Report Peer to Peer Systems & Blockchain project

Francesco Benocci 602495

14 luglio 2023

Contents

1 Introduction

The following report concerns the final project of P2P Systems and Blockchains course. It is about a Battleship Game developed using the Ethereum blockchain. The full project is available on github: github.com/Benocci/P2P_Project.

The project in two different parts: a classic front-end developed in HTML, css and Javascript and a Solidity contract which represent the back-end. The communication between the two parts takes place through the calls of the Solidity methods in Javascript and the events emitted in the contract.

2 Project Organization

The project is organized in a directory structure, the main directories are:

- contracts: which contains the file BattleShipGame.sol with the solidity code;
- src: which contains the front-end structure with index.html, js folder with the app.js and others javascript code and the css folder with style.css.S

3 Project functioning

3.1 Front-end side

The main website page is composed by a menu with three possible choices: to create a game, to join a random game or to join a specific one. This three actions are handled by function calls in the contract that generate (or take) all the information to start the game.

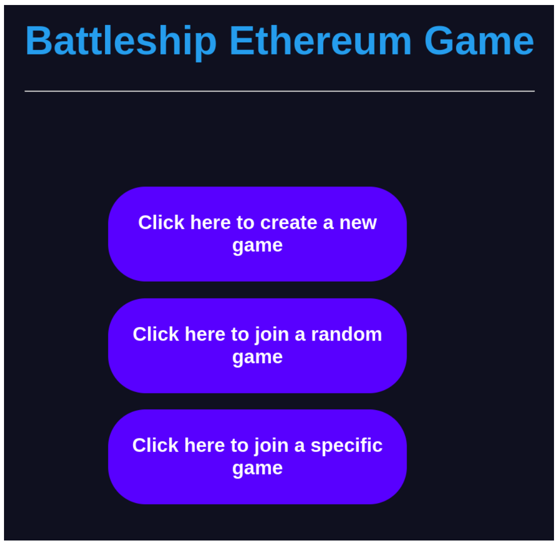
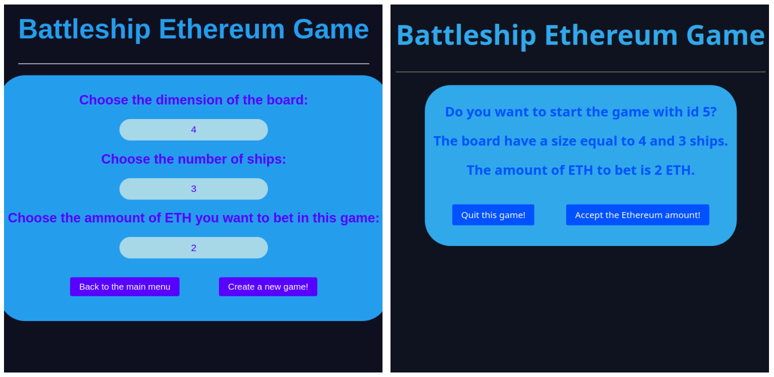

Battleship Ethereum Game
Click here to create a new game
Click here to join a random game
Click here to join a specific game

Figure 1: Menu

At this stage of the game we find the decision of the Ethereum amount to bet for playing a game. It was decided to entrust the choice of this value to the creator of the game, so when the other player tries to join the game, he can see the ETH Amount (and all the other information about the game). At this point he can decide to accept or refuse with no consequences. If he accepts to join with those properties the game can finally begin.


Battleship Ethereum Game

Choose the dimension of the board:

4

Choose the number of ships:

3

Choose the ammount of ETH you want to bet in this game:

2

Battleship Ethereum Game

Do you want to start the game with id 5?

The board have a size equal to 4 and 3 ships.

The amount of ETH to bet is 2 ETH.

Figure 2: Game creation and accepted ETH amount

The entire game is based on a 2D array representing the game board, where each cell contains either a 0 or 1. The value 0 means that the cell is empty, and 1 means that the cell contains a ship. It was decided that it is possible to place a single ship on any tile of the board with no constraints about the position, to avoid any type of overlapping of the ships and to guarantee the maximum freedom of positioning. After the placement phase (and thus the creation of the matrix) the players have to submit the board. This action results in the creation of the merkle tree.

