A Decentralized Sports-Betting Application on the Ethereum Blockchain.

ECO416: Fintech Final Project

Jason Yuan, Samuel Lee, Jordan Bowman-Davis, Quan (Ben) Shi

Abstract

The vast majority of sports betting is conducted through bookmakers, which act as centralized counterparties that set prices (or odds) for bets. While bookmakers are valuable in that they provide liquidity, security, and a centralized marketplace, they add transaction costs to wagers. We design a decentralized betting market that allows bettors to bet directly with each other, with the odds determined in market equilibrium as opposed to being set by a centralized bookmaker. Our betting market does away with the need for a centralized bookmaker, eliminating the transaction costs associated with the traditional betting model. Additionally, our product offers the anonymity and security benefits of the blockchain. This market is implemented as an Ethereum smart contract in which participants are automatically paired, and funds are distributed following result inputs from on-site reporters (oracles). The smart contract is deployed onto the Ropsten test network and also features a custom graphical user interface.

Economics of Betting Markets

First, it is necessary to explain the economics of traditional betting markets and the role of bookmakers. Unlike typical financial markets, in which a market maker matches buyers and sellers based on bid and ask prices, betting markets run through a single bookmaker, who set prices and act as a central counterparty. In order to earn profit, the bookmaker sets the price of bets above the fair value price. (Levitt 2004). Let's take a simple example. Assume that the bookmaker's models estimate a probability of 50% that a given event occurs: for example, Duke beats UNC in a basketball game. If the bookmaker was not concerned with making profit, it would set the odds at -100 for both outcomes. This means that a bettor would have to bet $100 to win $100. Consider the equation for implied probability:

$$Implied\ Probability = \frac{Amount\ Wagered}{Total\ Return}$$

In this case, the implied probability is 50%, as the amount wagered is equal to $100 and the total return (if the bettor is correct) is $200. Now, assume that the bookmaker seeks to profit from the bet, and thus sets the odds at -110 for both outcomes, meaning that one would have to bet $110 to win $100. In this case, the implied probability of each outcome is 55%. Notice that the probabilities of each event sum to 110%. This is referred to as the overround:

$$Overround = Implied\ Probability\ (Loss) + Implied\ Probability\ (Win)$$

From the overround, we can calculate the vigorish, or the bookmaker's long-run expected profit margin:

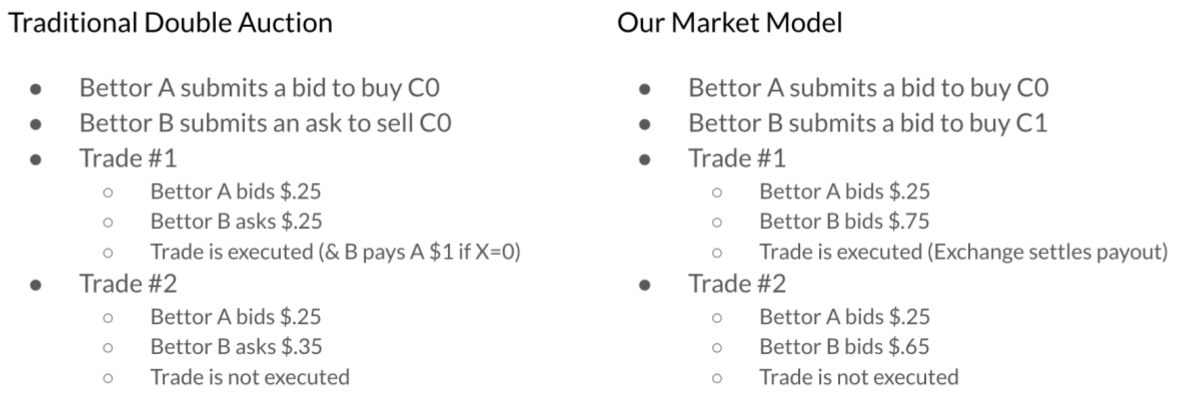
$$Vigorish = 1 - \frac{1}{Overround}$$

If we assume that the market also estimates a 50% probability of Duke winning, then there will be an equal number of people betting on each outcome. In this case, the bookmaker's profit margin will be equal to the vigorish – approximately 9%. This means that the bookmaker will pay out $100 for every $110 in accepted wagers. The goal of the bookmaker is to set the odds (price) of a wager such that there is an equal number of bettors for each outcome of the event. If the bookmaker is unable to balance the books, it places itself at risk of taking losses.
In this sense, bookmakers act as financial intermediaries for betting markets. In exchange for providing a market in which to place bets, they charge a premium by setting the price above the fair value price.

Our Market: Binary Option

We model the market as a market for Arrow-Debreu securities. To illustrate the market design, we consider a simple example. Assume that participants in the market seek to bet on a random variable, *X*, that realizes values *0* and *1* with probabilities *p* and *q*, respectively. Two types of contracts exist in this market, modeled as Arrow-Debreu securities: *C0* and *C1*. In order to place a bet that *X* will take on the value of *0*, a bettor purchases a share of *C0*. If the bettor is correct, and *X* takes on the value of *0*, the share is worth $1. If the bettor is incorrect, the share is worth $0.

