title: Statelessness, state expiry and history expiry description: Explanation of history expiry and stateless Ethereum lang: en

Statelessness, state expiry and history expiry {#statelessness}

The ability to run Ethereum nodes on modest hardware is critical for true decentralization. This is because running a node gives users the ability to verify information by performing cryptographic checks independently rather than trusting a third party to feed them data. Running a node allows users to submit transactions directly to the Ethereum peer-to-peer network rather than having to trust an intermediary. Decentralization is not possible if these benefits are only available to users with expensive hardware. Instead, nodes should be able to run with extremely modest processing and memory requirements so that they can run on mobile phones, micro-computers or unnoticeably on a home computer.

Today, high disk space requirements is the main barrier preventing universal access to nodes. This is primarily due to the need to store large chunks of Ethereum's state data. This state data contains critical information required to correctly process new blocks and transactions. At the time of writing, a fast 2TB SSD is recommended for running a full Ethereum node. For a node that does not prune any older data, the storage requirement grows at around 14GB/week, and archive nodes that store all data since genesis are approaching 12 TB (at time of writing, in Feb 2023).

Cheaper hard drives can be used to store older data but those are too slow to keep up with incoming blocks. Keeping the current storage models for clients while making data cheaper and easier to store is only a temporary and partial solution to the problem because Ethereum's state growth is 'unbounded', meaning that storage requirements can only ever increase, and technological improvements will always have to keep pace with continual state growth. Instead, clients must find new ways to verify blocks and transactions that doesn't rely on looking up data from local databases.

Reducing storage for nodes {#reducing-storage-for-nodes}

There are several ways to reduce the amount of data each node has to store, each requiring Ethereum's core protocol to be updates to a different extent:

- **History expiry**: enable nodes to discard state data older than X blocks, but does not change how Ethereum client's handle state data
- **State expiry**: allow state data that is not used frequently to become inactive. Inactive data can be ignored by clients until it is resurrected.
- **Weak statelessness**: only block producers need access to full state data, other nodes can verify blocks without a local state database.
- **Strong statelessness**: no nodes need access to the full state data.

Data expiry {#data-expiry}

History expiry {#history-expiry}

History expiry refers to clients pruning away older data that they are unlikely to need, so that they only store a small amount of historical data, dropping older data when new data arrives. There are two reasons clients require historical data: syncing and serving data requests. Originally, clients had to sync from the genesis block, verifying that each successive block is correct all the way to the head of the chain. Today, clients use "weak subjectivity checkpoints" to bootstrap their way to the head of the chain. These checkpoints are trusted started points, like having a genesis block close to the present rather than the very start of Ethereum. This means clients can drop all information prior to the most recent weak subjectivity checkpoint without losing the ability to sync to the head of the chain. Clients currently serve requests (arriving via JSON-RPC) for historical data by grabbing it from their local databases. However, with history expiry this will not be possible if the requested data has been pruned. Serving this historical data is where some innovative solutions are required.

One option is that clients request historical data from peers using a solution such as the Portal Network. The Portal Network

is an in-development peer-to-peer network for serving historical data where each node stores a small piece of Ethereum's history such that the entire history exists distributed across the network. Requests are served by seeking out peers storing the relevant data and requesting it from them. Alternatively, since it is generally apps that require access to historical data, it can become their responsibility to store it. There may also be enough altruistic actors in the Ethereum space that would be willing to maintain historical archives. It could be a DAO that spins up to manage historical data storage, or ideally it will be a combination of all these options. These providers could serve the data in many ways, such as on a torrent, FTP, Filecoin or IPFS.

History expiry is somewhat controversial because so far Ethereum has always implicitly guaranteed the availability of any historical data. A full sync from genesis has always been possible as standard, even if it relies on rebuilding some older data from snapshots. History expiry moves the responsibility for providing this guarantee outside of the Ethereum core protocol. This could introduce new censorship risks if it is centralized organizations that end up stepping in to provide historical data.

EIP-4444 is not yet ready to ship, but it is under active discussion. Interestingly, the challenges with EIP-4444 are not so much technical, but mostly community management. In order for this to ship, there needs to be community buy-in that includes not only agreement but also commitments to store and serve historical data from trustworthy entities.

This upgrade doesn't fundamentally change how Ethereum nodes handle state data, it just changes how historical data is accessed.

State expiry {#state-expiry}

State expiry refers to removing state from individual nodes if it hasn't been accessed recently. There are several ways this could be implemented, including:

- **Expire by rent**: charging "rent" to accounts and expiring them when their rent reaches zero
- **Expire by time**: making accounts inactive if there are no reading/writing to that account for some amount of time

Expiry by rent could be a direct rent charged to accounts to keep them in the active state database. Expiry by time could be by countdown from the last account interaction, or it could be periodic expiry of all accounts. There could also be mechanisms that combine elements of both the time and rent based-models, for example individual accounts persist in the active state if they pay some small fee prior to time based expiry. With state expiry it is important to note that inactive state is **not deleted**, it is just stored separately from the active state. The inactive state can be resurrected into the active state.

