FTSOv2: more data feeds and faster updates to the FTSO

Flare Network

Abstract

The FTSO (Flare Time Series Oracle) is a native protocol of the Flare Network, allowing data providers on the network to periodically determine consensus-driven time series values for a collection of data feeds. Typically, these data feeds represent prices of various assets, so that the FTSO provides access to decentralized price estimates for various smart contracts operating on the network and beyond. This document presents the design features of an improved version of the FTSO, referred to as FTSOv2, supporting higher frequency updates and a larger range of data feeds. These improvements are driven by optimizations to the core Commit-Reveal protocol of the FTSO, as well as a novel feature of the FTSOv2 that enables updates in every block. Thus, the FTSOv2 supports two types of feeds: *anchor* feeds updating every 90 seconds, and *block-latency* feeds determined by updates in intermediary blocks. Both types can scale to 1000 data feeds. Additionally, it consumes less than 10% of the available throughput of the Flare Network.

1 Introduction

The Flare Time Series Oracle (FTSO) is a system for reaching periodic consensus on specific time series on the Flare blockchain. It is powered by 100 data providers, who submit their individual data estimates. Their submissions are weighted by token delegation and the consensus values are calculated using a weighted median algorithm. In this way, the FTSO provides consensus-supported on-chain access to various data feeds; in its present form it updates 18 feeds every 3 minutes, which are stored on-chain and directly accessible to DApps. The current FTSO [6], termed FTSOv1, is implemented mainly with smart contracts. The FTSOv2 is an improved design, offering two key features: scaling of data feeds, whereby hundreds of feeds are available, and block-latency feeds, where value estimates are published more frequently. This is achieved by moving almost all computation off-chain, whilst retaining the whitelist and decentralisation of the previous version.

1.1 Requirements

Decentralization. As with the FTSOv1, this new version is required to be both secure and decentralized. This is achieved by incentivizing data providers to submit accurate data estimates, as in the FTSOv1. At the same time, no single provider should gain too much control over the data feeds. Through a mix of capping and weight management, the FTSOv2 rewarding process handles the balance between incentivization and decentralization.

Gas Fees. The storage of value estimates sent by about 100 data providers in the FTSOv1 bears significant cost that currently fills around 30% of available sustainable gas bandwidth with only 18 data feeds. Since the cost of data storage is very expensive, a scalable redesign of the system is required to minimise gas usage. Also, when possible, calculations and data storage should be outsourced off-chain to data providers, with agreement on the calculation results taking place on-chain; this reduces gas consumption of the computations themselves.

To this end, the FTSOv2 substantially reduces the amount of information per-feed required to be stored on chain and allows the providers to perform necessary computations such as median values locally, with only the necessary verifiability information uploaded to the chain.

Latency. By applying the above optimizations, the FTSOv2 allows each of the 100 providers to provide estimates for 1000 data feeds every 90 seconds. Whilst already a substantial improvement, the FTSOv2 supports an additional feature known as *block-latency feeds* that allows for valuations at an even higher frequency. The new feature publishes a value delta every block, so that the per-block value, is determined and published by tracking these deltas.

1.2 Document Structure

Section 2 introduces the core design features of the FTSOv2 that enable superior scaling to the v1 iteration. Section 3 explains the novel features that facilitate per-block value estimates. Additional technical details required to understand the block-latency feeds are left to the Appendix.

2 Scaling Feature: Anchor Feeds

2.1 Phases

The FTSOv2 protocol takes place in a sequence of *voting rounds*, with each iteration lasting one round, so that each data feed is updated once per round. This produces a sequence of values known as the *anchor feed*. Each voting round begins at the start of a new *voting epoch*, determining the value of each anchor feed for that 90 second epoch. The value of each feed is determined by aggregating value submissions from each participating data provider into a weighted median value. Each round takes place across two *voting epochs*, with rounds and epochs identified by *round ids* and *epoch ids* respectively, with enumeration aligned so that round i begins at the same time as epoch i. That is, in each voting epoch a new voting round begins, however, the duration of the voting round is longer than the epoch, so that more than one voting round may be proceeding at a time.

More specifically, each round of the FTSOv2 protocol proceeds in four phases: the *commit* phase, in which the data providers commit to their data vectors for the round, the *reveal* phase, where the data providers reveal the values underlying their respective commits, the *sign* phase, when providers collate data estimates to produce the median data values, and a *finalization* phase, ending the round when a provider collects sufficiently many signatures of the median values for the data estimates to be finalized. The phases of the FTSOv2 protocol, and the distinctions between voting rounds and epochs, are depicted in Figure 1.

2.1.1 Commit Phase

The commit phase begins the voting round and lasts the entire 90 second duration of the voting epoch i. In this phase, each data provider computes their individual estimate for each data feed and encodes it into a 4-byte vector using offset binary encoding, then publishes a hash commitment to the combination *data* of these vectors. The commitment is calculated as

$$\text{Commit Hash} = \text{Hash}(\mathit{address}, i, \mathit{rand}, \mathit{data})$$

where *rand* is a locally generated random number and *address* the data provider's address. This random number serves two purposes: it blinds the commit hash of the user from a search attack, and is used later (once revealed) to contribute to on-chain randomness. Each

