BattleShip game on Ethereum blockchain

Peer-to-peer and blockchain,
(2022-2023)

Aleandro Prudenzano - 603676

Contents

1 | Design

In order to implement the Battle ship game on the Ethereum blockchain the first step is to correctly understand the steps of the game:

- the two players must match one each other, whether randomly or sharing the `game_id`;
- players have to negotiate a common fee for the game;
- players have to build their boards by placing the ships, computing a Merkle tree and publish it on the chain;
- actually play the game by proposing a move and waiting for the opponent to reveal the single cell state;
- then when one of the two players turns out to be the winner, both players have to reveal the whole board in order to let the logic check whether the board is correctly formed or not.

1.1 | Finite state machine

Looking at the game steps it seems natural to design the smart contract as a finite state machine (FSM from now on) referencing the same states described in the previous analysis.
An enum is used to trace the game state:

```
enum GameState {
    // Game created but waiting for the second player to join
    WaitingForOpponent,
    // Players joined the game but need to agree on the fee
    FeeNegotiation,
    // Players agreed on the fee and need to commit the board
    WaitingCommitment,
    // Players committed the board and now the game starts
    WaitingForMove,
    // Players must submit Merkle proof in order to prove the move result
    WaitingForProof,
    // Game ended and now player must reveal the board
    WaitingForReveal,
    // Game ended and now we have a winner
    GameOver
}
```

This way the code can be written in a much easier and robust way because each transition function will be structured as follows:

- check if the game is in the correct state;
- check if data passed by the user are correct
- update internal data
- check if conditions for the transition to the next state are satisfied, and in that case move to it.

1.1.1 | Actual FSM

The actual FSM of the contract follows the following graphic representation:

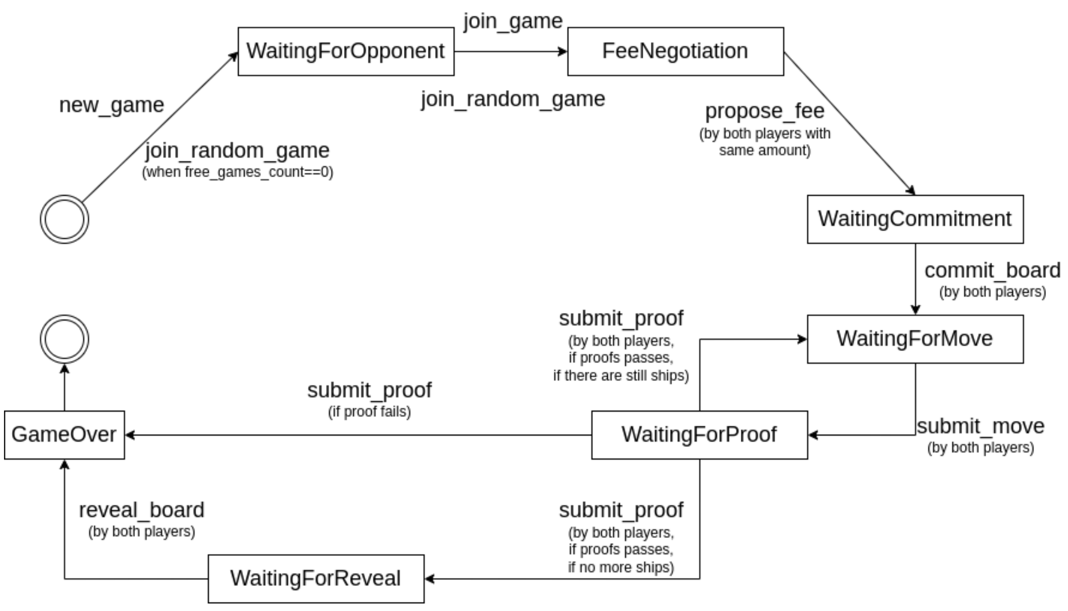


Figure 1.1: Contract finite state machine

1.2 | Game data

Game data are collected inside a `Game` struct and all of them are stored in a global private *mapping*, in order to be indexed only via the game ID:

```
struct Game {
    uint256 game_id;
    GameState game_state;
    Player A;
    Player B;
    uint256 position_in_games_list;
    uint8 rounds_elapsed;
    address winner;
}

// Collects actual games data, used for the game evolution
mapping(uint256 => Game) private id2game;
```