We employ a modified version of a Continuous Double Auction (CDA) market mechanism in order to facilitate the exchange of bets in our market. Our model is logically equivalent to the standard mechanism employed in traditional stock exchanges. In our market, bettors submit bids for *C0* and *C1* simultaneously. The exchange mechanism considers the highest prices, *R0* and *R1*, that bidders are willing to pay for *C0* and *C1*, respectively. If the sum of these prices are greater than or equal to $1, the trade is executed and the shares are sold. This requirement is analogous to the traditional CDA requirement that the bid price match or exceed the ask price in order for a trade to be executed. In order to illustrate why this condition must be met, consider the following example. Assume that the exchange executes a trade in which *R0* = $.30 and *R1* = $.60, and *p* + *q* = 1. If *X* takes the value of 0, then the owner of *C0* will receive $1 and the owner of *C1* will receive $0, and vice versa if *X* takes the value of 1. Regardless of the outcome, the exchange has to pay out $1. However, the exchange has only collected *R0* + *R1* = $.90, and thus is required to pay out more money than it has collected. By imposing the condition that the sum of the bids is greater than or equal to 1, we ensure that the exchange remains solvent and that the total implied probability of the bets is greater than or equal to one. An illustration of our market mechanism and the analogous CDA market in various situations is shown below.



Traditional Double Auction

- Bettor A submits a bid to buy C0
- Bettor B submits an ask to sell C0
- Trade #1
  - Bettor A bids $.25
  - Bettor B asks $.25
  - Trade is executed (& B pays A $1 if X=0)
- Trade #2
  - Bettor A bids $.25
  - Bettor B asks $.35
  - Trade is not executed

Our Market Model

- Bettor A submits a bid to buy C0
- Bettor B submits a bid to buy C1
- Trade #1
  - Bettor A bids $.25
  - Bettor B bids $.75
  - Trade is executed (Exchange settles payout)
- Trade #2
  - Bettor A bids $.25
  - Bettor B bids $.65
  - Trade is not executed

One reason why we chose this modified form of the Continuous Double Auction mechanism is that it is more easily generalizable to a larger number of outcomes. It is common in sports betting to bet on events with more than 2 outcomes. For example, instead of just betting on whether Princeton will beat Yale in an upcoming game, bettors may be interested in how much Princeton beats Yale by. An example of this situation with n = 5 outcomes is shown below:

- Outcome 1: Princeton beats Yale by 10 or more points.
- Outcome 2: Princeton beats Yale by less than 10 points.
- Outcome 3: Princeton ties Yale.
- Outcome 4: Yale beats Princeton by 10 or more points.
- Outcome 5: Yale beats Princeton by less than 10 points.

As long as the outcomes are mutually exclusive and collectively exhaustive, our market mechanism is easily adaptable to such situations. If an event has n possible outcomes, and C_1... C_n are Arrow-Debreu securities corresponding to each outcome, the exchange considers highest bids $R_1 \dots R_n$ on $C_1 \dots C_n$. And if $R_1 + \dots + R_n \geq 1$, the trade is executed.

In order to match buyers and sellers of the contract, we employ a FIFO (first-in-first-out) algorithm, also known as a price-time algorithm. In a FIFO algorithm, the earliest buy order at the highest price takes priority over any later orders at the same price, which take priority over any orders at a lower price. Importantly, the FIFO algorithm does not take into account the size of each order when determining priority.

Note that any typical odds scenario offered by a bookmaker can be translated into the language of an Arrow-Debreu security. Returning to our previous example with the binary random variable *X*, we see that the fair value of the contract *C0* in this scenario can be calculated with the following formula:

$$Fair\ Value\ =\ P(X=0)\ \times \$1\ +\ P(X=1)\ \times \$0\ =\ P(X=0)\ \times \$1$$

Thus, if the probability that $X = 0$ is 75%, then the contract *C0* (with payoff \$1) will be valued at \$0.75. Compare this pricing scheme to the following odds scenario. Assume that a bookmaker offers odds of -300 that Duke beats UNC, meaning that a bettor would have to bet \$300 to win \$100. These odds imply a probability of 75% that Duke wins. In the language of an Arrow-Debreu security, *C0* would cost \$.75, meaning that one would have to bet \$.75 to win \$.25. In other words, if a bettor bought 400 shares of *C0* at \$.75, their payoff would be \$100 if $X = 0$.

Unlike traditional betting, in which a centralized bookie sets the odds and acts as a counterparty for all bets, our system has no centralized counterparty. Instead, individual bettors submit bids according to the price that they think is fair. Assuming efficient and liquid markets, the price of the binary options will eventually converge to the fair price, which reflects the probability (or the best guess of the probability using all available information) that the specified event will occur. In this sense, the market (or individual bettors) sets the odds, rather than a bookie. Additionally, since the odds converge to their fair market value, our betting market eliminates the transaction costs associated with a centralized bookmaker.

