The Flare Data Connector

Flare Network

Abstract

This paper introduces the FDC (Flare Data Connector), Flare's native protocol for registering external events on Flare. The FDC protocol acts in response to user requests: a Flare user submits an attestation request to the FDC, asking Flare's providers to confirm the existence of an external event for use on the network. The FDC then gathers consensus on the veracity of the event amongst Flare's data providers and, assuming that the information in the request was correct, publishes the event on chain. Thus, similarly to the FTSO (Flare Time Series Oracle), the FDC is enshrined on Flare: the same data providers securing Flare's underlying consensus protocol also secure the FDC. The nature of attestation requests submitted to the FDC is flexible, but will typically take the form of bringing information from the wider decentralized blockchain ecosystem onto Flare for use by its smart contracts or dApps. The FDC runs perpetually on Flare, collecting user requests, and publishing a list of confirmed attestation requests every 90 seconds.

1 Introduction

The Flare network provides users and developers with access to high quality data in a secure and decentralized manner. In order to provide this data, Flare is on a mission to enshrine protocols for data uptake. Currently, Flare's primary protocol for gathering data is the Flare Time Series Oracle,[1] a mechanism by which decentralized price estimates for various assets are regularly published directly onto the Flare network. Crucially, this allows the Flare network to attest to information about data external to the network itself: the nature of these assets range from USD values of various tokens to real world assets such as gold.

To increase the utility of Flare's ability to securely provide data, the Flare Data Connector (FDC) introduced in this white paper makes it simpler for Flare users to import external events onto the Flare network. This in turn allows dApps and processes on Flare to operate on information about a wider range of events: off-chain events can be confirmed by Flare's validators, and thus exported to its users as data secured by Flare's underlying consensus.

The FDC provides a mechanism for the Flare network's providers to bring events within the wider blockchain ecosystem, and even beyond, onto Flare. Crucially, the FDC is a user focused protocol: unlike the FTSO, which updates a predetermined list of feeds, users submit requests for particular events of interest to the FDC, and Flare's providers confirm that the data is correct and should be published on Flare, as shown in Figure 1. Thus, rather than continuously monitoring other chains or data sources, which is not a realistic prospect, the FDC is a request based protocol. A user interested in an event, such as a certain transaction existing on an external blockchain in a given block, being available on Flare submits an attestation request for the event to the FDC. The Flare network's providers then confirm the truth of the event, in this case by checking the relevant blockchain, and then together attest to the event.

Once the FDC has confirmed an event, it is ready to be used within the Flare network. In this way, events submitted to the FDC by Flare's users are essentially made canonical on

[1]https://flare.network/wp-content/uploads/FTSOv2-White-Paper-1.pdf

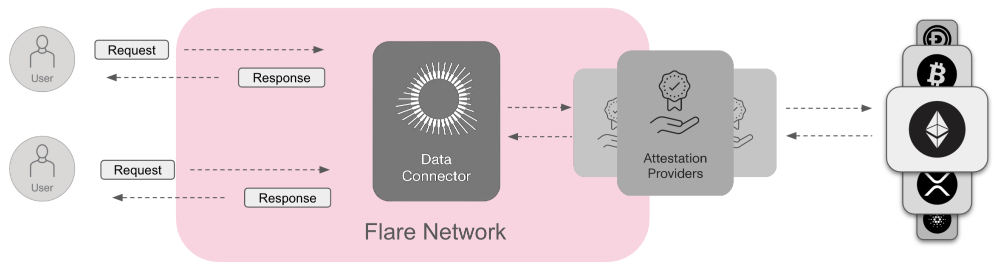

Figure 1: The Flare Data Connector

the Flare network, and their occurrence can be used by smart contracts, or otherwise on the network, with their truth supported by Flare's decentralized group of providers. This allows smart contracts and dApps on the Flare network to depend on external events without additional trust assumptions, as Flare's providers are both backing the FDC and the underlying consensus of the Flare network. For instance, the fact that a specific payment has been made on another blockchain may be required to e.g. free up certain funds on Flare, and trusting external transactions is notoriously challenging. On Flare, the FDC streamlines this process neatly, using only the same trust assumptions as the underlying consensus protocol.

Roadmap. Section 2 describes the phases of the FDC protocol, explaining how requests are collated, consensus is established, and attestations are finalized. Section 3 describes the procedure known as *bit voting*, which is used to determine which requests are handled in a given round of the FDC. Finally, Section 4 explains the mechanism by which providers are rewarded for their efforts in running the FDC.

2 The FDC Protocol

The FDC protocol takes place over a constant sequence of rounds, running every voting epoch on the Flare network - thus, the network is in a perpetual state of collecting and handling requests. The goal of the FDC is to upload information to the Flare network about external events, with each round independently handling newly requested events. That is, each round of the FDC publishes a set of attestations onto the chain, events that are now confirmed according to consensus among Flare's providers. Once an event is attested to, this information can be used in smart contracts or in other functionalities within the wider blockchain ecosystem.

The FDC is a request-based protocol: each round of the FDC has a *collect* phase, a duration of time in which user requests are sent to the protocol to be handled in that round. Following this, the providers collate the requests, determine which ones are to be handled this round, and attest to the outcome of each of them. At the end of each round, a Merkle root is published. This root contains the set of confirmed attestations for the round - the events which are now considered true according to consensus on the Flare network.

