CS6504 A3

Group Project

Dr Manish Singh

Alex Keegan & Andrew Graff

Contents

Introduction and Motivation

This project is to create a voting contract software program (Voting dApp/Decentralized application) that enables multiple end users to participate in a decentralized voting system. “Blockchain is the underlying technology that drives dApps. The core elements of the blockchain ecosystem ensure the security, performance, and execution of dApps functions”, (Popchev & Radeva, 2024).

Blockchain is important in this project as it encompasses the many values of what is required for a voting system to be immutable and secure.

“Think of DApps like neighbourhood block party. Instead of being run by one person or organization, everyone in the neighbourhood works together to make the party a success”, (Pandey, 2024).

This voting application will operate as an entirely peer to peer network. This allows all those who participate in the voting smart contract system to make a vote for a ballot created of their choice introducing reinforced transparency and legitimacy amongst those who interact with the program. This is interesting because it opens a ballot space where it eliminates any uncertainty on any third-party tampering and unauthenticated voters. Each user would have a copy of the latest ledger on the blockchain. “This public ledge represents all the data in the blockchain. Al the data in the public ledger is secure by cryptographic hashing and validated by a consensus algorithm (proof of work)” (McCubbin, 2024).

The usefulness of this application is something that could be very well forged soon, and maybe entice non-voters to regain trust in the system and cast their vote. This can be used in scenarios of decision making, elections etc. where the importance of one’s vote can make all the difference.

Overview

The structure of the Voting_Dapp will have three contracts that will make up the Voting_Dapp blockchain application. Aufiero et al.'s (2024) explains, "each dApp is composed of multiple smart contracts, each containing a number of functions that can be called to trigger a specific event, e.g., a token transfer".

These are Ballot, Ballot Manager and Ownable that operate as the pillars of our smart contract interaction to provide a smooth working ballot system. This will be created using Solidity ^0.8.0 (MIT).

Functions will be the major component of this application displaying some complexity involved in performing the many features of this application. Each of these components have a purpose within the overall functionality of the voting application. These were top 5 votes, contract owner controls for security, user options when creating a ballot, securing these against user address and importantly the voting system count. "The blockchain makes it possible to have a fully transparent election while keeping all the voters anonymous", (Tanwar et. al 2023).

A management system set up for the specified ballot duration and voting functions for users are some of the demonstrations of why this is a major component of the application.

Modifiers play a vital role in enforcing access control in this blockchain. Restricting functions to only authorized users. Such controls are checks/modifying and approvals. Constructors will initialize the ballot creation and ballot counter.

These components will all be intertwined to apply security measures, transparency and ultimately a working blockchain application. This will be done by calling the import of supporting contracts to link the functions, modifiers and constructors.

Detailed Description/Design

As part of the designing, notes were gathered to outline what the functionality of the blockchain would do. These were user operations, extent of contract owner capabilities and owners of this voting application. Brainstorming techniques were applied during the initial proposal for the design of application. “Developing the smart contact is the most central step in developing a decentralized application and what sets it apart from the development of traditional applications is the triggering of operations” (Chen, 2024).

InitalVotingDAppIdea.txt

```
- Any user should be able to create a ballot and become a Ballot Owner
- Any user can vote on any ballot
- Users should only be able to vote once per address
- Contract Owners can approve ballots as they come in / owners can delete ballots ???
- Contracts Owners should be able to view list of ballots waiting to be approved
- Users should be able to view a leaderboard of the top 5 most voted ballots
- Users can stake ETH to increase the weight of their vote up to a maximum of 2x (super vote)
- Ballot Owners should be able to enable / disable the staking currency (super voting) to increase vote weight
- Ballot Owners should be able to specify the amount of time for the ballot to run
- Ballots should have A MINIMUM of two options
- Ballots should run for a MINIMUM duration of 5 minutes. Time starts as soon as the ballot has been approved by the Contract Owner
- Users should be able to view the results of any ballot after it has finished (The winning option in the ballot), how many votes each option in the ballot got

Contract Owner - Owner of the contract, can accept ballots before voters can vote
Ballot Owner - Creator of ballot, can also vote for their ballot, can disable super voting, set amount of time for ballot to run
Ballot Voter/User - Has a list of all the ballots, can vote on all ballots, can sake ETH to super vote if enabled
```

The major components are functions, modifiers, constructor, structs and mappings across the two contracts Ballot and Ballot Manager.