The assumption that markets are liquid is a very strong assumption. Indeed, liquidity is one of the primary downsides of our design. When a bettor goes to the market to place a bet, there is no guarantee that there will be a counterparty to accept the bet. This introduces a challenge: if there are no participants in the market, then bettors will not want to enter the market. However, the market will not become sufficiently liquid until bettors enter. We acknowledge the limitations of our design with respect to this issue, especially as it compares to the traditional bookmaker model, in which a centralized counterparty is always readily available. When designing our market, we took the problem of liquidity provision into consideration, and explored the possibility of implementing the market using an Automated Market Maker (AMM), which would address the liquidity issue. While we chose to not implement an AMM for our final design, they are worth briefly discussing.

Alternative Market Designs

The primary alternative to our order book exchange is an Automated Market Maker (AMM). AMMs are agents, typically deployed as smart contracts, that pool liquidity and provide it to traders. The fundamental difference between an AMM and a centralized order book exchange is that AMMs use deterministic market making functions, also known as liquidity functions, to set the prices for trades, whereas order book exchanges match buyers and sellers one-to-one. Rather than trading directly with each other, as they would in an order book system, buyers and sellers trade directly with the liquidity pool. This means that AMMs are always able to fulfill trades.

Two real-world applications of AMMs are worth noting: Uniswap and Augur. Uniswap, a peer-to-peer system designed for exchanging cryptocurrency tokens, is the largest decentralized exchange by daily trading volume (Bloomberg 2020). Uniswap utilizes a type of AMM known as a Constant Function Market Maker (CFMM). In this model, liquidity providers send reserves of two assets, usually two cryptocurrencies, to a central pool. The market maker determines the price of each asset relative to each other using the following formula:

$$R_1R_2 = k$$

where R_1 and R_2 are the reserves of the assets and k is a constant. For each trade, the reserves of each asset must change such that the product of the reserves remains a certain constant. This guarantees that there will always be liquidity in the pool, since the reserves of one asset cannot go to zero, as this would imply a price of infinity. To illustrate how Uniswap determines the price of assets, consider a liquidity pool with R_1 and R_2 corresponding to reserves of Bitcoin and Solana, respectively. In order to purchase b units of Bitcoin, a trader must supply Solana, s, such that

$$(R_1 - b)(R_2 + s) = k$$

$$R_2 + s = \frac{k}{R_1 - b}$$

$$s = \frac{k}{R_1 - b} - R_2$$

Thus, the price of b Bitcoins is $k / (R_1 - b) - R_2$ Solana. (see Uniswap 2021 for more details).

Augur, a decentralized platform for prediction markets, provides a recent example of an AMM applied to betting markets. Augur utilizes the LS-LMSR (Liquidity-Sensitive Logarithmic Market Scoring Rule) as its market maker. The LS-LMSR is a modification of the LMSR, which was originally developed by Robin Hanson to be utilized in prediction markets, systems where people can place bets on the outcome of political or economic events. (Othman 2015). The original goal of prediction markets was to improve forecasting by ensuring that people have a financial stake in their forecasts. The LMSR was developed as a payment scheme to compensate a forecaster who makes a sequence of probability predictions for events. To illustrate how this works in a prediction market, consider a random variable *X* that can take on one of *n* values. The market maker announces a scoring rule, which in this case is the LSMR, and posts an initial distribution, p_0, which, in most cases, is the uniform distribution. Trader *t* posts an update, p_t, to the existing distribution, p_{t-1}. When the value of *X* is revealed, trader *t* receives

$$log(p_t(X)) - log(p_{t-1}(X))$$

from the market maker, in accordance with the LMSR. Now, assume that *X* is a binary outcome variable that can take on values of 0 and 1. Assume that the previous distribution places a probability p_{t-1} (X = 1) = .6, and trader *t* posts an update to the distribution with probability p_t (X = 1) = .5 (this is analogous to betting on the outcome X = 0). Assume that *X* ends up taking on a value of 1. The compensation for trader *t* is now given by:

$$log(.5) - log(.6) < 0$$

Notice that trader *t* loses money on the trade, which makes sense given that they placed a bet on the wrong outcome. (see Hanson 2007 and Abernathy 2014 for the derivation and further explanation of the LMSR). Prices are defined

$$p_i = \frac{exp(q_i/b)}{\sum_m exp(q_m/b)}$$

where p_i is the price of the *i*'th event, q_i is the amount of money already in the system that is to be paid out if the *i*'th event occurs, *b* is a parameter set by the market maker that determines how much liquidity is in the system, and *m* is the number of possible outcomes. (see Othman and Sandholm 2010 and Abernathy 2014 for derivation).

A clear drawback of the LMSR is that it exposes its operators to potential losses. The market maker must also determine the amount of liquidity in the market before any trades are made. If the liquidity parameter is set too low, then traders may have difficulty executing trades. However, increased liquidity leads to an increase in the expected losses for the market maker, thus necessitating a trade-off between operator losses and market functioning. (Othman and Pennock 2010). The LS-LMSR algorithm improves on the LMSR in a number of ways, most notably by introducing liquidity sensitivity (so that market depth increases by market volume) and easing the bound on maximum liquidity in the market. (see Othman and Sandholm 2010 and Othman and Pennock 2010 for more details on LS-LMSR).

