Merketh Battleship

A Battleship DApp implementation on the Ethereum blockchain

Marco Ruta

UniPi - Peer to Peer e Blockchain - 2022/2023

Contents

1 Smart Contracts

In the development of the Battleship DApp, two essential smart contracts were created to enable the game's functionalities and ensure a seamless gaming experience.

- The first smart contract, **GamesManager**, serves as a central coordinator, managing different Battleship games and facilitating the creation of new games or joining pre-existing games randomly or by ID. This smart contract acts as a bridge between players and game instances, providing a secure and decentralized way to access a game.
- The second smart contract, **Game**, represents an individual Battleship game and incorporates all the different phases and mechanics of the game. From bet agreement to game termination, the Game contract handles the intricate game-play aspects enforcing the rules of battleship game.

The choice of using two distinct smart contracts serves multiple purposes. Firstly, it improves code readability and re-usability, the GamesManager contract offers a general purpose service and could handle different types of Games. Additionally, this approach helps avoid the accumulation of excessive balance within a single contract, mitigating the potential vulnerability to reentrancy attacks. However, this choice will certainly result in higher gas cost for playing a game because for each game a new contract must be deployed on the blockchain.

1.1 GamesManager

The GamesManager smart contract allows the players to create a new game or join randomly or by ID a pre-existing game. It stores a list of all the joinable games with their associated pseudo-random ID and the different events that can be emitted from this smart contract are **NewGame**, **JoinGame**, **NoGame**, and **GameNotValid** which are used to inform players about the result of their requests on the contract. This smart contract exposes three functions:

- **createGame(small)**: this function is used to create a new game, it uses the Game constructor and accepts the bool small as parameter. The small parameter is used to decide between playing a full size game (8x8) or a small size game (4x4). At the end of the function execution will be emitted a **NewGame** event which will contain the address of the game owner, the unique ID of the game, and the dimension of the game.
- **joinGameByID(game)**: this function is used to join a pre-existing game using as parameter the shared ID of the game. If the owner of the game tries to double-access a match or the provided game ID is not valid a **GameNotValid** event will be emitted, instead if the request is legit, the second player is registered using the Game `matchJoin` method, the game is removed from the joinable game list, and finally a **JoinGame** event is emitted, informing the players that the game is ready to start.
- **joinRandomGame()**: this function is used to join a random pre-existing game. If there is no suitable game a **NoGame** event is emitted. If a suitable game is found the second player is registered using the Game `matchJoin` method, the game is removed from the joinable game list, and finally a **JoinGame** event is emitted, informing that the game is ready to start.

1.1.1 Design Choices

The random index used to access the joinable game list is computed by performing Keccak-256 hashing over three values: the `blockhash` of the latest block, the `address` of the message sender, and a private `nonce` that is stored in the contract and updated every method invocation. However, it's important to note that this solution is not truly random due to the blockhash usage, whose the value can be influenced, to some extent, by miners. This introduces predictability, as miners may manipulate the blockhash to affect the computed index. Considering the limitations and potential vulnerabilities of this pseudo-random approach, using an `external oracle` (e.g. provable for increased randomness might be considered in critical applications. However, for the current use case of accessing a random game from the list, the added complexity of an external oracle might outweigh the benefits.

1.2 Game

The Game smart contract allows the player to actually play a battleship game, this contract encapsulates the overall game logic and state by storing all the needed information and exposing all the needed functions.

```
struct Player {
    bool afk;
    uint256 bet;
    bool hasPaid;
    bytes32 board;
    uint8 hits;
    Shot[] shots;
    mapping(uint8 => bool) shotsMap;
}
```

The **Player** struct stores all the information about a player: the `afk` status, the last `bet` proposed by the player, the `payment status` of the player, the `board` of the player, and a `list` and a `map` of shots taken by the player. The shots list and the shotsMap mapping have two different purposes: the first one is to store all the information about a taken shot while the second one is used for fast lookup of the shots taken by the player so far.

```
enum ShotState {
    Taken,
    Hit,
    Miss
}

struct Shot {
    uint8 index;
    ShotState state;
}
```