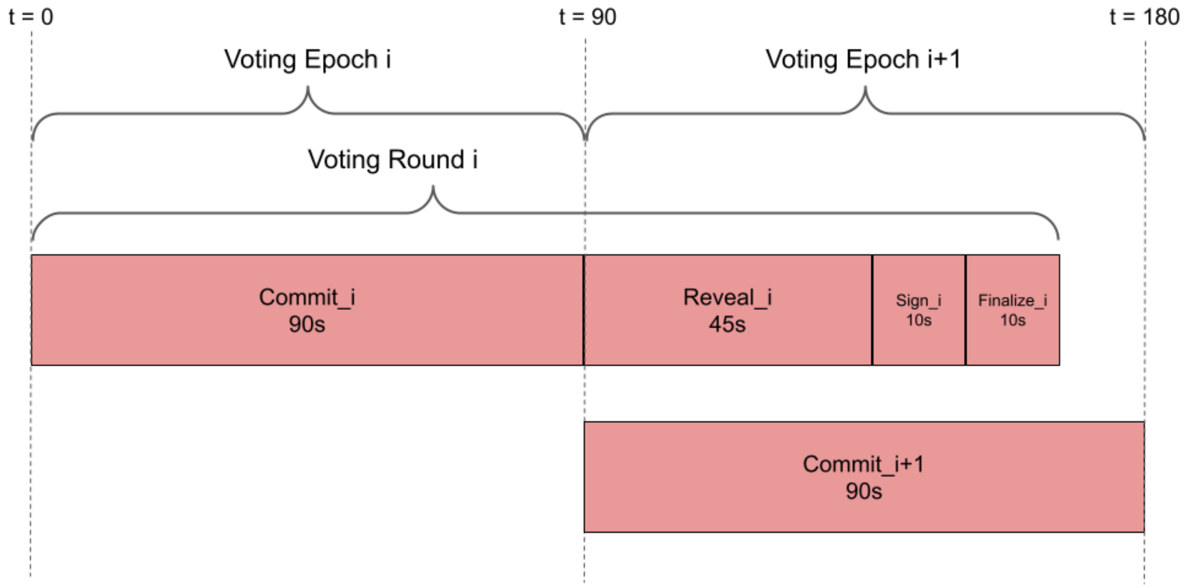

Figure 1: The phases of the FTSOv2 protocol.

provider's Commit Hash is uploaded to the chain in a commit transaction, which is valid as long as its block timestamp correctly matches up with the voting epoch i.

2.1.2 Reveal Phase

Beginning immediately after the commit phase, the reveal phase lasts 45 seconds and requires each provider to reveal their individual estimates committed to in the previous phase. To do so, they each complete a *reveal transaction*, revealing all inputs to their hash commitment. That is, each provider reveals its estimates $data$ and its random number $rand$. A reveal transaction is valid as long as the hash of the revealed data matches up with the hash commitment of the provider; validity of the reveal transaction can be confirmed off-chain, and also requires that the block timestamp of the transaction lies within the Reveal Phase. Note that providers do not need to explicitly publish their respective addresses as these are publicly available.

2.1.3 Sign Phase

The sign phase begins as soon as the reveal phase finishes, and has an initial duration of 10 seconds. During this phase, data providers collate submissions from the commit and reveal phases, filter out invalid submissions, and compute the weighted median values and rewards (see Sections 2.2 and 2.3) for each anchor feed. Each provider then packages together the valid submissions and results of their computation into a Merkle tree, and publishes a *sign transaction* consisting of the Merkle root and a signature of the root. Once again, note that the relevant computations are performed off-chain, with only the results themselves being committed on-chain.

2.1.4 Finalization Phase

The finalization phase begins at the end of the signing phase, and has an initial duration of 10 seconds. For this phase, a random selection of providers are chosen to participate, selected sequentially and independently with probability equivalent to their relative weight until more than 5% of the total weight of providers has been selected. Thus, the number of chosen providers varies, with enough providers chosen each round so that at least 5% of the total weight of providers are able to finalize. The results of this sampling are available in advance, so that providers know whether they have been selected for finalization before the phase begins.

Using the available signatures from the signing phase, each of the selected data providers can end the round by collating enough signatures for the same Merkle root and submitting them to the *relay contract*, which verifies that the signatures are valid and that a sufficient voting weight of signatures (at least 50%) have been submitted. Assuming these checks pass, the Merkle root is published on the voting contract for the round, and thus the data feeds become available for other smart contracts. If none of the selected providers have completed the finalization phase after 10 seconds, it is opened to all data providers and concluded once any provider has submitted a finalization.

Overlapping Phases. In practice, there is some overlap between the signing and finalization phases: the finalization process may be completed as soon as enough valid signatures are available for the voting round. In this case, signatures deposited during the signing phase but after finalization is completed are still rewarded as normal. Conversely, assuming finalization is not completed early, signatures deposited after the signing phase ends but before finalization is completed are considered valid and rewarded as usual.

2.2 Weighting

The FTSOv2 protocol is a stake-based protocol, in the sense that the contribution to data feed values and other procedures by each provider is not equal. Rather, each provider's contribution to the protocol is proportional to its weight. Weight can be acquired in two ways: firstly, provider stake, the proportion of FLR tokens owned by the provider itself. Secondly, providers gain weight according to FLR tokens delegated to their address by other entities, who in turn receive a share of provider rewards for their delegation. Different phases of the FTSO process use different definitions of weight, as described below. Note that providers charge a small fee to their delegators, which manifests as them retaining a proportion of the reward allocated to the delegations (see Section 2.3). The size of this fee is set by the providers themselves.

FTSO Calculation Weight. The median calculation for the FTSO uses only the delegation weight of the provider, the amount of wrapped Flare (WFLR) tokens delegated to the provider for participating in the FTSO protocol. If staking weight were taken into account, stake within the system held by providers inactive in the protocol would dilute the impact of WFLR delegation to active providers, potentially compromising the accuracy of the FTSO outputs.

Thus, for provider i with weight W_{D_i} of delegated tokens, the normalized FTSO calculation weight $W_{i,C}$ of the provider is equal to

$$W_{i,C} = \frac{W_{D_i}}{\sum_i W_{D_i}}.$$

This is the weight used for both the FTSO computation itself as well as the rewards on offer for successful FTSO participation.

Signing Weight. The signing and finalization phases use the combination of both staked weight and delegated weight. Thus, the *normalized* signing weight $W_{i,S}$ of the ith provider can be calculated as

$$W_{i,S} = \frac{(W_{S_i} + W_{D_i})}{W_{tot}}$$

where W_{S_i} denotes the stake and W_{D_i} the delegated stake of provider i, with W_{tot} the total weight of the system, $W_{tot} = \sum_i (W_{S_i} + W_{D_i})$. In practice, all normalization is implicit; weights are used directly and parsed as percentages.

2.2.1 Capping

In order to ensure that the system remains sufficiently decentralized, caps are enforced on the maximum weight that an individual provider can have in any given phase of the overall protocol. As the distinct phases of the FTSOv2 protocol have slightly different security requirements, the capping measures vary across the phases, as discussed below.

FTSO Data Feed Capping. The goal of capping the individual provider contribution to the FTSO data feeds is to ensure that no individual entity has too much of an input to the median computation, which would damage the core property of decentralization. However, too aggressive a cap would distribute the feed values across too many low weight parties, who have little investment in the system, which in turn may enable Sybil attacks or damage the accuracy of the estimates.

