Università di Pisa

Peer to Peer Systems and Blockchains (261AA)

EtherMind Report

Academic Year 2023/2024

Gabriele Pongelli
Jacopo Di Domenico

Contents

1 Introduction

This paper presents the Mastermind game that we implemented using Ethereum smart contracts; it prevents cheating and punishes the guilty players thanks to the use of the blockchain.

2 Initial Assumptions

Before starting to explain all the implementation details we will expose here a list of initial assumptions and values chosen for some constants. This because some of the decisions made during the implementation of the smart contract are driven by these assumptions. We assumed that:

- The number of possible colors a player can choose from is fixed to 6.
- Each guess/solution is composed by a total of 4 colors.
- The blockchain on which the contract will be deployed produces in average one block every 12 seconds (we searched for this value, and it appears that the Ethereum blockchain has an average block time of 12 seconds [1]).

In addition, for some game parameter variables we have fixed the following values:

- A match is composed of a total of 4 rounds.
- In each round, the CodeBreaker can submit up to 12 guesses.
- If the CodeBreaker is not able to guess the code submitted by the CodeMaker in the maximum number of guesses, 6 extra points will be assigned to the CodeMaker.
- The CodeBreaker can start a dispute after the end of a round only before a timer of 90 seconds expires. This timer is started when the CodeMaker submits the final solution at the end of the round. After the timer is expired, the CodeBreaker won't be able to start a dispute for the previous round.
- If a player decides to start the procedure for terminating the match prematurely for AFK reasons, it must wait at least 180 seconds (from the time the procedure is started) before it can end the match.

This second list contains values that can be arbitrarily tuned without impacting on the contract implementation decisions.

[1] `https://www.nervos.org/knowledge-base/block_time_in_blockchain_(explainCKBot)`

3 Implementation Details

From now on, we will use the following terms:

- *Match*: defines an instance of the game.
- *Private Match*: a match that can be joined only by a player with a specific address.
- *Public Match*: a match that can be joined by anyone.
- *Creator*: is the player that created a match.
- *Challenger*: is the player that joined an existing match.
- *Opponent*: w.r.t. one player, it is the other player that is playing in the same match.

3.1 Phases

We decided to design a match as a state machine where each state corresponds to a specific phase of the match. The possible phases are:

- `NOT_CREATED`: defines a match that has not been created yet.
- `PENDING`: the match has been created and the creator is waiting for a challenger.
- `STAKE_DECISION`: a challenger joined the match, and they now need to agree on the stake value for this match.
- `STAKE_PAYMENT`: the players of the match found an agreement on the stake value, and now they have to pay such value.
- `ROUND_START`: a new round is started. This phase is initially reached when both players have payed the agreed stake amount. Then instead is reached after that the previous round is ended and the CodeMaker submitted its solution.
- `GUESS_SUBMISSION`: the CodeBreaker has to make a guess. This phase is initially reached inside a round after the CodeMaker uploaded the hash of its solution. Then instead is reached after that the CodeMaker uploaded a feedback about the previous guess saying that the guess wasn't correct.
- `FEEDBACK_SUBMISSION`: the CodeBreaker has uploaded a guess and the CodeMaker has to upload the correspondent feedback.

- `ROUND_END`: the current round is terminated (either because the maximum number of guesses is reached or because the CodeBreaker submitted a correct guess) and now the CodeMaker has to upload the solution of this round round.
- `GAME_ENDED`: the match is ended and the players can discover who is the final winner.

A scheme of the state machine can be seen in Figure 1.