The **Shot** structure stores all the information about a shot: the `index` and the `shotState`. A shot can be in three different states: Taken if the shot was taken but not yet confirmed by the opponent, Hit and Miss if it was confirmed by the opponent to have or have not hit a ship of the opponent's fleet.

```
    // Addresses of the two Players.
    address public owner;
    address public adversary;

    // Mapping used to store all the information about the Players.
    mapping(address => Player) public players;

    // Final agreement on the bet amount.
    uint256 public bet;

    // Current turn.
    address public turn;

    // Current game phase
    Phase public gamePhase;

    // Winner of the game.
    address public winner;
```

The overall status of the game is encapsulated in these seven variables, which store all the needed data to properly manage a battleship game.

1.2.1 Game Phases

The fact of having different game phases makes the game more structured and allows to define **modifiers** that are used to check if a function is called in a legit game phase. Several other modifiers are defined to check if a function is called by a legit actor (e.g. OnlyPlayer checks that the function is called from the owner or the adversary). There are six different game phases:

- **Waiting**: the owner of the game is waiting for another player to join. In this phase only the method `matchJoin` can be used to register the second player.
- **Betting**: the two players make proposals on the bet until an agreement is reached and they can fund the agreed amount. In this phase only the methods `proposeBet` to propose a bet, `acceptBet` to accept the last opponent bet, and `betFunds` to fund the agreed ETH amount can be used.
- **Placement**: the two players place their fleet and commit their board. In this phase only the `commitBoard` method can be used.
- **Attack**: the two players shoot each other boards. In this phase only the methods `attack` to perform the first attack of the game and `counterattack` to confirm the opponent's previous attack and attack. These methods can be used by the players only in their turns.
- **Winner**: one of the two players wins and has to validate his board. In this phase only the method `checkWinnerBoard` can be used by the winner to validate his board.
- **End**: the only method available in this phase is the `withdraw` method that can be used only by the winner.

1.2.2 Game Events

This smart contract can emit nine different events that are used to inform the interested actors:

- **BetProposal**: informs that one player has proposed a bet on the queue.
- **BetAgreed**: informs that the two players agreed on the bet.
- **FundDeposited**: informs that the two players deposited the funds.
- **BoardsCommitted**: informs that the two players committed their board.
- **ShotTaken**: informs that a player took a shot.
- **Winner**: informs that the game has a winner (must be verified).
- **WinnerVerified**: informs that the victory was verified.
- **PlayerAFK**: informs that a player was reported as Away From Keyboard.
- **PlayerMove**: informs that a player previously reported as AFK made a move.

1.2.3 AFK management

One of the requirements of the project was to implement an AFK reporting system where a player is declared as AFK (and loses by default) if he does not take a move from the moment he is reported as AFK to the end of a timeout.

```
    modifier afkAllowed() {
        address opponent = msg.sender == owner ? adversary : owner;
        require(
            (gamePhase == Phase.Betting &&
                players[opponent].hasPaid == false &&
                players[opponent].bet == 0) ||
                (gamePhase == Phase.Placement &&
                    players[opponent].board == 0) ||
                (gamePhase == Phase.Attack && turn == opponent) ||
                (gamePhase == Phase.Winner && winner == opponent)
        );
        _;
    }

    modifier stillAFK(){
        if (players[msg.sender].afk) {
        // The player has done an action before the timeout -> No more AFK
        if (block.number < afk_timeout) {
            players[msg.sender].afk = false;
        } else if (block.number >= afk_timeout) {
            // The action made a move after the timeout -> AFK verified
            _declareWinner(msg.sender == owner ? adversary : owner);
            return;
        }
    }
    _;
}
```

The AFKAllowed modifier is applied to the **reportAFK** and **verifyAFK** methods and enforces that these methods are used only when the opponent can actually take a move. The opponent can take a move only:

- in the betting phase only if he has not funded the bet yet.
- in the placing phase only if he has not committed his board yet.
- in the attacking phase only if it is his turn to attack.
- in the winner phase only if he is the winner.