The obvious advantage of AMMs as opposed to central order books is that AMMs provide a guarantee of liquidity. A buyer or seller of an asset will always be able to find a counterparty, since the AMM determines the price of the asset and automatically purchases it. In the order book model, there is no guarantee that markets will be sufficiently liquid. Thus, buyers and sellers of a security may be faced with very large bid-ask spreads, or may not be able to even find a counterparty for their bet. However, AMMs have drawbacks that motivated us to utilize a central order book market design. A Constant Function Market Maker like Uniswap requires liquidity providers in order to function properly. These liquidity providers would be exposed to impermanent loss, in which the value of a token increases after it has been locked into the liquidity pool, and thus would need to be compensated with transaction fees. (Uniswap 2021). An LMSR market maker would possibly require the market maker to operate at a deficit. In order to guarantee solvency, we would need to provide funds to meet this deficit, essentially providing a subsidy for the market, or implement transaction fees to compensate the market-maker/solvency-provider for potentially operating at a loss. Since the original goal of this project is to circumvent transaction costs and achieve the most fair prices, this does not seem appealing.

Our decision to utilize an order book market was also motivated by the feasibility of implementation. The Solidity programming language does not support a floating point data-type by default. Thus, using logarithmic and exponential functions (which are necessary for the LMSR AMM) would require the implementation of Taylor series expansions and a custom floating point library, resulting in high gas fees and implementation complexity.

Smart Contract Implementation

In addition to providing liquidity and a centralized marketplace, one of the primary advantages of bookmakers is that they are trustworthy. Peer-to-peer betting, while avoiding the overhead costs of a bookmaker, introduces challenges related to compliance and trust. How can one ensure that the counterparty of their bet has enough money to pay off the bet, or that the terms of the contract are met? We address these issues by designing a smart contract to receive and distribute the funds.

Smart contracts, residing on a blockchain, autonomously enforce contracts via code. On the Ethereum network, deployed contracts are given a specific payable address containing both their functions and state (data), which are able to be called by other users with accounts on the network. We select this platform to develop our decentralized market due to its extensive resources and documentation, using the Solidity programming language within an Integrated Development Environment (IDE). The IDE allows for contract deployment onto a JavaScript virtual machine along with various testnets. This provides flexibility in terms of plentiful test ether (the currency exchanged within the market) and low transaction latency (no network congestion).

Our market is structured as two separate contracts - one containing the market mechanisms (PredictionMarket) and the other containing the result reporting functions (Oracle1). This separation is based on existing oracle architecture such as the Chainlink network, allowing for re-use of the oracle by multiple contracts with different parameters (eg. oracle quantity, result aggregation type, etc). At a high level, the PredictionMarket contract is based on the "order" structure, which contains a quantity, price, address, and outcome specified by a bidder. These orders are placed into the order book corresponding to their outcome, and are ranked by a price-time algorithm (highest price first, then break ties by earliest time). Following each bid, a matching function moves down the order books, attempting to pair bids that satisfy our model's parameters. If a pairing occurs, the orders are removed from the orderbooks. At the conclusion of the match being bet upon, reporters feed results into the Oracle1 contract, after which betters can redeem their earnings.

Computation on the Ethereum network incurs a fee known as the gas price, which is determined by the supply of the network's miners and the computing power demanded. To reduce this fee, computational efficiency is critical. The order books in the PredictionMarket contract are implemented using max heap data structures, which have a time complexity of $O(log\ n)$ for insertions, $O(log\ n)$ for deletion from the top, and $O(1)$ for queries to the top. This is more computationally efficient than maintaining a sorted array, which would need to be copied over to a new array at each insertion: $O(n)$. Several private helper functions exist to restructure the orderbook and retrieve its top values. Additionally, mappings are used between each better's address and their orders, as well as between each better's address and their position size (i.e. all their matched bids). Functioning as hash tables, mappings have $O(1)$ amortized time complexity for lookup and delete, and do not run the risk of expending excessive gas as they are not iterable.

The PredictionMarket constructor and publicly facing functions are as follows:

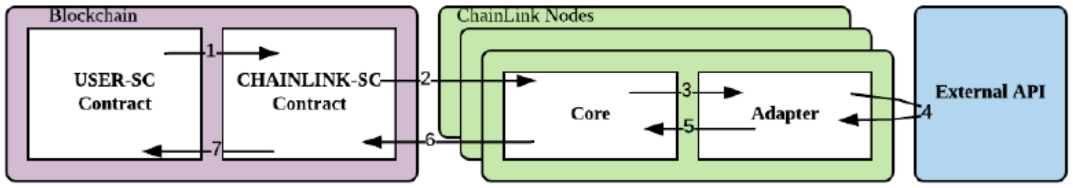
```
constructor(address oracle_addr)

function bid(uint price, uint quantity, uint outcome) public payable returns (uint)

function cancel(uint id, uint quantity) public returns (uint)

function cancelAll() public returns (uint)

function redeem() public returns (uint)
```

