CSS486 BLOCKCHAIN DEVELOPMENT

VOTING SYSTEM

Final Report

Submitted to

School of Information, Computer and Communication

Technology

Sirindhorn International Institute of Technology

Thammasat University

Group Members:

Mr. Chadchayasan Jirapittayamet 6222790246
Mr. Pawis Petchkaewkul 6222770537
Mr. Vasin Vorarit 622772038

Advisor: Dr.Watthanasak Jeamwatthanachai

Project Overview

The Blockchain-Based Voting System for Creating and Voting on Campaigns is a decentralized application (dApp) that utilizes blockchain technology to provide a secure and transparent platform for users to create and vote on campaigns. It ensures the integrity of the voting process by leveraging the immutability and decentralization offered by blockchain networks. The system comprises a blockchain network (Goerli testnet), campaign creation module, and user voting module. Users can create campaigns with relevant details, and other users can vote on these campaigns. The project utilizes ThirdWeb, React, and Solidity as the technical stack for development. The goal is to empower individuals to express their opinions, contribute to social causes, and participate in a democratic process through a user-friendly and decentralized platform.

Motivation

This report explores the motivations behind the development of a voting system connected with Metamask, a widely used cryptocurrency wallet and browser extension. By integrating Metamask into the voting process, numerous benefits can be realized, enhancing the transparency, security, and accessibility of democratic voting. This report outlines the key motivations for creating a voting system connected with Metamask and highlights the advantages it brings to stakeholders.

Trust and Transparency:
Integrating Metamask into the voting system fosters trust and transparency in the democratic process. The use of blockchain technology ensures that voting records are immutable, tamper-proof, and publicly verifiable. Every vote is recorded on the blockchain, creating a transparent and auditable trail of the voting process. This transparency promotes confidence among voters, as they can independently verify the integrity of the election results.

Enhanced Security:
Metamask's advanced security features provide a robust foundation for the voting system. By leveraging Metamask's encryption and private key management, the voting system can secure users' identities and voting credentials. This safeguards against identity theft, voter fraud, and unauthorized access to the voting system, ensuring the integrity of the electoral process. The integration of Metamask adds an extra layer of protection, enhancing the overall security of the voting system.

Accessibility and Convenience:
The integration of Metamask into the voting system improves accessibility and convenience for voters. Metamask's user-friendly interface and compatibility with various devices and platforms allow individuals to participate in the voting process seamlessly. Voters can easily access the voting system through their Metamask wallets, eliminating the need for complex setups or additional software installations. This increased accessibility empowers a wider range of individuals to engage in the democratic process, including those who may face physical or geographical limitations.

Decentralization and Independence:
By leveraging the decentralized nature of blockchain technology, a voting system connected with Metamask reduces reliance on centralized authorities and intermediaries. The elimination of a central authority reduces the risk of manipulation or bias in the voting process. Moreover, it ensures that no single entity has control over the voting system, preserving the independence and integrity of the democratic process.

Data Integrity and Auditability:
Metamask's integration with blockchain technology ensures the integrity and auditability of voting data. Each vote is permanently recorded on the blockchain, creating an immutable and tamper-proof ledger. This feature allows for post-election audits and recounts, providing a reliable mechanism to verify the accuracy of the voting results. The ability to audit the voting process strengthens trust in the electoral system and allows for a fair and transparent resolution of any disputes.

The creation of a voting system connected with Metamask brings numerous motivations and advantages to the democratic process. The integration of Metamask enhances trust, transparency, and security, while also improving accessibility and convenience for voters. By leveraging the benefits of Metamask and blockchain technology, the voting system can create a more robust, inclusive, and accountable electoral system. The adoption of a voting system connected with Metamask paves the way for a more transparent and democratic society.

Features

1. User Authentication:
    - The system utilizes secure authentication mechanisms, such as public-private key pairs, to authenticate users and ensure the integrity of their identities.

2. Campaign Creation:
    - Users can create campaigns by providing relevant details such as the campaign title, description, goals, and duration.
    - Each campaign is assigned a unique identifier and stored on the blockchain for transparency and accountability.

3. Campaign Browsing:
    - Users can browse through existing campaigns to gain insights into the different causes and initiatives.
    - Campaign details, including the creator, goals, and progress, are displayed to help users make informed voting decisions.

4. Vote Casting:
    - Users can cast their votes on the campaigns they support.
    - The system verifies each vote to ensure its authenticity and uniqueness.
    - Votes are cryptographically signed and timestamped, and the blockchain records them securely.

5. Vote Counting and Results:
    - The system processes and counts the votes for each campaign.
    - The vote count is transparently displayed, allowing users to monitor the progress and outcome of the campaigns.

