Course project – Governance and voting systems

1. *DAOs* Governance and *on chain* voting systems

Decentralized Autonomous organizations (DAOs)[1] are defined by means of public governance rules. The requirement of a set of public and immutable rules so that anyone can check that every action follows those rules, requires a transparency level that perhaps only the blockchain technology can achieve. Governance rules are described as smart contract code that is public and immutable once it is deployed on the blockchain and all actions performed (transactions) are recorded on the blockchain and are available for any user.

This model prevents many of the issues that arise on non-blockchain based governance systems, where rules are not clear and can be modified (or badly applied) following decisions of few participants, and where some actions can be hidden from the public.

One of the most common forms of governance in a DAO is a ballot: A mechanism is established to allow the DAO members express their preferences about the actions to be taken and to eventually execute those actions that have received sufficient support according to the governance rules.

Blockchain voting mechanisms may require a minimum participation or a minimum support from DAO members in order to execute a given action. This minimum support is defined by a threshold that can depend on the current state of the DAO. On the other hand, the actual value of casted votes must be defined. This way, this value may be directly related to the wealth or stake of a member in the DAO or, on the contrary, the same value for the votes from all members can be considered. A simple approach that lies in the middle uses quadratic voting, in which the cost of an additional vote grows quadratically. In this approach, wealthier members have more weight and can cast more votes, but the quadratic grow discourages casting too many votes. There are more complex mechanisms for defining the value or cost of a vote, but in this project we will use the simpler quadratic voting.

2. System for quadratic voting of proposals in DAOs

The objective of this project is the implementation of an *on chain* quadratic voting system of proposals for DAOs (`https://en.wikipedia.org/wiki/Quadratic_voting`).

The voting process is performed over a period of several days or weeks. An authorized user is in charge of opening and closing the voting period (we will use the address of the

[1]`https://ethereum.org/en/dao/`
`https://coinmarketcap.com/alexandria/article/what-is-a-dao`

voting contract creator for identifying him); no other user can perform this task. This user will receive the unspent voting budget when the voting period is over. An initial budget for proposal funding must be provided when opening a voting period. This budget can can be extended during the voting period with the tokens from votes received by the proposals that are approved.

Participants can register themselves at any time, even after the voting process is open. Participants can add new proposals during the voting process. Participants can transfer Ether at any time to buy tokens for casting votes. The Ether invested in tokens for voting proposals will be used for extending the funding budget if they are approved, or it will be returned to the participants at the end of the voting process if the proposal is not approved.

During the voting process, the participants may cast (*stake*) their votes to support the proposals as they wish. Each vote has a price measured in tokens. A participant can vote for several proposals or cast several votes for the same proposal. Nevertheless, the price of the votes grows quadratically for that proposal. In this way, a single vote for a proposal costs 1 token, but if the same participant casts a second vote for that proposal, the combined cost of the 2 votes is 4 tokens. Three votes for a proposal cost 9 tokens, and so on.

For example, if in a voting process there are 16 proposals and a participant has a credit of 16 tokens, he can cast one vote for each and all 16 proposals, or he can cast 4 votes for a single proposal (that cost 16 tokens), or any other combination (for instance, 2 votes for 4 different proposals also cost 16 tokens).

There are two types of proposals:

- **“*Signaling*” proposals:** These proposals can be casted but they have no associated budget. They are just used to express preferences in the DAO. When these proposals are voted, the tokens used for casting votes are retained during the voting period, and they are returned back to their owners when the voting period is over.
- **Funding proposals:** These proposals do have a budget. If the votes obtained by a funding proposal exceed a specific threshold, the proposal is approved. Funding proposals do not have to wait until the voting period finishes: they can be approved as soon as they exceed their threshold.

A proposal i is approved if the following two conditions hold: (1) the total budget of the voting process plus the amount collected by the received votes is **sufficient for funding the proposal**; and (2) the number of votes received **exceed a threshold** defined by the following formula:

$$threshold_i = (0{,}2 + \frac{budget_i}{totalbudget}) \cdot numParticipants + numPendingProposals$$

where:

- $budget_i$ is the budget of proposal i,
- $totalbudget$ is the total budget available for this voting process at the time this formula is computed,

- *numParticipants* is the number of participants at the time this formula is computed, and
- *numPendingProposals* is the number of funding proposals that have not yet been approved nor cancelled at the time this formula is computed (signaling proposals are not included here).

