TL;DR:

Proposal for a data shard block header that makes it easy for applications to share shard block space. This format makes it easy for an application to check if a data shard block contains a message for a given application, by only checking a very small number of chunks in the header of the block (logarithmic in the total number of applications which have data in the block).

Background

After the merge, Ethereum 2 will add data shards

to allow Ethereum to scale its total transaction throughput via rollups. Rollups come in two flavours:

- ZK Rollups

use computation integrity proofs to verify that execution of transactions has been performed correctly

- Optimistic Rollups

use fraud proofs that can challenge any incorrect execution, thus ensuring eventual consistency

Both flavours have in common that only a small amount of computation is needed on the execution chain, but a much larger data layer is required to verify that all data was published and made available to all full nodes.

In Eth2, there will be (initially) 64 shards, which will be contributing one shard blob at each slot. A shard blob consists of up to 2^{14}

prime field elements, where the field is determined by theBLS12_381 curve order q= 0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001

. The target block size is half this, so 2^{13}

prime field elements. Shard blobs are committed to byKZG commitments over BLS12_381. This means it is easy to open any number of positions in a blob, no matter whether they are adjacent or not.

Why we need a shard blob format

Most rollups will want to treat data shards like an inbox: They need to process all incoming data (potentially limited to some of the shard ids to limit the amount of data they need to process) and take actions on any data that is destined for them. This could be:

- Transaction data, the effect of which needs to be computed to determine the new state
- “Blocks”, which would be transaction data bundled with the effect on the state (state root update, potentially witnesses);
- Garbage data – since anyone can post anything to the data shards, and there is no validation, a rollup always needs to be able to process (and reject) any amount of garbage that is thrown at it.

Clearly the rollup construction will determine which of the mentioned formats the data will have. However, it will always need to be able to process the last point – garbage data. Anyone can try and spam one rollup if they are willing to pay to get the data included in shard data blobs.

Now the reason we need a shard data blob data format is that we want several rollups to share one blob. For example, if two rollups only have transactions for half a blob, it would be wasteful if they each had to get one full slot instead of sharing one. In addition, to minimize the power of block producers, it is a great advantage if very small bundles can get included. We thus want a format that allows several application data bundles to coexist in one blob.

Data format requirements

The shard blob data format is not enshrined at consensus layer. It’s purely a standard to allow applications to coexist nicely, and only interfere with each others operations minimally in the happy case. It is not a spam/DoS protection: Any application has to be able to process all incoming data. The only thing we are minimizing here is the amount of data that is required to verify that one has got the data destined for an application, and all of it. The goals are thus:

- Simplicity of the format – it should be easy to parse
- Only a small number of positions needs to be revealed to determine where to find an applications’ data, if any

- The header has to be “always valid” – there can’t be some part of the header that will make it invalid for applications that see this part of the header, but valid for others that don’t see it. We need a deterministic reveal protocol that allows querying the position of one applications data and
- Applications playing nicely should not cause any overhead for other applications (we can’t do much about those applications that don’t play nicely; so you can for example just pick the same application as another app. Serious apps however won’t do that as it will also be bad for them (need to process more data))

Shard blob data format

The suggested format is a header that uses n\leq 256

chunks

(total header length). Note that one chunk here is one field element, which can contain up to 31 full bytes; it is different from data availability chunks, which are composed of several field elements). Each field element will contain 31 bytes and all integers are encoded little endian. We use three integer types:

- int1

: 1 byte integer

- int2

: 2 byte integer

- int3

: 3 byte integer

Further we define

application_id ::= int3

First chunk – meta-header and first five applications

field name

type

description

version

int1

Version of header format, currently set to 0

length

int1

Total length of header in chunks, not including this chunk

multiplier

int1

Application start value multiplier

app_1_id

application_id

Application 1 ID

app_1_start

int2

Start chunk of application 1 data

app_2_id	application_id	Application 2 ID
app_2_start	int2	Start chunk of application 2 data
app_3_id	application_id	Application 3 ID
app_3_start	int2	Start chunk of application 3 data
app_4_id	application_id	Application 4 ID
app_4_start	int2	Start chunk of application 4 data
app_5_id	application_id	Application 5 ID
app_5_start	int2	Start chunk of application 5 data

Total = 1+1+1+5 * (3 + 2) = 28 bytes

Second and consecutive chunks – six applications each

field name	type	description
app_1_id	application_id	Application 1 ID
app_1_start	int2	Start chunk of application 1 data
app_2_id	application_id	