The chosen cap is 2.5% of the weight in the system. If the weight of any providers in a round exceeds 2.5% of the total, then that provider's weight is considered to be exactly 2.5% in that round. Since normalization is implicit, if this cap is active (e.g. some providers have too much weight) then the removed weight W_r is essentially redistributed across all providers proportionally to their existing, post-capped stake, e.g. the weight $W_{i,C}$ of provider i is updated to:

$$W_{i,C}{}^{*} = W_{i,C} \cdot \frac{100 + W_r}{100}.$$

so that in practice the capped providers have weight a little over the initial 2.5% cap.

Signing Weight. For signing weight, a more complicated two-step process is applied. As in the previous stage, the goal is to trade off decentralization against the possibility of disruption from many low-weight addresses.

First, the same capping process as described in the previous paragraph is applied to the signing weight, so that each provider has a capped weight $W_{i,S}{}^{*}$. Then, a process known as *diversity weighting* is applied, rescaling each provider's weight proportionally to its value to the power of 3/4. The purpose of diversity weighing is to further increase the decentralization by reducing the weight of larger providers. This relative increase in the power of low-weight providers is intended to decrease the number of low-weight providers required to end a signing phase, so that high-weight providers cannot halt progress by withholding signatures. Functionally, the signing weight $W_{i,\text{sign}}$ of provider i is set to:

$$W_{i,\text{sign}} = \frac{W_{i,S}{}^{*3/4}}{W_{s,tot}}$$

where $W_{s,tot}$ denotes the total post-weighted signing weight, $W_{s,tot} = \sum_i (W_{i,S}{}^{*})^{3/4}$.

2.3 Rewarding

Rewards for participating in the anchor feeds of the FTSOv2 may come from one of two sources: Flare's inflation and community funded pools. The aim of these rewards is to incentivize the FTSO outputs to be both accurate and prompt. To this end, rewards are split across accurate individual estimations, correct signing, and prompt finalizing. The first of these rewards handles accuracy, with the other two primarily focused on efficiency. Additionally, providers must be punished for deviating from the protocol, as well as rewarded for correct participation.

Selecting a Rewarded Feed. Each round, rewards are determined according to performance in a single randomly selected data feed, rather than an aggregation of performance across all anchor feeds. The choice of data feed is sampled uniformly at random amongst existing data feeds, and crucially is not known in advance. That is, the random seed determining which feed is to be rewarded in a given round is generated as part of the process generating randomness (see Section 2.4) in that round. This prevents providers from only focusing on the feed that is to be rewarded in a round; in order to maximize expected rewards, providers should submit an accurate estimate for each feed.

This applies both to assigning FTSO accuracy rewards and to determining eligibility for signing and finalization rewards: only one, randomly selected, feed is used. Since the rewarded feed is determined uniformly, providers expected returns are unchanged regardless of potentially varying accuracy across supported fees. However, there is a computational benefit, as the amount of work required to determine rewarding is lowered to only that needed for one feed per round.

2.3.1 FTSO Accuracy Rewards

The majority (around 80%) of available rewards are allocated for performance in the median computation of the FTSO round; these rewards are denoted by R_{FTSO}. They are assigned to incentivise submitting accurate value estimates close to the median value. FTSO accuracy rewards are allocated according to two criteria: rewards for submitting a value within the weighted interquartile range (called the primary reward band) of submitted values, and rewards for submitting a value within a percentage interval around the weighted median value (referred to as the secondary reward band), whose width is a parameter determined by governanace. In the case where a submission lies exactly on the border of the interquartile range (IQR), its eligibility, or lack thereof, for primary band rewards is determined randomly. Providers can be eligible for both rewards for the same submission; in practice, submissions close to the median will often lie within the IQR as well.

More precisely, let R_{IQR} denote the rewards available for submissions within the primary band and R_{PCT} for those in the secondary, satisfying $R_{FTSO} = R_{IQR} + R_{PCT}$. Let Σ_{IQR} and Σ_{PCT} denote the total (post capping) weight of providers whose submissions lie in the primary and secondary band respectively. Then, an individual provider i with weight ${W_{i,C}}^*$ whose submission lies within the primary band gets reward ${R_{IQR}}^i$ defined as

$$ {R_{IQR}}^i = \frac{{W_{i,C}}^*}{\Sigma_{IQR}} \cdot R_{IQR} $$

and similarly reward ${R_{PCT}}^i$ for submissions within the secondary

$$ {R_{PCT}}^i = \frac{{W_{i,C}}^*}{\Sigma_{PCT}} \cdot R_{PCT}, $$

with these rewards split amongst the provider and its delegators proportionally to their contribution to the provider's weight. In the very rare case that the secondary band is empty, which is a theoretical possibility, secondary band rewards for the round are burnt.

2.3.2 Signing Rewards

Signing rewards, R_{sign}, make up around 10% of the rewards for the round, and are allocated according to the weight of providers who submit valid signatures for the correct Merkle root in the sign phase or before finalization. These rewards are provided to encourage prompt and correct participation in the signing phase. However, in order to be eligible for signing rewards, a provider must have received accuracy rewards in the given round for the selected feed; this is to prevent providers participating only to receive signing rewards and not submitting diligent feed values.

Let Σ_{sign} denote the total weight of providers who correctly signed the agreed upon Merkle root in the sign phase or before finalization. Then, an eligible provider with weight $W_{i,sign}$ who delivered a correct signature receives the reward ${R_{sign}}^i$ corresponding to their contribution to the total weight,

$${R_{sign}}^i = \frac{W_{i,sign}}{\Sigma_{sign}} \cdot R_{sign}.$$

2.3.3 Finalization Rewards

The finalization rewards R_{fin} make up around 10% of the total rewards, and are distributed among the selected providers equally. That is, in a round where the number of providers selected to finalize is N_{fin}, each of these providers that submits a valid finalization in the allotted time period receives the same finalization reward ${R_{fin}}^i$ equal to:

$${R_{fin}}^i = \frac{R_{fin}}{N_{fin}}.$$

If none of the selected providers submit a valid batch of signatures of a correct Merkle root to the relay contract in the allotted time, then all rewards are instead allocated to the first other provider to do so. These rewards are provided to encourage prompt finalization of the FTSO data feed values.

As with signing rewards, providers are only eligible to receive finalization rewards if they have also received an accuracy reward in the same round. Note that this does not effect the amount of rewards assigned to each eligible provider: if N_{fin} providers are initially selected to finalize, each of those who received accuracy rewards and successfully finalizes receives a reward $\frac{R_{fin}}{N_{fin}}$ regardless of how many of those providers were both selected to finalize and received the necessary accuracy rewards to be eligible for finalization rewards. Corresponding rewards that would have been assigned to selected providers who did not first receive accuracy rewards are burnt.

