Chainship - Architecture

Filip Krawczyk

April 30, 2025

1 Introduction

2 Game logic

2.1 Game state

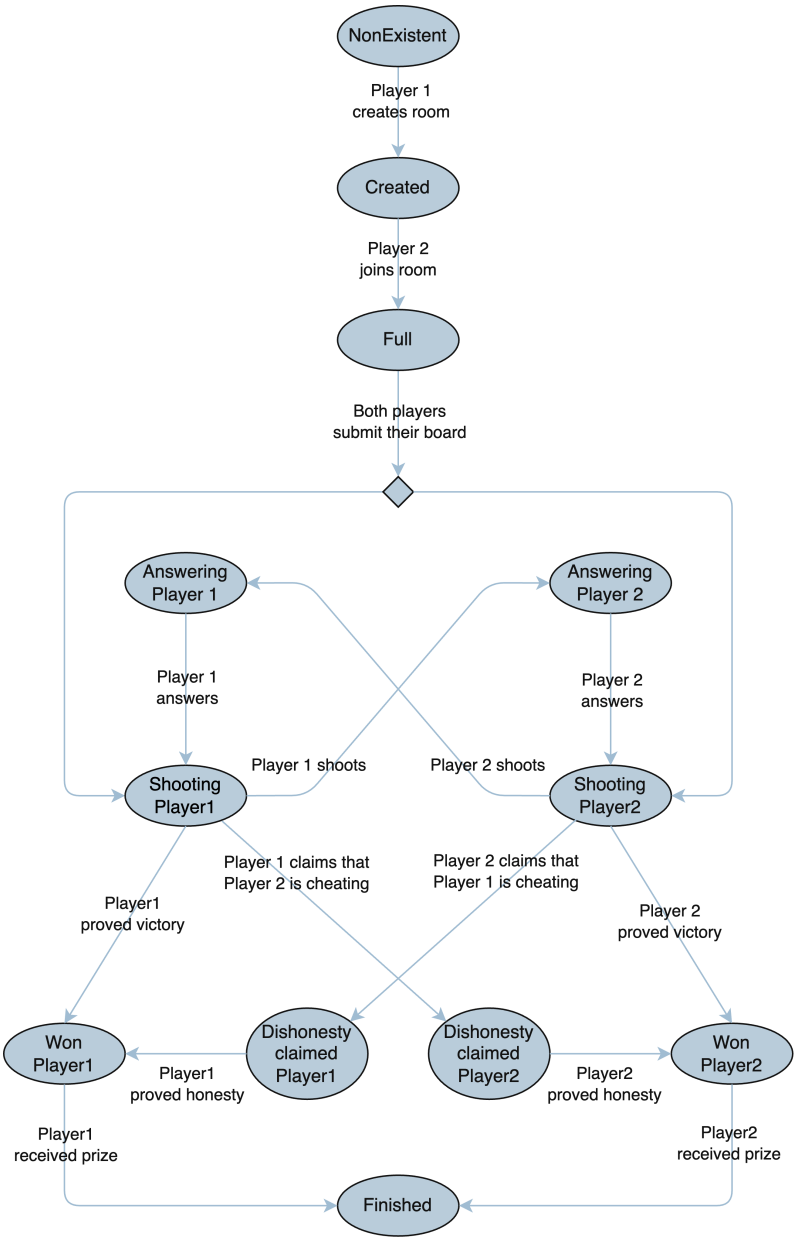


Figure 1: Game state

2.2 Randomizing first shooter

Since draw is not allowed, randomizing first shooter is crucial.

2.3 Proof of victory

Players can win in three ways:

1. Sinking all enemy ships.
2. Showing that the opponent didn't answer correctly to shots.
3. Enemy didn't answer in given time.

To prove correct answers, the player has to provide decommitment to their board, list of answers and list of enemy's shots. Then, the contract checks if following criteria are met:

1. Board commitment was correct
2. List of answers hashes to value stored in a contract
3. List of enemy's shots hashes to value stored in a contract
4. All answers were correct, given the board.

Additionally, when proving victory, every enemy's ship has to be hit.

3 Cryptography

3.1 Board commitment

Board is a $n \times n$ grid of cells. Each cell can be either empty or occupied by a ship. Players commit to the whole board before shooting stage begins. We define:

$$\text{BoardHash}(b) = \text{keccak256}(b_{1,1}, b_{1,2}, \ldots, b_{1,n}, b_{2,1}, b_{2,2}, \ldots, b_{2,n}, \ldots, b_{n,1}, b_{n,2}, \ldots, b_{n,n})$$

where $b_{i,j}$ is a boolean value that is true if there is a ship in cell (i, j) and false otherwise.

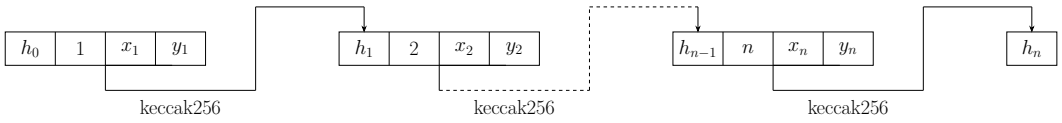
For small boards, it is easy to compute board hash for every possible board and therefore it wouldn't hide ships location. Because of that, randomness is introduced and board commitment is defined as follows:

$$\text{BoardCommitment}(b, r) = \text{keccak256}(\text{BoardHash}(b) \mid\mid r)$$

3.2 Shots hash

To reduce amount of data stored in a contract, shots are not stored directly. Instead, a single hash of all shots is stored. It is constructed in a way that makes it easy to iteratively compute it from previous hash and new shot.

Obviously, hash h_n is a function of all the shots x_i and y_i for $i \in \{1, 2, \ldots, n\}$, so given $h_n = \text{ShotsHash}(x_1, y_1, x_2, y_2, \ldots, x_n, y_n)$ it is computationally infeasible to find different $(x'_i, y'_i)_{1 \leq i \leq n}$ such that $h_n = \text{ShotsHash}(x'_1, y'_1, x'_2, y'_2, \ldots, x'_n, y'_n)$.



$$h_0 = 0$$

$$h_i = \text{keccak256}(h_{i-1} \mid\mid i \mid\mid x_i \mid\mid y_i) \text{ for } i \in \{1, 2, \ldots, n\}$$

$$\text{ShotsHash}(x_1, y_1, x_2, y_2, \ldots, x_n, y_n) = h_n$$

The contraction of the commitment also makes it easy to compute the commitment for a new shot (x_{n+1}, y_{n+1}) given the previous commitment h_n and the new shot. Smart contract only needs to store i and h_{i-1} for each player and whenever a new shot is made, it is broadcasted and the commitment is updated.

3.3 Answers hash

Hash of answers for shots is constructed in similar way except that answers a_i are included in the hashed value, that is:

$$h_i = \text{keccak256}(h_{i-1} \mid\mid i \mid\mid x_i \mid\mid y_i \mid\mid a_i)$$