Internal Optimistic Oracle

Using the Optimistic Oracle to verify arbitrary data type results proposed through an external contract

The Internal Optimistic Oracle

The Internal Optimistic Oracle (IOO) implements the Optimistic Oracle's (OO) internal escalation game logic for price requests and price proposals within another contract. The disputes are escalated through theOptimistic Oracle V2 toUMA's Data Verification Mechanism .

The IOO contract is meant to be utilized as a type of OO that permits customized escalation game logic and custom data structures. This pattern provides the most gas efficient route to building an OO integration by removing the need to perform cross-contract calls in all cases except for when a dispute is raised via the internal escalation game.

The IOO is intended to be the simplest implementation possible, allowing it to serve as a starting point for any project that can benefit from these functionalities.

The following table shows how the Internal Optimistic Oracle differs from theOptimistic Oracle V2 and why using the Internal Optimistic Oracle could be useful:

Optimistic Oracle V2 (OO) Internal Optimistic Oracle (IOO) Using the OO from an external contract incurs gas costs that, while not excessive, might be reduced with customized implementations like the IOO. Lower gas costs because price requests are processed locally. The requested data types from the OO are alwaysint256 . Customized data structures: the data requested from the oracle can be of any type (uint256 in this IOO implementation). The OO design is agnostic and adaptable to every use case, but adding more advanced functionality to price requests, such as defining who can propose values, requires some extra work. Additional customization is allowed: in the example implementation, a price request is coupled with themsg.sender , requiring the requester and proposer to be same. Disputes are resolved in theDVM through a vote of UMA token holders. Disputes are resolved in theDVM through a vote of UMA token holders. In this simple implementation, we highlight these functionalities, however, if you wish to go further, consider the following:

- The data type of the "price" (proposedPrice
-) could be any: structs, arrays, bytes. In the example, we choseuint256
- , but you can modify it to suit your requirements.(Note: For historical reasons, the data is referred to as a "price" throughout UMA's code, but can be any type of data, not just data related to asset prices.)
- Any further logic could be added to the requests and proposals, such as permitting only addresses on a whitelist to submit answers (proposePrice
- function).
-

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/InternalOptimisticOracle.sol (here) within the repo.

Contract creation and initialization

The constructor of the Internal Optimistic Oracle contract takes three parameters:

_finderAddress finder contract used to get addresses of other UMA contracts.

_currency the collateral token used to pay fees.

_timerAddress is used only when running unit tests locally to simulate advancement of time. For all the public networks (including testnets) zero address should be used.

```

Copy constructor( address_finderAddress, address_currency, address_timerAddress )Testable(_timerAddress) { 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. This address will vary depending on which chain this contract is deployed to.

Requesting a price

The following function allows you to request a price internally in the IOO. For simplicity theprice identifier has been hardcoded toYES_OR_NO_QUERY . This imposes a set of rules on how the assertion is formulated and formatted, which are specified inUMIP-107 .

TherequestPrice function takes the following arguments:

- timestamp
- timestamp used to identify the request, usually the current timestamp.
- ancillaryData
- the byte-converted question that we want to ask (e.g. 'q: "What uint256 are we looking for?"'
-)
- reward
- the amount of thecurrency
- defined in the constructor to pay to the proposer on settlement
- bond
- the bond in thecurrency
- defined in the constructor that we want to require on top of thefinalFee
- (~1500 USD worth of the currency). ThefinalFee
- can be viewed as the minimum bond required by the system to process a dispute, and the bond is any additional amount we to require on top.
- liveness
- time period during which the proposal can be disputed
-

Note: TherequestPrice function requires the caller to approve the IOO to spend thereward amount of thecurrency .

