Leveraging Kademlia DHT for Efficient Data Availability Sampling

Prabal Banerjee

Avail

Outline

Layout

Avail - Introduction

- Modular blockchain focused on data availability [tx ordering, publication]
- Data and execution agnostic - any environment [EVM, WASM, etc]
- Zero knowledge and optimistic rollups, validiums

Avail - Introduction

- Substrate based full nodes
- Thin light client (LC) used for data availability sampling (DAS)
- Rust libp2p LC networking

Layout

Layout

Erasure Coding

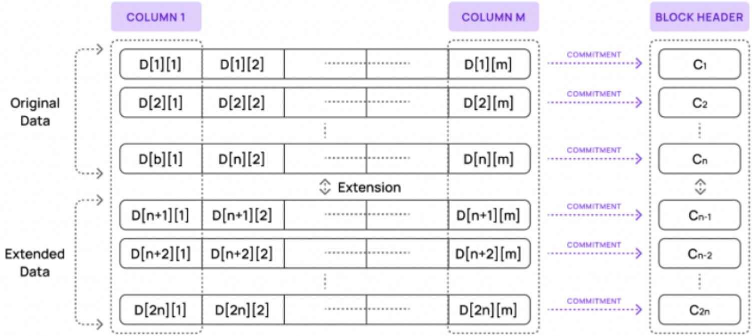
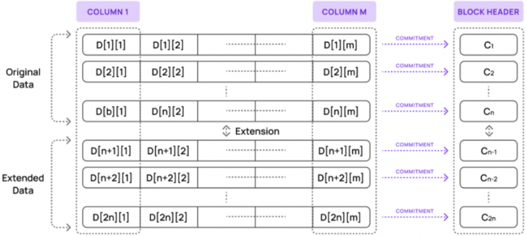


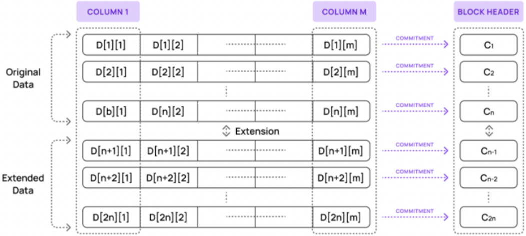
Figure: Erasure coding

Erasure Coding



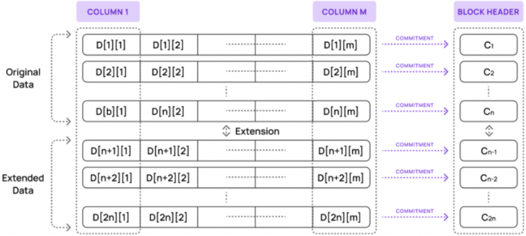
- Proposers split each block into $\boldsymbol{M} \times \boldsymbol{N}$ matrix

Erasure Coding



- Proposers split each block into $\boldsymbol{M} \times \boldsymbol{N}$ matrix
- Each cell is 32 bytes long [req for BLS381]

Erasure Coding



- Proposers split each block into $\boldsymbol{M} \times \boldsymbol{N}$ matrix
- Each cell is 32 bytes long [req for BLS381]
- Original matrix erasure coded into $\mathbf{2}\boldsymbol{M} \times \boldsymbol{N}$ size matrix - column size doubled

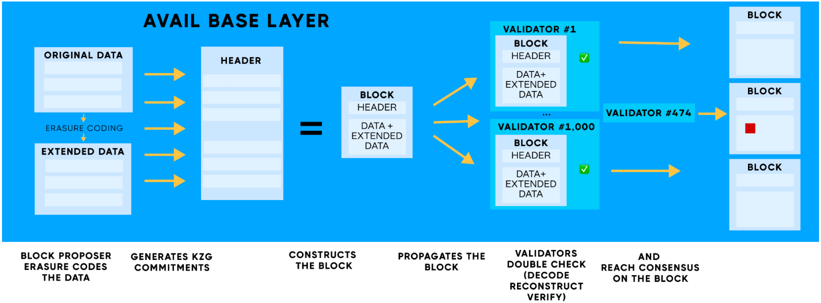
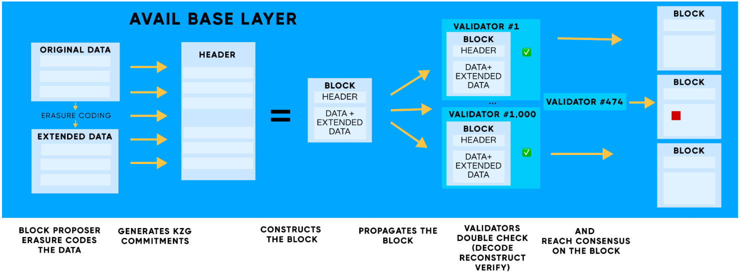


