Orb: Decentralized and Sustainable Oracles

Zahnentferner
The Stable Order

Sarthak Dengre
The Stable Order

Luca D'Angelo
zenGate Global
The Stable Order

Abstract

This paper presents the Orb Oracle Protocol. Orb Oracles are decentralized in the sense that any oracle token holder can submit values and thus can act as an oracle operator. And they are sustainable in the sense that oracle operators are rewarded for submitting values and thus have a source of revenue to cover their operational costs.

2012 ACM Subject Classification Applied computing → Economics

Keywords and phrases Blockchain, cryptocurrency, staking, reward distribution

Acknowledgements We thank the community of the Stability Nexus for their feedback and support.

1 Introduction

The great promise of *smart contracts* is to allow people to implement programs that automatically perform predefined actions when agreed conditions are satisfied. When smart contracts are deployed on decentralized blockchains, their execution is guaranteed to occur without reliance on a central authority above, or a single intermediary between, the people interested in the execution of the smart contract.

However, some of the most interesting applications of smart contracts require data that is external to the blockchain. For example, crypto-backed stablecoin [2] protocols (e.g. [1, 3, 4]) need to know the price of the underlying reserve asset in relation to the peg asset, in order to know how much of the reserve asset to give to a user when the user redeems an amount of stablecoins; a prediction market protocol needs to know the outcome of an event, in order to be able to transfer funds to the users who correctly predicted the outcome.

The components that allow data to be fetched from various sources and submitted to blockchains, to be used by smart contracts, are commonly called *Oracles*. Clearly, for an application to be fully decentralized, the oracles on which it depends need to be decentralized as well.

Despite their criticality, most oracles available nowadays are surprisingly centralized. Furthermore, the centralized entities operating these oracles often rely on venture capital, or grants from the blockchains on which they deploy their oracles, to fund their operations. This raises the concern of long-term sustainability of these oracles, since their future is unclear if their operators run out of capital[1].

Immutable, unstoppable, decentralized applications need a solid platform where every component on which they depend is sustainable and decentralized as well. This paper describes Orb, a decentralized and sustainable oracle protocol, aimed at providing a reliable,

[1] This is not only a hypothetical concern. For example, the oracle of the Milkomeda EVM sidechain of Cardano simply halted, when dcSpark decided to stop operating it due to financial considerations.

resilient, robust and anti-fragile foundation for truly decentralized applications that require real-world data.

2 Protocol Description

Orb is a protocol for the creation, operation and use of oracles where the data are temporal streams of values. Examples of temporal streams of values include: price of an asset over time; economic indicators (e.g. inflation indexes, base interest rates, unemployment, ...) over time; weather variables (e.g. temperature, rainfall, ...) in a location over time....

Every Orb oracle is associated with a fungible *oracle token*. Any oracle token holder can deposit oracle tokens into the oracle and become an *oracle operator*. The amount of tokens an operator has deposited is the operator's *stake*. The deposit and withdrawal of tokens, to increase or decrease one's stake, are subject to locking periods. During a deposit, the tokens are considered staked only after a *deposit-locking period* has elapsed. Likewise, an oracle operator can only withdraw tokens, after a *withdrawal-locking period* has elapsed since the last interaction of the oracle operator with the oracle.

Every oracle operator may submit values to the oracle. The oracle calculates a weighted average aggregate of the latest values submitted by all oracle operators. This weighted average takes into account the elapsed time of each submitted value, subjecting it to an exponential decay, and the stake of the oracle operator who submitted the value.

Anyone can fund an oracle with a *reward token* (typically, the underlying blockchain's native cryptocurrency). The balance of the oracle is a *reward pool* from which oracle operators are rewarded for every value submission. The reward for each value submission is calculated in a way that operators with higher stake and who submit values regularly (but not too frequently) receive higher rewards, since their submitted values contribute more to the average.

Oracle consumers are users or smart contracts who read data from the oracle. By default, anyone is a whitelisted oracle consumer. Oracle operators may vote to blacklist or whitelist oracle consumers, with voting power linearly proportional to their stake. If the weight of blacklist votes exceeds the weight of whitelist votes for an oracle consumer and certain quorum conditions are satisfied, the oracle consumer is disallowed to read data from the oracle.

