The goal of this post is to propose a system that would live on the eth1 chain and provides incentives for validators to include their preferred data in eth2 blocks. This would allow sequencers and other users that want data published on eth2 to be able to do so in real-time without needing to run a large number of validators themselves.

The mechanism will require two contracts on the eth1 chain:

- A light client of eth2 (which we already plan to have), which would create a registry of (at least hashes of) eth1 block headers on the eth2 chain
- A fee market manager contract (FMMC)

The purpose of the FMMC will be to mediate a market where senders can make bids to have particular blobs of data (represented by their commitments) included in the eth2 chain as shard block contents.

Desired properties

- Cryptoeconomically secure (you pay if and only if your data is included in the desired position)
- Bids are costless; on-chain gas costs are only needed to process transfers that result from successful bids
- Maximally gas-efficient

Parameters

Parameter	Value	Description
SHARD_COUNT	64	Number of shards
DISCOVERY_DELAY_BOUND	1024	Approximate bound on the number of slots that it takes before a beacon chain block is included into the shard chain

Mechanism: Bidding

In order to become a bidder, users must register an account in the FMMC, and deposit some quantity deposit_amount

of ETH. They must also register which shard or shards they plan to be bidding in (bidding_shards

); this would be a subset of the range [0, SHARD_COUNT-1]

. Users can change which shards they are eligible on but this requires an on-chain transaction.

A bid is an off-chain message of the form [[shard_id, slot, commitment, fee], signature]

. A bid represents a cryptoeconomically binding request to have a block with the given commitment

included in the shard chain, and being willing to pay the provided fee

for it.

The maximum allowed fee is cap = deposit // DISCOVERY_DELAY_BOUND // len(bidding_shards)

; this is enforced in the contract.

Mechanism: Proposing and fee collection

The bids are broadcasted together with the shard data contents that correspond to the commitments in a dedicated p2p network (eg. one subnet per shard). Proposers accept the valid bid which maximizes their profit from including the block, ie. fee - length_of_data * basefee

, where basefee

is an EIP-1559-style per-byte basefee. They include that block in that shard.

The FMMC has a function that accepts the following arguments:

- The ShardBlockCommitments

section of a beacon block, which contains all of the commitments to shard data that were included in that block

- A Merkle proof linking this section to the beacon block header
- A list of bids
- Signatures for each bid

The function validates the inputs; if the inputs are all valid, it processes the bids, removing the bid amounts from the bidders' deposits and sending the amounts to the address specified in the beacon block's graffiti.

Gas usage analysis

Gas usage is optimized if the FMMC function is called once per shard block. To make sure this happens we can add incentives that encourage a specific actor to send the transaction and reward them, perhaps even taking out a percentage of the fees in order to do this.

If this goal is satisfied, then the theoretical maximum gas cost is the cost of one call per beacon block, which contains:

Item	Max cost	Explanation
Shard block commitments	49152	16 (gas per byte) * 48 (commitment count) * 64 (SHARD_COUNT)
Merkle proof	2048	Approx. 128 byte long proof (assuming ~16 objects in block body)
Bids	5120	16 (gas per byte) * [1 (shard ID) + 4 (fee)] * 64 (SHARD_COUNT). Commitments already included above, slot is the same for all bids and already provided as an argument
Signature data	66560	64 signatures * 65 bytes * 16 gas per byte
Signature verification	204800	SHARD_COUNT * (100 (hashing) + 3100 (ECRECOVER))

The total is 327680 gas. Note that the bulk of the gas costs is per-signature; if not all shards are taken, then the gas costs are significantly reduced.

BLS aggregate signatures were not used because hash-to-curve is still too expensive, even more so than the ~4000 gas per ECDSA signature that we are using here.

Bid guarantee analysis

During each slot, a depositor is capable of publishing at most bidding_shards

bids. There are at most DISCOVERY_DELAY_BOUND slots (by assumption) between the current slot and the time that bids for the current slot will be processed. Hence, the account's balance can decrease by at most len(bidding_shards) * DISCOVERY_DELAY_BOUND

times the cap

. The fee cap was set such that deposit_amount = cap * len(bidding_shards) * DISCOVERY_DELAY_BOUND

, so the deposit is guaranteed to be large enough to be able to pay up.

Philosophically, note that this fee market scheme avoids the impossibility of using this kind of technique for channels because there is only a limited number of valid payees: the block proposer for each shard in each slot.

Extension: probabilistic payments

On-chain randomness can be used to determine for each slot exactly one shard whose bid may be paid. In this scheme, bidding would entail only a 1/64 probability of having to pay up. This implies that in equilibrium stated bids would be close to 64x higher, so economically this would make little difference.

However, this would greatly reduce on-chain costs: all of the above costs would decrease by 64x except a single Merkle proof for a single shard block commitment would need to be added:

Item	Max cost	Explanation
Shard block commitment	768	16 (gas per byte) * 48 (commitment count)
Merkle proof to commitment list	2048	Approx. 128 byte long proof (assuming ~16 objects in block body)
Merkle proof to commitment	3072	192 byte long proof (32 * log(64))
Bid	80	16 (gas per byte) * [1 (shard ID) + 4 (fee)]
Signature data	1040	65 bytes * 16 gas per byte
Signature verification	3200	(100 (hashing) + 3100 (ECRECOVER))

Total: 10208 (plus base tx sending costs)

The deposits required would need to be increased. Most conservatively, if we want to guarantee payment, the deposit would need to be high enough to cover the cap

in every slot, just in case the user gets unlucky and the system selects the shard that they sent a successful bid in during every slot. However, we can get much more capital efficiency if we are willing to accept a probabilistic guarantee, eg. 10 standard deviations: set required_deposit = cap * (len(bidding_shards) * DISCOVERY_DELAY_BOUND + sqrt(len(bidding_shards) * DISCOVERY_DELAY_BOUND) * 10

.

If we want gas costs to be even lower, we can use on-chain randomness to select eg. 1 of 10 headers that would need to be included, making the probability that a user needs to pay up only 1 in 640; an optimal tradeoff between risk/capital efficiency and gas efficiency can be found.