Figure: Avail base layer



- KZG commitments are generated for each row, placed in the block header
- Validators recreate the commitments and ensure they are correct

Layout

Data Availability Sampling

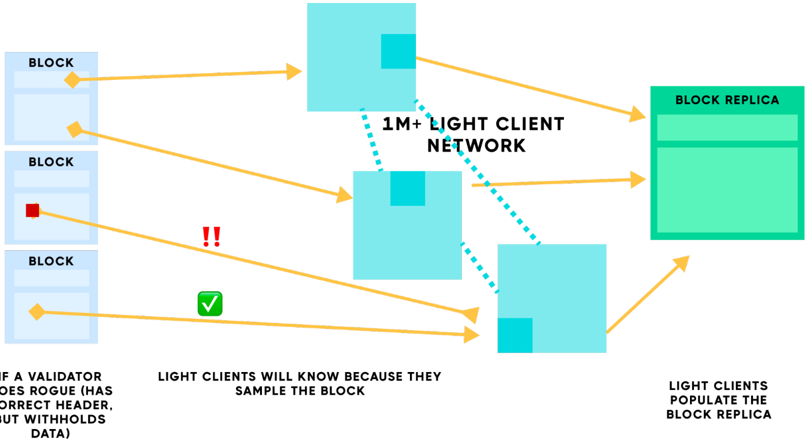
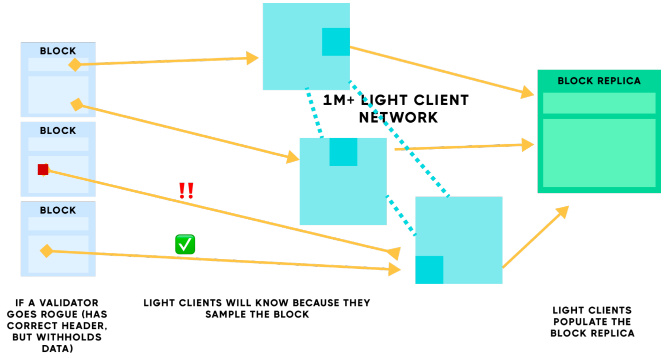


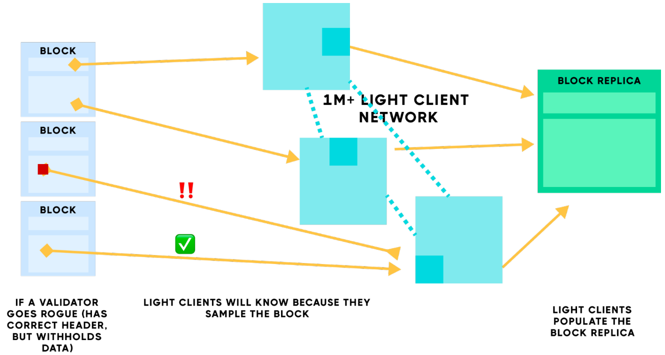
Figure: Light client network

Data Availability Sampling



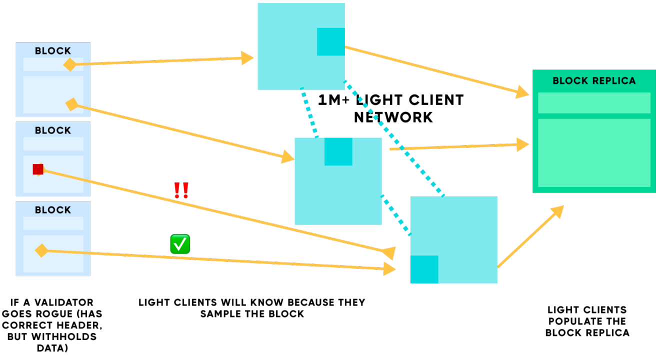
- DAS - performed on every block by LCs

Data Availability Sampling



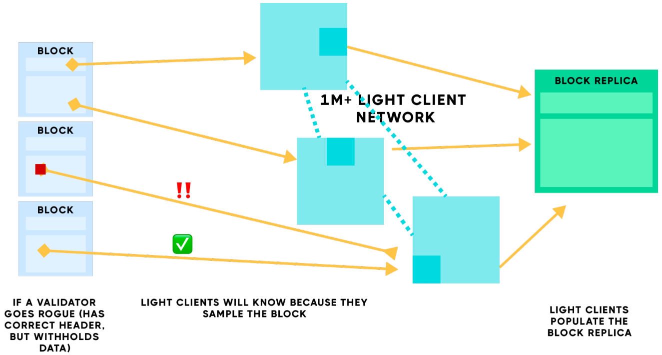
- DAS - performed on every block by LCs
- A number of random cells are retrieved