2.4 Additional Features

Gas Consumption. Each byte of published data used in the protocol costs 16 units of gas. The 3 transactions submitted by a provider in the protocol require:

- Commit: 32 bytes consisting of a single hash.
- Reveal: 4032 bytes, consisting of a 4 byte encoding per feed (with 1000 feeds) and one 32-byte random number.

- Sign: 65 bytes, consisting of a 33 byte compressed ECDSA signature and a 32 byte Merkle root.

Including the flat 21000 gas fee per transaction (incurred once in each of the three phases), the cost per-provider is 129064 units of gas. Since there are 100 data providers, the total cost for a round is around 12.9M gas per voting round. This totals around 8.4% of the sustainable gas throughput of the chain itself. Finalizing and publishing the Merkle root (and corresponding values) on-chain costs another 0.5% of available gas, keeping the whole process sustainable at around 9%.

In practice, the gas consumption of the FTSO is implicitly slightly higher, as some of the required computation piggybacks off of that done by the Flare System Protocol, the protocol underlying certain functionalities crucial to the Flare network. Parts of the system protocol that are necessary for the FTSO include computing and paying out rewards, and computing the weights of the providers. The costs computed in this section do not take into account costs of the Flare system protocol, as they are a necessary part of the Flare network that are incurred every 3.5 days. Although they consume a lot of gas, system protocol computation results are used across the Flare network and not just for the FTSO, so the concrete costs are not considered towards FTSO throughput requirements.

Randomness. The Flare network requires access to on-chain randomness for a variety of cryptographic features, including selecting random providers for the finalization phase and facilitating sortition for the block-latency feeds introduced later. This is supported by the FTSOv2, with a new random number generated each epoch. Specifically, the random numbers revealed by each party in the reveal phase are combined into an aggregate random number for the epoch. To do so, each of the provider-generated random numbers $rand_i$ are added together to make a combined random number

$$rand = \sum_i rand_i \mod N$$

where $N = 2^n$ denotes the maximum possible size of the individual n-bit random numbers. As long as all individual randomness contributions are added, and at least one $rand_i$ was random, the resulting output $rand$ is a random number. In order to track whether or not any random contributions have been omitted in an attempt to degrade the quality of a random number (for example by a provider failing to complete the reveal phase), the Merkle root contains a Boolean value storing this information. In this way, whether or not a provider may have deviated from the protocol to try to manipulate the randomness is stored along with the random number.

Penalization The discussion of the commit and reveal phases in Section 2.1 assumed that each provider correctly reveals the data underlying their commit. That is, proper functioning of the FTSO process requires that the data revealed in the reveal phase by each provider correctly hashes to their commitment published in the commit phase. However, this may not always be the case; either for malicious reasons, such as a provider backing out of their commitment, or just due to an honest error. Regardless of the root cause, this is disincentivized by a slashing a chunk of the rewards earned by the offending provider.

Additionally, it was assumed in the signing and finalization phases that only one root receives enough signatures to be finalized, implicitly requiring that each signer does not sign multiple messages. For example, an opportunistic provider may attempt to gain signing rewards without expending proper effort: whenever another provider signs a message, simply sign the same message, and collect the rewards for whichever signature they gave that was correct, and ignore the failed ones. To prevent this, a penalization is applied for each signature beyond the first given by a provider in a round.

Each mismatched reveal or excess signature is punished in the same way: by burning a lump sum of provider rewards. The size of the sum is determined by a combination of a parameter R_{pen}, the weight of the provider, and the total available accuracy rewards for the round. More formally, a provider with (normalized) calculation weight $W_{i,C}$ who requires penalization in a voting round with accuracy rewards R_{FTSO} is penalized by subtracting an amount

$$R_{pen}{}^{i} = R_{pen} \cdot (W_{i,C} \cdot R_{FTSO})$$

of their rewards for the round for each penalization accrued.

Burning. As well as rewarding the publication of signatures during the signing phase, the system punishes providers for failing to participate in this phase in a timely manner. If the signing process has not received enough weight of signatures before a certain number of blocks, DB_{sign}, has passed in the signing phase, then the burning process begins. For those providers who have not yet published a correct signature, a proportion of delegation fees are burnt in each subsequent block. Since provider fees are only obtained by providers who are rewarded for accurate data submissions in a given round, this burning procedure only affects successful providers who delay providing a signature. The proportion of fees burnt is quadratic in the number of blocks passed since DB_{sign}, until a maximum block count DB_{max} is reached, at which point all fees have been burnt. Formally, the proportion P_{burn} of burnt fees by a provider publishing a signature in block $DB_{pub} > DB_{sign}$ is defined $P_{burn} = \min(\text{Burn}, 1)$, where

$$\text{Burn} = \left(\frac{DB_{pub} - DB_{sign}}{DB_{max} - DB_{sign}}\right)^2.$$

The parameters of the burning system DB_{sign} and DB_{max} are set by governance, balancing the need for promptness with tolerance for provider outages or other latency issues.

3 Block-Latency Feeds

The FTSOv2 anchor feeds enable 100 data providers to submit data estimates in a commit and reveal scheme that enables secure feed values to be determined every 90s, which is now further supported by new *block-latency* feeds. These feeds enable data updates to be computed every block by publishing frequent incremental small value changes (updates) over time, rather than computing the values from scratch.

This new process relies on selecting random samples of data providers to submit incremental updates from the last stream value. Each chosen provider submits an update as a *unit delta*, stating whether the value should go up, down, or remain constant. This is then converted to a *numeric delta*, representing the percentage change of the value of a feed caused by a single unit delta. The size of the random sample, as well as the size of the numeric delta are two system parameters which enable the system to reflect desired data volatility whilst retaining appropriate levels of security.

3.1 Overview

Update Transactions. Updates to the data stream are given incrementally in a cadence of one or several for each block, with increments provided by data providers who are chosen by random sampling. Each data provider submits a transaction that proves their eligibility, determined through *cryptographic sortition* (introduced by Algorand [3]), and gives a unit value change, termed a unit delta, for each data feed in the FTSO.

Applied to the FTSO, cryptographic sortition is a process for selecting random providers to take part in rounds of the update protocol. Each block corresponds to a round of sortition, and the ith provider is selected to participate or not with probability proportional to its (signing) weight $W_{i,S}$. This selection is independent of that of the other providers, so that sortition does not pick a fixed number of users per round. Rather each user is in or out each round with a fixed probability, and does not know the status of other users until they reveal it themselves. Providers are able to cryptographically demonstrate that they have been selected to participate, and cannot cheat the process. More technical details can be found in Appendix A.