The stillAFK modifier is applied to all the methods that can be called by a player accused of being AFK. This modifier unsets the AFK status of the player if the invoked method is called within the timeout limit, otherwise it declares the opponent as winner.

```
    function reportAFK() external onlyPlayer afkAllowed {
        address opponent = msg.sender == owner ? adversary : owner;

        require(!players[opponent].afk);

        emit PlayerAFK(opponent);

        players[opponent].afk = true;
        afk_timeout = block.number + 5;
    }
```

The **reportAFK** method checks if the opponent is not reported as AFK yet, if so the opponent is set as AFK, the timeout is set to `block.number + 5`, and a **PlayerAFK** event is emitted.

```
    function verifyAFK() external onlyPlayer afkAllowed {
        address opponent = msg.sender == owner ? adversary : owner;

        if (players[opponent].afk && block.number >= afk_timeout) {
            _declareWinner(msg.sender);
        } else {
            players[opponent].afk = false;
            emit PlayerMove(opponent);
        }
    }
```

The **verifyAFK** method checks if a player (previously reported as AFK) made a move within the timeout limit. If so, the player is set as not AFK (to prevent verifyAFK method spamming) and a **PlayerMove** event is emitted. If he did not take a move the opponent is automatically declared as winner of the game.

1.2.4 Design Choices

- **board committing**: when a player commits his board he only must provide a bytes32 value that is the `root` of the merkle tree representing the board. In this phase there is no check on the board content and on the smart contract only the root of the tree is stored.
- **attack and counterattack**: the idea is to perform the check of the previous shot and the attack in a single transaction to reduce the gas usage. The first attack has no check phase because there is no shot to check and is performed with the `attack` method. All the other attacks will be performed with the `counterattack` method that is divided in two phases:
  - **check**: the player must provide the leaf representing the cell targeted by the opponent along all the values that where used to compute the merkle proof in the placing phase (salt, index, ship). In this way, on the smart contract side, the proof for the leaf is verified and the shot state is updated from Taken to Hit or Miss.
  - **attack**: the player provides the index of his next attack and append a Shot to his shots array. The shots will be verified in the next opponents attack.

  During the check phase, if an opponent hit a ship his hits status is increased and when a player reaches an hit status equal to the fleet he is declared as winner.
- **board checking**: when a player is declared as winner he must provide his board for verification. The player sends the data of his board (all the leaves of the MerkleTree) so that on the smart contract side, the multiproof for all the leaves is verified. After verifying the cells of the board the number of ships placed is computed and if it is valid the victory is confirmed.
- **Forfeit and AFK:** if a player decides to forfeit the game or is declared AFK, the opponent is automatically declared the winner without the needs to check the opponent's board. This mechanic may inadvertently lead frustrated players to forfeit or leave the game prematurely, especially if they have not made any successful hits with their shots. In such cases, the frustrated player may assume that the opponent has placed their entire fleet on the board and believe they have no chance of winning. However, it's important to consider that players may be aware that the presence of the opponent's fleet is checked only at the end of the game. This knowledge might encourage players to stay in the game since they know that if their opponent cheated by not placing the entire fleet, the cheater will be caught, and they will be declared the rightful winner.
- **proof and multiproof**: the module used to integrate the merkle trees in the project is @openzeppelin/merkle-tree. This module allows us to create Merkle Trees and verify proofs on the leavs. Two different methods of the module where used to verify proofs:
  - **StandardMerkleTree.verify**: returns a boolean that is true when the proof verifies that the value is contained in the tree given only the proof, merkle root, and leaf encoding. This method is used to verify the shot taken by the opponent in the check phase of the counterattack method.
  - **StandardMerkleTree.verifyMultiProof**: Returns a boolean that is true when the multiproof verifies that all the values are contained in the tree given only the multiproof, merkle root, and leaf encoding. This method is used to verify the whole board of the winner in the checkWinnerBoard method.