3 Protocol Specification

Every oracle created via the Orb protocol has immutable parameters, chosen by the oracle creator at the moment of deployment, and state variables, which vary as users interact with the oracle after deployment.

An oracle's parameters are a tuple of constants

$$C = (h, q, \Delta_{\text{dep}}, \Delta_{\text{wd}}, \alpha, \tau_w, \tau_r) \tag{1}$$

where:

- $h > 0$ is the half-life constant.
- q is the quorum constant.
- Δ_{dep} is the deposit-locking period.
- Δ_{wd} is the withdrawal-locking period.
- α is a reward factor determining which fraction of the oracle's balance is paid as reward.

- τ_w is an identifier of the oracle token used for computing weights of oracle operators.
- τ_r is an identifier of the reward token.

An oracle's state is a tuple of variables

$$V = \big(L_\ell, L_f, T_{\text{dep}}, T_{\text{op}}, P_{\text{user}}, W_{\text{user}}, T_{\text{user}}, \bar{p}, \tilde{p}, Q, \mathcal{P}, \lambda, \mathcal{G}\big), \tag{2}$$

where:

- U is the set of users (people and smart contracts) that can interact with the oracle.
- $L_\ell, L_f : U \to \mathbb{R}_{\geq 0}$ track the locked and unlocked token balances (`lockedTokens` and `unlockedTokens`)[2].
- $T_{\text{dep}}, T_{\text{op}} : U \to \mathbb{R}_{\geq 0}$ store the last deposit timestamp and the last operation timestamp.
- $P_{\text{user}} : U \to \mathbb{Z}$ stores each submitter's last reported value (`pof`).
- $W_{\text{user}} : U \to \mathbb{R}_{\geq 0}$ stores the last submission weight of each user (`wof`).
- $T_{\text{user}} : U \to \mathbb{R}_{\geq 0}$ stores the last submission time of each user (`tof`).
- $\bar{p} \in \mathbb{R}$ is the stored aggregate value (`aggregatedValue`).
- $\tilde{p} \in \mathbb{R}$ is the latest submitted value (`latestValue`).
- $Q \in \mathbb{R}_{\geq 0}$ is the stored aggregate weight (`aggregatedWeight`).
- $\mathcal{P} = \big((t_1, \bar{p}_1, \tilde{p}_1), \ldots, (t_m, \bar{p}_m, \tilde{p}_m)\big)$ is the time-ordered value history[3].
- $\lambda = (t_{\text{sub}}, t_{\text{last}}, L_{\text{tot}})$ stores the last value submission time, the last oracle interaction time, and the current total of deposited tokens.
- $\mathcal{G} = \big(\mathcal{B}, V_{\text{black}}, V_{\text{white}}, M_{\text{black}}, M_{\text{white}}, \mathcal{W}_{\text{black}}, \mathcal{W}_{\text{white}}\big)$ is the governance state, where:
  - $\mathcal{B} : U \to \{0, 1\}$ the blacklist indicator,
  - $V_{\text{black}}, V_{\text{white}} : U \to \mathbb{R}_{\geq 0}$ the accumulated blacklist and whitelist weights,
  - $M_{\text{black}}, M_{\text{white}} : U \to 2^U$ the sets of targets each voter has supported,
  - $\mathcal{W}_{\text{black}}, \mathcal{W}_{\text{white}} : U \times U \to \mathbb{R}_{\geq 0}$ the per-target stored weights.

The user balances of oracle tokens and reward tokens outside are denoted by $B_{\text{user}} : U \times \{\tau_w, \tau_r\} \to \mathbb{R}_{\geq 0}$, whereas the balances of the oracle itself is denoted by $B_{\text{oracle}} : \{\tau_w, \tau_r\} \to \mathbb{R}_{\geq 0}$.