Attestation Requests. An attestation request is defined according to its name: it is a request from a user of the Flare network to the FDC for a certain piece of off-chain information, such as an event on an external chain or a tweet, to be confirmed on the Flare network. A well-defined attestation request specifies an event and a data source from which that event can be confirmed, such as the existence of a certain transaction within a given block on a given blockchain. Additionally, it must contain a hash of the expected response. Providers that can attest to this request are confirming that the event did indeed happen as specified.

Consensus amongst Flare's providers that the event occurred allows the event to be confirmed on Flare, and its existence can be used by other protocols on the network.

Attestation Types. In order to be correctly handled by the FDC, each attestation request must conform to one of the pre-defined attestation types, defining the data source and response method for requests of the chosen type. Well-defined response methods must specify the type of information that is sufficient to prove the veracity of the attested event, which will be included in the attestation response.

For example, an attestation type might be the existence of a specific transaction in a certain block on an external network such as Bitcoin. For this attestation type, a valid request must include the data source (e.g. the Bitcoin blockchain), a block number, and a transaction ID, with the response including the header for the next block as verifying information.

Users are able to extend the set of supported attestation types: in order to propose a new type, explicit rules for attestations of that type, such as the data source and response methods, must be clearly defined. Implicitly, they must also convince providers to maintain the required infrastructure to support attestations of the selected type, otherwise they will never be handled by the FDC.

Weights. The FDC is a weighted protocol, in the sense that the contribution to consensus from an individual provider is proportional to their stake. The weight W_j of the jth provider is defined by first computing the proportion of staked FLR tokens and delegated WFLR tokens assigned to their address, relative to the total supply, denoted W_{j^*}. Then, a process known as diversity weighting is applied to the set of providers weights, so that the final weight W_j assigned to the jth provider is

$$W_j = \frac{W_{j^*}^{3/4}}{\sum_i W_{i^*}^{3/4}}.$$

This is the same as the weight used in the FTSO and the Flare Systems Protocol.

Merkle Roots. Each round, providers respond to a collection of attestation requests, with the set of answered requests determined by a process known as bit voting (see Section 3). Once the set of requests to be attested is determined, the information confirming these requests must be packaged into a single Merkle root. For each request, an ABI encoding of the response, defined as part of the attestation type, is hashed to generate the corresponding leaf of the Merkle tree. The root is then generated from these leaves.

2.1 Phases

A round of the FDC lasts around 3 minutes, and proceeds in a sequence of 3 phases. Firstly, the *collect* phase is a 90 second period in which users submit requests for the round. Secondly, the *choose* phase gives providers a 45 second window to determine consensus on which of the submitted requests will be handled. Finally, the *resolution* phase gives the providers 45 seconds to sign the consensus Merkle root and then finalize the round by collecting sufficiently many signatures over a single root.

In order to save gas, certain FDC phases run in parallel with phases of the FTSOv2 Scaling protocol and use the underlying infrastructure of the FSP, allowing transactions to be batched together. The remainder of this section handles the underlying technical details of each phase in turn, and how these are incorporated with the other Flare protocols to optimize gas efficiency.

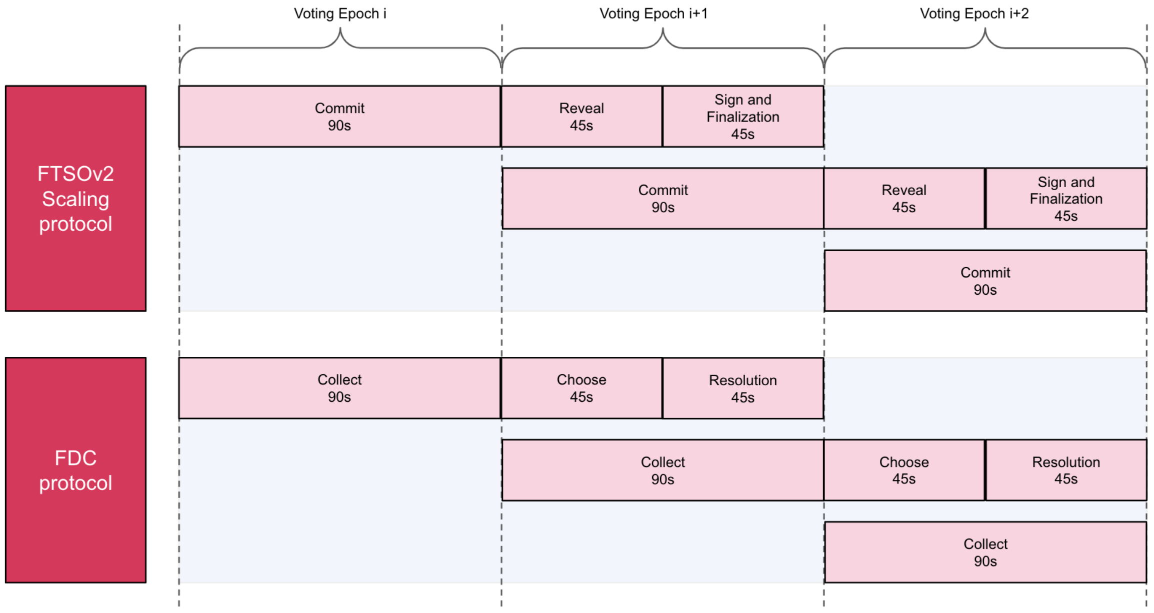

Figure 2: Phases of the FTSOv2 Scaling and FDC protocols.

