HorsEther: Ethereum Based Horse Race Betting System

Alex Mitchell
GSU Department of Computer Science
Blockchain & Applications
Atlanta, United States
amitchell61@student.gsu.edu

Kevin Aiken
GSU Department of Computer Science
Blockchain & Applications
Atlanta, United States
kaiken3@student.gsu.edu

Jenny Choi
GSU Department of Computer Science
Blockchain & Applications
Atlanta, United States
hchoi26@student.gsu.edu

Aarohi Savaliya
GSU Department of Computer Science
Blockchain & Applications
Atlanta, United States
asavaliya1@student.gsu.edu

Kevin Luna
GSU Department of Computer Science
Blockchain & Applications
Atlanta, United States
kluna4@student.gsu.edu

***Abstract*— In this paper, we cover HorsEther, a distributed app (DApp) for gambling on simulated horse races. There are a variety of similar DApps and platforms similar to our project covered. Additionally, we go over the design of the gambling game, the specific implementation details, and potential future improvements to the DApp. The significant challenges faced in the process of developing a DApp are also covered.**

I. Introduction

Decentralized currencies and their platforms lend themselves well to gambling-related applications. There are fewer legal restrictions on gambling with cryptocurrency, and transferring funds is simpler with cryptocurrencies. We took advantage of these aspects of the Ethereum platform to develop a game where users can bet Ethereum on a simulated horse race, and make money if they win.

II. Project Idea

This idea was selected as it is a practical application to run on the Ethereum network, but is not overly complicated, and has real-world value. Our idea provides a model that could be used for a system based on real horse races, with real Ethereum involved, instead of Ethereum from a test chain.

III. Related Work

There are various gambling games created with blockchain technology. Like our DApp, they all have the same general outline. A user bets a certain amount of currency in hopes of receiving a larger amount back.

Dicether is an Ethereum based dice game which supports four different games that users can choose from. Users can choose to play Classic Dice, Choose from 12, Flip a Coin and Keno. This game makes use of smart contracts with state channels.

Luckchain is a decentralized betting platform based on EOS. It supports betting of EOS block's hash and sporting event such as, football, baseball, soccer, boxing, and more.

There are many more betting platforms that do not make extensive use of blockchain technology but do allow users to fund their accounts with cryptocurrency. An example of this is "BetOnline", a platform that manages bets on a variety of sports. They allow users to deposit funds to their account with Ethereum, Bitcoin, Litecoin, and directly with USD through Visa and cash transfers. Our implementation differs from these in that we make transparent use of smart contracts to manage betting.

IV. Game Design

A. Overview

HorsEther has two groups of users, administrators, and bettors. A HorsEther user can view upcoming races, place bets, and view past races. Admins can create races, specifying the time the race occurs, and they can trigger race evaluation after a race occurs to randomly select a winner and then reward bettors.

Users can place bets on races that take place in the future. Each race has a time the race occurs, and after that time it becomes eligible to be evaluated, meaning the winner being randomly generated and bets being paid out. Bettors cannot bet on a race that has already passed its time. When a user creates a bet they will select an amount to bet, the race to bet on, and the horse they think will win. When they submit their bet, assuming it is valid, the Ethereum will be transferred to the game contract. After a race is scheduled to happen an administrator can run the race evaluation. Users who bet on the winning horse receive currency roughly correlated to the risk they took. This risk and reward system is similar to how real horse race betting websites operate.

B. Profit Mechanism

Assuming a user has a statistically average chance of betting on the correct horse, the game will generate income for the operator by paying out less money than is put in. For a predefined period before a horse race is set to begin, users can place their bet on a horse. Once the user places his/her bet, the amount will be transferred to the HorsEther contract. After the game generates the winning horse, the users that bet on the winning horse will be credited their reward into their account. The calculation for a bettor's reward is the value the user bet, times the number of horses, minus twenty percent. This formula reflects the odds of betting correctly, minus a twenty percent game fee. The remaining difference in the betting pool will be kept by the game.

