Project Report - AllInCasino

Distributed Systems

Pablo Díaz Viñambres, Pablo Landrove Pérez-Gorgoroso

December 14, 2023

1 Introduction

In this report we present how our idea for an online casino in the Ethereum blockchain was implemented. Our resulting website, AllInCasino, uses Smart Contracts for all user actions in a typical game. This ensures certain desirable properties to the players:

1. **Fairness**: Smart Contracts residing in the Ethereum network are public, transparent and inmutable. Moving all the important decisions and money handling inside those contracts gives users a fair experience when gambling.
2. **Anonymity**: The usage of account addresses as usernames in the games allows users to achieve a high degree of anonymity that traditional sites don't allow for.
3. **Trust**: Solidity Smart Contracts are not governed by a single authority, and instead reside in the decentralized Ethereum Blockchain.

For the Proof of Concept of the idea, we implemented three games in the casino:

1. **"Krazy" Dices**: A simple dices game, where users can choose to keep betting or stand in any of n rounds (by default, $n = 3$), and where the winners are the users with the highest dice sum.
2. **Blackjack**: A classic Blackjack game, with a dealer and a number of players that can be specified when deploying the system.
3. **Exponential Coinflip**: A coinflip game with a twist: users can choose to double the bet indefinitely. If both users agree on it, the game will continue, otherwise, the game will end.

2 Implementation

For the implementation, we used Solidity, a language to write Smart Contracts in the Ethereum blockchain. They cover all of the usual back-end and database functionality. In the front-end, we built the website with React and used the Web3 library to interact with the contracts. The deployment and testing inside the local Ganache network was made possible by Truffle suite.

2.1 Smart contracts

Our contract structure has two layers. There is a **Main Contract**, that has the responsibility of holding the actual user funds and includes methods like `addFunds()`, `retrieveFunds()`, and, most importantly, `modifyFunds()`. The former method is used to alter the amounts that each player has and can only be called by the other type of contracts, **Game Contracts**, that include the logic of the games themselves.

Game contracts include a way for users to register to the game, vote for its start and then perform the game-specific actions. We use `GameStateChanded()` events and expose variables or use getter functions for the external users to interact with the game public information.

2.2 Tests

We also included some Ganache tests to both the Main and Game contracts. Our testing is not exhaustive as even simple games contain too many execution branches for it to be practical.

Nonetheless, writing them proved to be very useful as a previous step to the Web3 interface implementation. Tests were useful both for discovering bugs and for understanding how the contract worked in every step.

2.3 Frontend and Web3 interface

In the React frontend, we used `ContractService` classes to provide a simple asynchronous interface to the contract methods and events. To add responsiveness we send events from the blockchain every time the state of a game is changed. When a client receives an event, he then uses calls to the blockchain to obtain the necessary information to update the web. We hooked up listeners to the events generated in the Game contracts and dynamically updated the UI using React `useState()` variables.

3 Challenges

3.1 Multi-Contract communications

For our contract model of Main and Game contracts, we needed trust between them. We couldn't allow a regular user to act as a Game Contract and modify the amounts of the accounts. For this, we used a bidirectional relationship set only by the owner during contract migration. Furthermore, we used an auxiliary library, `Structs.sol`, that contains the `Payment` structure that is used when calling `modifyFunds()` in the Main Contract.

3.2 Gas costs

An important fact we learned is that there is a big difference between `pure` or `view` Solidity functions and regular ones: `pure/view` functions only *view* the contents of the Blockchain, and thus are executed only locally and don't need gas. In the other hand, methods that *modify* the state of the Blockchain (adding/retrieving funds, game actions...) need gas.

This causes a problem in our games, since, per example, in Blackjack a user will need to pay gas in every step of the game, from entering, to voting for it to start, to hitting cards to standing and then retrieving his new funds from the casino. This

hurts both the user experience and the fairness of the gambling, since computing the real-life cost of the gas accounts of multiple CHF per game and the user must accept a MetaMask transaction every time.

To atenuate this effect, the efficiency of the contracts could still be improved to save some gas costs, or the players could play with higher stakes so the gas is only a small percentage of the bet.

3.3 Receiving blockchain events

Another problem we had is with the listening of events in the frontend. Our first implementation of Blackjack used multiple types of events for each action: `UserJoined`, `CardDealt`, `UserStood`, ... This caused problems since events are sometimes sent twice (from different blocks) and they also are not necessarily be received in order or in the expected way. In the end, we replaced them by a single event that transmits the entire (public) state of the players and the phase of the game.