The functions have specific roles with comments to understand its purpose, adding to the modularity of the design in this blockchain application. The layout and order of the blockchain design enables any re-visiting for adjustments to align or make changes to core functions.

Modifiers are implemented for specific functions to be called by the appropriate user. Leading to an added layer of security and transparency in the creation/modification of the ballots.

We have input screenshots of the respective contracts under each major component of our blockchain as part of the implementation and detailed description/design.

Implementation (cont.)

Structs and Mappings

One of the functionalities of our voting system is a contract created called Ballot. This is a voting system that entails the various operations and limitations. Here we can see information stored about the options chosen for each ballot and the number of votes.

```
contract Ballot {
    struct ballotOption {
        // stucture to store information about each ballot option
        string optionName;
        uint optionVotesAmount;
    }
    mapping(address => bool) public addressVotedOnBallot;
```

The user of the ballot option will have their address linked to the ballot, keeping track of addresses whether if a vote has been lodged or not.

State Variables

Ballot.sol - The ballot options stores records of all the options the user picks in the ballot. The address of the ballot owner and name of the ballot whether if the ballot has been approved or active. The total votes and unique id of the ballot are some of the stored information in the contract.

```
ballotOption[] public ballotOptions; // Stores stucts of all options user can pick in ballot

address public ballotOwner;
string public ballotName;
bool public ballotHasBeenApproved;
bool public ballotIsActive;
uint public ballotTotalVotes;
uint public ballotId;
```

BallotManager.sol – This contains a list of active ballots contracts. A variable to assign a unique ID to each ballot and the mapping linking ballot ID to its respective contracts.

```
contract BallotManager is Ownable {
    Ballot[] public ballots; // list of active ballots
    uint public ballotIdCounter; // variable to assign unique ID to each ballot
    mapping(uint => Ballot) public ballotIdMapping; // mapping of units (ids) to ballots

    // initalize the ballotIdCounter variable
    constructor() {    3440878 gas 3407800 gas
        ballotIdCounter = 0;
    }
```

Modifiers

Ballot.sol - There are modifiers put in place as restrictions of upholding the ballot activity and approval/authenticity of the contract or ballot owner. These are specified call functions from specified users (ballot owner). The execution of functions is applicable given whether if ballot is active and lastly the approval.

```
modifier onlyBallotOwner() {
    // modifier to restrict use of methods to only the Ballot Owner
    require(msg.sender == ballotOwner, "Error: This address is not the Ballot Owner!");
    _;
}

modifier onlyBallotIsActive() {
    // modifier to ensure that the ballot is still active
    require(ballotIsActive, "Error: This ballot has been ended by either the Contract or Ballot Owner!");
    _;
}

modifier onlyApproved() {
    // modifier that requires a ballot has been approved by the Contract Owner
    require(ballotHasBeenApproved, "Error: This ballot has not been approved by the Contract Owner!");
    _;
}
```

BallotManager.sol – This validation modifier will check to ensure the ballot ID is correct and checks the existence of the ballot (address) and range of ballot ID's.

```
modifier ballotIdIsValid(uint _ballotId) {
    // if BallotId is greater than the counter or 0, it can't be valid
    require(_ballotId > 0 && _ballotId <= ballotIdCounter, "Error: That is an invalid Ballot ID");

    // converts the ballot into it's address representation. If the address is 0, it means an address with the given ID was not found
    require(address(ballotIdMapping[_ballotId]) != address(0), "Error: The Ballot with that Ballot ID does not exist");
    _;
}
```

Constructor

Ballot.sol - This is where we have a constructor that will initialize the state variables in our contract. Address and name of the ballot owner, array of strings available in the ballot along with the ballot id. The ballot options array is populated with passed in options.

There is an infinite gas 980400 gas to ensure our ballot options does not exceed high gas usage when initialising what could be large arrays. These have implemented accordingly throughout the smart contracts.

```
constructor(address _ballotOwner, string memory _ballotName, string[] memory _ballotOptions, uint _ballotId) {    infinite gas 980400 gas
    ballotOwner = _ballotOwner;
    ballotName = _ballotName;
    ballotId = _ballotId;
    ballotTotalVotes = 0;
    ballotHasBeenApproved = false;
    ballotIsActive = false;


    for (uint i=0;i<_ballotOptions.length;i++) {
        // populate ballotOptions list with passed in options
        ballotOptions.push(ballotOption({optionName: _ballotOptions[i], optionVotesAmount: 0}));
    }
}
```

