Optimistic Arbitrator

Wrapping Optimistic Oracle functionality into individual function calls

The Optimistic Arbitrator

The Optimistic Arbitrator (OA) is a contract built on top of theOptimistic Oracle V2 (OO) in order to abstract some mechanics and make them easier to use and implement with an "assert and ratify" pattern, as opposed to the Optimistic Oracle's typical "request and propose" pattern.

It should be noted that the uses permitted by the OA were always permitted by the OO; the OA merely formalizes these specific modes of use in a more developer-friendly implementation for this particular purpose. This pattern can also be used as inspiration in your projects when you request and propose (or request, propose and dispute) requests with the OO in one function call.

In short, the OA allows a user to make an assertion, i.e., propose an answer to a question directly. They can optionally ratify the assertion by escalating the question immediately toUMA's Data Verification Mechanism (DVM) where all UMA token holders vote on it.

An assertion consists of using therequestPrice andproposePrice methods of the OptimisticOracle to ask and answer a question simultaneously. It has the advantage, compared to a traditionalpriceRequest , of receiving a response immediately and incurring no final cost if the assertion is truthful and undisputed.

To ratify an assertion, an initial assertion must be made and disputed, such that the question is escalated to theDVM , where UMA token holders vote on it. Ratifications can be useful when the user prefers to have their assertion voted on, at a higher cost, rather than optimistically accepted.

The following table shows how the Optimistic Arbitrator differs from the Optimistic Oracle and why using the Optimistic Arbitrator could be interesting:

Optimistic Oracle (OO) Optimistic Arbitrator (OA) In the basic scenario, the user will first pose a question (priceRequest), wait for someone to propose (proposePrice) an answer and potentially dispute (disputePrice) a wrong answer. The user makes an assertion by submitting a question (priceRequest) to the Optimistic Oracle and simultaneously proposing an answer (proposePrice). Optionally, they can ratify the assertion and present it to the DVM for a vote (disputePrice). The requester, proposer and disputer are generally different people. The requester and and proposer are always the same person. The disputer may be another person or the same one. The user waits for someone to respond to their question. The user does not wait, answering their own question immediately. The user pays only the reward for the question, which is optional and of their choosing (although, if it is 0, proposers would not be interested in answering). The user does not have to pay a reward since they are the proposer, but they must lock the bond amount to propose, which they will receive back if the proposal is not disputed (or determined to be correct after a dispute). In the simplest scenario, in which a user merely requests a price (i.e., asks a question), proposals and disputes are made by other users, so the requester is not charged or required to pay any fees beyond the reward. However if the reward is 0, most likely, they will never receive an answer to their question. When making only an assertion, the user must lock a bond at least equal to thefinalFee ; this bond is returned if the assertion was correct (undisputed). Therefore, the final cost of asserting truthfully is 0.

When asserting and ratifying, the function has a fixed final cost for the user equal to thefinalFee of thecurrency used (~1,500 USD). This is the cost of taking a dispute to theDVM for vote resolution. Disputes are resolved in theDVM where they are voted on by UMA token holders. Disputes are resolved in theDVM where they are voted on by UMA token holders.

Development environment and tests

Clone repository and Install dependencies

Clone theUMA dev-quickstart repository and install the dependencies. 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 gitclonegit@github.com:UMAprotocol/dev-quickstart.git cddev-quickstart 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 yarnhardhatcompile

```

Contract Implementation

The contract discussed in this tutorial can be found atdev-quickstart/contracts/OptimisticArbitrator.sol (here) within the repo.

Contract creation and initialisation

The constructor of the Optimistic Arbitrator contract takes two parameters:

_finderAddress the Finder contract stores the addresses of all of the other relevant UMA contracts, which vary from chain to chain.

_currency the collateral token used to pay fees.

```

Copy constructor(address_finderAddress,address_currency) { finder=FinderInterface(_finderAddress);
currency=IERC20(_currency);
oo=OptimisticOracleV2Interface(finder.getImplementationAddress(OracleInterfaces.OptimisticOracleV2)); }

```

As part of initialization, theoo variable is set to the address of theOptimisticOracleV2 implementation as discovered through thegetImplementationAddress method in theFinder contract.

Making an assertion

The following function allows a user to make an assertion. Behind the scenes it uses anOO.priceRequest with aYES_OR_NO_QUERY price identifier . This imposes a set of rules on how the assertion is formulated and formatted, as specified inUMIP-107 .

ThemakeAssertion function takes the following arguments:

- timestamp
- timestamp used to identify the assertion, usually the current timestamp.
- ancillaryData
- the byte-converted question for which we want to assert (e.g.'q: Was the price of BTC above 18000 USD for the duration of October 10 2022 UTC time considering top 5 volume weighted markets'
-)
- assertedValue
- our response to the question. (e.g.1e18
- for yes0
- for no)
- bond
- the bond in thecurrency
- defined in the constructor that we want to require on top of thefinalFee
- (~1,500 USD worth of the currency). ThefinalFee
- can be viewed as the minimum bond required by the system, and the bond is any additional amount we require on top of that.
- liveness
- time period during which the proposal can be disputed
-

Note: ThemakeAssertion function requires the caller to approve the OA to spend thebond+finalFee amount of thecurrency .

