tokenRewards

start in epoch 6

oracle

hourOfDay in post()

settlePost: 2 day lag

processVote: 12 hour lag

post(): in hours 22 and 23 only

gameStart on initi

Test 9 is for testing the timing restrictions.

data must be submitted in GMT hour = 22

If data are submitted in hour = 22, another can be submitted in hour = 23 by any other oracle token account. Second data submission is subject to voting by oracle collective.

sequential data submissions must be sent by different token accounts

rejections:

* processVote too soon
* post wrong hour
* settle post after odds post before process
* settle after odds process, right hour, but not 2 days after odds
* settle post after settle post processed
* odds post after settle post processed
* settle post with start times in past
* post in hour 1 and hour 2,
* vote fail reject
* send again
* succeed

after init before odds

succeeds

* wd book twice
* fund book
* wd tokens

fails

* wd tokens twice in same epoch rejects
* wd shares ofter funding in same epoch fails
* bet

Test 9B

two submissions, second wins

test 9C

submission in hour 2 only, rejected

test 9D

submission only in hour 1, succeeds

test 9E

submission by same account on same prop number succeeds

test 9F

submission needs to be different if not same proposal number

adjustments:

These were done primarily for myself, but I figured others may find them helpful. so I am not following a recommended script. I generally applied a final ‘end-to-end’ approach where I executed a sequence of transactions and check final account balances that I modeled independently in a spreadsheet. While I debugged my program looking at various intermediate steps, once the contract was debugged and modeled correctly in spreadsheets, the final matching of various accounts generates a simple test that the intermediate transactions were correctly processed.

In general, the final balances checked via “assert.equal” statements are in yellow in the Excel workbook.

You can run these tests by putting the various Javascript files from smart/hardhat-testlibrary into the smart/hardhat-test folder. Then run “npx hardhat test” in the smart directory.

# Test 1: margins and odds

Test of basic bet payoffs and margin accounting. The spreadsheet shows how various bets should affect the margin before and after results are processed in the settlement. Several separate bets are made on four different matches, for a total of 12 bets. Each match includes bets for both home and away teams, and so involves netting exposure for the bookies.

The bettors should receive their initial bet, plus the payout implied by the odds on their bet. By looking at the resulting balance of the players after redeeming their winning bets we can see that they are attributed their payouts correctly. Specifically, the contract that matches the final balance numbers will have to get the following correct:

1. Odds for match
2. translating odds to payouts on favorite and underdog
3. paying the LP
4. paying the oracle
5. redeeming the correct matches to the appropriate bettors with the appropriate amounts
6. result for match
7. fee to LP
8. fee to Oracle

The amount of eth going in should be attributed to the correct parties.

Initial End

LP 3.0 2.9735

Bettor 2 1.0 0.695

Bettor 3 1.0 1.3001

Oracle 0 0.031325

Sum: 5.0 5.00

The contract rounds down, so the data do not match exactly. However, it highlights the basic methods of

# Test 2: margin and bet test with different outcomes

This is just a repeat of Test 1 with different outcomes for the same matches and odds. It highlights how one can adjust these data on the spreadsheet and then see that they are correctly processed in the contracts. In the first test, the winners for matches 0 through 3 were {underdog, underdog, favorite, tie}, while in this test is is {favorite, favorite, tie, underdog}. Note that team/player 0 is the favorite, and team 1 is the underdog.

# Test 3: Odds Updates

Here we test updating odds by looking at the ultimate bettor redemptions. Initial ‘unadjusted’ odds for the favorite were originally 1.999, which is then updated to 1.800 (input as 999 and 800).

Acct1 balance should be 1 + 0.95\*(0.999\*0.1) = 1.949

Acct2 balance should be 1 + 0.05\*(0.800\*0.1) = 1.760

Accounts 1 and 2 started at 1 Avax, and placed bets for and after redeeming their bets ended at 1.949 and 1.760 respectively, highlighting this account received the winnings accurately.

