Blockchain and Overlay Networks - Challenge Task 2018

Alexander Hofmann, Nico Strebel, Ivan Taraca

May 16, 2018

Chapter 1

Introduction

This is a report for the challenge task in the lecture Blockchains and Overlay networks.

The goal of this task is to design and develop the game Tic-Tac-Toe as a Decentrallized Application (DApp) running in the Ethereum blockchain, in the form of a smart contract (SC). Through this DApp, users can challenge each other and bet ethers.

The full source code to our project can be found in the Github repository: `https://github.com/a-a-hofmann/D-TicTacToe`

1.1 Requirements

For this project, the following four requirements were defined:

1. The fully functional Tic-Tac-Toe game (text or graphics based) must run entirely within Smart Contracts (SC).
2. The SC must be able to store more than one game, i.e., more than one game can be played at same the time.
3. The SC must automatically pay the bet to the winner or divide the bet in the case of a draw.
4. Report documenting the Smart Contract and its operation.

1.2 Assumptions

During the design and development of this project, the following facts were assumed:

- The SC can be deployed in a private testnet.
- The SC may run on a single node with multiple addresses.
- An address cannot play against itself.
- Player X always starts.
- The players play honestly. E.g. if a player knows he is going to lose, he will continue to play and allow the game to end. The SC implements no form of time constraints, and thus, it would be possible to stall the game indefinitely. In this case, no player would receive the bets.

In the rest of the document, we show how the implementation fulfills, and goes beyond, the requirements set.

Chapter 2

Design and Implementation

2.1 Architecture

The DApp can be seen as a two-tier application. The backend is a smart contract (SC), thus its logic and storage reside on the Ethereum blockchain. A web application (web UI), which implements only presentation logic, allows a user to interact with the backend in a user-friendly manner. The web UI and the SC communicate in two ways:

1. The UI initiates an action, to which the SC responds.
2. The UI reacts to SC events emitted as result of other function calls, e.g. a new game was created.

In the following section we describe the details regarding the smart contract. The web UI is described afterwards.

2.2 Smart Contract

We first describe the functionality and design of the smart contract. We attempted to make the most out of the features offered from solidity and smart contracts. Thus, extensive use of events and function modifiers was made. Examples include modifiers such as the `mayJoinGame` decorator or events such as `GameStarted` or `GameUpdateEvent`. In the following subsections, we shall attempt to show examples of our use of such features, some will, however, be omitted for brevity.

In addition to the base requirements, a few admin functionalities were added (enabling/disabling fee mechanism). To protect these functions, the SC extends Openzeppelin's [1] the `Ownable` contract.

2.2.1 Game Phases Description

We begin by defining the possible states in which a game can be:

- Betting phase: player one creates a game and sets a bet amount, which must be matched by an opponent in order to play.
- Ongoing phase: an opponent has matched player one's bet and the game has started. During this phase, players alternate placing their marks on the board..
- Game over phase: this phase is further divided in three cases (in all cases, a commission may apply):
  - One of the players has won. The winner wins the pot.
  - The game ends in a draw. The pot is split back to both players.
  - One of the players has forfeited the game. The other players wins.

[1] `https://openzeppelin.org`

2.2.2 Game struct

The main data structure of the smart contract is the game struct. This structure groups all information regarding a single game in a single object. This object is then saved in a public array of the contract. This allows anyone, at anytime, to access the entire history of the games since the contract's deployment, including participating players, results, and bets placed. By using this array, the SC can host any number of concurrent Tic-tac-toe games, fulfilling the second requirement.

Each game object, contains the following information (Listing A.1):

- Game id.
- Participating players.
- The bet placed by each player.
- The current game phase.
- Whose turn it is to play next.
- The board as a flat (one-dimensional) array of addresses. Each cell contains either the address zero (empty cell), or one of the participating player's address.
- Fee: the smart contract allows the option to enable commissions, in which case a percentage of the bets is withheld inside the SC. In the case that no fees are enabled, the fee is set to zero. Further details on this topic can be found further down in the report.
- The winner of the game. In case of an ongoing game or a draw, the field is set to zero.

2.2.3 Betting phase

The betting phase can be further divided in two steps: (i) creating a game and (ii) joining a game.

Creating a game

In order to create a new game, a player sends an amount of ether to a `payable` function, the `createGame` function. The sender of the message and the amount of ether sent with the transaction are saved inside the newly created object as player X and betting amount, respectively, of the game. Additionally, if there are any fees, these are also saved into the game object at creation.