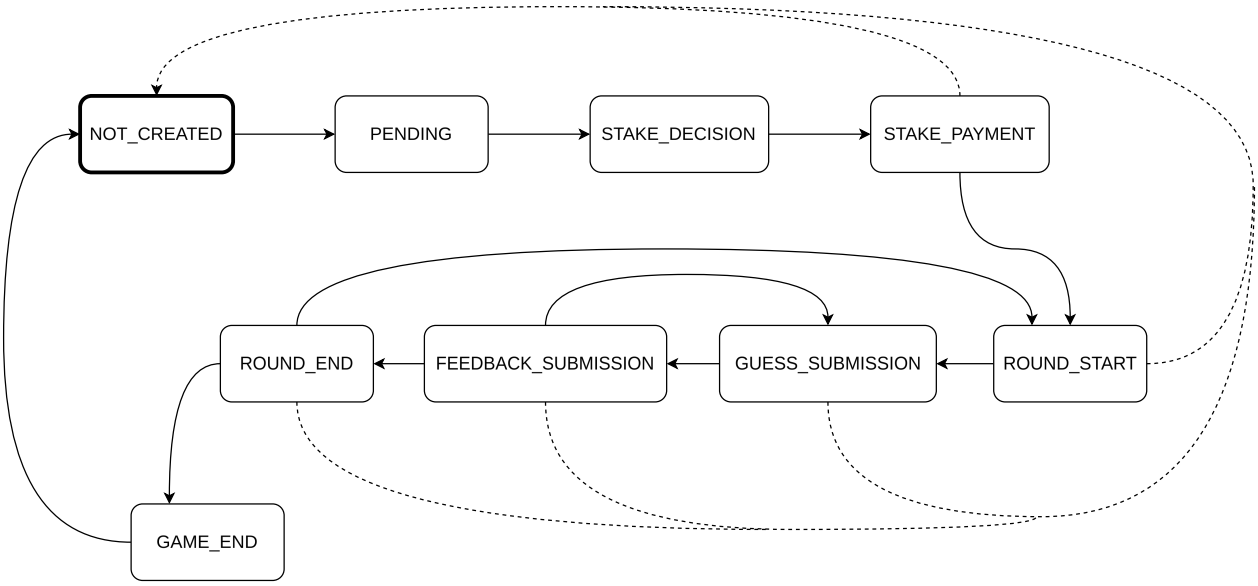

Figure 1: State machine representing the evolution of a match. The node with a bold border is the initial state. Normal transitions are represented with "full" arrows, instead transitions that caused a premature end of the match (for a dispute or for AFK) are represented with dashed arrows.

3.2 Match Implementation

In order to allow a fast and efficient search-and-retrieve process for a specified match state (which is performed very frequently), we decided to create a structure called `MatchRegister` which handles that, beside of proper creation, initialization, and deletion of matches. There are 2 cases to consider in a search-and-retrieve process:

1. The retrieval of match state given the identifier of the match.
2. The retrieval of a random public match.

The first case is addressed by the use of a mapping from addresses to match states. For what concerns the second case instead, we decided to solve it by storing a reference (the identifier)

to the public matches that are still waiting for a challenger inside an array. In this way we can randomly choose a public match by simply selecting at random one element of the array, and then retrieve its state using the reference stored in the array.

In case a public match that is `PENDING` advances to the next state, it should be removed from the array, but since we have only the identifier of the match we should search it in the array, and this operation will become always more costly as the number of public matches in state `PENDING` increases (and taken to the limit, a lot of gas will be required for someone to join a match). To solve this problem, instead of storing only the state of the match in the mapping, we decided to store also an index (incremented by 1) of the match in the array. In this way, the removal of a match from that array doesn't depend on the number of elements inside it. To better understand the final organization, the scheme in Figure 2 can be useful.

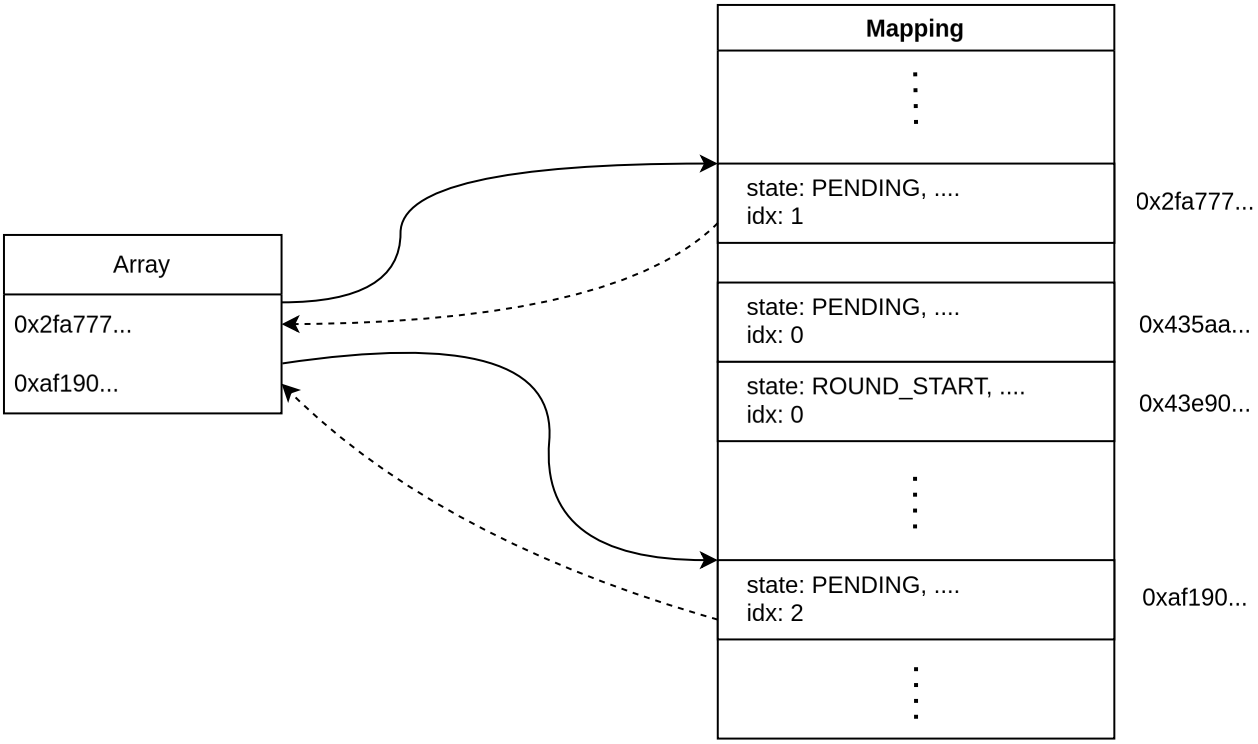

Figure 2: Implementation of the `MatchState` structure.

