Turn Based Game on Blockchain

Arora, Harsh	MCS202208
Chakraborty, Arghyadip	MCS202206
Gangamreddypalli, Namratha	MCS202105
Gantait, Arunava	MCS202201
Iyer, Paarth	MCS202218

December 2022

Contents

1 Problem Statement

The aim of this project is to implement a simple turn-based game on the blockchain. The particular game implemented is **Antakshari**, which is played as follows:

- The first player submits a word
- At each subsequent turn, the player submits a word that begins with the last letter of the previous word.

The validity of each submitted word must be checked in some way, e.g. by making calls to a dictionary maintained by some trusted source outside the blockchain. *Timing constraints* and *elimination criteria* may also be introduced. Passing on a turn may be allowed.

Finally, the game should have some consistent termination and winning criteria.

2 Main Idea

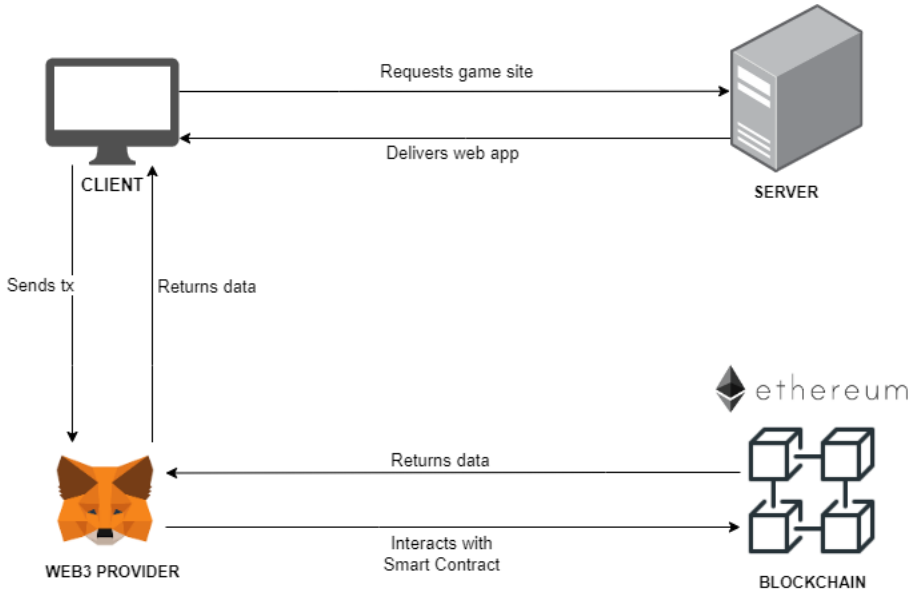

Figure 1: Architecture diagram

The client accesses the game website by entering the url in their browser with the Metamask extension. After which they play the game by using our frontend and sending transactions via Metamask, which interacts with the Ethereum blockchain, and hence our smart contracts deployed on it.

Key Components

Ethereum: It is a decentralized, open-source blockchain with smart contract functionality. Ether is the native cryptocurrency of the platform.

Smart Contract: They are the building blocks we use to create blockchain applications. They are programs that we can write with source code and deploy to the blockchain. They are written in the Solidity programming language.

MetaMask: It serves as a web3 provider and a crypto wallet, whose purpose is to interact with the blockchain. It is distributed as a browser extension.

Ethers js: It is a popular library that encodes and decodes any data prior to, or after interaction with the blockchain.

3 Implementation

3.1 Game features

Judge: There is a predecided judge who is responsible for approving (validating) each submitted word in the game.

Validation Condition: Apart from the condition mentioned in the problem, a word is deemed valid by the judge if it has not been used previously (thus we do not allow repitition of words) and also if it comes from the dictionary implemented in `worddict.js`.

Lives: We have implemented a system of lives. All new players get 3 lives. A life is lost upon submission of an invalid word. Losing all lives results in elimination. When a player passes a turn, they lose a life.

Terminating/winning condition: Our terminating condition is through **last man standing**. When all but one player have been eliminated through the process described above, the remaining player is declared the winner.

3.2 Development tools

3.2.1 Truffle

The Truffle Suite: Truffle advertises itself as "the most comprehensive suite of tools for smart contract development".

How it can be used : Truffle can be used as a development environment, testing framework, and asset pipeline for blockchains using the *Ethereum Virtual Machine* (**EVM**). It manages the entire workflow efficiently for any blockchain-based development. Its key features include:

- Built-in smart contract compilation, linking, deployment, and binary management.
- Deployments and transactions through MetaMask to protect our mnemonic.
- External script runner that executes scripts within a Truffle environment.

How we use it : Our backend is Truffle-based. We have used it for testing our software without having to use the actual Ethereum blockchain and spending actual gas on it.

Ganache: Ganache is a tool for producing "One-click blockchains". It is a personal blockchain for developing software for Ethereum or even Corda.