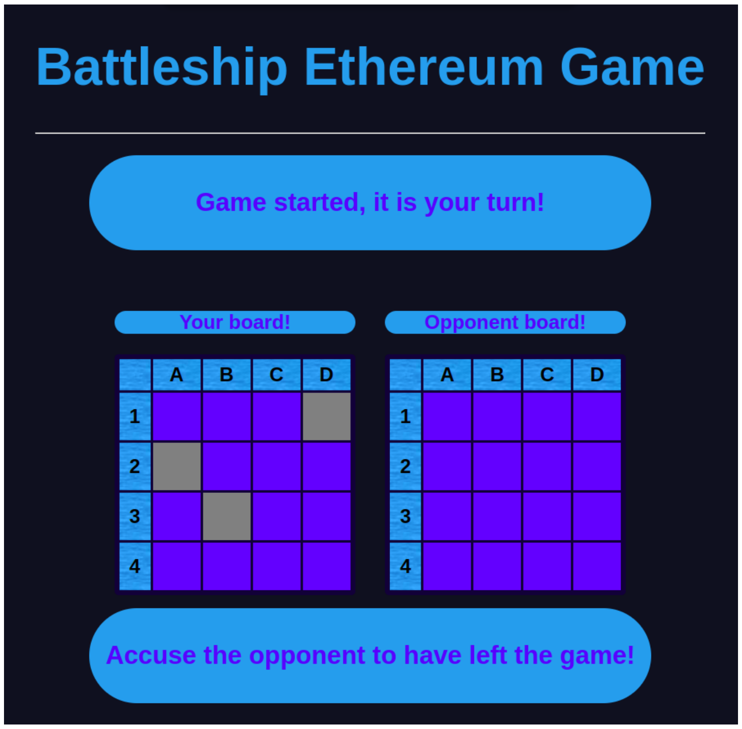

Battleship Ethereum Game
Game started, it is your turn!
Your board!
Opponent board!
A
B
C
D
1
2
3
4
A
B
C
D
1
2
3
4
Accuse the opponent to have left the game!

Figure 3: Game board

To generate the first layer of the merkle tree (the leaves layer) all the values of the matrix are concatenated with a random salt and than hashed with Keccak256. The leafs just generated are then placed into an array, which is then inserted into another array called ”merkleTree”, after that there is the generation of the merkle tree with a concatenation of the leaves to generate level after level all the tree up to the root. Lastly the root is placed in the last position of the array ”merkleTree”. At this point after sending the Merkle root to the opponent player the battle phase can begin.

During his turn (identified by a boolean) a player can click a cell of the opponent board to attack that cell, which results in a function call to the contract to send the coordinates of the fired cell to the opponent. He generates a Merkle Proof array with all the cells necessary to rebuild the merkle root and he sends it to the contract with a function call. The result of this call (and its validity check) is sent to the opponent player with the response of his shot.

When a player hits all the opponent's ships the front-end captures the event from the contract, shows the winner and a button to go back to the main menu.