Data Availability Sampling



- DAS - performed on every block by LCs
- A number of random cells are retrieved
- Data is verified against the commitments from headers

Data Availability Sampling



- DAS - performed on every block by LCs
- A number of random cells are retrieved
- Data is verified against the commitments from headers
- And block confidence is thus calculated

Layout

Logic Separation

- Nodes generate proofs for the requested cells
- Cell size: 80 bytes [32 bytes padded data + 48 bytes proof]
- Light Client logic is separated:
  1. Light Client - responsible for DAS
  2. App Client - reconstructs the data for a given app ID

Layout

From IPFS to KAD

- Initial architecture used IPFS
- Entire blocks were delivered to P2P network
- Cells were encoded into columns, and columns into blocks [IPLD]
- This approach proved to be inefficient for random sampling and too rigid for individual cell retrieval

- Obvious optimization was replacing IPFS with KAD - remove unnecessary intermediate step
- Network traffic decreased - needed cells could be pinpointed and downloaded
- In-memory store decreased - not holding entire columns just for few needed cells

From IPFS to KAD

- Initial architecture used IPFS
- Entire blocks were delivered to P2P network
- Cells were encoded into columns, and columns into blocks [IPLD]
- This approach proved to be inefficient for random sampling and too rigid for individual cell retrieval

- Obvious optimization was replacing IPFS with KAD - remove unnecessary intermediate step
- Network traffic decreased - needed cells could be pinpointed and downloaded
- In-memory store decreased - not holding entire columns just for few needed cells

Layout

Layout

DAS on Light Client

(1) Receive block header from the node
(2) Calculate the number of random cells needed for block confidence threshold
(3) Randomly generate individual cell positions in the block matrix
(4) Try to retrieve cells from KAD
(5) If (4) fails, retrieve the delta via RPC call to Nodes
(6) Calculate block confidence and compare against threshold
(7) If (6) passed check, upload the delta downloaded via RPC to KAD
(8) Signal the App Client that the block has been verified

DAS on Light Client

(1) Receive block header from the node
(2) Calculate the number of random cells needed for block confidence threshold
(3) Randomly generate individual cell positions in the block matrix
(4) Try to retrieve cells from KAD
(5) If (4) fails, retrieve the delta via RPC call to Nodes
(6) Calculate block confidence and compare against threshold
(7) If (6) passed check, upload the delta downloaded via RPC to KAD
(8) Signal the App Client that the block has been verified

DAS on Light Client

(1) Receive block header from the node
(2) Calculate the number of random cells needed for block confidence threshold
(3) Randomly generate individual cell positions in the block matrix
(4) Try to retrieve cells from KAD
(5) If (4) fails, retrieve the delta via RPC call to Nodes
(6) Calculate block confidence and compare against threshold
(7) If (6) passed check, upload the delta downloaded via RPC to KAD
(8) Signal the App Client that the block has been verified

DAS on Light Client

(1) Receive block header from the node
(2) Calculate the number of random cells needed for block confidence threshold
(3) Randomly generate individual cell positions in the block matrix
(4) Try to retrieve cells from KAD
(5) If (4) fails, retrieve the delta via RPC call to Nodes
(6) Calculate block confidence and compare against threshold
(7) If (6) passed check, upload the delta downloaded via RPC to KAD
(8) Signal the App Client that the block has been verified

DAS on Light Client

(1) Receive block header from the node
(2) Calculate the number of random cells needed for block confidence threshold
(3) Randomly generate individual cell positions in the block matrix
(4) Try to retrieve cells from KAD
(5) If (4) fails, retrieve the delta via RPC call to Nodes
(6) Calculate block confidence and compare against threshold
(7) If (6) passed check, upload the delta downloaded via RPC to KAD
(8) Signal the App Client that the block has been verified