The unit delta for each data feed, taken from the transaction, is converted to a numerical delta that represents a concrete value difference. These differences yield a data stream for each feed, which is stored on-chain without history.

Incentives. Incentives to push updates to the block-latency feeds are offered for two purposes: to improve the accuracy of the stream values relative to the anchor feed, and to drive volatility through increased granularity and greater value variation. Incentives are offered to reward activity and maintain the security of the system. One type of incentive is for accuracy relative to the FTSO values, in which the value of the block-latency feed at the time of FTSO publication is compared to the regular FTSO reward bands. This links the stream values to the anchor feed values, and prevents a competing data valuation from forming, as providers are rewarded for keeping the feed close to the *true* value represented by the FTSO data.

A second type of incentive is for volatility, the speed at which the feed value fluctuates. For this incentive, third parties offer monetary incentives that increase either the number of eligible data providers chosen by sortition, the size of the numeric deltas, or both, as well as encouraging the participation of these data providers. Increasing the number of providers setting deltas or the size of the deltas themselves naturally increases the speed at which the value can change. This pool is simply distributed uniformly for participation in order to prevent manipulation, without regard for the actual behavior of the data stream.

3.2 Choosing Providers for Feed Updates

In each block, eligible providers have the opportunity to submit an update transaction. A transaction contains the data for an incremental update to each data feed (including updates of 0 for feeds that a provider does not cover), together with metadata proving the provider's eligibility to submit such a transaction in this block. Specifically, an update transaction is a contract call with the following data in Solidity syntax:

```
struct FastUpdates {
    uint sortitionBlock;
    SortitionCredential sortitionCredential;
    Deltas deltas;
}
```

The custom types `SortitionCredential` and `Deltas` are discussed in Appendix A.3 and Section 3.3, respectively.

Transaction Submission. Each block corresponds to a round of sampling providers by sortition. As soon as the block appears, each provider has the necessary information to deterministically compute their credential for this round of sortition. Those whose credentials are acceptable are eligible to submit a single transaction with `FastUpdates` data, simultaneously proving their eligibility and declaring updates to each data feed.

The choice of eligible providers is pseudo-random and unpredictable. The amount of providers in a block is variable; it follows a binomial distribution as shown in equation (4), with mean value e that is a parameter which can be set by governance or by offering incentives.

Submission Window. It is not feasible to require that eligible providers submit their transaction in the same block as the round of sortition in which they are chosen, or even in the one afterwards. Therefore, each round of sortition provides credentials that are active for several blocks afterwards, the number of which is referred to as the *submission window* and denoted s. Thus, a round of sortition corresponding to block k entitles the eligible providers to submit a transaction in any of blocks $k, k+1, \ldots, k+s-1$. The `sortitionBlock` field of `FastUpdates` is the block number beginning the round of sortition that the transaction sender wishes to authenticate against.

3.3 Encoding of updates

An update for a single block-latency feed is a *delta* value, including 0, encoded using the standard *two's complement* format for a signed integer with a fixed number of bits. The entire set of updates is provided as a packed array of signed-integer deltas, ordered according to a predetermined standard for ordering data feeds. Deltas are only allowed to have one magnitude, in either direction, or be zero, and the three possible deltas are encoded as

$$00 \to 0 \qquad 01 \to +1 \qquad 11 \to -1 \qquad 10 \to \text{unused}$$

If larger value variations in a single block are desirable, the volatility incentive provides a mechanism to increase the value of e, the expected number of contributing providers, so as to make this possible.

Numeric Deltas. Each data feed has a configurable numeric delta increment, so that ± 1 in a unit delta increment corresponds to an actual value update by that numeric delta. For simplicity the feeds' numeric deltas are all determined by a single parameter, the *precision* p, and are dynamic: when a feed has current value P, a unit delta increment δ updates the value to ΔP, defined as:

$$\Delta P = (1+p)^{\delta} P.$$

The precision can be tuned via the volatility incentive, with a base value chosen by governance, and is represented as a fixed-point number in the interval $(0,1)$ with a fractional part of 15 bits. As a result, there is a hard minimum value of $2^{-15} = 1/32768$.

Data Streams. Updates to the block-latency feeds generate a stream of values for each feed, where the value as of block n is the value as of block $n-1$ plus the overall delta in block n, defined as the application of each numeric delta increment of that block. This value is stored on chain and is maintained at each update transaction, so that the live value can be used in smart contracts.

3.4 Economic Incentives

The FTSO offers rewards to encourage honest participation, and the same is true for updating the block-latency feeds. Additionally, the protocol has a separate incentive towards the specific goal of reflecting volatility. Providers are rewarded for their updates if the block-latency data stream is sufficiently close to the next anchor feed value. Individuals are allowed to buy temporary increases in the precision and sample size (subject to controls described below), distributed as rewards to providers of updates to the block-latency feeds during the period of increase, to encourage greater responsiveness to volatility.

Total Reward and Distribution. The rest of this section describes several sources of reward for update providers, namely, from participation (denoted R_p), from accuracy (denoted R_a), and from volatility-related offers (denoted R_v). These funds are determined at different intervals as follows:

- R_p is set at the start of each *reward epoch*, a period of several voting epochs in which reward levels are fixed; between reward epochs, reward sizes may be modified.
- R_a is calculated at the end of each voting epoch.
- R_v varies block-by-block.

Combining these rewards, it follows that during each block the total reward R_t satisfies

$$R_t = R_p/b_{re} + R_a/b_{pe} + R_v, \tag{1}$$

where b_{re} is the number of blocks in the reward epoch and b_{pe} is the number of blocks in the voting epoch. All of these components of rewards are inflationary and allocated specifically for use by the block-latency update protocol.

Proportional Distribution. Each update in a block is assigned an equal share of the total reward for the block, allocated to the provider of that update. Equivalently, the participation reward is allocated in proportion to the number of updates to the block-latency feeds made by a provider during the reward epoch, the accuracy reward in proportion to those made during the voting epoch, and the volatility reward in proportion to the number of updates in each block.

Weighted Uniform Distribution. Providers are chosen by cryptographic sortition, with probability proportional to their weight. Thus, their average number of transactions over time is an accurate proxy for their weight. Therefore, over a large period of time, uniform distribution of rewards to block-latency update transactions should perform equivalently to rewarding participating providers according to their weight.