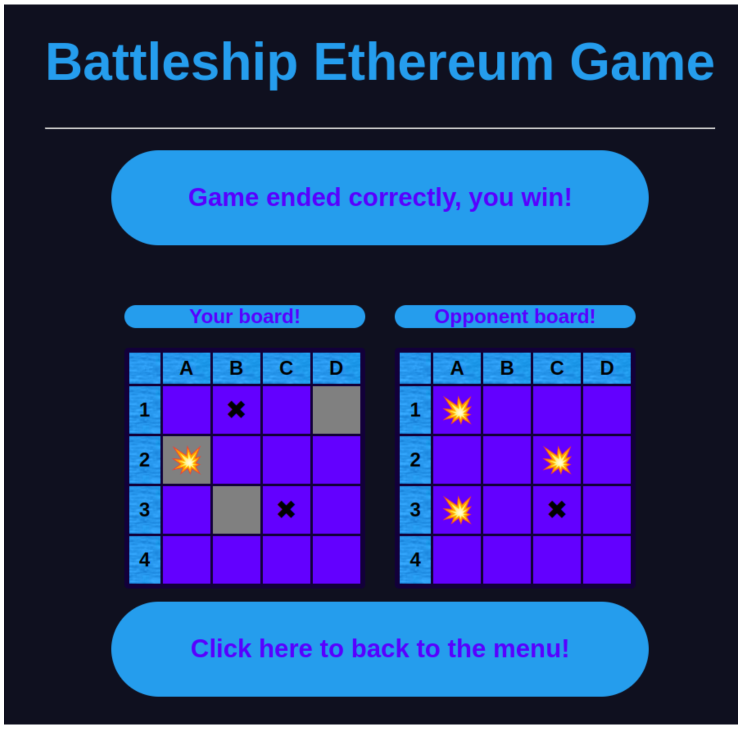

Battleship Ethereum Game
Game ended correctly, you win!
Your board!
Opponent board!
A
B
C
D
1
2
3
4
Click here to back to the menu!

Figure 4: Game ended, you win!

3.2 Contract side

In the Solidity code a game is represented with the struct "gameInfo". This struct holds all the information about a game, including the creator and joiner addresses, board size, number of ships, ether amounts, merkle roots, and ship counts for both players. All the games are saved in a map called "gameList" which associates them with an unique Id. There is also an array containing all the Id of the all the joinable games.

The creation of a game consists in a function of the contract that adds the "gameInfo" to the map after the generation of the Id. The Id is put into the array. At this point, another player can call the joinGame function that adds all the remaining information about the game. After that, to complete the first part of the game, the joined player has to accept the ETH amount sending his part of the bet.

At this point there is the placement phase, which ends with the submission of the respective merkle roots. After this the game can start and the hit/miss communication works with two function:

- `shot(uint256 _gameId, uint256 _row, uint256 _col)` called by the shooter.

Besides the gameId it has as arguments the coordinates of the fired cell. It sends them to the opponent player with an emit of the event "ShootShip";

- `shotResult(uint256 _gameId, uint256 _row, uint256 _col, uint256 _result, bytes32 _hash, bytes32[] memory _merkleProof)` called by the player who was shot. This function takes as input the merkle proof generated by the front-end and the result of the shot and uses them ti recunstruct the merkle root to make the comparison with the one saved in the "gameInfo". It also checks if all the ships have been hit and in that case declares the end of the game with the "GameEnded" event.

The last function of the contract is the "accuseOpponent". This function is called when a player want to accuse the other player to have left the game. It works with two values of the "gameInfo": the first one is ".accuser" that contains the address of the accuser, with the second one is ".accuastionTime" that contains the block number of when the accuse was made plus five. When the function is called and the current block is equal or higher than the accusationTime the game end and the accuser wins.

4 Main decisions

4.1 Value constraints

The constraints on the values of the game board are as follows:

- *boardSize*: must be a positive integer power of 2 to have a complete merkle tree without completion values;
- *shipNum*: must be a positive integer less than the power of boardSize divided by 2.

4.2 Merkle tree

The merkle tree is represented by a 2D array created inside the function `createMerkleTree`. Each array represents one level of the tree, with the first one (index 0) that is created with the Keccak256 of the concatenation of the value of any cell with a random salt.

```
var temp = [];
for (let i = 0; i < boardSize; i++) {
  for (let j = 0; j < boardSize; j++) {
    temp.push(window.web3Utils.soliditySha3(myBoardMatrix[i][j].toString()
    + Math.floor(Math.random() * 10)));
  }
}
```

Then subsequent levels are built from previous arrays iteratively.

```
while (temp.length > 1) {
  const nextLevel = [];
  for (let j = 0; j < temp.length; j += 2) {
    const leftChild = temp[j];
    const rightChild = temp[j + 1];
    nextLevel.push(window.web3Utils.soliditySha3(leftChild + rightChild.slice(2)));
  }
  temp = nextLevel;
  merkleTreeMatrix.push(nextLevel);
}
```