2.1.1 Collect Phase

The collect phase begins an FDC voting round and has a duration of 90 seconds. During this round, users send attestation request transactions to the chain, which include an event to be attested to and a hash of the expected response to the request. This hash is required to ensure that the response to the request is an attestation of its truth, rather than a response to a question of correctness. In later phases, these requests will need to be indexed: thus, in the collect phase, the requests are ordered chronologically by the time they are received.

2.1.2 Choose Phase

The choose phase begins immediately after the end of the collect phase and lasts for 45 seconds. In this phase, each provider examines the requests made in the collect phase and determines which ones they are able to respond to; some requests may be invalid, or be of an attestation type not supported by the individual provider. They then publish a *bit vote* B^j denoting which requests they are able to respond to. For the jth provider with bit vote B^j, the ith entry of B^j is a 1 if the provider is able to attest the ith request, and a 0 otherwise. Thus, a bit vote is a binary vector of length equal to the number of requests for the round. To ease notation, a request is said to be in B^j if the corresponding entry of B^j is a 1.

Data providers need to submit their corresponding bit vote by the end of the choose phase. To save on gas consumption, providers can batch their submission in the choose phase of the FDC together in the same transaction as the Reveal phase for the FTSO.

2.1.3 Resolution Phase

The resolution phase begins after the choose phase has finished, and lasts for 45 seconds. This phase can be understood in three parts, handled in turn.

Bit Voting. First providers run a local deterministic algorithm, as part of the *bit voting* process, to determine the consensus bit vector $\mathbf{B}$ for the voting round; bit voting is described in more detail in Section 3 and aims to find the best subset of requests that can be jointly confirmed by the network. The bit vector $\mathbf{B}$ is a binary vector with length equal to the number of requests, defining which attestation requests will be confirmed. More explicitly, if the ith entry of the consensus bit vector is 1, then the ith request is included in the current voting round, with the 0 entries denote unhandled requests.

Signing. Secondly, once they have computed the consensus bit vector $\mathbf{B}$, each data provider computes the corresponding Merkle root M, defined by the Merkle tree whose leaves are the hashes of the responses to each attested request. They then sign the root with the (ECDSA) key corresponding to their address, with the jth provider's signature denoted $Sig_j(M)$. This signature is published on-chain. Note that providers will only be rewarded for providing signatures if they do so within the first 10 seconds of the phase or before the round is finalized, whichever comes later.

As in the choose phase, gas consumption is reduced by batching transactions together: each provider can send their signature alongside the signature sent in the sign phase of the FTSO.

Finalization. Thirdly, the round of the FDC is finalized. Once a sufficient weight (50%) of signatures for a matching Merkle root is available on-chain, finalization can be completed. For this process, a random selection of providers are chosen to participate. These providers are selected independently, with probability equal to their weight. Chosen providers are sampled sequentially until enough providers have been included so that at least 5% of the total weight of providers has been selected. This sampling is performed in advance of the round, with shared randomness generated as in the FTSO, so that providers know which rounds they are eligible to finalize in.

The selected providers have a period of 20 seconds, measured from the beginning of the resolution phase, to complete finalization. To do so, they collate enough signatures (those of at least 50% of total provider weight) over the same Merkle root M from the first part of the phase and publish them. To finalize, a collection of signatures is submitted to the finalization contract, along with M itself. Assuming that the signatures are correct and sufficient, the round finalizes, with the root M containing the confirmed attestations for the round. If none of the selected providers completes the round in the allotted 20 seconds, finalization is opened up, with any provider now able to submit a finalization transaction.

Security of the Signing Process. In the resolution phase, each provider implicitly commits to a Merkle root by publishing a signature of the Merkle root M, rather than M itself. In this way, M is only published later in a finalization transaction, when the 50% weight threshold for the signatures is reached. It is important for the security of the FDC protocol that a provider who has not computed M can not reverse-engineer the value of M from the published signatures $Sig_j(M)$. Otherwise, they could piggy back on an existing vote for M, voting for it without computing it; potentially, this could allow a provider to vote for a request that is has not confirmed.

More formally, it is required that the ith provider with access to N signatures $Sig_j(M)$ from N different public keys but who does not know M is unable to compute a valid signature $Sig_i(M)$ for M corresponding to their own public key. This follows from an extension of the message hiding property of the ECDSA signature scheme used: that many signatures of a message with distinct key pairs do not leak enough information about M for a provider to generate a signature over it with their key. This property is shown in Appendix A - note that it relies on specific qualities of the ECDSA scheme, and need not hold for an arbitrary signature scheme.

One side effect of the this mechanism is that finalization can only be performed by a provider who knows M, as they must submit M as part of the finalization transaction; merely submitting a sufficiently large weight of signatures is not enough. This is only a problem in cases where all providers selected to finalize were not able to compute M, perhaps because it contains information from data sources they do not support. However, this outcome is unlikely, and even in this case once finalization is opened up to all providers plenty of providers will be able to finalize promptly.