Tech Stack

Our project utilizes the following tech stack:

1. ThirdWeb: ThirdWeb is a decentralized web framework that enables the development of blockchain-based applications. It provides the necessary infrastructure and tools to build decentralized applications (dApps) that leverage blockchain technology. By utilizing ThirdWeb, our project benefits from the decentralized nature of the web, ensuring transparency, security, and resilience.

2. React: React is a popular JavaScript library for building user interfaces. It allows for the creation of interactive and dynamic web applications with a component-based architecture. By utilizing React, our project ensures a smooth and intuitive user experience. It enables the development of a user-friendly interface that is responsive and provides real-time updates, enhancing user engagement and satisfaction.

3. Solidity: Solidity is a programming language specifically designed for writing smart contracts on the Ethereum blockchain. Smart contracts are self-executing contracts with the terms of the agreement directly written into code. By utilizing Solidity, our project can define the behavior and logic of the voting system's smart contracts. It enables the implementation of secure and reliable business logic, ensuring the integrity and trustworthiness of the voting process. In our project, Solidity plays a crucial role in implementing the logic and behavior of the smart contracts that govern the voting system. Here's how we use Solidity in our project. Campaign Contract that we create a Solidity contract specifically for managing campaigns. This contract includes functions to create a campaign, store campaign details, and handle the voting process. It defines the rules and conditions for creating and voting on campaigns, ensuring the fairness and integrity of the system.

4. Goerli Testnet: By utilizing the Goerli testnet, we can take advantage of its stability and reliability. This ensures that our voting system operates smoothly during the testing and development phases. We can have confidence in the consistency and dependability of the network, allowing us to focus on building and refining our application without worrying about frequent disruptions or inconsistencies. The decentralized network of validators in Goerli enhances the resilience of our voting system. It reduces the risk of censorship or attacks, ensuring the security and integrity of the voting process. This is essential for our project as we aim to create a trustworthy and tamper-proof platform

where users can express their opinions and participate in campaigns without concerns about manipulation or interference.Goerli's faster block time is beneficial for our voting system, especially during testing scenarios that involve time-sensitive operations. With quicker transaction confirmations, we can simulate real-time voting scenarios more effectively. This allows us to assess the responsiveness and performance of the system, ensuring a smooth and efficient user experience.

By combining ThirdWeb, React, and Solidity, our project leverages the benefits of decentralized web development, interactive user interfaces, and secure smart contract programming. This tech stack allows us to build a robust and user-friendly blockchain-based voting system that promotes transparency, security, and user engagement.

Smart Contract

```
// // SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.9;

contract Voting {
    struct Campaign{
        address owner;
        string title;
        string description;
        uint256 target;
        uint256 deadline;
        uint256 amountCollected;
        string image;
        address[] voters;
    }
```

This is a **`struct`** called **`Campaign`**, which has eight properties: **`owner`**, **`title`**, **`description`**, **`target`**, **`deadline`**, **`amountCollected`**, **`image`** and **`voters`**. It will be used to store Campaign Data.

```
    mapping(uint256 => Campaign) public campaigns;

    uint256 public numberOfCampaigns = 0;
```

This first line defines a **`mapping`** called **`campaigns`**, which maps an **`uint256`** to a **`Campaign`** object. This mapping will be used to store the campaign data. The second line is to set the numberOfCampaigns to be zero.

```
function createCampaign(address _owner, string memory _title, string memory _description,
    uint256 _target, uint256 _deadline, string memory _image) public returns(uint256) {
    Campaign storage campaign = campaigns[numberOfCampaigns];
```

This function is called **`createCampaign`**, which takes six parameters: **`_owner`** of type **`address`**, **`_title`** of type **`string`**, **`_description`** of type **`string`**, **`_target`** of type **`unit256`**, **`_deadline`** of type **`uint256`** and **`_image`** of type **`string`**. It's marked as **`public`** which means that anyone can call this function and the return type is **`uint256`**. Then the next line is to create the variable called **`campaign`**.

```
    //if everything okay?
    require(campaign.deadline < block.timestamp, "The deadline should be a date in the future");
```

The first lines check whether the **`_deadline`** is in the past or not. If it is invalid, the function will throw an error with the corresponding message.

```
campaign.owner = _owner;
campaign.title = _title;
campaign.description = _description;
campaign.target = _target;
campaign.deadline = _deadline;
campaign.amountCollected = 0;
campaign.image = _image;

numberOfCampaigns++;
return numberOfCampaigns - 1; //return the index of the most newly created campaign
```

