UNIVERSITY OF PISA

DEPARTMENT OF COMPUTER SCIENCE

P2P Systems and Blockchains

Hit and Sunk! Playing Battleship on Ethereum

Lorenzo Angeli 539036

ACADEMIC YEAR 2022/2023

Main Decision

The concept involved utilizing Truffle Box, a tool that allows developers to focus on the distinctive aspects of their app. In detail, the "pet shop" example discussed in class was used as the project's foundational structure.

Library details

To execute the test (test/battle ships.js), you are required to install the keccak library for computing the 256-bit keccak hash (using npm install keccak256), and a library for constructing Merkle trees (using npm install merkletreejs). In order to work, the project uses the keccak library to compute the 256 bit keccak hash and a library to build Merkle trees and verify proofs.

Data Structure

In the project, three different arrays were used:

1. *BattleShip[] private battleships*
2. *ShipSunk[] private shipsunks*
3. *InfoStake[] private infostakes*

The first is an array of BattleShip.

```
struct BattleShip {
    uint id;
    uint n;
    uint state;
    uint turn; //0: turn of p1, 1: turn of p2
    address p1; //player1 public address
    address p2; //player2 public address
    uint stake_p1; //stake proposal player1
    uint stake_p2; //stake proposal player2
    uint stake_lock;
    bytes32 root_p1; //Merkle Tree root player1
    bytes32 root_p2; //Merklet Tree root player2
    uint balance; //balance in Wei of the game
    address winner;
}
```

This structure is essential for memorizing a particular battleship game. Each game is associated with a unique ID. The state is an enumeration which defines the current game situation, it is an unsigned integer between 0 and 6:

- 0: both players did not join the match (waiting for the opposing player).
- 1: the second player entered the game.
- 2: both players have set the stake.
- 3: both players have set their own merkle tree root.

- 4: both players have paid the stake.
- 5: the player who wins the game must verify that he has not cheated by calling the method *verify_winner*
- 6: the game is over and the winner received the reward in Wei.

The three variables *stake_p1*, *stake_p2* and *stake_lock* are used to manage the phase in which the two players agree on the stake in case of victory. The first two variables are used to fix the proposed stake value of the players. The *stake_lock* is an unsigned integer between 0 and 2:

- 0: it is up to player1 to propose the stake
- 1: it is up to player2 to propose the stake
- 2: an agreement was found between the two players for the stake

It is player1 who starts proposing to player2 the stake by calling the smart contract method *propose_stake*. This method will set the variable *stake_p1* with the value proposed by the player1 and the variable *stake_lock* to 1. At this point, player2 will make his proposal (*stake_lock* will be set to 0) and if this will be equal to the value set by player1, *stake_lock* will be set to 2, an agreement will be found and it will be possible to move on to the next phase (and the next state). If not, player1 will make a new proposal and so on, until they reach an agreement.

The second is an array of ShipSunk. This structure is used to count boats that have been hit by torpedo. Each item in the array corresponds to a different game, the match to which it refers is identified by id.

```
struct ShipSunk {
    uint id; //unique
    uint ss_p1; //number of piece of ships sunk
    uint ss_p2; //number of piece of ships sunk
}
```

The third is an array of InfoStake.

```
struct InfoStake {
    uint id; //unique
    bool p1_paid;
    bool p2_paid;
}
```

This array maintains player stake payout information. The two boolean variables refer to the two players and are set to true if that particular player has deposited Wei on the smart contract. Once both variables are true, the game can begin and from this moment it is no longer possible to withdraw the Wei paid.

Mapping

Mappings in Solidity are hash tables that store data as key-value pairs, where the key can be any of the built-in data types supported by Ethereum. Six different map were used:

1. *mapping(uint → Torpedo[]) private torpedo*
2. *mapping(address → bool) private players*
3. *mapping(uint → bool) private game_id*
4. *mapping(uint → Accuse) private accuse*
5. *mapping (uint → CountCheat) private countcheat*
6. *mapping (uint → bool) private reply*

The first mapping associates each game (with a unique id) with all the torpedoes launched by both players up to that moment, and it returns an array of Torpedo struct when given the id of a match.

```
struct Torpedo {
    uint x;
    uint y;
    uint player; {0->player1, 1->player2}
    bool verify;
    uint result; //{0->miss, 1->hit, 2->not set}
}
```

The *player* variable can have a value of 0 or 1. It will be 0 if the torpedo was launched by player1, it will be 1 if the torpedo was launched by player2. The *verify* variable check if this torpedo is verified or not; if it is false it means that I have not already verify it and the player must call the *reply_torpedo* to tell if it is a miss or hit, otherwise it means that he has already done this operation. The *result* variable will contain 0 if the ship has been missed, 1 if it has been hit or 2 if the *reply_torpedo* method has not yet been called.

