Battleship on Ethereum

A Battleship DApp implemented
on the Ethereum blockchain

Pierfrancesco Bigliazzi

Peer-to-peer systems and blockchains
2022-2023

Summary

1 Smart contracts

The battleship DApp is an application that allows a user to create a battleship game from scratch and makes it joinable by other users or to join a previously created game in two different ways: random or by using the ID of the game created.
Two smart contracts were created to enable these functionalities:

- **GamesManager:** This contract acts as a sort of "central unit" that allows the users to manage different battleship games. It allows the creation of a new game and the partecipation to a previously created game randomly or by ID.
- **Game:** The contract that implements the actual game. It manages all the aspects related to the battleship game (placing the ships, shooting them, bet decisions, etc. . .).

Even if the overall project can be done with just one contract, the choice of using two has been made mainly for two reasons:

- **Readability:** with two contracts the management of the application is easier and less convoluted.
- **Separation of functionalities:** with this approach we avoid to accumulate excessive balance within a single contract and so mitigating possible attacks like re-entrancy attack.

Of course, this approach results in a higher cost in terms of gas since each new game represents a new contract that must be deployed on the blockchain.

1.1 GamesManager

The Games Manager smart contract allows a user to create a new game or to join a previously created game randomly or by sharing its ID.

Data structures used:

- **joinableGames:** data structure used to store all the games that have been created with all their parameters, including their IDs.
- **joinableGamesIndexes:** association between the games created and an array of indexes where each index refers to a specific game.
- **joinableGamesStatus:** array which stores the status of the games. A game can be joinable (true) or not joinable (false).
- **nonce:** random private number used for the generator of the game ID.

Functions used:

- **CreateGame:** this function creates a game and adds it to the list of available games.
- **JoinGameID:** when a user tries to join a game by ID, this function initially checks if the game is valid and if the user is not the owner of the game, then it registers him as the second player of the game.
- **JoinGameRandom:** this function is used to join a random pre-existing game. To join the game the function generates an index used to access to the list of joinable games. The computation of the index is performed with Keccak256 over three different values: the blockhash of the latest block, the address of the calling user and a nonce. The nonce is incremented each time the function is called. Also in this case, the function registers the user as second player.
- **_addGame:** internal function that insert the game in the list of the joinable games. It also set its status as joinable.
- **RemoveGame:** internal function called when a user has joined a game. It removes the game from the list of joinable games and delete its status.

Events emitted:

- **GameCreated:** this event is emitted when a new game has been created by a user. It stores in the transaction logs the address of the user who created the game (owner) and the ID of the game (_game).
- **JoinedGame:** when a user joins a game, both randomly and by ID, the contract emits this event. It stores the address of the player who joined the game (player) and the ID of the game (_game).
- **NoGameAvailable:** when a user tries to join a game randomly it can happen that all the games may be already occupied or no game has been created yet. This event stores the address of the player who tried to join the game (player).
- **NoValidGame:** if a user tries to join a game by ID he must insert an ID that corresponds to a game present in the list of joinable games otherwise this event is emitted and stores the address of the player who tried to join the game (player).

1.2 Game

The game smart contract implements the actual game that will be played by the users. It contains all the data structures and functions needed to implement the overall game logic.

Data structures used:

- **owner:** the actual creator of the game.
- **player1, player2:** the players of the game. player1 is always the owner of the game.
- **winner:** the address of the player who won the game will be stored here.
- **turn:** the turn is represented with the addresses of the players.
- **AFKplayer:** mapping between the address of the player and a boolean. If the player has been reported as AFK then this value is true otherwise is false.
- **BetsProposed:** bets proposed by the users.
- **PlayerPaid:** mapping between the address of the player and a boolean. If the player has fund his bet then this value is set to true otherwise is false.
- **BoardMerkleTreeRoot:** where it will be stored the root of the Merkle tree.
- **HittedShips:** mapping between the users and the number of ships that has been hitted during the game.
- **ShotTaken:** when a player takes a shot the index and the state will be stored here.
- **ShotsMap:** for each player it stores the index of a selected cell and if that cell has been already selected (true) or not (false).
- **bet:** amount on which both players agreed.
- **timeout:** number of the block used to implement the AFK mechanism.
- **currentPhase:** the game is divided in different phases:
    1. InitialPhase: phase where the owner needs to wait for a second player to join.
    2. BetPhase: phase where the players propose their bet and make an agreement about them.
    3. PlacementPhase: phase where each player place their ships on the board.
    4. ShootingPhase: phase where the game starts and each player fires a shot in turn.
    5. WinningPhase: phase where the game is over but the winner has to validate his board configuration before.
    6. EndPhase: phase where the winner is verified and can withdraw his prize.