The constructor takes the address of the oracle contract, which is used to return the winner in the redeem() function. The bid() function takes the price, quantity, and outcome, and feeds it into the appropriate order book. It is a payable function, meaning that ether can be transferred to the contract by calling it. When a user calls the bid function, they must send an amount of ether equal to price x quantity, otherwise the transaction is rejected. After a bid is successfully executed, a Bid_ID pointing to that bid is returned to the caller. The cancel() function consumes a specific Bid_ID and quantity, checks that the caller of this function is indeed the owner of the Bid_ID, and cancels a user-specified quantity of the remaining unmatched bids of Bid_ID. The cancelAll() function cancels all unfilled orders belonging to the caller of the transaction and refunds the associated ether. Finally, the redeem() function releases the funds stored in the contract corresponding to the matched and winning bids of the caller (if the caller has zero matched contracts on the winning outcome, zero ether is redeemed). Error handling is built-in to ensure that the proper amount of ether is transferred for each bid, canceled orders are never matched, and that the Oracle1 contract returns a proper value. Source code for the smart contract can be found at the link below.

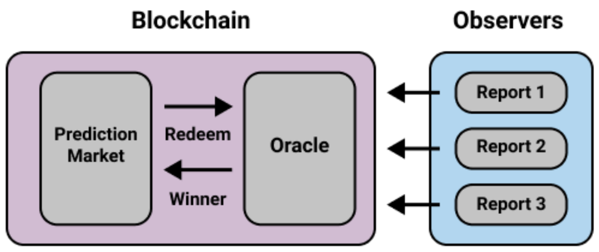
https://github.com/jasony123123/Eth-Prediction-Market/blob/master/contracts/PredictionMarket.sol

Oracle Design

The Ethereum blockchain itself cannot access information outside of network transactions, through API calls or otherwise. Doing so would break the ability to form consensus, as data pulled at one moment may be different than that pulled an instant later from other nodes. Oracles solve this by querying data from external sources and encoding it onto the blockchain. This can introduce a significant amount of unreliable data into the network if the queries are not robust. Decentralized oracle networks seek to eliminate this risk by pulling from multiple external sources for a single request and rating the quality of individual oracles against the consensus. We loosely model our Oracle1 contract on one such oracle network, ChainLink. The ChainLink network works through a ChainLink smart contract (SC), in which users can enter in the parameters relevant for their query (query data type, number of oracle responses desired, etc). ChainLink core, which is external to the blockchain, then picks up on this event through listeners, scheduling and balancing the workload across various external oracle services. Finally, adapters implement predefined or custom subtasks to pull information through HTTP requests, JSON parsing, blockchain conversions, and more. A diagram of this system is shown below:



Our Oracle1 contract simplifies this implementation by forgoing external nodes in favor of having our data input directly into the blockchain. This reduces the architectural complexity of maintaining nodes that reside on the web, but lacks the external result aggregation and robustness of queries possible through the ChainLink nodes. From a high level, the Oracle1 contract takes an integer input from three defined Ethereum addresses passed into the constructor as reporters. The inputs correspond to an outcome, with the number of possible outcomes specified upon construction. Once all inputs have been recorded, an aggregator computes the most often seen report, defaulting to report 1 if there is no consensus. A diagram of the Oracle1 contract and its interaction with the PredictionMarket contract is shown below:



The Oracle1 constructor and publicly facing functions are as follows:

```
constructor(uint outcomes, address _owner, address report_addr1, address report_addr2,

address report_addr3)

function report1(uint won1) public

function report2(uint won2) public

function report3(uint won3) public

function clear() public

function winner() public view returns(uint)
```

The constructor takes the number of outcomes of the event, which is used later in the aggregator. It also takes the addresses of three reporters, which are to be present at the event and input the result after its conclusion. In future iterations of this project, functionality could be added to customize the number of reporters and the corresponding report arrays. These addresses could then be chosen at random from a pool of available observers, similar to assignment through the ChainLink core node. Each report function takes the integer corresponding to the result, while the clear function clears all the reports. Finally, the winner function tallies up the count of each outcome and returns the outcome with the highest count. The priority ranking is report1, report2, report3 in the event of a tie, and ideally would correspond to the trustworthiness of the reporters. Error handling is built in to ensure that each reporter submits one and only one result, the results are valid, and that only the owner address can call the clear function: preventing outside actors from disrupting the result collection process.

The reporter addresses, reports, and report counts are all stored in fixed length arrays, and could be made dynamic in future iterations, at the cost of more gas fees. Arrays could also be added to permit different aggregators, and thus different end results being transferred to the calling contract. Booleans or addresses come to mind. In addition, different aggregation methods may be incorporated to include averages or weighted averages as opposed to the most often used value. It is important to keep in mind that Solidity does not fully support floating point values, so events which use floats may not be accurately reported (eg. diving or race times). One of the most critical improvements if the Oracle1 contract is to be shifted into production is making the contract and reporting addresses payable, in order to offset the gas cost associated with reports.