The second map takes an address (a player) and if the address is playing a game the result is true, otherwise is false.

The third mapping takes an integer that corresponds to the id of a game and if the game exists it returns true otherwise false. It will return false if that passed id has never been used for any game up to that point.

The fourth map takes an integer that corresponds to the id of a game and return an Accuse object in the event that one of the two players has accused the other of leaving the game.

```
struct Accuse {
    bool accused; //if true the player is accused, otherwise false.
    uint blocknumber;
    address player; //accused player
}
```

Player A can accuse player B only if it is not A's turn to play. The *accused* variable is true if the player whose address is saved in the *player* variable is accused to have left the game. When the player is accused, i memorize the block number (that is the number of the last block of the chain of the blockchain in that instant) inside the variable *blocknumber*. Once the *accuse_adversary* method has been called, if the time limit of 5 blocks has been exceeded before the accused player has delayed a *launch_torpedo* or a *reply_torpedo* (it depends on the game situation) he lost by forfeit. In case the 5 block limit has been exceeded, the Wei can be obtained in two ways:

- with the *reward_accuser* method, clearly this call will only take effect if called by the winning player.
- if the accused player will call the *launch_torpedo* method or a *reply_torpedo* (it depends on the game situation).

The phrase "it depends on the game situation" implies that in case the last method was a *launch_torpedo* from the opposing player, in my case I will have to send a *reply_torpedo*. In case the opposing player has sent me a *reply _torpedo*, he will be up to me to launch a torpedo with the *launch _torpedo* method.

The fifth map takes an integer that corresponds to the id of a game and return an CountCheat object in case one of the two players tried to cheat.

```
struct CountCheat {
    uint p1; //number of times the player1 has attempted to cheat
    uint p2; //number of times the player2 has attempted to cheat
}
```

This item counts the number of times players have attempted to cheat. After four attempts the player loses at the table. The first player to pass four cheat attempts is disqualified from the match. The other player wins the match along with the reward in Wei.

The last map takes an integer that corresponds to the id of a game and return true or false. This map is used to implement game turns together with the unsigned integer variable *turn* contained in the BattleShip struct. The use of the turn variable together with the map guarantees the correct handle of game turns. The *turn* variable is set to 0 if it is player1's turn, to 1 if it is player2's turn. In particular, when we talk about player 1's turn we mean that it's up to player 1 to execute the launch_torpedo or wait for player 2 to execute the reply_torpedo. To clarify whether it is the first or second move we introduce a new boolean variable with the *reply* map. It is sufficient to pass the id of the game to the map to get the result of that value. If the result of the map is false it means that the player whose turn it is will have to launch_torpedo. If the variable is false instead it's up to the other player to call the reply_torpedo. Let's summarize when said:

- if the value is true and turn=0 (game turn variable) player 2 must make the reply_torpedo.
- if the value is true and turn=1 player 1 must make the reply_torpedo.
- if the value is false and turn=0 the player 1 must launch_torpedo.
- if value is false and turn=1 player 2 must launch_torpedo.

Project Structure

The most important files of the project are these:

- *contracts* Folder: contains three files (*BattleShips.sol*, *MT.sol* and *SafeMath.sol*). The first is the smart contract and the others are libraries. The library contains a utility method and a method to date a leaf to verify that it belongs to the Merkle Tree.
- *migrations* Folder: contains the *1_deploy_contracts.js* file used to deploy the contract and library.
- *src/js/app.js*: this is the file that contains the logic of dAPP and the handling of all the callbacks. All the contract calls (setters, getters and events) are handled in this file.
- *src/index.html*: contains the graphical interface to be able to interact with the smart contract.
- *test/battle_ships.js*: truffle comes standard with an automated testing framework to make testing your contracts a breeze.
- *truffle-config.js*: used to set the optimization of the smart contract code, the network and the version of Solidity to use.

All file structure is shown in Figure 1.

Methods

In this section we will describe the most important methods that have been implemented in the smart contract.

