Crowthereum

Crowthereum is a **crowd-funding** platform comprising a smart-contract written in Solidity and a sample front-end that could be used conveniently by the users.

The users can submit projects that are meant to be funded by other users (possibly by the project owner themself). We refer to the users that create projects as **project owners** and to the people that fund them as **funders**.

Each project has a funding deadline and if the project has not been funded till that moment, the project expires and the funders can **reclaim** their money.

The project owner is expected to work on each of the milestones after it is started, the first milestone starts exactly the moment the funding completes. Each **milestone** has a corresponding **cost** and **duration**, after completing a milestone and having it approved by a pre-specified part of the funders, they can claim the money and start working on the next one. If, on the other hand, the project owner fails to demonstrate to the funders that they have indeed completed the milestone and the duration of the milestone is exceeded the project **expires** and the funders can reclaim their money (minus the amount already allotted to the completed milestones).

It is designed to be **safe**, **private** and **lightweight**. The contract will only need to be supplied with a hash of the description of the project and the necessary data defining the deadlines and costs of respective milestones of a project. We avoid storing the project description in the contract, because it can be pricey and compromises the privacy of the users.

In the below section we define the necessary basic functionalities that we set to provide and in the next sections we describe more technically the design decisions that were made and give a rationale behind these.

Basic Functionalities:

The whole life cycle of a project can be divided into three main phases.

First Phase (Project setting)

- The project owner creates a project definition with predefined durations and partial costs of the milestones. This is an array of pairs (*numberOfSeconds*, *milestoneCost*). The project owner also provides a hash of title with description and a special parameter *projectAlpha*.
- Project owner posts on a platform or many platforms his description as a text file that hashes itself to the hash defined in the project definition. For instance, there is a git repo with multiple project descriptions linked to project definitions with a valid hash.
- Project owner calls a method of the contract createProject(), providing all the necessary arguments, which creates the project definition on the blockchain.

Second Phase (Gathering money)

- Investors can look at multiple platforms and search for a project that matches their interest and then they can verify that the description is linked to a project already created on ethereum and crowdfund it, by calling a fundProject(project, amount).
- The money is locked in the contract until the crowdfunding project fails or the corresponding milestone's deadline passes without being approved. The money can be then reclaimed by the funder with a call claimFunds(project).
- If the necessary amount is collected in the contract, the deadline for the first milestone is set. (The first deadline is defined as the time that the goal was met plus the time defined for the first milestone by a project owner)

Third Phase (Execution of the project)

- The project owner fulfills their promise and does some work as specified in the description of the milestone. Once the work is done and observable to all investors the investors should vote for the milestone to be completed using voteMilestone().
- If enough investors have noticed that the work is already done and voted for the milestone, then the project owner can collect the money associated with the part of the project.
- Once the milestone has been voted to be completed, the time is ticking for the next deadline of the next milestone and the game continues until all the milestones are completed and the game ends.
- In case the deadline is hit and there are not enough votes for the milestone, then the whole project fails and all the uncollected money for the unfinished milestones can be reclaimed by the investors.

Design decisions

Privacy of the projects

We wanted to give maximal possible privacy of the project details to the users. That is, if a project owner wishes to share the project with a small number of potential funders, they can divulge the details to said party.

We accomplish this by hashing the description and title of the project (using keccak256 algorithm) and sending only the hash to the blockchain (therefore making it public). The funders can then, when given a description and title, find their project in the contract. All the other data that is being kept in the contract is necessary for the validity of the process.

The project owner can therefore post their project description and title to any third party website and share it with the world. This brings also an additional benefit of not storing additional data and reduces the cost of creating a project on the blockchain.

Non-unanimous milestone completion

We want to have that for a milestone to be completed not all of the funders have to vote for it. Not only this, but it would be convenient to have a possibility of defining the percentage of people that are required to vote for a milestone.

Because of the nature of ethereum, a user can have multiple accounts, so the voting power cannot be fixed for every account. Otherwise, it would be beneficial to the funder to fund the project from multiple accounts. We conclude that the only plausible solution seems to be, to have the voting power proportional to the amount they invested. Every function that would be smaller than f(x) = x would make it profitable to divide the funding into different parts and assume different identities.

Thus, we want to define a percentage of the stake that is required for the milestone to be completed. We define the minimal voting strength to be 1 - α and parametrise the project by $\alpha \in (0, 1)$. Which we will call a *projectAlpha*.

Funder safety

Consider the following scenario, the project owner creates a project with a cost of 100 ETH and *projectAlpha* equal to 1%. They wait for the project balance to reach a near 1 ETH and then fund the project with the remaining 99 ETH. Thus, they obtain the full voting power necessary to accept the milestones and can take the whole money (their 99 ETH + 1 ETH from other users) themselves.

