This post further explores Merkle Mountain Ranges (MMRs) in the context of stateless clients. Previous posts:

1. History, state, and asynchronous accumulators in the stateless model
2. A cryptoeconomic accumulator for state-minimised contracts

TLDR

: We introduce batching and cyclic partitioning of history objects, called “logs” for short. The end result is a log accumulator significantly improved over the (already remarkable!) plain MMR.

A multi-MMR (3MR—short for MMMR) is a set of MMRs that will serve for log accumulation. Each shard is given one 3MR formed of 2^n

MMRs, labelled 0, 1, ..., 2^n-1

. Given a collation with height i

, log accumulation is done as follows:

- Batching

: Create a “log batch” by assembling all the logs produced in the collation in a Merkle tree where the leaves are individual log hashes.

- Cyclic partitioning

: Accumulate the log batch root in the 3MR by pushing it to the MMR labelled i

modulo 2^n

.

Given a log l

, the corresponding witness w

is the concatenation of:

- the Merkle path from the hash of l

to the log batch root

- the 3MR witness of the log batch root

Improvements over plain MMR log accumulation:

1. Deterministic witness update events

: Thanks to batching, the 3MR progresses by exactly one “tick” per collation, so the witness for a log in collation i

will get updated at precise future collation heights dependent on i

only. This means users no longer need to constantly monitor collations for possible witness update events.

1. Reduced witness update events

: Thanks to partitioning, individual MMRs get a log batch root 2^n

times slower than otherwise. This buys us two things. (For concreteness, we set a collation interval of 8 seconds and the 3MR size parameter n = 15

.) * No need for witness auto-updating

: In the worst case, the smallest witness update event gap is 2^n*8

seconds (> 3 days). This means the benefits of short-term witness auto-updating are almost nullified. For validators, this saves CPU and RAM otherwise spent on witness auto-updating. Even more crucially, it saves bandwidth and latency in shard synchronisation because the witness updates in the auto-update window do not need to be downloaded.

- Ultra-low frequency witness updates

: For illustration below are the worst case number of witness update events over different time frames after log creation. (I think the average case is about half the worst case.) * 1 day: 0 update events

- 1 week: 1 update event
- 1 month: 3 update events
- 1 year: 7 update events
- 16 years: 11 update events
- 1 day: 0 update events
- 1 week: 1 update event
- 1 month: 3 update events
- 1 year: 7 update events
- 16 years: 11 update events
- No need for witness auto-updating

: In the worst case, the smallest witness update event gap is 2^n*8

seconds (> 3 days). This means the benefits of short-term witness auto-updating are almost nullified. For validators, this saves CPU and RAM otherwise spent on witness auto-updating. Even more crucially, it saves bandwidth and latency in shard synchronisation because the witness updates in the auto-update window do not need to be downloaded.

1. Ultra-low frequency witness updates

: For illustration below are the worst case number of witness update events over different time frames after log creation. (I think the average case is about half the worst case.) * 1 day: 0 update events

- 1 week: 1 update event
- 1 month: 3 update events
- 1 year: 7 update events
- 16 years: 11 update events
- 1 day: 0 update events
- 1 week: 1 update event
- 1 month: 3 update events
- 1 year: 7 update events
- 16 years: 11 update events
- Cheap historical data availability

: In order to update a witness for any

given log it suffices to have the log batch root of every collation since genesis. By putting the log batch root in the collation header, every SPV node for a given shard now becomes a full data availability provider for log witness updates.

After operating for 32 years the constituent MMRs of the 3MR will have grown to have \lceil\log(\frac{32*365.25*24*60*60}{8*2^n})\rceil = 12

mountain roots. So each shard will have 2^n*12*32

bytes (exactly 6MB) of 3MR overhead.