Event-Based Prediction Market

Building a prediction market that uses the UMA Optimistic Oracle for settlement and event identification In this section, we'll talk about theEvent Based Prediction market contract , which you can find in thedeveloper's quick-start repo . This tutorial will show how event-based OO data requests can be used in a binary prediction market.

You will find out how this smart contract works and how to test it and deploy it. Refer to theOptimistic Oracle V2 contract for additional information on the event-based price requests.

The Event-Based Prediction Market

This smart contract lets you set up prediction markets that are based on an event-based price request.

For example, you could ask a question like, "Will BTC be over 1M when the first Moon hotel opens?"

(Note: Although these are called "price" requests in the code, you can request any kind of data from the optimistic oracle. If it's helpful, you can mentally replace "price" with "data" when you see it in the code and documentation.)

In addition, an event-based price request cannot expire early and automatically returns the requester's reward in the event of a dispute.

By participating in the contract, users are issued long and short position tokens, which, when held, provide exposure to the prediction market.

This contract's logic is a streamlined version of theLongShortPair with an event-based pricing request.

Development environment and tests

Clone the dev-quickstart repo

```
Copy git clone git@github.com:UMAprotocol/dev-quickstart.git cd dev-quickstart
```

To install dependencies, you will need to install the long-term support version of nodejs, currently nodejs v16, and yarn. You can then install dependencies by running yarn with no arguments:

```
Copy yarn
```

Compiling your contracts

We will need to run the following command to compile the contracts and make the typescript interfaces so that they are easy to work with:

```
Copy yarn hardhat compile
```

Contract design

Contract creation and initialization

The contract is created by setting up the prediction market with a series of parameters. With the parameters, you can choose what kind of collateral you want to use with_collateralToken . With_pairName , we can choose a name for the pair of long and short tokens._customAncillaryData is where we define the price request question. The addresses of the_finder and_timerAddress set the rest of the contracts addresses we interact with. For reference,here is the full list of UMA contract deployments.

```
Copy constructor( stringmemory_pairName, ExpandedERC20_collateralToken, bytesmemory_customAncillaryData, FinderInterface_finder, address_timerAddress ) ...
```

```

Once the contract has been deployed, the owner can callinitializeMarket() after approving theproposerReward amount to be paid to the wallet that resolves the price request. To keep things simple,proposerReward is set to10e18.

Also observe howpriceIdentifier is set to"YES_OR_NO_QUERY" .

This function sets up the prediction market by getting the proposer reward and calling_requestOraclePrice . This last function starts the price request inOptimistic Oracle V2 and sets up a number of options that are explained below.

```

Copy function_requestOraclePrice()internal{ OptimisticOracleV2Interface optimisticOracle=getOptimisticOracle();

collateralToken.safeApprove(address(optimisticOracle),proposerReward);

optimisticOracle.requestPrice(priceIdentifier, requestTimestamp, customAncillaryData, collateralToken, proposerReward);

// Set the Optimistic oracle liveness for the price request. optimisticOracle.setCustomLiveness(priceIdentifier, requestTimestamp, customAncillaryData, optimisticOracleLivenessTime);

// Set the Optimistic oracle proposer bond for the price request. optimisticOracle.setBond(priceIdentifier,requestTimestamp,customAncillaryData,optimisticOracleProposerBond);

// Make the request an event-based request. optimisticOracle.setEventBased(priceIdentifier,requestTimestamp,customAncillaryData);

// Enable the priceDisputed and priceSettled callback optimisticOracle.setCallbacks(priceIdentifier,requestTimestamp,customAncillaryData,false,true,true);

priceRequested=true; }

```

_requestOraclePrice is in charge of initializing the price request in theOptimistic Oracle V2 by performing the following actions:

1. Create the price request with the above-mentioned parameters.
2. Define the custom liveness period that a proposed oracle response must sit through before being accepted as truth.
3. Specify the bond that the proposer and disputer must post in order to resolve a request.
4. Set the type of request to Event Based.
5. Turn on thepriceSettled
6. andpriceDisputed
7. callbacks so that our contract can use OO callbacks to respond to these kinds of events
8. 

Long and Short tokens creation and redemption

Any user can now call thecreate function with thetokensToCreate parameter to mint the same number of short and long tokens. Having both tokens in the same proportion means being in a neutral position, as is the case when callingcreate .

Holding only long tokens (by transferring short tokens to another wallet or adding liquidity to an AMM pair), gives exposure to the long position and vice versa.

```

Copy functioncreate(uint256tokensToCreate)publicrequestInitialized{ collateralToken.safeTransferFrom(msg.sender,address(this),tokensToCreate);

require(longToken.mint(msg.sender,tokensToCreate)); require(shortToken.mint(msg.sender,tokensToCreate));

emitTokensCreated(msg.sender,tokensToCreate,tokensToCreate); }

```

At any time a token holder with both tokens in the same proportion can exchange them for collateral withredeem.