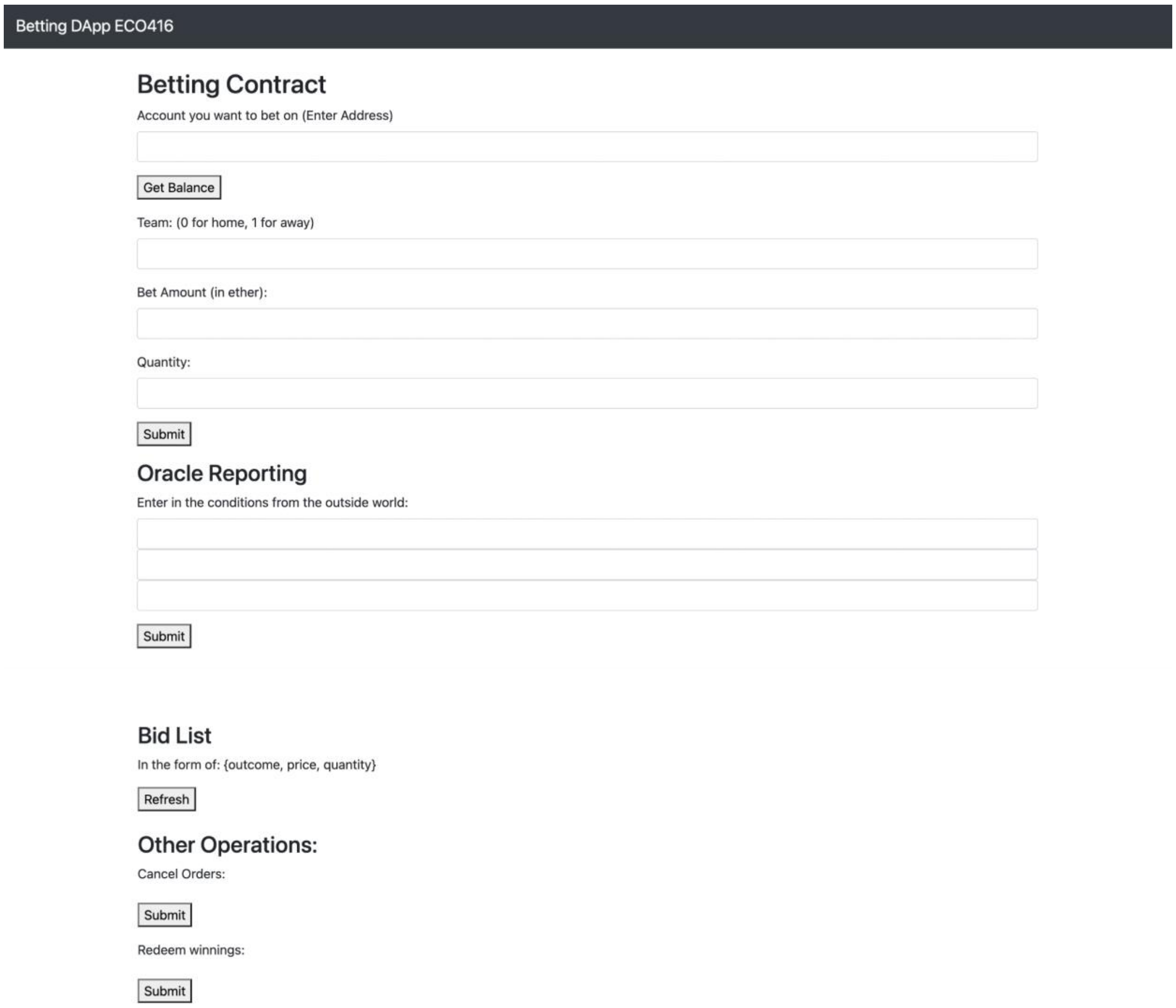
In tests on the Ropsten testnet, reports varied in cost between approximately $0.50 and $0.75 and took approximately 5-15 seconds to transact. The Oracle1 contract should ideally receive payment from the calling contract and distribute it to the reporters to incentivize their work. A penalization mechanism could also be implemented, holding payment from reporters and locking it into the contract if there is a lack of consensus. In future iterations, off-chain aggregation could be used to perform the collection of results without incurring significant gas fees by only submitting the consensus according to the method chosen. This is one of the key features of ChainLink 2.0.

Custom Front End

Although the functions contained with our smart contract are able to fulfill the functionality of the sports betting system, forcing users to interact directly with the smart contract functions to place bets would be a poor user experience. To improve on this, we integrate the smart contracts with a custom front-end to allow a better user experience. To create our front-end, we used several key pieces of software to help integrate the smart contract, the blockchain, and the browser together into one web application:

- Truffle/Ganache: Truffle is a development environment for smart contracts that integrates compilation, testing, and deployment of Smart Contracts, while Ganache allows us to host a local Ethereum Blockchain to store all the transactions used in the bidding process, additionally providing a front-end for us to see transactions present in the Blockchain. We used Truffle to compile and deploy our smart contracts for the front end, and Ganache to operate the blockchain.
- Web3.js: Web3 is a JavaScript library that allows us to communicate to the blockchain through compiled smart contract functions. We used a variety of Web3 functions to set up a coding environment where JavaScript code can directly call smart contract functions.
- MetaMask: MetaMask is a Chrome Browser extension that functions as a cryptocurrency wallet that can be used with the Ethereum blockchain.