- new_game(uint n): take the size of the board as input and create a new game if the player calling this method is not already in another game.
- join_casual_game(): join a game with a random opponent. It returns id of the game if the player finds an available game, otherwise -1.
- join_game(uint _id) public check_players(): join the newly-created game with id of the game (play with a friend).
- propose_stake(uint _id, uint stakevalue): method used by the two players to decide the stakes of the two players. The first to propose the stake is always player1. As arguments this method takes the id of the game and the stake value he wants to propose (in wei).
- set_root(uint _id, int[] memory board, uint[] memory random_board): once the two players have configured their board by inserting the boats (via front end), they will have to memorize the root of the Merkle Tree on the smart contract. This method takes the game id and two arrays of unsigned int as arguments. The first matrix in the form of an array contains the display board represented on the array where the value is -1 if there is no boat in the cell or

```
/
├── p2p-report.pdf
└── battleship-dapp
    ├── build (generated)
    ├── contracts
    │   ├── BattleShips.sol
    │   ├── MT.sol
    │   └── SafeMath.sol
    ├── migrations
    │   └── 1_deploy_contracts.js
    ├── node_modules (generated)
    ├── src
    │   ├── css
    │   │   ├── bootstrap.min.css
    │   │   ├── bootstrap.min.css.map
    │   │   └── styles.css
    │   ├── images
    │   │   ├── drop.png
    │   │   └── flame.png
    │   ├── js
    │   │   ├── app.js
    │   │   ├── bootstrap.min.js
    │   │   ├── truffle-contract.js
    │   │   └── web3.min.js
    │   └── index.html
    ├── test
    │   ├── battle_ships.js
    │   └── mylib.js
    ├── truffle-config.js
    ├── package.json
    ├── package-lock.json (generated)
    └── bs-config.json
```

Figure 1: The project *directory* structure

a value >=0 if there is a boat. This matrix associates an index with the boats and places them on the board. The second matrix contains the random salts, to be concatenated with the elements of the board. Basically, this method calculates the root of the Merkle Tree directly on the smart contract through the *calculateRoot* method present in the MT library.

- transfer_wei(uint _id): method used by the players to send wei to the contract after an agreement has been found between the players.
- recover_wei(uint _id): in case i deposited the wei and set the root but not the second player, i can withdraw the deposited wei. As soon as both players have transferred the money to the contract it will no longer be possible to use this method.
- launch_torpedo(uint _id, uint x, uint y): during his turn the player must throw a torpedo at his opponent to try to hit one of his boats. The x, y coordinates represent values from 0 to n-1 (with n dimension of the table).
- reply_torpedo(uint _id, string memory result, string memory randvalue, uint x, uint y, bytes32[] memory proof, uint256[] memory positions): method used to respond to the player who has just sent a torpedo and reveal whether or not he has hit one of his boats.
- verify_winner(uint _id, int [] memory board): this method if called by the player who lost the game has no effect. Alternatively, if called by the player who has won, it re-checks the matrix passed by argument comparing it with the torpedoes received to verify whether up to that moment the player had expressed the truth or not. In case he cheated, the stake is sent to the other player.
- accuse_adversary(uint _id): method invoked by the player who wants to accuse the other of having abandoned the game. if it's my turn I can't accuse the other player. if I have accused a player he is under indictment i cannot re-indict him again.
- reward_accuser(uint _id): method used by the accuser to obtain the entire reward. In the event that he has accused a player, if the next five blocks from the time the accuse_adversary method is called contain no calls from the accused player, the latter will forfeit the game. Therefore, the stake will be paid to the player who originally started the accusation.
- next_block(): method used to create a new block on Ganache and simulate victory by forfeit (by abandonment).

Matrices

In the implementation of the dAPP backend (src/js/app.js) some matrices have been used. The *board_1* array represents the matrix of the board and contains in each cell a string, whose value is "0" if no ship has been placed on that cell, "1" if the cell contains a ship. To build the Merkle Tree we have to assign to each leaf of the tree the contents of a cell of the game board. Following the indications provided by the project requirements it is necessary to concatenate the value of the cell (which as we know can assume the value "0" or "1") with a random value (random salt). For this purpose, the *M_random* matrix was introduced which will contain the random values assigned to the leaves. The M_final array contain the results of the cells of the table concatenated to the random values. The *board_ship* is an array of the board where on the cells there are the ids of the positions of the ships.

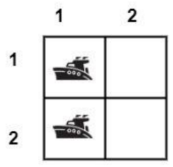

Figure 2: Board 2x2

In the case of a board like in Figure 2 the associated *board_ship* is this:

$$\begin{bmatrix} 0 & -1 \\ 1 & -1 \end{bmatrix}$$

where the cells containing the values -1 represent the empty cells (without boats) while the others contain boats. Each boat is assigned a unique id >= 0. This is because there may be boats with length greater than 1 and with this type of representation i am able to identify them.