When verifying multiple proofs for a Merkle tree, it is generally more efficient to use the verifyMultiProof function instead of calling verify on each individual leaf. The verifyMultiProof function is specifically designed to handle multiple proofs efficiently: when verify is called individually on each leaf the algorithm traverses the Merkle tree from the root to the respective leaf for each proof. The verifyMultiProof optimizes this process by performing a single traversal of the Merkle tree, reusing intermediate nodes that are shared among the proofs.

- **board and ship dimension**: the game can have only two possible configurations: the standard game is played on a 8x8 board with a fleet of 10 ships and the small game is played on a 5x5 board with a fleet of 5 ships. There is no constraint on the ship size: all the ships of the fleet are represented by a 1x1 cell. This choice was made to let the proof on the merkle tree be as simple as possible: if we want to introduce fixed ship shapes (e.g. carrier 3x1) we would add to each leaf indication about which kind of ship occupies the cell and what is the ship orientation. Adding this information to the leaf will result in having a more demanding computation of the merkle trees and the proofs. In addition the contract has reached its maximum size limit, as imposed by the Ethereum mainnet. This limitation makes it challenging to add any further complexity to the contract, such as incorporating fixed ship shapes and orientation. As a result, any attempt to add checks for the board configuration would require a significant restructuring of the entire codebase.

2 Frontend

In the development of the Battleship DApp frontend the React framework along with the React Router v6 package for routing were used. React Router v6 served as an abstraction of the MVC pattern in the following ways:

- **Routing as a Controller:** In MVC, the `controller` handles application logic, including navigation and user action handling. React Router v6's `routing` components fulfill a similar role. By defining routes and associating them with components, we can control which component should be rendered based on the current URL. This allows us to manage navigation, perform actions, and load data based on route changes.
- **Components as Views:** in MVC, the `view` represents the user interface. In React Router v6, React `components` act as views. Each route can be associated with a specific component that represents the view for that route. When the route changes, the associated component is rendered, providing the desired UI to the user.
- **Contract state as Model:** In the context of the Battleship DApp, the `smart contract` state serves as the `model` in the MVC pattern. The model represents the data and state of the application. With the help of the web3 package, we can access the contract state, interact with the blockchain, and retrieve or listen for information.
- **Actions and Loaders:** although not directly tied to the MVC pattern, React Router v6 introduces the concepts of `actions` and `loaders`. Actions allow us to perform specific tasks when a route is matched. Loaders, on the other hand, facilitate asynchronous data loading for specific routes. By utilizing loaders, we can fetch data before rendering the associated view, enhancing user experience and separating concerns related to data retrieval from the view components.

By leveraging React Router v6 for routing, utilizing React components as views, incorporating actions and loaders, and accessing the contract state as the model, we were able to build an architecture that aligns with the MVC pattern.

2.1 Routing Structure

The core of the routing strategies in our Battleship DApp is implemented using the `BrowserRouter` from React Router v6 in the `App.js` file. This `BrowserRouter` acts as a controller, allowing us to define various components, loaders, and actions for each specific route in our application. The route hierarchy in the `BrowserRouter` adheres to the logical structure adopted in the backend, effectively organizing the frontend views. Two main routes, namely `Root` and `Game`, are defined to represent the **GamesManager** and the basic view of a specific **Game**, respectively. These main routes are further divided into nested routes using the `children` property to handle more specific phases and scenarios within the game.