Combining them all, we created the following front-end:



- **Account Login**: To access one's account, one needs to login through the MetaMask chrome extension system and input one's account address into the account address input space. This ensures that one user cannot access another user's account and bid on their behalf. The next steps to further improve on this aspect would be to create a user registration page so that a user can more seamlessly create an account and obtain their address to start bidding through our system.

- **Bid**: To bid, the user must input their desired bid (outcome 0 or 1), price on bid (in ether), and quantity (number of bids they would like to place). Then, upon pressing "submit," a MetaMask confirmation window pops up, prompting the user to confirm the transaction before the bid is submitted. Gas that is consumed is charged out of the bidder's account.

- **Oracle Report**: To simulate the oracle reporting, we have three input slots, each representing the reports of trusted sources from the outside world. A next step would be using an Oracle Reporting technology, such as Chainlink, to obtain information from more complex outside sources.

- **Bid List**: The Bid List displays a list of all the unfulfilled bids that a user has placed through the system.
- **Cancel Orders**: The cancel orders button operates just like the smart contract function cancelAll(). By pressing it, the unfilled bids of the user are cancelled, and the ether put in is refunded to the user.
- **Redeem Winnings**: All of the winning and matched bids that the user submitted will be redeemed in the form of ether sent to the address of the correct user.

The code used to make the front end possible can be accessed with the following Github repository: https://github.com/botake/bettingdapp

