Creating a Fallback Oracle with Uniswap & Tellor

Christopher Pondoc

cpondoc@tellor.io

Abstract

In this litepaper, I describe an implementation for designing a fallback oracle that initially grabs value from a Uniswap liquidity pool, and then falls back to the Tellor oracle depending on certain criteria (value discrepancy, data freshness, and amount of liquidity in pools). To determine when to fall back to Tellor, the user of the oracle sets the levers for the different criteria, allowing full customization of the data retrieval experience.

1 Introduction

Uniswap's automated market maker (AMM) and associated liquidity pools can be used to retrieve relative values between two ERC-20 tokens. In practice, Uniswap can be utilized as an oracle for smart contracts. However, the data that comes from such an oracle can be unreliable. For instance, sometimes there is too little liquidity in a pool, which raises obvious security threats but also makes for a relatively bad data point. Thus, one should always have a fallback in the case that Uniswap's data is considered "bad."

In my case, I decided to utilize the Tellor oracle as a fallback. I decided to fallback on three separate criteria, each of which could be specified by the user for each call:

- Liquidity: Is there enough liquidity in the Uniswap Pool for the given price feed to have a reliable value?
- Data Freshness: Is the data recent enough – against Tellor's retrieved value – in order for a reliable value?
- Threshold: Is the given price close enough to the value of Tellor's to merit a reliable value?

2 Smart Contract Deep Dive

Let's take a look at how I implemented such an oracle from a high-level design.

2.1 Constructor

First and foremost, the contract inherits the 'UsingTellor' smart contract. The contract's constructor consists of an address of an eligible Tellor oracle contract – for testing purposes, this would often be an instance of Tellor playground – a mapping of data IDs, and a corresponding mapping of smart contract addresses for each liquidity pool. The constructor requires that the amount of data IDs and contracts are equal, and then assigns the internal mapping of price feeds to liquidity pools using a loop.

```
/**
 * @dev Sets up respective addresses + pools, and also creates the mapping of
↪  price feeds
 * @param _tellorAddress the address of the tellor contract
```

```
 * @param _dataIds a list of data IDs
 * @param _contracts a list of corresponding Uniswap pool contracts
 */
constructor(address payable _tellorAddress, uint[] memory _dataIds, address[]
↪  memory _contracts)
UsingTellor(_tellorAddress)  {
  // Length of Data IDs and Contracts should be the same
  require(_dataIds.length == _contracts.length, "Data IDs and Contracts are not
  ↪  same length");
  for (uint i = 0; i < _dataIds.length; i++) {
    priceFeeds[_dataIds[i]] = _contracts[i];
  }
}
```

2.2 'grabNewValue' Function Header

Majority of the contract's internal logic resides in 'grabNewValue', which is what a user would call to get a new value from the oracle. The function's parameters include an eligible data ID, and the levers for each criteria: one for liquidity, one for data freshness, and a last one for threshold.

```
function grabNewValue(uint256 _dataId, uint128 _liquidityBound, uint256
↪  _timeDifference, uint256 _percentDifference) external view returns (uint256,
↪  uint256, uint)
```

2.3 Uniswap Data

The function first grabs data from Uniswap: it defines a IUniswapV3Pool using the corresponding pool contract address from the 'priceFeeds' mapping, and passes the pool into the function 'grabUniswapData'. Within the 'grabUniswapData' function, 'slot0' is called on the Uniswap pool, and a series of mathematical operations are performed on the returned square root price in order to grab the value from Uniswap's oracle. Similarly, by using the index of the observation of the latest pool price, the function also returns the timestamp.

```
function grabUniswapData(IUniswapV3Pool _pool) internal view returns (uint256,
↪  uint256) {
  // Get current state of the Uniswap Pool and retrieve price
  (uint160 sqrtPriceX96,, uint16 observationIndex,,,,) = _pool.slot0();
  uint256 uniswapPrice = uint(sqrtPriceX96)
    .mul(uint(sqrtPriceX96))
    .mul(1e10) >> (96 * 2);

  // Find corresponding Uniswap observation to get timestamp
  (uint32 blockTimestamp,,, bool initialized) =
  ↪  _pool.observations(observationIndex);
  if (!initialized) return (0, 0);

  return (uniswapPrice, uint256(blockTimestamp));
}
```

2.4 Tellor Data

Subsequently after, the function grabs the current value from Tellor's oracle using 'grabTellorData'. This function utilizes the 'getCurrentValue' from 'UsingTellor', which returns both the oracle value and the timestamp. If these values are not able to be retrieved, the function returns two zeroes; else, the updated value are returned.

```
function grabTellorData(uint256 _dataId) internal view returns (uint256,
↪  uint256) {
  (bool ifRetrieve, uint256 value, uint256 _timestampRetrieved) =
  ↪  getCurrentValue(_dataId);
  if (!ifRetrieve) return (0, 0);
  return (value, _timestampRetrieved);
}
```