Gas Consumption. In order to minimize gas consumption, certain transactions required of the providers in the FDC are submitted as part of transactions already required of the same providers within Flare's pre-existing FTSO protocol. This is facilitated by having the protocols run concurrently (as in Figure 2), so that the transactions in the appropriate phases of the FDC can be submitted alongside other required. In this way, the gas consumption by providers in those transactions is reduced to the additional call data required for the FDC transactions batched in with FTSO ones, which is minimal. Additional gas is consumed by the users of the protocol when submitting attestation requests; the users must pay these gas fees themselves.

3 Bit Voting

As part of the choose phase, data providers must decide which attestation requests made to the FDC will be answered in the current voting round. Not all providers will be able to verify every request: for example, they may not perfectly align on which external data sources they support, or they may not store the same amount of historical data. Thus, a process must be in place to determine which requests will be handled for the round. Consensus signing on Flare requires any more than 50% agreement in terms of provider weight, so in abstract any request which above 50% of provider weight can attest could be confirmed. However, since the attestations are packaged into a single Merkle root M, each provider that signs the root must answer the exact same set of requests. Thus, an agreed set of attestations must be determined. Given that the dual goals for a round of the FDC are to maximize both request throughput and the fees earned by providers for successful attestations, the goal of the choose phase is to determine a set of attestations that has both a large total amount of requests and fees, and a good degree of total support from providers.

The method for doing this is known as *bit voting*. Each provider submits a binary vector known as a *bit vote*, with the jth provider submitting a vector B^j whose ith entry is 1 if the provider can attest the ith request and 0 if they cannot[2]. The bit voting procedure takes as input the set of requests r_i, fees for those requests f_i, and bit votes B^j; it outputs a consensus bit vector $\mathbf{B}$ such that

- The combined weight of providers supporting every request in $\mathbf{B}$ is above 50%
- A large amount of fees are rewarded to providers for attesting to the requests included in $\mathbf{B}$

with the requests in $\mathbf{B}$ denoting the confirmed requests for the round. An ideal consensus bit vector should have both wide support, to ensure the validity of the requests, and a large amount of fees, both to ensure most valid requests are handled and to adequately reward

[2]Alongside this, providers submit two bytes indicating the number of requests they have observed this round, and discard bit votes from providers whose number does not match their own. This does not effect the ability of the procedure to find consensus on a bit vector with 50% or more support.

providers. Thus, the goal of the bit voting process is to define a good compromise between the dual goals: to determine a consensus bit vector handling a large number of requests that are supported by many providers.

In theory, there is some canonical bit vector $\mathbf{B}^*$ maximising the amount of fees earned for the round, or indeed any other chosen metric for an optimal bit vector. However, in many cases there are too many possible combinations of providers and bit vectors to deterministically find $\mathbf{B}^*$ with acceptable efficiency. Thus, a tree search procedure is run by providers off-chain to determine an acceptable consensus bit vector $\mathbf{B}$ that performs well against a certain value function. The remainder of this section describes the value function and associated bit voting procedure.

3.1 Value Function

For a candidate consensus bit vector $\mathbf{B}$, let $S_\mathbf{B}$ denote the set of (indices of) all providers whose bit vote supports $\mathbf{B}$, where a provider supports $\mathbf{B}$ if they can attest to all requests on which $\mathbf{B}$ has value 1. That is, j is in $S_\mathbf{B}$ if $B^j_i = 1$ for each i such that $\mathbf{B}_i = 1$. Note B^j need not be 0 on all indices for which $\mathbf{B}$ is 0, as the provider may be able to handle more requests than contained in the consensus bit vector. Then, the value function on an input bit vector $\mathbf{B}$ is defined as

$$V(\mathbf{B}) = \min\left(\sum_{j\in S_\mathbf{B}} W_j\ ,\ 0.8\right) \cdot \sum_{i:\mathbf{B}_i=1} f_i$$

where W_j denotes the weight of the jth provider and f_i the fee of the ith request.

This can be extended to the value of a single request or index: the value of the ith request $V(r_i)$ is equal to the fee of the request multiplied by the (capped) support, e.g.

$$V(r_i) = \min\left(\sum_{j:B^j_i=1} W_j\ ,\ 0.8\right) \cdot f_i.$$

Similarly, the value function can be extended to a bit vote B^j by setting the value $V(B^j)$ as equal to the weight of the voter multiplied by the fees of the supported requests, e.g.

$$V(B^j) = W_j \ \cdot \sum_{i:B^j_i=1} f_i.$$

This value function is chosen to balance the need for throughput and provider rewards: provider rewards roughly correspond to the value of handled requests, distributed to the weight of supporting providers. Simpler value functions, such as those optimising entirely for throughput or supporting weight, do not balance the dual requirements of rewards and throughput simultaneously. The aim of the bit voting process is to select a consensus bit vector $\mathbf{B}$ that performs well, preferably optimally, against this value function.

