Author: Csaba Kiraly, in collaboration with Leonardo Bautista-Gomez and Dmitriy Ryajov, from theCodex.storage research team.

Note: this document describes the current state of our thinking, result of a collaborative effort and numerous discussions with other teams. It would not had been possible without the contribution and ideas of @dankrad , @djrtwo , @fradamt , @AgeManning , @Nashatyrev , @matt , @pop , and @Echo .

TL;DR

- Danksharding was planned for 32MB blocks, but our current networking stack

can't handle that, becoming the bottleneck

. With HW accelerated KZG on the horizon for the block encoding, our networking stack will have to scale even more.

- DAS encompasses two different concepts: Data Availability

achieved by dispersal to custody, and Sampling

from custody. We can use this distinction to our advantage, designing an efficient dispersal, and an efficient sampling protocol.

- liteDAS

is our sampling protocol, designed to provide low-latency, bandwidth efficient, and robust sampling.

- Dispersal

can be done with protocols similar to GossipSub, but changes are required.

- With the combination of deterministic custody assignments and Topic Routing

, we can find peers fast enough for dispersal and for sampling.

- To enable sampling, we should also enable Ephemeral Connect

, not supported by the current stack.

- 2D encoding

(or some other locally repairable code) is required for in-network-repair, the key to availability amplification

.

Introduction

The original Danksharding proposal targeted 32 MB blocks, and this size was mainly chosen due to compute constraints for the KZG commitments used in the DAS data structure. Since then, there have been various iterations on the network design, aiming to efficiently make the block data available in a P2P structure, and to let nodes sample from this structure. However, none of these constructs convincingly support blocks of 32MB, making the performance of the networking solution the real bottleneck. In this post we look at these networking issues and propose solutions, with the aim of making 32 MB (and possibly beyond) achievable.

DAS: Data Availability vs. Sampling

The goal of DAS (Data Availability Sampling) is to ascertain, with high probability, that a given block of data was made available to anyone interested, and to do this without requiring any single node in the system to hold - or even to temporarily receive - the whole block of data.

That is, we want to keep the bandwidth requirements of individual nodes at the levels of the current (after EIP-4844) Ethereum network, while handling orders of magnitude larger blocks. From the networking perspective, this is an important constraint on the Ethereum DAS design.

For our discussion on networking aspects, it is important to emphasize that there are two distinct parts to DAS: the "making data available" part, and the "sampling" part.

Making Data Available

First, the data has to be made available. Availability means that nodes in the system get "enough" data to be able together to reconstruct the original block. Thus, availability is a system level property

: data is either available to the whole system, or not available. At least, this is what we want to achieve.

As in P2P systems in general, the amount of data should better not be “just enough”, but it should be overwhelmingly enough

, meaning the data can be reconstructed even if there is churn, network partitioning, or a large portion of malicious nodes. Only at this point we can say that the data was made available.

In other words, we should design a protocol that makes sure that there are no borderline situations: the data is either not available (wasn’t released by the source), or it is overwhelmingly available. As we will see later, it is the interplay between erasure coding and a robust and largely redundant P2P networking structure that makes this possible.

Sampling

Second, there is sampling. Sampling is an individual node’s view of what was made available to the system. It is a single node convincing itself that the data was made available, and the technique it uses for this is to retrieve a few pieces of the block. The sample

is the (typically random) selection of the pieces to retrieve, while sampling

is the retrieval of these pieces. If this sampling is successful, and with a few independence assumptions

, the node can convince itself that the data was indeed made available.

Importantly, in the background, behind theprobabilistic calculations, there are those nasty independence assumptions. Essentially it is assumed that data is (was, or will be) released independent of what sample was selected. This unfortunately can be gamed, and the more the block producer and its “accomplices” knows about the sample selection of a node, the easier it is to introduce correlation and thus fool someone. Thus, sample selection

and limited disclosure of the selection

is an important part of the security guarantees.

Note that in this writeup we call the pieces of the block segments

, but they also go by the name “column” when 1D encoding is used, “cell” when 2D encoding is used, or sometimes the confusing term “sample” is used both the segment as well as for the list of selected segments.