Here's an overview of the router's structure:

- **Root Route:**
  - Path: "/"
  - Elements: `<AlertPopup />` and `<Root />`
  - Error Element: `<Error />`
  - Children Routes:
    - "/home": Renders the `<Home />` component and executes the `homeAction`.
    - "/wait/:address": Renders the `<Waiting />` component while using the `waitingLoader` to handle asynchronous data loading.
- **Game Route:**
  - Path: "/game/:address"
  - Element: `<Game />`
  - Loader: `gameLoader`
  - Action: `gameAction`
  - ID: "game"
  - Children Routes:
    - "/game/:address/bet": Renders the `<Betting />` component and executes the `bettingAction`.
    - "/game/:address/placing": Renders the `<Placing />` component and executes the `placingAction`.
    - "/game/:address/attacking": Renders the `<Attacking />` component and executes the `attackingAction`, also utilizing the `attackingLoader` for data loading.
    - "/game/:address/end": Renders the `<End />` component and executes the `endAction`.
    - "/game/:address/withdraw": Renders the `<Withdraw />` component and executes the `withdrawAction`.

By structuring our router in this way, we ensure that the frontend closely aligns with the backend's logical flow. Each route is associated with the relevant components, actions, and loaders, providing a smooth user experience and enabling efficient handling of asynchronous data retrieval. The use of nested routes facilitates a clear separation of concerns and allows for the creation of a well-structured and organized frontend.

2.1.1 MUI Components

All the components of the Battleship DApp are created using the Material-UI (MUI) packages, which provide a robust set of UI components and styling utilities. For customization purposes, we have a dedicated `/customTheme` directory where all the custom components are declared and exported. To ensure consistent theming and styling throughout the application, we wrap the entire application in a **ThemeProvider**. This ThemeProvider utilizes the MUI custom theme to apply a cohesive color palette and styling to all components.

Moreover, we employ an **AlertProvider**, which is declared and exported in the contexts/Alert-Provider directory. The AlertProvider serves as a context provider for generating global alerts at the application level. This allows us to display important messages or notifications to users, enhancing user communication and overall user experience.

Furthermore, within each component, we implement several **useEffect** listeners that are initiated during component mounting and removed during unmounting. These listeners are responsible for monitoring various blockchain events that are relevant to the specific component's functionalities. The components interact with the Ethereum blockchain using **actions** to call methods on the smart contract. These actions are triggered by user interactions within the application.

To optimize data loading and avoid redundant requests, some components utilize their own dedicated loader, while others leverage the route loader of one of their parent components. For example, all children components of the Game component can utilize the gameLoader, which fetches all the game-related data from the blockchain.

By combining passive blockchain interaction through listeners and useEffect and active interaction through actions, our components seamlessly interact with the Ethereum blockchain, allowing users to engage in Battleship games with real-time updates and accurate information.

3 Gas Analysis

In the following table is reported the gas cost of each method exposed by the contracts. The gas cost for each transaction is automatically retrieved by the `test_maxGas.js` test.

Method	**Gas Cost**
createGame	3,762,591
checkWinnerBoard	350,372
joinRandomGame	102,692
counterattack	82,633
reportAFK	76,917
firstAttack	79,231
joinGamebyID	79,078
acceptBet	56,802
forfeit	56,916
commitBoard	53,691
proposeBet	52,421
withdraw	32,927
verifyAFK	34,850

From the table we can notice some important aspects:

- The **createGame** method incurs the highest gas cost, which was expected. This is because it deploys a new contract instance of the Game on the blockchain. Deploying a smart contract involves a substantial amount of computation and storage, making it more expensive compared to other methods.
- The gas cost difference between the **firstAttack** and **counterAttack** methods is relatively small. This suggests that the check phase of the counterAttack method, which verifies the validity of the attack, does not significantly impact the overall cost. The primary cost driver in both methods is likely the execution of the attack itself, rather than the subsequent checks.
- The **checkWinnerBoard** method ranks as the second most expensive method. One potential optimization to reduce the gas cost could involve checking only the cells that have not been verified yet during the attack phase. However, the decision to check the entire table in checkWinnerBoard allows for more complex checks over the ships in the future. It might be a trade-off between gas cost and flexibility in future updates.