3.1 Auxiliary Definitions

▶ **Definition 1** (Exponential Decay Factor). *For $\Delta \geq 0$, the exponential decay factor is defined as:*

$$\delta(\Delta) = 2^{-\Delta/h} \tag{3}$$

▶ **Definition 2** (Time-Dependent Functions). *Given the current state V and time t, the following time-dependent functions are defined:*

- *The* decayed weight *of user u at time t is* $W_{\text{decayed}}(u, t) = W_{\text{user}}(u) \cdot \delta\big(t - T_{\text{user}}(u)\big)$.
- *The* decayed aggregate weight *at time t is* $Q_{\text{decayed}}(t) = Q \cdot \delta(t - t_{\text{sub}})$.
- *The* decayed aggregate value *at time t is* $\bar{p}_{\text{decayed}}(t) = \bar{p} \cdot \delta(t - t_{\text{sub}})$.
- *The* ideal decayed weighted mean *at time t is*

$$P(t) = \frac{\sum_{x \in U} P_{\text{user}}(x) \cdot W_{\text{decayed}}(x, t)}{\sum_{x \in U} W_{\text{decayed}}(x, t)}. \tag{4}$$

[2] The names between parentheses are the names of the state variables in our reference implementation in `https://github.com/StabilityNexus/OrbOracle-Solidity` and are intended to ease the mapping between the mathematical specification and the reference implementation.

[3] We write $\mathsf{append}(\mathcal{P}, x)$ for appending a triple x to the ordered value history.

3.2 Auxiliary State Update Procedures

The following procedures are used by many operations specified in subsection 3.3. The *unlock* procedure is called by any operation that requires knowing the current unlocked oracle token balance of the user. The *recompute* procedure is called whenever users vote to blacklist or whitelist. The *reweight* procedure updates the weights of every blacklist/whitelist vote cast by a user. It is called automatically when users withdraw or on demand by users when they deposit and explicitly synchronize their weights[4].

$$\text{Unlock}(u,t): \begin{cases} \text{if } t \geq T_{\text{dep}}(u) + \Delta_{\text{dep}} \text{ and } L_\ell(u) > 0 \text{ then} \\ \quad L_f(u) \leftarrow L_f(u) + L_\ell(u) \\ \quad L_\ell(u) \leftarrow 0 \\ \text{else} \\ \quad \text{skip} \end{cases} \tag{5}$$

$$\text{Recompute}(x): \begin{cases} \text{if } V_{\text{black}}(x) - V_{\text{white}}(x) > q \cdot \left(L_{\text{tot}} - (V_{\text{black}}(x) + V_{\text{white}}(x))\right) \text{ then} \\ \quad \mathcal{B}(x) \leftarrow 1 \\ \text{else} \\ \quad \mathcal{B}(x) \leftarrow 0 \end{cases} \tag{6}$$

$$\text{Reweight}(u,w): \begin{cases} \forall x \in M_{\text{black}}(u): \\ \quad V_{\text{black}}(x) \leftarrow V_{\text{black}}(x) - \mathcal{W}_{\text{black}}(x,u) + w \\ \quad \mathcal{W}_{\text{black}}(x,u) \leftarrow w \\ \quad \text{Recompute}(x) \\ \forall x \in M_{\text{white}}(u): \\ \quad V_{\text{white}}(x) \leftarrow V_{\text{white}}(x) - \mathcal{W}_{\text{white}}(x,u) + w \\ \quad \mathcal{W}_{\text{white}}(x,u) \leftarrow w \\ \quad \text{Recompute}(x) \end{cases} \tag{7}$$

3.3 Protocol Operations

Each operation defined in the folowing subsections is an action that a user may perform to interact with an oracle. The subsections define how they update the oracle's state.

[4] Reweight is an expensive procedure. This is the reason why it is called automatically only on withdrawals, when it is necessary, but only on explicit demand by users after deposits, if they wish.