Cost of Distribution. The amount that each block-latency update provider may claim is included in the Merkle tree that contains both FTSO values and rewards for FTSO providers, and so incurs no additional computational cost on-chain.

3.4.1 Reward for Participation

The participation reward R_p of Equation (1) is a lump sum taken from reward offers for the FTSO (including inflationary and community offers), irrespective of the content of the updates. It may be seen as a start-up fund to encourage providers to build low latency update-capable infrastructure, and once participation reaches a sufficient level may be decreased or eliminated by governance so that incentives are performance-based.

3.4.2 Reward for Accuracy

The role of the accuracy reward R_a of Equation (1) is to maintain agreement between the block-latency and anchor feeds of the FTSO. These rewards are based on the FTSO reward system that defines several *reward bands* around the median value in each epoch, which encourage providers to predict the median value closely. The proposed accuracy rewards simply adjust the incentive as required so that providers are rewarded for the block-latency feeds matching the anchor values.

The FTSO Reward Bands. Recall that the FTSO defines two reward bands for anchor feeds:

- The primary reward band, centered on the median value and precisely wide enough to contain the value predictions of 50% of the total weight of providers.
- The secondary reward band, centered on the median value and having a fixed percentage width.

FTSO providers are rewarded based on whether their individual predictions fall within these bands. The block-latency feeds offer additional rewards from these bands. These rewards, as for the FTSO, are funded from inflation and calculated off-chain, but independently of the FTSO rewards. The width of the secondary band and the proportion of total rewards allocated to each band are configurable for the block-latency feed protocol separately from the FTSO.

The Reward Bands for Block-Latency Feeds. To determine rewarding, the block-latency is considered to be a *provider* whose *commit* in voting epoch i is evaluated for the reward bands at the end of that epoch alongside the submission of the regular providers. In this role, the accuracy reward R_a is set as though the block-latency feed as a whole were a provider that had made a commit for the value of the data stream at the end of voting epoch i. This includes the random choice of data feed: accuracy rewards in a given round are determined relative to the implied commit of the block-latency feed for the data feed chosen for FTSO anchor rewards in that round. Note that this use of the stream as a provider is just for allocating rewards, and not while setting the width of the reward bands themselves. For the reward calculation, the block-latency feed is awarded a proportion of the inflationary rewards for the round, with the proportion assigned to anchor and block-latency feeds being a parameter configurable by governance.

Reward Bands as Incentives for Accuracy. The above band-based rewards, which are distributed among update providers based on the proximity of the block-latency feed's value to the FTSO median value, encourage delta updates to move the feed towards the FTSO anchor value as each voting epoch progresses. This promotes agreement between the two values, preventing them from forming independent feeds. Conversely, FTSO providers are not rewarded or penalized at all for disagreement, which encourages them to predict the true value regardless of the block-latency trend; if the disparity is sufficiently large, then no accuracy rewards are given to block-latency update providers until the data streams return to agreement. This allows the anchor and block-latency feeds to support each other as sources of truth, each decided by complementary consensus processes.

3.4.3 Incentive for Volatility

Volatility is the confluence of two conditions: a large range of changes in value and high frequency of such variations. The range of variation r quantifies the degree of volatility that may be reflected by block-latency feed updates at the most frequent possible cadence. Specifically, it is defined to be $r = pe$, where p is the precision of individual updates and e is the expected number of providers giving updates per-block, corresponding to the mean of a binomial distribution.

Individuals such as DApps or other customers of the data stream may seek to fund a particular degree of volatility by setting r, which is translated to functional changes in p and e. The pricing of this is such that the cost of increasing e is exponential in the current value of e, with the goal of making it very expensive to increase the number of update transactions per block past a level that is deemed, by governance, to be the maximum tolerable amount of throughput to be occupied by the block-latency feeds.

Active Duration of Volatility Incentives. Incentive offers (described below) are made with the transfer of a corresponding monetary value m. Each offer has a *duration of effect* T_v, a parameter controlled by governance that determines the number of blocks for which it is valid for and after which the offer expires. In each of the blocks within the duration of effect, the total reward R_v is increased by m/T_v, which according to the strategy described above is allocated uniformly to updates in that block.

Format of an Incentive Offer. An incentive offer is a transaction with an associated `value` transfer (representing m) and calldata of the form:

```
struct IncentiveOffer {
    ufixed8x8 rangeIncrease;
    ufixed8x8 rangeLimit;
}
```

which specifies a particular amount of increase in the range of variation in terms of the (as-yet unsupported in Solidity) unsigned fixed-point type with 8-bit integer part and 8-bit fractional part.

The range increase must be non-negative to prevent malicious reversion of incentives; the range decreases correspondingly at the end of the duration of effect. The required field `rangeLimit` is a limiting value that allows multiple independent offers to be made blindly without overshooting the range that any of them actually desires.

Rationale. This format for an incentive offer has the appealing feature that a party interested in representing a certain amount of volatility via the data stream may do so by directly stating that desire. By contrast, an alternative solution that would support altering both p and e would suffer from the fact that they have a more abstract meaning, including a many-to-one relationship with r, and a subtle interaction whose importance is not necessarily easy to understand.

Incentive Contribution Equation. The meaning of an incentive offer is expressed through three quantities: the total contribution c, the variation range r, and the expected sample size e; the precision p is implicitly involved through the relation $r = pe$. The contribution c is expressed as a function of r and e in the form:

$$c = Ar + \exp(e/B), \tag{2}$$

where A and B are parameters to be specified. The meaning of c is the total amount of contributions through active incentive offers that have brought r and e to their present values. A represents the cost of increasing the value of r by 1, which is configurable by governance. The setting of B determines the maximum increase in e permitted by a single incentive offer; this cannot be too high, as a larger e represents a larger proportion of each block being used for block-latency updates. This equation and its relation to incentive offers is fully analyzed in Appendix B.

Strategy for the Volatility Award. The effect of larger values of e, representing more providers contributing in each block, on the choice of updates is subtle. Since no direct information naturally passes between the providers who make those updates during the same block, each provider knows only the expected number e, as well as their personal estimate of the value var of the total variation of the data over the duration of one block. Therefore the ideal strategy is for each provider to decide on the sign of the variation (i.e. whether the value goes up or down) and then with probability $\min(1, var/e)$ submit an update with a unit delta of that sign or else of zero.