```

Copy functionrequestPrice( uint256timestamp, bytesmemoryancillaryData, uint256reward, uint256bond, uint64liveness )public{ bytes32requestId=_getId(msg.sender,timestamp,ancillaryData); require(address(requests[requestId].currency)==address(0),"Request already initialized"); require(_getIdentifierWhitelist().isIdentifierSupported(priceIdentifier),"Unsupported identifier"); require(_getCollateralWhitelist().isOnWhitelist(address(currency)),"Unsupported currency"); require(timestamp<=getCurrentTime(),"Timestamp in future");

requests[requestId]=Request({ proposer:address(0), disputer:address(0), currency:currency, settled:false, proposedPrice:0, reward:reward, finalFee:_getFinalFee(), bond:bond, liveness:liveness, expirationTime:0 });

if(reward>0) currency.safeTransferFrom(msg.sender,address(this),reward); }

```

Proposing a price

Once a price has been requested, it can be proposed.

These are the arguments thatproposePrice receives:

- timestamp
- timestamp used to identify the priceRequest, usually the current timestamp.
- ancillaryData
- the byte-converted question that we want to ask.
- proposedPrice
- the price proposed to the request
-

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

Note: Because of the_getId function definition, the requester and the proposer must be the same person.

```

Copy functionproposePrice( uint256timestamp, bytesmemoryancillaryData, uint256proposedPrice )public{
Requeststoragerequest=requests[_getId(msg.sender,timestamp,ancillaryData)];
require(address(request.currency)!=address(0),"Price not requested"); require(request.proposer==address(0),"Price already proposed");

request.proposer=msg.sender; request.proposedPrice=proposedPrice;
request.expirationTime=uint64(getCurrentTime())+request.liveness;

request.currency.safeTransferFrom(msg.sender,address(this),request.bond+request.finalFee); }

```

Disputing a price

Once a price has been proposed it can be disputed by callingdisputePrice , this function takes the following parameters:

- timestamp
- timestamp used to identify the price request, usually the current timestamp.
- ancillaryData
- the byte-converted question that has been asked initially and already answered by the proposer
-

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

Note: Because of the_getId function definition, the requester, proposer and disputer must be the same person.

```

Copy functiondisputePrice(uint256timestamp,bytesmemoryancillaryData)public{
bytes32requestId=_getId(msg.sender,timestamp,ancillaryData); Requeststoragerequest=requests[requestId];
require(request.proposer!=address(0),"No proposed price to dispute"); require(request.disputer==address(0),"Proposal already disputed"); require(uint64(getCurrentTime())<request.expirationTime,"Proposal past liveness");

request.disputer=msg.sender;

// If the final fee gets updated between the time the request is made and the time the dispute is made, the // disputer will have to pay the final fee increase x2. This is a very rare edge case that we are willing to accept.
uint256finalFeeRise=_getFinalFeeRise(request); uint256initialBalance=request.currency.balanceOf(address(this));

request.currency.safeTransferFrom( msg.sender, address(this), request.bond+request.finalFee+2*finalFeeRise );

request.currency.approve(address(oo),2*(request.bond+request.finalFee+finalFeeRise)+request.reward);

bytesmemorydisputeAncillaryData=_getDisputeAncillaryData(requestId);
oo.requestPrice(priceIdentifier,timestamp,disputeAncillaryData,request.currency,request.reward);
oo.setBond(priceIdentifier,timestamp,disputeAncillaryData,request.bond);
oo.proposePriceFor(request.proposer,address(this),priceIdentifier,timestamp,disputeAncillaryData,1e18);
oo.disputePriceFor(msg.sender,address(this),priceIdentifier,timestamp,disputeAncillaryData);

// If the final fee has decreased, refund the excess to the disputer & proposer.
uint256balanceToRefund=request.currency.balanceOf(address(this))>initialBalance ?
request.currency.balanceOf(address(this))-initialBalance :0; if(balanceToRefund>0)
request.currency.safeTransfer(msg.sender,balanceToRefund); }

```

Tests and deployment

All the unit tests covering the functionality described above are availablehere . To execute all of them, run:

```

Copy yarntesttest/InternalOptimisticOracle/*

```

Before deploying the contract check the comments on available environment variables inthe deployment script .

In the case of the Görli testnet, the defaults would use the Finder instance that references theMock 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 Internal Optimistic Oracle contract on Görli, run:

```

Copy NODE_URL_5=YOUR_GOERLI_NODEMNEMONIC=YOUR_MNEMONICyarnhardhatdeploy--networkgoerli--tagsInternalOptimisticOracle

```

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

Get the libraries and connect to the necessary contracts that we are going to use. In this tutorial we are usingTestnetERC20 as thecurrency as described in the deployment script.