3.3.1 Token Deposit

For $a > 0$, the deposit operation $\delta_{\text{dep}} : U \times \mathbb{R}_{>0} \times \mathbb{R}_{\geq 0} \to V \to V$ updates state as follows:

$$\delta_{\text{dep}}(u, a, t)(V) = \begin{cases} \text{Unlock}(u, t) \\ L_\ell(u) \leftarrow L_\ell(u) + a \\ T_{\text{dep}}(u) \leftarrow t \\ T_{\text{op}}(u) \leftarrow t \\ L_{\text{tot}} \leftarrow L_{\text{tot}} + a \\ t_{\text{last}} \leftarrow t \\ B_{\text{user}}(u, \tau_w) \leftarrow B_{\text{user}}(u, \tau_w) - a \\ B_{\text{oracle}}(\tau_w) \leftarrow B_{\text{oracle}}(\tau_w) + a, \end{cases} \tag{8}$$

3.3.2 Token Withdrawal

For $a > 0$ with $L_f(u) \geq a$ and $T_{\text{op}}(u) + \Delta_{\text{wd}} \leq t$, the withdrawal operation $\chi : U \times \mathbb{R}_{>0} \times \mathbb{R}_{\geq 0} \to V \to V$ updates the state as follows:

$$\chi(u, a, t)(V) = \begin{cases} \text{Unlock}(u, t) \\ L_f(u) \leftarrow L_f(u) - a \\ L_{\text{tot}} \leftarrow L_{\text{tot}} - a \\ T_{\text{op}}(u) \leftarrow t \\ t_{\text{last}} \leftarrow t \\ B_{\text{user}}(u, \tau_w) \leftarrow B_{\text{user}}(u, \tau_w) + a \\ B_{\text{oracle}}(\tau_w) \leftarrow B_{\text{oracle}}(\tau_w) - a, \end{cases} \tag{9}$$

3.3.3 Value Submission

Only token holders with unlocked stake may submit values (`submitValue`). Let $v \in \mathbb{Z}$ denote the new submission and t the current timestamp. Set $w = L_f(u)$, $p_u = P_{\text{user}}(u)$, $w_u = W_{\text{user}}(u)$, and $t_u = T_{\text{user}}(u)$. Using the decay factor (3), we compute the updated aggregate weight and value as:

$$Q' = \big(Q - w_u \cdot \delta(t_{\text{sub}} - t_u)\big) \cdot \delta(t - t_{\text{sub}}) + w, \tag{10}$$

$$\bar{p}' = \frac{\big(\bar{p} \cdot Q - p_u \cdot w_u \cdot \delta(t_{\text{sub}} - t_u)\big) \cdot \delta(t - t_{\text{sub}}) + v \cdot w}{Q'}. \tag{11}$$

The contract pays out:

$$n = \alpha \cdot B_{\text{oracle}}(\tau_r) \cdot \frac{w}{Q'} \cdot \big(1 - \delta(t - t_u)\big). \tag{12}$$

The value submission operation $\phi : U \times \mathbb{Z} \times \mathbb{R}_{\geq 0} \to V \to V$ updates the state as follows:

$$\phi(u, v, t)(V) = \begin{cases} \text{Unlock}(u, t) \\ P_{\text{user}}(u) \leftarrow v \\ W_{\text{user}}(u) \leftarrow L_f(u) \\ T_{\text{user}}(u) \leftarrow t \\ \bar{p} \leftarrow \bar{p}' \\ \tilde{p} \leftarrow v \\ Q \leftarrow Q' \\ t_{\text{sub}} \leftarrow t \\ t_{\text{last}} \leftarrow t \\ \mathcal{P} \leftarrow \mathsf{append}\big(\mathcal{P}, (t, \bar{p}', v)\big) \\ B_{\text{user}}(u, \tau_r) \leftarrow B_{\text{user}}(u, \tau_r) + n \\ B_{\text{oracle}}(\tau_r) \leftarrow B_{\text{oracle}}(\tau_r) - n. \end{cases} \tag{13}$$

3.3.4 Value Reading

Anyone who is not blacklisted may read oracle values (`readValue` and `readLatestValue`). This refreshes the liveness timestamp. The state update is:

$$\rho(u, t)(V) : \Big\{ t_{\text{last}} \leftarrow t. \tag{14}$$

3.3.5 Vote to Blacklist

The blacklist vote operation $\beta : U \times U \times \mathbb{R}_{\geq 0} \to V \to V$ requires $\mathcal{B}(u) \neq 1$, $L_f(u) > 0$, and that u has not voted on x before. It updates the state as follows:

$$\beta(u, x, t)(V) = \begin{cases} \text{Unlock}(u, t) \\ V_{\text{black}}(x) \leftarrow V_{\text{black}}(x) + L_f(u) \\ \mathcal{W}_{\text{black}}(x, u) \leftarrow L_f(u) \\ M_{\text{black}}(u) \leftarrow M_{\text{black}}(u) \cup \{x\} \\ T_{\text{op}}(u) \leftarrow t \\ t_{\text{last}} \leftarrow t \\ \text{Recompute}(x). \end{cases} \tag{15}$$