Also note that sampling is not the only technique we could use to “convince ourselves” that the data was available. Without trying to list all possibilities, we could use succinct proofs, trusted friends, etc. Sampling, however, is trustless and spreads segments of data to all nodes in the system, eventually enabling an extra layer of redundancy and thus recovery.

Interest: Custody vs. Sample

Both when data is made available, and when nodes are sampling, segments of the block get delivered to nodes. Which segments get to which node, however, is driven by different objectives in these two phases.

We call the segments that should be delivered to a node it’s interest

. We have two types of interest in the system. They are similar, but serve different purposes and have some fundamental differences:

- custody

: segments getting to nodes as part of making the data available are taken into custody and used to serve sampling. That is, custody has a double goal: providing availability and at the same time serving as a sampling infrastructure.

- sample

: sampling is for the individual node to convince itself about availability with very high probability by checking that enough segments are in custody.

Interest can change over time

, for example from epoch to epoch for the sample selection, or with some other time granularity for custody. Changing it or keeping it fixed has both security and network efficiency implications.

DAS Phases: Dispersal vs. Sampling

The fundamental differences between the requirements of data availability and sampling, and between the properties of custody and sample, are also reflected in the network design. We can differentiate between two phases of segment distribution:

- dispersal

: in which segments of the block are distributed in the P2P network to provide overwhelming availability and custody.

- sampling

: in which nodes collect a random sample of segments from custody.

The two phases can use different P2P network constructs. In what follows we focus on these network constructs, deriving fast, robust, and bandwidth efficient protocols both for dispersal and for sampling.

FullDAS networking

[

FullDAS - LossyDAS - liteDAS - Data Availability Sampling Components

960×720 80 KB

](https://ethresear.ch/uploads/default/original/3X/9/3/93a404791201f9e15e8d268a971c5fd54d120d7c.jpeg)

In the original design, GossipSub was planned to be used for dispersal, with many (512 for columns and 512 for rows) topics, distributing columns and rows into custody at all the staked nodes (beacon nodes with validators). For sampling, instead, a separate DHT was planned to be set up from all full nodes (staked and non-staked), serving all the sampling queries. This presented several challenges, as we have highlighted in our related post and paper.

As a consequence, in PeerDAS

and in SubnetDAS

we have tweaked the design to provide intermediate solutions, with compromises both in functionality and in scalability.

In PeerDAS, we have modified custody assignment and dispersal to be based on the NodeID, making all full nodes (both staked and non-staked) be part of the custody system. Sampling is then made from this P2P structure using a Request/Response (Req/Resp) protocol from existing peers of a node, adding more pressure on peer discovery (Discv5), and the number of peers that nodes have to sustain.

SubnetDAS, instead, modifies the design to use the GossipSub distribution mechanism all the way, both for dispersal and for sampling. This, again, limits our possibilities, also sacrificing the unlinkability of sampling queries.

A fast and bandwidth-efficient implementation of these proves to be challenging even for blocks of a few MBs, and this is where we arrive to our main topic: how to make DAS work for blocks of 32 MB and beyond?

In what follows, we introduce the important parts of the stack:

1. liteDAS: fast and efficient sampling from custody
2. Finding custody peers fast:
3. Custody allocation and sample selection
4. Topic Routing
5. Ephemeral Connect
6. Custody allocation and sample selection
7. Topic Routing
8. Ephemeral Connect
9. Making dispersal more efficient
10. Availability amplification with 2D encoding

liteDAS: The Sampling Protocol

Sampling, at an abstract level, sounds very simple:

- take a random sample of block segment IDs

- ask for these from nodes custodying it
- wait for all the responses
- declare success if received all, otherwise declare failure

In reality, however, things get much more complicated. We have dealt with some of the complication related to sample selection in our post on LossyDAS, IncrementalDAS, and DiDAS, but we have left out all the networking aspects from that post.

To design a sampling protocol, besides the draft outline above, we should also answer questions like:

- when should a node start sampling?
- when should sampling end, what should be the timeout?
- if we know several nodes custodying the same segment, which one to ask?
- what to do if we do not know a node that is supposed to custody a segment?
- what to do when sampling fails?
- etc.

Moreover, we should design a protocol that is fast, robust, and bandwidth efficient.

