F1 Fee Distribution Draft-00

Dev Ojha

October 22, 2018

Abstract

In a proof of stake blockchain, validators need to split the rewards gained from transaction fees each block. Furthermore, these fees must be fairly distributed to each of a validator's constituent delegators. They accrue this reward throughout the entire time they are delegated, and they have a special operation to withdraw accrued rewards.

The F1 fee distribution scheme works for any algorithm to split funds between validators each block, with minimal iteration, and the only approximations being due to finite decimal precision. Per block there is a single iteration over the validator set, which enables only rewarding validators who signed a given block. No iteration is required to delegate, and withdrawing only requires iterating over all of that validators slashes since delegation it began. State usage is minimal as well, one state update per validator per block, and one state record per delegator.

1 F1 Fee Distribution

In a proof of stake model, each validator has an associated stake, with delegators each contributing some amount to a validator's stake. The validator is rewarded transaction fees every block for the service they are providing the network. In the F1 distribution, each validator is permitted to take a commission from the fees they receive, and the remaining fees should be evenly distributed across the validator's delegators, such that every delegator the percentage of the validator's stake that came from the delegator is the proportion of that validator's remaining tx fees which they are getting. Iterating over all delegators for every validator each block is too expensive for a blockchain environment. Instead there is an explicit withdraw fees action which a delegator can take, which will give the delegator the same total amount of fees as though they were receiving it every block.

Suppose a delegator delegates x stake to a validator at block h. Let the amount of stake the validator has at block i be s_i and the amount of fees they receive at this height be f_i. Then if a delegator contributing x stake decides to withdraw at block n, the rewards they receive is

$$\sum_{i=h}^{n} \frac{x}{s_i} f_i = x \sum_{i=h}^{n} \frac{f_i}{s_i}$$

However s_i will not change every block. It only changes if the validator gets slashed, or if someone new has bonded or unbonded. Handling slashes is relegated to subsection 2.2. Define a period as the set of blocks between two changes in a given validator's total stake. A new period begins every time that validator's total stake changes. The above iteration will be converted to iteration over periods. Let the total amount of stake for the validator in period p be n_p. Let T_p be the total fees this validator accrued within this period. Let h be the start of period p_{init}, and height n be the end of p_{final}. It follows that

$$x \sum_{i=h}^{n} \frac{f_i}{s_i} = x \sum_{p=p_{init}}^{p_{final}} \frac{T_p}{n_p}$$

Let p_0 represent the period from when the validator first bonded until the first change to the validators stake. The central idea to the F1 model is that at the end of the kth period, the following is stored at a state location indexable by k: $\sum_{i=0}^{k} \frac{T_i}{n_i}$. When a delegator wants to delegate or withdraw their reward, they first create a new entry in state to end the current period. Let the index of the current period be f. Then this entry is created using the previous entry as follows:

$$\sum_{i=0}^{f} \frac{T_i}{n_i} = \sum_{i=0}^{f-1} \frac{T_i}{n_i} + \frac{T_f}{n_f} = entry_{f-1} + \frac{T_f}{n_f}$$

Where T_f is the fees the validator has accrued in period f, and n_f is the validators total amount of stake in period f.

The withdrawer's delegation object has the index k for the period which they started accruing fees for. Thus the reward they should receive when withdrawing is:

$$x \left(entry_f - entry_k\right) = x \left(\left(\sum_{i=0}^{f} \frac{T_i}{n_i} \right) - \left(\sum_{i=0}^{k} \frac{T_i}{n_i} \right) \right) = x \sum_{i=k}^{f} \frac{T_i}{n_i}$$

The first summation is the state entry for f, and the second sum is the state entry at k. It is clear from the equations that this payout mechanism maintains correctness, and required no iterations.

T_f is a separate variable in state for the amount of fees this validator has accrued since the last update to its power. This variable is incremented at every block by however much fees this validator received that block. On the update to the validators power, this variable is used to create the entry in state at f.

This fee distribution proposal is agnostic to how all of the blocks fees are divied up between validators. This creates many nice properties, for example only rewarding validators who signed that block.

2 Additional add-ons

2.1 Commission Rates

Commission rates are the idea that a validator can take a fixed $x\%$ cut of all of their received fees, before redistributing evenly to the constituent delegators. This can easily be done as follows:

In block h a validator receives f_h fees. Instead of incrementing that validators "total accrued fees this period variable" by f_h, it is instead incremented by $(1 - commission\_rate) * f_p$. Then $commission\_rate * f_p$ is deposited directly to the validator. This scheme allow for updates to a validator's commission rate every block if desired.