3.3.6 Vote to Whitelist

Analogously, the whitelist vote $\omega : U \times U \times \mathbb{R}_{\geq 0} \to V \to V$ enforces the same preconditions and updates the state as follows:

$$\omega(u, x, t)(V) = \begin{cases} \text{Unlock}(u, t) \\ V_{\text{white}}(x) \leftarrow V_{\text{white}}(x) + L_f(u) \\ \mathcal{W}_{\text{white}}(x, u) \leftarrow L_f(u) \\ M_{\text{white}}(u) \leftarrow M_{\text{white}}(u) \cup \{x\} \\ T_{\text{op}}(u) \leftarrow t \\ t_{\text{last}} \leftarrow t \\ \text{Recompute}(x). \end{cases} \tag{16}$$

3.3.7 Weight Synchronisation

This operation (`updateUserVoteWeights`) updates the state as follows:

$$\psi(u, t)(V) : \begin{cases} \text{Unlock}(u, t) \\ \text{Reweight}\big(u, L_f(u)\big) \\ t_{\text{last}} \leftarrow t. \end{cases} \tag{17}$$

3.3.8 Reward Token Funding

Funding operations advance the liveness timestamp and increase the reward pool balance:

$$\varphi(u, a, t)(V) : \begin{cases} B_{\text{user}}(u, \tau_r) \leftarrow B_{\text{user}}(u, \tau_r) - a \\ B_{\text{oracle}}(\tau_r) \leftarrow B_{\text{oracle}}(\tau_r) + a \\ t_{\text{last}} \leftarrow t. \end{cases} \tag{18}$$

Here, a is the amount of reward tokens transferred to the contract. This action is unrestricted and prepares future submissions to receive rewards.

4 Theorems

Orb oracles enjoy many important properties. The following theorems show some of them. Theorem 3 shows that the constant-time update rule computes the same result as the Ideal Decayed Weighted Mean function (Equation 4). Since a naive computation of this function would require a sum over all previously submitted values, whenever a new value is submitted, this theorem shows that Orb oracles can efficiently compute a new decayed weighted mean value of an unbounded number of submitted values in constant time.

▶ **Theorem 3** (Equivalence of the Ideal Decayed Weight Mean Function and the Constant-Time Update Rule)**.** *The Ideal Decayed Weighted Mean function $P(t)$ and the corresponding constant-time update rule $\bar{p}_k(t)$—that is, $\bar{p}$ after the k-th update—are equal.*

Proof. The equivalence is proven by induction on the number of updates k and a user set U.

- Base case ($k = 1$ and $|U| = 1$, so that $t(k) = t(1) = 0$): Initially, there is only one user and no previous submissions, so $Q = \bar{p} = p_u = w_u = 0$. Thus, $Q^{'} = w$ and $\bar{p}_1(0) = \frac{v \cdot w}{w} = v = P(0)$.

- Inductive case (Assume by the induction hypothesis that $P(t) = \bar{p}_k(t)$ and $|U| = n$): At the $k+1$ update, there is a user $u \in U$ with unlocked stake that submits a new value v at a time $t(k+1) = t > t_{\text{sub}}$. So,