Application 2 ID

app_2_start

int2

Start chunk of application 2 data

app_3_id

application_id

Application 3 ID

app_3_start

int2

Start chunk of application 3 data

app_4_id

application_id

Application 4 ID

app_4_start

int2

Start chunk of application 4 data

app_5_id

application_id

Application 5 ID

app_5_start

int2

Start chunk of application 5 data

app_6_id

application_id

Application 5 ID

app_6_start

int2

Start chunk of application 5 data

Total = 6 * (3 + 2) = 30 bytes

Each entry for an application consists of an application ID and a start chunk. The start chunk has to be multiplied with 2**multiplier

to get the start chunk for the application. multiplier

can simply be set to 0

at the moment as shard blobs have fewer than 2^{16}

chunks; the multiplier allows to keep using the same format as shard blobs are extended to larger sizes, at the cost of some granularity. So the data for the application identified by app_1_id

starts at app_1_start * 2**multiplier

, the data for the application identified by app_2_id starts at app_2_start * 2**multiplier

, and so on. Applications can either have an internal way of determining the length of their data, or they can use the start of the next application id to determine the length of the data for the application, however they might have to be able to deal with padding in this case.

Generating the header

An honest proposer would collect the data blocks from applications, and sort them in ascending order by application ID. Then they would generate the header.

Application ID 0 is interpreted as an empty entry and can be inserted anywhere by the block producers; it might be required for padding. Any application ID 0 fields should be ignored by clients.

Interpreting the header

Since the header is sorted, the client can find the correct chunk that contains the relevant application data using a binary search algorithm. It starts in the middle of the chunks; if the application id it is looking for is smaller than the first application id in the chunk, it continues searching in the first half of the chunks; if it is larger then the last application ID in the chunk, it continues in the second half of chunks etc.

We cannot trust that the block builder is sorting applications according to the application ID. Since the header is not validated by anyone, we cannot rely on anything. Instead, we will simply specify the exact binary search algorithm that is used to find an application in the header. We will then define the following outcome:

- If the binary search algorithm, as defined below, encounters an error (e.g. a chunk where the first application ID > the last application ID) or does not find the application ID, then the blob does not contain any data for the application

This has two consequences:

- One has to follow the binary search algorithm to find an application’s data. If we instead just let someone reveal to use the one chunk that contains application ID and its start chunk, it may be that the binary search would not have found this data and thus a different user would believe there is no data in this chunk.
- One can never reject a full header as invalid just because some part of it is invalid. Some application IDs may only encounter valid parts of the header while looking for this application, and the header will be fully valid in their view. Invalid parts of the header only lead to a zero output (no data for this application) if they are encountered during the binary search

Binary search algorithm

In pseudocode

function find_application_id(app_id): left = -1 right = length + 1 current = left + (right - left) // 2 while left < current < right: if chunk[current].app_1_id >= app_6_id: return AppIdNotFound if chunk[current].app_1_id <= app_id <= chunk[current].app_1_id: // Iterate through app_1_id to app_6_id in chunk[current] if (found app_id in app_*#id): return app#*_start * 2**multiplier else: return AppIdNotFound if chunk[current].app_1_id < app_id: right = current current = left + (right - left) // 2 if chunk[current].app_6_id > app_id: left = current current = left + (right - left) // 2 return AppIdNotFound

As can be seen from this function, it will never return an error; it can only return that the app ID was not found, and due to the binary search construction, will do so by accessing at most 9 chunks (meta-header + 8 queries).

Who needs to process the header and when

If we assume all blocks are maximally fragmented, then processing the header takes up to 9 chunks per shard blob. This means an application wanting to check all shard blobs for relevant data would need 9*64 field elements or 18,432 bytes. This would clearly be quite significant if all of these had to be posted to the execution layer.

However, I will argue that a rollup will (almost) never have to analyze shard headers on chain:

- For ZK Rollups, this is obvious, as all the proofs would be inside the computational integrity proof.
- For optimistic rollups, it would not be necessary to provide a proof to the data unless an invalid state transition happens. For example, if the sequencer has left out a data block that’s destined for the application, a fraud proof has to give a proof to only that data.

In this context, 18 kB seem ok. This quantity is mostly relevant for sequencers and anyone wanting to follow a rollup; in both cases, this is less than beacon chain data and the actual typical rollup data per beacon block, so the overhead is not high even in the worst case.

Application IDs

An application ID is a 24-bit integer. Application IDs should be reserved to avoid collisions. Of course nothing prevents an application to intentionally use another application's ID; use of this format is voluntary. However, we can minimize this for all applications that play nicely, and it's likely that any application wanting to get significant adoption is better of by playing nicely, which will also make it more efficient.

One idea is to deploy an Eth1 smart contract that allows anyone to burn 1 ETH to get assigned a random application ID. For serious projects that would be a small price to pay to avoid undesirable collisions with other projects.