BallotManager.sol – When the Ballot Manager contract is deployed, the constructor will initialize the ballotIdCounter variable. Infinite gas set at 3440878 gas.

```
contract BallotManager is Ownable {
    Ballot[] public ballots; // list of active ballots
    uint public ballotIdCounter; // variable to assign unique ID to each ballot
    mapping(uint => Ballot) public ballotIdMapping; // mapping of units (ids) to ballots

    // initalize the ballotIdCounter variable
    constructor() {    3440878 gas 3407800 gas
        ballotIdCounter = 0;
    }
```

Functions

Ballot.sol - We have 8 functions in our Ballot contract, each with a specific role in the voting mechanism. There are error messages for the user should the requirements are not met.

The ballot owner who can start or end the ballot process (once approved) and checks to see whether the ballot is active, and the user has not already voted. This will then allow the vote to be accounted for.

```
// function for Ballot Owners to start their ballot once it has been approved
function startBallotVoting() external onlyBallotOwner {    28889 gas
    require(ballotHasBeenApproved == true, "Error: The Ballot has not been approved, therefore you cannot start voting!");
    // approve the ballot to start voting
    ballotIsActive = true;
}

function endBallotVoting() external onlyBallotOwner {    31027 gas
    require(ballotHasBeenApproved, "Error: This ballot was never approved, and therefore cannot be ended!");
    require(ballotIsActive, "Error: This ballot is not currently active, and therefore cannot be ended!");
    ballotIsActive = false;
}

function voteForBallot(uint indexOfSelectedOption) external {    infinite gas
    require(ballotIsActive, "Error: This ballot has ended, or was never started");
    // check the index of selected option is within range
    require(indexOfSelectedOption < ballotOptions.length, "Error: That index is out of range, and is not a valid option!");

    // check the user has not already voted
    require(!addressVotedOnBallot[msg.sender], "Error: This address has already voted on this ballot!");

    ballotOptions[indexOfSelectedOption].optionVotesAmount += 1;
    ballotTotalVotes++;

    addressVotedOnBallot[msg.sender] = true;
}
```

A procedure in place where results of the ballot will only be retrieved once the ballot ends (winning option and winning option votes).

Approval functions that determine whether if the ballot has been approved and returning the status. And lastly, functions to return total votes.

```
function getResultsOfBallot() external view returns(string memory winningOption, uint winningOptionVotes) {   infinite gas
    require (!ballotIsActive, "Error! Results of ballot cannot be retrieved until ballot ends!");
    uint winningOptionIndex;
    uint winningVoteAmount;

    for(uint i=0;i<ballotOptions.length;i++) {
        if (ballotOptions[i].optionVotesAmount > winningVoteAmount) {
            winningVoteAmount = ballotOptions[i].optionVotesAmount;
            winningOptionIndex = i;
        }
    }
    return (ballotOptions[winningOptionIndex].optionName, winningVoteAmount);
}

function setApproval(bool _newApprovalStatus) external {   24761 gas
    ballotHasBeenApproved = _newApprovalStatus;
}

function getApproval() public view returns (bool) {   2519 gas
    return ballotHasBeenApproved;
}

function incrementTotalVotes() external {   infinite gas
    ballotTotalVotes += 1;
}

function getTotalVotes() public view returns (uint) {   2455 gas
    return ballotTotalVotes;
}
```

BallotManager.sol – We have 11 functions in our BallotManager contract, these functions marry closely with the designing and management of the ballot system.

We have a function that creates a ballot with specific user requirements and a system where a count occurs when each ballot is created. This information (new ballot) is stored in a ballot array along with the mapping of the ballot to the current ballot id counter.

There are a few functions only the contract owner can call and that is deleting and approving the ballot.

```
function createBallot(string memory _ballotName, string[] memory _ballotOptions) public {   infinite gas
    // ballot must have 2 options
    require(_ballotOptions.length >= 2, "Error: Ballots must have at least 2 options to vote on!");
    require(_ballotOptions.length <= 5, "Error: Ballots must have a max of 5 options");
    // increase the ballot counter every time a new ballot is created for unique ID
    ballotIdCounter++;

    // create a new ballot object
    Ballot newBallot = new Ballot(msg.sender, _ballotName, _ballotOptions, ballotIdCounter);

    // add the information about the new ballot to the structure array
    ballots.push(newBallot);
    // map the new ballot to the current id counter in the mapping
    ballotIdMapping[ballotIdCounter] = newBallot;
}

function deleteBallot(uint _ballotId) public onlyOwner ballotIdIsValid(_ballotId) {   infinite gas
    // remove the ballot from the array of ballots
    delete ballots[_ballotId - 1];
    delete ballotIdMapping[_ballotId];
}

function approveBallot(uint _ballotId) public onlyOwner ballotIdIsValid(_ballotId) {   infinite gas
    require(!ballots[_ballotId - 1].getApproval(), "Error: This Ballot has already been approved!");
    ballots[_ballotId - 1].setApproval(true);
}

function updateTotalVoteCount(uint _ballotId) external ballotIdIsValid(_ballotId) {   infinite gas
    Ballot ballot = ballotIdMapping[_ballotId];
    ballot.incrementTotalVotes();
}
```