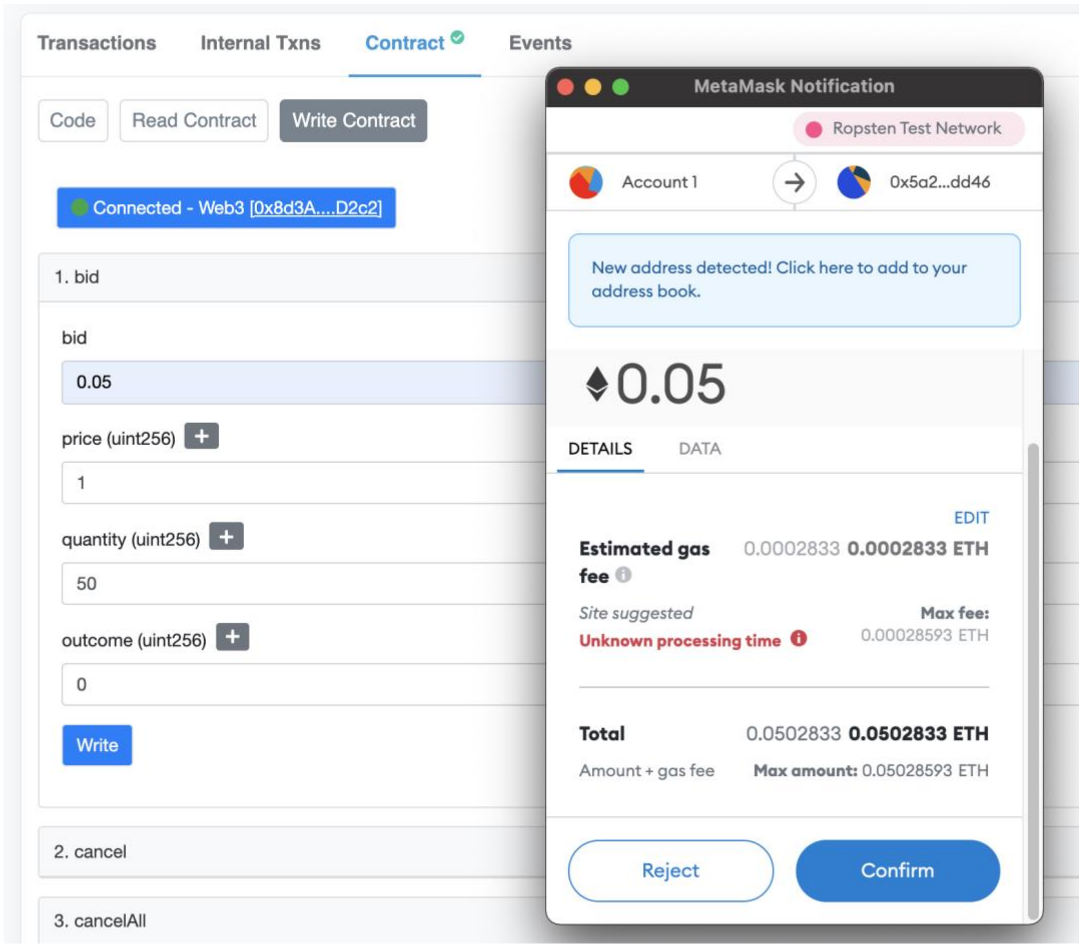
Live Deployment

The Ethereum "mainnet" is where all the transactions involving real, valuable ether are conducted. In addition to the mainnet, there are several other "testnets" that users can submit their smart contracts to and transact with them. The testnets are, from a software perspective, indistinguishable from the mainnet: there are miners mining the transactions and running the same mining and Ethereum Virtual Machine (EVM) code as the mainnet. The main distinction between mainnet and testnet is that anyone can get fake ether on the testnet via smart contracts known as faucets. Faucets are essentially addresses on the ethereum testnet that dispense a small amount of ether (typically between 0.1-5 ether) every 24 hours to anyone who requests it. After obtaining test ether from a faucet on the Ropsten network (the most active testnet), we deployed the smart contracts onto the Ropsten network. This way, anyone across the world can submit bids and interact with our smart contracts, provided they have a digital wallet like MetaMask installed and have some (test) ether in their wallet. The smart contracts can be found and interacted with at the following addresses:

- PredictionMarket, 0x5a2eD53Bf123E824b1559E514Dd7009Bf14add46 (etherscan link)
- Oracle1, 0x52dDAF1B3D014Da8b7A261Ee3091E9206b943AE0 (etherscan link)

An example of using the live contract with MetaMask is shown below. This transaction can also be further explored at this link:

https://ropsten.etherscan.io/tx/0x1f5f9dc1203f095696016417934b667bfa566eac7bb6c0f61fb9a10e3359350d



The live deployment also showcases another powerful feature of blockchain based DApps: once deployed, it is fully decentralized, scalable, anonymous, and censorship-free. Anyone can interact with the betting market, regardless of government censorship. Anyone can view the source code, and convince themselves that the market is secure and there are no hidden backdoors or incentives. And unlike current web services, our DApp is 100% available all the time since it runs on the thousands of Ethereum nodes distributed globally. This decentralization means there is virtually no chance of downtime whether that be from a malicious attack such as DDoS (Distributed Denial of Service) or an accidental server failure (which plagues even the most advanced companies, as showcased by Facebook’s DNS failure in late 2021).

Conclusion

Through modelling our market as a market for Arrow-Debreu securities, we were able to create a smart-contract, accompanied by a front-end web application, for a decentralized betting platform. As mentioned previously, the next steps to take to better our application would be to create more complex smart contract functions, build in more security measures in both the front-end implementation and the smart contract, implement a custom login interface so users wouldn’t have to go through MetaMask logins, and modify our Oracle for more seamless integration with more complex real-world data (i.e. with ChainLink), to name a few.

Sources

Abernathy, Jake. (2014). A Brief Introduction to Prediction Markets. Personal Collection of Jake Abernathy, University of Michigan, Ann Arbor MI. http://www.probabilityandfinance.com/GTP2014/Slides/Abernethy1.pdf

Aigner, Andreas A., Gurvinder Dhaliwal. (2021). UNISWAP: Impermanent Loss and Risk Profile of a Liquidity Provider. arXiv.org. https://arxiv.org/abs/2106.14404

Bloomberg. (2020, October 16). DeFi Boom Makes Uniswap Most Sought After Crypto Exchange. *Bloomberg.* https://www.bloomberg.com/news/articles/2020-10-16/defi-boom-makes-uniswap-most-sought-after-crypto-exchange

Ellis, Steve, et al. (2017). *Chainlink: A Decentralized Oracle Network (v1.0)* [White paper]. https://research.chain.link/whitepaper-v1.pdf?utm_source=chainlink&utm_medium=whiepaper-page&utm_campaign=research

Levitt, Steven D. (2004). Why are gambling markets organized so differently from financial markets? *The Economic Journal, Volume 114, Issue 495,* 223-246. https://doi.org/10.1111/j.1468-0297.2004.00207.x

Hanson, Robin. (2007). Logarithmic market scoring rules for modular combinatorial information aggregation. *The Journal of Prediction Markets, 1*, 3-15. https://doi.org/10.5750/jpm.v1i1.417

Othman, Abraham, David M. Pennock. (2010). A Practical Liqudity-Sensitive Automated Market Maker. *ACM Transactions on Economics and Computation, Volume 1, Issue 3*, 1-25. https://doi.org/10.1145/2509413.2509414

Othman, Abraham, Tuomas Sandholm. (2010). Automated Market-Making in the Large: The

Gates Hillman Prediction Market. *EC ‘10: Proceedings of the 11th ACM conference on Electronic commerce*. https://doi.org/10.1145/1807342.1807401

Othman, Abraham. (2015). Augur’s Automated Market Maker: The LS-LMSR. https://augur.mystrikingly.com/blog/augur-s-automated-market-maker-the-ls-lmsr

Sauer, Raymond D. (1998). The Economics of Wagering Markets. *Journal of Economic Literature*, *Vol. 36*, *Issue 4*, 2021-2064. https://www.jstor.org/stable/2565046

Uniswap. (2021). *Uniswap v3 Core* [White paper]. https://uniswap.org/whitepaper-v3.pdf