$$
\begin{aligned}
Q^{'} &= [Q_k - w_u \cdot \delta(t_{\text{sub}} - t_u)] \cdot \delta(t - t_{\text{sub}}) + w && (19)\\
&= \left[\sum_{x \in U} W_{\text{user}}(x) \cdot \delta(t_{\text{sub}} - t_x) - w_u \cdot \delta(t_{\text{sub}} - t_u)\right] \cdot \delta(t - t_{\text{sub}}) + w && (20)\\
&= \sum_{x \neq u} W_{\text{user}}(x) \cdot \delta(t_{\text{sub}} - t_x) \cdot \delta(t - t_{\text{sub}}) + w && (21)\\
&= \sum_{x \neq u} W_{\text{user}}(x) \cdot \delta(t - t_x) + w && (22)\\
&= \sum_{x \in U} W_{\text{user}}(x) \cdot \delta(t - t_x) && (23)\\
&= \sum_{x \in U} W_{\text{decayed}}(x, t) && (24)
\end{aligned}
$$

and,

$$
\begin{aligned}
\bar{p}_{k+1}(t) &= \frac{(\bar{p}_k \cdot Q_k - p_u \cdot w_u \cdot \delta(t_{\text{sub}} - t_u)) \cdot \delta(t - t_{\text{sub}}) + v \cdot w}{Q^{'}} && (25)\\
&= \frac{(\sum_{x \in U} p_x \cdot w_x \cdot \delta(t_{\text{sub}} - t_x) - p_u \cdot w_u \cdot \delta(t_{\text{sub}} - t_u)) \cdot \delta(t - t_{\text{sub}}) + v \cdot w}{Q^{'}} && (26)\\
&= \frac{\sum_{x \neq u} p_x \cdot w_x \cdot \delta(t_{\text{sub}} - t_x) \cdot \delta(t - t_{\text{sub}}) + v \cdot w}{Q^{'}} && (27)\\
&= \frac{\sum_{x \neq u} p_x \cdot w_x \cdot \delta(t - t_x) + v \cdot w}{Q^{'}} && (28)\\
&= \frac{\sum_{x \in U} p_x \cdot w_x \cdot \delta(t - t_x)}{Q^{'}} && (29)\\
&= \frac{\sum_{x \in U} P_{\text{user}}(x) \cdot W_{\text{decayed}}(x, t)}{\sum_{x \in U} W_{\text{decayed}}(x, t)} && (30)\\
&= P(t) && (31)
\end{aligned}
$$

◀

The next theorems concern robustness against delays, which are defined as follows.

▶ **Definition 4** (Oracle Operator Delay). *Define an oracle operator's delay as:*

$$\Lambda_x(t) = \frac{W_{\text{decayed}}(x, t) \cdot (t - T_{\text{user}}(x))}{\sum_{x \in U} W_{\text{decayed}}(x, t)} \tag{32}$$

▶ **Definition 5** (Oracle Delay). *From the definition of the oracle operator delay, define the oracle's delay as:*

$$\Lambda(t) = \sum_{x \in U} \Lambda_x(t) \tag{33}$$

Theorem 6 shows that, if an oracle operator becomes inactive, the delay that it causes to the oracle gradually converges to zero. This is important, given that Orb oracles do not delete values submitted by inactive oracle operators, but only decay them.

▶ **Theorem 6** (Inactive Oracle Operator Delay). *If an oracle operator $u \in U$ becomes inactive, the operator delay $\Lambda_u(t)$ caused by this inactivity converges to 0.*

Proof. By definition, the oracle delay for an oracle operator $u \in U$ is:

$$\Lambda_u(t) = \frac{W_{\text{decayed}}(u,t) \cdot (t - T_{\text{user}}(u))}{\sum_{x \in U} W_{\text{decayed}}(x,t)} \tag{34}$$

$$= \frac{w_u \cdot \delta(t - t_u) \cdot (t - t_u)}{\sum_{x \in U} w_x \cdot \delta(t - t_x)} \tag{35}$$

$$= \frac{w_u \cdot 2^{\frac{-(t-t_u)}{h}} \cdot (t - t_u)}{\sum_{x \in U \setminus \{u\}} w_x \cdot 2^{\frac{-(t-t_x)}{h}} + w_u \cdot 2^{\frac{-(t-t_u)}{h}}} \tag{36}$$

$$= \frac{w_u \cdot (t - t_u)}{\sum_{x \in U \setminus \{u\}} w_x \cdot 2^{\frac{(t-t_u)}{h}} \cdot 2^{\frac{-(t-t_x)}{h}} + w_u} \tag{37}$$

$$= \frac{w_u \cdot (t - t_u)}{\sum_{x \in U \setminus \{u\}} w_x \cdot 2^{\frac{(t_x-t_u)}{h}} + w_u}. \tag{38}$$

