Insurance Claim Arbitration

Using the Optimistic Oracle to allow for verification of insurance claims This section covers the insurance claims arbitration contract , which is available in the developer's quick-start repo . This tutorial shows an example of how insurance claims can be resolved and settled through the Optimistic Oracle V2 contract.

You will find out how to test and deploy this smart contract and how it integrates with the Optimistic Oracle.

Insurance Arbitrator Contract

This smart contract allows insurers to issue insurance policies by depositing the insured amount, designating the insured beneficiary, and describing the insured event.

Anyone can submit a claim that the insured event has occurred at any time. Insurance Arbitrators resolve the claim through the Optimistic Oracle by passing a question with a description of the insured event in ancillary data using the YES_OR_NO_QUERY price identifier specified in UMIP-107 .

If the claim is confirmed and settled through the Optimistic Oracle, this contract automatically pays out insurance coverage to the beneficiary. If the claim is rejected, the policy continues to be active and ready for subsequent claim attempts.

Development environment and tests

Clone repository and Install dependencies

Clone the UMA 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/InsuranceArbitrator.sol (here) within the repo.

Contract creation and initialization

_finder parameter in the constructor points the Insurance Arbitrator to the Finder contract that stores the addresses of the rest of the UMA contracts. The Finder address can be fetched from the relevant networks file, if you are on a live network, or you can provide your own Finder instance if deploying UMA protocol in your own sandboxed testing environment.

_currency parameter in the constructor identifies the token used for settlement of insurance claims, as well as the bond currency for proposals and disputes. This token should be approved as whitelisted UMA collateral. Please check Approved Collateral Types for production networks or call getWhitelist() on the Address Whitelist contract for any of the test networks.

Alternatively, you can approve a new token address with addToWhitelist method in the Address Whitelist contract if working in a sandboxed UMA environment.

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

```

Copy constructor( FinderInterface_finder, address_currency, address_timer )Testable(_timerAddress) { finder=_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 throughgetImplementationAddress method in theFinder contract.

Issuing insurance

issueInsurance method allows any insurer to depositinsuredAmount ofcurrency tokens by designating an insurance beneficiary (insuredAddress ) and defining the insured event (insuredEvent ). Before calling this method, the insurer should have approved this contract to spend the required amount ofcurrency tokens.

```

Copy functionissueInsurance(stringcalldatainsuredEvent, addressinsuredAddress, uint256insuredAmount)externalreturns(bytes32policyId) ...

```

Internally, the issued policy is stored in theinsurancePolicies mapping using the calculatedpolicyId key that is generated by hashing the current block number with the provided insurance parameters in the internal_getPolicyId function.

After pullinginsuredAmount from the caller in theissueInsurance method, the contract emits aPolicyIssued event including thepolicyId parameter that should be used when claiming insurance.

Submitting insurance claim

Anyone can submit an insurance claim on the issued policy by calling thesubmitClaim method with the relevantpolicyId parameter. This method will initiate both a data request and proposal with the Optimistic Oracle. A proposal bond is required, hence the caller should have approved this contract to spend the required amount ofcurrency tokens for the proposal bond.

```

Copy functionsubmitClaim(bytes32policyId) ...

```

After checking that thepolicyId represents a valid unclaimed insurance policy, the contract gets the currenttimestamp and composesancillaryData that will be required for making requests and proposals to the Optimistic Oracle:

```

Copy uint256timestamp=getCurrentTime();//notethat getCurrentTime is exported from testable to enable easy time manipulation.
bytesmemoryancillaryData=abi.encodePacked(ancillaryDataHead,claimedPolicy.insuredEvent,ancillaryDataTail);
bytes32claimId=_getClaimId(timestamp,ancillaryData); insuranceClaims[claimId]=policyId;

```

The resultingtimestamp andancillaryData parameters are hashed in the internal_getClaimId method that is used as a key when storing the linkedpolicyId in theinsuranceClaims mapping. This information will be required when receiving a callback from the Optimistic Oracle.

The concatenatedancillaryData will have a valid question as specified inUMIP-107 forYES_OR_NO_QUERY price identifier.

An Optimistic Oracle data request is initiated without providing any proposer reward since the proposal will be done within thesubmitClaim method:

```

Copy oo.requestPrice(priceIdentifier,timestamp,ancillaryData,currency,0);

```

Before the proposal is made, the Optimistic Oracle allows the requesting contract (this Insurance Arbitrator) to set additional parameters like bonding, liveness, and callback settings. This requires passing the samepriceIdentifier ,timestamp , andancillaryData parameters to identify the request.

Total bond to be pulled from the claim initiator consists of the Optimistic Oracle proposer bond and final fee for the relevantcurrency token. This contract sets the proposer bond as a fixed percentage (constantoracleBondPercentage ) frominsuredAmount . When calling thesetBond method, the Optimistic Oracle calculates and returns the total bond that would be pulled when making the proposal:
```

```

Copy uint256proposerBond=(claimedPolicy.insuredAmount*oracleBondPercentage)/1e18;
uint256totalBond=oo.setBond(priceIdentifier,timestamp,ancillaryData,proposerBond);

```

Optimistic Oracle liveness is set by calling thesetCustomLiveness method. This contract uses 24 hours so that verifiers have sufficient time to check the claim, but one can adjust theoptimisticOracleLivenessTime constant for testing. (You probably don't want to wait a full day to resolve your test requests!)

```

Copy oo.setCustomLiveness(priceIdentifier,timestamp,ancillaryData,optimisticOracleLivenessTime);

```

In contrast to earlier versions, the Optimistic Oracle V2 by default does not use callbacks and requesting contracts have to explicitly subscribe to them if intending to perform any logic when a data request has changed state. Here, in callingsetCallbacks , this contract only subscribes to a callback for settlement, as implemented in thepriceSettled method. (Note: Subscribing to any other callbacks that are not implemented in the requesting contract would make data requests unresolvable.)

```

Copy oo.setCallbacks(priceIdentifier,timestamp,ancillaryData,false,false,true);

```

AftertotalBond amount ofcurrency token is pulled from the claim initiator and approved to be taken by Optimistic Oracle, this contract proposes1e18 representing an answer ofYES to the raised question. Requesting and proposing affirmative answers atomically allows us to reduce the number of steps taken by end users and it is most likely expected that the insured beneficiary would be initiating the claim.

```

Copy oo.proposePriceFor(msg.sender,address(this),priceIdentifier,timestamp,ancillaryData,int256(1e18));

```

Disputing insurance claim

For the sake of simplicity this contract does not implement a dispute method, but the disputer can dispute the submitted claim directly through Optimistic Oracle before the liveness passes by calling itsdisputePrice method:

```

Copy functiondisputePrice( addressrequester, bytes32identifier, uint256timestamp, bytesmemoryancillaryData ) ...

```

The disputer should pass the address of this Insurance Arbitrator contract asrequester and all the other parameters from the original request when the claim was initiated as emitted by the Optimistic Oracle in itsRequestPrice event.

If the claim is disputed, the request is escalated to the UMA DVM and it can be settled only after UMA voters have resolved it. To learn more about the DVM, see the docs section on the DVM:how does UMA's DVM work .

Settling insurance claim

Similar to disputes, claim settlement should be initiated through the Optimistic Oracle contract by calling itssettle method with the same parameters:

```

Copy functionsettle( addressrequester, bytes32identifier, uint256timestamp, bytesmemoryancillaryData ) ...