2.5 Calculation of Criteria and Thresholds

The final part of the function involves the three different criteria: a conditional statement checks whether or not the liquidity is within the proper bounds, the value is within the threshold, and if the value is fresh enough. Comparing liquidity and timestamps are relatively straightforward (simple less than or greater than comparison); determining if a value is within a specific threshold is different. At its mathematical core, a percentage is a fraction equal to the percent number divided by 100, or $\frac{p}{100}$, where p is the percent number. Similarly, the percentage difference of the Uniswap and Tellor values is equal to $\frac{|u-t|}{u}$, where u = Uniswap's value and t = Tellor's value. Since we want to compare these values, we have:

$$\frac{p}{100} < \frac{|u-t|}{u}$$

Finally, rather than calculating a percentage difference – which is not optimal, since decimals are not allowed in Solidity – we can use cross-multiplication to derive another boolean to return:

$$p \cdot u < 100 \cdot |u-t|$$

These are implemented in practice below:

```
/**
 * @dev Determines if a value from Uniswap is within a specific percentage range
 * of Tellor's value
 * @param _uniswapValue value of a specific price id from Uniswap pool
 * @param _tellorValue value of a specific price id from Tellor oracle
 * @param _percentLever the whole number percentage of how close the values
↪  should be
 * @return bool if value is within threshold
 */
function isWithinThreshold(uint256 _uniswapValue, uint256 _tellorValue, uint256
↪  _percentLever) internal pure returns (bool) {
  // Calculate difference between the two values
  uint256 valueDifference = unsignedDifference(_uniswapValue, _tellorValue);

  // Determine if within or outside of threshold
  return valueDifference.mul(100) < _percentLever.mul(_uniswapValue);
}

/**
 * @dev Determines if a value from Uniswap is fresh enough to use against
↪  Tellor's
 * @param _uniswapTime timestamp of value from Uniswap pool
 * @param _tellorTime timestamp of value from Tellor oracle
 * @param _timeLever how fresh the data should be
 * @return bool if value is fresh enough
 */
```

```
function isWithinTime(uint256 _uniswapTime, uint256 _tellorTime, uint256
↪  _timeLever) internal pure returns (bool) {
  // Determine time difference
  uint256 timeChange = unsignedDifference(_uniswapTime, _tellorTime);

  // Check if data is fresh compared to expected lever
  return timeChange < _timeLever;
}
```

2.6 Full Function

Finally, the function returns the appropriate value. The full function is described below.

```
function grabNewValue(uint256 _dataId, uint128 _liquidityBound, uint256
↪  _timeDifference, uint256 _percentDifference) external view returns (uint256,
↪  uint256, uint) {
  // Set up Uniswap Pool and retrieve respective values
  IUniswapV3Pool uniswapPool = IUniswapV3Pool(address(priceFeeds[_dataId]));
  (uint256 uniswapPrice, uint256 uniswapTimestamp) =
  ↪  grabUniswapData(uniswapPool);

  // Retrieve Tellor Value
  (uint256 tellorPrice, uint256 tellorTimestamp) = grabTellorData(_dataId);

  // Checking if values are close enough together
  if (uniswapPool.liquidity() < _liquidityBound ||
  ↪  !isWithinThreshold(uniswapPrice, tellorPrice, _percentDifference)
    || !isWithinTime(uniswapTimestamp, tellorTimestamp, _timeDifference)) {
    return (tellorPrice, tellorTimestamp, 2);
  }

  return (uniswapPrice, uniswapTimestamp, 1);
}
```

2.7 Testing Methodology

My testing methodology for the Uniswap fallback oracle was to use TellorPlayground in accompaniment with small enough liquidity pools to manipulate the values. From there, I tested all require statements, and also looked in-depth at each of the three criteria: liquidity, data freshness, and the value thresholds. Naturally, this was a relatively difficult process, and I am always open to more feedback!

Thank you to Nick Fett, Tally Wiesenberg, and Brenda Loya for providing feedback and helping me brainstorm the Oracle design. And of course, thank you to the entire Tellor team for the opportunity to work with them during the Summer of 2021.

3 Sources

Hayden Adams, "Uniswap v3 Core" Uniswap, v3, March 2021. [Online serial].
Available: https://uniswap.org/whitepaper-v3.pdf.

4 Contact

- For more information on my work, check out at github.com/cpondoc.

- For more information on Tellor, check out tellor.io.
- This project idea came from the Tellor Bounties page. To check out other bounties, go to tellor.io/bounties.