So, if the oracle operator u is inactive, then w_u and t_u remain constant as $t \to \infty$. As long as a subset $Y \subseteq U \setminus \{u\}$ remains active, then for $y \in Y$, $t_y \to \infty$ as $t \to \infty$, so that $t_y - t_u \to \infty$. Hence, for each active $y \in Y$, the term $w_y \cdot 2^{\frac{t_y - t_u}{h}}$ in the denominator tends to infinity, while the numerator $w_u \cdot (t - t_u)$ grows only linearly in t. Thus,

$$\lim_{t \to \infty} \Lambda_u(t) = 0$$ ◀

Theorem 7 shows that the oracle delay is never greater than the difference between the current time and the submission time of the latest value submitted by the most delayed oracle operator. This theorem is true thanks to the fact that the constant-time update rule carefully removes the influence of non-latest values in the computed weighted average.

▶ **Theorem 7** (Optimally Small Oracle Delay). *The oracle delay $\Lambda(t)$ is bounded above by $(t - t^*)$, where $t^* = \min_{x \in U} t_x$.*

Proof. Expanding the definitions of oracle delay and oracle operator's delay, we have that:

$$\Lambda(t) = \sum_{x \in U} \Lambda_x(t) \tag{39}$$

$$= \sum_{x \in U} \frac{W_{\text{decayed}}(x,t) \cdot (t - T_{\text{user}}(x))}{\sum_{x \in U} W_{\text{decayed}}(x,t)} \tag{40}$$

$$= \sum_{x \in U} \frac{W_{\text{decayed}}(x,t) \cdot (t - t_x)}{\sum_{x \in U} W_{\text{decayed}}(x,t)} \tag{41}$$

$$\leq \sum_{x \in U} \frac{W_{\text{decayed}}(x,t) \cdot \max_{x \in U}(t - t_x)}{\sum_{x \in U} W_{\text{decayed}}(x,t)} \tag{42}$$

$$= \sum_{x \in U} \frac{W_{\text{decayed}}(x,t) \cdot (t - \min_{x \in U} t_x)}{\sum_{x \in U} W_{\text{decayed}}(x,t)} \tag{43}$$

$$= \sum_{x \in U} \frac{W_{\text{decayed}}(x,t) \cdot (t - t^*)}{\sum_{x \in U} W_{\text{decayed}}(x,t)} \tag{44}$$

$$= \frac{\sum_{x \in U} W_{\text{decayed}}(x,t) \cdot (t - t^*)}{\sum_{x \in U} W_{\text{decayed}}(x,t)} \tag{45}$$

$$= (t - t^*) \tag{46}$$

Thus,

$$\Lambda(t) \leq (t - t^*)$$

◀

Theorem 8 shows that as long as the oracle has been funded at least once, the balance of the oracle's reward pool, will always remain positive, ensuring that there is always a positive reward to incentivize oracle operators to continue submitting values.

▶ **Theorem 8** (Sustainable Rewards). *If* $0 \leq \alpha < 1$ *and* $B_{\text{oracle}}(\tau_r) > 0$ *at a given time* t*, then* $B_{\text{oracle}}(\tau_r) > 0$ *for all times* $t' > t$*.*

Proof. By assumption, since $\alpha \in [0, 1)$ and $B_{\text{oracle}}(\tau_r) > 0$ at a given time t, then any update must by definition occur at a time $t' > t$, where the remaining oracle balance will be:

$$\begin{aligned} B_{\text{oracle}}(\tau_r) - n(t') &= B_{\text{oracle}}(\tau_r) - \alpha \cdot B_{\text{oracle}}(\tau_r) \cdot \frac{w}{Q'} \cdot \big(1 - \delta(t' - t_u)\big) && (47) \\ &= B_{\text{oracle}}(\tau_r)(1 - \alpha \cdot \frac{w}{Q'} \cdot \big(1 - \delta(t' - t_u)\big)) && (48) \\ &> B_{\text{oracle}}(\tau_r)(1 - \alpha) && (49) \\ &> 0 && (50) \end{aligned}$$

Thus, for all future updates, and therefore all future times, since the remaining balance is positive and becomes the new balance, the new balance is also positive. ◀

5 Discussion

Orb Oracles are decentralized in the sense that any oracle token holder can submit values and thus can act as an oracle operator. Thanks to its efficient constant-time weighted average value update rule, an arbitrarily large number of oracle operators can participate, with no performance impact. There are, therefore, no limits to the decentralization.

