F1 Fee Distribution Draft-01

Dev Ojha

December 6, 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

1.1 Context

In a proof of stake blockchain, each validator has an associated stake. Transaction fees get rewarded to validators based on the incentive scheme of the underlying proof of stake model. The fee distribution problem occurs in proof of stake blockchains supporting delegation, as there is a need to distribute a validator's fee rewards to its delegators. The trivial solution of just giving the rewards to each delegator every block is too expensive to perform on-chain. So instead fee distribution algorithms have delegators perform an explicit withdraw transaction, which when performed yields the same total amount of fees as if they had received them at every block.

This details F1, an approximation-free, slash-tolerant fee distribution algorithm which allows validator commission-rates, inflation rates, and fee proportions, which can all efficiently change per validator, every block. The algorithm requires iterating over the validators every block, and withdraws require iterating over all of the corresponding validator's slashes whilst the delegator was bonded. The former iteration is cheap, due to staking logic already requiring iteration over all validators, which causes the expensive state-reads to be cached.

The number of slashes is expected to be 0 or 1 for most validators, so the latter term also meets the blockchain's efficiency needs.

The key point of how F1 works is that it tracks how much rewards a delegator with 1 stake for a given validator would be entitled to if it had bonded at block 0 until the latest block. When a delegator bonds at block b, the amount of rewards a delegator with 1 stake would have if bonded at block 0 until block b is also persisted to state. When the delegator withdraws, they receive the difference of these two values. Since rewards are distributed according to stake-weighting, this amount of rewards can be scaled by the amount of stake a delegator had. Section 1.2 describes this in more detail, with an argument for it being approximation free. Section 2 details how to adapt this algorithm to handle commission rates, slashing, and inflation.

1.2 Base algorithm

In this section, we show that the F1 base algorithm gives each delegator rewards identical to that which they'd receive in the naive and correct fee distribution algorithm that iterated over all delegators every block.

Even distribution of a validators rewards amongst its validators weighted by stake means the following: Suppose a delegator delegates x stake to a validator v 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 are

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

Note that s_i does not change every block, it only changes if the validator gets slashed, or if any delegator alters the amount they have delegated. We'll relegate handling of slashes to subsection 2.2, and only consider the case with no slashing here. We can change the iteration from being over every block, to instead being over the set of blocks between two changes in validator v's total stake. Let each of these set of blocks be called a period. A new period begins every time that validator's total stake changes. Let the total amount of stake for the validator in period p be n_p. Let T_p be the total fees that validator v accrued in period p. 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 which begins when the validator first bonds. 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: $\sum_{i=0}^{k} \frac{T_i}{n_i}$. Let the index of the current period be f. When a delegator wants to delegate or withdraw their reward, they first create a new entry in state to end the current period. Then this entry is created using the previous entry as follows:

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

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 ended by bonding. (They start receiving rewards for period $k+1$) The reward they should receive when withdrawing is:

$$x \sum_{i=k+1}^{f} \frac{T_i}{n_i} = x \left(\left(\sum_{i=0}^{f} \frac{T_i}{n_i} \right) - \left(\sum_{i=0}^{k} \frac{T_i}{n_i} \right) \right) = x \left(Entry_f - Entry_k \right)$$

It is clear from the equations that this payout mechanism maintains correctness, and requires no iterations. It just needed the two state reads for these entries.

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, and is then reset to 0.

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 it is possible to 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's account. This allows for efficient updates to a validator's commission rate every block if desired.

2.2 Slashing

Slashing is distinct from withdrawals, since it lowers the stake of all of the delegator's by a fixed percentage. Since no one 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 honest withdrawer must pay for due to the gas.

2.3 Inflation

Inflation is the idea that we want every staked coin to create more staking tokens as time progresses. The purpose being to drive down the relative worth of unstaked tokens. Each block, every staked token should produce x staking tokens as inflation, where x is calculated from a function $inflation$ which takes state and the block information as input. Let x_i represent the evaluation of $inflation$ in the ith block. The goal of this section is to auto-bond inflation in the fee distribution model without iteration. This is done by preserving the invariant that every state entry contains the rewards one would have if they had bonded one stake at genesis until that corresponding block.

In state a variable should be kept for the number of tokens one would have now due to inflation, given that they bonded one token at genesis. This is $\prod_0^{now}(1 + x_i)$. Each period now stores this total inflation product along with what it already stores per-period.

Let R_i be the fee rewards in block i, and n_i be the total amount bonded to that validator in that block. The correct amount of rewards which 1 token at genesis should have now is:

$$Reward(now) = \sum_{i=0}^{now} \left(\prod_{j=0}^{i} 1 + x_j \right) * \frac{R_i}{n_i}$$

The term in the sum is the amount of stake one stake becomes due to inflation, multiplied by the amount of fees per stake.

Now we cast this into the period frame of view. Recall that we build the rewards by creating a state entry for the rewards of the previous period, and keeping track of the rewards within this period. Thus we first define the correct amount of rewards for each successive period, proving correctness of this via induction. We then show that the state entry that gets efficiently built up block by block is equal to this value for the latest period.

