Report

Introduction

In the following we describe the implementation of a DApp that implements a Tic Tac Toe game. The game is played by two designated players. The players can bet on the game (in ether) and get paid out the bets at the end of the game. The game is modeled by a smart contract and represented by a web application front end.

Tools

There are several options available to develop a smart contract in the Solidity language for the Ethereum blockchain. One can use a publicly available testnet such as the Ropsten testnet[1] or set up their own private testnet. The process to setting up a private testnet for the Ethereum Client written in Go (geth) is described in detail here[2]. Alternatively, when setting up a private testnet, there are frameworks available that provide such a private testnet. In this project we use such framework named Truffle Suite[3]. This framework allows to compile smart contracts to their JSON interface (including the ABI of the contract) and to deploy the compiled contracts onto the Ethereum blockchain - either on a public or private network.

The Truffle Suite comes bundled with Ganache[4], a tool to launch a local development blockchain server. It allows to interact with the blockchain via a GUI or a CLI version. Also Ganache makes it possible to monitor all the transactions as well as mined blocks.

When using the Truffle Suite in combination with Ganache to develop smart contracts in Solidity, accounts are automatically created, unlocked, and transactions are signed. Furthermore blocks are mined instantly. This simplifies the development process considerably and makes the interactions of a developer with the blockchain more convenient.

Approach

In this challenge task the DApp (or smart contract) is required to be able to store more than one game, so that multiple games can be played simultaneously. We discussed several approaches to implement a solution that satisfies this requirement. Finally, we choose to implement a single smart contract that stores all the games as structs in a mapping. The major advantage is, that users interact with a single contract which results in less overhead. We also implemented a small proof of concept, where we setup a factory contract (hub) that deployed a game contract, whenever a new game is created. Even though, this simplified

[1] https://github.com/ethereum/ropsten [visited April 2018]
[2] https://github.com/ethereum/go-ethereum/wiki/Private-network [visited April 2018]
[3] http://truffleframework.com/ [visited April 2018]
[4] http://truffleframework.com/ganache/ [visited April 2018]

the logic of the contract, e.g. game id does not have to be tracked anymore, it resulted in higher gas prices for transactions. In the end, we dropped this approach, because it is too expensive (wrt. to transaction costs) to deploy a new contract to the blockchain for every game.

Implementation

We implemented a smart contract as decentralized application (DApp) and a frontend. In the following we will explain both parts.

Smart Contract

The DApp consists of single smart contract written in Solidity. Specifically, we use the latest version of Solidity that is available, i.e. 0.4.23. We use a struct to represent a game. Each game has an id (sequential number), a bet size, a board, an active player, a player X, a player Y, a turn counter, and a state.

The board is a fixed-size two dimensional array of addresses. Initially, all cells in the board default to address(0), indicating the cells are not occupied by a player yet. The variables player X, player Y and active player are also of type address. The id and bet size are of type uint, which defaults to an unsigned integer of 256 bytes. Similarly, turn counter is also an unsigned integer but only 8 bytes, because the number of turns is limited to 9 (assuming a board dimension of 3 and a board size of 3 x 3 = 9). We use this variable to check if the game ended with a draw. Active is a boolean variable keeping track of the game state.

To support multiple games being played at the same time, we have a mapping of game Ids to games. Additionally, we keep track of the number of games in a variable game counter.

Our smart contract exposes only three public methods, which can be called by users. First, there is a method to create a new game. It takes no parameter, but is payable. In other words a user calling this function can (optionally) send an amount of ether which will be deposited in the balance of the smart contract. Calling this method adds a new game to our games mapping (and increases the game counter). It sets the id, bet size and player X, where the bet size is the amount of ether a user sent with the transaction and stores the sender address as player X. Finally, it emits a event indicating the game has been created with the id of the game.

Second, there is a method to join an existing game. It takes a single parameter game id and is also payable. It uses modifiers to validate the user input for the specified game. It checks if the game is not full, if the sender of the transaction is not already stored as player of this game, and if the amount sent with the transaction is valid, i.e. at least the amount the first player has sent when creating the game. If any of these requirements are not met, the transaction will be reverted. If the player sent an amount of ether that is larger then the bet size, then the overpaid amount will be refunded immediately after the method has been executed. The method body sets player O to the sender address, sets the state of the game

to active and (randomly*) selects a first player to start the game. Finally, we emit an event indicating the game is active now with the game id.

Third, there is a method to place a mark. It takes a game id, a row and a col. Once again, we validate the user input for the specified game. If the game is not active (either not started, or already over), or if the sender of the message is not the active player, or if the specified position is already occupied, then the transaction is reverted. Otherwise, the specified position is set to the address of the sender and the turn counter is incremented.