How it can be used : Using Ganache enables:

- `console.log` in Solidity
- Forking any Ethereum network without waiting to sync
- Ethereum JSON/RPC support
- Impersonating any account without requiring any actual private keys

- Listening for JSON-RPC 2.0 requests over HTTPS/WebSockets
- Mining blocks instantly, on-demand, or at an interval.

among other things.

How we use it : Ganache has also been used for testing, by simulating players using fake private keys and addresses and observing how our code interacts with requests and events.

3.2.2 User Interface tools

Svelte : It is a front-end, open-source JavaScript framework for building UI components for web applications, similar to React, Angular, or Vue. However, unlike the others which ship a runtime component such as Virtual DOM to the browser in order to make the code run, Svelte is a compiler. It converts the declarative code written by the programmer to imperative code that works with native browser APIs. As a result, it provides highly performant code in a small package.

Tailwind : It is a collection of tiny CSS utility classes for quickly and consistently building good-looking websites. Adam Wathan, the creator of Tailwind, maintains that "semantic class names are the reason CSS is hard to maintain", i.e., it is nigh impossible to name CSS classes in a sane way and that is what makes CSS so challenging. Unlike frameworks such as Bootstrap and Material, which address this challenge by creating styles for high-level components such as buttons, dropdowns and forms, Tailwind takes a more **functional** approach by providing utility classes that can be composed together in various ways to build components.

3.3 Backend

3.3.1 Events

Events are essentially a way to write to the Ethereum logs. These logs send out notifications when new entries are made and have been set up in a way that you can subscribe or watch for new entries. We listen to these events to update the frontend.

Defining events :

```
1       ...
2       event PlayerJoined(address player);
3       event PlayerRejected(address player);
4       event PlayerLeft(address player);
5       event PlayerWon(address player);
6       event GameStarted(address cont);
7       event Approval(string word);
8       event Turn(address player, uint256 playerLives, uint256 nextTurn, string word,
            bool correct);
9       ...
```

Emitting events :

```
1     ...
2     function startGame() external {
3       require(msg.sender == owner, 'Not Game Owner!');
4       if (!started) {
5       started = true;
6       if (alive <= 1) {
7         winner = owner;
8         emit PlayerWon(winner);
9       } else emit GameStarted(address(this));
10    }
```

```
11  }
12    ...
```

3.3.2 Code

```
truffle
├── contracts
│   ├── WordGame.sol
│   └── WordGameFactory.sol
├── migrations
│   └── 1_deploy_contract.js
└── truffle-config.js
```

Files

- `WordGame.sol` : This source file contains the word-game contract which contains the functions that implement the game
- `WordGameFactory.sol` : This creates a **factory contract** for the Word Game contract class. A factory contract is used to deploy new smart contracts from a smart contract class
- `1_deploy_contract.js` : This deploys the factory contract
- `truffle-config.js` : This is the Truffle configuration file

3.4 Frontend

The frontend allows the user to create games, join existing games, and look at words already played.

3.4.1 Code

```
client
├── src
│   ├── lib
│   │   ├── CreateJoin.svelte
│   │   ├── Game.svelte
│   │   ├── GameScreen.svelte
│   │   ├── MetaMask.svelte
│   │   ├── Navbar.svelte
│   │   ├── PlayersTab.svelte
│   │   ├── WaitingScreen.svelte
│   │   └── WordsTab.svelte
│   └── App.svelte
└── scripts
    ├── judge.js
    └── worddict.json
```

Files

- `Game.svelte` : Combines and displays either the Gamescreen or the Start/Join screen or the Waiting screen depending on the stage
- `GameScreen.svelte` : Contains the HTML UI elements of the game screen, like text field to enter the word, button to submit the word, etc, and the code for events related to these elements.

- `MetaMask.svelte` : Checks if the user has installed the MetaMask plugin and connects to the user's MetaMask account
- `Navbar.svelte` : Contains UI elements for the navigation bar
- `PlayersTab.svelte` : Shows the list of players who currently joined the game.
- `CreateJoin.svelte` : Contains buttons to Start and Join the game, and their functionality.
- `WaitingScreen.svelte` : Displays the waiting screen which is the screen shown after we have joined a game but the game has not started yet
- `WordsTab.svelte` : Contains and displays a list of the last five/six words submitted
- `judge.js` : Waits for an approval event from WordGame contract and sets the approval to be true if the word is in the dictionary `worddict.json`
- `worddict.json` : This is a dictionary of valid words

4 Challenges/Improvements

It was not possible to include a timer because you have to keep track of transaction hash rate: essentially the only way to know time is when a new block is added to the blockchain. For an auction, this is fine since any transaction after the time limit can be discarded. However, for our game, we want to ensure that every player has a time limit after which the turn proceeds to the next player and every player needs to know which player failed to answer within the time limit.

It was also not possible to include a Merkle tree due to lack of time which would have gotten rid of the judge. Instead, we have used a JS script `judge.js`. It is itself outside the blockchain but is connected to the blockchain via an account. It listens for an approval event from the game contract and sends approval accordingly.