1.3 | Matchmaking

The matchmaking is designed in order to allow both join knowing the game id and randomly, moreover it is designed in order to make a match in constant time, minimizing the transaction fees.
This is obtained by building a list of free games from which we can randomly pick one, and remove from it, always in constant time:

```
// Collects games which are pending for players to join
// used for random matchmaking and game join
uint256[] private free_games;
uint256 private free_games_count;
```

1.4 | Leave accusation

In order to prevent funds locked in the game when one of the two players leaves the game, an accusation mechanism is implemented.
During the phases `WaitingForMove`, `WaitingForProof`, `WaitingForReveal`, the user who already made his action can call the `make_accusation` method of the contract. This method will move the opponent in the `accused` phase, now:

- the accuser can call `check_accusation` in order to check if 5 blocks are elapsed from the accusation one, and in this case move the game to the `GameOver` phase;
- the accused can make his action in order to retrieve the accusation and continue playing.

1.5 | Frontend

A webapp has been developed in order to properly interact with the on-chain contract. The web application is completely client-side and developed in Svelte, a frontend framework that helps a lot during the development. It also allows the use of svelte-web3 a svelte-style wrapper around `web3.js`.
It uses the metamask browser extension in order to interact with the blockchain, moreover listens for the contract events to synchronize the render of the match.

2 | Demo

2.1 | Deploy

In order for the demo to work you need to have:

- a working docker installation;
- `docker-compose` installed;
- `ganache` and `truffle` installed.

First you need to create a new project in ganache, then you have to deploy the contract on ganache, in order to do it you have to edit the `truffle-config.js` in the `battleship` folder of the attached project with:

```
development: {
 host: "127.0.0.1",      // Localhost (default: none)
 port: 7545,             // Standard Ethereum port (default: none)
 network_id: "*",        // Any network (default: none)
},
```

and changing the port value accordingly with your ganache installation. In order to actually deploy the contract you have to run the command `truffle migrate` while in the `battleship` folder and copy the contract address from the output of the command:

```
  ----------------------
 > transaction hash:    0x61e6885ff42555...acb5d04b8d84335
 > Blocks: 0            Seconds: 0
 > contract address:    0x3849c1588CFB9a93fbCc5a4253a39f2611D2F620
 > block number:        710
 > block timestamp:     1704149487
 > account:             0x6259FFb74191102A24A5a4b96cCAF2Fd93D35b52
 > balance:             99.769690023614298805
 > gas used:            4486857 (0x4476c9)
 > gas price:           2.500000008 gwei
 > value sent:          0 ETH
 > total cost:          0.011217142535894856 ETH
```

Then paste the contract address in the file `frontend/src/lib/constants.js` instead of the current value of `CONTRACT_ADDRESS`, this is needed because the frontend has the contract address hardcoded in order to communicate with the contract.
In the end, in order to run the frontend and actually play the game you have to run the command `docker-compose up --build` while in the root folder of the project (the same folder which contains `docker-compose.yml`).
Once the building process is done you can navigate your browser to http://localhost:8000/ to play the game.

2.2 | Game instructions

2.2.1 | Browser and metamask setup

To actually play a demo game you have to open two different browsers (or in alternative you can use two chrome/chromium windows from different profiles), then you have to install metamask and setup different accounts in those two windows. If the application says `Application not yet connected`, then check that metamask is correctly configured and you logged in it and then reload the browser page.

2.2.2 | Game start

To start a new game click on `Create new game` and accept the transaction. Now you will be provided with a game ID that you can copy and share to the other player. In alternative the other player can click on `Join game`, since there should be only one free game.