Each proposal has a different value for the threshold, and it can change along the voting period. Observe that the computation of the threshold may change under certain circumstances. **Nevertheless, the threshold of a proposal must only be computed and checked when the proposal receives votes.**

As soon as the conditions hold for a proposal to be approved, the voting contract must **execute the proposal**, transferring the budgeted funds by means of a call to an external contract that implements the `IExecutableProposal` interface. Besides, the proposal state must change to "approved", and the total budget for the voting process must be modified, among other tasks. See the description of `openVoting` below for detailed information.

You are free to design the contracts that you want to conform your project, but you must implement at least the following ones:

2.1. Proposal contracts and `IExecutableProposal` interface

Proposals are represented by contracts that must implement the interface `IExecutableProposal`:

```
interface IExecutableProposal {
  function executeProposal(uint proposalId, uint numVotes,
    uint numTokens) external payable;
}
```

When a proposal is added to the voting process, a contract address implementing this interface must be provided. When the proposal is approved, function `executeProposal` must be invoked on that address, sending the budgeted amount for that proposal in the calling transaction. You can test the system programming a small testing contract that emits an event whenever this function is invoked, informing about the balance of that contract to verify that it has effectively received the Ether in the transaction.

2.2. `QuadraticVoting` contract

`QuadraticVoting` is the contract that handles the voting process. It must include at least the following functions:

- When this contract is created, it must be provided the price in Wei of each token and the maximum number of tokens that can be used for the voting process. The constructor must create the ERC20 token contract for handling tokens. There are additional details about ERC20 in Section 3.
- **`openVoting`:** This function opens the voting period. It can only be executed by the user that created this contract. The transaction calling this function must transfer the

total initial budget that will be available for funding proposals. Remember that this total budget will be modified when proposals are approved: it will be increased with the Ether value of the tokens used for voting the approved proposals, and it will be decreased by the amount transferred to approved proposals.

- **`addParticipant`:** Function used by participants to register themselves for the voting process. Participants can register at any time, even before the voting period is started. Participants must transfer Ether when registering to buy tokens (at least one token) that will be used for casting their votes. This function must create and assign those tokens to the participant.

- **`removeParticipant`:** Function used by participants to remove themselves from the system. A participant that invokes this function cannot cast votes, create proposals nor buy or sell tokens unless it adds itself to the voting system.

- **`addProposal`:** Function that creates a proposal. Any registered participant can create proposals, but this function succeeds only when a voting process is open. This function receives all attributes for the proposal: title, description, budget (it can be zero if it is a signaling proposal) and a contract implementing the `ExecutableProposal` interface, that will be the payee (the payment recipient) of the budgeted amount in case this proposal is approved. This function must return an identifier of the proposal.

- **`cancelProposal`:** Cancels a proposal given its identifier. This function can only be executed when the voting period is open. A proposal can only be cancelled by its creator. Approved proposals cannot be cancelled. The tokens corresponding to votes received by this proposal so far must be returned back to their owners.

- **`buyTokens`:** This function is used by participants to buy additional tokens for casting votes.

- **`sellTokens`:** This function is used by a participant to return unspent tokens and receive the Ether invested in them.

- **`getERC20`:** Returns the ERC20 contract that uses the voting process for handling tokens. Participants can use this contract to operate with their tokens (transfer them, approve their use by other participants, etc.).

- **`getPendingProposals`:** Returns an array with the identifiers of all pending funding proposals. This function can only be executed when the voting period is open.

- **`getApprovedProposals`:** Returns an array with the identifiers of all approved funding proposals. This function can only be executed when the voting period is open.

- **`getSignalingProposals`:** Returns an array with the identifiers of all signaling proposals (those with no budget). This function can only be executed when the voting period is open.