Functions used:

- **constructor:** the constructor sets owner and player1 with the address of the user who created the game. The current phase at this moment is the initial phase.
- **RegisterSecondPlayer:** registers the second player and set the current phase to bet phase.
- **BetAgreement:** called when a player proposes a bet. The bet must be > 0.
- **BetAccepted:** called by the other player when he accepts the proposed bet. It sets bet to the agreed amount.
- **BetDeposited:** called when a player decides to fund his money. When both players call this function then the game goes into the placement phase.
- **ReportAFK:** sets a player as AFK. This function can be called only in specific phases of the game. The phases are: bet phase, placement phase, shooting phase and winning phase. It also sets the timeout to the block number + 5.
- **VerifyAFK:** validates a player as no more AFK and reset the timeout. Also this function can be called in specific phases of the game.
- **GetShotsTaken:** returns the shots of a player that have been set to taken. Used mainly for frontend purposes.
- **InitialCommit:** commits the initial board configuration of the player. It stores the Merkle tree root passed as argument. If both players have committed their boards the function moves on to the next phase and decides the starting player randomly.
- **FirstAttack:** starts the first attack of the first player. In this case there's no need to check the board configuration so it calls directly the next function.
- **Attack:** this function performs the actual attack. It updates the shots taken by the player and the cells selected by the player.
- **CounterAttack:** checks if the previous shot hitted a ship or not and start a new attack. First, the player has to validate his board in order to not cheat and send a wrong board configuration. Then, it updates the shot taken by the adversary by marking the previous shot as HIT or MISS. After that, the player can start its attack. If a player has hitted all the ships the function moves on to the winning phase.

- **CheckBoard:** checks the board configuration of the winner. Before a player can collect his prize he needs to validate his entire board configuration by checking all the leaves in the Merkle tree. After that, the function checks if the player has cheated in other ways, e.g. by providing false indexes or a wrong map of the selected cells during the game.
- **WinCondition:** if the winner has provided a correct board configuration and he has not cheated, then the game can go into the final phase.
- **Withdraw:** the player can collect his prize.
- **CheckProof:** checks the validity of a leaf in the Merkle tree.
- **CheckMultiProof:** checks the validity of all the leaves in the Merkle tree.
- **EncodeLeaf:** get the hash of a leaf and returns it.

Events emitted:

- **BetProposed:** informs that a player has proposed a bet. It stores the address of the proponent and the bet.
- **BetAgreed:** notifies the agreement between the players on a bet. It stores the addresses of the players and the bet.
- **BothPlayersPaid:** notifies that both players has funded their bet. It stores the addresses of the players.
- **BoardCommitted:** informs that a player has committed his board. It stores the address of the player and the root of the Merkle tree.
- **BothPlayersPlacedShips:** informs that both players have committed their board configuration. It stores the addresses of the players.
- **ShotTaken:** notifies that a player has taken a shot that needs to be validated. It stores the address of the player and the index of the selected cell.
- **Winner:** informs that a player has hitted all the opponent ships. It stores the address of the winner.
- **BoardChecked:** informs that the configuration of the board of the winner has been checked and it is valid. It stores the address of the winner.
- **WinnerVerified:** notifies that the winner has been verified and he can withdraw his prize. It stores the address of the winner.
- **AFKWarning:** notifies that a player has been reported as AFK. It stored the address of the AFK player.
- **NoMoreAFK:** notifies that a player has been validated as no more AFK. it stores the address of the player.