In this way, even if the number of matches grows, the time for a search-and-retrieve is constant, and this is extremely important since any additional execution time given by this operation would have been charged unfairly to the players.

3.3 Stake Decision Protocol

We decided to implement a simple protocol for the decision of the stake value on-chain. In this way:

1. Each step of the game is completely defined inside the smart contract.
2. Each step of the game is consistent across multiple match instances. If instead the decision of the stake was performed off-chain, different methods could be used, and we should have trusted the players in the definition of a method that was fair for both players.

We decided to implement a protocol in which players can update their previously proposed value by proposing a new one before the opponent propose its value. In this way, if someone mis-typed a value for a proposal, it can correct it (at the expense of making a new call to the contract). To do this, we keep track of who made the last proposal (see subsubsection 3.4.1). An agreement is reached when both players have proposed the same value as their last proposal.

A scheme of said protocol can be seen in Figure 3. To note that the first value is always

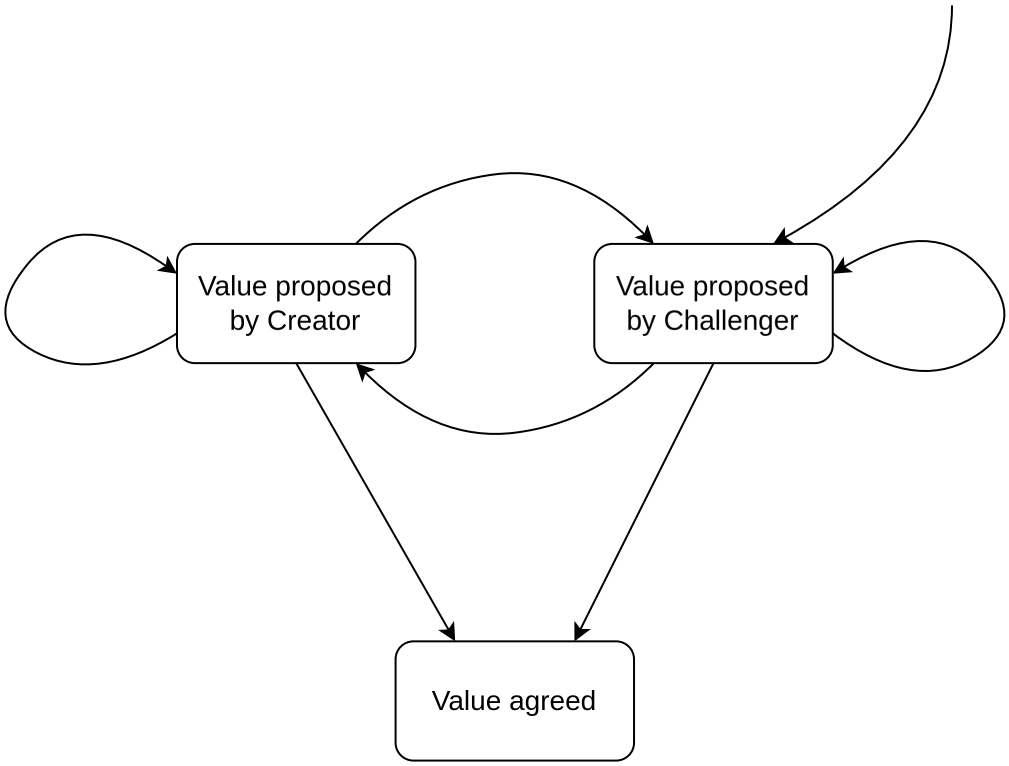

Figure 3: Protocol schema.

proposed by the challenger. This because the first proposal is made together with the join action. In this way we reduce the numbers of steps needed to reach an agreement, and we can easily handle the case in which there isn't an initial stake value proposed since now this case is no more possible.

3.4 Storage Optimizations

Since the state of each match has to be stored in the stable storage, and since the more stable storage is required, the more a user of the contract needs to pay in order to create a new match (which is where new storage is required), we decided to make some optimizations to try to decrease the costs. The results can be seen in the Cost Analysis.

3.4.1 State Flags

The state of each match, besides other variables, contains some variables used as flags that are needed for the handling of some cases. In particular, each match contains 8 flag variables:

1. `IS_PRIVATE`: says whether the current match is private or public.
2. `CHALLENGER_PROPOSED_STAKE`: says whether the last stake value in the Stake Decision Protocol has been made by the challenger or not.
3. `CREATOR_PAYED`: says whether the creator has payed the stake or not.
4. `CHALLENGER_PAYED`: says whether the challenger has payed the stake or not.
5. `IS_CHALLENGER_CODEMAKER`: says whether the CodeMaker in the current round is the challenger or not.
6. `AFK_CHECK_ACTIVE`: says whether an AFK Check is currently being performed or not.
7. `CB_WAITING`: says whether in this phase of the game, the CodeBreaker has to make and action or not.
8. `CM_WAITING`: says whether in this phase of the game, the CodeMaker has to make and action or not.