- **`getProposalInfo`:** Given a proposal identifier, this function returns the data associated with that proposal. This function can only be executed when the voting period is open.

- **`stake`:** This function is used by participants to cast a number of votes for a proposal. It receives a proposal identifier and the number of votes, and casts those votes from the participant invoking this function. It must compute the number of tokens required for casting those votes and check that the participant has approved the use of those tokens to the address of this contract (`address(this)`). Remember that a participant may cast votes several times for a given proposal with a quadratic cost (with respect to the total number of votes).

  This function must transfer the required amount of tokens from the participant account to this `QuadraticVoting` contract account in order to be able to operate with them. Since this transfer is performed by this contract code, the voter must have previously approved the use of his tokens to this contract address (the token approval should not be programmed in `QuadraticVoting`: it must be done by the participant with the ERC20 contract prior to executing this function; the ERC20 contract can be obtained using `getERC20`).

- **`withdrawFromProposal`:** Given an amount of votes and a proposal identifier passed as arguments, this function removes (if possible) that amount of votes previously casted by the participant invoking this function. A participant can only remove votes that he has previously casted for that proposal, and the proposal cannot have been approved nor cancelled. Remember that this function must return the tokens used for casting those votes back to the participant (for instance, if the participant had casted 4 votes for a proposal and withdraws 2 of them, 12 tokens must be returned back to the participant).

- **`_checkAndExecuteProposal`:** **internal** function that checks if the conditions for executing a funding proposal hold and, if they hold, executes it using the function `executeProposal` from the contract provided when this proposal was created. This call to the external contract must transfer the budgeted amount for executing the proposal. Remember that the total amount available for proposals must be updated (and do not forget the amount received for the tokens used for voting the proposal that is to be executed). Besides, the tokens related to votes for this proposal must be removed, as its execution consumes them.

  Signaling proposals are not approved during the voting process: all signaling proposals are executed when the voting process is closed with `closeVoting`.

  When calling to `executeProposal` in the external contract, the maximum amount of gas must be limited to prevent the proposal contract from consuming all the gas in the transaction. This call must consume at most 100000 gas.

- **`closeVoting`:** Function used for closing the voting period. It can only be executed by the user that created the voting contract. When the voting period finishes, the following tasks must be done, among others:

  - The funding proposals that have not been approved must be dismissed and the tokens used for voting them must be returned to their owners.
  - All signaling proposals must be executed (no Ether is transferred in this case) and the tokens used for voting them must be returned to their owners.

- The remaining voting budget not spent on any proposal must be transferred to the owner of the voting contract.

When a voting process is closed, new proposals and votes must be rejected and the `QuadraticVoting` contract **must be set to a state that allows opening a new voting process.**

This function might consume a lot of gas, take this into account when programming and testing it.

3. ERC20 *Tokens*

You must use ERC20 tokens to handle the voting process. Furthermore, the implementation used for this project must allow the creation of fresh tokens for participants. To that end, you must implement your ERC20 token contract inheriting it from the code available in the repository of OpenZeppelin:

- Version for Solidity 0.8.x is available at
  `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/`
- Version for Solidity 0.7.x is available at
  `https://github.com/OpenZeppelin/openzeppelin-contracts/tree/solc-0.7/contracts/token/ERC20`

You can find more information at:

`https://docs.openzeppelin.com/contracts/4.x/api/token/erc20`

Some comments about this ERC20 implementation:

- OpenZeppelin documentation recommends your implementation to import their source files directly from Internet (using directives like `import "@openzeppelin/...";` where the symbol `@` indicates the compiler to download the code from a repository at `https://github.com/`). Your ERC20 contract must inherit from this OpenZeppelin implementation and you should implement the minimum necessary set of functions to make your system work.
- To enable the creation of fresh tokens, you must implement an external function that invokes `_mint`, an internal function that is inherited from the OpenZeppelin implementation. You can take a look at the implemenation `extensions/ERC20Capped.sol` available in the OpenZeppelin repository, but you should adapt it to the requirements of this project. In particular, remember that you must control who is authorized to execute this function: you could lose the control over the voting tokens otherwise, and an attacker might manipulate them. Consider if giving permission to the contract owner only is appropriate.
- Similarly, you can use the internal function `_burn` to remove tokens. You can take a look at the OpenZeppelin implementation in `extensions/ERC20Burnable.sol`. You should also control the users authorized to execute this function.