V. Program Architecture

Our architecture consists of two primary components: Smart contracts written with Solidity, and a React.js frontend making use of web3.js through a library called Drizzle.

A. Blockchain

HorsEther uses one smart contract, HorsEther.sol. It is written in Solidity and deployed to our test Ganache with Truffle. The contract effectively acts as a back end, storing data about races and the bets placed on races, running code to create more races and bets, and running code to process the result of a race.

B. Frontend

Our frontend is a React.js web app that renders content based on routes. It interacts with our smart contract through Drizzle, which is an abstraction of web3.js. User input and interaction with our contract is handled through our frontend, as well as giving users a view of what is happening in the smart contract. Functions that change the contract or transfer funds are handled through Metamask, requiring confirmation before funds are exchanged.

VI. Smart Contract Implementation

A. HorsEther Data Structuring

All of the permanent data stored concerning races is stored on the blockchain via an array of race structs. Each race struct contains an array of horses represented by their index, a boolean flag indicating whether or not a race is paid out, the time at which a race runs, an array of bet ids, and the index of the winning horse in the horse array. The bet ids are used to lookup a corresponding bet struct through a mapping. This bet struct then contains the address of the bettor, marked payable so the bettor can be transferred funds, whether or not they have been awarded yet, the index of the horse they bet on, and the amount they bet. We used a mapping for bets because dynamic arrays of structs within another struct is not well supported.

In addition to these structs, there are some simple variables. *adminHashPass* stores the result of *keccak256(abi.encode("password"))*, which is used to authenticate. A variable to track the number of bets placed since the contract was deployed is also in place, *betsInSystem*. This is used to give unique incrementing ids to new bets so they can be looked up via the betIdToBet mapping.

Between the array of race structs, the mapping of bet structs, and a couple of variables, all the information needed to operate HorsEther is stored.

B. Creating Races

Creating a race is simple. The *createRace* function takes an array of unsigned integers representing horses numbers, the time a race occurs formatted in Unix time, the number of seconds since January 1st, 1970, and a password. The first check before any code is run is checking if the hash of the password given as a parameter matches the stored *adminPassHash*. This is important, as users should not have the ability to create races. After authenticating, the time is checked to ensure the race takes place at a later time than the time the function is called. A race that takes place in the past would not be able to accept any bets and would be a waste of Ethereum to run.

If the requirements are met, a new race struct is pushed into the races array. The array of *horseNumbers* is passed into the created struct. Whether the race is paid out is set as false. The *raceTime* parameter is passed to the new struct. An empty array of bet ids is initialized. The winning horse is set to -1, as this represents no winner. After running this function, there will now be a race to bet on.

C. Placing Bets

To create a bet the *createBet* function is called. The parameters for this function are the index of the race the bettor wants to bet on in the array of races, the index of the horse they would like to bet on in the array of horses in the race struct, and the amount they would like to bet.

There are several require statements used to make sure the bet is acceptable. There must sufficient Ethereum sent to cover the amount parameter. The race must exist. The race's time must not have passed yet. Finally, the horse must exist in the race. Once all these conditions are met, the bet will be entered.

When a bet is entered, first *betsInSystem* is incremented. Then a new bet id is created, equal to *betsInSystem*. An instance of the Bet struct is created with the address of the bettor, a value of false for if the bettor has been rewarded, the index of the horse they are betting on, and the amount they are betting. This is stored as a value in the mapping using the previously declared new bet id. Finally, the bet Id is pushed into the correct race's array of bet ids, so that it can be looked up later.

D. Evaluating races

The most complex method in our contract is the method to evaluate a race. It is called with a couple of important parameters. First, the index of the race to be evaluated, and second the password for authentication, to limit the function to being runnable by only admins.

Several guards are in place. There is a check to see if the correct password is provided as a parameter. Then there is a check to see if the race has not yet occurred. The boolean flag indicating whether a race is evaluated yet is checked. Finally, it is checked that the race exists. If all of these requirements are met, the code begins to run.