This is done prematurely to avoid the following scenario:

1. Two players start a game. The `commissionEnabled` field, which is public, is set to false, so the players expect that the winner of the game, shall receive the full amount of the bets, or that it is split two-ways in case of a draw.
2. While playing, the owner enables the `commissionEnabled` fee.
3. The players, finish their game, and since the field is set to true, they only receive a part of the entire pot, the rest being withheld as a commission.
4. The players feel ripped off (rightly so).

Finally a `NewGame` event is emitted, which can be caught by client applications to update their list of games.

```
function createGame() public payable {
    require(msg.value > 0);
    address[] memory _board = new address[](9);
    uint _fee = commissionEnabled ? fee : 0;
    uint gameId =
        games.push(Game(games.length, msg.sender, address(0), msg.value, Status.BET, msg
            .sender,    _board, _fee, address(0))) - 1;
    emit NewGame(gameId);
}
```

Listing 2.1: createGame function.

Joining a game

In order to join a game, the opponent sends an amount and the id of the game he wishes to join. But first, he must fulfill certain preconditions. Specifically, the following aspects are ascertained before allowing a player to join a game:

1. The game exists.
2. The game is in betting phase.
3. The game has no other opponent
4. The player is not attempting to play against himself/herself.
5. The amount sent from the opponent is enough to cover the bet placed by the player that created the game. Note that we only check whether the amount is equals or greater. It is up to the opponent/client application to check that only the exact amount needed is sent.

Once the SC has checked the preconditions, it is simply a matter of updating the opponent and status fields and emitting the `GameStarted` event. Thus, all subscribing clients are notified that the game has started.

```

function joinGame(uint gameId) public payable mayJoinGame(gameId) {
    games[gameId].playerY = msg.sender;
    games[gameId].status = Status.ONGOING;
    emit GameStarted(gameId);
}

modifier mayJoinGame(uint gameId) {
    // 1) Game exists.
    require(gameId < games.length);
    // 2) Game is in betting phase.
    require(games[gameId].status == Status.BET);
    // 3) Game has empty seats.
    require(games[gameId].playerY == address(0));
    // 4) Check that the opponent is not the same as the creator of the game.
    require(games[gameId].playerX != msg.sender);
    // 5) Check if the bet is sufficient in order to join the game.
    require(msg.value >= games[gameId].bet);
    _;
}
```

Listing 2.2: joinGame function.

2.2.4 Ongoing phase: Playing a game

In order to make a move, two inputs are necessary: the game `id` and the cell `index` to fill with the player's mark.

We begin by first checking the preconditions. The `gameExists, gameIsOngoing, and isParticipating` modifiers should be self-explanatory. The `checkInput` decorator checks three things:

1. Is the `index` valid, i.e. is it in the range [0, 9)?
2. Is the cell empty, i.e. does it contain the address zero?
3. Is it the player's turn?

If these conditions are fulfilled, we store the player's address in the cell. Afterwards, we check if the game is over, either because a player won, or because of a draw. Otherwise, we set the next player, and fire the `GameUpdateEvent` which the client side applications can catch to update the UI with the updated game state.

```

function playerMove(uint gameId, uint index) public gameExists(gameId) gameIsOngoing(
    gameId) isParticipating(gameId) checkInput(gameId, index) {
    games[gameId].board[index] = msg.sender;

```

```
    address winner = checkWinner(gameId);
    if (winner != address(0)) {
        gameOverWinnerCase(gameId, winner);
    } else if (checkDraw(gameId)) {
        gameOverDrawCase(gameId);
    } else {
        if (msg.sender == games[gameId].playerX) {
            games[gameId].whoseTurn = games[gameId].playerY;
        } else {
            games[gameId].whoseTurn = games[gameId].playerX;
        }
    }
    emit GameUpdateEvent(gameId, games[gameId].board, games[gameId].whoseTurn, winner);
}

modifier checkInput(uint gameId, uint index) {
    require(index < 9);                    // Valid index.
    require(games[gameId].board[index] == address(0));  // Cell is empty.
    require(games[gameId].whoseTurn == msg.sender);     // Own turn.
    _;
}
```

Listing 2.3: playerMove function. This function both allows players to make their move, as well as checks the game over conditions.