DAS on Light Client

(1) Receive block header from the node
(2) Calculate the number of random cells needed for block confidence threshold
(3) Randomly generate individual cell positions in the block matrix
(4) Try to retrieve cells from KAD
(5) If (4) fails, retrieve the delta via RPC call to Nodes
(6) Calculate block confidence and compare against threshold
(7) If (6) passed check, upload the delta downloaded via RPC to KAD
(8) Signal the App Client that the block has been verified

DAS on Light Client

(1) Receive block header from the node
(2) Calculate the number of random cells needed for block confidence threshold
(3) Randomly generate individual cell positions in the block matrix
(4) Try to retrieve cells from KAD
(5) If (4) fails, retrieve the delta via RPC call to Nodes
(6) Calculate block confidence and compare against threshold
(7) If (6) passed check, upload the delta downloaded via RPC to KAD
(8) Signal the App Client that the block has been verified

DAS on Light Client

(1) Receive block header from the node
(2) Calculate the number of random cells needed for block confidence threshold
(3) Randomly generate individual cell positions in the block matrix
(4) Try to retrieve cells from KAD
(5) If (4) fails, retrieve the delta via RPC call to Nodes
(6) Calculate block confidence and compare against threshold
(7) If (6) passed check, upload the delta downloaded via RPC to KAD
(8) Signal the App Client that the block has been verified

Layout

Application Clients

- All apps are assigned a uniqueID
- App clients reconstruct app data - retrieve more data than LCs
- Row wise stored data is verified by commitment equality
- If commitments check out - clients proceed with data reconstruction

Application Clients

- All apps are assigned a uniqueID
- App clients reconstruct app data - retrieve more data than LCs
- Row wise stored data is verified by commitment equality
- If commitments check out - clients proceed with data reconstruction

Application Clients

- All apps are assigned a uniqueID
- App clients reconstruct app data - retrieve more data than LCs
- Row wise stored data is verified by commitment equality
- If commitments check out - clients proceed with data reconstruction

Application Clients

- All apps are assigned a uniqueID
- App clients reconstruct app data - retrieve more data than LCs
- Row wise stored data is verified by commitment equality
- If commitments check out - clients proceed with data reconstruction

Layout

App Client Data Retrieval Steps

(a) Try to retrieve all the relevant rows from KAD
(b) If (a) fails, try to retrieve missing rows from Nodes
(c) If (b) fails, try to retrieve all the individual cells of those rows from KAD
(d) If (c) fails, try to retrieve $>$50% of needed cells (column wise) from KAD - enough for erasure coded data to be restored

Layout

Implementation challenges

- Light clients need to prioritize KAD instead of node network
- Internal stress testing revealed some interesting findings regarding at-scale Kademlia use
- The main challenge is delivering all of the cells into DHT in under the block time
- Huge number of very small data chunks create CPU strain just from handling stream multiplexing and connections
- Fine tuning Kademlia parameters can further optimize the network for specific use case
  - Reducing replication factor speeds up cell delivery
  - Raising max record size to 8kb had no apparent performance penalties
  - Huge memory consumption with default memory allocator (small heap chunks never deallocated)

Implementation challenges

- Light clients need to prioritize KAD instead of node network
- Internal stress testing revealed some interesting findings regarding at-scale Kademlia use
- The main challenge is delivering all of the cells into DHT in under the block time
- Huge number of very small data chunks create CPU strain just from handling stream multiplexing and connections
- Fine tuning Kademlia parameters can further optimize the network for specific use case
  - Reducing replication factor speeds up cell delivery
  - Raising max record size to 8kb had no apparent performance penalties
  - Huge memory consumption with default memory allocator (small heap chunks never deallocated)

Implementation challenges

- Light clients need to prioritize KAD instead of node network
- Internal stress testing revealed some interesting findings regarding at-scale Kademlia use
- The main challenge is delivering all of the cells into DHT in under the block time
- Huge number of very small data chunks create CPU strain just from handling stream multiplexing and connections
- Fine tuning Kademlia parameters can further optimize the network for specific use case
  - Reducing replication factor speeds up cell delivery
  - Raising max record size to 8kb had no apparent performance penalties
  - Huge memory consumption with default memory allocator (small heap chunks never deallocated)

