Development of a Smart Contract in Solidity - Project Report

CMPE483 - Project 1
Fall Term 2023

Alp Tuna - 2019400288
Roaa Bahaa - 2023680081
Lynn Janzen - 2023690267

1 Overview

Introducing MyGov, a straightforward smart contract written in Solidity. This contract goes beyond basic token functionality, serving as a tool for decentralized decision-making within a community. MyGov lets you do things like giving your vote to someone else, donating in regular money or special tokens, and deciding on which projects to support. It's not complicated – you can propose projects, take part in surveys, and get all the details about what's happening. MyGov is designed to keep things fair and open so that everyone in the community has a say. It's like having a simple and transparent system for everyone to participate in decision-making on the blockchain.

2 Teamwork

In our team of three, we adopted a collaborative approach to tackle various aspects of our blockchain smart contract development project. Linnus was the main responsible person for the tests. She helped us with sophisticated tests and shortened the development process of the smart contract significantly. She wrote more than 20 tests for the contract. Alp was the main responsible person for the project related tasks of the smart contract including but not limited to voting, delegating the votes and reserving the project grant. Roa was the main person implementing survey related logic, including but not limited to submitting surveys, getting survey info and taking surveys. Overall, we split the tasks relatively equally. However, we did not ignore the other members' work, we reviewed and checked the correctness of their code logic.

3 Implementation:

3.1 Contract Overview

- Inheritance: The MyGov contract inherits from the OpenZeppelin ERC20 contract, which is an implementation of the ERC-20 standard for fungible tokens.

- Structs: The contract defines three structs (Survey, Project, Voter) to represent surveys, projects, and voters, each containing various properties related to their respective entities.
- Constants and Variables: The contract declares several constants, such as costs for survey submission and project proposals, along with various state variables to keep track of token supply, member counts, reserved stable coins, and more.
- Mappings and Arrays: The contract uses mappings and arrays to store information about surveys, projects, and voters.
- Events: The contract defines two events (SurveySubmitted and ProjectSubmitted) that can be emitted to notify external entities when a survey or project is submitted.

3.2 Constructor

- The contract has a constructor that initializes the MyGov contract with a specified token supply and a reference to an external USD stable coin contract.

3.3 Token Donation Functions

- donateUSD(uint amount): Allows users to donate USD stable coins to the contract.

- donateMyGovToken(uint amount): Allows users to donate MyGov tokens to the contract.

3.4 Information Retrieval Functions

- getUSDBalanceOfAccount(address _account): Returns the USD stable coin balance of a specified account.

- isMyGovMember(address _address): Checks if an address is a MyGov member based on their MyGov token balance.

- surveyWithIdExists(uint survey_id): Checks if a survey with the given ID exists.

- projectWithIdExists(uint project_id): Checks if a project with the given ID exists.

- tookSurvey(uint survey_id, address _account): Checks if an account has taken a survey with the given ID.

3.5 Survey Submission Functions

- submitSurvey(...): Allows a MyGov member to submit a new survey with specified parameters.

- takeSurvey(...): Allows a MyGov member to take a survey and record their choices.

- getSurveyResults(uint surveyid): Returns the results of a survey, including the number of participants and their choices.

- getSurveyInfo(uint surveyid): Returns information about a survey, including its IPFS hash, deadline, number of choices, and the maximum choices allowed.

3.6 Project Proposal and Voting Functions

- submitProjectProposal(...): Allows a MyGov member to submit a new project proposal with specified parameters.

- reserveProjectGrant(uint projectid): Allows the owner of a project to reserve a project grant if certain conditions are met.

- withdrawProjectPayment(uint projectid): Allows the owner of a project to withdraw the payment for the current project if certain conditions are met.

- voteForProjectProposal(uint projectid, bool choice): Allows a MyGov member to vote for or against a project proposal.

- voteForProjectPayment(uint projectid, bool choice): Allows a MyGov member to vote for or against a specific payment of a funded project.

3.7 Information Retrieval Functions (Projects)

- Various functions to retrieve information about projects, including their owners, information, the total number of project proposals, the total number of funded projects, and more.

3.8 Faucet Function

- faucet(): A faucet function to distribute MyGov tokens to users, allowing them to receive 1 MyGov token from the contract.