2.2.5 Game over phase

A game is over when either one of the two players managed to get three marks over a horizontal, vertical or diagonal row, one player forfeited the game, or the game ended in a draw.

Player won After each move, we check whether a player has won. For Tic-tac-toe there are only 8 winning conditions. Instead of allocating space for an array to contain these permutations, we simply hard-code them and check each one, one after the other. The `checkPattern` then checks for each triple of indexes, whether there is a winner.

```
function checkWinner(uint gameId) internal view returns(address winner) {
  address[] memory board = games[gameId].board;

  winner = checkPattern(board, 0, 1, 2);
  if (winner != address(0)) return winner;
  winner = checkPattern(board, 3, 4, 5);
  if (winner != address(0)) return winner;
  winner = checkPattern(board, 6, 7, 8);
  if (winner != address(0)) return winner;
  winner = checkPattern(board, 0, 3, 6);
  if (winner != address(0)) return winner;
  winner = checkPattern(board, 1, 4, 7);
  if (winner != address(0)) return winner;
  winner = checkPattern(board, 2, 5, 8);
  if (winner != address(0)) return winner;
  winner = checkPattern(board, 0, 4, 8);
  if (winner != address(0)) return winner;
    winner = checkPattern(board, 2, 4, 6);
  if (winner != address(0)) return winner;

  return address(0);
}

function checkPattern(address[] board, uint8 x, uint8 y, uint8 z) internal pure returns(
    address winner) {
  if (board[x] != address(0) && board[x] == board[y] && board[x] == board[z]) {
    return board[x];
  }
  return address(0);
}
```

Listing 2.4: checkWinner and checkPattern function.

Draw The draw case is simpler. The game is a draw if all cells have a mark, and there is no winner. Whether there is a winner is implicit in the control flow of the `playerMove` function (Listing 2.3).

```
function checkDraw(uint gameId) internal view returns(bool draw) {
  address[] memory board = games[gameId].board;
  draw = true;
  for(uint8 i = 0; i < board.length; ++i) {
    draw = draw && board[i] != address(0);
  }
  return draw;
}
```

Listing 2.5: checkDraw function.

Payout Depending on whether the game was won, or it ended in a draw, the SC will proceed to calling the `gameOverWinnerCase` or the `gameOverDrawCase`, paying the winner or splitting the bet between the players respectively. The game's status and winner fields are updated, the winner is payed, and the clients are notified via the `GameWon` event, in case of a win. By calling these functions inside the `playerMove` function, the SC fulfills requirement 3). For details, refer to Listing A.2 in the Appendix.

2.2.6 Forfeiting a game

At any moment during the betting and ongoing phases of a game, the game can be forfeited. In this case two possible outcomes are possible.

During the betting phase, when no opponent has yet joined, the player which created the game, can quit and receive its full bet back.

If the game has already started and is not yet over, a player may forfeit the game. This is equivalent to losing the game.

2.3 Web UI

The web UI is a reactjs & redux app. It only implements presentation logic, and simplifies interacting with the SC through the UI, thus the entirety of the logic runs fully in the SC. The UI uses web3js in order to communicate with the Smart Contract. It does so by calling the smart contracts functions, as well as subscribing to solidity events. A few screenshots can be found in the appendix.

The UI also allows the contract's owner to access the functions in the SC marked with the `onlyOwner` modifier, but only if the current account matches that of the owner (also a public field of the SC). Through a sort of admin interface, the owner can enable/disable fees, and withdraw the fees accumulated in the SC.

2.4 Tools Used

We used the Truffle Framework [2] to help us with the compilation and deployment of the smart contract and to automatically move the abi (application binary interface) to the UI in order to easily instantiate the SC in the front-end with web3js. Additionally, we used Ganache [3] (also part of the Truffle Framework), an easy to deploy blockchain to increase development speed. The truffle suite of software and reactjs are both release under the permissive MIT license. Web3js under LGPL.

[2] `http://truffleframework.com/`, Your ethereum swiss army knife.
[3] `http://truffleframework.com/ganache/`, One click blockchain.

