Decentral Store

Decentralized Storage Solution Project Proposal

Team

Alexandru Toader

Andrei-Cornel Besliu

Ioan Slavu

Purpose of the Project

The project aims to design and develop a decentralized storage solution leveraging **blockchain technology** to provide secure, reliable, and distributed data storage. This solution targets decentralized applications (dApps) and services, ensuring:

- **Data Integrity**
- **Enhanced Security**
- **Elimination of Single Points of Failure** associated with centralized storage systems.

Needs the Project Tries to Satisfy

- **Data Security**: Protecting data from unauthorized access, tampering, and breaches.
- **Reliability**: Guaranteeing data availability and durability without reliance on a single provider.
- **Scalability**: Efficiently handling growing data storage demands.
- **Cost-Effectiveness**: Reducing expenses compared to centralized storage providers.
- **Decentralization**: Strengthening resilience and enabling user control over data.

Components of the Project

1. Decentralized Storage System

- Develop the core storage infrastructure using decentralized protocols (e.g., **IPFS**).
- Ensure data redundancy and distribution across multiple nodes.
- Implement encryption and access control mechanisms.

2. Smart Contracts

- Design and deploy smart contracts on a suitable blockchain platform (e.g., **MultiversX**).
- Manage storage agreements, payments, and incentives for storage providers.
- Facilitate data retrieval and access permissions.

3. Web Application (WebApp)

- Create a **user-friendly interface** for interacting with the decentralized storage system.
- Enable seamless **upload, management, and access** of data.
- Integrate with smart contracts to handle transactions and permissions.

Timeline for Development and Task Assignation

Phase 1: Planning and Design (1 Week)

- **All Team Members**:
  - Define project requirements and specifications.
  - Create high-level system design and architecture.

Phase 2: Development (2 Weeks)

Andrei-Cornel BEȘLIU – Decentralized Storage

- Research and create a solution for the storage infrastructure (e.g., **IPFS**).

Alexandru TOADER – Smart Contracts

- Design smart contracts to act as a middleman for file storage transactions.
- Develop a solution for managing user roles.

Ioan SLAVU – WebApp

- Design and develop the **web application interface**.
- Integrate the web app with **smart contracts** and storage system APIs.

Phase 3 & 4: Deployment, Testing, and Documentation (1 Week)

- **All Team Members**:
  - Collaborate to ensure seamless integration of all services.
  - Conduct individual testing for each solution to ensure functionality.
  - Write comprehensive project documentation.

Github Repo

https://github.com/AlexTove/multiversx-descentralized-storage

Storage Methodology

To store data effectively, we needed a way to provide a reliable storage environment. Typically, solutions like **Network File Systems (NFS)** are used, which rely on a central server to handle all file storage activities. However, considering the decentralized nature of our project, this architecture contradicts our goals. A centralized resource introduces a single point of control, enabling one entity to potentially dominate the entire storage network.

Why IPFS?

To align with our decentralized objectives, we chose to use an implementation of **IPFS (InterPlanetary File System)**. IPFS operates on a **peer-to-peer network** for file storage. The network consists of multiple nodes implementing the IPFS protocol. Each file stored in the network is:

- **Replicated** across these nodes.
- Identified by a **unique hash ID**, called a **CID (Content Identifier)**.

Implementation Details

For our project, we opted for **Kubo nodes** (available at Kubo GitHub Repository), a well-supported IPFS implementation with an active community.

To create our test platform, we built our own **Kubo IPFS cluster** using **Docker Compose**, with the following configuration:

- **3 Kubo Nodes**: Simulate a minimal peer-to-peer network.
- **Control Node**: Exposes port 5001 to execute **Kubo RPC commands** for file addition and queries.

This setup allows our application to store blockchain data in a decentralized, redundant, and secure manner, utilizing **file encryption** to ensure data safety.

Smart Contract Overview

Key Features

1. **File Upload**

   - Allows users to upload files and store their metadata on the blockchain.
2. **File Management**
   - Users can view, remove, and tag their uploaded files.
3. **Tag Management**
   - Enables users to add and remove tags for better file categorization and searchability.
4. **User Isolation**
   - Ensures each user's files are managed independently, maintaining privacy and scalability.

Data Structures

FileMetadata Struct

The `FileMetadata` struct is used to store metadata for each file. It includes the following fields:

- **`file_hash`**: A hash representing the file content.
- **`file_size`**: The size of the file in bytes.
- **`file_name`**: The name of the file.
- **`file_type`**: The MIME type of the file.
- **`file_tags`**: A list of tags associated with the file.
- **`file_cid`**: The IPFS Content Identifier (CID) of the file.
- **`timestamp`**: The time the file was uploaded.
- **`uploader`**: The blockchain address of the uploader.