Capping. In the above process, the combined weight $\sum_{j\in S_\mathbf{B}} W_j$ of providers supporting the bit vector $\mathbf{B}$ does not contribute directly to the value function. Instead, weight supporting a consensus bit vector beyond 80% of provider weight is not taken into account. In practice, once the support for a bit vector is sufficiently high, further support does not improve consensus: for example, two bit vectors with 80% support or 100% support are functionally equally valid on Flare, as only 50% is needed for consensus - and even if some support is lost between the bit voting and signing (perhaps due to provider misbehaviour or services outages), either will typically sit well above the 50% threshold after the reveal phase. Thus, the support of a vector in the value function can safely be capped. It is expected that many vectors will reach this support cap, with the result that the consensus bit vector should be the vector with maximal fees amongst those reaching the support cap. This means that provider rewards are not maximized, but the trade-off is an increase in throughput, slightly shifting the balance of the value function.

3.2 The Bit Voting Process

The bit voting process is performed off-chain: providers use the list of requests and associated fees f_i and the votes of all providers B^j, and calculate the consensus bit vector $\mathbf{B}$. A tree-search method is used to test likely candidate bit vectors for a fixed duration, outputting the tested bit vector $\mathbf{B}$ which maximises the value function. This vector becomes the consensus bit vector for the round, with its included requests defining the corresponding Merkle root M. In cases where the space of possible bit vectors is small enough to be exhaustively searched, this will be the optimal choice; in other cases, it will just be a good choice. Crucially, the algorithm is deterministic, in the sense that each provider will find the same $\mathbf{B}$.

The process can be understood in three phases, which are described in more detail further on:

- First, the list of votes and requests is pre-processed, batching together sets of similar votes and requests to avoid unnecessary computation.
- Then, four binary trees are constructed to organize the space of possible consensus bit vectors in a methodical way.
- Finally, the trees are traversed efficiently to test likely consensus bit vectors. The highest value tested vector is chosen as the consensus bit vector for the round.

3.2.1 Pre-Processing

Before beginning with the tree search, the input space (the set of votes, weights, and fees) is simplified to streamline the search process. The goal of the pre-processing is to prevent time wasted exploring bad or redundant candidate bit vectors. To that end, the following steps are taken:

- All requests with less than 50% total weight support are discarded, as it is not possible to reach consensus on such requests.
- All votes that support all remaining requests are set aside and will always be included: these votes will support any sensible bit vector.
- All votes that support none of the remaining requests are discarded, as they can not support any candidate bit vectors.
- All requests supported by all remaining bit votes are set aside and will always be included: since all votes support these requests, they can be included in a bit vector without compromising its weight.
- Sets of votes that agree on all remaining requests are aggregated into single votes with weight equal to the combined weight of the providers who submitted those votes.
- Sets of requests on which remaining bit votes agree are aggregated into single requests with fee equal to the combined fees of those requests.

The final two steps of this pre-processing are to simplify the search procedure: votes that agree on all attestation requests or attestation requests on which all votes agree can be treated as single large votes or requests for the purposes of searching for high value bit vectors.

3.2.2 Constructing Trees

After the pre-processing stage is complete, binary trees are constructed to order the remaining space of candidate consensus bit vectors. Each tree allows the space to be searched in a different order: two trees focus on navigating the space by bit votes, and another two do so by requests.

Nodes. Both types of tree will use the same type of nodes; the difference will be in how the trees are constructed and traversed. Each node N of a tree will store a set of votes V_N and requests R_N, along with their associated weights and fees. Every node (implicitly) contains the requests and votes that were defined to be always included as part of the pre-processing, with the root node of each tree containing all votes and all requests. Other requests and votes will be discarded from nodes as the trees are constructed.

Using this definition, a node can be assigned a value. The value of a node is defined using the value function for bit vectors: the (capped) weight of stored votes multiplied by the remaining fees, e.g.

$$V(N) = \min\left(\sum_{j \in V_N} W_j, 0.8\right) \cdot \left(\sum_{i \in R_N} f_i\right).$$

Tree Structure. All trees will be constructed in a somewhat similar manner. Each tree will arrange the nodes either by requests or by votes, such that each level of the tree corresponds to a particular request or vote, and the depth of the tree will be equal to the number of them. At each level, child nodes will represent a decision to include or exclude the particular request (or vote), with including a request (or vote) resulting in removing non-compatible votes (or requests). At the leaves of the tree, each vote or request has been decided upon: thus, the leaves correspond to candidate consensus bit vectors.

There are two trees each for sorting by requests or by votes, representing different orders of the underlying objects, resulting in four trees in total. Each tree will be described in more detail below, but in summary they are constructed as follows:

- Choose either votes or requests to sort the space by.
- Select an ordering for the post-processed requests (or votes).
- Initialize a root node containing all requests and votes.
- At each level of the tree, each node's children is determined by either including or excluding the request (or vote) corresponding to that level.
- At the end of this process, each leaf corresponds to a candidate consensus bit vector.

Attestation Request Trees. The trees ordered by attestation requests select an ordering $r_1, \ldots, r_n$ for the post-processed requests. Then, at depth d the children of a node are determined by either including the request r_d and excluding votes that do not support r_d, or including the votes and excluding the request. More formally, once an order for the requests is chosen, child nodes are determined according to the following rule, beginning from the root node containing all requests and votes:

- The left hand child (child 0) at depth d is defined by removing the request r_d from the set of requests of the parent node.
- The right hand child (child 1) is defined by removing all votes that do not support r_d from the parent node.