Appendices

Appendix A

Additional Code Listings

```
struct Game {
  uint gameId;
  address playerX;
  address playerY;
    uint bet;
  Status status;
    address whoseTurn;
  address[] board;
    uint fee;        // Fee as percentage of total.
  address winner;
}
```

Listing A.1: Game struct.

```
1  function gameOverWinnerCase(uint gameId, address winner) internal {
2    games[gameId].status = Status.DONE;
3    games[gameId].winner = winner;
4    uint winnings = games[gameId].bet * 2;
5    if (games[gameId].fee > 0) {
6      uint commission = winnings / 100 * games[gameId].fee;
7      winnings = winnings - commission;
8      balance = balance + commission;
9    }
10   payout(winner, winnings);
11   emit GameWon(gameId, winner);
12 }
```

Listing A.2: gameOverWinnerCase function. Lines 5-9 might be of interest to the reader, since they show how the commission mechanism works. The analogous function for the draw case is omitted for brevity.

```
function gameOverDrawCase(uint gameId) internal {
  games[gameId].status = Status.DONE;
  uint winnings = games[gameId].bet;
  if (games[gameId].fee > 0) {
    uint commission = winnings / 100 * games[gameId].fee;
    winnings = winnings - commission;
    balance = balance + (2 * commission);
  }
  payout(games[gameId].playerX, winnings);
  payout(games[gameId].playerY, winnings);
  emit GameDraw(gameId);
}
```

Listing A.3: gameOverDrawCase function. The bet, minus fees are split evenly between the players.

Appendix B

Screenshots

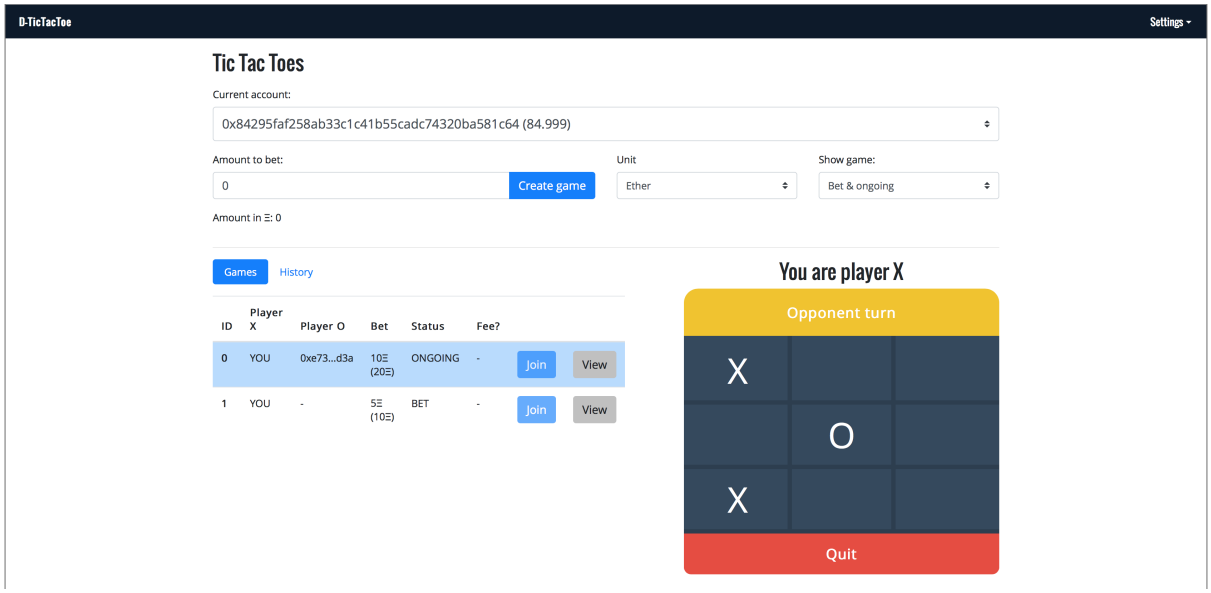

Figure B.1: List of games and an ongoing game.