```

Copy functionmakeAssertion( uint256timestamp, bytesmemoryancillaryData, int256assertedValue, uint256bond, uint64liveness )public{ _pullAndApprove(bond+_getStore().computeFinalFee(address(currency)).rawValue);
_makeAssertion(timestamp,ancillaryData,assertedValue,bond,liveness); }

```

Ratifying an assertion

Once an assertion has been made, it can be ratified. This function will, behind the scenes, dispute an existing assertion in the OA, i.e., an OO dispute will be made against the OO proposal made inmakeAssertion . The method accepts only the two arguments that we utilized inmakeAssertion :

- timestamp
- timestamp used to identify the assertion, usually the current timestamp.
- ancillaryData
- the byte-converted question which we want to assert.
-

Note: TheratifyAssertion function requires the caller to again approve the OA to spend thebond+finalFee amount of thecurrency .

```

Copy functionratifyAssertion(uint256timestamp,bytesmemoryancillaryData)public{
OptimisticOracleV2Interface.Requestmemoryrequest=oo.getRequest( address(this), priceIdentifier, timestamp, ancillaryData );
_pullAndApprove(request.requestSettings.bond+_getStore().computeFinalFee(address(currency)).rawValue);
oo.disputePriceFor(msg.sender,address(this),priceIdentifier,timestamp,ancillaryData); }

```

Ratifying and asserting

This function allows the user to both assert and ratify, in a single transaction, hence simplifying the arguments. This is the function to utilize if we wish to initiate ratification immediately. This kind of pattern might be useful if you want to bypass the OO and directly use the UMA DVM when asking/asserting questions. We've seen this as a useful integration path in a number of projects that maintain their own internal escalation games within their contracts and just want to use the UMA DVM for arbitration.

assertAndRatify takes three arguments:

- timestamp
- timestamp used to identify the assertion, usually the current timestamp.
- ancillaryData
- the byte-converted question for which we want to assert (e.g.'q: "Was the price of BTC above 18000 USD for the duration of October 10 2022 UTC time considering top 5 volume weighted markets"'
-)
- assertedValue
- our response to the question. (e.g.1e18
- for yes0
- for no)
-

Note 1: Here, we do not set abond on top of thefinalFee because it is set to zero by default due to the fact that theproposer anddisputer are the same wallet. Adding a bond would increase the final cost of ratification for users.

Note 2: TheassertAndRatify function requires the caller to approve the OA to spend the2*bond+finalFee amount of thecurrency . In this function we are acting as the Optimistic Oracle requestor, proposer and disputer all in one transaction. Proposing and disputing a price both require the posting of the final fee and so to do these actions in one go we are required to pull 2x the final fee as payment.

```

Copy functionassertAndRatify( uint256timestamp, bytesmemoryancillaryData, int256assertedValue )public{
_pullAndApprove(2*_getStore().computeFinalFee(address(currency)).rawValue);
_makeAssertion(timestamp,ancillaryData,assertedValue,0,0);
oo.disputePriceFor(msg.sender,address(this),priceIdentifier,timestamp,ancillaryData); }

```

Tests and deployment

All the unit tests covering the functionality described above are available[here](#) . To execute all of them, run:

```

Copy yarntesttest/OptimisticArbitrator/*

```

Before deploying the contract check the comments on available environment variables in[the deployment script](#) .

In the case of the Görli testnet, the defaults would use the Finder instance that references the[Mock Oracle](#) implementation for resolving DVM requests. This exposes apushPrice method to be used for simulating a resolved answer in case of disputed proposals. Also, the default Görli deployment would use the already whitelistedTestnetERC20 currency that can be minted by anyone using itsallocateTo method.

To deploy the Optimistic Arbitrator contract on Görli, run:

```

Copy NODE_URL_5=YOUR_GOERLI_NODEMNEMONIC=YOUR_MNEMONICyarnhardhatdeploy--networkgoerli--
tagsOptimisticArbitrator

```

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

```

Interacting with deployed contract

The following section provide instructions on how to interact with the deployed contract from the Hardhat console, though one can also use it for guidance for interacting through another interface (e.g. Remix or Etherscan).

