NEAR light client

A trustless off-chain light client for NEAR with DA-enabled features, such as KZG commitments, Reed-Solomon erasure coding & storage connectors.

Features

A fully featured, injectable, dependable implementation of the NEAR light client protocol.

- Injectable
- : Interoperability with dependencies is very hard to maintain and eventually falls by the wayside.
- With the light client being injectable, developers can inject cryptographic semantics and the light client logic should be the same, with the only difference being what environment the application uses.
- Robust
- : With an open source, contributor-friendly effort, stringent semantic versioning and audits, we can be sure that the light client will be robust and maintainable.
- Dependable
- : The Protocol itself should not be dependent on many dependencies which would detract from its usability. Ideally, the light client should be#[no_std]
- , and it should be environment-agnostic.

info To learn more, check theNear light client specification .

Blob verification

The light client provides easy access to transaction and receipt inclusion proofs within a block or chunk. This is useful for checking any dubious blobs which may not have been submitted or validating that a blob has been submitted to NEAR.

A blob submission can be verified by:

- taking the NEAR transaction ID from Ethereum for the blob commitment.
- Ask the light client for an inclusion proof for the transaction ID or the receipt ID if you're feeling specific; this will give you a Merkle inclusion proof for the transaction/receipt.
- once you have the inclusion proof, you can ask the light client to verify the proof for you, or advanced users can manually verify it themselves.
- armed with this knowledge, rollup providers can have advanced integration with light clients and build proving systems around it.

Implementations

At present, there are currently two implementations of the light client. Thestd and thezk version.

Off-chain client

This is the first light client that was used to build on the logic of the protocol, it has the greatest environment assumptions and a bunch of dependencies.

Theoff-chain light client , which syncs with the final head of the chain, applies the sync protocol and stores the block producers and the next header. This is the most basic way to run the light client, you can configure the{ENVIRONMENT}.toml to set your trusted checkpoints, state and the host exposed and this will work out of the box, syncing with the state and catching up if needed.

State

It also stores some information in the state database. This contains a few tries insled , namely block producers, headers and any cachable information used for verification.

Interface

It exposes an HTTP interface for querying state and providing proofs. It exposes a JSON-RPC implementation to be more compatible with users already aware ofNEAR RPC nodes.

ZK client

It's a fully featured light client protocol, providing sync and transaction verification. It exploits STARK acceleration for the STARK-friendly functionality and parallel proving for the Merkle verification. We aim to also fold verification and syncing with

proof recursion so the light client can act lazily, syncing when needed, vs syncing eagerly.

tip Check out the ZK light client on theSuccinct network here . The initial implementation of the ZK light client protocol can befound here . It leveragesSuccinct's prover network andplonky2x SDK as a proving system. This will allow developers to pay for proof generation from the proof market.

Circuits

Below are the current circuits for the ZK light client.

note The current circuits serve only a "one-shot" command style for syncing/verification and no autonomous proving. Since the ZK client integrates with a Solidity Contract on chain and the circuit must have a statically aligned size, we have to minimize as much call data as possible, opting to witness verification in the circuit rather than store information.

Sync

Syncs to the nexthead , either the last header of the next epoch or the next header in the current epoch.

Public inputs:

- trusted_header_hash
- : This is the last header that was synced or the checkpoint header. We use this to query the header info and witness the header is valid, as well as to ensure the header was once synced.

Public Outputs:

- next_header_hash
- : The header hash of the next header that has been synced in the protocol.

Verify

Verifies a batch of transactions/receipts, this wraps the Merkle proof verification of multiple transactions in a parallelised circuit. This allows us to witness the verification of arbitrary amounts of transactions and only pay for verification on Ethereum once, with the relay of the results for the transactions/receipts calldata being the most fees.

Public inputs:

- trusted_header_hash
- : The last header that was synced. We can use this to also determine if we need to sync in this verification. And to query the header information.
- transaction_or_receipt_ids
- : The transactions or receipts to be verified.

Public Outputs:

- transaction_or_receipt_results
- : The IDs and the result of their verification