Furthermore, the exponential decay ensures that, if any oracle operator becomes inactive, the contribution of its latest value submission to the average eventually and quickly becomes negligible. Considering only the latest value submitted by each oracle operator prevents a single operator from dominating the average and bounds the oracle delay.

Orb oracles as defined here deliberately avoid outlier rejection, since the rejection of a new outlier value that was submitted in reaction to a sharp change in the source value would mean that the oracle would delay to take that sharp change into account. However, variations of the Orb oracle protocol with outlier rejection could be defined and implemented. There is a trade-off between speed and robustness against possibly, but not necessarily, "wrong' outlier values.

Locking periods for depositing tokens to become an oracle operator and for withdrawing tokens to cease to be an oracle operator deter governance attacks whereby a malicious user would briefly buy or borrow tokens just to act as an oracle operator for a short time.

Orb oracles are sustainable in the sense that oracle operators are rewarded for submitting values and thus have a source of revenue to cover their operational costs. The reward formula discourages too frequent submission and prevents draining of the reward pool by a single oracle operator. It also ensures that the reward pool, if it is ever funded at least once, never becomes empty and thus there are always incentives for oracle operators to continue submiting values.

Orb oracles are also resistant to free-riding. Oracle operators are free to set their own terms and conditions for using the oracle and, if an oracle consumer is not satisfying the terms and conditions, a weighted majority of oracle operators can blacklist this consumer. In particular, the terms and conditions may involve requirements to fund the oracle. In this way, free-riding consumers, who use the oracle but do not contribute to its funding, may be blacklisted. This governance-based free-riding resistance further contributes to oracle sustainability.

The token-based oracle operation is flexible enough to encompass a wide-range of operational arrangements. For example: a centralized first-party oracle could be set up by keeping all the tokens under the control of the first-party; a permissioned consortium of oracle operators could be set up by distributing oracle tokens to the members of the consortium and making these tokens non-transferable; a fully permissionless oracle could be set up by using an oracle token that can be minted to oracle consumers in proportion to their alignment to the oracle's correct operation.

References

1 Bruno Woltzenlogel Paleo, Luca D'Angelo, Mohammad Shaheer, and Giselle Reis. Gluon w: A cryptocurrency stabilization protocol. Cryptology ePrint Archive, Paper 2025/1372, 2025. URL: https://eprint.iacr.org/2025/1372.

2 Bruno Woltzenlogel Paleo. *Stablecoin*, pages 1–5. Springer Berlin Heidelberg, Berlin, Heidelberg, 2019. doi:10.1007/978-3-642-27739-9_1671-1.

3 Joachim Zahnentferner, Dmytro Kaidalov, Jean-Frédéric Etienne, and Javier Díaz. Djed: A formally verified crypto-backed pegged algorithmic stablecoin. Cryptology ePrint Archive, Report 2021/1069, 2021. https://eprint.iacr.org/2021/1069.

4 Joachim Zahnentferner, Dmytro Kaidalov, Jean-Frédéric Etienne, and Javier Díaz. Djed: A formally verified crypto-backed pegged autonomous stablecoin. In *IEEE International Conference on Blockchain and Cryptocurrency*, 2023. https://icbc2023.ieee-icbc.org/program/icbc-2023-final-program.

A License

B Official Implementations and Deployments of Orb

https://github.com/StabilityNexus

C Official Orb Channels

- https://x.com/StabilityNexus

[5] https://creativecommons.org/licenses/by-nc-nd/4.0/

12 Orb: Decentralized and Sustainable Oracles

- https://discord.gg/7jS9qJNjJv