Oracle Specification

Compound Engineering

Abstract

The Oracle at Compound take real-life price data (from exchanges, etc) and posts the prices to the Compound Money Market. This is a crucial aspect of Compound because if the prices differ too much from real-life, the protocol has the risk of losing equity or assets. This document specifies version 1.0 of the protocol, which we plan to deploy at launch.

Considerations

- The Compound protocol relies on a single *aggregated* price for each asset, which should answer the question “what is an asset’s approximate fair value”
- We should source data from multiple high volume exchanges *for each asset* to determine the price. We can probably rely on one exchange for a fair price, but this has risk.
- The oracle system needs to be resilient -- if a price gets misreported (off by 10) in some way, we don’t lose all equity. We can do this by capping / limiting price movements (e.g. 20% per oracle period, say 5 minutes, though this could “drift” up to being existentially wrong in a span of an hour), or by excluding data sources that seem wrong in some way (if two exchanges are off by more than 5% in price, we could use the price that’s closer to the legacy price).
- Don’t forget, we can always start simple (centralized, posting prices) and upgrade to a more advanced system post launch.

Simple Oracle with Dampening and C.H.A.D.

We will build an oracle that has the ability to directly set the prices via standard signed Ethereum transactions. However, we insert strict limits on the ability to move prices

over a given threshold. We require a simple human interaction to verify larger price movements prior to accepting updated prices.

In a 🌰🐚

The poster (i.e. a machine running a price gathering script) constantly pulls the price for all assets relative to Ether, combining on its own the price from all available sources (e.g. Bitrex, Binance, Poloniex, etc). Every so often it decides to post the price for an asset to the blockchain, which it does by calling `setPrice(asset, price)` with an updated price. Assuming the price is within normal bounds (explained below), the price is accepted and stored in the Oracle. At the beginning of every hour, the price is stored as a new anchor price, and every price for the rest of the hour must be within 10% of that price.

All new prices must be within 10% of the anchor price, which is set every period (one period should be about an hour). For example, if the price is successfully set to 1.5 OMG/ETH at the beginning of the hour, then for the rest of the hour, the price is capped between 1.35 and 1.65 OMG/ETH for that hour period. Any attempts to move that price outside of these bounds would be capped to the nearest allowed bound. At the beginning of the next hour, the first accepted price (e.g. in the bounds of 1.35 to 1.65) becomes the new anchor for that hour. Thus, if the that price were set to 1.60, then the new bounds would be 1.44 to 1.76 OMG/ETH for the rest of the hour. This guarantees the price will never move more than 10% per hour (see asterisks below).

What happens when the price (in real life) moves above the bounds of 1.65 OMG/ETH? We have a specific hardware-key (controlled by a human, "Chad"). This key has the ability to set a new pending anchor point, which on the next price update, will be set as the new anchor. Thus, Chad could set the pending anchor point for OMG to be 1.70 OMG/ETH. When a new price comes in, say 1.75 OMG/ETH, it would check against the pending anchor point, store itself as the new anchor point, and delete the old anchor point (along with the pending anchor itself). This allows Chad to move the price anchoring, but not control the price himself.

Implementation

Oracle Variables

- *"poster" - Ethereum key stored on machine*
- *"chad" - Hardware key held by human*
- *maxSwing* on the interval (0..1) representing a percentage
- *blocksPerPeriod*
- *assetPrices: Asset -> Price*
- *anchors*: Asset -> (Period, Price)
- *pendingAnchor*: Asset -> Maybe Price

Functions

setPrice(asset, price)

- Fail when msg.sender is not *poster*
- Let **currentPeriod** = (blockNumber / *blocksPerPeriod)* + *1*
    - The current period will start pretty high (e.g. if blocks per period is 200, then we're current in the 29590th period)
- If *pendingAnchor*[asset]
    - **anchorPeriod** = 0
    - **anchorPrice** = pendingAnchor[asset]
    - This is, pending anchor allows us to reset the anchor to a new value, ignoring the old anchor
    - Let **swing** = $abs(price - anchorPrice) \div anchorPrice$
        - Swing the percentage difference between the new price and the anchor's price, which we verify against *maxSwing*
    - Fail when **swing** > *maxSwing*
- Else
    - (**anchorPeriod**, **anchorPrice**) = anchors[asset]
    - That is, we use whatever the last anchor for this asset was.
    - If **anchorPeriod** is not zero
        - Cap to max (see derivation of this formulae below):
        - If **price** > $anchorPrice \times (1 + maxSwing)$
            - Set **price** = $anchorPrice \times (1 + maxSwing)$

        - If **price** < $anchorPrice \times (1 - maxSwing)$
            - Set **price** = $anchorPrice \times (1 - maxSwing)$
    - Else
        - **anchorPrice** = **price**
        - Setting first price. Accept as is.
- Fail if **anchorPrice** or **price** is zero
- Set *pendingAnchor* = Nothing
    - Pending anchor is only used until an in-range price is received, then that price becomes the new anchor
- If **currentPeriod** > **anchorPeriod**:
    - Set *anchors*[asset] = (currentPeriod, price)
    - The new anchor is if we're in a new period or we had a pending anchor, then we become the new anchor
- assetPrices[asset] = price

setPendingAnchor(asset, price)

- Fail when msg.sender is not *chad*
- Set *pendingAnchor*[asset] = price

Notes

- Asterisks: in certain adverse scenarios, the price could move 20% or 30% in a short period. This is still below the danger limit.

Appendex A: Proof that cap to max is within the bounds of max swing

We will prove that if we set the price to $anchorPrice \times (1 + maxSwing)$ or $anchorPrice \times (1 - maxSwing)$ then, based on that price **swing** will always be within $maxSwing$.

- From the formula, **swing** = $abs(price - anchorPrice) \div anchorPrice$
- We aim to prove the cases where **swing** = *maxSwing*, thus:
- $maxSwing = abs(x - anchorPrice) \div anchorPrice$

- To solve this equation, we multiply both sides by $anchorPrice$
- $anchorPrice \times maxSwing = abs(x - anchorPrice)$
- The abs could be positive or negative, we handle the cases separately.
- When the abs(...) term has a positive operand:
  - $anchorPrice \times maxSwing = x - anchorPrice$
  - $anchorPrice + anchorPrice \times maxSwing = x$
  - $x = anchorPrice \times (1 + maxSwing)$
  - q.e.d.
- When the abs(...) term has a negative operand:
  - $anchorPrice \times maxSwing = -1 \times (x - anchorPrice)$
  - $anchorPrice \times maxSwing = anchorPrice - x$
  - $x + anchorPrice \times maxSwing = anchorPrice$
  - $x = anchorPrice - anchorPrice \times maxSwing$
  - $x = anchorPrice \times (1 - maxSwing)$
  - q.e.d.