Instead of using a `boolean` type for each of these flags, occupying 1 byte for each variable, we decided to use a single variable of type `uint8` that acts as a bit vector, where each bit correspond to one of these flags. In this way, for each match we end up using only 1 byte instead of:

$$1\ byte \times 8\ flags = 8\ bytes$$

3.4.2 Color Combinations and Feedbacks

In order to be able to check the correctness of the feedbacks specified by the CodeBreaker when starting a dispute, we have to store all the guesses and associated feedbacks of the current round in stable storage. Since a guess is composed by a combination of 4 colors, if we have to use one variable for each of them, we require for each new guess to store 4 additional bytes. For what concernes the feedbacks, if we use a variable for the number of correct colors in the correct positions, and a variable for the number of correct colors in the wrong positions, we require for each new feedback to store 2 additional bytes.

Since the maximum number of guesses is 12, at each round we require to store a maximum of:

$$(guessSize + feedbackSize) \times maxGuessNumber = (4 + 2) \times 12 = 72 \; bytes$$

Since we have fixed the number of colors to 6, and we have fixed that a guess is a combination of 4 colors, we decided to represent a combination, which we called *Code*, using only a `uint16` variable, where:

- The bits b_0, b_1, b_2 are used to represent the first color.
- The bits b_3, b_4, b_5 are used to represent the second color.
- The bits b_6, b_7, b_8 are used to represent the third color.
- The bits b_9, b_{10}, b_{12} are used to represent the fourth color.

For what concerns the feedbacks, since each of the properties of a feedback can assume values in $[0, 4]$, we decided to represent them using only a `uint8` variable, where:

- The bits b_{0-3} are used to represent the number of correct colors in the correct position.
- The bits b_{4-7} are used to represent the number of correct colors in the wrong position.

Doing in this way, are able to halve the maximum number of bytes required at each round to a total of:

$$(guessSize + feedbackSize) \times maxGuessNumber = (2 + 1) \times 12 = 36 \; bytes$$

3.4.3 Phases' Shenaningans

In our implementation there is a total of 9 possible phases in which a match can be. We could simply have used an `enum` to represent them using a single byte, but then the operation of testing if a match is in one between multiple phases (which is an operation performed at each call) would required a high number of operations (other than less readable code). For this reason we decided to represent them using an `uint8` value, where each bit corresponds to one phase of the match (as we did for the subsubsect:flags). In this way we can simply perform those tests passing around a single byte and exploiting only some bitwise operations.

Doing in this way, however, leaves out one of the phases (the phases are 9, but 8 bits are used). Recalling the fact that we used a mapping for the storage of the match states (see subsection 3.2), since all the values inside a mapping are represented by default with all the bytes set to zero, we decided to associate the phase `NOT_CREATED` to the case in which all the bits of out `uint8` are set to zero. In this way we can still easily check if a match is in this state while using only 1 byte of data, and in addition all the states inside the mapping will be automatically initialized in the `NOT_CREATED` phase.

4 Cost Analysis

In order to perform the following cost analysis we used a Node package called `hardhat-gas-reporter`[2].

Solidity and Network Configuration				
Solidity: 0.8.24	Optim: true	Runs: 200	viaIR: false	Block: 30,000,000 gas
Network: ETHEREUM	L1: 6 gwei			2869.62 eur/eth

Results					
Contracts / Methods	Min	Max	Avg	n. calls	eur (avg)
`checkWinner`	87,363	97,012	90,508	23	1.56
`createMatch`	98,550	142,719	99,274	732	1.71
`dispute`	97,225	98,867	98,411	18	1.69
`joinMatch`	42,194	86,538	62,655	361	1.08
`newFeedback`	46,586	84,203	52,237	3240	0.90
`newGuess`	44,661	79,041	50,222	3325	0.86
`newSolutionHash`	44,416	59,890	52,599	572	0.91
`payStake`	37,479	74,615	61,348	991	1.06
`stakeProposal`	32,627	40,626	36,695	371	0.63
`startAfkCheck`	55,203	58,230	57,435	319	0.99
`stopMatchForAfk`	66,229	94,179	83,385	271	1.44
`uploadSolution`	64,524	127,657	102,906	655	1.77
Deployments				% of limit	-
EtherMind			3,005,962	10 %	51.76

Above it is possible to see a table summing up the cost analysis of the contract. The first portion of the table shows Solidity and network configuration used for this analysis. It is important to note that the *eur/eth* value does change in accordance to the value of Ethereum, even a small change has been noted to almost double the average cost to deploy the contract. The second portion estimates the used gas for each method, with the Min, Max and the Avg quantity tracked during the test, the number of calls performed during test for each method and the average cost in euro for each call. The reason for the disparity between the Max and the Min value for used gas is due to the fact that the consumption is dependant on the execution of the method, that is in case of a revert the cost in gas is lower compared to executing the method completely. Finally we can see the total cost of deploying the contract and the average amount of gas used for the whole contract.