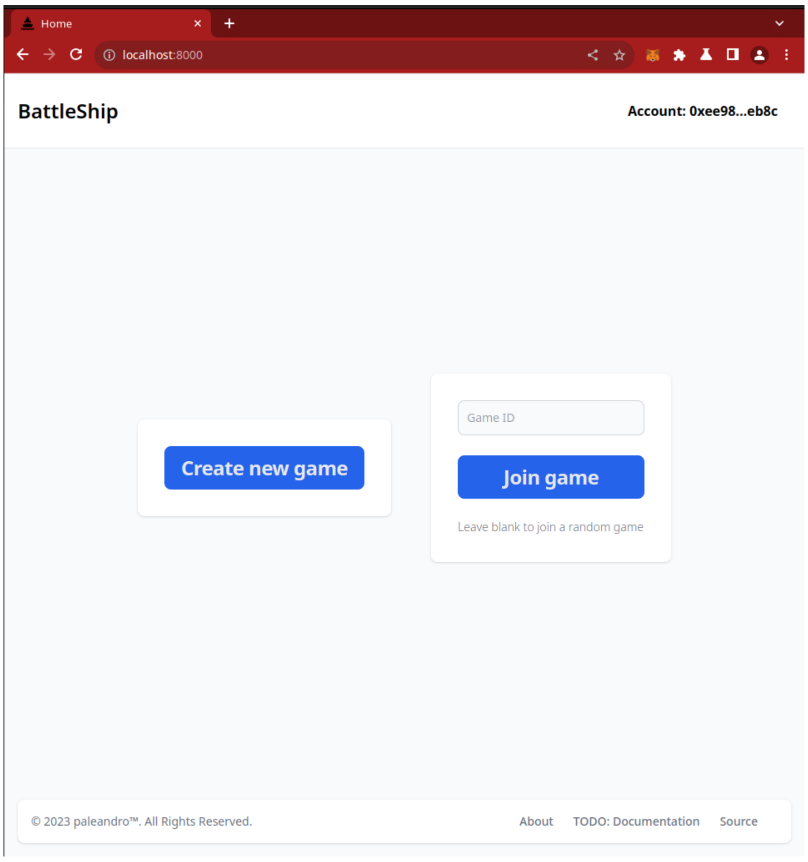


Figure 2.1: Starting menu

2.2.3 | Fee negotiation

Now the two players needs to negotiate a common fee to bet on. You can type the value in *gwei* inside the input box, otherwise you can click on one of the buttons to set the scale of the input box, and then increase or decrease the value with the arrows:

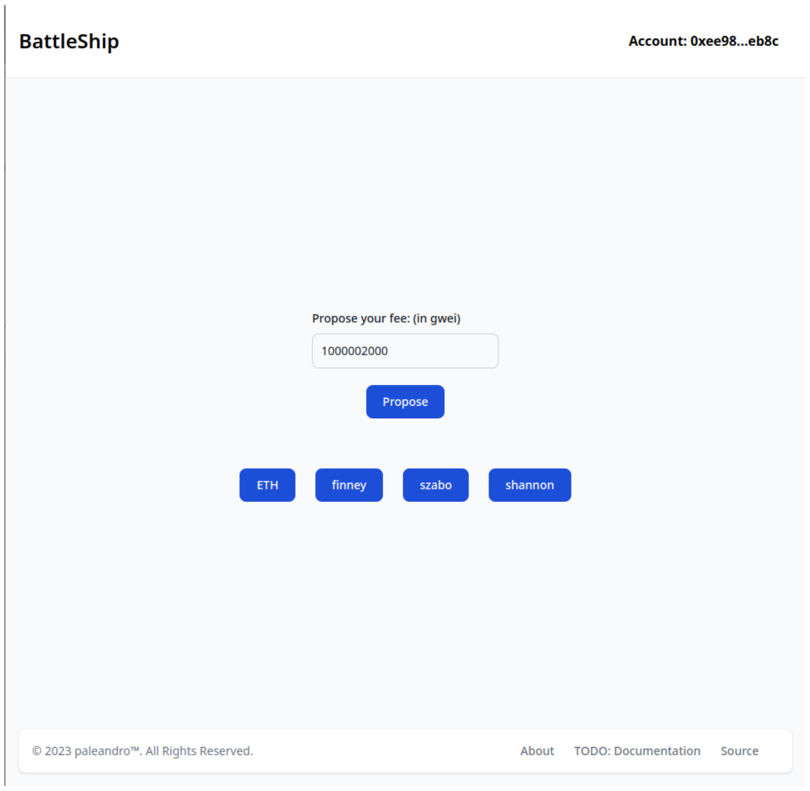
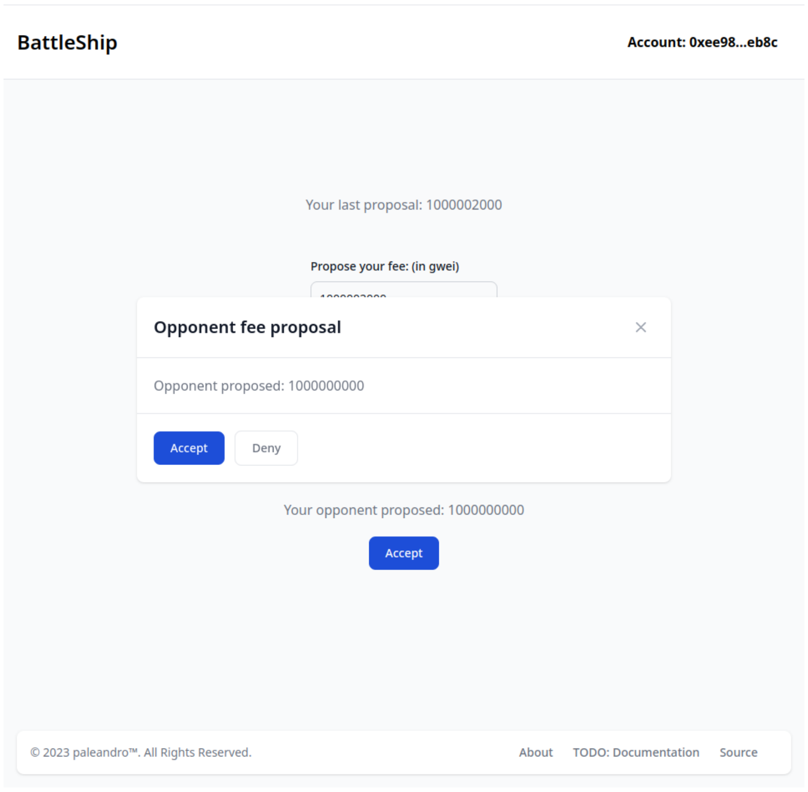