The first step in the method is to generate a random number. As generating a truly random number is an unsolved problem with Ethereum contracts, our implementation generates a pseudo-random number based on the blockhash of the previous block. It then takes the last digit of that number as a random number between 0 and 9. For simplicity, our group only operates horses of 5 races, so cascading if statements turn the random number between 0 and 9 into a random number between 0 and 4 to select the winning horse.

Once the winning horse is determined the bets are paid out. Every bet id in the races array of bet ids is iterated through, checking if they bet on the correct horse, and paying out to addresses that placed a winning bet, based on how much they put in. Potentially at this point, an issue can occur, and the evaluation will fail and be reverted if there is not enough money stored in the contract to pay out the bettor.

Once the bettors are paid out, the race will be marked paid out, and the winner will be saved into the race struct so the result can be viewed later.

VII. Frontend Implementation

A. Technology Choice

The main technology used in our frontend is a Javascript framework called React. We chose this technology to give our project more structure than simply a loose collection of Javascript and HTML. React is also very common in the workforce for frontend development, so it was a beneficial technology to learn by developing HorsEther.

The second big component of our frontend is Drizzle, a library built on top of web3.js and Redux, a state container. We chose to use this abstraction of web3 instead of web3 because it significantly boosted our productivity. After doing a short tutorial, we found we could do things we were unable to do after hours of using web3. This was due to a mix of Drizzle having better documentation and up-to-date tutorials, and being more powerful overall.

B. Program initialization

The entry point of our frontend is index.html in the public folder under the app folder in the main directory. index.js renders all of our components by editing the root div in index.html, specifically by rendering the App component in App.js in that div. In index.js we also instantiate Drizzle with our contract and a disabled fallback configuration. The disabled fallback configuration is to force Drizzle to fail if Metamask is not installed, instead of the default behavior of using a local blockchain if the Metamask connection fails.

Once App.js is loaded, Drizzle is initialized, with failure being displayed if Metamask is not found. Once Drizzle is successfully initialized, a navbar is rendered, then inside of a set of routes controlled by a switch, the rest of our app is rendered. The component being rendered is determined by the current URL. An important note is that the Drizzle instance and state are passed to the inner component so that they can make use of it to trigger contract related code.

C. Betting

Home.js consists of a link to Bet.js. Inside of that bet component, we set an initial state, and bind a function to pick up changes to the bet form inputs. We make a cached call to *getNumberOfRaces* in our contract which tells Drizzle to call this contract function again and re-render our component if there is a new block on the blockchain that interacts with our contract. Our *placeBet* function calls the *createBet* method in our contract with the specified parameters. An important part of this is converting the amount of Ethereum put into the form into wei. The *getTxStatus* reflects the *stackId* from the *cacheSend* method from Drizzle which holds the status of the *placeBet* transaction. The same method is used in a couple of other classes to monitor the transaction status of various functions.

The *getRaces* function loads an HTML table into the component state by calling races one by one based on the number of races gotten from the cached call to *getNumberOfRaces*. Races that have already taken place are ignored while iterating through races. This table of races is loaded into the state because if it changes it is automatically re-rendered, so if a user pressed "Load Races" and there has been a change the table will be updated. Finally, all of our form inputs are hooked up to items in the class state with the *handleInputChange* function. This makes sure the *placeBet* call is made with the current form inputs. The rest of the class is a large amount of JSX being returned in a render function. JSX is a syntax extension to Javascript that allows you to write easier to read code for React to dynamically render HTML with.

D. Viewing Past Races

PastRaces.js's purpose is to render the past races that are not visible in Bet.js. It uses a very similar getRaces function, but instead of ignoring the races in the past, it ignores races that are not in the past. Other than this difference, PastRaces.js is a subset of the functionality of Bets.js.

E. Profile

Profile.js makes a couple of static calls directly using web3 to get account address and balance. This page is primarily meant for future expansion, ideally for adding the past bets this user has placed.

F. Administration panel

The Admin class in Admin.js is the most complicated piece of Javascript code we developed. The same form handling behavior is implemented from Bets.js. Similar code is used to call contracts on button presses. A *getRaces* function similar to the *getRaces* function from Bet.js is used to show all races, past and present. We specifically call contract functions to create races and evaluate races in Admin.js, in a similar way to the contract calls in Bet.js.