4.3 Merkle proof

The merkle proof is an array generated by the function `createMerkleProof` each time a player has to prove the result of a shot. The function takes for each level(array) of the merkle tree the siblings of the node to verify that the result is valid. The `flatIndex=(row * boardSize) + col` allows to find the sibling.

```
for (let i = 0; i < (merkleTree.length - 1); i++) {
  if (flatIndex % 2 == 0) {
    merkleProof.push(merkleTree[i][flatIndex + 1]);
    flatIndex = flatIndex / 2;
  }
  else {
    merkleProof.push(merkleTree[i][flatIndex - 1]);
    flatIndex = (flatIndex - 1) / 2;
  }
}
```

4.4 Inactivity accuse

The mechanism of accusation of inactivity consists of two Solidity functions:

- *accuseOpponent*: set the accuser and the accusationTime of the game in the gameList and emit an event for the opponent;
- *verifyAccuse*: check that the accuseTime has passed and if so, end the game and send the ETH.

The mechanism in the JS side works with an accusationBlock value which is set in case of accusation and which enables the call to verifyAccuse at each captured event (and therefore at each block).

5 User manual

To try out the project is needed Ganache, Truffle and Metamask as saw at laboratory lessons. So first of all is necessary to execute Ganache and start an appropriate workspace. The second thing to do is compile the contract, to do this in the P2P_Project/migrations folder is necessary to run the command "truffle migrate".

After this in the P2P_Project folder to execute the code is possible to run "npm run dev" and a browser page is opened automatically. Assuming to have already a network created on Metamask the only things to do is log in and choose an account. To use another account on the same network is necessary to open "http://localhost :3001/" and disable all the synchronization settings. After this it is possible to open another browser page and go to "http://localhost:3000/" being careful to change the account on Metamask.

6 Potential vulnerabilities

6.1 Random Number Generator

The function "`getRandomNumber()`" generate a random number using the sender's address and block timestamp. These two values can be manipulated by miners or attackers in certain situations. For this reason this implementation is not secure for critical use cases like gambling games and it is recommended to use an external oracle or a trusted randomness provider.

6.2 Gas Limitation

There are loops that iterate over arrays in the contracts. These loops could potentially consume excessive gas, leading to out-of-gas errors. It would be advisable use other data structures to avoid gas limitations.

6.3 Integer Underflow/Overflow

The contract performs various arithmetic operations without explicit checks for integer underflow or overflow. This can lead to a "underflow/overflow attack" but since solidity 8.0 the checks are done by default by the compiler and therefore this problem does not arise.

6.4 Front-Running Attacks

Some functions, such as joinGame(), amountEthDecision(), involve actions based on the state of the blockchain (e.g., block.timestamp, block.number). Attackers can attempt to manipulate these values to their advantage in certain situations.

7 Gas evaluation

The functions of the contract with the respective gas evaluation are presented below, all the values presented are an average of the gas value:

- `createGame`: around 240000;

- `joinGame`:
  - random join: around 60000;
  - specific join: around 67000;
- `amountEthDecision`: around 36000;
- `submitBoard`: around 50000
- `shot`: around 31000;
- `shotResult`:
  - miss: around 50500;
  - hit: around 54000;
  - hit with final win: around 63500.

In the code each function has indicated an example of gas used and total price.

7.1 Gas consumed in a 8x8 game

We consider a 8x8 game board with three ship placed, assuming that the players make the maximum number of misses possible. In this case we have the following function call by the two players:

7.2 Gas used by player 1 (creator)

1. 1x createGame
2. 1x submitBoard
3. 64x shot
4. 61x shotResult miss
5. 2x shotResult hit
6. 1x shotResult win

7.3 Gas used by player 2 (joiner)

1. 1x joinGame (random or specific)
2. 1x amountEthDecision
3. 1x submitBoard
4. 63x shot
5. 61x shotResult miss
6. 2x shotResult hit

7.4 Total gas consumed

A practical evaluation made with a simulation led to a value of gas consumed equal to 10923121. This value is consistent with the average value presented with the previous calculations which is 240000+ 60000+ 36000+ 50000x2+ 31000x(64+63)+ 50500x(61+61)+ 54000x(2+2)+63500= 10813500.