2.2 Slashing

Slashing is distinct from withdrawals, since not only does it lower the validators total amount of stake, but it also lowers each of its delegator's stake by a fixed percentage. Since noone is charged gas for slashes, a slash cannot iterate over all delegators. Thus we can no longer just multiply by x over the difference in stake. The solution here is to instead store each period created by a slash in the validators state. Then when withdrawing, you must iterate over all slashes between when you started and ended. Suppose you delegated at period 0, a y% slash occured at period 2, and your withdrawal is period 4. Then you receive funds from 0 to 2 as normal. The equations for funds you receive for 2 to 4 now uses $(1 - y)x$ for your stake instead of just x stake. When there are multiple slashes, you just account for the accumulated slash factor.

In practice this will not really be an efficiency hit, as we can expect most validators to have no slashes. Validators that get slashed a lot will naturally lose their delegators. A malicious validator that gets itself slashed many times would increase the gas to withdraw linearly, but the economic loss of funds due to the slashes should far out-weigh the extra overhead the withdrawer must pay for due to the gas.

2.3 Inflation

Inflation is the idea that we want every staked coin to grow in value as time progresses. Each block, every staked token should each be rewarded x staking tokens as inflation, where x is calculated from function which takes state and the block information as input. This also allows for many seemless upgrade's to x's algorithm. This can be added efficiently into the fee distribution model as follows:

Make each block have an inflation number, by which every staked token should produce x additional staking tokens. In state there is a variable for the sum of all such inflation numbers. Then each period will store this total inflation sum in addition to $\sum_{i=0}^{end} \frac{T_i}{n_i}$. When withdrawing perform a subtraction on the

inflation sums at the end and at the start to see how many new staking tokens to produce per staked token.

This works great in the model where the inflation rate should be dynamic each block, but apply the same to each validator. Inflation creation can trivially be epoched as long as inflation isn't required within the epoch, through changes to the x function.

Note that this process is extremely efficient.

The above can be trivially amended if we want inflation to proceed differently for different validators each block. (e.g. depending on who was offline) It can also be made to be easily adapted in a live chain. It is unclear if either of these two are more desirable settings.

2.4 Delegation updates

Updating your delegation amount is equivalent to withdrawing earned rewards and a fully independent new delegation occuring in the same block.

2.5 Jailing / being kicked out of the validator set

This basically requires no change. In each block you only iterate over the currently bonded validators. So you simply don't update the "total accrued fees this period" variable for jailed / non-bonded validators. Withdrawing requires *no* special casing here!

3 State pruning

You will notice that in the main scheme there was no note for pruning entries from state. We can in fact prune quite effectively. Suppose for the sake of exposition that there is at most one delegation / withdrawal to a particular validator in any given block. Then each delegation is responsible for one addition to state. Only the next period, and this delegator's withdrawal could depend on this entry. Thus once this delegator withdraws, this state entry can be pruned. For the entry created by the delegator's withdrawal, that is only required by the creation of the next period. Thus once the next period is created, that withdrawal's period can be deleted.

This can be easily adapted to the case where there are multiple delegations / withdrawals per block. Keep a counter per state entry for how many delegations need to be cleared. (So 1 for each delegation in that block which created that period, 0 for each withdrawal) When creating a new period, check that the previous period (which had to be read anyway) doesn't have a count of 0. If it does have a count of 0, delete it. When withdrawing, decrement the period which created this delegation's counter by 1. If that counter is now 0, delete that period.

The slash entries for a validator can only be pruned when all of that validator's delegators have their bonding period starting after the slash. This seems

ineffective to keep track of, thus it is not worth it. Each slash should instead remain in state until the validator unbonds and all delegators have their fees withdrawn.

4 Implementers Considerations

This is an extremely simple scheme with many nice benefits.

- The overhead per block is a simple iteration over the bonded validator set, which occurs anyway. (Thus it can be implemented "for-free" with an optimized code-base)
- Withdrawing earned fees only requires iterating over slashes since when you bonded. (Which is a negligible iteration)
- There are no approximations in any of the calculations. (modulo minor errata resulting from fixed precision decimals used in divisions)
- Supports arbitrary inflation models. (Thus could even vary upon block signers)
- Supports arbitrary fee distribution amongst the validator set. (Thus can account for things like only online validators get fees, which has important incentivization impacts)
- The above two can change on a live chain with no issues.
- Validator commission rates can be changed every block
- The simplicity of this scheme lends itself well to implementation

Thus this scheme has efficiency improvements, simplicity improvements, and expressiveness improvements over the currently proposed schemes. With a correct fee distribution amongst the validator set, this solves the existing problem where one could withhold their signature for risk-free gain.