4 Tests

4.1 Testing Approach and Structure

For testing our smart contract we used Truffle in conjunction with Ganache. Truffle is an advanced framework for smart contract development. It provides a suite of tools for testing and deploying contracts.Ganache is a part of the Truffle Suite and functions as a personal blockchain for Ethereum development. It simulates a blockchain environment.
The testing process for the MyGov contract involved writing automated test scripts using JavaScript and deploying the contract to the Ganache simulated blockchain environment.

Before running the tests, the contract instances of *MyGov* and *USD_Stable_Coin* are deployed using *USD.new(initialUSDSupply)* and *MyGov.new(tokenSupply, usdInstance.address)*. This setup ensures that the contract environment is initialized and

ready for interaction. Several initial configurations were included to ensure sufficient myGov token and USD funds for the test accounts in order to be able to perform the actions defined in the contract.
The tests are executed sequentially, with each test performing specific interactions with the smart contract followed by assertions to validate the expected outcomes. The use of assert and expect from the chai library provides readable assertions.
Tests for error conditions, such as attempting to take a survey more than once, are included. These tests ensure that the contract correctly handles and reports errors, an essential aspect of robust smart contract behavior. The test cases for project, survey and faucet are independent of each other and implemented as individual scripts.

4.2 Test case for the Faucet Function:

1. **Single Use of Faucet per User**: To verify that a user can only receive tokens from the faucet once.

4.3 Test cases for the functionality concerning survey activities

1. **Existence of Survey**: Checks whether a survey with a given ID exists within the contract.
2. **Survey Participation**: Verifies whether an account has successfully taken a survey.
3. **Restriction on Repeated Participation**: Ensures that an account cannot take a survey more than once.
4. **Survey Results Accuracy**: Confirms that the survey results are correctly tallied and reported.
5. **Survey Information Validity**: Checks the correctness of the survey's information, such as the hash, deadline, number of choices, and the maximum number of choices allowed.
6. **Ownership Verification**: Validates that the survey is owned by the correct account.
7. **Total Surveys Count**: Confirms the number of surveys created in the contract.

4.4 Test cases for the functionality concerning project activities

1. **Existence of a Project (Positive Test)**: To verify if a project with a specified ID exists within the contract.
2. **Existence of a Project (Negative Test):** To ensure that the contract correctly identifies non-existing projects.
3. **Total Number of Projects**: To confirm the correct count of project proposals in the contract.
4. **Total Number of Funded Projects:** To validate the count of funded projects in the contract.
5. **Correct Project Ownership:** To check if the ownership of a project is correctly assigned.
6. **Duplicate Voting on Project Proposal:** To test the contract's handling of duplicate votes on a project proposal.
7. **Duplicate Voting on Project Payment:** To ensure the contract prevents double voting on project payments.

8. **Project Information Accuracy:** To verify the correctness of the project information stored in the contract.
9. **Project Grant Reservation by Owner:** To test that only the project owner can reserve the project grant.
10. **USD Received by Project:** To check if the USD received by a project matches the expected payment.

4.4 Problems

The above test cases were successfully tested using the 10 default accounts provided by the truffle suite. However, when we attempted to use the ganache network and create 100 - 300 accounts, we ran into issues which we were not able to solve. We created an ethereum network and linked to the truffle framework by specifying host and address in truffle-config.js. The network had been successfully started and also logged transactions. However, for any transaction other than creating a contract, we subsequently ran into an “invalid opcode” error which we were not able to detect.

5 Measurement of Gas Costs

5.1 Influences on Gas Consumption

Gas refers to the unit that measures the computational effort required to execute operations like transactions and smart contract function calls. The total gas cost of an operation is determined by gas amount (quantity of gas required to perform an operation) and gas price (amount of Ether the sender is willing to pay per unit of gas). Several factors influence the amount of gas used in transactions and contract executions. More complex contracts with extensive computational logic consume more gas. Operations that involve storing or modifying data on the blockchain typically use more gas due to the higher cost of storage.

5.2 Our Test Setup

In our testing framework using Truffle, we have structured JavaScript scripts to measure the gas costs of various contract functions. We focused on measuring gas costs for **non-view-only** functions in the smart contract. View-only functions in Ethereum do not alter the blockchain state and are generally executed locally on the node, which means they do not consume any gas when called externally. Unlike view-only functions, non-view functions perform state changes on the blockchain, and as a result, they consume gas. These functions include transactions that alter the state of the contract, such as updating variables, transferring tokens, or executing logic that affects the contract's data.

