Tic Tac Toe using Ethereum Smart Contracts

Timo Hegnauer Lenz Baumann Cyrill Halter

2018
May

This report documents the implementation of the *Tic Tac Toe* game logic by means of an *Ethereum Smart Contract* written in the *Solidity* programming language. The smart contract should be compilable with the *SOLC* compiler version `>= 0.4.21`.
The source-code can be found here: `https://github.com/timohe/ethTicTacToe` along with instructions and further information on how to run using geth.

1 Functionality

The smart contract supports the following functionality:

1. The full *Tic Tac Toe* game logic is implemented entirely within the smart contract and can be accessed either via the *GETH* console or a custom web interface.
2. The smart contract is able to store and operate multiple games at the same time.
3. A betting system was implemented where each player has to pay a wager of 5 ether in order to join a game. The winner is awarded the entire pot of 10 ether. In case of a tie the pot is split up equally between the two players.
4. A business model was developed that allows for the exploitation of those people that are naive enough to spend 5 ether on a game of *Tic Tac Toe* without reading the contract.

2 Events

The we are using the following Events to trigger actions in the javascript:

```
event HostedGame(address sender ,string log);
event JoinedGame(address sender ,string log);
event Error(address sender ,string error);
event GameOver(address host ,string whoWon);
event TriggerBoardRefresh(address host);
```

These events are passing an address so the user can filter to only receive the event relevant to his games.

3 The Game Struct

The `Game` struct contains a set of variables used to define the state of a *Tic Tac Toe* game between two players.

```
struct Game
{
    address opponent;
    bool isHostsTurn;
    bool gameNotOver;
    uint turnNr;
    mapping(uint => mapping(uint => uint)) board;
}
mapping (address => Game) games;
```

In rough terms, a game consists of a player who hosts the game, an opponent and a game board. The `Game` struct stores the opponents address, as well as the game board, which is conceptualized as a 2d uint mapping. The host address is implicitly stored as the key to a mapping from host addresses to games. This implies that a player may host only one game at a time. The `Game` struct also keeps boolean variables indicating whose turn it is and whether the game has ended. The current turn is stored as a uint.

4 Hosting and Joining a New Game

In order to start a game between two parties, one of them needs to act as a host to initialize it in the smart contract. The second player can then join the hosted game.

```
function hostNewGame() payable rightAmountPaid public {
    clearBoard(msg.sender);
    Game storage g = games[msg.sender];
    g.gameNotOver = true;
    emit HostedGame(msg.sender, "successfully hosted Game!");
}
```

The `hostNewGame` method is called by the game's host to initialize a new game. Any old `Game` that was persisted in the smart contract for the host's address is first cleared out of storage. `clearBoard` is a helper method that deletes each variable in a game struct. After clearing, a new `Game` is initialized and the `gameNotOver` boolean is set to true, indicating that a game is in progress. Furthermore an event is passed to the user to inform him that he successfully hosted the game.

```
function joinExistingGame(address host) payable rightAmountPaid public {
    Game storage g = games[host];
    if(g.opponent == 0 && msg.sender != host)
    {
        g.opponent = msg.sender;
    }
    emit JoinedGame(msg.sender, "successfully joined Game!");
}
```

The `joinExistingGame` enables a second player to join a game hosted by another player. To do this, he must know the host's address. This method also checks whether a second player has already joined the game and whether the host is trying to play a game against himself, in which case it fails.

```
modifier rightAmountPaid {
    if(msg.value != pot){
        emit Error("You need to make a transaction of 5 eth...");
    }else{ _; }
}
```

The `rightAmountPaid` modifier is appended to both the `hostNewGame` and the `joinExistingGame` methods and ensures that a transaction of 5 ether is made by each player to the smart contract upon entering a game.

