specific to cryptocurrencies, not present in cash, because they are digital files and not physical objects.
The first solution to perform a double spending attack is to basically send the two transaction involving the same bitcoins in the network, so I could try to send a transaction to buy the car and spend the same bitcoins by sending a transaction to buy the sailboat.
Every node has a *MemPool*, a temporary memory that contains a collection of all Bitcoin transactions awaiting that a consensus is reached to be included in the next mined block. The two transactions I sent are placed in the *MemPool* of the miners but only one will be inserted in the next block by an honest miner, while the other will not be confirmed in the next block. If the two transactions are validated simultaneously by two different miners, the blockchain forks but sooner or later one of the two chains will be abandoned and only one of the two mined transactions will finally be inserted on the long-term chain. For this reason, the longest chain rule tells to wait for six confirmation before accepting a transaction.

I could also be a malicious miner, so I could validate a block and include both transactions in the same block, but the other nodes check the validity of my block and reject it.
If I am a malicious miner and I spend all of my bitcoins on the honest blockchain to buy the car but I keep a private copy of the blockchain where I do not include this transaction, I still have all my bitcoins in my private blockchain. If I manage to have a longer chain than the honest one, I can broadcast my version to the rest of the network and it will be accepted due to the longest chain rule. If the attack succeeds the old chain is abandoned, so I have already received the car (because my first transaction was been confirmed in the honest blockchain) but I still have all my bitcoins (because now my copy of the blockchain is accepted as the official), which I can use to buy the sailboat.
The problem for me is that I can perform this attack only if I have more hashing power than the other nodes of the network combined, so if I control the 51% of the total hashing power (for this reason this attack is also called *51% attack*), because I need to add blocks to my version of the blockchain faster than other nodes of the network to their version.

2.2 Question 2: the Bitcoin's Lightning Network's anti-cheating mechanism

The basic idea is that Alice and Bob should have an interest to publish the most updated balances of the transaction closure. To incentivize this thing, if someone violates this rule the other party can punish him taking the full amount in the channel, so if Alice tries to cheat Bob by publishing the transaction most favorable to her, Bob can take all the *Satoshi* in the channel.
The anti-cheating mechanism used by the Lightning Network exploits two features of the blockchain:

- **Time Lock**
  The time lock essentially makes bitcoins spendable only at some point in the future, and is of two types:
  - `CheckLockTimeVerify (CLTV)`: locks the bitcoins for a specific time, they are spendable at the block N, so at a specific date and time
  - `CheckSequenceVerify (CSV)`: locks the bitcoins for a relative time, they are spendable in N blocks from now, so time is counted from now

  This thing can be exploited by Bob, who can make the transaction to Alice spendable for example in 1000 blocks from now, using a *CSV lock* with a value of 1000. In this way this transaction cannot be registered until the next 1000 blocks are mined.

- **Hash Preimages (Secrets)**
  These are strings randomly generated and impossible to guess. A *preimage*, or *secret*, is cryptographically hashed through a hashing function and the resistance of the *preimage* is that anyone who knows the *secret* can reproduce the hash, but not the other way around, because the hashing function is one-way.
  This is another way to lock bitcoins, in fact Bob can include the hash of the *preimage* in the output script, and now the hash must be present in the unlock script to unlock the output of the transaction.

In practice Alice and Bob, after opening their channel, generate two symmetric commitment transactions, which differ only in the output locking scripts. From now on, for each payment between Alice and Bob a new commitment transaction is generated and the output of these transactions contains the locks described above.
So, when Alice and Bob change their balances to 13 *Satoshi* for Alice and 7 *Satoshi* for Bob, they both generate a *secret* and send the hash of the *secret* to the other. When they receive this hash, they can generate the commitment transaction. In the commitment generated by Bob and sent to Alice, 7 *Satoshi* are specified which must return to him if Alice signs and publishes this transaction and 13 *Satoshi* on an output with a time lock of N blocks and a hash lock

including the hash received from Alice. Alice can unlock this second output if she signs the transaction, but only after that N blocks have been mined. Bob, on the other hand, can unlock it only by knowing Alice's *secret* (hashed in the *preimage* received from her). Alice does the same and generates a symmetric commitment transaction, so there are now two commitments, but only one of them can be confirmed.
Then, if Alice decides to sign and publish this transaction, which is more favorable to her, on the blockchain, Bob can immediately redeem his bitcoins, while Alice can redeem her bitcoins after a time delay, due to the anti-cheating mechanism. If during this time frame Bob detects that Alice is cheating, he can redirect the transfer to himself, but to do so he need the *preimage*, that is the *secret* that Alice generated for this payment. Alice can do the same thing if Bob is cheating. But, when Bob and Alice generate a new commitment transaction, they send each other the *preimage* used in the previous commitment transaction, revealing their previous *secret*. So, when the last transaction between Bob and Alice, which changes their balances to 11 *Satoshi* for Alice and 9 *Satoshi* for Bob, takes place, and Alice tries to publish the previous one, more favorable for her, Bob knows the *secret* used by Alice in the previosu commitment transaction, the one relating to the transaction that she wants to publish. With this *secret* Bob, in the time frame guaranteed by the *time lock*, can publish the previous transactions proving that Alice is cheating and he can redeem his bitcoins and get the full amount of the channel, punishing Alice.
Obviously Alice, the cheater, does not notify Bob when she decides to publish the transaction by cheating him, so Bob must periodically monitor the channel to detect the cheating, or he can delegates this work to a third party.