When var is accurate, less than e, and common knowledge among all providers, this strategy results in an expected total block variation of var. Even if $var > e$ (the ratio exceeds 1), the actual sample may be large enough to reach var if everyone follows the same strategy.

Consensus. The volatility reward itself is a general incentive to participate, especially in response to increased e. In times of genuine volatility, participating honest providers will have the same information about the direction of movement and will make matching updates; in times without volatility, their updates will have no direction. In the former case, the magnitude of the total update in each block will be approximately e while in the latter case, it will be approximately 0 (barring intentionally malicious updates in both cases).

Sortition as rate-limiting. In each round of sortition, the sample size e is the expected number of update transactions that may occur in a single block. This has the effect of throttling updates, which prevents blocks from being monopolized by block-latency feed updates and also slows the rate of updates to the point that successive ones have an opportunity to react to each other, which is desirable for the representation of steady long-term trends. By contrast, multiple updates in the same block must, even in principle, be made blind to each other. This is the domain in which the incentive system encourages volatility.

References

[1] S. Micali, M. Rabin, and S. Vadhan. "Verifiable random functions". In: *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. 1999, pp. 120–130. DOI: `10.1109/SFFCS.1999.814584`.

[2] Sharon Goldberg et al. *NSEC5 from Elliptic Curves: Provably Preventing DNSSEC Zone Enumeration with Shorter Responses*. Cryptology ePrint Archive, Paper 2016/083. `https://eprint.iacr.org/2016/083`. 2016. URL: `https://eprint.iacr.org/2016/083`.

[3] Yossi Gilad et al. *Algorand: Scaling Byzantine Agreements for Cryptocurrencies*. Cryptology ePrint Archive, Paper 2017/454. `https://eprint.iacr.org/2017/454`. 2017. URL: `https://eprint.iacr.org/2017/454`.

[4] Keep network. *AltBn128.sol.* 2020. URL: `https://github.com/keep-network/keep-core/blob/988f0a007c66ce8eafbecec7adbd1cedcee16897/solidity-v1/contracts/cryptography/AltBn128.sol`.

[5] Team Rocket et al. *Scalable and Probabilistic Leaderless BFT Consensus through Metastability.* 2020. arXiv: `1906.08936 [cs.DC]`.

[6] Flare Network. *The Flare network and FLR token.* 2022. URL: `https://flare.network/wp-content/uploads/Flare-White-Paper-v2.pdf`.

Appendices

A Cryptographic sortition

Cryptographic sortition is a way of deterministically yet unpredictably choosing a small sample from a large population of *participants* (for the FTSO, these will be the whitelisted providers), in such a way that no participant can know which others are sampled until they reveal themselves with proof.

This discussion makes open references to the FTSO and its functionality, since that is its application in this proposal. It is not an entirely abstract presentation of cryptographic sortition.

A.1 Elements of Cryptographic Sortition

The technique is based on verifiable random functions (VRFs [1]) and so incorporates both the elements of pseudorandom number generation and cryptographic identity. Effectively, each participant P has a personal pseudorandom function rand_P that is unusable by others. The random numbers thus generated are used to select participants who will be allowed to submit data updates, a process that already has the name *sortition.* This secure PRNG-based sortition is therefore *cryptographic sortition.*

A.1.1 Seed

In each block B there is a *seed* value seed_B that, for each participant P, is the input to rand_P. Successive seeds may be computed either deterministically or randomly, and both are recommended here for security. Seed manipulation is the main avenue for influencing the outcome of sortition and can have two effects:

1. When new participants join, a brute-force computation may allow their particular identities to be chosen advantageously (that is, more likely to be sampled) when entered into sortition using some future seed.
2. When seed succession occurs, a brute-force computation may suggest actions by existing identities that could influence the new seed advantageously to them (that is, more likely to include them in a sample).

A.1.2 Score and Proof

Each participant P generates, in each block B, a number $\mathsf{score}_{B,P}$, where

$$\mathsf{score}_{B,P} = \mathsf{rand}_P(\mathsf{seed}_B).$$

This number cannot be computed by another participant $P' \neq P$ by definition of rand_P, but it is not yet possible to verify that a given number is truly $\mathsf{score}_{B,P}$. This is accomplished by the supplementary value $\mathsf{proof}_{B,P}$ and a matching verification algorithm. Intuitively, given a cryptographic signature scheme sig_P in which signatures are deterministic, and some choice of hash function hash, define:

$$\begin{aligned} \mathsf{proof}_{B,P} &= \mathsf{sig}_p(\mathsf{seed}_B), \\ \mathsf{score}_{B,P} &= \mathsf{hash}(\mathsf{proof}_{B,P}). \end{aligned}$$

Then the verification algorithm is signature verification. Since most signature schemes, in particular elliptic-curve-based ones, are not deterministic, this procedure will fail to give a unique possibility for the value of $\mathsf{score}_{B,P}$, which renders the score useless. An implementation of elliptic curve VRFs ([2, §4.1]) is possible and resembles the above, but manages to contain the nondeterminism just to the proof and not the score. This is the implementation also used in Algorand.

A.1.3 Selection

With VRFs fully implemented, each participant generates only a single acceptable score per block. This is used, similarly as in proof of work, in comparison with some threshold value, there called *difficulty* but more easily understood as an *amenability* A, where participant P is sampled in block B if and only if $\mathsf{score}_{B,P} < A$. If the score has a range up to N (say, $N = 2^{256}$), then the probability of P being sampled is $\Pr_P = A/N$ and, if the pseudorandomness properties of the VRF are adequate, is independently and uniformly distributed among participants.

Weighted Sampling. Applications may not treat all participants equally: in Algorand, wealth is an advantage in sortition, and in this document, signing weight is. With each provider's weight scaled proportionally to $w_{B,P}$ a whole number, P is allowed to generate $w_{B,P}$ scores using equation (3), below. Each score may accompany a different feed update transaction, which are selected independently using the amenability criterion above. Effectively, an actual participant P has a presence as $w_{B,P}$ virtual participants for selection in block B.

A.2 Next Seed Choice

The evolution of the seed from block to block is necessary to perform multiple rounds of sortition. Both pseudorandom and predictable succession are vulnerable to or offer protection against different exploits, and are suggested in combination.

A.2.1 Pseudorandom Base Seed

The FTSO features a random number for each reward epoch, determined by summing independent, arbitrary submissions by providers in their reveal transactions in the previous epoch. This is entirely unpredictable before the first block of the reward epoch, but is moderately susceptible to manipulation, since the adversary could wait until they are certain, or likely to be certain, that all other submissions are known, and then choose whether to reveal their own in order to influence the final result. This is a withholding attack on the seed.