Initially, the contracts are deployed using Truffle, which involves gas costs that are measured and logged. Contract functions are then executed within test cases. For each function call, the gas used is captured. After a function call, the transaction receipt is analyzed, and the gasUsed property is extracted. For two functions we compare gas costs for single execution vs. multiple executions within a loop.
In the following we list our measured gas amounts and provide an analysis of some selected functions.

5.3 Results

delegateVoteTo	100,451
voteForProjectProposal	119,240
voteForProjectPayment	107,140
reserveProjectGrant	80,797
withdrawProjectPayment	114,806
usdTransfer	26,854
transfer	29,677
submitSurvey	241,455

***submitProjectProposal**:* 432,973
The *require* statements in this function involve reading from the contract's state, which consumes small amounts of gas. *donateMyGovToken* and *donateUSD* are called to deduct the necessary MyGov tokens and USD Stable Coins for the project proposal. These operations involve modifying the contract's state (such as updating balances), which is more gas-intensive than read operations. A new project is created and added to the projects array *(projects.push()).* This step involves writing to Ethereum's storage, which is one of the most gas-consuming operations in smart contracts. Setting various properties of the project, such as *ipfsHash, voteDeadline*, *paymentAmounts*, *paymentSchedule*, etc., further involves multiple storage writes. Initializing new arrays (like *yesVoteCountsOfPayments*) and setting default values for the project's properties consume additional gas due to storage operations. Emitting the *ProjectSubmitted* event consumes gas, but typically less than storage operations. Event logs are cheaper in terms of gas but still add to the total cost. The function concludes by returning the new project's ID. This doesn't significantly impact gas usage.

faucet Function:
Gas Used in single execution: 96,466
Average Gas (n=9) in loop execution: 79,366
The primary gas-consuming operations are the token transfer and the update to the *usedFaucet* mapping. Both of these involve writing to the contract's state. The initial call to

faucet for a user might consume slightly more gas due to the change of the *usedFaucet[msg.sender]* from its default value (false) to true. However, this difference is likely minimal, and the primary cost driver remains the token transfer operation.

donateMyGovToken*:* 34,502
This function deals mainly with token transfers and a single conditional state update, which are less gas-consuming compared to the array and storage operations in *submitProjectProposal*.

takeSurvey:
Gas Used in one execution: 152,168
Average Gas over loop (n=6) : 100,868
The function starts with several *require* statements which are primarily read operations.. However, they still contribute to the total gas cost due to the need to access and evaluate contract state. The function iterates over the choices array to validate each choice. This involves read operations and conditional checks, which consume gas for each iteration, depending on the length of the choices array. It again iterates over the choices array, incrementing corresponding values in the *surveys[surveyid].results* array. This is a write operation and is more gas-intensive as it involves modifying the contract's state. It also updates the *surveys[surveyid].choices[msg.sender]* mapping to record the choices made by the user, another write operation. Finally, it increments *surveys[surveyid].numTaken*, indicating another participant has taken the survey. This is a state change and thus consumes gas.
The initial execution may involve setting up state (like the first time a user's choices are recorded) which could be more gas-intensive compared to subsequent executions where the state is only updated.

6 Possible improvements

6.1 Comprehensive Unit Testing

One key area for improvement in our smart contract implementation is the expansion of our unit test coverage. While we have executed a set of initial unit tests, enhancing the test suite to cover a wider range of scenarios and edge cases would provide greater confidence in the robustness and reliability of our smart contract. This could involve testing various input combinations, exploring boundary conditions, and incorporating negative test cases to ensure the contract behaves as expected under diverse circumstances.

6.2 Increased Testing with Multiple Accounts

To enhance the resilience and security of our smart contract, it is advisable to conduct testing with a higher number of accounts. This involves simulating interactions between different users and testing how the contract responds to varying account states and permissions. By incorporating a broader range of user scenarios, we can identify potential vulnerabilities or unexpected behaviors that may not be apparent in a limited testing environment. This step would be important for ensuring the contract's functionality in real-world usage.