[2] `https://www.npmjs.com/package/hardhat-gas-reporter`

5 Vulnerabilty Analysis

In this section we will list all the vulnerabilities that we have found and solved during the design and development of this contract.

5.1 Salted Hash Solutions

Keeping in mind the values we chose about the total number of possible colors and the number of colors needed to form a combination, we noticed that the total number of possible combinations is not very high:

$$nColors^{colorsPerCombo} = 6^4 = 1296\ combinations$$

This could be exploited by a malicious CodeBreaker in this way:

1. The CodeMaker chooses a combination of colors at the beginning of a round and submits its hash to the smart contract.
2. The smart contract stores that hash in stable storage, where it can be read by everyone.
3. The CodeBreaker reads the hash published on the blockchain
4. The CodeBreaker performs an off-line brute-force attack to find the correspondent color combination. Assuming that the attacker has a computer with which it can compute up to 1000 hashes/sec (normally it is much higher), it would require less than 2 seconds to find the correspondent color combination.
5. The CodeBreaker submit the combination found with the brute-force attack as guess and win the round.

To solve this problem, we decided to require the hash of the solution to be salted with a 4 bytes random number chosen arbitrarily by the CodeMaker, and revealed only at the end of the round together with the solution. In this way, the cardinality of the possible values that will be used to calculate the hash is much higher:

$$nColors^{colorsPerCombo} \times saltValues = 6^4 \times 2^{32} \approx 5 \cdot 10^{12}\ combinations$$

Now, always assuming that the attacker has a computational power of 1000 hashes/sec, this is the time required to find the hash input through brute-force:

$$\frac{totalValues/hashRate}{2} = \frac{5 \cdot 10^{12}/1000}{2} = \frac{5 \cdot 10^9}{2} \approx 86\ years$$

5.2 Change of Solution

There could be the possibility for the CodeMaker to submit the hash of a solution and then, at the end of the match, upload a a different solution-salt combination that leads to the same hash. Since we are using keccak256 as function to calculate the hashes, the CodeMaker being able to perform such an attack means that it succeeded in finding a collision. But if that is true it means that the Second Preimage Resistance property isn't valid for keccak. Since keccak is a cryptographic hash function, which is also second-preimage resistant, this attack isn't feasible.

5.3 Timestamp Manipulation

When a dispute request is received by the contract some checks are performed to verify its correctness, and one of these is that the request is being sent within the time limit for disputes.

If we used the timestamp provided with the transaction, we could be subject to an attack from malicious miners manipulating that timestamp value. Such an attack could be exploited in this way:

1. The miner join/creates a match.
2. When it plays the role of CodeMaker, it provides false feedbacks to the CodeBreaker so that it earns more points.
3. When the CodeMaker upload the solution at the end of the round, the timestamp of that transaction is saved and used as a reference for the dispute correctness control.
4. If the CodeBreaker starts a dispute, the miner add that transaction to the block being mined and set the timestamp a little more in the future so that the dispute request appears to be out of the time limit. The low time limits we used for disputes makes this even more possible.
5. The CodeBreaker is not able to call for a dispute, even if the miner as CodeMaker provided wrong feedbacks.

To solve this problem we determine the passage of time by taking note of the block number (`block.number`) when the CodeMaker submits the final solution. Knowing that on average the Ethereum blockchain mines a block for every 12 seconds (as we stated in out initial assumptions), we calculate the future block number that will represent the time limit in this way:

$$limitBlock = block.number + \frac{disputeTimeDelay}{12}$$

This result is then stored inside the state of the relative match, and when a dispute is requested this value is then checked against the current block number to see if the dispute is valid or not. This solution does introduces some granularity in the time measurement, but for our use-case is not a deal breaker.

The same time-measuring mechanism is used for the AFK Checks during the game.

5.4 Freezed Funds during Payments

Recall that AFK Checks could originally be started during the game.

One particular vulnerability is present during the stake paying phase (before the game begins): if a player pays his stake quota, but the opponent doesn't and stops all the interactions within that match, the money of the first player are stuck in the contract forever. To solve this problem we decided to add the possibility of flagging the opponent as AFK also during the payment phase, thus allowing a player to retrieve its money if necessary.

5.5 Re-entrancy Attack

Another possible vulnerability we had to consider is the one relative to the re-entrancy attacks. A re-entrancy attack exploits a vulnerability in smart contracts when a function makes an external call to another contract before updating its own state. This allows an external contract, possibly malicious, to "re-enter" the original function and repeat certain actions, like withdrawals, using the same state. To prevent such attacks, all withdrawal actions are performed as the last operation before terminating the execution of functions that include withdrawal behaviours. This prevents such attacks since, if performed, re-entrancy will fail due to the usage of a state that isn't the correct one to dispense the prizes/punishments anymore.