4. Implementation details and tests

Size of contract compiled code. You must take into account that there exists a limit on the maximum size of EVM code of 24Kb approx. Therefore, it might be the case that the compiled code of the `QuadraticVoting` contract exceeds that size. In that case, you should try to reduce the size of the compiled code, for example downgrading the compiler version to any version previous to 0.8.0 to avoid systematic overflow checks (and adding code for just the arithmetic expressions that must be checked), or revising some contract functions to reduce their size. You can figure out how much you must reduce making test compilations commenting out some simple functions.

Testing the system. You must perform tests for all functionalities, first separately, and then integrating several functionalities. You can use a testing script for the latter that tests the largest amount of functionalities.

Check vulnerabilities. Your system must be protected against the attacks and vulnerabilities seen in class.

5. Optional: redesigning `closeVoting`

As a general rule, it is not recommended the implementation of functions containing loops over user data in smart contracts, as they could be attacked with DoS attacks based on producing such a large amount of user data that blocks the functions processing them. In the case of `QuadraticVoting`, the function `closeVoting` can be attacked this way if the number of proposals and participants is large enough: in that case, the system might end up locked if this function cannot be executed in a single transaction, as it would not be possible to close the voting process and return the tokens to their owners nor execute the signaling proposals.

You can optionally redesign this contract functionality to fix that vulnerability. There are two ways of solving this situation:

"Favor pull over push." This technique is based on changing the design of the contract so that the tasks related to token refunding and signaling proposals execution is performed on demand (*pull*) instead of including them in `closeVoting` (*push*). You can get more information at:

`https://eth.wiki/en/howto/smart-contract-safety#favor-pull-over-push-for-external-calls`

If you decide to implement this technique, you should keep the function `closeVoting` to reject new votes and proposals, and provide new functions for enabling participants for requesting the refund of voting tokens and the execution of signaling proposals. You should foresee a specific contract state in which the voting process is not open but that allows the execution of the tasks related to `closeVoting`. To that end, you should design how the contract should behave for allowing all participants recover their tokens during a period of time before reseting the contract for another voting process.

"Resumable function." Another way of fixing this problem is to redesign `closeVoting` as a *resumable* function. You can get additional information about this technique at:

`https://eth.wiki/en/howto/smart-contract-safety#dos-with-block-gas-limit`

The idea of this technique consists in checking in this function code that the transaction has enough gas available to execute each iteration in the loop, and store the iteration number in a state variable and exit otherwise. This way, this function can be invoked again in a different transaction to resume the loop from the saved iteration number when it was interrupted in the previous execution of `closeVoting`. In the particular case of `closeVoting` you must take into account that this check should be done in several loops. Furthermore, you should avoid setting the gas consumption of an iteration with a constant (except in the first iteration): you should compute the consumption in terms of the cost of the previous iterations in the loop.

6. Project assessment

The submission must include the source code and a report describing the system and the design details that you consider. In this report you must add a section analyzing the most relevant attacks that might be performed to your system and describing the mechanisms that you have used to avoid them.

The system developed must satisfy all requirements described in previous sections. The following aspects will also be considered in the assessment of your project:

- The project should use the resources available in the Solidity language, such as libraries, modifiers, inheritance, etc.
- You must implement this project taking into account the most relevant vulnerabilities to avoid possible attacks. Take into account that this system will execute external contract code that might be malicious.
- Code maintainability is also very important. Low-level coding is not recommended unless it is strictly necessary, but on the other hand you should avoid excessive gas consumption.
- You should document these aspects using short comments in the source code.
- The optional redesign of `closeVoting` will be assessed with up to 15 % extra in the project mark. If you add the optional design, you must include a section in the report with a description of the design of this part.

You must submit your project before 23:59, 7 May 2022.