Contract Storage

Files Storage

- **Storage Type**: `MapMapper<ManagedAddress, ManagedVec<FileMetadata>>`
- **Purpose**: Maps each user's blockchain address to their list of uploaded files.

Functions

1. File Upload

- **Parameters**:
  - `file_hash`: Hash of the file content.
  - `file_size`: File size in bytes.
  - `file_name`: Name of the file.
  - `file_type`: MIME type of the file.
  - `file_cid`: IPFS CID of the file.
- **Purpose**: Uploads file metadata to the contract.

- **Validation**:
  - `file_hash`, `file_name`, `file_type`, and `file_cid` must not be empty.
  - `file_size` must be greater than 0.

---

2. View User Files

- **Parameters**:
  - `user_address`: The blockchain address of the user.
- **Returns**: A list of files uploaded by the specified user.
- **Purpose**: Fetches all files uploaded by a specific user.

---

3. Remove File

- **Parameters**:
  - `file_cid`: The IPFS CID of the file to remove.
- **Purpose**: Deletes a file from the user's uploaded files.

---

4. Retrieve Uploaded Files

- **Returns**: A list of files uploaded by the caller.
- **Purpose**: Retrieves all files associated with the current user.

---

5. Add Tag to File

- **Parameters**:
  - `file_cid`: The IPFS CID of the file.
  - `tag`: The tag to add to the file.
- **Purpose**: Adds a new tag to the file metadata.
- **Validation**:
  - Ensures the tag doesn't already exist for the file.

---

6. Remove Tag from File

- **Parameters**:
  - `file_cid`: The IPFS CID of the file.
  - `tag`: The tag to remove from the file.
- **Purpose**: Removes an existing tag from the file metadata.
- **Validation**:
  - Ensures the tag exists for the file before attempting to remove it.

Frontend: Next.js Project

The frontend of the application is built using **Next.js**, and it directly interacts with the **MultiversX blockchain** to provide a seamless decentralized storage experience.

Features

1. **Blockchain Integration**:
   - Connects directly with the **MultiversX blockchain**.
   - Authenticates users via their wallets.
   - Executes calls to endpoints and views from deployed smart contracts.
2. **IPFS Integration**:
   - Communicates with a backend server to upload and retrieve files stored on **IPFS**.
   - Enables users to upload, manage, and download their files.

User Workflow

1. **Wallet Connection**:
   - Users authenticate by connecting their blockchain wallet.
   - Once authenticated, they can view their files stored in the system.
2. **File Management**:
   - Users can **upload files**, **download files**, and **add tags** to organize them.
   - Files can be sorted, and users can view statistics, such as storage usage or file types.
3. **File Upload Process**:
   - When a user uploads a file:
     - The frontend sends a request to the backend server to upload the file to **IPFS**.
     - Once the file is successfully uploaded, the backend returns the file's metadata.
     - The frontend then initiates a blockchain transaction to store the file's metadata on the blockchain, associating it with the user's wallet.
4. **File Retrieval**:
   - Users can easily retrieve their files, as metadata stored on the blockchain points directly to the file on IPFS.

Backend: Express.js Server

The backend is implemented using **Express.js** to handle the communication between the frontend and **IPFS**. It acts as an intermediary to overcome the limitations of using the **js-kubo-rpc-client** library directly in the frontend.

Key Features

1. **File Management via IPFS**:
   - The backend uses the **js-kubo-rpc-client** library to upload and download files from **IPFS**.
   - Ensures secure and efficient file handling.
2. **Overcoming Limitations**:
   - Initially, the project attempted to use **js-kubo-rpc-client** directly in the **Next.js API endpoints**.
   - This approach failed due to **Node.js-specific dependencies** required by the library.
   - To resolve this, a standalone backend server was introduced to handle file communication with IPFS.

API Endpoints

1. **`POST /upload`**:
   - Receives the file from the frontend.
   - Uploads the file to IPFS via **js-kubo-rpc-client**.
   - Responds with the file's metadata, including its **CID (Content Identifier)**.
2. **`GET /file/:cid`**:
   - Retrieves a file from IPFS based on its CID.
   - Returns the file to the frontend for user download.

Technical Workflow

1. **File Upload**:
   - The frontend sends the file to the `/upload` endpoint.
   - The backend uploads the file to IPFS and returns its metadata (e.g., CID).
   - The frontend initiates a blockchain transaction to store the metadata on-chain, associating it with the user's wallet.
2. **File Retrieval**:
   - The frontend requests the file from the `/file/:cid` endpoint using the file's CID.
   - The backend fetches the file from IPFS and returns it to the frontend for user access.