Let $start, end$ denote the start/end of a period.

Suppose that $\forall f > 0$, $Reward(end(f))$ is correctly constructed as

$$Reward(end(f)) = Reward(end(f-1)) + \sum_{i=start(f)}^{end(f)} \left(\prod_{j=0}^{i} 1 + x_j \right) \frac{R_i}{n_i}$$

and that for $f = 0$, $Reward(end(0)) = 0$. (With period 1 being defined as the period that has the first bond into it) It must be shown that assuming the supposition $\forall f \leq f_0$,

$$Reward(end(f_0+1)) = Reward(end(f_0)) + \sum_{i=start(f_0+1)}^{end(f_0+1)} \left(\prod_{j=0}^{i} 1 + x_j \right) \frac{R_i}{n_i}$$

Using the definition of $Reward$, it follows that:

$$\sum_{i=0}^{end(f_0+1)} \left(\prod_{j=0}^{i} 1 + x_j \right) * \frac{R_i}{n_i} = \sum_{i=0}^{end(f_0)} \left(\prod_{j=0}^{i} 1 + x_j \right) * \frac{R_i}{n_i} + \sum_{i=start(f_0+1)}^{end(f_0+1)} \left(\prod_{j=0}^{i} 1 + x_j \right) \frac{R_i}{n_i}$$

Since the first summation on the right hand side is $Reward(end(f_0))$, the supposition is proven true. Consequently, the reward for just period f adjusted for the amount of inflation 1 token at genesis would produce, is:

$$\sum_{i=start(f)}^{end(f)} \left(\prod_{j=0}^{i} 1 + x_j \right) \frac{R_i}{n_i}$$

TODO: make this proof + pre-amble less verbose, and just wrap up into a lemma. Maybe just leave this proof or the last part to the reader, since it easily follows from summation bounds.

Now note that

$$\sum_{i=start(f)}^{end(f)} \left(\prod_{j=0}^{i} 1 + x_j \right) \frac{R_i}{n_i} = \left(\prod_{j=0}^{end(f-1)} 1 + x_j \right) \sum_{i=start(f)}^{end(f)} \left(\prod_{j=start(f)}^{i} 1 + x_j \right) \frac{R_i}{n_i}$$

By definition of period, and inflation being applied every block, $n_i = n_{start(f)} \left(\prod_{j=start(f)}^{i} 1 + x_j \right)$. This cancels out the product in the summation, therefore

$$\sum_{i=start(f)}^{end(f)} \left(\prod_{j=0}^{i} 1 + x_j \right) \frac{R_i}{n_i} = \left(\prod_{j=0}^{end(f-1)} 1 + x_j \right) \frac{\sum_{i=start(f)}^{end(f)} R_i}{n_{start(f)}}$$

Thus every block, each validator just has to add the total amount of fees (The R_i term) that goes to delegates to some per-period term. When creating a new period, $n_{start(f)}$ can be cached in state, and the product is already stored in the previous periods state entry. You then get the next period's $n_{start(f)}$ from the current tm-power entry. This is thus extremely efficient per block.

When withdrawing, you take the difference as before, which yields the amount of rewards you would have obtained with $(\prod_0^{begin\ bonding\ period} 1+x)$ stake from the block you began bonding at until now. $(\prod_0^{begin\ bonding\ period} 1+x)$ is known, since its included in the state entry for when you bonded. You then divide the entitled fees by $(\prod_0^{begin\ bonding\ period} 1+x)$ to normalize it to being the amount of rewards you're entitled to from 1 stake at that block to now. Then as before, you multiply by the amount of stake you had initially bonded. TODO: (Does the difference equating to that make sense, or should it be shown explicitly)

Note that the inflation function could vary per block, and per validator if ever a need rose. If the inflation rate is the same for everyone then there can be a single global store for the entries corresponding to the product of inflations. Inflation creation can trivially be epoched as long as inflation isn't required within the epoch, through changes to the $inflation$ function.

Again note that this process is extremely efficient.

2.4 Withdrawing with no iteration over slashes

TODO: Fill this out. Core idea: you use the same mechanism as previously, but you just make that blocks x_j term negative. (So a 20% slash would be equivalent to an inflation on that validator of -20%) This foregoes the constant inflation per validator, may or may not be worth it depending on expected number of slashes

2.5 Auto bonding fees

TODO: Fill this out. Core idea: you use the same mechanism as previously, but you just don't take that optimization with n_i and the n_{start} relation. Fairly simple to do.

2.6 Delegation updates

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

2.7 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.

5 TO DOs

- A global fee pool can be described.
- Determine if auto-bonding fees is compatible with inflation and comission rates in conjunction
- mention storage optimization in the uniform inflation and iteration over slashing case: only have one storage location w/ refcount of the product of inflation results
- Remove iteration over slashes by the same normalization trick used in inflation
- Add equation numbers
- perhaps re-organize so that the no iteration