For sampling, we propose liteDAS

, a new Req/Resp protocol that aims to provide close to minimum sampling latency with close to minimal bandwidth usage, avoiding excessive resource utilization both on the happy path and when there are issues. The key observations behind liteDAS are:

- There is no “perfect time” to ask for a segment, since we don’t know when dispersal will make these arrive to custody. It is better to ask early and wait.
- Sampling nodes have some time (the dispersal time) to prepare for sampling. We can use this time to make sampling efficient.
- We can hide our sample better if we ask for segments one-by-one, asking each from distinct nodes.
- Having a fast and efficient way of finding custody nodes for a given segment is essential.

The messaging primitive

we propose is a request with an explicit timeout

. This simple primitive allows us to start sampling early (at slot start), and build a nice dynamic behavior with an initial period to handle the happy path, and a second period to handle potential issues, as outlined below:

- At T_0

(slot start, or when public parameters are known) * decide sample (list of segment IDs) using local randomness * eventually consider using DiDAS and/or LossyDAS

- eventually consider using DiDAS and/or LossyDAS

- For each ID:
- select list of candidate peers who might custody given ID
- select P=1

of these candidate nodes (eventually ranking), and send request with long timeout (e.g. 4s). P

is the “parallelism” parameter.

- select list of candidate peers who might custody given ID
- select P=1

of these candidate nodes (eventually ranking), and send request with long timeout (e.g. 4s). P

is the “parallelism” parameter.

- decide sample (list of segment IDs) using local randomness
- eventually consider using DiDAS and/or LossyDAS
- eventually consider using DiDAS and/or LossyDAS
- For each ID:
- select list of candidate peers who might custody given ID
- select P=1

of these candidate nodes (eventually ranking), and send request with long timeout (e.g. 4s). P

is the “parallelism” parameter.

- select list of candidate peers who might custody given ID
- select P=1

of these candidate nodes (eventually ranking), and send request with long timeout (e.g. 4s). P

is the “parallelism” parameter.

- at T_1=4s
- for every ID still missing
- select next candidate node and send request with short timeout
- use timeouts and the number of outstanding queries to avoid traffic explosion
- eventually search for new candidates (this can also be done proactive, before T_1

)

- select next candidate node and send request with short timeout
- use timeouts and the number of outstanding queries to avoid traffic explosion
- eventually search for new candidates (this can also be done proactive, before T_1

)

- Optionally, extend sample according to IncrementalDAS with some period
- for every ID still missing
- select next candidate node and send request with short timeout
- use timeouts and the number of outstanding queries to avoid traffic explosion
- eventually search for new candidates (this can also be done proactive, before T_1

)

- select next candidate node and send request with short timeout
- use timeouts and the number of outstanding queries to avoid traffic explosion
- eventually search for new candidates (this can also be done proactive, before T_1

)

- Optionally, extend sample according to IncrementalDAS with some period

Intuitively, this is fast (reducing sampling latency close to minimum), bandwidth efficient (sending only one Req/Resp per segment on the happy path), and robust (explores sampling options once custody should be there).

Fast

Sending the sampling query at the start of the slot allows us to minimize the sampling delay to a single 1-hop latency on top of the dispersal delay (the delay with which the node we queried received the segment or column).

In more detail, to understand why this is fast, let's see what would be the fastest. The minimum sampling latency would be what one could achieve if the node asks, right when it knows what samples it needs, all its peers that could have the sample, i.e. all peers that plan to custody that segment. That would however mean that our node receives lots of responses, adding considerable extra load. In liteDAS we only ask P

of them at a time, as a compromise between bandwidth efficiency and latency.Our data on GossipSub latency distribution and our initial simulations show that the latency compromise is very limited on the happy path.

Bandwidth efficient

The protocol receives every sample (or column) exactly once on the happy path. On the unhappy path, we can set a compromise between following a sequential logic with short timeouts, or allowing some parallelism (and thus duplicates) by changing P

in later phases. Importantly, if the data is actually not available, it will still keep resource utilization in limits.

Robust

In the first phase, the robustness of the above sampling relies on the robustness of dispersal. Dispersal already employs a number of reliability techniques, thus, when on the happy path, it is very likely that for every ID, the single node we select will receive the sample.