Figure 2.2: Fee negotiation menu

Then by clicking on `Propose` and signing the transaction in metamask you can propose that value as game fee.
Now the other user will receive a notification in order to decide to accept the opponent proposal, or in alternative, make his own proposal:



BattleShip Account: 0xee98...eb8c

Your last proposal: 1000002000

Propose your fee: (in gwei)

Opponent fee proposal

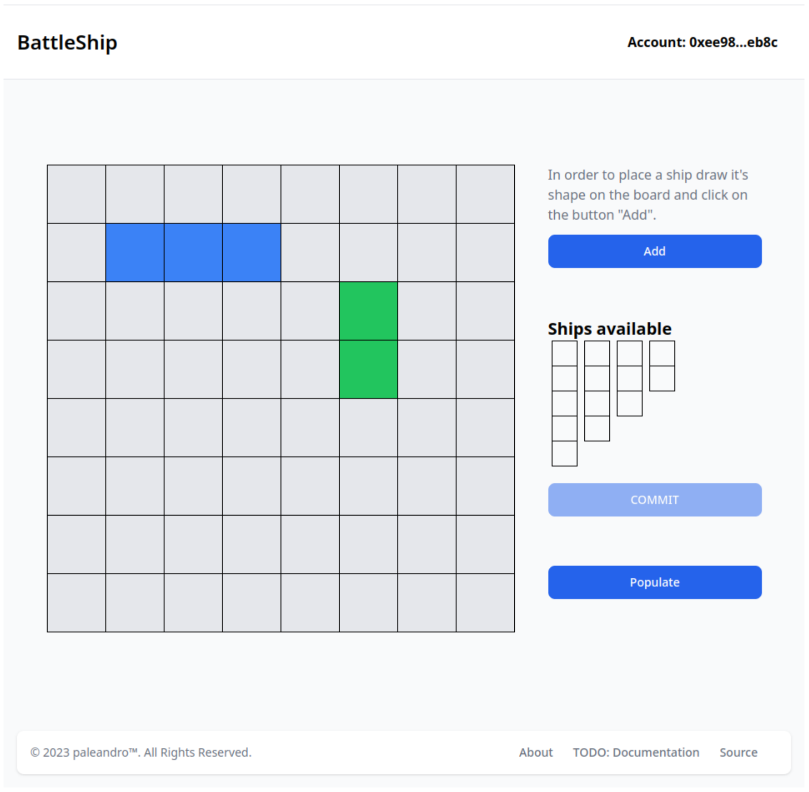
Opponent proposed: 1000000000

Your opponent proposed: 1000000000

© 2023 paleandro™. All Rights Reserved. About TODO: Documentation Source

Figure 2.3: Fee proposal

Once both players agreed on common fee the game moves on to the board building phase. In order to place a ship the user has to draw it with sibling cells, then click on the button `add` which will enable only on valid configuration. In the end, once all the ships have been placed, the button `commit` will bright up and the user can now publish it's board. The same transaction which publishes the board is the one in which the user pays the fee of the game to the contract.
For demo purposes there is also a `Populate` button which sets the ship in the top left corner avoiding wasting time during demo and various tests.



