One of the problems that we face in thecurrent sharding spec is the slowness of cross-shard communication. Within each shard, blocks come every six seconds, but between shards it takes six minutes or more for messages to get across. We can get around this problem by creating a computing model that we can design as a contract system on top of the current proposed sharded VM.

The basic idea behind the approach is for each underlying state object (ie. contract), we sometimes store multiple copies in the state, each with a different “dependency signature” (a set of claims about what state roots in what shards that copy is conditional on). Once the shard knows which claims about the state are true or false, the dependencies can be resolved, invalid copies of the state objects deleted, and the valid copy given the status of being the “official” copy of that contract.

Implementation

In each shard there exists a central manager contract M

, which has the power to create and delete “state-bearing” child contracts. Its main activity is processing transactions

. Each transaction has a TypeID

, a set of input indices idx_1 ... idx_n

, and a data

field (think “indices” = “internal addresses”).

The intent of a transaction is to execute a piece of code f

(where h(f) = TypeID

), which takes as input C_{old, 1} ... C_{old,n}

as well as data

and outputs a set of new values C_{new,1} ... C_{new,n}

. The C_{old}

values are the existing state of the objects specified by the transaction, and the C_{new}

values are the desired new state.

A state object at index idx

associated with some TypeID

has address hash(M, TypeID, idx)

(we assume that a CREATE2-like mechanism exists where the manager contract M

can create contracts with such addresses).

The above basically just implements a state model with staticaccess lists on top of the current contract-based state model. However, we now want to go further, and use this to speed up cross-shard transaction times.

Moving contracts across shards

We introduce a functionality that allows contracts to be moved from one shard to another with latency equal to one round of block time. To make this possible, we introduce a concept of a depedency signature

: a boolean formula over state roots in other shard (eg. (S_1, H_1, R_1)\ AND\ (S_2, H_2, R_2)\ AND\ NOT((S_3, H_3, R_3)))

). Each (S, H, R)

tuple, where S

is a shard ID, H

is a slot number and R

is a state root, is a “dependency”. At some point in the future, for any dependency, every shard will be able to learn through a crosslink whether or not it is “correct” (ie. whether or not the state root at that slot of that shard actually is the given root). An address of a contract now equals hash(M, TypeID, idx, dep)

.

Moving a contract starts by calling a function of the manager contract on the source shard S_1

, specifying the index idx

. The manager contract checks that the contract is eligible to be moved (the contract itself may specify conditions for this), and gets from the contract its dependency dep

and state \sigma

. It creates a receipt (containing (idx, \sigma, dep, ROOT_1)

) that can be included in shard S_2

. If dep = TRUE

(ie. no dependencies), the contract on shard S_1

is destroyed; otherwise, it is frozen temporarily, and when shard S_1

learns what slot H

and with what state root R

the receipt was included in the shard, the dependency is set to dep\ AND\ NOT((S_1, H, R))

.

Once the receipt is included in shard S_1

, any user can immediately

call a function in the manager contract in shard S_2

, which includes the receipt from shard S_1

, and creates a contract with state \sigma

in shard S_2

with index idx

and dependency dep\ AND\ (S_1, H, R)

.

When shard S_2

later learns that R

is in fact the state root of shard S_1

at slot H

, the manager contract can remove the dependency and move the contract to the address that does not have that dependency; if it learns that R

is not

the state root, it can simply delete the contract.

At this point, we now have a slightly improved version ofencumberments (improved because dependencies can be resolved out of order).

Transactions with dependencies

Now, suppose that one wishes to execute some transaction involving a contract with a nonempty dependency signature. If the manager contract receives a request to make a transaction where all inputs and outputs have the same dependency signature, then it simply executes the transaction as before, modifying the contracts in-place.

The more difficult case is where someone desires to perform an operation where different accounts have different dependency signatures. Let the dependency signatures for inputs i_1

… i_n

be dep_1

… dep_n

. We destroy the contracts at the input addresses (hash(M, TypeID, i_j, dep_j)

) and create two sets of new contracts:

- n

output objects all

of whose addresses have the dependency signature dep_1\ AND ...\ AND\ dep_n

, representing the case where all dependencies are valid.

- n

to cover the case where some of the input dependencies turn out not

to be valid. For the “leftover” output representing i_j

, the dependency signature is d_j\ AND\ NOT(d_1\ AND\ ...\ d_{j-1}\ AND\ d_{j+1}\ AND\ ...\ AND\ d_n)

.

Now, we have a system where we can move state objects between shards optimistically, and perform transactions with them, all with confirmation times of a few seconds, even if the underlying cross-shard transaction capability is slow.

To increase efficiency, we can require either a significant bond to be posted or a light client proof that some state root is probably valid before allowing contracts that include it as a dependency; we can then simplify the protocol by disallowing operations using any dependency signatures that contain NOT

clauses, which allows us to simply use lists of dependencies instead of boolean formulas, taking the union of two lists instead of an AND

; if there is a contract that contains a dependency with a NOT

clause that turns out to be correct, then anyone wishing to benefit from that contract must simply wait until crosslinks from shards can come in and anyone can call the manager contract to replace it with an equivalent dependency-free contract.

Consequences

If most activity is conducted through mechanisms such as these, there is no longer significant pressure for greatly decreasing layer 1 cross-shard communication times, and in fact we can choose to increase

those times to reduce beacon chain overhead. Furthermore, this means that there is no need for super-quadratic sharding; we can scale by adding as many shards as we want, and simply allow inter-crosslink time to increase to keep beacon chain overhead the same; cross-shard communication will feel

instant because it’s done on this layer 2 mechanism.