If instead there are issues with dispersal, our sampling also becomes less latency sensitive: sampling should anyway not succeed until availability through custody is widespread. Thus, we have time to “hunt” for segments. After the initial phase, our protocol will cycle through potential other peers taking in custody the same segment. At that point, we can also send more queries in parallel, eventually allowing some duplicates.

Possible Extensions

Beyond the basic idea, there are a few other tweaks that can improve dynamic behaviour:

1. Nodes can anticipate the hunt for peers that could provide a given segment. These are the peers that will eventually custody that segment.
2. Since sample selection is done using local randomness, nodes could even prepare the sample (list of IDs) well in advance, anticipating the search for candidate peers with slots or even epochs.
3. To avoid traffic explosion, we recommend controlling the number of outstanding queries, together with timeouts. This practically means that we can avoid traffic explosion even in cases of an unavailable block or when there are large-scale network issues.
4. As mentioned before, asking one node at a time works, but this can be increased to e.g asking 2 or 3 in parallel. These are compromises that can even be made locally, we can't really mandate them anyway. (Actually we could, using Rate Limiting Nullifier-like techniques, but that's out of scope here and seems too complex for the scope).
5. We could also include a HAVE list in the query response message. The reason behind this is that it gives extra information on the state of diffusion that we would not have otherwise. Given the 1D or 2D structure, a have list can be compressed very well to a small bitmap, so it is not much extra bytes. Eventually, to stay generic on the message primitive, we can make it a flag in the request whether we want a HAVE list in the response.
6. Finally, it is worth mentioning that sampling nodes could use the information collected during protocol execution to feed back segments into custody. One way to make this happen is to introduce a Resp with a NACK (negative acknowledgment) style message when the timeout has passed and the segment is not in custody. This “counter-request” would mean that the sampling node should send the segment when it receives it from someone.

Finding Custody Peers Fast

liteDAS still assumes we have the peers to send requests to. What if we don't have these, and we have to search for peers taking in custody a given ID? What if we also have to go through a convoluted connection procedure before requesting the segment?

As highlighted before, we can anticipate the search for custody peers. Still, there are times when we need to search for peers with a specific ID in custody, and that should be relatively fast. This is a crucial point currently being debated in discussions around DAS. As part of the sampling process, we need to find and connect to peers in a few seconds, which seems to be one of the main limiting factors. There is a similar problem for dispersal as well when building the mesh for efficient column(/row) based distribution, although in that case timing is less critical.

The problem with the current protocol stack was summarized well by AgeManning aspart of the PeerDAS spec discussions. Getting individual segments from new peers is awfully inefficient with the current stack. Clearly, the problem has two mayor underlying aspects:

- Finding nodes is inefficient

: currently this would be done using Discv5. Since Discv5 has no topic search, it is a random walk on the DHT, enumerating peers in hope of finding a good one. We propose to make this faster by deterministic custody assignment and by what we call “topic routing”.

- Connecting to nodes is inefficient

: we intend to solve this by introducing “ephemeral connect”.

Deterministic NodeID-based Custody Assignment

First, lets see what we can do to find custody nodes for a given segment fast. The key to this is to assign custody wisely.

Since we need system-wide availability, custody has to be assigned properly, providing good coverage. A randomized local selection would provide this, but fortunately we can do better. A nice property of custody is that we do not need to hide who takes custody of what

. We can use this property to make our system more efficient by deriving custody interest from the NodeID

(eventually with some rotation scheme and using public randomness) in a deterministic way. It seems that publicly exposing the interest of nodes for the purposes of custody does not influence security. It also seems that we can expose interest publicly well ahead of time (keeping custody stable in time, similar to what we do with attestations today), and it still does not create security issues.

Note that there is a catch here: by using the NodeID, we are not binding custody to validators, as in the original proposal, but to full nodes (including non-staked nodes). Thus, we cannot really mandate custody. We currently do not want to mandate it anyway, but later we might introduce Proof Of Custody, and then we have to revisit this point. It is also relatively easy to create a large amount of new NodeIDs, potentially exposing custody to targeted sybil attacks.