In order to place a ship draw it's shape on the board and click on the button "Add".

© 2023 paleandro™. All Rights Reserved.

Figure 2.4: Ship placement

Once the board is committed by both the players the actual game can start. In order to make a bet the user has to click on the chosen cell in the opponent board, which will be all grey at the beginning. Once the move has been committed on the blockchain the opponent will see the cell marked in black in it's own board, while the current player will see it black in the opponent one.

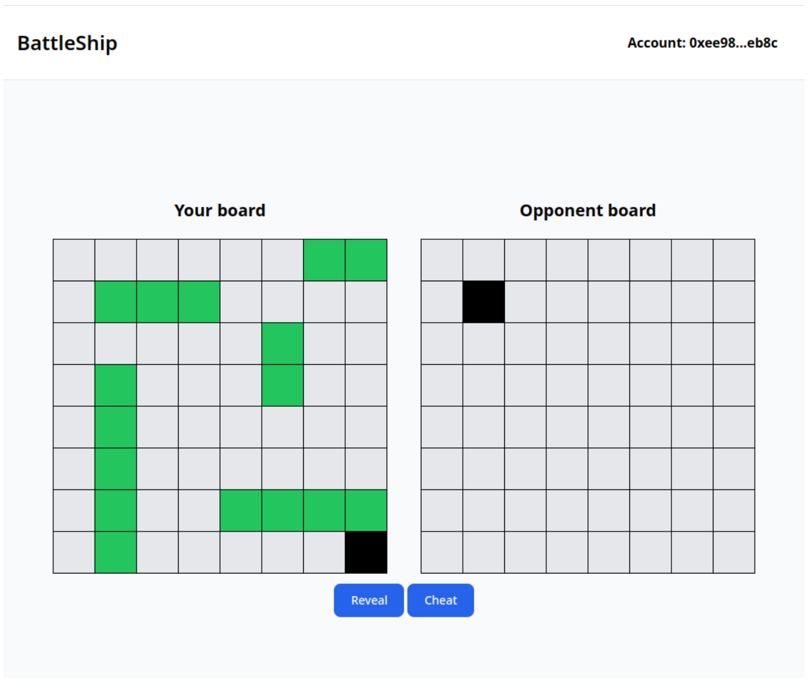


Figure 2.5: Move disposition

Now the users have to reveal their moves, for the sake of demo each player can both reveal it's cell for real, or cheat (by providing a fake nonce in the reveal phase). By clicking the reveal button the frontend

sends the status of the opponent chosen cell, the nonce for that cell and the merkle proof, in order for the smart contract to check it.
Once you reveal your cell the graphic will update the status of that cell for both you and your opponent, showing a blue cell in case of miss (you know, the water) and a red cell in the case of a hit (actual fire).