```

In case the liveness has expired or a dispute has been resolved by the UMA DVM, this call would initiate apriceSettled callback in the Insurance Arbitrator contract:

```
```

Copy functionpriceSettled(bytes32,// identifier passed by Optimistic Oracle, but not used here as it is always the same. uint256timestamp, bytesmemoryancillaryData, int256price) ...

```

Based on the received callback parameters, this contract can identify the relevantclaimId that is used to get the stored insurance policy:

```

Copy bytes32claimId=_getClaimId(timestamp,ancillaryData);

```

Importantly, all callbacks should be restricted to accept calls only from the Optimistic Oracle to avoid someone spoofing a resolved answer:

```

Copy require(address(oo)==msg.sender,"Unauthorized callback");

```

Depending on the resolved answer, this contract would either pay out the insured beneficiary and delete the insurance (in case of1e18 representing the answerYES , the insurance claim was valid) or reject the payout and re-open the policy for any subsequent claims:

```

Copy // Deletes insurance policy and transfers claim amount if the claim was confirmed. if(price==1e18) { deleteinsurancePolicies[policyId]; currency.safeTransfer(claimedPolicy.insuredAddress,claimedPolicy.insuredAmount);

emitClaimAccepted(claimId,policyId); // Otherwise just reset the flag so that repeated claims can be made. }else{ insurancePolicies[policyId].claimInitiated=false;

emitClaimRejected(claimId,policyId); }

```

Tests and deployment

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

```

Copy yarntesttest/InsuranceArbitrator/*

```

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 Insurance Arbitrator contract on Görli, run:

```

Copy NODE_URL_5=YOUR_GOERLI_NODEMNEMONIC=YOUR_MNEMONICyarnhardhatdeploy--networkgoerli--tagsInsuranceArbitrator

```

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_NODEMNEMONIC=YOUR_MNEMONICyarnhardhatconsole--networkgoerli

```

Initial setup

From the Hardhat console, start by adding the requiredgetAbi dependency for interacting with UMA contracts and use the first two accounts as insurer and insured beneficiary:

```

Copy const{getAbi}=require("@uma/contracts-node"); const[insurer,insured]=awaitethers.getSigners();

```

Grab the deployed Insurance Arbitrator contract:

```

Copy constinsuranceArbitratorDeployment=awaitdeployments.get("InsuranceArbitrator"); constinsuranceArbitrator=newethers.Contract( insuranceArbitratorDeployment.address, insuranceArbitratorDeployment.abi, ethers.provider );

```

Issue insurance

AssumingTestnetERC20 was used ascurrency when deploying, mint the required insurance amount (e.g. 10,000 TEST tokens) and approve the Insurance Arbitrator to pull them:

```

Copy constinsuredAmount=ethers.utils.parseEther("10000"); constcurrency=newethers.Contract(awaitinsuranceArbitrator.currency(),getAbi("TestnetERC20"),ethers.provider); await(awaitcurrency.connect(insurer).allocateTo(insurer.address,insuredAmount)).wait(); await(awaitcurrency.connect(insurer).approve(insuranceArbitrator.address,insuredAmount)).wait();

```

Issue the insurance policy and grab the resultingpolicyId from the emittedPolicyIssued event:

```

Copy constissueReceipt=await(awaitinsuranceArbitrator.connect(insurer).issueInsurance( "Bad things have happened", insured.address, insuredAmount )).wait(); constpolicyId=(awaitinsuranceArbitrator.queryFilter( "PolicyIssued", issueReceipt.blockNumber, issueReceipt.blockNumber ))[0].args.policyId;

```

Submit insurance claim

First calculate the expected proposer bond:

```

Copy constproposerBond=insuredAmount.mul(awaitinsuranceArbitrator.oracleBondPercentage()).div(ethers.utils.parseEther("1"));

```

Fetch the expected final fee from theStore contract (which is discovered through theFinder):

```
```

Copy constfinder=newethers.Contract(awaitinsuranceArbitrator.finder(),getAbi("Finder"),ethers.provider);
conststore=newethers.Contract(awaitfinder.getImplementationAddress(ethers.utils.formatBytes32String("Store")),
getAbi("Store"), ethers.provider); constfinalFee=(awaitstore.computeFinalFee(currency.address)).rawValue;

```

Calculate the expected total bond and provide funding/approval for the insured claimant:

```

Copy consttotalBond=proposerBond.add(finalFee);
await(awaitcurrency.connect(insured).allocateTo(insured.address,totalBond)).wait();
await(awaitcurrency.connect(insured).approve(insuranceArbitrator.address,totalBond)).wait();

```

Now initiate the insurance claim and grab request details from theRequestPrice event emitted by the Optimistic Oracle:

```

Copy constoo=newethers.Contract(awaitinsuranceArbitrator.oo(),getAbi("OptimisticOracleV2"),ethers.provider);
constclaimReceipt=await(awaitinsuranceArbitrator.connect(insured).submitClaim(policyId)).wait(); constrequest=
(awaitoo.queryFilter("RequestPrice",claimReceipt.blockNumber,claimReceipt.blockNumber))[0].args;

```

Dispute insurance claim

Before liveness passes, the insurer can dispute the claim through the Optimistic Oracle. First, they must fund and approve with the same bonding amount:

```

Copy await(awaitcurrency.connect(insurer).allocateTo(insurer.address,totalBond)).wait();
await(awaitcurrency.connect(insurer).approve(oo.address,totalBond)).wait();

```

If you are on a testnet like Göerli, in order to simulate UMA voting on a testnet, you can use theMock Oracle :

```

Copy constmockOracle=newethers.Contract(awaitfinder.getImplementationAddress(
ethers.utils.formatBytes32String("Oracle")), getAbi("MockOracleAncillary"), ethers.provider);

```

Now initiate the dispute and grab the vote request details from thePriceRequestAdded event emitted by the Mock Oracle:

```

Copy constdisputeReceipt=await(awaitoo.connect(insurer).disputePrice(request.requester, request.identifier,
request.timestamp, request.ancillaryData)).wait(); constvoteRequest=(awaitmockOracle.queryFilter("PriceRequestAdded",
disputeReceipt.blockNumber, disputeReceipt.blockNumber))[0].args;

```

Settle insurance claim

Before settling the claim, we can take a look at the vote request as seen by UMA voters:

```

Copy console.log("identifier:",ethers.utils.parseBytes32String(voteRequest.identifier));
console.log("time:",Number(voteRequest.time));
console.log("ancillaryData:",ethers.utils.toUtf8String(voteRequest.ancillaryData));

```

TheancillaryData should start withq:"Had the following insured event occurred as of request timestamp: Bad things have happened?" . It is then followed by theooRequester key with our Insurance Arbitrator address in its value.
```

In order to simulateYES as the resolved answer we would pass1e18 as theprice parameter in the Mock OraclepushPrice method:

```

Copy await(awaitmockOracle.connect(insured).pushPrice( voteRequest.identifier, voteRequest.time, voteRequest.ancillaryData, ethers.utils.parseEther("1") )).wait();

```

Now we can settle the request through the Optimistic Oracle and observe the emittedClaimAccepted from our Insurance Arbitrator contract:

```

Copy constsettleReceipt=await(awaitoo.connect(insured).settle( request.requester, request.identifier, request.timestamp, request.ancillaryData )).wait(); constclaimSettlementEvent=(awaitinsuranceArbitrator.queryFilter( "ClaimAccepted", settleReceipt.blockNumber, settleReceipt.blockNumber ))[0]; console.log(claimSettlementEvent);

```

The above settlement transaction should also transferinsuredAmount tokens to the insured beneficiary as well as return the proposer bond to the claim initiator.

Alternatively, if0 value was resolved, the settlement transaction should emit theClaimRejected event without paying out theinsuredAmount and returning the bond to the disputer, along with half of the proposer's bond.

Previous Internal Optimistic Oracle Next Optimistic Arbitrator Last updated1 month ago On this page *Insurance Arbitrator Contract * Development environment and tests * Contract implementation * Tests and deployment * Interacting with deployed contract