6 Instructions

In this section we will explain how to use the contract, what each method does and all the events emitted by the contract.

6.1 Create a Match

In order to create a new match, a user can invoke the following method:

```
function createMatch(address otherPlayer)
```

where the parameter `otherPlayer` can have 2 possible values:

1. The address of another player: in this case a private match is created where only the player identified by the specified address can later join the match.
2. The value zero: in this case a public match is created, and anyone can later join the match.

If the match is correctly created, this method will emit the following event:

```
MatchCreated(id, creator)
```

where:

- The value of `id` is the identifier of the newly created match.
- The value of `creator` is the address of the player who created the match (thus is the address of the caller of this contract method).

6.2 Join a Match

In order to join an existing match, a user can invoke the following method:

```
function joinMatch(address id, uint256 stake)
```

where:

- The parameter `id` can have 2 possible values:
  1. The value zero: in this case a public match is selected at random from the public matches which still waiting for a challenger.
  2. The identifier of an existing match, which can be a public one or a private one. In this last case, the address of the user joining the match must correspond to the address specified by the creator (see subsection 6.1).
- The argument value passed to `stake` is the proposed amount of Wei to place as game stake proposal.

If the user successfully joined a match, this method will emit the following events:

1. `MatchStarted(id, creator, challenger)`

   This event is emitted to signal that the match is no more pending. Its arguments are:

   - The value of `id` is the identifier of the match.
   - The value of `creator` is the address of the player who created the match.
   - The value of `challenger` is the address of the player who joined the match (thus is the address of the caller of this contract method).

2. `StakeProposal(id, by, proposal)`

   This event is emitted to signal that the challenger made a stake proposal. Its arguments are:

   - The value of `id` is the identifier of the match.
   - The value of `by` is the address of the player who made the proposal (in this case the address of the challenger).
   - The value of `proposal` is the value proposed in Wei.

6.3 Propose/Update/Confirm a Stake

In order to propose/update/confirm a value, a user can invoke the following method:

```
function stakeProposal(address id, uint256 stake)
```

where:

- The argument value passed to `id` is the identifier of the match.
- The argument value passed to `stake` is the amount in Wei.

The call to this function can have different effects:

- If the last stake value was proposed by the opponent, and the `stake` argument value is different from that one, a new stake proposal is made with the new value.
- If the last stake value was proposed by the same player now invoking the method, the stake value previously proposed is updated to the `stake` argument value.
- If the last stake value was proposed by the opponent, and the `stake` argument value equal to that one, then the stake value is confirmed.

In case of a proposal/update this method will emit the following event:

```
StakeProposal(id, by, proposal)
```

where:

- The value of `id` is the identifier of the match.
- The value of `by` is the address of the player who made the proposal/update.
- The value of `proposal` is the proposed/updated value in Wei.

Instead, in case of a confirmation this method will emit the following event:

```
StakeFixed(id, stake)
```

where:

- The value of `id` is the identifier of the match.
- The value of `stake` is the confirmed stake value in Wei.

6.4 Pay a Stake

After a stake value has been fixed (see subsection 6.3), in order to pay it a user can invoke the following method:

```
function payStake(address id)
```

where the argument value passed to `id` is the identifier of the match. If the stake payment is terminated successfully, this method will emit the following event:

```
StakePayed(id, player)
```

where:

- The value of `id` is the identifier of the match.
- The value of `player` is the address of the player who successfully payed the stake (thus is the address of the caller of this contract method).

In case, after this payment, both players have payed the stake, then this method will emit also the following events:

1. `GameStarted(id)`

   This event is emitted to signal that the game is finally started. The argument value of `id` is the identifier of the match.

2. `RoundStarted(id, round, codemaker, codebreaker)`

   This event is emitted to signal the start of a new round. Its arguments are:

   - The value of `id` is the identifier of the match.
   - The value of `round` is the number of the round that is just started (in this case it will be `1`).

- The value of `codemaker` is the address of the player that will play as the CodeMaker in this round.
- The value of `codebreaker` is the address of the player that will play as the CodeBreaker in this round.

In this case, as a side effect, if the opponent started an AFK Check, the check will be disabled.

6.5 Submit Solution Hash

At the beginning of a new round, in order to submit the hash of the new solution for the new round, the CodeMaker can invoke the following method:

```
function newSolutionHash(address id, bytes32 solutionHash)
```

where:

- The argument value passed to `id` is the identifier of the match.
- The argument value passed to `solutionHash` is the salted hash of the solution produced using the keccak256 algorithm (see subsection 5.1).

If the upload terminated successfully, this method will emit the following event:

```
SolutionHashSubmitted(id, from)
```

where:

- The value of `id` is the identifier of the match.
- The value of `from` is the address of the player who uploaded the hash (thus is the address of the caller of this contract method).

In this case, as a side effect, if the opponent started an AFK Check, the check will be disabled.