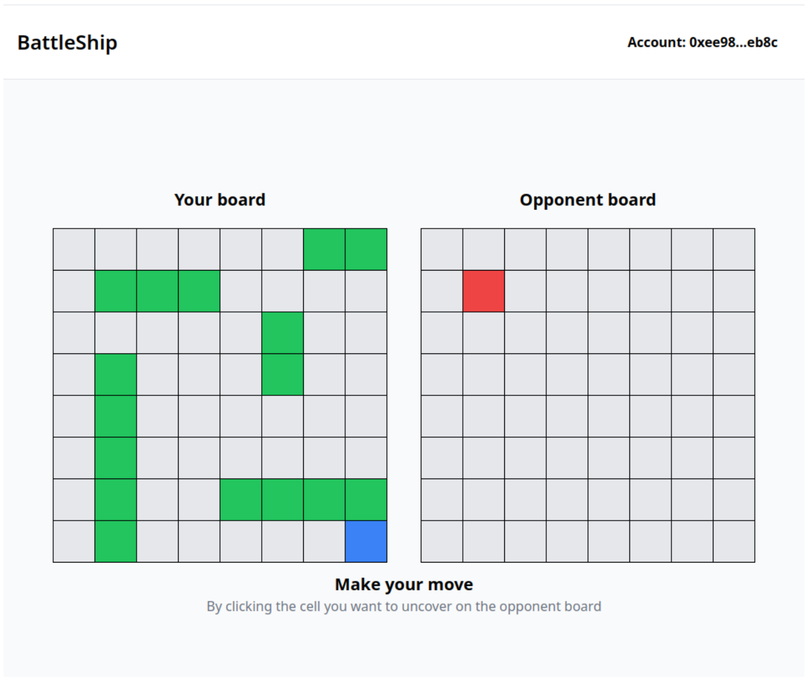


Figure 2.6: Cells revealed and board updated

If you already made your move, but the opponent is not responding, you can accuse it of leaving by clicking the `Accuse of leaving` button at the bottom:

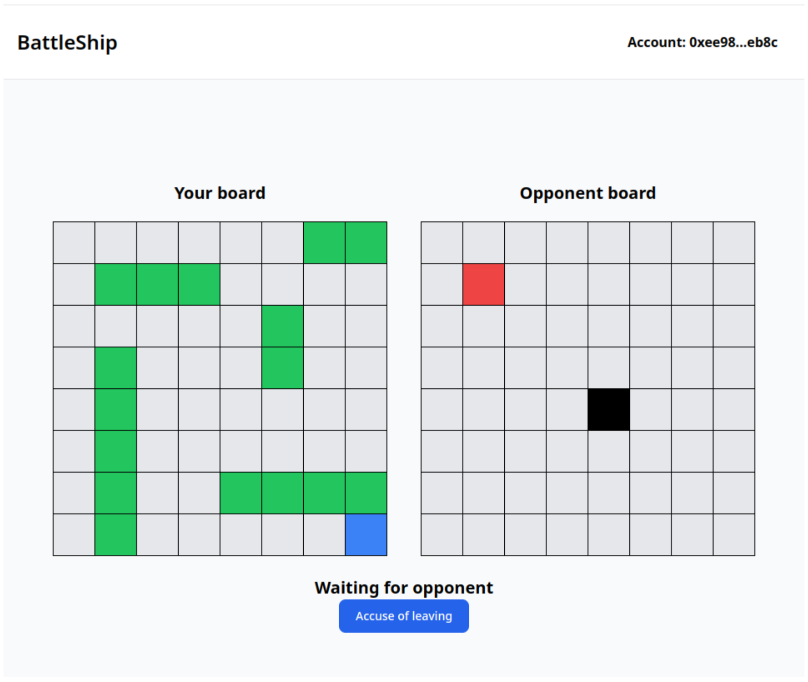
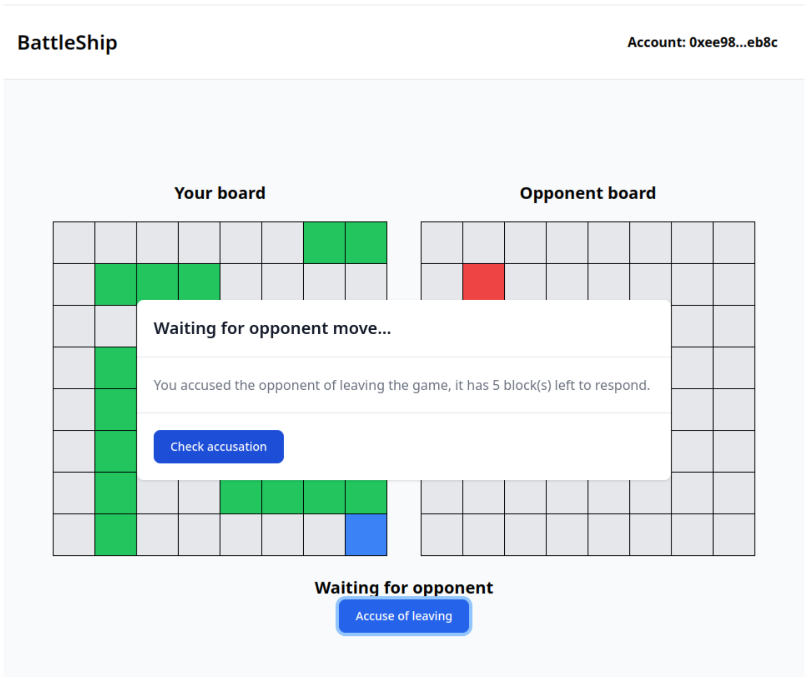