These lines set the campaign properties to be equal to the input data. The increase the numberOfCampaigns by one. Then return the index of the most newly created campaign.

```
function isDuplicate(uint256 _id) public view returns (bool) {
    for(uint i = 0; i < campaigns[_id].voters.length; i++){
        if(campaigns[_id].voters[i] == msg.sender) return true;
    }
    return false;
}
```

This function called `**isDuplicate**` is used to check the duplicate voter so the voter can not vote in the same campaign. Which takes one parameter: `**_id**` type of `**uint256**`. It's marked as `**public**` which means that anyone can call this function. And the return type is `**Boolean**`. The algorithm selects the campaign by _id then loops through the campaign voter then if the msg.sender is already in the voter array then it will return `**true**` which means it is duplicate but if it returns `**false**` it means that it is not duplicate.

```
function vote(uint256 _id) public {
    require(!(isDuplicate(_id)),"You have already voted");
    Campaign storage campaign = campaigns[_id];
    campaign.voters.push(msg.sender);
    campaign.amountCollected++;
}
```

This function is called `**vote**` which is used to vote in the campaign. Which takes one parameter: `**_id**` type of `**uint256**`. It's marked as `**public**` which means that anyone can call this function. This function is required to call the `**isDuplicate**` function to check that the voter has already voted in this campaign or not. Then select the campaigns by `**_id**` then assign the select campaign to a variable campaign. Then push the `**msg.sender**` type of `**address**` into the voter array. Then increment the amountCollected by one.

```
function getVoters(uint256 _id) view public returns(address[] memory) {
    return campaigns[_id].voters;
}
```

This function is called `**getVoters**` which is used to see the voter of the selected campaign. Which takes one parameter: `**_id**` type of `**uint256**` t's marked as `**public**` which means that

anyone can call this function. And the return type is the array type of `**address**`. The function will select the campaign by `**_id**` then return the array of voters.

```
    function getCampaigns() public view returns(Campaign[] memory){
        Campaign[] memory allCampaign = new Campaign[](numberOfCampaigns); //create empty array with numberOfCampaigns index

        for(uint i = 0; i < numberOfCampaigns; i++){
            Campaign storage item = campaigns[i];
            allCampaign[i] = item;
        }
        return allCampaign;
    }
}
```

This function is called `**getCampaigns**`**.** It's marked as `**public**` which means that anyone can call this function. The return type is an array of campaigns. The function will create an empty array with `**numberOfCampaigns**` index. Then loop through the campaigns then append the campaigns into the variable called `**allCampaign**` then return the allCampaign.

Backend Part

client/context/index.js

```
export const StateContextProvider = ({ children }) => {
  const { contract } = useContract('0xD5877e664d63bf69E5a1ccA623bD5b5FcabA3c6f');
  const { mutateAsync: createCampaign } = useContractWrite(contract, 'createCampaign');

  const address = useAddress();
  const connect = useMetamask();
```

This part is used to connect to the smart contract by using the `**useContract**` built in function. And connect to the Metamask by using the `**useMetamask**` built in function.

client/context/index.js

```
const publishCampaign = async (form) => {
  try {
    const data = await createCampaign([
      address, // owner
      form.title, // title
      form.description, // description
      form.target,
      new Date(form.deadline).getTime(), // deadline,
      form.image
    ])

    console.log("contract call success", data)
  } catch (error) {
    console.log("contract call failure", error)
  }
}
```

This part is used to call the `**createCampaign**` function from the smart contract and use the data from the form as an input for the function.

client/context/index.js

```
 const isDuplicate = async (pId) =>{
    const data = await contract.call('isDuplicate', pId);

    console.log(data)
    return data;
 }
```

This part is used for calling the `**isDuplicate**` function from the smart contract.

client/context/index.js

```
const vote = async (pId) => {
  try {
    const data = await contract.call('vote', pId);
    return data;
  } catch (error) {
    alert("You Already Voted")
  }
}
```

This part is used for calling the `**vote**` function from the smart contract.

client/context/index.js

```
const getCampaigns = async () => {
  const campaigns = await contract.call('getCampaigns');

  const parsedCampaings = campaigns.map((campaign, i) => ({
    owner: campaign.owner,
    title: campaign.title,
    description: campaign.description,
    target: ethers.utils.formatEther(campaign.target.toString()),
    deadline: campaign.deadline.toNumber(),
    // amountCollected: ethers.utils.formatEther(campaign.amountCollected.toString()),
    amountCollected: campaign.amountCollected.toNumber(),
    image: campaign.image,
    pId: i
  }));

  return parsedCampaings;
}
```