5 Playing a Move

The main *Tic Tac Toe* game logic is conceptualized in the `play`, the `youWon` and the `isTie` methods.

```
function play(address host, uint row, uint column) public{
    Game storage g = games[host];
    if(!g.gameNotOver){
        emit Error(msg.sender, "The game is Over");
        return;
    }
    uint player;
    if(msg.sender == host){
        player = 1;
    }
    else if(msg.sender == g.opponent){
        player = 2;
    } else{
        emit Error(msg.sender, "You are not part of this game");
        return;
    }
    if((g.isHostsTurn && player != 1) || (!g.isHostsTurn && player == 1)){
        emit Error(msg.sender, "Its not your turn! Wait for your opponent to play");
        return;
    }else{
```

```
            if(row >= 0 && row < 3 && column >= 0 && column < 3 && g.board[row][column] == 0){
                g.board[row][column] = player;
                g.turnNr ++;
                emit TriggerBoardRefresh(host);
                if(youWon(host)){
                    if(player == 1){
                        host.transfer(10 ether);
                        emit GameOver(host, "host");
                        g.gameNotOver = false;
                    }else{
                        g.opponent.transfer(10 ether);
                        emit GameOver(host, "opponent");
                        g.gameNotOver = false;
                    }
                    g.isHostsTurn = !g.isHostsTurn;
                    return;
                }

                if(isTie(host)){
                    host.transfer(5 ether);
                    g.opponent.transfer(5 ether);
                    g.isHostsTurn = !g.isHostsTurn;
                    emit GameOver(host, "tie");
                    g.gameNotOver = false;
                    return;
                }
                g.isHostsTurn = !g.isHostsTurn;
                return;

            } else {
                emit Error(msg.sender, "Your choice of field was not valid");
            }
        }
    }
```

The `play` method first performs a set of preliminary checks including whether the game is still in progress, whether the player trying to make a move is part of the game and it is his turn and whether the move is valid for the *Tic Tac Toe* playing field. If this is the case, the move is performed. The smart contract then evaluates whether the game has been won by the player making the move using the `youWon` method. If this is the case, the pot amount of 10 ether is transferred to the winner and the game is ended. Otherwise, the smart contract evaluates whether the game has resulted in a tie using the `isTie` method. In this case, the pot is divided equally between the host and the opponent and the game is ended. If the game does not end, the `Game` struct's `isHostsTurn` variable is switched and the game continues.

```
    function youWon(address host) internal view returns (bool didYouWin){
        Game storage g = games[host];
        for (uint i; i < 3; i++){
```

```
        if(g.board[i][0] != 0 && g.board[i][0] == g.board[i][1] &&
                g.board[i][1] == g.board[i][2] ){ return true; }
        if(g.board[0][i] != 0 && g.board[0][i] == g.board[1][i] &&
                g.board[1][i] == g.board[2][i]){ return true; }
    }
    if(g.board[0][0] != 0 && g.board[0][0] == g.board[1][1] && g.board[1][1]
            == g.board[2][2]){ return true; }
    if(g.board[2][0] != 0 && g.board[2][0] == g.board[1][1] && g.board[1][1]
            == g.board[0][2]){ return true; }
    return false;
}
```

The `youWon` method checks whether a winning constellation has been reached in a game. To achieve this, it checks both the rows and the columns, as well as both diagonals of the `Game` struct's `board` mapping for matching numbers. Since this check is always performed after a player has made a move and since therefore only he has the possibility of winning the game at that point, the method does not have to discern between player wich reduces the complexity of the method and reduces gas cost.

```
function isTie(address host) internal view returns (bool isItATie){
    Game storage g = games[host];
    if(g.turnNr > 8){ return true; }
}
```

The `isTie` method evaluates whether the game has ended in a tie by checking if the game has reached its ninth move without any one of the two parties winning.

6 Exploiting the Rich and Naive

5 ether is a lot to spend on a single game of *Tic Tac Toe*. It can therefore be expected that only those with money to spare (or rather throw away) will be willing to participate in a game with stakes as high as these. To profit from this, the `withdraw` method allows the owner of the contract to extract the funds that belong to the smart contract. This is, for one, to gain access to ethers that may have been left over from a game that was never ended, but also to blatantly steal from those rich enough to play a game of *Tic Tac Toe* for 5 ether with the intent of redistributing the gained money in a robbin-hood-esque fashion.
Furthermore, this contract has an educational value for people blindly trusting often called "trust-free" blockchain applications and creates awarness of the consequence of such behaviour.

```
constructor () public {
    owner = msg.sender;
}
```

The contract's constructor sets the owner of the contract

```
function withdraw() public onlyOwner {
    require(address(this).balance > 0);
    owner.transfer(address(this).balance);
}
```

The `withdraw` method enables the withdrawal of the funds in the contract if there are any.

```
modifier onlyOwner() {
    require (msg.sender == owner);
    _;
}
```

The `onlyOwner` modifier is appended to the `withdraw` method and ensures that only the contract's owner may execute the method.