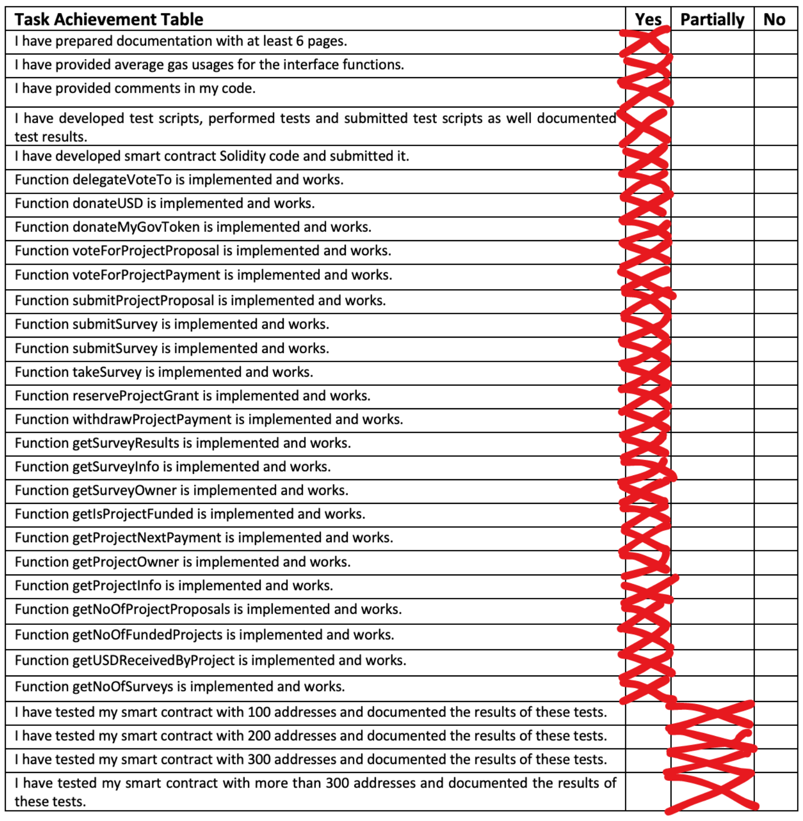
6.3 Exploration of Testnets in Addition to Localnets

While local testing environments are valuable for initial development, integrating testnets into our testing strategy can show issues that may not be visible in a controlled local setting. Testnets replicate the decentralized nature of public blockchains, allowing us to assess the performance and reliability of our smart contract under conditions similar to the mainnet. Deploying and testing on popular testnets like Ropsten, Rinkeby, or Kovan provides an opportunity to validate the contract's functionality in a more dynamic and realistic network environment.

6.4 Gas Consumption Optimization

Efficient use of gas is crucial for deploying and interacting with smart contracts on a blockchain. Assessing and optimizing the gas consumption of our contract can lead to cost savings and improved scalability. This involves reviewing the contract's code and identifying areas where gas costs can be reduced, such as optimizing loops, minimizing storage usage, or leveraging gas-efficient design patterns.

7 Task Achievement Table



Task Achievement Table	Yes	Partially	No
I have prepared documentation with at least 6 pages.	X		
I have provided average gas usages for the interface functions.	X		
I have provided comments in my code.	X		
I have developed test scripts, performed tests and submitted test scripts as well documented test results.	X		
I have developed smart contract Solidity code and submitted it.	X		
Function delegateVoteTo is implemented and works.	X		
Function donateUSD is implemented and works.	X		
Function donateMyGovToken is implemented and works.	X		
Function voteForProjectProposal is implemented and works.	X		
Function voteForProjectPayment is implemented and works.	X		
Function submitProjectProposal is implemented and works.	X		
Function submitSurvey is implemented and works.	X		
Function submitSurvey is implemented and works.	X		
Function takeSurvey is implemented and works.	X		
Function reserveProjectGrant is implemented and works.	X		
Function withdrawProjectPayment is implemented and works.	X		
Function getSurveyResults is implemented and works.	X		
Function getSurveyInfo is implemented and works.	X		
Function getSurveyOwner is implemented and works.	X		
Function getIsProjectFunded is implemented and works.	X		
Function getProjectNextPayment is implemented and works.	X		
Function getProjectOwner is implemented and works.	X		
Function getProjectInfo is implemented and works.	X		
Function getNoOfProjectProposals is implemented and works.	X		
Function getNoOfFundedProjects is implemented and works.	X		
Function getUSDReceivedByProject is implemented and works.	X		
Function getNoOfSurveys is implemented and works.	X		
I have tested my smart contract with 100 addresses and documented the results of these tests.		X	
I have tested my smart contract with 200 addresses and documented the results of these tests.		X	
I have tested my smart contract with 300 addresses and documented the results of these tests.		X	
I have tested my smart contract with more than 300 addresses and documented the results of these tests.		X	