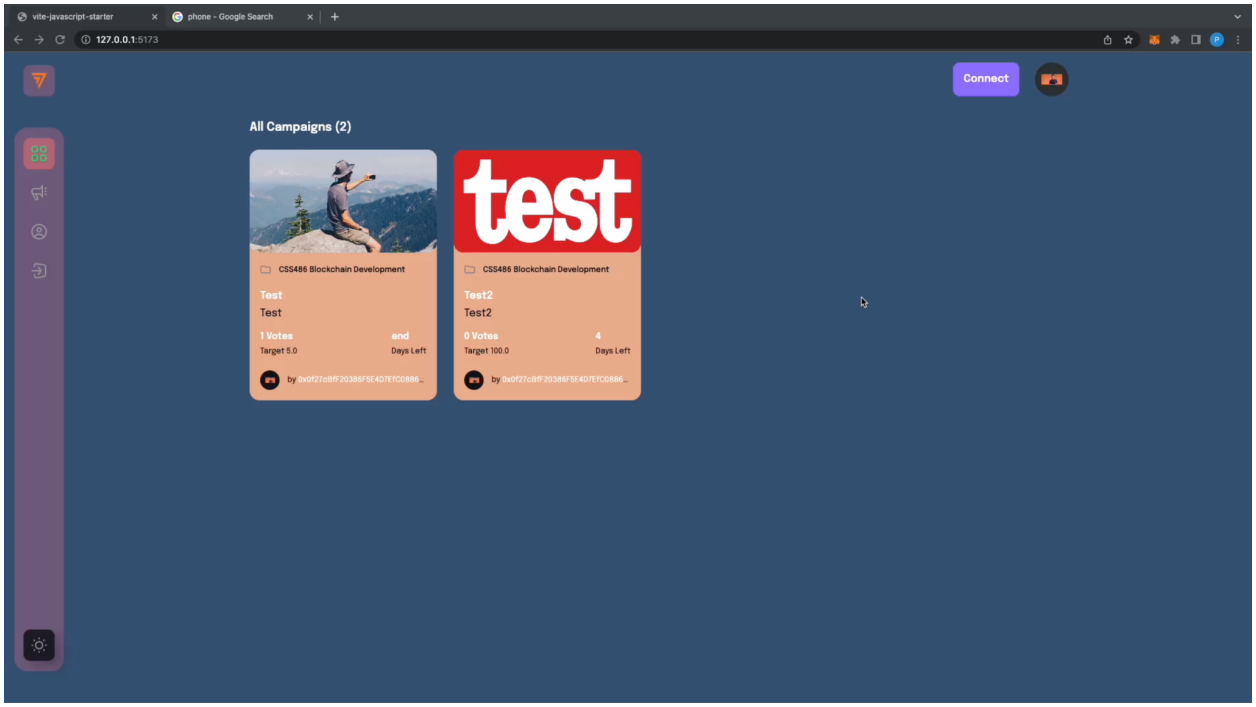
This part is used for calling the `**getCampaign**` function from the smart contract.

Approach of running application

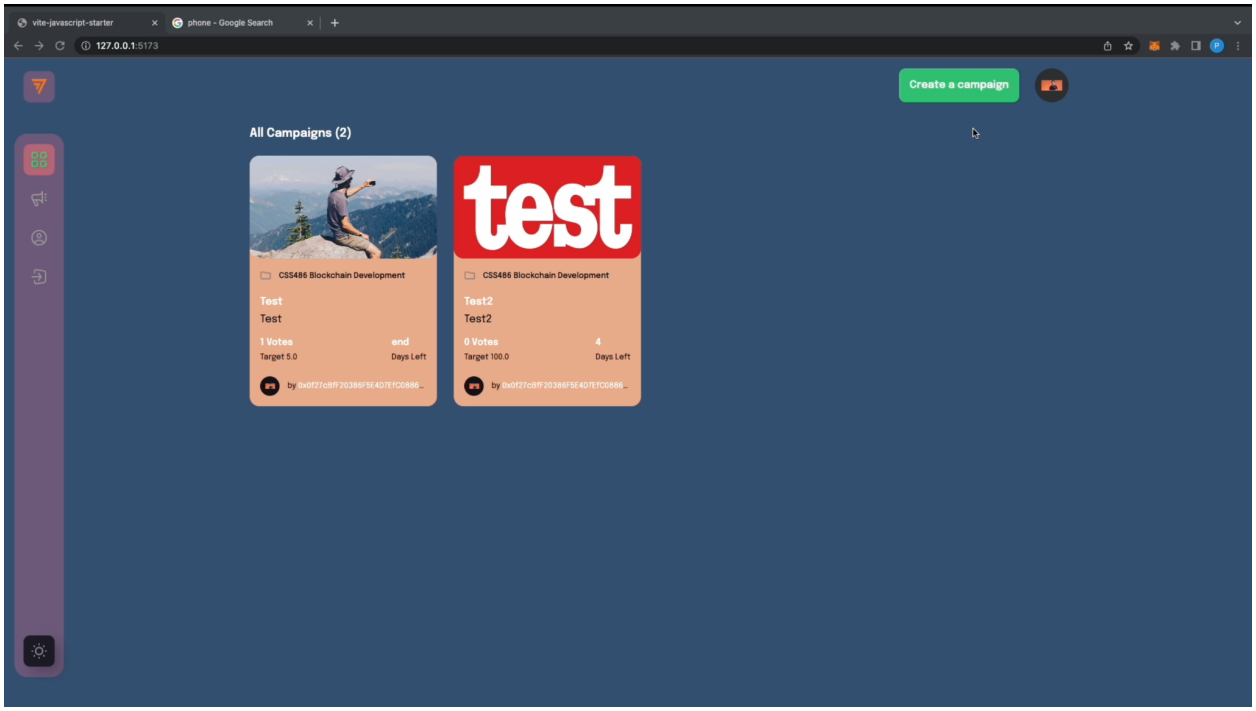
1. cd to the client folder
2. npm install
3. npm run dev