Gas Cost Evaluation

An evaluation of the cost of the gas of the functions provided by the smart contract was made throught truffle test. The estimate was made by simulating a game with n=4 (4 x 4 matrix). Using the *estimateGas* method it was possible to estimate it. *estimateGas* generates and returns an estimate of how much gas is necessary to allow the transaction to complete. It is important to underline that transaction will not be added to the blockchain. So it was necessary to call the method twice, once with the *estimateGas* method and once without in this way:

```
var gasPrice = await web3.eth.getGasPrice();
var gasEstimate = await instance.new_game.estimateGas(n, {from: accounts[0]});
await instance.new_game(n, {from: accounts[0]});
console.log("gas estimation = " + gasEstimate + " units");
console.log("gas cost estimation = " + (gasEstimate * gasPrice) + " Wei");
```

The table 1 shows the gas costs of the various smart contract methods. In particular, on the second column we see them expressed in units of gas and on the third column expressed in Ether (1 ETH = 1000000000000000000 Wei).

We also provide an estimate of the total cost of the smart contract for a game played on an 8 x 8 board, assuming that each player gets all guesses wrong. We can estimate the cost of the game based on the table 1.

The 2 table, in addition to showing the estimated costs of the methods, also tells us how many times these methods must be called to complete a game on an 8 x 8 board. Let us assume that the two players immediately agree on the stake value, that they never call the *recover_wei* method (method used to recover the Wei paid to the smart contract) and that neither of them accuses the other player of having abandoned the game. The total cost is given by the following formula:

$$TotalCost = \sum_{x \in operations} (GasCostEstimation_x) \cdot (Call_x)$$

By doing the calculations i get:

$TotalCost = 0.0043767 \cdot 1 + 0.00206874 \cdot 1 + 0.00166616 \cdot 2 + 0.0032543 \cdot 2 + 0.00123754 \cdot 2 +$

$+0.0020097 \cdot (64 + 63) + 0.00306706 \cdot (64 + 63) + 0.00236316 \cdot 1 = 0.66353228\ ether$

Table 1: Gas cost estimates

Operation	**Gas Estimation**	**Gas Cost Estimation**
new_game	218835 units	0.0043767 ether
join_game	103437 units	0.00206874 ether
propose_stake	83308 units	0.00166616 ether
set_root	162715 units	0.0032543 ether
transfer_wei	61877 units	0.00123754 ether
recover_wei	53141 units	0.00106282 ether
launch_torpedo	100485 units	0.0020097 ether
reply_torpedo	153353 units	0.00306706 ether
verify_winner	118158 units	0.00236316 ether
accuse_avversary	104573 units	0.00209146 ether
reward_accuser	37380 units	0.0007476 ether

Table 2: Gas cost 8 x 8 game estimate

Operation	**Gas Estimation**	**Gas Cost Estimation**	**Call**
new_game	218835 units	0.0043767 ether	1
join_game	103437 units	0.00206874 ether	1
propose_stake	83308 units	0.00166616 ether	2
set_root	162715 units	0.0032543 ether	2
transfer_wei	61877 units	0.00123754 ether	2
recover_wei	53141 units	0.00106282 ether	0
launch_torpedo	100485 units	0.0020097 ether	64+63
reply_torpedo	153353 units	0.00306706 ether	64+63
verify_winner	118158 units	0.00236316 ether	1
accuse_avversary	104573 units	0.00209146 ether	0
reward_accuser	37380 units	0.0007476 ether	0

Deployment Cost

The table 3 shows an estimate of costs to deploy the two libraries MT and SafeMath and the contract BattleShips where the gas price is set at 2.500000008 gwei (1 ETH = 0.000000001 gwei).

Table 3: Deployment cost

Contract/Library	**Cost**
BattleShips	0.012573537540235532 ether
MT	0.002415480007729536 ether
SafeMath	0.000179772500575272 ether

Potential Vulnerability

The *SafeMath* library has been introduced to make addition and subtraction operations between unsigned integers safe. This system avoids overflow and underflow problems. Another important aspect is that related to Ethereum addresses. In Ethereum, there are two global addresses: msg.sender and tx.origin. The first gives the direct sender of the message. The second instead gives the origin of the transactions, thus the user address from which it was originally sent, in practice this will always be a user. It is good practice in our case to always use msg.sender to refer to the person who called the smart contract method. This prevents vulnerabilities as we saw in class in the example of the Malicious dAPP. Finally, we are also able to handle the reentrancy attack. In fact, to transfer money from the smart contract to an EOA, we use the *transfer* method, which sets a gas limit and is more safe than the *send* method and the *address.call.value* method.