1.3 Design choices

The Battleship DApp is not an exact replica of the original game since it has some features and some changes that are not present in the original version.
The following design choices have been made during the development of the application:

- **Ship dimension:** The board is composed by 8x8 cells on which a player can place 5 ships. A ship can occupy only 1 cell. This choice has been made in order to keep the Merkle tree and the computation of the Merkle proofs as simple as possible. If we want ships with different shapes (e.g. 1X2, 1x3, 1x4) we need to add some information to the game and in particular to the ship representation. Each ship has to be represented as a triple: position of the ship, length and orientation on the board. This would mean to add more demanding computation of the Merkle tree and the proofs. Also, the management of the game changes significantly since that during the placement phase a player not only has to decide which cell will be occupied by a ship but also if the cell will be occupied by a part of a ship or by an entire ship. The game has to handle this possibility and also possible wrong placement of the ships (e.g. a 2x1 ship placed diagonally).

- **Attack and counterattack:** During the shooting phase a player has to do two things: take a shot and check if the previous opponent shot has hitted a ship on the board. These two actions can be done during a single transition instead of two. This would result in a reduction of the gas usage.
  The first attack has not a previous shot to check. Instead, all the other attacks are divided in two phases:

  1. **Check**: the player must verify the Merkle proof of the leaf that represents the cell selected by the opponent. In order to do that he must provide the three values used to compute the leaf of the tree (index, ship, salt), retrieve the leaf and verify it. After the verification the state of the shot is changed from Taken to Hit or Miss.
  2. **Attack**: after the check phase it starts the actual attack. The player provides the index of the cell that he wants to hit and add the shot to his Shot array.

- **Proof and Multiproof:** In order to build the tree and verify the proofs on the leaves the module @OpenZeppelin/MerkleProof has been used. In particular the two methods have been used:

  - **MerkleProof.verify**: Given the proof, the Merkle tree root and the leaf encoding it returns a boolean that is true when the proof is present inside the tree and false otherwise.
  - **MerkleProof.multiProofVerify**: Given the proofs, the proofs flags, the Merkle tree root and the leaves it returns a boolean that is true

when the proofs matches with all the values contained in the tree and false otherwise.

The `multiProofVerify` function is used when the player has to verify multiple proofs instead of calling the `verify` function on every single leaf. This process is optimized by `multiProofVerify` by performing a single traversal of the Merkle tree instead of multiple traversals like in the case of calling `verify` function each time.

- **AFK management:** One of the requirement of the project was to implement a AFK mechanism that reports a player as AFK and if he does not take a move until a timeout expires he loses by default.

```
1          modifier CheckAFK(){
2             if(AFKplayer[msg.sender] == true){
3                 if(block.number < timeout){
4                     AFKplayer[msg.sender] = false;
5                 }else if (block.number >= timeout){
6                     WinCondition(msg.sender == player1 ?
                          player2 : player1);
7                     return;
8                 }
9             }
10            _;
11        }
```

The modifier CheckAFK is applied to all the functions that can be called by a player accused of being AFK. This modifier sets AFKplayer to false if the method is called within the timeout limit, otherwise it declares the opponent as winner.

2 Frontend

During the development of the project the React framework along with the React Router v6 package for routing.
React is a javascript library that allows us to build user interfaces based on components. These components are modular and reusable.
React Router enables "client side routing". This feature allows the DApp to update the URL from a link click without making another request for another document from the server. This enables fast user experiences because the browser doesn't need to request an entirely new document or re-evaluate CSS and Javascript assets for the next page. Features used:

- **Router:** The `BrowserRouter` from React Router v6 has been defined in the `App.js` file. This router allows us to define routes in our application and associate to each route a specific component along with actions and loaders. By defining routes and associating them with components we can control which component should be rendered based on the current URL.
- **Components:** The actual component allows us to define the UI that will be displayed on the page. Each component is associated with a route and when the route changes the associated component is rendered.
- **Actions:** Action is a feature that allows to perform specific tasks when a route is matched and in particular it allows us to send the transactions on the blockchain.
- **Loaders:** Loader is a feature that allows us to handle asynchronous data loading for specific routes. By using loaders, we can fetch data before rendering the associated component in order to enhance the user experience.