By deriving custody from NodeID, we can speed up search in two ways. First, we can derive custody from the leftmost bits of the nodeID. This allows us to use the Discv5 Kademlia search instead of a random walk, since Kademlia is prefix-based. Second, if a custody rotation scheme is used (nodes changed what columns/rows they custody with some period), we can calculate this from the NodeID, without contacting the peers or getting updated ENR records from Discv5.

We already use this NodeID-based deterministic custody assignment trick in PeerDAS: it allows us to make the dispersal more efficient, and it also allows sampling peers to find custody nodes easier, just based on the NodeID and a small amount of metadata published together with the NodeID in the ENR.

Note that the same is not true for the sample, where we do care about making it relatively hard to figure out what will be sampled by a given node, as otherwise we would make it easy to attack individual nodes. Hence, we cannot derive the sample (list of segment IDs) from the NodeID, and we also better avoid sending the whole sample (again, the list of IDs, not the data) to a single node. We thus prefer nodes to request samples on-the-fly, preferably exposing interest selectively to a diverse set of peers, limiting the exposure of the individual node.

Topic Routing: efficiently finding nodes for a given column or row

Another option we have at hand is to introduce custody node search into our protocol directly, circumventing the difficulties with Discv5.

In current proposals, columns are distributed using GossipSub, with different column IDs mapping to different GossipSub topics. The column ID space is seen as a flat list, without considering any relation between topics.

However, we do not have to handle the row and column ID space as a flat ID space

. We can use almost any distance metric, and require nodes to keep a few neighbors with column/row ID close to them, enabling faster search. Some examples to explain how this might work:

- Circlular: if a node participates in column C, it has to keep at least N neighbors from the range [C-D, C+D] mod NUM_COLS

, for some selected D

and N

- Hypercube style: if a node participates in column C, it has to keep at least N neighbors from columns C+2^j mod NUM_COLS

, for j in [0..log2(NUM_COLS}-1]

- Kademlia style: if a node participates in column C, it has to keep at least N neighbors from the set of columns we get by keeping the j MSB (most significant) bits fixed, and flipping bit j+1

The first one (circular) is not really scalable, the last two have good scalability. However, if we keep the number of columns(and rows) sufficiently low, any of these will work good enough to find potential neighbors for any C. If we want to scale higher, we might want to pick the best one for our use.

Note that the topic ID space is dense, with each ID representing a topic, and the number of nodes we have is expected to be higher than the number of topics. In a typical DHT, instead, keys and node IDs are sparse in the ID space. Having a dense ID space allows us to simplify search.

It is relatively easy to extend any of the above techniques to handle both rows and columns, either by handling the column and row ID spaces as two separate ID spaces, or by defining a joint distance function.

Also note that when sampling in 2D, we look for either a node in column C, or a node in row R. Any of these will suffice, making our search faster.

Ephemeral Connect: fast connect for sample retrieval

Connection issues are mainly due to a protocol limitation, since we are operating over libP2P, which was not originally designed for fast connectivity. We make our life even harder by keeping a hard limit on connected peers, and by keeping that full in many nodes (it is known that some peers do not accept connections, and as a consequence others are overloaded, having their peer count full, and thus also not accepting new connections).

We list a few possible solutions to this problem below:

- libP2P-ephemeralConnect: we could add a new flag to the connect request indicating a connection that is timing out fast. This can be allowed-in on the receiving side above peer count limits.
- Use a new primitive outside of the libp2p framework: If we want a secure encrypted protocol, but something more efficient than a libP2P modification, we can e.g. easily reuse our Discv5 DHT primitives here, making a Discv5 request in a handshake. That employs only 2 roundtrips, and a few bytes of data. Of course, this is just an example of what could be achieved. Other custom protocols with similar latency/bandwidth characteristics can be derived as well.
- Query forwarding: If we really do not want to add another primitive, we can introduce query forwarding. A node can take one of it's peers according to Topic Routing, and send to it a forwardable query. We'll get the sample in 2 hops.

In our opinion these can solve the connectivity issues, getting us to FullDAS. However, more testing is needed.

The Dispersal Protocol

Initially, we intend to use GossipSub for dispersal for several reasons. It is a publish-subscribe protocol which maps well to the erasure coding structure, creating separate topics per column (if using 1D erasure coding), or for columns and rows (if using 2D erasure coding). It is also battle-tested, already used in Ethereum for block diffusion and attestation aggregation, showing fast and reliable operation, as discussed in our related post measuring latency distributions for block diffusion and for attestations.