Documentation Used:

https://github.com/multiversx/mx-sdk-dapp
https://github.com/ipfs/js-kubo-rpc-client

https://docs.multiversx.com/developers/smart-contracts

https://github.com/ipfs/kubo

Setup

Prerequisites

- **Node.js**
- **Rustup** - The Rust toolchain installer
- **MxPy** - Tool for interacting with the blockchain
- **SC-Meta** - Universal smart contract management tool
- **Docker** (including Docker Compose)

IPFS Setup

IPFS is used for decentralized file storage. By default, an IPFS cluster is configured using Docker Compose.

Bring up the IPFS cluster:

```
bash
Copy code
docker compose up -d
```

Custom Configuration

To Change the IPFS Node Ports:

1. Open the `docker-compose.yml` file.
2. Locate the `ports` section for the IPFS service.
3. Modify the host-to-container port mapping as desired. For example:

   ```
   yaml
   Copy code
   ports:
     - "5002:5001"  # Maps host port 5002 to container port 5001
   ```

4. Restart the IPFS services:

   ```
   bash
   Copy code
   docker compose down
   docker compose up -d
   ```

Blockchain Setup

1. **Navigate to the SmartContract directory**:

```
bash
Copy code
cd dc-smart-contract
```

2. **Build Dependencies**:

```
bash
Copy code
sc-meta all build
```

3. **Generate Interactors**:

```
bash
Copy code
sc-meta all snippets
```

4. **Deploy the Blockchain Application**:

```
bash
Copy code
cargo run deploy
```

Note: Copy the contract address printed in the console for frontend configuration.

5. **Custom Configuration**:
   - To change blockchain network ports, review configuration files like `Cargo.toml` or scripts before deployment.

Backend Setup

The backend server, built with Express.js, facilitates communication between the frontend and IPFS, handling file uploads and downloads.

1. **Navigate to the backend directory**:

```
bash
Copy code
cd backend
```

2. **Install Dependencies**:

```
bash
Copy code
npm install
```

3. **Custom Configuration**:
   - **Change Backend Server Port**: Open the server configuration file (e.g., `server.js` or `.env`) and update the port definition:

```
javascript
Copy code
const PORT = process.env.PORT || 3000;
```

4. **Start the Backend Application**:

```
bash
Copy code
npm run dev
```

Frontend Setup

The frontend, built with Next.js, connects to the blockchain, interacts with the backend, and provides a user interface for file management.

1. **Navigate to the frontend directory**:

```
bash
Copy code
cd frontend
```

2. **Configure Environment Variables**:

```
bash
Copy code
cp .env.example .env
```

Update the `.env` file:

```
env
Copy code
DECENTRALSTOREADDRESS=<your_contract_address>
NEXT_PUBLIC_BACKEND_URL=http://localhost:3000
```

3. **Install Dependencies**:

```
bash
Copy code
npm install
```

4. **Start the Frontend Application**:

```
bash
Copy code
npm run dev
```

Additional Configuration Steps

Changing Backend Port

1. Update the port in `backend/server.js` or `.env`:

```
javascript
Copy code
const PORT = process.env.PORT || 3000; // Change 3000 to your desired
port
```

2. Update Docker configurations if containerized:

```
yaml
Copy code
ports:
  - "new_host_port:container_port"
```

Changing Frontend Port

1. In the `.env` file, set:

```
env
Copy code
PORT=4000  # Or any desired port
```

Updating Docker Compose Ports

1. Adjust service ports in `docker-compose.yml`:

```
yaml
Copy code
services:
  ipfs:
    ports:
      - "5002:5001"
```

2. Restart Docker services:

```
bash
Copy code
docker compose down
docker compose up -d
```

Integrating Blockchain Contract Address

1. Copy the contract address after deployment.
2. Update the frontend `.env` file:

```
env
```

```
Copy code
DECENTRALSTOREADDRESS=<copied_contract_address>
```

Project Workflow Summary

1. **IPFS Setup**: Start the IPFS cluster using Docker Compose.
2. **Blockchain Deployment**: Build, deploy smart contracts, and copy the contract address.
3. **Backend Setup**: Configure and run the Express.js backend for IPFS interaction.
4. **Frontend Setup**: Configure environment variables, customize ports, and start the Next.js frontend.
5. **Customization**: Adjust configurations in Docker Compose, backend, and frontend files.