Implementation challenges

- Light clients need to prioritize KAD instead of node network
- Internal stress testing revealed some interesting findings regarding at-scale Kademlia use
- The main challenge is delivering all of the cells into DHT in under the block time
- Huge number of very small data chunks create CPU strain just from handling stream multiplexing and connections
- Fine tuning Kademlia parameters can further optimize the network for specific use case
  - Reducing replication factor speeds up cell delivery
  - Raising max record size to 8kb had no apparent performance penalties
  - Huge memory consumption with default memory allocator (small heap chunks never deallocated)

Implementation challenges

- Light clients need to prioritize KAD instead of node network
- Internal stress testing revealed some interesting findings regarding at-scale Kademlia use
- The main challenge is delivering all of the cells into DHT in under the block time
- Huge number of very small data chunks create CPU strain just from handling stream multiplexing and connections
- Fine tuning Kademlia parameters can further optimize the network for specific use case
  - Reducing replication factor speeds up cell delivery
  - Raising max record size to 8kb had no apparent performance penalties
  - Huge memory consumption with default memory allocator (small heap chunks never deallocated)

Implementation challenges

- Light clients need to prioritize KAD instead of node network
- Internal stress testing revealed some interesting findings regarding at-scale Kademlia use
- The main challenge is delivering all of the cells into DHT in under the block time
- Huge number of very small data chunks create CPU strain just from handling stream multiplexing and connections
- Fine tuning Kademlia parameters can further optimize the network for specific use case
  - Reducing replication factor speeds up cell delivery
  - Raising max record size to 8kb had no apparent performance penalties
  - Huge memory consumption with default memory allocator (small heap chunks never deallocated)

Implementation challenges

- Light clients need to prioritize KAD instead of node network
- Internal stress testing revealed some interesting findings regarding at-scale Kademlia use
- The main challenge is delivering all of the cells into DHT in under the block time
- Huge number of very small data chunks create CPU strain just from handling stream multiplexing and connections
- Fine tuning Kademlia parameters can further optimize the network for specific use case
  - Reducing replication factor speeds up cell delivery
  - Raising max record size to 8kb had no apparent performance penalties
  - Huge memory consumption with default memory allocator (small heap chunks never deallocated)

Implementation challenges

- Light clients need to prioritize KAD instead of node network
- Internal stress testing revealed some interesting findings regarding at-scale Kademlia use
- The main challenge is delivering all of the cells into DHT in under the block time
- Huge number of very small data chunks create CPU strain just from handling stream multiplexing and connections
- Fine tuning Kademlia parameters can further optimize the network for specific use case
  - Reducing replication factor speeds up cell delivery
  - Raising max record size to 8kb had no apparent performance penalties
  - Huge memory consumption with default memory allocator (small heap chunks never deallocated)

Optimizations and future development

- Switch from TCP to QUIC yielded some net positive results
- Replacing default Rust allocator for jemalloc allowed for a smaller memory footprint
- Introducing polynomial multiproofs
  - Polynomial commitment scheme that allows for efficiently creating/verifying opening proofs for multiple polynomials at multiple points
  - Allows for grid coalescing - faster, more secure system and far greater throughput

Optimizations and future development

- Switch from TCP to QUIC yielded some net positive results
- Replacing default Rust allocator for jemalloc allowed for a smaller memory footprint
- Introducing polynomial multiproofs
  - Polynomial commitment scheme that allows for efficiently creating/verifying opening proofs for multiple polynomials at multiple points
  - Allows for grid coalescing - faster, more secure system and far greater throughput

Optimizations and future development

- Switch from TCP to QUIC yielded some net positive results
- Replacing default Rust allocator for jemalloc allowed for a smaller memory footprint
- Introducing polynomial multiproofs
  - Polynomial commitment scheme that allows for efficiently creating/verifying opening proofs for multiple polynomials at multiple points
  - Allows for grid coalescing - faster, more secure system and far greater throughput

Optimizations and future development

- Switch from TCP to QUIC yielded some net positive results
- Replacing default Rust allocator for jemalloc allowed for a smaller memory footprint
- Introducing polynomial multiproofs
  - Polynomial commitment scheme that allows for efficiently creating/verifying opening proofs for multiple polynomials at multiple points
  - Allows for grid coalescing - faster, more secure system and far greater throughput

Optimizations and future development

- Switch from TCP to QUIC yielded some net positive results
- Replacing default Rust allocator for jemalloc allowed for a smaller memory footprint
- Introducing polynomial multiproofs
  - Polynomial commitment scheme that allows for efficiently creating/verifying opening proofs for multiple polynomials at multiple points
  - Allows for grid coalescing - faster, more secure system and far greater throughput

Thanks!