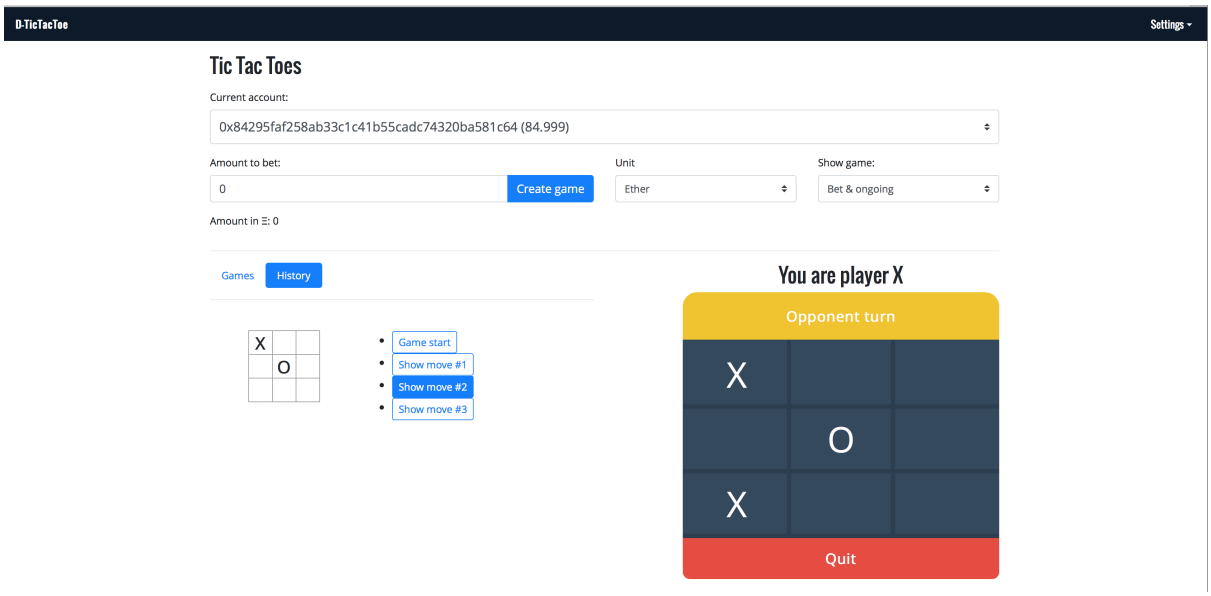

Figure B.2: Past moves and an ongoing game. Thanks to solidity events, it is easy to reconstruct all player moves for a game.

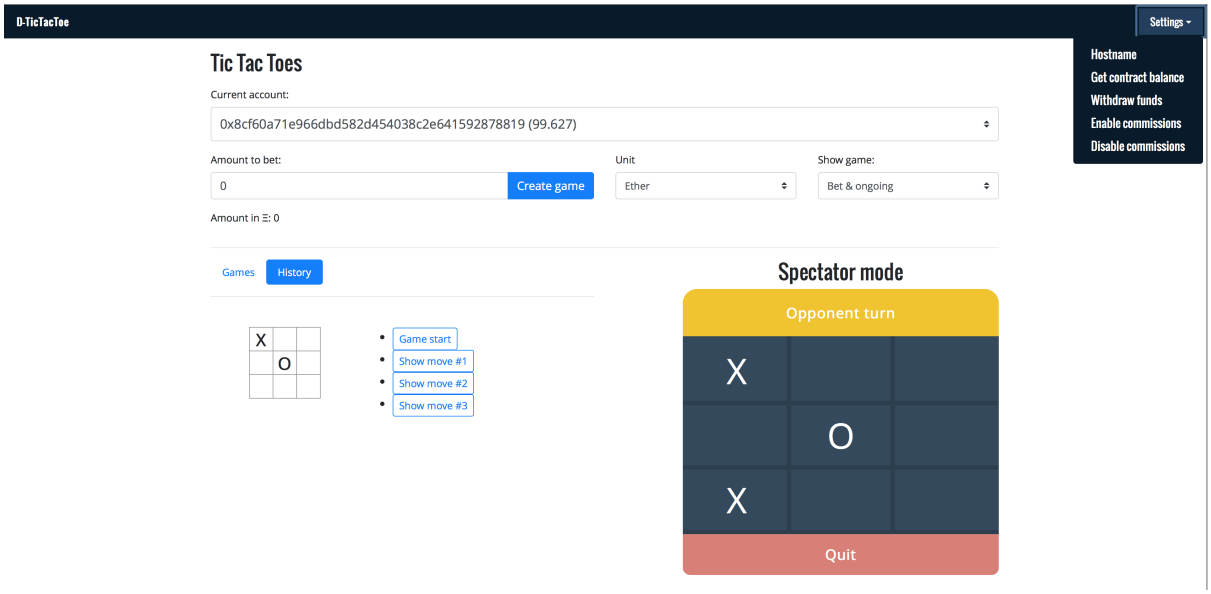

Figure B.3: Admin menu. Only visible when using the contract's owner account.