Challenge task - TicTacToe

Group Rio
Lukas Eisenring 14-709-455
Fabian Leisibach 13-931-175
Stefan Würsten 14-725-931

Abstract. The challenge task in the Blockchains and Overlay Networks lecture (FS18) at the University of Zurich was the implementation of a blockchain based TicTacToe. This report gives and overview on the implementation and shows an in depth documentation of the smart contract and its operation.

Keywords: Blockchains and Overlay Networks, Ethereum, Blockchain, TicTacToe, DApp, Code documentation

1 Requirements of the challenge Task

The TicTacToe should be implemented in the size of a 3x3 board. The game can be text or graphics based. The Smart Contract can be deployed in a private testnet or in the Ropsten testnet. To run the game just using one node with multiple addresses is sufficient. An address cannot play against itself. The implementation of the Smart Contract (SC) must be able to store more than one game and the SC must automatically pay the bet to the winner or divide in a case of draw.

2 Our implementation

Our implementation of the TicTacToe follows the following assumptions by the challenge task itself:

- You can use just one node in the blockchain with multiple addresses.
- An address cannot play against itself.

We also came up with further assumptions:

- Two players sit next to each other in front of the computer and play in the same browser window.
- Addresses are already logged in.

Following a short description of our TicTacToe can be found. The idea of our game is, that the two players sit next to each other in front of the computer and play in the same browser window. Our implementation runs on a private testnet and one node, but can also run on all Ethereum based networks. On the first screen, all the addresses of the node are visible. The first player can select his address first, then the second player can select his address. Then a slider bar appears, where the bet can be defined. With pressing the start button the game is started. If the game participants and their corresponding addresses have enough Ether, the bet for each player is deposited. If there are not sufficient funds, an error message is displayed. One can then try to redeposit or to abort. After the successful deposition, the game field loads. The player who gets to play first is highlighted with green. If one person wins, a message is displayed with the winners address and the amount that was won then one is taken back to the lobby. If both tie, then the Ether is returned to both and one is taken back to the lobby.

The technical aspect of our implementation is split into two parts. On one side we have the Smart Contract on the other side the frontend, which makes use of the official Ethereum Javascript API, that allows us to retrieve and send data from our Smart Contract to our graphical interface. Once the contract is mined the contract address can be updated in a configuration file. With the Ethereum Javascript API we also use jQuery and jQuery Ui. Due do the fact that mining can take longer and the API is based on asynchronous callbacks debugging is more complex.
To simplify the process of running the TicTacToe we included an installation guideline as a Readme.md file in the source code.

3 Smart contract

The smart contract consists of the game object in which all the information about one game is stored. There is a counter, that indicates the number of games and an array where all the games are stored.

```
contract TicTacToe {

        struct Game {
                uint incremental;
                address[2] players;
                uint moveCounter;
                address[9] board;
                uint deposit;
                bool[2] depositPayed;
                bool finished;
        }
        uint counter = 0;
        mapping (uint => Game) games;
        event GameObject( uint gameId, uint incremental, uint moveCounter );
...}
```

Once the game is created, the player pay their deposit. This function is called when the first player has sufficient funds even when the second does not. If the first has not sufficient funds then the deposit function is not called at all. If the bet is available is checked by the frontend. (In the case only the first player has paid and the 2nd not due to insufficient funds, the deposited money is returned to the first player.)

```
function deposit( uint gameId ) public payable {
                if ( msg.sender == games[gameId].players[0] && msg.value ==
games[gameId].deposit && !games[gameId].depositPayed[0] ) {
                        games[gameId].depositPayed[0] = true;
                        games[gameId].incremental++;
                }
                else if ( msg.sender == games[gameId].players[1] && msg.value ==
games[gameId].deposit && !games[gameId].depositPayed[1] ){
                        games[gameId].depositPayed[1] = true;
                        games[gameId].incremental++;
                }

                GameObject( gameId, games[gameId].incremental, games[gameId].moveCounter );
        }
```

Once the funds deposited a turn can be made. Preconditions such as that the current player is the message sender and if field id is valid are checked. A field will be assigned to the players address but only if the game is running and the field is empty (the field is empty if has not been already selected by one of the players during the game). After each turn it will be checked if the game has a winner.

```
function play ( uint gameId, uint fieldId ) public {
                // check if fieldId is a valid number
                require( fieldId >= 0 && fieldId <= 8 );
                // check if its your turn
                require( getCurrentPlayer( gameId ) == msg.sender );
                // only if game is running and field is empty, your turn is valid
                if( isRunning( gameId ) && isFieldEmpty( gameId, fieldId ) ){
                        games[gameId].board[ fieldId ] = msg.sender;
                        games[gameId].incremental++;
                        hasWinner( gameId );
                        games[gameId].moveCounter++;
                }
        }
```

Following are functions used in the code snippets above. The one below checks if game has been won by one of the players. It checks all eight possibilities. But there are also other internal functions such as one that checks if a game is still running or one, to get the current player.

```
function hasWinner ( uint gameId ) private {
                bool row1 = games[gameId].board[0] == games[gameId].board[1] &&
(games[gameId].board[0] == games[gameId].board[2]) && !isFieldEmpty( gameId, 0 );
                bool row2 = games[gameId].board[3] == games[gameId].board[4] &&
games[gameId].board[3] == games[gameId].board[5] && !isFieldEmpty( gameId, 3 );
                bool row3 = games[gameId].board[6] == games[gameId].board[7] &&
games[gameId].board[6] == games[gameId].board[8] && !isFieldEmpty( gameId, 6 );
                bool col1 = games[gameId].board[0] == games[gameId].board[3] &&
games[gameId].board[0] == games[gameId].board[6] && !isFieldEmpty( gameId, 0 );
                bool col2 = games[gameId].board[1] == games[gameId].board[4] &&
games[gameId].board[1] == games[gameId].board[7] && !isFieldEmpty( gameId, 1 );
                bool col3 = games[gameId].board[2] == games[gameId].board[5] &&
games[gameId].board[2] == games[gameId].board[8] && !isFieldEmpty( gameId, 2 );
                bool dia1 = games[gameId].board[0] == games[gameId].board[4] &&
games[gameId].board[0] == games[gameId].board[8] && !isFieldEmpty( gameId, 0 );
                bool dia2 = games[gameId].board[2] == games[gameId].board[4] &&
games[gameId].board[2] == games[gameId].board[6] && !isFieldEmpty( gameId, 2 );
                // 3 in row, column or diagonal
                if( row1 || row2 || row3 || col1 || col2 || col3 || dia1 || dia2 ){
                        // he gets 2x the deposit
                        address winnerAddress = getCurrentPlayer( gameId );
                        winnerAddress.transfer( 2 * games[gameId].deposit );
                        games[gameId].finished = true;
                }
                // no winner / draw
                else if( games[gameId].moveCounter >= 8 ){
                        // both get 1x the deposit
                        games[gameId].players[0].transfer( games[gameId].deposit );
                        games[gameId].players[1].transfer( games[gameId].deposit );
                        games[gameId].finished = true;
                }
        }
```

4 Improvements

For a next step we were thinking about what we could enhance to make our TicTacToe even better. At the moment we don't have a timeout function. Meaning, a player could just stop playing once he realizes that he loses. Once a player stops, the game will just keep running, but not abort. For our implementation we believe this is sufficient since both players sit in front of the same screen, so the players make sure the game is played until the end. But if in a next step we would implement, that the game can be played from different screens, which would be the next step for our implementation, the above mentioned timeout would have to be added. Besides that, also our code would have to be optimized to reduce gas consumption.