This is an undesirable exploit of the contract. We define a solution to be **funder safe** if (assuming all funders to be honest) it is not profitable for the project owner to fund their project and then abuse the voting power. In order to assure funder safety we have discussed several approaches to solve this issue.

One possibility would be to enforce a minimum amount of funds that can be funded, so that the situation where the project owner can get the necessary voting power does not occur. However, this results in a setting where every user has to vote for the milestone to be completed, equivalently one could think of this as setting projectAlpha to 0.

Another way would be to have a group of external validators, which would be paid for observing the state of the project and voting when a milestone is completed. However, to do this efficiently we would actually want to solve the consensus problem, it is impossible to distinguish between validators that are adversary from others from the point of view of contract.

Finally, we can consider a funding fee and it seems to be the only plausible solution, therefore we incorporated it into the contract. To have that the project owner will not gain anything by the described exploit we want to have that

$\gamma \ = projectFee(\alpha)$

$projectGoal \cdot (1 - \alpha)\, \gamma > \alpha \cdot projectGoal$

That is, the fee is higher than the gain (the amount of money the project owner can claim from the honest funders). Simplifying, we get

$\gamma > \frac{\alpha}{1-\alpha}$

Thus, we want to set the projectFee to *projectAlpha*/(1-*projectAlpha*) and round it up.

In order to avoid using floating operations and float numbers we set the projectAlpha to be in permille’s. That is *projectAlpha* * 1/1000 is the actual alpha value as considered in the above equations.

Excess Value

A funder cannot be sure that the ether they are sending to the contract will not be excessive. We considered several options, one is to accept the additional value and letting the project balance be greater than the project goal. This, however breaks the fee solution, the project owner can send an arbitrarily big amount of ether and obtain the necessary voting power. Taking the excess and not increasing the voting power of the funder would allow the project owner to try the following strategy: the project owner could try to fund the whole project (by using greater gas price) just before the next transaction of a real funder is put on the blockchain. The project owner would then have the whole voting power and could take the excess money from the other funder. Therefore, we decided that we could not allow any excess and it is returned to the funder together with the excessive fee.

Project owner safety

The project owner, in contrast to the funders, has very weak guarantees of safety. In particular, they have to start working on a milestone (and possibly spend money while doing so) without any assurance that they will receive a reward, because after all, the funders may stop cooperating. We could not think of any sensible solution to this problem that would not use any centralised entity or compromise the investor safety. Therefore, the project founder should try to divide the project into many milestones, so that in case of finishing a milestone and getting no reward, they risk only a small amount of resources.

Project structure

Solidity backend

- The whole contract is inside /contract/Projects.sol. All the functions that are exposed are safe and validate their arguments, so neither the project nor the contract can go into an invalid state.
- We also created some unit tests to regularly check if our contract does not have serious logic issues.

Simple frontend

Our frontend is divided into 3 parts:

- Main page where you can find active, expired and finished projects.
- New project page where you can configure a new project and send a request to the contract.
- Project details page where you can check the details of the project and interact with the project. The users can fund the project, vote for milestone completion, claim funds (only as project owner and only after the milestones were completed) or reclaim investment if the project expired

To interact with the chain we use MetaMask, to mock publishing project description we decided to deploy a json file (BulletinBoard.json).

Encountered Issues

Design Issues

- Just like it is in existing non-blockchain solutions (such as Kickstarter or Indiegogo) we thought of ways that the project owner could give some perks to the funders as a reward. We considered releasing tokens that could be exchanged for something else after the project was completed. However, this does not give any better guarantees to the funders than the already implemented solution. They cannot have any certain guarantee that they get rewards after the completion, because the contract cannot ascertain if the funders received anything. Within the current architecture they can claim ownership of some perks by using their signature (their public account number and the size of their contribution is visible to the project owner and to the general public as well). For instance, a project owner could create another contract that would gratify the funders after completion of the project with some virtual perks (e.g. SaletaCoins or CryptoKitties).
- It is inherently impossible to differentiate between an honest and dishonest funder, many issues stem directly from this.

Technical Issues

- We decided to use view functions to easily obtain data to present it on the front-end. We also used them to control our logic there. When we tried to implement isProjectExpired(projectHash) we encountered a problem. The keyword `now` that can be used in solidity points to the moment when the last transaction occurred within the contract. One can solve it by providing current time as an argument or emulating the function outside of the contract.
- We decided to deploy json file to mock a simple bulletin board where we publish any description sent from our website (it could be seen on http://localhost:3004/projects).