After each turn, we check if the game is over (win or draw). If the game is over, we emit an event indicating this fact with the game id and the final board state. To check if the game is over, first we check if there is a win. Since we know the row and col which changed, we check the current row and col. If the changed cell is in the diagonal and/or anti-diagonal we also check these. We never check all possibilities only the once we have to using the domain knowledge of knowing the changed position. If there is no win, then we simply check if the turn counter is equal to the board size (9) to check for a draw. If the game is not over, we simply toggle the active player and emit an event indicating a game move with the game id, the board, and the active player.

Frontend

As a frontend for users to interact with the smart contract, we implemented a web application using React[5] together with Web3.js[6]. When the application is first launched, we extract the Application Binary Interface (ABI) and the address of the deployed contract form the JSON interface. Next Web3 is used to attach to the deployed contract. We subscribe to all events from the contract and set up handlers accordingly to deal with each event. Finally, we create a new game with two random players with the default bet size of 0.1 ether. This means when the web application is initially launched a default game is created with random accounts for the two players and a default bet size.

In the center of the screen we display the board of the currently selected game. Below, we display information about the players and their balances. In our solutions both players play using the same web application, it is assumed that both players use the same device. We indicate which of the two players is active and the balances are updated automatically. When the game is over, we display an alert indicating the result. We present the user with an option to quit or to create a new game.

If a user decides to create a new game, they will be presented with a dialog, containing a form to configure the new game. In this case the user is able to choose the accounts for player X and player O from a predefined list of ten accounts. Furthermore the user is able to specify a bet size for the next game. If the form is submitted the game will be set up and displayed.

[5] https://reactjs.org [visited April 2018]
[6] http://web3js.readthedocs.io/en/1.0/index.html [visited April 2018]

During a game, a user at any time, is able to create a new game, and also to switch between all active games via a drop down menu in the navigation bar.

Limitations

We can identify two limitations in our approach. First, we want to randomly select a player to start a game. Implementing randomness in smart contracts is not trivial. In our approach we rely on the block number, which we divide by 2 (the number of players) and use the remainder to determine which player can move first. However, the block number can be influenced by the miners and therefore our approach is not ideal.

Second, if a player at some point decides to stop playing the bets will be locked in the balance of the contract. They will not be refunded. This problem could partially be solved by implementing a time limit for turns. We could implement this again using the block number and if a user takes longer to make a move than the turn limit, the bets could be refunded to both players.

Conclusion

In carrying out the challenge task, which its focus is to implement a decentralized application running on Ethereum blockchain by developing a Tic Tac Toe game with a betting system that is all embedded in a Smart Contract. Truffle suite development framework was used and installed, Truffle Suite helps us to take care of built in smart contact compilation, linking deployment and binary management as well as automating contract testing for rapid development. Truffle Suite comes bundled with a dependency package called Ganache. Ganache was installed and enabled us to develop smart contracts, create accounts, unlock, and inspect transactions as well as blocks that were mined. For the frontend, we implemented a web application using React along with Web3.js. React a JavaScript library for building user interfaces and Web3.js is an ethereum JavaScript API that can be used in interacting with remote or local Ethereum node.

When the application is launched, the extracted Application Binary Interface (ABI) and the address of the contract will be deployed from the JSON interface. As part of what were experimented on, we enabled more than one game to be stored in order to play multiple games simultaneously, we concluded on using a single smart contract by storing the games as structs in a stated mapping. We simulated scenarios part of which were to start a new game, where a user will be requested to select accounts for player X and player O from the already generated list, and then place a bet price and the game is set. Once the game is set, a mark X or O (depending on the first player) is placed on any of the fixed 3 by 3 cells of the Tic Tac Toe board. After each turn, the smart contract checks if there is a win, a draw or moves left. If there is a win, the winner, either player X or Player O will be declared the winner and the balance on the smart contract will be transferred automatically to the winner's account and the game ends. If it is a draw, it declares the game a draw, return the balance to each player's account and the game ends; if we have move(s) left the game continues only if it is neither a draw nor a win.

Also we implemented some functions that validated the state of the current game, the input, the bet prices, the players and occupied positions. In addition, it is possible to create a new game and switch between existing games and play. Key notable limitation observed in plain sight was that, the randomness created in this game by relying on the block numbers to select the first player is not the best practice and thus, the blocks can easily be influenced by a miner and also, funds are locked on the smart contract and cannot be refunded if a player or both players decide to stop the game halfway. However, this can be solved by implementing a time limit on the smart contract.