Overall our React implementation consists of a couple of key patterns, namely binding form inputs to the state and making cached and one-time contract calls.

VIII. Challenges

We faced many challenges in developing HorsEther. About an equal number of challenges were faced with the frontend as the blockchain backend. The frontend challenges consisted of issues with Web3.js and Drizzle, in addition to the challenges in learning and developing in React.

A. Contract Challenges

We found that storing data in Solidity is best done following certain patterns that may not be obvious to someone who does not have solidity experience. This lead to us going through several iterations of our data structure before landing on our final structure. Originally we tried to use a struct of horses as an object in the struct of races, but the contract was much simpler and easier to work with when this struct was changed to simply an array of horse numbers within the races struct.

One of the big issues we found was that passing a string array as a parameter is not supported. We originally wanted to pass in horse names and allow users to bet on horses by name. There were two solutions for this: passing an array of bytes and manipulating the data, and just using horse numbers instead of horse names. We decided to use horse numbers for the sake of simplicity.

B. React Challenges

React is a challenging framework to learn. A lot of our React developing was in parallel with learning React, resulting in a lot of challenges that were resolved after learning more about how React works. Primarily these challenges involved input forms, buttons, and re-rendering on data updates.

C. Web3 Challenges

A lot of time was spent trying to get Web3 to successfully deploy a contract. However, this was tricky as many of the examples in the Web3 documentation are quite vague, and did not work as expected. Many online examples outside of the current documentation were sufficiently outdated as to be unusable. After spending a lot of time to do very little with

Web3, we looked into Drizzle, a library built on top of Web3 and Redux provided by the same team that makes Truffle. Drizzle and its provided examples dramatically increased our productivity. Drizzle lends itself better to the architecture of a React application because of the state management with Redux. After beginning to use Drizzle, we did not have any significant challenges related to Web3.

IX. Potential Improvements

There are a variety of obvious improvements for our app. The first one is improving how we evaluate races when they pass their race time. The second is changing our frontend to automatically re-render when data on the Blockchain changes. The winner selection uses a pseudo-random process that needs to be made more random to avoid users being able to win bets more often than they should. Finally, our current authentication scheme needs significant improvement or replacement.

Using an automatic system to evaluate races when they are eligible would make HorsEther more realistic and reduce the necessary involvement of an admin. One of the obvious approaches to do this is to use a cron job running on the same system that serves the React app. A sufficiently configured cron job would be able to call a contract function every minute to view races that are passed their race time, but not evaluated, and then call evaluate on these races. This is effectively the same as when an admin manually selects a race for evaluation, but automated.

Regarding potential frontend improvements, currently, a user must click “Load Races” to see races. However, in the admin panel where the “Overall number of races: “ can be seen, Drizzle is used to monitor the blockchain and automatically update when new blocks are created that interact with our contracts. Ideally, a user would never have to click a “Load Races” button, and would instead have everything blockchain related load automatically, like the “Overall number of races:” section. We implemented our frontend with the “Load Races” section due to this being easier, but user experience would be much better with automatic updating.

Finally, currently, we are using a pseudo-random system to choose winning horses. Generating secure random numbers with a blockchain is considered a difficult problem. A potential improvement could be passing in the random winner when a race is evaluated, but this could become a security risk, or seem like a lack of transparency since a bad user or administrator could select race winners to benefit themselves.

Our authentication scheme is mainly in place to demonstrate that the admin functions should use authentication. It was not designed or analyzed to be particularly secure. Our system allows an attacker to view the hash they need the inputted password to hash to so that they can access admin functions. This means an attacker could brute force the password if it is sufficiently short, like the example password that is currently set, “password”. A potential improvement would be to use a long password that does not appear in dictionaries and then hash it multiple times. Multiple hashing would cost more gas, but would also take an attacker a longer time to brute force. The current authentication scheme as it is should not be used in a real production blockchain application.