6.6 Submit a Guess

In order to submit a guess, the CodeBreaker can invoke the following method:

```
function newGuess(address id, uint16 guess)
```

where:

- The argument value passed to `id` is the identifier of the match.
- The argument value passed to `guess` is the encoded guess to upload (see subsubsection 3.4.2).

If the upload terminated successfully, this method will emit the following event:

```
GuessSubmitted(id, from, guess)
```

where:

- The value of `id` is the identifier of the match.
- The value of `from` is the address of the player who uploaded the guess (thus is the address of the caller of this contract method).
- The value of `guess` is the encoded guess that has been uploaded.

In this case, as a side effect, if the opponent started an AFK Check, the check will be disabled.

6.7 Submit a Feedback

In order to submit a feedback, the CodeMaker can invoke the following method:

```
function newFeedback(address id, uint8 feedback)
```

where:

- The argument value passed to `id` is the identifier of the match.
- The argument value passed to `feedback` is the encoded feedback to upload (see subsubsection 3.4.2).

If the upload terminated successfully, this method will emit the following event:

```
FeedbackSubmitted(id, from, feedback)
```

where:

- The value of `id` is the identifier of the match.
- The value of `from` is the address of the player who uploaded the feedback (thus is the address of the caller of this contract method).
- The value of `feedback` is the encoded feedback that has been uploaded.

In addition to that, if the round ends after the submission of this feedback (due to the guess number limit that has been reached, or due to a correct guess), this method will emit also the following event:

```
RoundEnded(id, round)
```

where:

- The value of `id` is the identifier of the match.
- The value of `round` is the number of the round that is just ended.

In this case, as a side effect, if the opponent started an AFK Check, the check will be disabled.

6.8 Submit Final Solution

At the end of a round, in order to submit the solution of correspondent to this round, the CodeMaker can invoke the following method:

```
function uploadSolution(address id, uint16 solution, bytes4 salt)
```

where:

- The argument value passed to `id` is the identifier of the match.
- The argument value passed to `solution` is the encoded color combination that constitutes the solution to upload (see subsubsection 3.4.2). It has to be the same used for the calculation of the hash submitted at the beginning of the round.
- The argument value passed to `salt` is the same salt used for the calculation of the hash submitted at the beginning of the round.

If the upload terminated successfully, this method will emit the following events:

1. `SolutionSubmitted(id, from, solution)`

   This event is emitted to signal that the solution has been successfully submitted. Its arguments are:

   - The value of `id` is the identifier of the match.
   - The value of `from` is the address of the player who uploaded the solution (thus is the address of the caller of this contract method).
   - The value of `solution` is the encoded solution that has been uploaded.

2. `ScoresUpdated(id, creatorScore, challengerScore)`

   This event is emitted to signal that the scores have been updated. Its arguments are:

   - The value of `id` is the identifier of the match.
   - The value of `creatorScore` is the updated score of the creator.
   - The value of `challengerScore` is the updated score of the challenger.

3. Based on whether the game ends with the end of this round or not, the following events are also emitted:

   - `GameEnded(id)`

     This event is emitted in case the game ended because the last round of the match is terminated. The argument value of `id` is the identifier of the match.

   - `RoundStarted(id, round, codemaker, codebreaker)`

     This event is emitted to signal that the game didn't end and the start of a new round. Its arguments are:

     - The value of `id` is the identifier of the match.
     - The value of `round` is the number of the round that is just started.
     - The value of `codemaker` is the address of the player that will play as the CodeMaker in this round.

– The value of `codebreaker` is the address of the player that will play as the CodeBreaker in this round.

In this case, as a side effect, if the opponent started an AFK Check, the check will be disabled.

If instead the hash calculated using the uploaded solution and salt doesn't match, this method will emit the following events:

1. `PlayerPunished(id, player, reason)`

   This event is emitted to signal that the player has been punished. Its arguments are:

   - The value of `id` is the identifier of the match.
   - The value of `player` is the address of the player that has been punished (thus is the address of the caller of this contract method).
   - The value of `reason` is a message explaining the reason behind the punishment.

2. `MatchEnded(id)`

   This event is emitted to signal that the entire match is terminated. The argument value of `id` is the identifier of the match.

In this last case, as a side effect, the player whose address is different from the one included in the `PlayerPunished` event will receive the sum of the stake values payed by both players at the beginning of the match.

6.9 Dispute

In order for the CodeBreaker of the previous round to dispute one or more feedbacks submitted by the CodeMaker of the previous round, it can invoke the following method:

```
function dispute(address id, uint8[] feedbackIdx)
```

where:

- The argument value passed to `id` is the identifier of the match.
- The argument value passed to `feedbackIdx` is a list of valid feedback indexes of the past round that the user want to dispute. The indexes are assigned starting from zero and following the order of their submission.

Note that this method can be called by the old CodeBreaker only in the following situations:

1. Before submitting the hash of the solution for the incoming round (this because the old CodeBreaker will play the role of CodeMaker in the new round).
2. Before the time limit for the dispute is expired (see section 2).

If at least one of these conditions is not satisfied, the call to this function will be reverted.