In PeerDAS, we are in fact mapping the dispersal problem to a relatively small number of GossipSub topics. While this is a reasonable first approach, there are aspects that are specific to the dispersal problem and allow for performance optimisations.

Making dispersal even more efficient

We have studied how fast blocks can be dispersed into custody usingour custom DAS simulator, both for the 1D and for the 2D erasure coded case. If column and row topics are correctly populated, GossipSub can disperse the block data in seconds to thousands of nodes, as shown in our presentations at EthereumZurich'23, and in later presentations (EDCON'23, EthPrague'23, EthCC'23). However, in these talks we are speaking of GossipSub-like pub-sub protocols for a reason. GossipSub, as is, is not the best option we have. We argue that we can make a few changes to make it work well for our case. Eventually, we might also derive a custom protocol specifically targeting the dispersal use case.

In what follows, we list a few of the possible differences between a generic use of GossipSub topics and dispersal, showing some of the optimizations that can be made. We will dedicate a separate post to extend these in detail.

Compared to generic GossipSub topics, DAS "topics" are extremely structured

- With the 512 row and 512 column structure, we can map any ID into 1+9 bits (1 bit for direction, 9 bits for position) or a 9+9 bits ID space.
- We can then use compact representations in message IDs, in HAVE lists, and in many other data structures.

- Given the compact representation, we can also piggyback diffusion state information easily on other messages.

With many topics, finding peers can become hard

- We address this partly by deterministic NodeID-based custody assignment,
- Partly by “topic routing” based on a topic ID proximity metric.

DAS segments of a single block are unordered, which we can use to our benefit

- We can speed up delivery by “rarest first” like techniques. Rarest-fist scheduling is used in P2P dissemination to cut the tail of the latency distribution, equalizing the diffusion of different pieces. Rarest-first requires precise “have” information, which is typically not available, but it can be approximated. For example, a technique we have already implemented in our simulators is to make sure a copy of every single segment is sent out by the block producer before sending the second copy to another neighbor. We start by sending only one copy of each segment, to one of the block producer’s column/row neighbors (randomized), seeding segments in random places in the network. Only then we send second copies to other peers. Later, one can use information collected through e.g. IHAVE bitmaps to estimate which segments are diffusing well, and which seem to be hindered, prioritizing further seeding based on this information.
- We might also speed up delivery by “node coloring” techniques: these techniques are about specializing nodes to prioritize the distribution of a specific subset of the ID space they custody.

With TCP we pay the cost of in-order reliable delivery

- GossipSub defaults to using in-order reliable delivery over the topic mesh, which has an inherent cost.
- DAS does not need ordering between segments (to the contrary, shuffling is beneficial).
- DAS does not need P2P link-level reliability (because of all the other reliability techniques we have: multi-path, EC, pull, cross-feeding between columns and rows).
- Thus, using e.g. randomized best-effort delivery can provide overall better performance.

Duplicate reduction techniques

- With GossipSub we pay a relatively high bandwidth cost of duplicate deliveries. Basically, every single message traverses every topic mesh link at least once. Sometimes even twice, when sent from both sides close in time (less than the one-way latency). This means that on average, every message is sent at least D/2 times, where D is the target mesh degree.
- These duplicates serve multiple purposes: contribute to robustness, to low latency, and to protecting sender identity. However, in DAS, we do not need to protect sender identity. Because of the traffic volume, we can’t protect it. This, and the fact that we are distributing over several topics in parallel, allow for optimizations.
- One option is to use the IDONTWANT proposal
- We might also use techniques based on diffusion state, such as Push-Pull phase transition: in the first steps of the diffusion of a segment, there are almost no duplicates. Most of the duplicate load is happening in the last steps of the diffusion. One can emphasize Push at the beginning (even with an increased degree, sending to more nodes than without this technique), and change to gossip-and-pull towards the end of diffusion if the diffusion state can be estimated.