Figure 2.7: Accuse of leaving

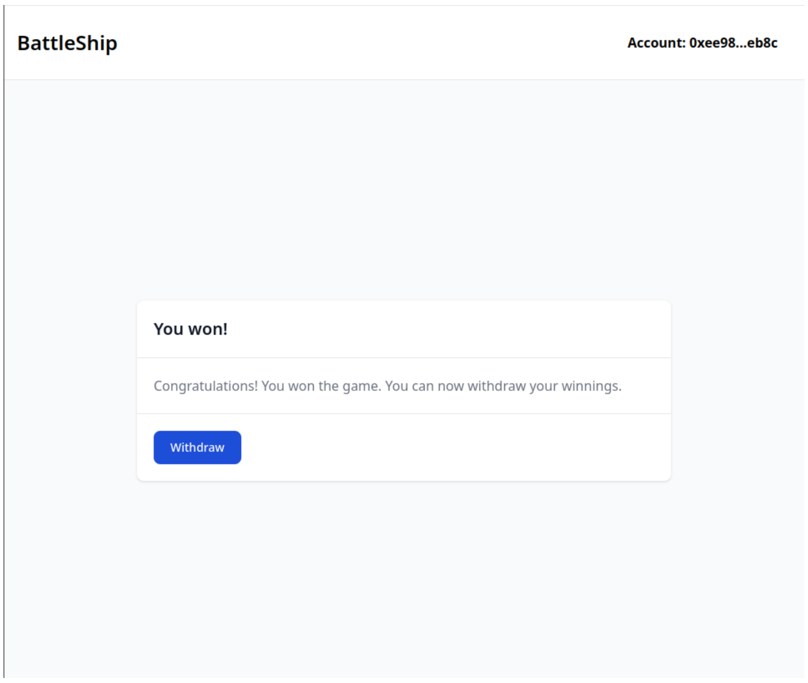
Once you accused the opponent, it has 5 blocks time to make his move, and you can check the status by clicking `Check accusation` in the modal:



You accused the opponent of leaving the game, it has 5 block(s) left to respond.

Figure 2.8: Check leaving

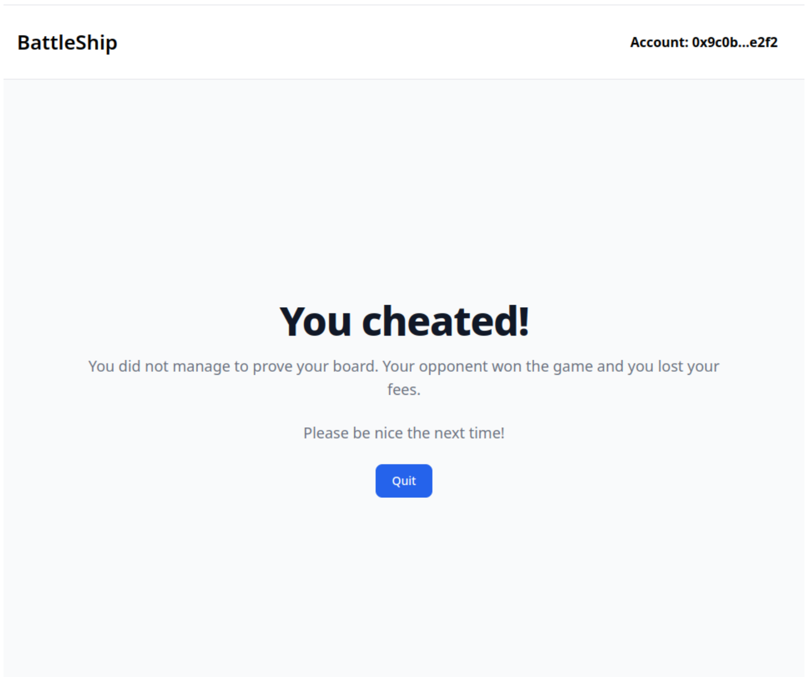
If the opponent makes his move in time, the modal will disappear, instead if the opponent actually left the game, then you will be moved to the winning page.
Once the game is over the users have to reveal their boards by clicking on the `Reveal board` button. Then after the board check (or previously if the opponent cheated), if you are the winner the page will be:



Congratulations! You won the game. You can now withdraw your winnings.

Figure 2.9: Winning page

and you can click on `Withdraw` in order to collect your win.
Instead you were found as a cheater:


BattleShip

Account: 0x9c0b...e2f2

You cheated!