We can estimate the total cost for both standard and small games by considering gas costs and multiplying method calls. For each game, we multiply proposeBet, depositFund, and commitBoard by two. In a standard game, we multiply counterattack by 62, and in a small game, we multiply it by 14. By summing up these gas costs, we obtain the overall estimated expenses for each game considering the maximum number of shots possible (this process is automatized by the `gas_analysis.py` script).

Game	**Gas Cost**
Small	10,200,902
Standard	6,234,518

4 Vulnerabilities

In the following section are reported which are some of the possible vulnerabilities that can affect the overall project and how some vulnerabilities where addressed during the development.

- **Re-entrancy attacks**: this vulnerability is patched on two different level. The withdraw method (the only payable method) uses the `transfer` method which should be safe, morehover, the contract balance, due to the existance of a contract for each single game, is exactly the game prize, so any successive withdraw would result in a 0 ETH withdraw.
- **AFK abusing**: while the presence of the two methods reportAFK and verifyAFK and the two modifiers AFKallowed and stillAFK ensure that a player can be reported only when he actually is not taking any action there still is an issue: if a player knows the 5 blocks rule he could report an opponent as AFK, make some other transaction unrelated with the game (e.g. creating new games) and after verify the AFK status of the opponent. In this case the player who reported can "manipulate" the blockchain to make the opponent appear as AFK faster than usual.
- **Custom clients**: the developed client uses cryptographically secure random number as salts, generated by exploiting the browser capabilities. An attacker could spread a sutom client copy that uses weak salts (or worse, not random salts) and this could lead to potential brute force attacks with aim to discover the fleet placement of the opponent.
- **Weak randomness**: while using the blockhash is better than using the block timestamp for pseudo random number generation this approach is still not perfect (as discussed in the GameManager secition). A better solution could be using an external oracle but it still is a matter of trade-offs.
- **Arithmetic overflow**: to avoid arithmetic overflows and underflows the SafeMath library of openzeppelin is used for any uint8 and uint256 arithmetic operation.
- **Access control**: all the methods, by using the modifiers, can only be called in legit game phases and by game actors. This should prevent some vulnerabilities related to access control problematics.
- **React and Cross-Site Scripting (XSS) Mitigation**: one of the most significant concerns in web development is the risk of Cross-Site Scripting attacks, where malicious scripts are injected into a web application and executed in the context of unsuspecting users' browsers. React solves XSS automatically through JSX escaping, which treats dynamic content as plain text and not executable code.
- **DoS**: to address potential DoS issues in the system, a measure can be implemented to limit the number of games a player can create and leave in the waiting phase. By setting a reasonable threshold on player game ownership, it ensures that a player cannot flood the system with an excessive number of unplayable games, which would otherwise disrupt the random game joining process for other players.

5 Usage

These are the general steps to run the application:

- Start `Ganache cli` or `Ganache application`. If using the ganache cli we can run `ganache -d` to generate the accounts and the keys linked to the blockchain in a deterministic way.
- Navigate to /backend
- Run `npm install` to install all the needed dependencies.
- Run `truffle test` to start all the tests and check that everything works as expected.
- Run `truffle migrate` to deploy the GamesManager smart contract on the blockchain.
- Navigate to ../frontend
- 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 demo is run over two different browser (i'm using Firefox and Opera) that have two different Metamask accounts.

- Start `Ganache cli` or `Ganache application`.
- Navigate to /backend
- Run `truffle migrate` to deploy the GamesManager smart contract on the blockchain.
- Navigate to ../frontend
- Run `npm start` to start the application.
- Import two different accounts in the two browser via Metamask.
- Create a small game (or a standard one if you have time) from the first account.
- Join the game by ID or join a random game from the second account.
- Propose a bet with one of the account.
- Accept or propose a new bet with the other account.
- Fund the ETH from both the accounts.
- Place the fleet from both the account and commit the board.
- Start the attacks from the second account and perform attacks in alternate way.
- Hit all the ships of the opponent or end the game by forfeit/AFK.
- Verify the winner board (not needed for forfeit or AFK)
- Withdraw the ETH!