Master Degree in Computer Science

Peer To Peer and Blockchain Project: Battleship

Francesco Caprari:580154

Academic Year: 2022-2023

Contents

1 Dapp Application

The project involves creating a **Battleship game** where match management occurs on the blockchain. The project consists of a Solidity file that manages the core of the application and a JavaScript and HTML part for the frontend of the project.

2 Smart Contracts

To implement the functionalities, a single contract is implemented, which is used for the operations of creating or participating in a match and for managing the various phases of the actual gameplay. Additionally, the **@openzeppelin/contracts** library is used for verification operations on the Merkle Tree and to perform arithmetic operations safely.

2.1 Structure

To store the information of each match, i use a **struct** that will contain details about the participating players, the state of the game boards and the stakes involved. Similarly, to store the information of each player participating in the game, an additional struct is used, which includes not only the actual address but also other information regarding the current game.

Struct player

- *address payable player*: the address of the player.
- *bytes32 Root*: the root of the merkle tree.
- *uint hitb* : number of cells hit where is a piece of ship.
- *uint money*: money bet by the player.
- *Shoot[] Shoots*: track the shot made by the player and the result of each shot.

Struct SingleGame

- *Player firstPlayer*: info of the first player.
- *Player secondPlayer*: info of the second player.
- *address playerturn*: who is the player in the current turn.
- *bool firstShot*: boolean var used for check the phase.
- *phase currph*: store the current phase of the game.
- *uint moneyDeal*: the money bet by the two player.
- *uint8 Boardim*: dimension of the board.
- *uint8 Boats*: number of cells where there is a part of a ship.

- *uint256 BlockNumber*: the last block number of the game, this element is used in the function to report an inactive player.

ListGames: to store each game, it is used a map of **SingleGame**, which associates the corresponding game with each ID. Using this type of structure, each function of the Smart Contract will require the Game's Id as input.

2.2 Project choice

The smart contract includes functions to manage the creation and participation in a game.

- *CreateGame()*: the function generates a random Id, create using a block timestamp, the player's address, and a nonce, using a nonce ensures that a different number is generated each time. The id is assigns it to the new game, the msg.sender address is assigned at the first player and also the size of the board and the number of cells occupied by the ships are stored in the contract. The game transitions to a new phase where it awaits for a new player.

- *Joingame(uint Id)*: with this function, the contract using the Id assign the address of the msg.sender to the second player and the game is no longer joinable by other players, after we move into a phase were each player bets the ETH he want use.

- *JoinRnd()*: it is identical to the previous function, except that the game is chosen random, using a block timestamp, the player's address and a nonce modulo the number of available matches.

When both players are participating in the Game, there is a phase where they bet their own ETH, if the amount is equal, the placing phase begins where users decide the position of Ships. To commit the root, in the FrontEnd part, the player construct the Merkle tree and extract the root, which is passed as input to the function of the smart contract and stored in the corresponding game's struct. When both roots have been inserted, an event is emitted indicating that the shooting phase begins. For the shooting phase there are two functions:

- *attack(uint256 Id,uint8 pos)*: with this function, the player indicate the position he wants to target and an event is triggered for the opponent.

- *verify(uint256 Id,memory proof, leaf, pos,shotr)*: who will handle the event, check the result of the shot and call this function where the Player generate a Merkle tree proof demonstrading that his secret board configuration satisfies the hit/miss result he claim, the player will certify the correctness of the result. If the player hit a ships the contract update the number of cells hits and it is checked if the player has won. At last the contact emit an event that notify the player who has taken a shot of the shooting result.

If the number of cells with a positive outcome is equal to the number of cells occupied by the ships, the contract send an event (**event CheckWinner()**) to the potential winner. At that point, the user can invoke the function *verifyWin()*. This function perform multiple verification on the remaining leaves of the Merkle tree that were

not verified during the game, the function indeed receives the multiple proof and the remaining leaves to be checked. The contract verify also the if boats are in the right position corresponding at the shot hit the contract store.
function Playerafk(): The smart contract also features a function to check if the opposing player is inactive. To achieve this, every time a move is made in the game, the current block number is stored in the **BlockNumber** variable of the struct. During the various phase, if the player report the opponent, the contract checks whether the block number is greater than the last block number of the game plus a predefined value.

3 Front End

For the frontend part, in JavaScript, there is a listener function that remains responsive to events emitted by the smart contract and acts accordingly and there are various functions called when they are invoked by different buttons, that execute the functions of the smart contract. Except for the initial forms to create or join an existing match, the others are shown to the user only once a certain event is received, indicating a change in the game phase. When a new game is created, or a user joins, the **game's ID** and the **user's address** are saved. These will be used later for interactions with the smart contract. Since the player can choose from 3 possible board sizes: *4x4*, *6x6*, *8x8*, the number of occupied cells by the ships and the current match's board size are also stored. This data is used later to appropriately generate forms and tables. When the user wants to send their game board to the smart contract, there are initial checks on the correct placement of the ships on the board, then, there is a phase where the **Merkle tree** is created from the leaves 0 and 1 by summing them with a random value, the tree and the grid with 0,1 values is also saved and will be used later to generate the proof to be sent to the smart contract. When the event informing the player about the shot being fired is captured, the player checks in the array whether the position contains a ship or not, then he **generate a proof**, which he will send to the smart contract to demonstrate the accuracy of the shot's outcome. To invoke the **verifywin** function, the "view" function (**GetTocheck()**) of the smart contract is called first, which returns the cells for which the proof has not yet been submitted. Then, using the returned values, the multi-proof is generated to be sent to the smart contract. Two **libraries** are used, one for managing operations on the *Merkle Tree* and the other to utilize the *keccak256* hash function used to create the nodes of MerkleTree.

