Takneek: Hall 2

Documentation

Innovate: OverTime

Y24
Daksh M Jain
Shivansh Jaiswal
Harshit Karnani

Y23
Pratyush Singh
Arnav Gupta
Abhimanyu Solanki

September 4, 2024

Contents

1 Introduction

This project aims to develop a decentralized blockchain-based platform for automating task allocation. The platform leverages Ethereum, a decentralized, open-source blockchain that supports smart contracts—self-executing contracts with publicly available code. Ethereum enables trustless transactions and decentralized applications (dApps) by ensuring that all operations are transparently recorded on the blockchain.

In this project, workers register with their availability, expertise, and wage requirements, while tasks are managed by an Admin. The smart contract handles task allocation and payments entirely on-chain, ensuring transparency and eliminating the need for intermediaries. MetaMask is integrated for secure account management, allowing users to interact seamlessly with the Ethereum blockchain. This architecture guarantees that all interactions are immutable, verifiable, and securely executed within the Ethereum ecosystem.

2 Contracts

The smart contract is written in Solidity. The contract named `Overtime` facilitates the registration of workers, the addition of tasks by an admin, and the automated allocation of these tasks based on workers' availability, expertise, and wage requirements.

2.1 Worker Registration (`registerWorker`)

- `hours`: Number of hours available
- `expertise`: Expertise level of the worker
- `minWage`: Minimum acceptable hourly wage
- `wallet`: Wallet address for payment

The `registerWorker` function allows any user to register as a worker on the platform. This function requires three parameters: the number of hours the worker is available, their expertise level, and the minimum hourly wage they are willing to accept. The contract maintains a mapping of registered workers and categorizes them based on their expertise and minimum wage. It aims to maximize the total number of tasks fulfilled by greedily choosing the most expensive workers that satisfy the bound on the hourly wage requirements, thus allowing efficient task allocation, and reserving the cheaper labor for other tasks. It is also possible for the tasks to be divisible and thus be divided between different workers to make the task completion more efficient.

2.2 Task Allocation (`addTask`)

- `time`: Required time for the task
- `expertise`: Minimum Expertise level required
- `dependencies`: Task dependencies

- `wage`: Hourly wage
- `deadline`: Task deadline
- `divisible`: Whether the task is divisible

Tasks are added to the platform by the admin using the `addTask` function. Each task specifies the required time, expertise level, hourly wage, deadline, dependencies (if any) and whether the task can be divided among multiple workers. The task allocation process is executed in a decentralized manner, ensuring that tasks are assigned to workers who meet the criteria and have the availability to complete the work. If the task is divisible, it can be split among multiple workers until the required time is met. If it is not divisible, the task is assigned to a single worker who satisfies the conditions.
The function to allocate the newly added task to workers is executed every time `addTask` is called.

2.3 Check Status (`checkStatusTask`)

- `taskId`: ID of Task

The `checkStatusTask` function is executed every 5 minutes, and attempts to add more workers to the already existing tasks, and with that it properly allocates workers to tasks which are not yet fully allocated.

2.4 Payment Processing (`completeTask`)

- `taskId`: ID of Task
- `address`: Address

Upon the completion of a task within the deadline, the admin triggers the `completeTask` function, which transfers the payment to the worker. This function ensures that the worker is paid according to the agreed-upon hourly wage and the time they spent on the task. The payment is executed on-chain, providing transparency and ensuring that workers receive their compensation directly to their wallets.

3 Front End

The front-end of the application is developed on NextJS. It enables users to register as workers, view available tasks, and manage their accounts via MetaMask. It also allows admin to manage the tasks, and analyse the task statuses.

3.1 MetaMask Integration

MetaMask is a popular Ethereum wallet used for interacting with decentralized applications (dApps). The front-end integrates MetaMask to manage user accounts securely. Upon loading the application, the front-end checks if MetaMask is installed and connected. If MetaMask is not detected, the user is prompted to install it. When a user attempts

to connect their wallet, the front-end requests account access through MetaMask. Once connected, the user's Ethereum address is stored and used for further interactions with the smart contract.

3.2 Task Management Interface

The user interface allows the admin to add new tasks, specifying all necessary details such as required time, expertise level, and hourly wage. Workers can view available tasks and the status of their assigned tasks. The interface dynamically updates based on the user's role (admin or worker) and the data fetched from the blockchain, ensuring that all information is current and accurate.

4 API Endpoints

1. **POST** `/addTask`: Adds a new task to the smart contract.
   - **Request Body:**
     - `time` (number): Task duration.
     - `expertise` (string): Required expertise.
     - `dependencies` (array): Task dependencies.
     - `wage` (number): Offered wage.
     - `deadline` (number): Task deadline.
     - `divisible` (boolean): Task divisibility.
   - **Response:**
     - **Success:** `200 OK` with a confirmation message and transaction hash.
     - **Error:** `500 Internal Server Error` with an error message.
2. **POST** `/addWorker`: Adds a new worker to the smart contract.
   - **Request Body:**
     - `hours` (number): Availability in hours.
     - `expertise` (string): Worker expertise.
     - `min_wage` (number): Minimum acceptable wage.
     - `wallet` (string): Worker's wallet address.
   - **Response:**
     - **Success:** `200 OK` with a confirmation message and transaction hash.
     - **Error:** `500 Internal Server Error` with an error message.
3. **GET** `/checkStatus`: Retrieves the status of all tasks.
   - **Response:**
     - **Success:** `200 OK` with an array of tasks, with the `task_id`, `worker_id`, `status`
     - **Error:** `500 Internal Server Error` with an error message.

4. **POST** `/checkWallet`: Checks the wallet balance of a worker.
    - **Request Body:**
        - `worker_id` (string): Worker ID.
    - **Response:**
        - **Success:** `200 OK` with worker ID and balance.
        - **Error:** `500 Internal Server Error` with an error message.

5 Integration

The pipeline involves deploying the `Overtime` smart contract on the Ethereum Sepolia testnet using Hardhat. This contract manages tasks and workers, including registration and task assignments. We expose the contract's functionality to the front end via an Express.js API, using `ethers.js` to interact with the blockchain and Infura as the provider for Sepolia. The API enables the front end to add tasks, register workers, check task statuses, and retrieve contract details, ensuring direct and efficient interaction between the smart contract and the web application.