Also note that the oracle contract balance is 0.09 avax, reflecting the payment of 0.05\*(0.999+0.800).

Finally, the betting contract balance is 4.910, reflecting the payment to the oracle at settlement.

# Test 4: Redemptions and Withdrawals

This tests bettor redemptions and withdrawals. Users bet, redeem, and withdraw. Withdrawal checks to make sure gas fee accounts for the difference between the ether going out and the EOA’s change in ether.

**Fails:**

1. An attempt to redeem with no bets
2. Attempt to redeem with active bets

The biggest test here is just to make sure the units are correctly handled, so that when one withdraws 1 avax, one gets 1 avax. The contract adjusts the decimals of avax sent to the contract to conserve on memory, as using 18 decimals would quickly create problems when the user balances are uint32 (10 decimals) and the total margin amounts use uint64 (20 decimals). In the unit transformation, it is essential to verify that the adjustments for avax sent in are correctly reversed with bettors redeem and withdiraw their balances.

Thus, here I calculated not only the change in the user’s account upon withdrawal, but adjust for the gas cost. Here the gas price is set to 200 gwei.

# Test 6: data submission

Tests that data sent to the oracle are processed correctly in their sequencing requirements, and also whether the votes were majorities for passage or not. There are three token owners

Account[0]: 4.0MM tokens

Account[1]: 2.5MM tokens

Account[2]: 1.5MM tokens

Total of 8.0MM tokens

* The initial process succeeds because there were no other votes, and the initial proposer has all of his tokens count as a yes vote; the majority yes vote is unanimous.
* Attempt to send updated odds is successful, as the total vote is 5.5MM yes, 2.5MM no. Odds for match 1 change from 500 to 600
* Attempt to send updated odds fails, as the vote total is 4.0MM yes and 4.0MM no. A majority yes is needed, so odds stay at 600.

Fails

1. Try to send data to the betting contract prior to the end of the 11-hour cure period
2. Try to send odds to the oracle while the odds currently being voted upon have not been processed.
3. Try to process a settlement though the data under review are an odds update.

# Test 7: LP payout

This tests that LPs are allocated eth correctly given their eth investments and LP revenue. It also tests whether the LPs are allocated their appropriate amount of token rewards. LPs are credited with shares based on the current share price, which is the ratio of LP eth to LP shares. A withdrawal involves redeeming those shares, which then sends the requisite amount of eth to the LP.

From the start of the first contest to settlement, no investment or withdrawal is allowed by the bookies. Outside of this time window the ratio of eth allocated to the bookies (margin[0]) and the shares owned by the bookies, is the share price. Investors are given shares at this price, and shares are redeemed at this price.

Initially, we have two bookies who invest a total of 100,000 units. Their initial share allocation is also 100000 units, as the ratio of

Initial LP amounts

shares eth

account 0 60k 6 eth

account 1 40k 4 eth

A bet is made for 0.5 ETH each period, which loses. This is sent to the bookie margin at settlement, adding to the LP’s eth. There are no winners, so nothing leaks to the oracle.

In the first two epochs, the LPs split the 0.5 ETH 60:40 based on their relative shares.

Initial shares and eth invested are consistent.

In period 3 account[0] withdraws 10k shares, and account[1] withdraws 5k shares, the contract now has 85k shares outstanding. The value of LP shares is the same, as ETH was distributed consistent with the value of shares redeemed.

After another settlement and 0.5 ETH in LP revenue, a new LP adds 2 ETH, and receives 17258 shares, based on the ETH/share value of 1.16e-4.

After two more settlements, the final LPs all withdraw all their shares. The final amounts received must correctly capture the fact that the relative ownership for accounts 0 and 1 changed both when they withdrew shares, and from the deposit of account 2.

Also checked are the token rewards. Each period LPs can receive a portion of the 2e7 tokens available each week for redemption based on their pro-rata ownership of the LP pool (ie, their relative share amount).

