Decentralized Voting System - Report

Final Exam - Blockchain Technologies 1

Team members:

Aigerim Askarova,

Sagi Alpyssov,

Sabira Kurbankhozha

Introduction

This project implements a decentralized voting system on an Ethereum test network. The main goal was to create a secure and transparent platform for elections, where participants can vote and receive on-chain rewards. The system combines smart contract development, ERC-20 token incentives, and frontend interaction via MetaMask, while operating entirely on test networks without using real cryptocurrency.

Users can create elections with a title, a list of candidates, and a deadline. Each valid vote triggers the issuance of 10 VOTE tokens. The system ensures that only valid votes are accepted, deadlines are enforced, and duplicate voting is prevented. In our implementation, the frontend provides immediate visual feedback, for example disabling the vote button when the election has ended.

1. System Architecture

The application consists of three main layers: Smart Contracts, Frontend, and MetaMask integration.

1.1 Smart Contracts

Two contracts were developed: VoteToken.sol and VotingSystem.sol.

VoteToken.sol is a standard ERC-20 token contract. Its mint function can only be called by the voting system, which ensures controlled issuance of reward tokens. After deployment, ownership is transferred to the voting system

```
await voteToken.transferOwnership(await votingSystem.getAddress());
```

This decision guarantees that only valid votes trigger token rewards, preventing misuse.

VotingSystem.sol manages the election lifecycle. It stores titles, candidate lists, deadlines, vote counts, and records of voters who have already participated. Each vote triggers VoteToken.mint

```
voteToken.mint(msg.sender, 10 * 10 ** 18);
```

Events such as ElectionCreated, VoteCast, and ElectionFinalized allow the frontend to respond in real time. During development, listening to these events was crucial to update the UI immediately after each blockchain state change.

1.2 Frontend Application

The frontend, built with JavaScript and ethers.js v6, allows users to:

- Connect their MetaMask wallet;
- Create elections with a title, candidates, and duration;
- Vote for candidates;
- View current vote counts and VoteToken balances.

The interface updates automatically after each confirmed transaction. For example, when a vote is cast

```
const tx = await votingContract.vote(electionId, candidateId);
await tx.wait();
```

After the transaction is mined, the frontend updates the balance

```
balanceEl.innerText = "VoteToken Balance: " + ethers.formatUnits(balance,
18);
```

Additionally, the vote button is dynamically disabled if the election has ended or has been finalized

```
document.getElementById("vote").disabled = finalized || now >=
Number(deadline);
```

This small detail was added after observing that users could try voting after the deadline in early tests.

1.3 MetaMask Integration

MetaMask serves as the bridge between users and the Ethereum network. The app detects it via window.ethereum and requests account access

```
await window.ethereum.request({ method: "eth_requestAccounts" });
```

Before submitting transactions, the frontend verifies the network

```
if (chainId !== LOCAL_CHAIN_ID) {
  statusEl.innerText = `Please switch MetaMask to Localhost 8545 (chainId
31337). Current chainId: ${chainId}`;
  return;
}
```

This prevents accidental transactions on the wrong network. All transactions, including election creation and voting, are signed securely through MetaMask.

2. Smart Contract Logic

The voting logic ensures correctness and security:

Election Creation

createElection stores the election title, candidates, and deadline

```
Election storage e = elections[electionCount];
e.title = _title;
e.candidates = _candidates;
e.deadline = block.timestamp + _duration;
```

A minimum of two candidates is required

```
require(_candidates.length >= 2, "At least 2 candidates required");
```

Voting

Each vote undergoes validation

```
require(block.timestamp < e.deadline, "Voting ended");
require(!e.hasVoted[msg.sender], "Already voted");
require(_candidateId < e.candidates.length, "Invalid candidate");
```

On success, the voter receives 10 VOTE tokens. This direct token reward reinforces participation and was a design choice to motivate active engagement.

Finalization

After the deadline, elections can be finalized to prevent further votes

```
e.finalized = true;
emit ElectionFinalized(_electionId);
```

This mechanism ensures that once an election closes, the results cannot be modified.

3. Frontend-to-Blockchain Interaction

The frontend serves as the interface for user interaction:

- Wallet Connection: Users must approve account access via MetaMask.
- Transaction Handling: All blockchain calls are asynchronous. The UI waits for confirmation before updating the interface.

- State Synchronization: Vote counts and token balances are updated automatically after transactions.
- Event Listening: Frontend listens to events like VoteCast to immediately reflect the latest vote tally.
- Validation Feedback: Input fields and buttons are dynamically updated. Users are informed via messages or disabled elements if they attempt invalid actions, such as voting after the deadline.

This integration was iteratively improved while testing, adding feedback mechanisms that prevent confusion and enhance the user experience.

4. Deployment and Test ETH

Deployment uses the Hardhat local network and follows a specific sequence. First, VoteToken.sol is deployed, followed by VotingSystem.sol, which receives the token contract address as a constructor parameter. After deployment, ownership of the token contract is transferred to the voting system to allow automatic minting of reward tokens

```
await voteToken.transferOwnership(await votingSystem.getAddress());
```

After deployment, the deployed contract addresses must be inserted into the frontend configuration (app.js) to ensure proper interaction

```
const VOTING_ADDRESS = "0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512";
const VOTETOKEN_ADDRESS = "0x5FbDB2315678afecb367f032d93F642f64180aa3";
```

The frontend can then be launched using a local server (for example, npx serve frontend). Users connect MetaMask to the local Hardhat network (Chain ID 31337) and import a pre-funded account provided by Hardhat for testing. Each account includes 10,000 test ETH for seamless interaction without using real cryptocurrency.

For public testnets, such as Sepolia, test ETH can be obtained through faucets. All network interactions remain confined to test environments, ensuring compliance with academic and technical requirements.

5. Testing

Automated tests cover all core functionality:

- Election creation with proper parameters;
- Voting logic and token issuance;
- Prevention of double voting;
- Enforcement of deadlines;
- Election finalization.

Time manipulation with Hardhat (evm_increaseTime and evm_mine) confirmed the correct behavior for deadline-based voting. All tests passed successfully, demonstrating the system's reliability.

Example test snippet:

```
const balance = await voteToken.balanceOf(voter1.address);
expect(balance).to.equal(ethers.parseUnits("10", 18));
```

Additionally, testing revealed minor UI improvements, like automatically disabling buttons and showing status messages for better user guidance.

6. Conclusion

The Decentralized Voting System provides a secure and transparent platform for elections, combining smart contracts, ERC-20 incentives, and a responsive frontend. Users can create elections, vote securely, and receive participation tokens, all on test networks.

Throughout development, careful attention was given to validation, state synchronization, and user feedback. The system is fully tested, reliable, and ready for further extension. The project demonstrates a practical implementation of blockchain principles, combining backend contract logic with frontend usability.