Functions around the voting process and count. These are updating/incrementing the total vote count, start and ending voting for a specified ballot, users to vote for specified options in a ballot and retrieving specific information about a ballot contract.

```
function ballotOwnerStartVoting(uint _ballotId) public ballotIdIsValid(_ballotId) {    infinite gas
    Ballot ballotToStart = ballotIdMapping[_ballotId];
    ballotToStart.startBallotVoting();
}

function ballotOwnerEndVoting(uint _ballotId) public ballotIdIsValid(_ballotId) {    infinite gas
    Ballot ballotToEnd = ballotIdMapping[_ballotId];
    ballotToEnd.endBallotVoting();
}

function vote(uint _ballotId, uint selectedOption) public ballotIdIsValid(_ballotId) {    infinite gas
    Ballot ballotToVoteOn = ballotIdMapping[_ballotId];
    ballotToVoteOn.voteForBallot(selectedOption);
}

function getSpesificBallotInformation(uint _ballotId) public ballotIdIsValid(_ballotId) view returns (Ballot)
    // map the ballotId
    Ballot ballot = ballotIdMapping[_ballotId];

    // return the ballot data from the structure
    return ballot;
}
```

Functions topFiveBallots – this function returns the top 5 ballots. The design of this function mirrors the ballots array to sort in descending order according to total votes, it will then determine how many ballots to return (more or less than 5 ballots).

An array will be created for the top 5 ballots.

```
function topFiveBallots() public view returns(Ballot[] memory) {    infinite gas
    // create a mirror copy of the ballots array for use in this function
    Ballot[] memory sortBallots = new Ballot[](ballots.length);
    for (uint i = 0; i < ballots.length; i++) {
        sortBallots[i] = ballots[i];
    }

    // bubble sort the ballots in decending order according to total votes
    Ballot swap;
    for(uint i = 0; i < sortBallots.length; i++) {
        for (uint j = 0; j < sortBallots.length - 1; j++) {
            if (sortBallots[j].getTotalVotes() < sortBallots[j + 1].getTotalVotes()) {
                swap = sortBallots[j];
                sortBallots[j] = sortBallots[j + 1];
                sortBallots[j + 1] = swap;
            }
        }
    }

    // if there are less than 5 ballots, the return length will be set to this
    // if there are more than 5 ballots then the return length will be set to 5
    uint lengthToReturn = (ballots.length <= 5) ? ballots.length : 5;

    // create an array of size lengthToReturn to store the top ballots
    Ballot[] memory topBallots = new Ballot[](lengthToReturn);

    // place the top values (up to 5) vfrom the newly sorted array into placeholder array
    for (uint i = 0; i < lengthToReturn; i++) {
        topBallots[i] = sortBallots[i];
    }

    return topBallots;
}
```

The final two functions are getting approved and unapproved ballots. It is specified only the contract owner can call the unapproved ballots.

Related work

A similar project would be Election voting DAPP using Smart contracts (Block Chain) - Part II https://www.linkedin.com/pulse/election-voting-dapp-using-smart-contracts-block-part-vemulakonda-1f (Vemulakonda, 2021).

The difference between the above smart contract design and our design is the features. We have created highly complex features in our solidity script to enable the many different functions of the voting application.

These are to see the top 5 votes and some security features in the modifiers in the creation or approval process of the ballot. The Election dApp is created for simple basic uses where there is flexibility in the Voting dApp that allows for many different ballots style voting.

Differences	
Voting_dApp	**Election dApp**
Creation of ballots (Flexibility)	Voting election based
Only contract owner can approve ballot	Only smart contract can add valid candidates
Gas fees	Gas fees
Features: Top 5 votes Ballot approvals Voting count ETH fees Ballot owner can start and end Contract owner management Retrieve ballot information	Features: Two candidate voting option Web app

Whilst the similarity in both projects is on the principles of voting systems, there are vast differences in the features of the decentralised application. The features in our Voting_dApp application offer diversity and security.