Fortunately, the FTSO also features a *quality* predicate for the random value, which reflects whether any provider's commit was not followed by a corresponding reveal, and therefore whether withholding occurred. This at least provides visibility into attempts to manipulate the random value.

In each reward epoch, numbered n_E, the *base seed* base_{n_E} is defined to be the random value for this epoch.

A.2.2 Predictable Seed Succession

During the reward epoch, blocks numbered n_B allow seeds

$$\begin{aligned} \mathsf{seed}_B &= \mathsf{hash}(\mathsf{base}_{n_E}, n_B), && \text{(without weighting)} \\ \mathsf{seed}_B &= \mathsf{hash}(\mathsf{base}_{n_E}, n_B, i), && \text{(with weighting)}, \forall P (0 \leq i < w_{B,P}) \qquad (3) \end{aligned}$$

where the latter form is used in combination with the weighted sampling process described earlier.

Despite the total predictability of these seeds within a single reward epoch, this is secure against the withholding attack described for pseudorandom succession. Its vulnerability is that, since it is predictable, it can be used to craft an advantaged identity to add as a participant, when that selection happens. This suggests the necessary precaution that the set of participants remain fixed throughout the reward epoch.

As it happens, there is a natural time to update the whitelist of block-latency feed providers, the time it is already done in the FTSO: at the start of a new reward epoch, based on delegations in a block of the previous epoch chosen via the newly active random value. Therefore, providers' identities in the whitelist are committed before knowledge of the base seed that would be manipulated to bias them. This whitelist is valid and unchanging during exactly one reward epoch, during which time predictable succession is used for new seeds.

Manipulation. This method is essentially impossible to manipulate, since the factors that can influence the seed and the selection of identities for the whitelist are outside the adversary's direct control. The quality of the random value makes it obvious when a withholding attack on the base seed occurs, though it does not prevent manipulation, but merely exposes the decreased trustworthiness of selection by sortition during that reward epoch.

A.3 The SortitionCredential Type

According to the implementation [2], a sortition credential should be expressed as

```
struct SortitionCredential {
    uint256 replicate;
    G1Point gamma;
    uint256 c;
    uint256 s;
}
```

where the `replicate` field corresponds to the value i in equation (3) and `G1Point` is a type, defined in the Solidity library `AltBn128.sol` [4], representing a point on an elliptic curve as a single number x plus a sign $\mathsf{sgn}(y)$ to distinguish the branches of the square root, using the Weierstrass form of the curve:

$$y^2 = x^3 + 3 \quad \text{over the field } \mathbb{F}_p,$$

$p = 21888242871839275222246405745257275088696311157297823662689037894645226208583.$

This provider-supplied data is complemented by on-chain information:

```
struct SortitionState {
    uint baseSeed;
    uint blockNumber;
    uint scoreCutoff;

    uint weight;
    G1Point pubKey;
}
```

in which the first three fields represent the other data in equation (3) as well as the amenability (as a cutoff value for the score) A of Appendix A.1.3, and the last two fields represent registered data for the provider sending the transaction.

A.4 Statistics

The sample size obeys a binomial distribution: with n participants and probability of sampling $p = \Pr_P$ for each participant P, the probability of k successes is the binomial distribution

$$B(n, k; p) = \binom{n}{k} p^k (1-p)^{n-k}. \tag{4}$$

It is not the hypergeometric distribution, as in Avalanche [5]. Success for each participant is independent of the others, and the success condition $\mathsf{score}_{B,P} < A$ is not the same as the condition of “having a particular feature of which there are a fixed number in the population”.

The expected number of successes, i.e. the expected sample size, is $e = np = (A/N)n$ with variance approximately also equal to e. The main concern for the variation of k is that it arises that $k = 0$, an interruption in the stream of block-latency feed updates. This probability is of course $(1-p)^n$, or with large n and $p = e/n$, simply $\exp(-e)$, thus descending exponentially from $\exp(-1) \approx 0.37$ when the expected sample size is 1.

B Mathematics of the Volatility Incentive

This appendix concerns the mathematical details relating the format of a volatility incentive offer to Equation (2).

B.1 Differential Form of the Incentive Contribution Equation

Equation (2) governs the effect of an incentive offer through its differential form,

$$dc = A\,dr + \frac{1}{B}\exp(e/B)\,de = A\,dr + \frac{1}{B}(c - Ar)\,de,$$

or,

$$de = B\frac{dc - A\,dr}{c - Ar} = B\,d\log(c - Ar),$$

which is a differential equation that when solved recovers the previous non-differential one. An offer supplies the values of dc and dr, respectively the associated contribution and the specified range increase (the latter possibly capped by the range limit), from which de can be obtained. The post-offer values of c, e, r, and p are respectively

$$c' = c + dc, \qquad e' = e + de, \qquad r' = r + dr, \qquad p' = \frac{r'}{e'}.$$

In addition to the previously stated requirement that dr is nonnegative, de is also required to be nonnegative, since otherwise one could decrease e for free by simply offering a contribution of 0. More strictly, it is required that the *excess* $x = c - Ar$ and its differential $dx = dc - A\,dr$ both be nonnegative (and the former actually positive).

B.1.1 Numerical Concerns

In this transition from the differential equation involving infinitesimal dc, dr, and de to the *finitesimal* version where those values are specific, probably small numbers, it must be determined whether to use c or c', and r or r', in the equation for de. As it is, the value of de is unbounded, and by choosing $dr = 0$ an attacker may buy any amount of de with a proportionally priced offer dc. This is at odds with the desired exponential behavior of the total contribution as a function of e. Therefore, a given incentive offer is applied using the modified differential form

$$de = B\frac{dc - A\,dr}{c' - Ar'} = B\frac{dx}{x + dx}.$$

This fraction is always less than 1, making B the maximum allowed increase in e per incentive offer.

B.1.2 Pricing of the Range Limit

The range limit in an incentive offer may decrease the true value of dr from the value of `rangeIncrease`. As it is, this means that less of the contribution is spent on increasing r and, therefore, more of it is spent on increasing e, which in the event of multiple independent offers being made with the same limit means that $de \approx 1$ may occur repeatedly. To prevent this, only a portion of the offer is accepted, in the same ratio as the true and given values of dr, and refund the rest. This means that once the limit is reached, the offer has no effect and no cost beyond the cost of the transaction itself.