The general goal of this post is to create a “layer 1” state execution framework that is maximally simple, and is intended to be used in practice together with considerable layer 2 infrastructure (ie. HLLs and smart contracts acting as standard libraries) on top; it is intended that different setups can compete with each other so there could be different types of contracts that work in very different ways, some roughly emulating ethereum 1.0-style contract interaction, others more UTXO-like, etc etc.

It assumes that there exists a VM called “EWASM” which takes as input code and data, runs the code, eventually exiting with a return value or an error code, and with us having the ability to specify a foreign function interface.

- There exists a global state value, TOTALFEE

. By default, TOTALFEE

is incremented by 1 gwei per block, but if desired more complex policies can be implemented.

- There exists one type of account, a contract. Each contract has a piece of code (a bytearray), and a piece of storage (another bytearray); its address is the hash of its initcode plus a salt (as in CREATE2). Contracts also store the TOTALFEE

value at the last time they were modified (TOTALFEE_WHEN_LAST_MODIFIED

).

- Contracts pay rent based on total size. That is, if a contract has stored TOTALFEE_WHEN_LAST_MODIFIED

F1, and it gets modified when the global TOTALFEE

is F2, the contract’s ETH balance is reduced by (F2 - F1) * (100 + len(code) + len(storage))

, and its TOTALFEE_WHEN_LAST_MODIFIED

is updated to F2. If the new balance is less than 0, the contract is deleted.

- There exists a special “poke” operation, with 0 gas cost but a limited number of calls per block, which acts as a no-op to some target account except that it counts as “modification”. This can be used to delete contracts that have not pay rent. In general, we rely on voluntary pokes by miners to clear out old accounts.

- Contract code is EWASM, with exposed operations READ_STORAGE

, SET_STORAGE

and CALL

, the latter calling a contract on the same shard and sending a specified amount of ETH to it.

- There exists a contract at a specific address (eg. 0x10) on each shard, which returns environment data (eg. block number, block hashes, timestamp…)

- There exists a contract at a specific address (eg. 0x20) on each shard, which has two functions generateReceipt

(which takes as input an amount of ETH, a target shard, a target address and calldata) and claimReceipt

(which takes as input a Merkle branch). Calling these functions is how contracts talk to contracts on other shards.

Not yet specified: account abstraction schemes. There are ideas here that can still be debated (eg. store nonces or not).

Another possible feature: we add to a contract’s FFI another operation, “yank”, which deletes the contract and creates a receipt which triggers the creation of the contract with the same code and storage and ETH balance on another shard (see cross-shard yanking).

Another possible feature: DELEGATECALL, so libraries can easily be built.

A possible alternative to rent: a contract maintains a TOTALFEE_TTL

value, and it can be destroyed if the current global TOTALFEE

exceeds TOTALFEE_TTL

. When a contract is touched it consumes extra G gas; G * MINFEE

(see this paper for the “minfee” concept) is added to the contract’s TOTALFEE_TTL

.

The following can all be done at “layer 2” or otherwise without modifying this basic scaffolding:

- Sleep-wake mechanisms (see the original post for how this can be done as a layer 2)
- Cross-shard calls
- High-level-language schemes for storage, eg. this one, as they can be implemented by using smart contracts as data stores
- The ZEXE framework, and other smart contract frameworks that don’t work based on smart contracts the way we have them today
- Contracts that use Merkle trees for storage
- Contracts that use accumulators with non-inclusion proofs for storage