You did not manage to prove your board. Your opponent won the game and you lost your fees.

Please be nice the next time!

Figure 2.10: Cheating page

3 | Cost analysis

The following costs are extracted via the ganache testnet and could differ from the actual deploy on main-net.

Method name	Complexity	Estimated gas fee (gas)
Contract deploy	O(1)	4486857
`new_game`	O(1)	118919
`join_game` with existing ID	O(1)	75400
`join_random_game`	O(1)	75736
`join_random_game` without existing game	O(1)	119121
`propose_fee`	O(1)	55749
`propose_fee` moving to the next status	O(1)	60683
`commit_board`	O(1)	58570
`commit_board` moving to the next status	O(1)	109569
`submit_move`	O(1)	41478
`submit_move` moving to the next status	O(1)	46809
`submit_proof`	O(1)	57505
`submit_proof` moving to the next status	O(1)	72549
`reveal_board`	O(1)	242473
`reveal_board` moving to the next status	O(1)	255728
`withdraw`	O(1)	46204
`make_accusation`	O(1)	81961
`submit_move` with pending accusation	O(1)	44462
`submit_proof` with pending accusation	O(1)	89328
`reveal_board` with pending accusation	O(1)	255381
`check_accusation`	O(1)	34745

Table 3.1: Gas fee table

Let's assume a full match in which each shot is a miss (except for the last 16) and that the player already know each other and no one leaves the game, then player will make the following calls:

- `new_game`;
- `join_game`: let's suppose that the users know one each other and that they share the game id;
- 2 calls to `propose_fee`: let's suppose that both players already agreed on the fee and they just submit it without actually negotiate it;
- 2 calls to `commit_board`;
- 128 calls to `submit_move`, `submit_proof`: cause we are assuming that all the shots are miss, except for the last 16;
- 2 calls to `reveal_board`;
- 1 call to `withdraw`.

For a total of 14997119 gas that can be divided by two, on average, to get the cost for each player.
At the current cost of gas (15 gwei per gas) and at the current cost of Ethereum (2024.93€ per ETH), a single battleship match would cost the users 0.224956785 ETH, approximatively 455.52€

4 | Vulnerability analysis

4.1 | Reentrancy

A reentrancy vulnerability arise when a target contract calls a function from the attacker controlled ones, and then this one calls again the target contract. This could be exploited, for example in the withdraw function because the target account calls a payable function of an attacker controlled address, if the contract does not upgrade it's status before actually calling the payable method an attacker can call the withdraw method multiple times and get more coins or tokens than what it should get.
The project contract is not vulnerable to reentrancy attack because the withdraw method update the player status before actually calling the paying method:

```
require(!player.withdrawed, "You already withdrawed");

uint256 amount = current_game.A.proposed_fee * 1e9;
if (current_game.winner != address(0)) {
    amount <<= 1;
}

// Avoid reentrancy
player.withdrawed = true;
payable(msg.sender).transfer(amount);
```

this way if the attacker controlled contract calls again the withdraw method the transaction will be reverted because of the `require` call.

4.2 | Denial of Service

A denial of service vulnerability allows an attacker to block the usage of the contract, or part of it to all the other users.
The contract in analysis is vulnerable to a weak form of denial of service because an attacker can deploy own contract that calls `join_random_game` in loop and empty the game queue. This way the attacker will match every new game avoiding the possibility to actually play.
However this is not a real concern because the attacker should pay a lot of gas fee in order to saturate all the list. A possible fix would be to make a maximum number of game in which a user could be at the same time, and track this statistic in a new mapping, however is not a valid patch because the user can generate numerous addresses and repeat the attack.

4.3 | Locked funds

Due to how the game, the withdraw and the accusation mechanisms are implemented an attacker can force the user to lock some funds in the contract without having the possibility to get them back. Once the matchmaking is done and the game fee is negotiated, both user should publish their merkle tree root and pay the fee, however if only one of the two players actually does it and the other one does not, the first one will have it's funds locked.
There is no way in the contract to withdraw those funds because:

- the withdraw function can't handle this case, because can be called only in the `GameOver` state;
- the accusation mechanism works only in `WaitingForMove`, `WaitingForProof`, `WaitingForReveal`.

A possible fix to this vulnerability could be to handle this case in the withdraw method and move the game in the `GameOver` phase with no winner.