A related article on a decentralised voting-based system was read in the international journal for research. Raskar et al. (2023) mentions “democracies rely heavily on voting, but younger generations in particular are becoming least active votes”. This is an interesting read of democracy and the necessity for a better advanced system to battle against flaws in the current voting system which relates to the idea of this dApp.

The common goal across all these projects is to provide an immutable, secure and transparent voting system. It is discovered this is very likely to be achieved by a blockchain application.

Discussion and conclusion

What went well, what didn't:

We approached this project with an open mind and fair idea of the blockchain and solitude script writing during our lab sessions. Ideas were thought out and executed to our strengths. The initial planning was clear, understanding of the group assignment project combined with preferences and offering availability to assist each other when needed. Some barriers were getting test ETH ethers to deploy and test the contracts. One of the challenges was with setting the duration of the ballot, because there is no way to schedule tasks with blockchain technology we had to omit the setting of the duration and changed it over to a Boolean system. Where the contract is either active or unactive. Which is defined at the discretion of the Ballot Owner or Contract Owner. Additionally due to time contracts we didn't implement the ETH staking part of the contract nor the frontend.

Future work:

To endeavour the journey into the blockchain world and seeing the wonders of this field. More exploration into the creation of solitude script writing/blockchain and navigating between front and back ends of the application. Seeing other ways blockchain could be implemented. Future work would include the creation of the Frontend for the DApp, as well as the implementation of the ETH staking features which were discussed in the design phase.

Your takeaway:

Proactivity and collaboration were key to our progress and success in this project. We learned the importance of thorough planning and prompt communication were driving factors into the completion of this group assignment. Additionally, our group learnt how to design and develop blockchain technology. By doing this we increased our understand of what a DApp is, what a smart contract is, and their significance in the wider blockchain they are implemented in.

Group Work Allocation

Both members of the team assisted in all areas of the project in some capacity. However, as a group we decided to split the project into two parts, the report and the implementation of the DApp to complete the project efficiently and play to the strengths of each team member. Alex worked on the DApp primarily helping where required to Andrew who worked primary on the report. Andrew also assisted Alex with portions of the DApp when it was required. Both members of the team worked on the design phase together and were in constant communication with one another to ensure that the report and the DApp were consistent with one another.

References

Aufiero et al. (2024).	Aufiero, S., Ibba, G., Bartolucci, S., Destefanis, G., Neykova, R., & Ortu, M. (2024). Dapps ecosystems: Mapping the network structure of smart contract interactions. *arXiv preprint arXiv:2401.01991*.
Chen (2024).	Chen, Z. March 2024. Applied and Computational Engineering 48(1):46-52 DOI:10.54254/2755-2721/48/20241132
McCubbin, (2024).	McCubbin, G. February 14, 2024. The Ultimate Ethereum Dapp Tutorial. The Ultimate Ethereum Dapp Tutorial (How to Build a Full Stack Decentralized Application Step-By-Step) | Dapp University
Pandy, (2024).	Pandy, A. October 2, 2024. *What is DApp (Decentralized Apps)?*. What Is a DApp? Decentralized Apps Explained - Intellipaat
Popchev & Radeva, (2024).	Popchev, I. & Radeva, I. Decentralized Application (dApp) Development and Implementation. Cybernetics and Information Technologies, 2024, Sciendo, vol. 24 no. 2, pp. 122-141. https://doi.org/10.2478/cait-2024-0019
Raskar et al. (2023).	Raskar, Atharva & Pansare, Mayur & Chumbalkar, Vishal & Kamble, Tushar & Desai, Mrs. (2023). Decentralized Voting System using Blockchain. International Journal for Research in Applied Science and Engineering Technology. 11. 2119-2122. 10.22214/ijraset.2023.52071
Tanwar et al (2023).	Tanwar, S., Gupta, N., Kumar, P. *et al.* 06 May 2023. Implementation of blockchain-based e-voting system. *Multimed Tools Appl* 83, 1449–1480 (2024). https://doi.org/10.1007/s11042-023-15401-1
Vemulakonda (2021).	Vemulakonda, N. June 15 2021. Election voting DAPP using Smart contracts (Block Chain) – Part II. Election voting DAPP using Smart contracts (Block Chain)- Part II | LinkedIn