4 Vulnerabilities

One possible issue lies in the final check on the ship positions. In the smart contract, it is only verified that the number of positions occupied by the ships matches the number of ships declared at the beginning of the game. A better approach would be to send the encrypted coordinates when the user commits the board and then verify the positions when the user sends the board to check for victory.
Another potential issue arises from the fact that during the shooting phase, two transactions are invoked from the smart contract, one from the player and one from the opponent. To save **GAS**, it might have been advisable to create a function that ac-

complishes both tasks: storing the hit position and providing the relevant proof for the previous shot.

5 Gas analysis

Table 1: Tabel of Gas

Gas Cost	
Operations	**Cost**
createGame()	310591 - 415010
joinGame()	119228 - 153217
joinrndGame()	104086
proposedBet()	110415 - 144402, 120979 - 154967
attack()	59732 - 76716
verify()	50924 - 74008
Playerafk()	33455 - 32216
checkwin()	126184 - 264416
storeRoot()	91781 - 125780, 97482 - 131458

In The fig.1 there are the cost of each operation in the smart contract. The cost varies in the case of **storeRoot** and **proposedBet**. This is because an event is emitted when both players have called the function, as a result, the cost will be different for one of the two players. The cost varies also in almost all functions depending on the size of the board chosen by the player and the number of games being played at that moment the function is called.
The **Total** gas cost spend by **one player** to complete a game with an 8x8 grid, where each player never hits a position where a boat is located and the boats on the board occupy 14 positions, is about: **5.217.351**, is similar for the other player who join the game. The total cost includes the gas used to create the game, pay the amount, commit the Merkle tree root and the various phases in which the user shoots the shot and sends the tree proof.

6 Files

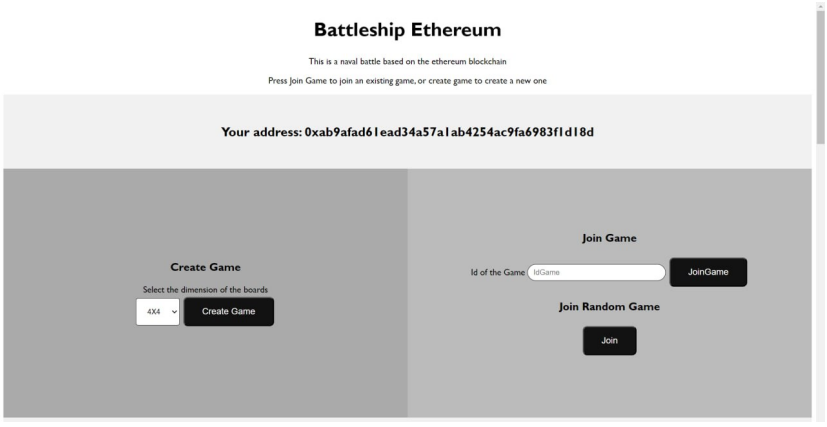
In contract Folder there is the smart contract **GameRule.sol**, while in the folder src there are the files **js/app.js**, **index.html** and the file *.css* for style management. There are also files in the **Test folder** used for gas analysis and to test some of the functions of the smart contract.

7 User Manual, Prerequisites and Instruction to Execute

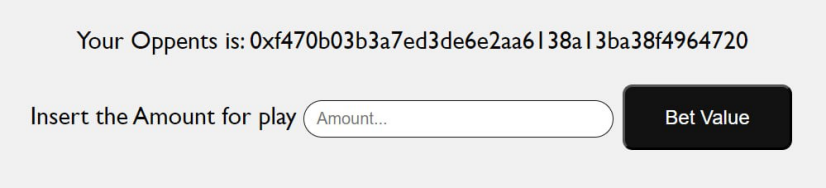
- Install Ganache to simulate the Ethereum blockchain and load the Truffle configuration file (The *truffle -config.js* file in the smart contract folder).
- Insert the correct port corresponding to the port present in Ganache in the truffle-config.js configuration file.
- Install **npm** to execute.
- In the Battleship folder run **npm ci** for installing the libraries.
- In the same folder **truffle migrate** to deploy the contract on the blockchain.
- In the same folder, run **npm run dev**, at this point, the default web browser opens, displaying the main page at localhost:3000, before starting, make sure you have logged in to the Metamask extension with one of the Ganache accounts and ensure that the selected network in Metamask is localhost:7545.
- open also another browser in the same localhost:3000 and in localhost:3001, sync Options disable all. Make sure to use two different accounts.