Start Hardhat console with:

```

Copy NODE_URL_5=YOUR_GOERLI_NODE MNEMONIC=YOUR_MNEMONIC yarn hardhat console --network goerli

```

Initial setup

You will need to get the libraries and connect to the necessary contracts that we are going to use. In this tutorial we are using theTestnetERC20 token as thecurrency as described in the deployment script.

```

Copy const{getAbi,getAddress}=require("@uma/contracts-node"); const{ethers}=require("hardhat"); constsigner=(awaitethers.getSigners())[0];

constoptimisticArbitratorDeployment=awaitdeployments.get("OptimisticArbitrator"); constoptimisticArbitrator=newethers.Contract( optimisticArbitratorDeployment.address, optimisticArbitratorDeployment.abi, ethers.provider ); constfinder=newhre.ethers.Contract( "0xDC6b80D38004F495861E081e249213836a2F3217",// Finder address used in the deployment getAbi("Finder"), ethers.provider ); conststore=newhre.ethers.Contract( awaitfinder.getImplementationAddress(ethers.utils.formatBytes32String("Store")), getAbi("Store"), ethers.provider ); constcurrency=newhre.ethers.Contract( awaitoptimisticArbitrator.currency(), getAbi("TestnetERC20"), ethers.provider ); constmockOracle=newhre.ethers.Contract( awaitfinder.getImplementationAddress(ethers.utils.formatBytes32String("Oracle")), getAbi("MockOracleAncillary"), ethers.provider );

```

Make an assertion

First we need to mint thebond + finalFee amount and approve the Optimistic Arbitrator to pull the tokens. This amount ofcurrency tokens will be used to propose the assertion to the Optimistic Oracle behind the scenes:

```

Copy constbond=ethers.utils.parseUnits("500",18); constfinalFee=awaitstore.computeFinalFee(currency.address); consttotalAmount=bond.add(finalFee.rawValue);

await(awaitcurrency.connect(signer).allocateTo(signer.address,totalAmount)).wait(); await(awaitcurrency.connect(signer).approve(optimisticArbitrator.address,totalAmount)).wait();

```

Then we can proceed to make the assertion in the Optimistic Arbitrator:

```

Copy constYES_ANSWER=ethers.utils.parseUnits("1",18);

letrequestTimestamp=await(awaitethers.provider.getBlock("latest")).timestamp;

constancillaryData=ethers.utils.toUtf8Bytes( q: "Was the price of BTC was above 18000 USD for the duration of October 10 2022 UTC time considering top 5 volume weighted markets" );

await( awaitoptimisticArbitrator.connect(signer).makeAssertion( requestTimestamp, ancillaryData, YES_ANSWER,// We are asserting affirmatively the ancilliaryData yes or no question bond, 60// 1 minute for test purposes ) ).wait();

```

Then, wait one minute for the proposal's liveness time to expire. Then, we may settle the assertion to receive back thebond + finalFee as we haven't been disputed. We also can verify the assertion's result that we proposed,YES_ANSWER , meaning that

the assertion has been accepted.

```

Copy await(awaitoptimisticArbitrator.connect(signer).settleAndGetResult(requestTimestamp,ancillaryData)).wait();
constresult=awaitoptimisticArbitrator.connect(signer).getResult(requestTimestamp,ancillaryData);// YES_ANSWER
console.log("The assertion has been accepted :",result.eq(YES_ANSWER));

```

Make assertion and ratify

To ratify an assertion and escalate its resolution to the DVM, we will utilize theassertAndRatify method, which combines assertion and ratification into a single transaction.

The Optimistic Arbitrator must be approved once more, but this time the fee is double thefinalFee , as both proposing and disputing in the Optimistic Oracle require placing a bond.

At the end of the procedure, we will receive half of the bond, or1 x finalFee . The remainder (1 x finalFee) will be consumed; asserting and ratifying will cost the caller1 x finalFee :

```

Copy consttotalAmountAssertAndRatify=ethers.BigNumber.from(2).mul(bond.add(finalFee.rawValue));
requestTimestamp=await(awaitethers.provider.getBlock("latest")).timestamp;

await(awaitcurrency.connect(signer).allocateTo(signer.address,totalAmount)).wait();
await(awaitcurrency.connect(signer).approve(optimisticArbitrator.address,totalAmount)).wait();

```

We can then proceed to assert and verify:

```

Copy
letreceipt=await(awaitoptimisticArbitrator.connect(signer).assertAndRatify(requestTimestamp,ancillaryData,YES_ANSWER)).wait();

```

As we are in a test environment, we can simulate the vote that will occur after we assert and ratify by pushing a price to themockOracle :

```

Copy constdisputedPriceRequest=(
awaitmockOracle.queryFilter(mockOracle.filters.PriceRequestAdded(),receipt.blockNumber,receipt.blockNumber) )[0];

awaitmockOracle.connect(signer).pushPrice( disputedPriceRequest.args.identifier, disputedPriceRequest.args.time,
disputedPriceRequest.args.ancillaryData, YES_ANSWER// The original assertion is accepted );

```

At this point, we can callsettleAndGetResult to settle the accepted price request and then confirm that the result validates the assertion:

```

Copy await(awaitoptimisticArbitrator.connect(signer).settleAndGetResult(requestTimestamp,ancillaryData)).wait();
constresultRatify=awaitoptimisticArbitrator.connect(signer).getResult(requestTimestamp,ancillaryData);// YES_ANSWER
console.log("Assert and ratify result is a yes answer :",resultRatify.eq(YES_ANSWER));

```

Previous Insurance Claim Arbitration Next Event-Based Prediction Market Last updated1 month ago On this page *The Optimistic Arbitrator * Development environment and tests * Contract Implementation * Tests and deployment * Interacting with deployed contract

Was this helpful? Edit on GitHub