This process is repeated until each request has been considered e.g. the depth of the tree is equal to the number of requests that were left after the pre-processing phase. Removing requests or votes from a node decreases the value of the node correspondingly; thus, the value of a child node can not exceed the value of its parent, a fact that will be used in all trees to simplify searching.

Leaves of the tree correspond to candidate bit vectors containing precisely the requests r_d for which the path from the root node to the leaf traverses the right hand (child 1) node at depth d. The value of a leaf corresponds to the value of the candidate consensus bit vector it represents.

Two possible orderings of the requests are considered, resulting in two separate trees: first in descending order of value, then in ascending order, with ties broken in ascending order of arrival time in the collect phase[3].

Vote Trees. The vote-based trees are constructed in a similar manner, except that children are determined by deciding on individual votes rather than requests. Again, starting with a root node containing all requests and votes, and denoting by v_j the jth vote remaining after pre-processing, the two children of a node at depth d are defined according to the following rule:

- The left hand child (child 0) is defined by removing the dth vote from the parent node.
- The right hand child (child 1) is determined by removing all requests not supported by the dth vote from the parent node.

Once again, the process is iterated until all votes have been considered, with the depth of the tree equal to the number of votes remaining after pre-processing. Each leaf corresponds to a candidate bit vector consisting of remaining requests, those supported by all the votes where the right hand child is traversed in the path from the root to the leaf.

As before, two possible orderings of the votes are considered, giving two more trees: first in descending order of value, then in ascending order, with ties sorted in ascending order of publication in the choose phase[4].

3.2.3 Traversing the Trees

Each leaf of a tree represents a candidate bit vector with a given value. Furthermore, the space of all leaves of any given tree represents all possible consensus bit vectors: thus, traversing a single tree in its entirety would be sufficient to find the optimal consensus bit vector. However, performing this traversal may not be possible within the time frame allotted by the commit phase.

To handle this issue, only a bounded number of nodes are visited, determined in the manner described below. Then, the bit vector corresponding to the visited leaf with the highest value is chosen as the consensus bit vector for the round. Thus, an order is required for searching the trees and visiting the nodes. The process works as follows:

[3]For requests that are grouped during pre-processing, the lowest arrival time amongst the batched requests is used.

[4]Again, votes batched together during pre-processed use the lowest arrival time amongst those votes for breaking ties.

1. Initialize a counter at 0 and a maximal leaf value tracker at 0.
2. Determine whether to search by request or by votes first: if there are less requests, start by searching by requests, otherwise, start by votes.
3. Construct the first two trees corresponding to whether the search is by request or votes.
4. Starting from the root node of each of the two trees, traverse the trees in parallel. For the descending value tree, the tree is searched from the right, taking the right hand child 1 node at every opportunity and setting the value tracker to the value of that leaf. Each time a node is traversed, increment the counter by 1. For the ascending value tree, search from left to right: take the left hand child 0 node at each opportunity. Then, return to the root node and begin to traverse the tree step-by-step, from node to leaf, heading towards the next leaf in order (either from the right or left). At each node, check the value $V(N)$: if the value does not exceed the current maximum value, do not follow the path any further, as the resulting leaf cannot exceed the current maximum value.
5. Whenever a leaf is visited with higher maximum value, update the maximal value tracker to the value of that leaf.
6. Repeat the process until all leaves have been visited or the counter reaches a maximum value threshold C, whichever comes first.
7. If the process was terminated by searching the whole space, return the visited leaf with the highest value.
8. Otherwise, begin the same search for the remaining trees with a fresh counter, choosing paths in the same manner (right-first for descending order, and left-first for ascending) until the whole tree is traversed or the counter reaches its maximum C. Note that the maximal leaf value tracker is not reset, so that paths can be discarded if they do not exceed the maximum value found in the first set of searches.
9. Return the consensus bit vector $\mathbf{B}$ corresponding to the leaf achieving the maximum value tracker across both trees. In case of a tie, return the leaf from the first search.

The maximum counter value parameter is set by governance, and is required to trade-off the need to check enough candidate bit vectors against the need for the algorithm to run sufficiently fast.

At the end of this process, each provider will have computed the same consensus bit vector $\mathbf{B}$, which determines the included requests for the round.

4 Incentivization

As with all protocols on the Flare network, providers are incentivized to ensure that they participate correctly in the FDC. Incentives align with the goals of the FDC: serving attestation requests for correct events and establishing consensus on what events are correct. Thus, rewards are provided for voting on and attesting to requests, with providers rewarded relative to their weight.

Rewards for the FDC come from two sources: inflationary funds within the Flare network and reward offers provided as the fees of the attestation requests. Each attestation request has an individual reward offer provided by the requester: requests with higher fees are prioritized by the system, reflected in them being more likely to be selected during the bit voting process described in Section 3.

Inflationary Rewards. Each voting round of the FDC shares an inflationary reward amongst contributing providers. Ultimately, the FDC should be self-sustaining, with the fees from the requests maintaining the system without the need for inflationary rewards. However, this may not initially be the case - thus, the inflationary rewards can be seen as an incentive for providers to build FDC-compatible infrastructure, and in the future may be removed or reduced.