8 Interface

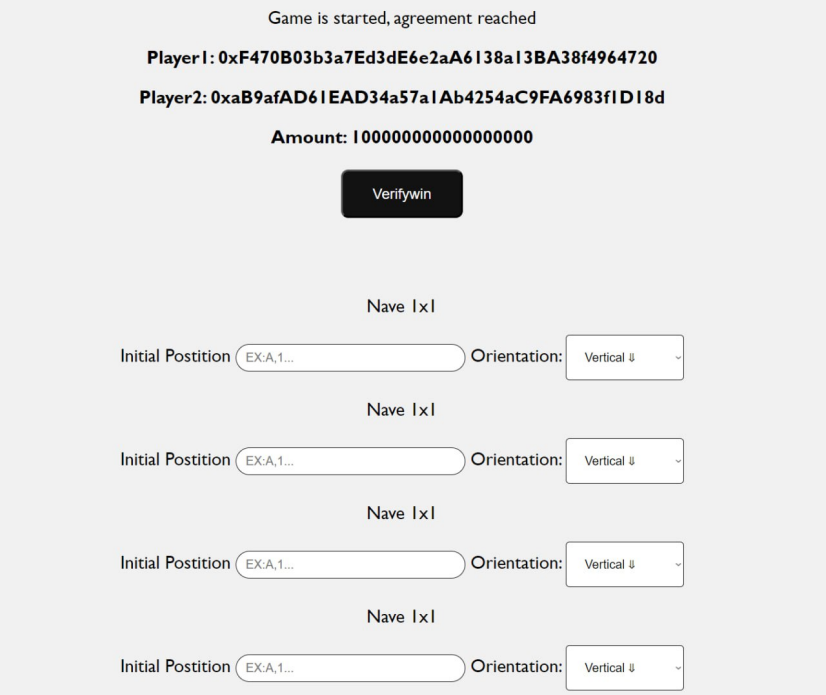
The player can choose the size of the game board using the dropdown menu, there are three possible choices: 4x4 with 4 "cells ships", 6x6 with 10 "cells ships", and 8x8 with 14 "cells ships". By clicking **Create Game**, the game is created, and the game's ID is displayed on the screen. A challenger, if present, can either play a random game or enter the specific game ID.



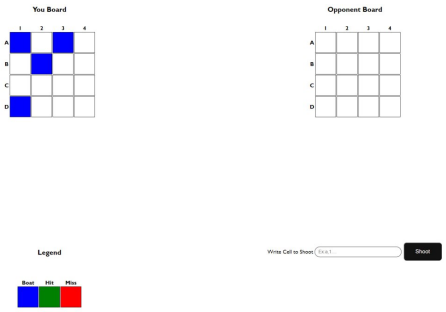
When there are two players in the game, both will see a screen where they can enter the amount to wager.



If the Ethereum amount is the same, both will see a game summary and the form where users can decide the placement of their ships, indicating the starting position and orientation.

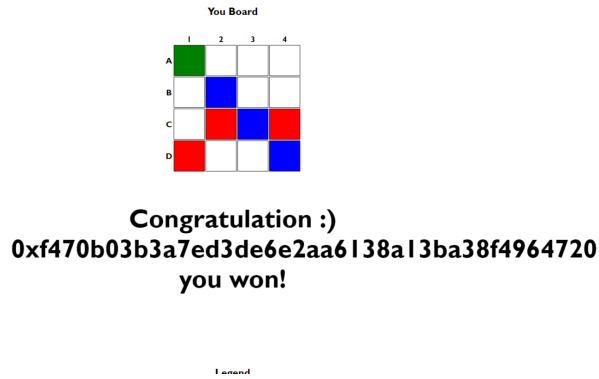


After both players have committed their boards to the smart contract, the shooting phase begins. The players will see their own board and their opponent's board. At the bottom right, there will be a form to select which cell to target. Initially, the shooting form will appear for the player who created the game, once the button is clicked, the form will disappear and appear for the other player.
When the shot is fired a transaction is emitted to the smart contract, but the opponent will also initiate a transaction it is automatically created when it receives the shooting event. With this transaction the player will send a Merkle proof demonstrating that his secret board configuration satisfies the hit/miss result. If the test is correct, an event is sent from the smart contract that informs the player who shot whether the shot hit the target.



When the player proposes an amount and waits for the other player, he can report it. The same happens when a user commits the positions of the board also in the case where the user is waiting for the opponent to take the shot.

When a user hits all the ships, a possible victory event is emitted, which generates a button on the screen. If clicked, it triggers the execution of the *verifyWin* function that checks for the player's potential victory



If the *verifyWin* function successfully completes, it effectively certifies the player's victory and is displayed on the screen.
(ADVISE:Since many forms and paragraphs are displayed to the user after receiving an event, it might take a few seconds before the elements appears on the screen.)