Recap: state size management techniques

In order to prevent the ethereum state size from growing without limit, we need some way to “expire” old state, so that participating nodes in the network no longer need to store that state. Even if most clients are stateless, it seems reasonable to expect that eventually the system will scale enough that the network cannot afford to indefinitely guarantee availability of all state. There are two approaches to expiring old state:

1. Explicitly delete it, and perhaps move it into some separate Merkle tree so someone who cares about that state object can get the Merkle branch and use it to revive the state object at some future time.

2. Do not move the object in the tree structure; instead, simply flag that position in the tree as “expired” so nodes do not make an effort to store it (and the protocol does not expect them to do so). Expired objects can be accessed (and de-expired) by sending a transaction that provides the Merkle proof (aka. witness) to access that state.

(1) corresponds to “classical storage rent”, (2) corresponds to the easiest extension of traditional “stateless clients” to a model where old state can be forgotten. Both approaches allow individual actors who care about specific state objects to keep track of the Merkle branches that they can later use to revive those objects if they get expired. However, both have distinct flaws.

(1) suffers from edge cases when contracts can be re-created at the same address at which a contract was already expired. That is, if a contract at address A is created, then expired, then the transaction that created the contract at address A is re-played, that could create a new object at address A which would interfere with the revival of the original object. Another type of situation is when an object is created at address A, then expired, then revived, then modified (eg. by sending contained funds to another account), then expired, then a revival is made using the Merkle branch from the first expiry. This violates conservation rules and could be used to print coins; additional Merkle proofs need to be added to prove that a contract has not yet been revived from some given state from which a revival is attempted.

(2) suffers from a different problem. Suppose two adjacent addresses (meaning, no object exists in between them) A1 and A2 are both expired. Then, not only are A1 and A2 no longer accessible (unless someone still stores the Merkle branches), but also all addresses between A1 and A2. This means that, if there are N addresses in total, ~1/N of all possible address space is no longer accessible. By the time half of addresses are expired, ~1/4 of the address space is inaccessible. As time goes on, it becomes harder and harder to find space to generate new addresses. And because new addresses get concentrated in the remaining “accessible” space, this effect is exponential: the accessible space halves once per N years.

Proposal

I propose a version of (2) that is modified to solve the above problem. As in many proposed implementations of (2), accounts are either “active” or “expired”; an expired account is an account that has not been touched for >= 1 year. To access an expired account, you need to provide a witness; when an expired account is accessed, that account is automatically de-expired (touching any

account resets its 1 year time-to-expiry). The modification is as follows:

- We add a 32-bit “epoch prefix” to each address (meant to be interpreted as an integer). For example, an address with epoch prefix 9 would look like: 0x00000009de0b295669a9fd93d5f28d9ec85e40f4cb697bae

, with the 00000009

being the epoch prefix.

- The Merkle path would depend directly on the epoch prefix and not it hash (so merkle_path_key = address[:4] + hash(address[4:])

instead of merkle_path_key = hash(address)

as in the status quo). This ensures that “fresh” address space is contiguous.

- An address cannot be touched unless the epoch prefix of that address is less than or equal to the number of years that the chain has been in operation.

- A CREATE3 opcode would be added, that takes epoch prefix as an argument, and creates a contract at an address with that epoch prefix.

It would be recommended and the default for users and contracts to always create accounts with the newest possible epoch prefix because they will be certain that the full state of the newest epoch prefix is still accessible. To preserve the ability to have “counterfactual addresses” (addresses that users interact with on-chain [eg. by sending ETH or ERC20 tokens] or off-chain [eg. by interacting in a channel] before the contract code has been published), it would continue to be possible to create contracts with older epoch prefixes. However, users wishing to create counterfactual addresses that they leave un-created for a long time would take on the responsibility of storing old state branches for that account.

After many years of operation, it is expected that the active state would consist of (i) the entire portion of the address space with the most recent epoch prefix, and (ii) specific portions of the older states that correspond to accounts that have been actively used.

Note that this scheme can be naturally extended to contracts; in fact, it is in a contract’s own interest to voluntarily follow a schema where the portion of storage prefixed with some bytes representing the number N refers to data connected with addresses from year N. This could be naturally used to store eg. token balances.