The inflationary rewards for a given round are assigned on a per reward epoch basis, with a quantity I initially assigned to the epoch. Then, for each attestation type att_i, an inflationary reward weight $I_i \in [0, 1)$ and a threshold T_i are determined by governance. For each type, if the number of attestations of that type in a given reward epoch is less than this threshold, a proportion of the total inflationary reward I is burnt equal to the corresponding inflationary weight I_i. The total inflationary reward I_{reward} for the epoch is then

$$I_{reward} = I \cdot \sum_{i \in A} I_i$$

where A is the set of indices of types whose threshold number of attestations were met in the epoch. This reward is then split evenly among voting rounds, so that the inflationary reward I_{round} available to each voting round in the reward epoch is equal to

$$I_{round} = I_{reward}/N_{rounds},$$

where N_{rounds} is the number of voting rounds in the reward epoch, which is typically 3360.

Value of a Round. The value of a round of the FDC is computed as the sum of all fees for confirmed requests (those supported in the consensus bit vector $\mathbf{B}$) and available inflationary rewards. Fees for unconfirmed requests are burnt. Each request has its own fee f_i, set by the requester, which must exceed a base fee, the value of which is a parameter that can be set by governance. Thus, the value $Val(r)$ of a round r is

$$Val(r) = I_{round} + \sum_{\{i:\mathbf{B}_i=1\}} f_i.$$

This reward is then split into two parts: 90% of it is assigned for rewarding the signing of Merkle roots, and 10% is set aside for finalization, so that the available rewards are

$$Val_M(r) = 0.9 \cdot Val(r)$$

for signing the correct Merkle root and

$$Val_F(r) = 0.1 \cdot Val(r)$$

for successful finalization.

Delegation Rewards As with other Flare protocols, providers will be rewarded relative to their weight W_i. A provider's weight is a combination of staked FLR owned by the provider itself and delegations from Flare's users. Correspondingly, each providers' rewards for participation in the FDC are split between the provider and its delegators, each receiving a share of the rewards proportional to the amount of weight they give to the provider. The provider may charge the delegators a percentage fee for its services. For the rest of this section, unless otherwise specified any time rewards are assigned to a provider it is implied that these rewards are split accordingly.

Rewarding Attestations. The value of the round represents the total available reward for the round given the confirmed requests. However, this value is distributed to providers proportionally to their weight. For the jth provider with (normalized) weight W_j, assuming that the provider both published a bit vote supporting the consensus bit vector and signed the correct consensus Merkle root in the first 10 seconds of the resolution phase[5], the reward $R_{M,j}(r)$ assigned to the provider in round r for signing is:

$$R_{M,j}(r) = W_j \cdot Val_M(r),$$

the proportion of the value of the signing reward that corresponds to that providers weight.

Burnt Rewards for Unsuccessful Providers. This rewarding mechanism burns a proportion of the value of the round: providers are rewarded relative to their weight against all providers, not just the successful ones. Thus, some of the rewards are burnt that would have been earned had all providers been successful. This is to properly incentivize providers to submit bit votes supporting every attestation request that they can verify. If instead the value was distributed to providers relative to their weight amongst successful providers, it could be in a provider's interest to somehow minimizing the support of the winning bit vector, in order to maximise individual rewards. In turn, this may incentivize not publishing the densest possible bit vote. Since this is undesirable, normalized weight within the entire system is used instead, with rewards that would have been assigned to unsuccessful providers being burnt.

Burnt Rewards for Bad Voting. An additional source of burning occurs in cases of providers signing the correct Merkle root in rounds where they submitted a bit vote that did not support the consensus Merkle root or did not submit a bit vote at all: such providers are still rewarded for helping to build consensus, but their sub optimal participation in the bit voting process induces a partial punishment. These providers are only given 80% of their assigned rewards. That is, a provider who signs the correct Merkle root but did not submit a vote that supports the consensus bit vector is allocated the reward:

$$R_{M,j}(r)' = 0.8 \cdot R_{M,j}(r).$$

Finalization Rewards. Finalization rewards are assigned to the selected providers who successfully complete the finalization process within the 20 second grace window. Each provider who successfully completes this finalization receives their share of the reward. That is, let F denote the set of providers selected to finalize the round. Then, a provider with index j in F who successfully submits a valid finalization transaction in the 20 second time window receives an equal share of the finalization reward

$$R_{F,j}(r) = Val_F(r) \cdot \frac{1}{|F|}.$$

Note that while only the first finalization transaction submitted is successful, the other selected providers must still submit a valid transaction to receive rewards. The share of the rewards for each selected provider who does not submit a valid transaction is burnt.

If none of the selected providers complete a finalization transaction in the allotted time, finalization is opened up and the first provider to complete a finalization transaction window after this time receives the entire available reward $Val_F(r)$. In this case, the rewards are assigned entirely to the finalizing provider, and not shared with its delegators.

[5]Or before the round is finalized

Penalization. As well as rewarding the providers for correct participation in the FDC protocol, incentives against explicitly malicious behaviour are introduced in the form of punishments. There are a variety of types of behaviour that must be penalized:

- Submitting a bit vote that supports the consensus bit vector, but then not signing the valid consensus Merkle root before the end of the round.
- Repeating and contradicting transactions, such as submitting signatures for multiple different Merkle roots in a round.
- Submitting a signature for a Merkle root that is not equal to the consensus Merkle root for the round.