Sample Scenarios

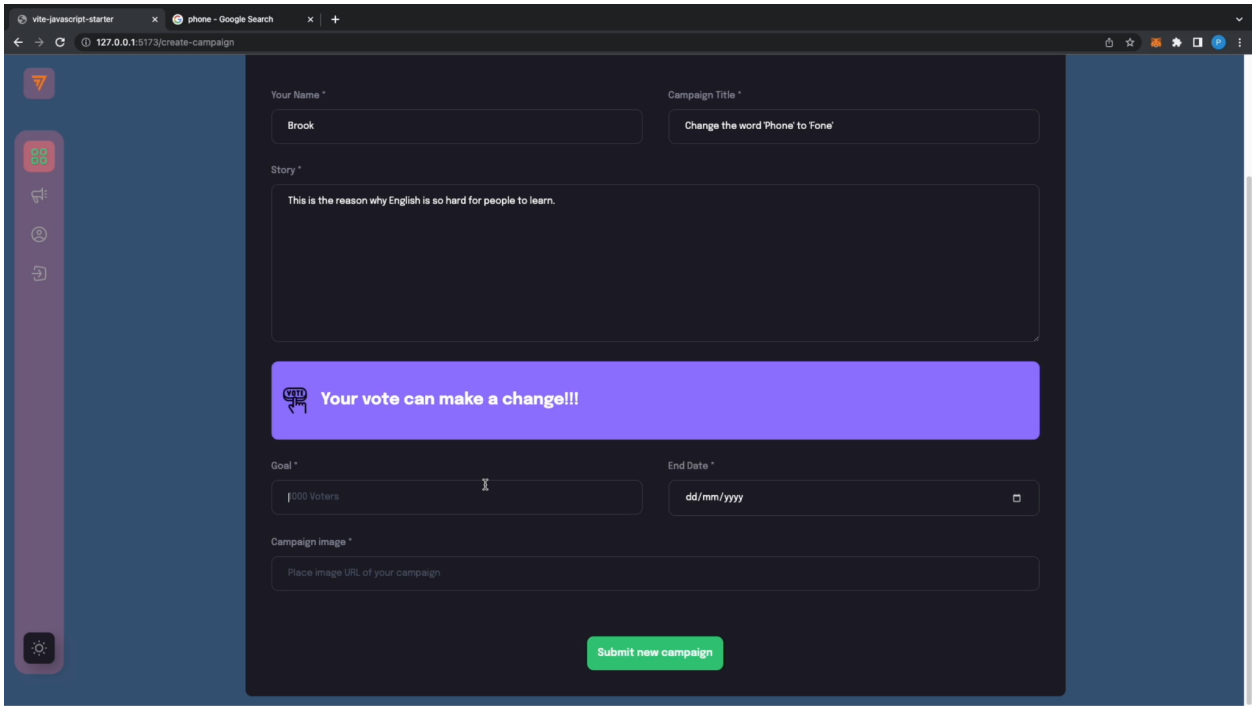
1.Click the connect button to connect to the metamask account.



2.When connected to the metamask click create campaign to create a campaign.



3. Fill in the form then click Submit new campaign. Then pay the gas fee.



4. Click the campaign to see the campaign details. Then Click the vote for this Campaign button to vote the campaign. Then pay the gas fee.