The way this would work is probably to have a state tree for specific time periods (perhaps ~1 year). Whenever a new period begins, so does a completely fresh state tree. Only the current state tree can be modified, all others are immutable. Ethereum nodes are only expected to hold the current state tree and the next most recent one. This requires a way to time-stamp an address with the period it exists in. There are several possible ways to do this, but the leading option requires addresses to be lengthened to accommodate the additional information with the added benefit that longer addresses are much more secure. The roadmap item that does this is called address space extension.

Similarly to history expiry, under state expiry responsibility for storing old state data is removed from individual users and pushed onto other entities such as centralized providers, altruistic community members or more futuristic decentralized solutions such as the Portal Network.

State expiry is still in the research phase and not yet ready to ship. State expiry may well happen later than stateless clients and history expiry because those upgrades make large state sizes easily manageable for the majority of validators.

Statelessness {#statelessness}

Statelessness is a bit of a misnomer because it does not mean the concept of "state" is eliminated, but it does involve changes to how Ethereum nodes handle state data. Statelessness itself comes in two flavors: weak statelessness and strong statelessness. Weak statelessness enables most nodes to go stateless by putting responsibility for state storage onto a few. Strong statelessness completely removes the need for any node to store the full state data. Both weak and strong statelessness offer the following benefits to normal validators:

- nearly instant syncing

- ability to validate blocks out-of-order
- nodes able to run with very low hardware requirements (e.g. on phones)
- nodes can run on top of cheap hard drives because there is no disk reading/writing required
- compatible with future upgrades to Ethereum's cryptography

Weak Statelessness {#weak-statelessness}

Weak statelessness does involve changes to the way Ethereum nodes verify state changes, but it does not completely eliminate the need for state storage in all nodes on the network. Instead, weak statelessness puts the responsibility for state storage onto block proposers, while all other nodes on the network verify blocks without storing the full state data.

In weak statelessness proposing blocks requires access to full state data but verifying blocks requires no state data

For this to happen, Verkle trees must already have been implemented in Ethereum clients. Verkle trees are a replacement data structure for storing Ethereum state data that allow small, fixed size "witnesses" to the data to be passed between peers and used to verify blocks instead of verifying blocks against local databases. Proposer-builder separation is also required because this allows block builders to be specialized nodes with more powerful hardware, and those are the ones that require access to the full state data.

Statelessness relies on block builders maintaining a copy of the full state data so that they can generate witnesses that can be used to verify the block. Other nodes do not need access to the state data, all the information required to verify the block is available in the witness. This creates a situation where proposing a block is expensive, but verifying the block is cheap, which implies fewer operators will run a block proposing node. However, decentralization of block proposers is not critical as long as as many participants as possible can independently verify that the blocks they propose are valid.

Read more on Dankrad's notes

Block proposers use the state data to create "witnesses" - the minimal set of data that prove the values of the state that are being changed by the transactions in a block. Other validators do not hold the state, they only store the state root (a hash of the entire state). They receive a block and a witness and use them to update their state root. This makes a validating node extremely lightweight.

Weak statelessness is in an advanced state of research, but it relies upon proposer-builder separation and Verkle Trees to have been implemented so that small witnesses can be passed between peers. This means weak statelessness is probably a few years away from Ethereum Mainnet.

Strong statelessness {#strong-statelessness}

Strong statelessness removes the need for any blocks to store state data. Instead, transactions are sent with witnesses that can be aggregated by block producers. The block producers are then responsible for storing only that state that are needed for generating witnesses for relevant accounts. The responsibility for state is almost entirely moved to users, as they send witnesses and 'access lists' to declare which accounts and storage keys they are interacting with. This would enable extremely lightweight nodes, but there are tradeoffs including making it more difficult to transact with smart contracts.

Strong statelessness has been investigated by researchers but is not currently expected to be part of Ethereum's roadmap - it is more likely that weak statelessness is sufficient for Ethereum's scaling needs.

Current progress {#current-progress}

Weak statelessness, history expiry and state expiry are all in the research phase and are expected to ship several years from now. There is no guarantee that all of these proposals will be implemented, for example, if state expiry is implemented first there may be no need to also implement history expiry. There are also other roadmap items, such as Verkle Trees and Proposer-builder separation that need to be completed first.

Further reading {#further-reading}

- Vitalik statelessness AMA
- A theory of state size management
- Resurrection-conflict-minimized state bounding
- Paths to statelessness and state expiry
- EIP-4444 specification
- Alex Stokes on EIP-4444
- Why it's so important to go stateless
- The original stateless client concept notes
- More on state expiry
- Even more on state expiry