Merkle Tree

A Merkle Tree Library (documentation: *merkletreejs*) has been introduced to prevent players from cheating. Once both players have entered the game they will have to configure their game board with the ships and then calculate the root of the Merkle Tree via *set_root* method.

```
//convert string[] to int[]
var intBoard = App.board_ship.map((integer) => parseInt(integer));
var intRandomBoard = App.M_random.map((integer) => parseInt(integer));
//set root
await instance.set_root(Number(gameid), intBoard, intRandomBoard);
```

This method takes the game id and two arrays of unsigned int as arguments. The first matrix in the form of an array (intBoard) contains the display board represented on the array where the value is -1 if there is no boat in the cell or a value >=0 if there is a boat. The second matrix (intRandomBoard) contains the random salts, to be concatenated with the elements of intBoard.

With the *reply_torpedo* (which is a smart contract method) we reply to the adversary by telling him whether the sent torpedo (launched with the *launch_torpedo* method) hit a ship or not. The *reply_torpedo* method contains 7 arguments:

- *gameid*: represents the unique identifier of the game.
- *result*: it is a string that can take on two values. It is set to "1" if the opponent hit the ship, "0" otherwise.
- *randvalue*: represents the value of the array *M_random* at position (x+n*y).
- x: abscissa coordinate of the torpedo to check.
- y: ordinate coordinate of the torpedo to check.
- *proof*: represents the proof used to verify that the player is not trying to cheat by declaring falsehood. For example, he could say that there are no boats at the (x,y) coordinates of his board.
- *positions*: to ensure the order of proofs and concatenate the leaves correctly and then be able to compute the hash with keccak256, the positions array was used. For more information, see the following link: merkle-tree-verification-fix.

```
var leaf = keccak256(M_final[Number(x)+Number(n)*Number(y)]);
const leaves = M_final.map(v => keccak256(v));
const tree = new MerkleTree(leaves, keccak256, {sort: true});
var proof = tree.getHexProof(leaf);
var positions = tree.getProof(leaf).map(x => x.position === 'right' ? 1 : 0);
var randvalue = M_random[x+n*y].toString();
await instance.reply_torpedo(gameid, result, randvalue, x, y, proof, positions);
```

At this point, *reply_torpedo* can verify that the player is not trying to cheat. To do this, use the *verify* method provided by the *merkletreejs* library.

```
verify(root, keccak256(abi.encodePacked(result, randvalue)), proof, positions)
```

This method returns a boolean value which will be false in case the player is trying to cheat.

Set Up Instructions

Let's see a user manual with the instructions to set it up and try it. Make sure you have:

- Node.js
- npm
- Truffle
- Ganache

- MetaMask (on the browser via extension)

To start the application locally follow these steps:

1. *Install modules*: Place a terminal in the project root, run *npm install*, this generates the node_modules folder. After that, install the *MerkleTree.js* library with *npm install merkletreejs* and the *keccak256* library with *npm install keccak256*.

2. *Run Ganache*: open Ganache and create a new workspace with the name of your choice and linking the truffle project to this workspace by adding the truffle-config.js file.

3. *Project Migrate*: now, it is enough to deploy the contract and the library on the local Ganache blockchain from the root folder of the project through the *truffle migrate* command.

4. *MetaMask*: we need an Ethereum wallet to interact with the project. We already have 10 accounts with plenty of (fake) Ether in our Ganache instance. Let's connect Metamask and the Ganache accounts to the Metamask wallet: open Metamask and login into the application, add the local network implemented by Ganache to Metamask. Click in the network drop down menu at the top of the page and select "Add Network", select "Add a network manually" at the bottom of the next page, add a new network to Metamask with the network details from the Ganache network (in our case it will be HTTP://127.0.0.1:7545). The ChainID is always the same for local networks: 1337. Save and connect to the new local network. The Ganache accounts need to be added to Metamask first. Open Metamask and click on the account symbol in the upper right corner and select "Import account" from the drop-down menu. We need to provide the account's private key to add it to Metamask. Go back to Ganache and click the key symbol in Account 1 to show its private key and past the Private key on Metamask. Remember to run Ganache whenever you want to run the project.

5. *Run dAPP*: you can start your dAPP with the command *npm run dev.* The browser should open automatically, if that doesn't work you can manually connect to http://localhost:3000.