```

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

constiooDeployment=awaitdeployments.get("InternalOptimisticOracle"); constioo=newethers.Contract( iooDeployment.address, iooDeployment.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( awaitioo.currency(), getAbi("TestnetERC20"), ethers.provider ); constmockOracle=newhre.ethers.Contract( awaitfinder.getImplementationAddress( ethers.utils.formatBytes32String("Oracle") ), getAbi("MockOracleAncillary"), ethers.provider );

```

Request a price and propose a price without disputes

First we need to mint thereward amount and approve the IOO to pull them. This amount ofcurrency tokens will be used to pay the proposer of the price request we are going to create.

```
```

```
Copy constreward=ethers.utils.parseUnits("100",18);

await(awaitcurrency.connect(signer).allocateTo(signer.address,reward)).wait();
await(awaitcurrency.connect(signer).approve(ioo.address,reward)).wait();
```

Request the price:

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

constbond=ethers.utils.parseUnits("500",18);

constliveness=10;// 10 seconds for testing purposes

constancillaryData=ethers.utils.toUtf8Bytes( q: "What uint256 are we looking for?"  );

await( awaitioo.connect(signer).requestPrice( requestTimestamp, ancillaryData, reward, bond, liveness ) ).wait();
```

Then we can propose a price to answer the question. The proposer needs to post a bond equal to therequest.bond + finalFee in order to do this so let's mint and approve the tokens as before:

```
Copy constfinalFee=awaitstore.computeFinalFee(currency.address); consttotalAmount=bond.add(finalFee.rawValue);

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

And propose the price:

```
Copy constcorrectAnswer=ethers.utils.parseEther("1"); await( awaitioo .connect(signer)
.proposePrice(requestTimestamp,ancillaryData,correctAnswer) ).wait();
```

Now we need to wait 10 seconds for the liveness period to be over and settle and get the price to verify the correct answer:

```
Copy await( awaitioo.connect(signer).settleAndGetPrice(requestTimestamp,ancillaryData) ).wait();

console.log( "Price is the correct answer: ", (awaitioo.getPrice(requestTimestamp,ancillaryData)).eq(correctAnswer) );
```

Request a price and propose a price with a dispute

After requesting and proposing a price we can dispute the price to escalate it to the DVM to be resolved by UMA voters. In this example we will be proposing a correct answer and disputing it, asking the DVM to determine the answer. The first steps are the same as before where we request a price and propose an answer:

```
Copy // get a new request timestamp requestTimestamp=await(awaitethers.provider.getBlock("latest")).timestamp;

// mint and approve the reward await(awaitcurrency.connect(signer).allocateTo(signer.address,reward)).wait();
await(awaitcurrency.connect(signer).approve(ioo.address,reward)).wait();

// request a new price constlongerLiveness=3600; await( awaitioo .connect(signer)
.requestPrice(requestTimestamp,ancillaryData,reward,bond,longerLiveness) ).wait();

// mint and approve the bond await( awaitcurrency.connect(signer).allocateTo(signer.address,totalAmount) ).wait();
await(awaitcurrency.connect(signer).approve(ioo.address,totalAmount)).wait();
```

```
// propose a price await( awaitioo .connect(signer) .proposePrice(requestTimestamp,ancillaryData,correctAnswer) ).wait();
```

Then we can dispute the price:

```
Copy // mint and approve the bond for the disputer await(
awaitcurrency.connect(signer).allocateTo(signer.address,totalAmount) ).wait();
await(awaitcurrency.connect(signer).approve(ioo.address,totalAmount)).wait();

// dispute the price constreceipt=await( awaitioo.connect(signer).disputePrice(requestTimestamp,ancillaryData) ).wait();
```

As we are in a test environment, we can simulate the vote that would normally occur by pushing a price to the mockOracle:

```
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, correctAnswer// The original price is accepted );
```

At this point, we can callsettleAndGetPrice to settle the accepted price request and then confirm that the result matches the proposed price:

```
Copy await( awaitioo.connect(signer).settleAndGetPrice(requestTimestamp,ancillaryData) ).wait();

console.log( "Price is the correct answer: ", (awaitioo.getPrice(requestTimestamp,ancillaryData)).eq(correctAnswer) );
```

Previous Deposit Box Next Insurance Claim Arbitration Last updated1 month ago On this page *The Internal Optimistic Oracle * Development environment and tests * Contract Implementation * Tests and deployment * Interacting with deployed contract

Was this helpful? Edit on GitHub