In case the call is not reverted, this method will emit the following events:

1. `PlayerPunished(id, player, reason)`

   This event is emitted to signal that the player has been punished. Its arguments are:

   - The value of `id` is the identifier of the match.
   - The value of `player` is the address of the player that has been punished. This will correspond to the address of the old CodeMaker in case at least one of the feedbacks identified by the indexes specified is wrong w.r.t. its relative guess and the solution of the past round. Instead, if none of the feedbacks specified is wrong, this will correspond to the address of the old CodeBreaker.
   - The value of `reason` is a message explaining the reason behind the punishment.

2. `MatchEnded(id)`

   This event is emitted to signal that the entire match is terminated. The argument value of `id` is the identifier of the match.

In this case, as a side effect, the player whose address is different from the one included in the `PlayerPunished` event will receive the sum of the stake values payed by both players at the beginning of the match.

6.10 Request an AFK Check

In order to start an AFK Check, a player can invoke the following method:

```
function startAfkCheck(address id)
```

where the argument value passed to `id` is the identifier of the match. Note that this method can be called by a player only in the following situations:

1. If a player has already payed the stake, but the opponent doesn't.
2. If the CodeMaker doesn't provide the hash of the new solution for the incoming round. In this case only the CodeBreaker can call this method.
3. If the CodeMaker doesn't provide the feedback for the last guess that has been submitted. In this case only the CodeBreaker can call this method.
4. If the CodeMaker doesn't provide the final solution for the current round. In this case only the CodeBreaker can call this method.
5. If the CodeBreaker doesn't provide a new guess for the solution of the current round in case the round isn't ended with the last guess, and in case the CodeMaker has already submitted the feedback relative to the last guess. In this case only the CodeMaker can call this method.

If none of these conditions is satisfied, the call to this function will be reverted.

In case the call is not reverted, this method will emit the following event:

```
AfkCheckStarted(id, from, time)
```

where:

- The value of `id` is the identifier of the match.
- The value of `from` is the address of the player that requested the AFK Check (thus is the address of the caller of this contract method).
- The value of `time` is the number of seconds that the player has to wait before being able to stop the match for AFK.

6.11 Stop the Match for AFK

In order to stop the match for AFK, a player can invoke the following method:

```
function stopMatchForAfk(address id)
```

where the argument value passed to `id` is the identifier of the match. Note that this method can be called by a player only when:

1. A call to `startAfkCheck` has already been done (see subsection 6.10).
2. The time specified the event `AfkCheckStarted` emitted with the call to `startAfkCheck` is expired.
3. The opponent hasn't made any **correct** move in the meanwhile.

If at least one of these conditions is not satisfied, the call to this function is reverted.

In case the call is not reverted, this method will emit the following events:

1. `PlayerPunished(id, player, reason)`

   This event is emitted to signal that the player has been punished. Its arguments are:

   - The value of `id` is the identifier of the match.
   - The value of `player` is the address of the player that has been punished (thus is the address of the opponent of the caller of this contract method).
   - The value of `reason` is a message explaining the reason behind the punishment.

2. `MatchEnded(id)`

   This event is emitted to signal that the entire match is terminated. The argument value of `id` is the identifier of the match.

In this case, as a side effect, the player whose address is different from the one included in the `PlayerPunished` event will receive the sum of the stake values payed by both players at the beginning of the match.

In the rare event of an internal failure (it should never happen), this method will emit the following events:

1. `Failure(id, message)`

   This event is emitted to signal that an internal failure has occurred. Its arguments are:

   - The value of `id` is the identifier of the match.
   - The value of `message` is a message explaining the failure.

2. `MatchEnded(id)`

   This event is emitted to signal that the entire match is terminated. The argument value of `id` is the identifier of the match.

In this case, as a side effect, both players will be refunded with the stake value they payed initially.

6.12 Retrieve the Final Reward

At the end of the game it is necessary to check who has the highest score and to reward the winner (or both players in case of draw), the method is called like this:

```
function checkWinner(address id)
```

where the value of `id` id the identifier of the match. This function will revert if:

- The funds in the contract are insufficient (though unlikely a check is performed).
- The player that called this function is the CodeMaker and has not waited the dispute timeout. The reason for this is that the CodeBreaker must have the chance to review the last round and perform a dispute.

In the cases where the function doesn't revert two cases are possible:

1. One player has an higher score than the other thus he is declared the winner, the contract transfers the stake to that player, and the following event is emitted:

   ```
   RewardDispensed(id, to, reward)
   ```

   where:

   - The value of `id` is the identifier of the match.
   - The value of `to` is the address of the player that won the match and is being rewarded.
   - The value of `reward` is the reward amount in Wei dispensed to the player.

2. If instead the result of the match is a draw, the game stake will be divided by 2 (it is surely even) and both player are awarded their portion of the stake. In this case two `RewardDispensed` events will be emitted, one for each player.

Regardless of the result of the match the method execution will terminate with the emission of the event:

```
MatchEnded(id)
```

where the value of `id` id the identifier of the match, signaling the end of the game.