PROJECT REPORT

Team Members

1. Patel, Liyansu Ghanshyambhai : patel2464@saskpolytech.ca
2. Bhojak, Krishna Vipulkumar : bhojak3091@saskpolytech.ca
3. Patel, Vivekkumar Kaushikkumar : patel6562@saskpolytech.ca

Problem Definition

Crowdfunding platforms have revolutionized the way projects are funded by allowing creators to raise funds directly from supporters. However, traditional crowdfunding platforms often suffer from issues such as high fees, lack of transparency, and centralized control, leading to mistrust and inefficiencies. This project aims to address these challenges by developing a decentralized crowdfunding platform on the Ethereum blockchain, ensuring transparency, reducing costs, and eliminating the need for intermediaries.

Design and Architecture

The platform is built using Solidity for smart contract development and Next.js with React for the frontend interface. The architecture comprises:

- **Smart Contracts**: Handle campaign creation, contributions, request approvals, and fund disbursements on the Ethereum blockchain.
- **Frontend Interface**: Provides users with the ability to create campaigns, contribute, and manage funding requests. Communicates with the blockchain via Web3.js.
- **Routing**: Utilizes `next-routes` for dynamic routing, enabling seamless navigation between different campaign pages.
- **Testing Framework**: Implements testing using Mocha and Chai to ensure the reliability of smart contracts and frontend components.

Implementation

Smart Contracts (`Campaign.sol`)

- **Campaign Management**: Allows managers to create funding requests and finalize fund disbursements.
- **Contribution Handling**: Ensures contributors meet the minimum contribution criteria.

Key Functions:

- **Constructor**: Initializes the campaign with a minimum contribution and manager address.

```
constructor(uint minimum, address creator) {
    manager = creator;
    minimumContribution = minimum;
}
```

- **Contribute**: Enables users to contribute funds and become approvers.

```
function contribute() public payable {
    require(msg.value > minimumContribution, "Minimum contribution not
met.");
    approvers[msg.sender] = true;
    approversCount++;
}
```

- **Create Request**: Allows the manager to create a spending request.

```
function createRequest(string memory description, uint value, address
payable recipient)
    public authorization {
        Request storage newReq = requests.push();
        newReq.description = description;
        newReq.value = value;
        newReq.recipient = recipient;
        newReq.complete = false;
        newReq.approvalCount = 0;
}
```

Frontend Interface (`CampaignShow.js`)

- **Campaign Overview**: Displays details like minimum contribution, balance, requests, and manager address.
- **Contribution Form**: Allows users to contribute to the campaign.

Rendering Campaign Details:

```
renderCards() {
    const {
        balance,
        manager,
        minimumContribution,
        requestCount,
        approversCount,
    } = this.props;

    const items = [
        {
            header: manager,
            meta: 'Address of Manager',
            description: 'The manager created this campaign and can create 
requests to withdraw money.',
        },
        {
            header: minimumContribution,
            meta: 'Minimum Contribution (wei)',
            description: 'You must contribute at least this much wei to 
become an approver.',
        },
        // Additional card items...
    ];

    return <Card.Group items={items} />;
}
```

Routing (`routes.js`)

Defines dynamic routes for the application:

```
const routes = require('next-routes')();

routes
    .add('/campaigns/new', '/campaigns/new')
    .add('/campaigns/:address', '/campaigns/show')
    .add('/campaigns/:address/requests', '/campaigns/requests/index')
    .add('/campaigns/:address/requests/new', '/campaigns/requests/new');

module.exports = routes;
```

Testing and Results

Smart Contract Testing (`Campaign.test.js`)

Utilizes Mocha and Chai for testing smart contract functionality.

Test Cases Include:

- **Deployment**: Checks if the contract is deployed with the correct manager.

  ```
  it('deploys a campaign', () => {
      assert.ok(campaign.options.address);
  });
  ```

- **Contributions**: Verifies that contributions are recorded if they meet the minimum requirement.

  ```
  it('allows contributions and marks approvers', async () => {
      await campaign.methods.contribute().send({
          value: '200',
          from: accounts[1],
      });
      const isContributor = await
  campaign.methods.approvers(accounts[1]).call();
      assert(isContributor);
  });
  ```

- **Minimum Contribution Enforcement**: Ensures that contributions below the minimum are rejected.

  ```
  it('requires a minimum contribution', async () => {
      try {
          await campaign.methods.contribute().send({
              value: '5',
              from: accounts[1],
          });
          assert(false);
      } catch (err) {
          assert(err);
      }
  });
  ```

Frontend Testing

Uses React Testing Library and Jest for component testing.

Test Scenarios Include:

- **Campaign Details Rendering**: Checks that campaign information displays correctly.
- **Contribution Form Functionality**: Ensures that contribution submissions work and feedback is provided.

Results:

All test cases pass successfully, indicating that smart contracts and frontend components function as intended. The platform effectively handles edge cases, such as rejecting insufficient contributions and prohibiting unauthorized actions.

Conclusion

The decentralized crowdfunding platform effectively addresses the limitations of traditional systems by leveraging the Ethereum blockchain for enhanced transparency and security. Smart contracts manage funds trustlessly, reducing intermediary costs. Comprehensive testing confirms the platform's reliability, providing a robust foundation for deployment. Future improvements may include advanced analytics, refined user interfaces, and multi-blockchain support to enhance accessibility and scalability.