```

Copy functionredeem(uint256tokensToRedeem)public{ require(longToken.burnFrom(msg.sender,tokensToRedeem)); require(shortToken.burnFrom(msg.sender,tokensToRedeem));

collateralToken.safeTransfer(msg.sender,tokensToRedeem);

emitTokensRedeemed(msg.sender,tokensToRedeem,tokensToRedeem); }

```

Any long-short token holder can settle tokens for collateral withsettle if the oracle has processed the price request.

The returned collateral amount is a function oflongTokensToRedeem ,shortTokensToRedeem , andsettlementPrice .

```

Copy functionsettle(uint256longTokensToRedeem,uint256shortTokensToRedeem) public returns(uint256collateralReturned) { require(receivedSettlementPrice,"price not yet resolved");

require(longToken.burnFrom(msg.sender,longTokensToRedeem));
require(shortToken.burnFrom(msg.sender,shortTokensToRedeem));

// settlementPrice is a number between 0 and 1e18. 0 means all collateral goes to short tokens and 1e18 means // all collateral goes to the long token. Total collateral returned is the sum of payouts. uint256longCollateralRedeemed=(longTokensToRedeem*settlementPrice)/(1e18); uint256shortCollateralRedeemed=(shortTokensToRedeem*(1e18-settlementPrice))/(1e18);

collateralReturned=longCollateralRedeemed+shortCollateralRedeemed;
collateralToken.safeTransfer(msg.sender,collateralReturned);

emitPositionSettled(msg.sender,collateralReturned,longTokensToRedeem,shortTokensToRedeem); }

```

Price request lifecycle callbacks: priceSettled and priceDisputed

When the price request we set up above is settled in theOptimistic Oracle V2 , thepriceSettled function of this contract is invoked.

This function calculates and stores settlementPrice as0 ,0.5 , or1 . This number is used in thesettle function to calculate the collateral to pay in exchange for long tokens and short tokens.

```

Copy functionpriceSettled(bytes32identifier, uint256timestamp, bytesmemoryancillaryData, int256price)external{ OptimisticOracleV2Interface optimisticOracle=getOptimisticOracle(); require(msg.sender==address(optimisticOracle),"not authorized");

require(identifier==priceIdentifier,"same identifier");
require(keccak256(ancillaryData)==keccak256(customAncillaryData),"same ancillary data");

// We only want to process the price if it is for the current price request. if(timestamp!=requestTimestamp)return;

// Calculate the value of settlementPrice using either 0, 0.5e18, or 1e18 as the expiryPrice. if(price>=1e18) { settlementPrice=1e18; }elseif(price==5e17) { settlementPrice=5e17; }else{ settlementPrice=0; }

receivedSettlementPrice=true; }

```

In the same way, this contract'spriceDisputed function is called when a price request is disputed. This function re-starts the same price request with the bond amount that was given back to the requester, which in our case is theEventBasedPredictionMarket .

```

Copy functionpriceDisputed(bytes32identifier, uint256timestamp, bytesmemoryancillaryData, uint256refund)external{ OptimisticOracleV2Interface optimisticOracle=getOptimisticOracle(); require(msg.sender==address(optimisticOracle),"not authorized");

requestTimestamp=getCurrentTime(); require(timestamp<=requestTimestamp,"different timestamps");
require(identifier==priceIdentifier,"same identifier");
require(keccak256(ancillaryData)==keccak256(customAncillaryData),"same ancillary data");
require(refund==proposerReward,"same proposerReward amount");

_requestOraclePrice(); }

\`\`\`

Tests

To execute the EventBasedPredictionMarket tests, run:

```
Copy yarn test test/EventBasedPredictionMarket/*
```

Deployment

Before deploying the contract check/edit the default arguments defined in the deployment script .

To deployEventBasedPredictionMarket in Görli network, run:

```
Copy NODE_URL_5=YOUR_GOERLI_NODEMNEMONIC=YOUR_MNEMONICyarnhardhatdeploy--networkgoerli--tagsEventBasedPredictionMarket
```

Optionally, you can verify the deployed contract on Etherscan:

```
Copy ETHERSCAN_API_KEY=YOUR_API_KEYyarnhardhatetherscan-verify--networkgoerli--licenseAGPL-3.0--force-license--solc-input
```

Previous Optimistic Arbitrator Next Voting with a 2-Key Contract Last updated1 month ago On this page *The Event-Based Prediction Market * Development environment and tests * Contract design

Was this helpful? Edit on GitHub