2.1 Routing structure

Two main routes have been defined in the project: **Root** and **Game**. These main routes are divided into subroutes thanks to the use of the `children` property. Overview:

- **Root:**
  - Path: "/"
  - Elements: <AlertPopup/> and <Root/>
  - Error Element: <Error/>
  - Children:
    - "/home"
      - Element: <Home/>
      - Action: HomeAction
    - "/wait/:address"
      - Element: <Waiting/>
      - Loader: WaitingLoader
    - "/game/:address"
      - Element: <Game/>
      - Loader: GameLoader
      - Action: GameAction
      - Id: "game"
- **Game:**
  - Path: "/game/:address"
  - Element: <Game/>
  - Loader: GameLoader
  - Action: GameAction
  - Id: "game"
  - Children:
    - "/game/:address/bet"
      - Element: <Bet/>
      - Action: BetAction
    - "/game/:address/fund"
      - Element: <Fund/>
      - Action: FundAction
    - "/game/:address/place"
      - Element: <Place/>
      - Action: PlaceAction

* "/game/:address/shoot"
    - Element: <Shoot/>
    - Loader: ShootLoader
    - Action: ShootAction
* "/game/:address/win"
    - Element: <Win/>
    - Action: WinAction
* "/game/:address/end"
    - Element: <End/>
    - Action: EndAction

This structure was used in order to respect the logical flow of the smart contracts and to ensure an organized frontend.

3 Vulnerabilities

In this section are reported the potential vulnerabilities of this application and how some of the most common vulnerabilities that affect smart contracts are handled:

- **Re-entrancy Attacks:** Thanks to the use of `transfer` method, which is known to be hardcoded to prevent re-entrancy attacks, this kind of attack is not possible. Moreover, the balance after the call of `Withdraw` is 0 because it is entirely transferred to the winner.
- **AFK exploit:** While the usage of the AFK mechanism prevents a user from leaving a game pending, it can be abused to delay significantly the progress of the game. This is done by being reported and then wait until shortly before the timeout limit.
- **DoS attacks:** As reported above, an attacker can slow down the progress of the games by abusing of the AFK mechanism. Furthermore, to report an adversary has a cost so the opponent can be repeatedly being reported in order to make the user lose money.
- **Cross-site scripting:** One of the main advantages of React is that it uses JSX syntax which converts every potential harmful scripts or HTML tags into plain text and not executable code.

4 Gas Analysis

In the following table is reported the gas cost of the most relevant functions used in the smart contracts:

Function	**Cost**
CreateGame	2155317
JoinGameID	77606
JoinGameRandom	111222
BetAgreement	49505
BetAccepted	54830
BetDeposited	56950
InitialCommit	56658
FirstAttack	104122
CounterAttack	80296
CheckBoard	234191
Withdraw	32608
ReportAFK	75889
VerifyAFK	61157

Table 1: Cost of a full game on a 8x8 board

A few notes:

- The `CreateGame` function is the most expensive one. This was expected since it has to deploy an instance of `Game` smart contract on the blockchain.
- The cost difference between `FirstAttack` and `CounterAttack` is relatively small. Indeed the cost of `FirstAttack` is smaller than `CounterAttack`. Probably this due to the fact that the first attack has to instantiate some storage variables.
- The cost of `CheckBoard` depends a lot on how fast the game is. If a player hits all the ships on the first try then the cost will be much smaller than a game where a player misses all the shots.

To calculate the total cost of a game where each player misses all the shots we need to take into consideration that some methods will be called more than once. In our case `BetDeposited` and `InitialCommit` have to be called twice while `CounterAttack` has to be called 127 (64+63) times since it is the maximum number of instances that we can have.
The total gas cost is: **13,303,649**

5 User manual

Steps to run the application:

- Start Ganache application
- Navigate to **BattleshipProject** folder
- Run `truffle migrate` to deploy the **GamesManager** smart contract on the blockchain.
- Run `truffle test ./test/test_file.js` to start the test you want.
- Navigate to **frontend** folder.
- Navigate to **battleship** folder.
- Run `npm install` to install all the needed dependencies.
- Run `npm start` to start the application.
- Connect to localhost:3000 and start to play.

The game is run over two different browsers (I used Firefox and Chrome) that have two different accounts that were imported by Ganache. If the menu is not displayed on the page try to reconnect with Metamask and reload the page.

- Create a game with an account.
- Join the game by ID or randomly with the second account.
- Propose a bet with an account.
- Accept the proposal with the other account.
- Fund the ETH with both accounts.
- Place the ships with both accounts. All the ships must be placed.
- After the ships have been placed, commit the boards with both accounts.
- Start the game. The turn is displayed on the top right of the page.
- Perform the attack alternatively with both accounts.
- Hit all the ships of the opponent.
- Verify the board of the winner.
- Withdraw the prize.