- One option is to use the IDONTWANT proposal
- We might also use techniques based on diffusion state, such as Push-Pull phase transition: in the first steps of the diffusion of a segment, there are almost no duplicates. Most of the duplicate load is happening in the last steps of the diffusion. One can emphasize Push at the beginning (even with an increased degree, sending to more nodes than without this technique), and change to gossip-and-pull towards the end of diffusion if the diffusion state can be estimated.

The Need for 2D Encoding: availability amplification through in-network repair

Erasure Coding is mostly presented in DAS descriptions as a tool to make sampling more robust. However, we should highlight that we can use Erasure Coding in the design for two purposes:

1. to amplify sampling, and
2. to amplify availability.

Amplifying sampling

is well understood: using a code with rate R=K/N (e.g. 1/2), and assuming the data is not available, every single sampled segment reduces the chance of not finding out about the data not being available by a factor of 2. We like this exponential property, letting us reach a very low False Positive (FP) rate with only a few segments sampled. In the 2D case, the ratio is not as good as in 1D, but we still have a nice exponential property.

Amplifying availability

is a different thing. It means that if data is available, we make it overwhelmingly available. We achieve this by using in-network repair

during dispersal. This is where our erasure coding structure and dispersal network structure meet. By organising nodes according to the code, in columns and rows, and by making nodes participate in both rows and columns, we enable local repair in the EC structure during dispersal.

This in-network repair is then amplifying availability, as we have shown in simulations already in ourEthereumZuri.ch'23 talk. Basically, if not enough samples are released, segment count stays below 75%, while row/coumn-based sampling (SubnetSampling), used by validators, and also by full nodes in many of our constructs, remains very low. Segment sampling also remains very low.

If instead one more segment is released, the result is amplified by the dispersal structure to almost 100%.

For this amplification mechanism to work, however, we need the 2D code. We cannot really do it in the 1D case, because in the 1D case, repair could only be done in nodes having K pieces, basically the whole block. In the 2D case, instead, we can repair in any single row or column, so individual nodes can repair and then send. In other words, we need a code with local repair. We can make such a code with 2D RS (there are also other code constructs with local repair capabilities, if we want).

Thus, while we could map segment IDs to custody in many ways, we do it by columns and rows to enable in-network repair and availability amplification during dispersal.

Conclusions

Our aim with this post was to outline the main building blocks of FullDAS, the networking stack we propose for DAS with blocks of 32MB and beyond. We have discussed new ways to disperse the block data into a P2P structure of custody, while also amplifying data availability. We have also shown the tools needed to implement fast, bandwidth-efficient, and robust sampling from this structure, using liteDAS, Topic Routing, and improved connection mechanisms.

At this point it might be evident to many that we are building something similar to a DHT, a point raised a few times in the past: the difference between dispersal to custody and sampling, and the generic concept of a DHT is not that big. With FullDAS we are in fact building a special purpose distributed storage structure, with custom seeding, repair, and retrieval, each optimized for the DAS encoding and for the DAS purpose.

Some of the above techniques have already been evaluated in simulation, while others are work in progress. We have written two simulators for this purpose:

- one with a high-level of abstraction, written in Python, for large scale experiments. Here protocol behaviour is approximated, but a larger parameter space can be explored.
- one with a lower level abstraction, using directly the nim-libP2P stack. We already used this simulator to simulate diffusion with a modified GossipSub implementation with 128 columns and 128 rows, and to run sampling from this structure to 1000s of nodes.

Both of these are work in progress, and we plan to release further posts as the refinement of FullDAS protocols and the evaluation goes on.

References

The original Danksharding proposal

The DAS data structure

Our previous post on sampling techniques: LossyDAS, IncrementalDAS, and DiDAS

Scalability limitations of using Kademlia DHTs for DAS

The PeerDAS proposal

The SubnetDAS proposal

Our post on measuring GossipSub latency distribution, both when used for blocks and when used for attestations

Part of PeerDAS specification discussions about peer search and connectivity issues

Proof Of Custody

Our previous presentations on DAS:

- EthereumZurich’23
- EDCON’23
- EthPrague’23
- EthCC’23)

Our DAS simulators:

- DAS simulator with a high-level of abstraction, written in Python, for large scale experiments
- DAS simulator with a lower level abstraction, using directly the nim-libP2P stack over Shadow