A provider who is penalized in any way in a given voting round loses a quantity of their rewards in the reward epoch equal to 30 times their expected signing rewards for the voting round. That is, a provider with weight W_j who is penalized in round r receives a penalization amount

$$P_j(r) = 30 \cdot W_j \cdot \mathit{Val}_M(r).$$

Note that this penalization quantity $P_j(r)$ is taken from the provider's rewards from the reward epoch across all Flare protocols, and thus it is possible for a provider to lose rewards from other protocols if they receive penalizations in excess of received FDC rewards across the reward epoch. Additionally, the formula $P_j(r)$ for each penalization is a parameter set by governance, and may change as the system evolves.

Appendices

A Signing in the Resolution Phase

As part of the security argument of the signing process in Section 2.1.3, a particular property of the ECDSA signature scheme was required. Signatures are published somewhat asynchronously; thus, in order to prevent providers who do not know the Merkle root M producing a valid signature for it, an extension of message hiding is needed. In particular, if n providers have published signatures over M using n distinct private keys, it should not be possible for another provider who does not know M to produce a valid signature of it for their own key pair. The requisite property of the ECDSA signature scheme is shown below.

To understand the proof, some details of the ECDSA signature scheme are necessary. A well-defined instance of the ECDSA scheme requires a base point G of prime order N, with public-private key pairs taking the form $(d \times G, d)$ for a randomly chosen integer $d \in [1, N-1]$. Then, the signing process of a message M in the ECDSA scheme consists of the following steps

1. Sample a random integer $k \in [1, N-1]$.
2. Compute the point $(x, y) = k \times G$.
3. Compute $r = x \mod N$ and $s = k^{-1}(M + rd) \mod N$.
4. Publish the pair (r, s) as the signature.

Note that if either of the pair (r, s) are 0, the signature is invalid, and the process is repeated with a freshly sampled k until neither r nor s is 0.

Rather than directly prove the security of the root signing process for an arbitrary signature scheme, properties of the underlying ECDSA instance will be used instead. Namely, it is secure under the forgery resistance of ECDSA: that seeing n signatures for distinct messages under the same public key pk is not sufficient to forge a new signature of a new message under pk. Below, it is shown that the security of the asynchronous signing process follows from the forgery resistance property of the ECDSA scheme.

Theorem 1. *Let $\{(pk_i, sk_i)\}_{i=1,\dots,n+1}$ be $n+1$ distinct ECDSA key pairs. Then, under the forgery resistance assumption of the ECDSA, it is not possible to produce a valid signature $Sig_{n+1}(M)$ of an unknown, uniformly random, message M for the last key pair given access to n signatures $\{Sig_i(M)\}_{i=1,\dots,n}$ over M, each corresponding to a different one of the first n key pairs.*

Note that the statement of the theorem assumes that the unknown message M, corresponding to the Merkle root for the round of the FDC, is uniformly random. Thus, the security argument assumes that the Merkle root is uniformly random. For this, it is assumed that hashing the information proving the correctness of the handled attestation requests into a Merkle root produces uniformly random values from the perspective of an attacker who does not know the proofs. If an attacker were somehow able to guess M without computing it, the security of the system would be compromised; however, in such a situation, it is clearly not possible to prevent the attacker from producing a signature of M.

Proof. Assume, for a contradiction, that there is an algorithm $\mathcal{A}$ that is able to solve the underlying problem $\mathcal{P}$: on input the n signatures $\{Sig_i(M)\}_{i=1,\dots,n}$ over M from n distinct key pairs and a new key pair (pk_{n+1}, sk_{n+1}), an algorithm $\mathcal{A}$ solves $\mathcal{P}$ if it is able to output a valid signature $Sig_{n+1}(M)$ of M corresponding to pk_{n+1}. To complete the proof, it is shown that an algorithm solving a simplified version of $\mathcal{P}$, where it gets additional access to the private keys $\{sk_i\}_{i=1,\dots,n}$, is able to contradict the forgery resistance assumption of the ECDSA scheme; the validity of this simplification follows from the fact that providing $\mathcal{A}$ the private keys makes the problem easier.

For each i, let $(r_i, s_i) := Sig_i(M)$ denote the two elements of the ECDSA signature. Then define

$$\begin{aligned} s_i' &= r_i \cdot s_i \\ &= k_i^{-1}(r_i^2 \cdot d_i + r \cdot M) \end{aligned}$$

where d_i is the ith secret key and k_i the random integer generated during the signing procedure that produced (r_i, s_i). Now, the pairs

$$(r_i, s_i') = (r_i, k_i^{-1}(r_i^2 \cdot d_i + r_i \cdot M))$$

correspond to n valid signatures over n distinct uniformly random messages $r_i^2 \cdot d_i$ with private key M and public key $M \times G$.

Since $\mathcal{A}$ is able to solve $\mathcal{P}$ with access to the private keys, it can output a signature (r_{n+1}, s_{n+1}) of M with key pair (pk_{n+1}, sk_{n+1}). Using this signature, $\mathcal{A}$ can create the pair (r_{n+1}, s_{n+1}'), which corresponds to a valid signature over d_{n+1} with private key M.

Since the pair (r_{n+1}, s_{n+1}') is a valid forgery of an ECDSA signature, it follows that such an algorithm $\mathcal{A}$ can not exist under the forgery resistance assumption of the ECDSA. □