I excluded a tokenReward function for account 0 after the fourth settlement. The final token amounts

# Test 8: Oracle token holder payout

case A: basic

case B: withdraw

case c: deposit

case D: account2 votes 1/3 the time

money not sent to other accounts

case E: account1 votes 2/3 of the time

money sent upon withdraw

case F: account 4 claims token rewards

dilution

This tests the oracle’s ability to pay out its revenue correctly. Oracle token holders can withdraw their share of the oracle’s at any time subject to two conditions

there is no vote taking place. If there is a vote, token holders cannot withdraw to prevent double voting. As a vote can only happen once per day, and voting periods last 12 hours, after which anyone can process the vote, this leaves many hours where a token holder can withdraw.

Token holder has not claimed AVAX that epoch. This prevents token holders from withdrawing and depositing within the same epoch. This makes it easier to allocate avax to token holders, however, it is not necessary. Allowing token holders to withdraw and deposit multiple times in a short period, however, creates complicated and potentially dangerous attack surface. As we need to monitor the token holder’s last withdrawal epoch for accounting purposes, the data needed for this already exists, so it was a simple restriction to add.

Case A: In the base case, Test8a, three accounts deposit tokens in the oracle contract at various times. In each epoch the oracle receives 0.1 AVAX in revenue that needs to be allocated to the oracle owners, which is represented as 100. Oracle revenue is only claimed by having tokens deposited in the oracle contract.The spreadsheet calculates this two ways, one by summing the payments (column AA), the other by using the liqPool number as in the contract (columns AB). The latter matches more precisely due to the rounding

Rounding always reduces the payments to withdrawers, so rounding errors will not cause an insolvence. They do make it more difficult to test the contract, however, as one cannot be certain a discrepancy is due to an innocuous rounding error or something more dangeruos.

case b: Account 0 withdraws part of his tokens early on.

This is case a, but here we have a withdrawal after the third epoch, as well as a final withdrawal for account 0 at the end. We are verifying the contract correctly adjusts the token payouts for this complication.

Case C: account 0 adds more tokens midway.

Case D: Account 2 votes 3 times over 3 epochs. As there were 6 votes over this period, he voted one-third of the time. Thus, his payment is one-third that in Case C, which is otherwise identical. The AVAX surrendered by Account 2 is re-added to the oracle pool. As Account 2 withdrew at the end, this extra avax is not distributed to the other oracle token holders in this case.

Case E: Account 1 votes 4 times over 3 epochs. Here the token holder gets two-thrirds of his AVAX, as he voted onely that portion of times. As he withdraws second, this affect the third (by account0) and fourth (by account2) withdrawals, so that all 0.5 avax are distributed.

Case F: Acct4 claims token rewards as an LP. This makes sure the token rewards are handled correctly. When token rewards are claimed users reset their token accounts, which sends them their earned AVAX to their EOA, and puts their token epochBase at the current epoch.

There are 5 payments generated, and the oracles enter and exit and different times. In epoch 3 account 0 increases her token investment. The spreadsheet shows how the payments are calculated in the orange cells, and this is equivalent to the amounts calculated in the green cells. The contract uses the algorithm in the green cells. The difference is due to rounding in the contract. I used bet amounts of ~20 finneys, because this was convenient for test nets where I had limited amounts of ether, so the rounding truncation is economically inconsequential, and does not create an insolvency risk because it shortchanges withdrawers, not the contract.

# Test 9: Oracle token holder payout

This is a rerun of test 8, only account 0 adds to her token deposit as opposed to withdrawing. I thought this would be silly but it identified a bug, so I’m very glad I did this test. Using the same logic as in test 8, it highlights how oracle revenue is allocated to depositors as a function of their proportional deposited amount each week.

# Test 10: input check

Betting odds and start times are packed into a single uint256, along with other data, for each epoch’s match. To demonstrate this method is correct, we input the data, make bets that adjust these other data, and then pull and decode the information from this uint256.

The tests show that the odds input can be successful decoded, and that the update in the odds was accurately recorded and did not affect the other variables.