Status: an idea I've already talked with some people before, but still deserves to be written up in one place and hasn't yet.

We already know that it is relatively easy to supportasynchronous cross-shard communication, and we can extend that to solve train-and-hotel problems via cross-shard yanking, but this is still highly imperfect, because of the high latency inherent in waiting for multiple rounds of cross-shard communication. Being able to perform cross-shard operations synchronously, in one transaction, would be ideal. But how do we do this?

First of all, a solution would inherently require data/state execution separation. To see why, suppose that there is a transaction between shard A and shard B, that depends on state and makes changes in both shards. For simplicity, suppose that the transaction is trying to book a train and hotel. Head of the chain is in green.

Now, suppose that shard A reorgs, but not

shard B.

Oh no! The hotel is now booked and not the train. To avoid this, we would have to add a fork choice rule where A2 no longer being part of the canonical chain would also kick B2 off the canonical chain, but this would mean one single reorg on one shard could potentially destabilize every shard.

The abstraction that we use instead is the state execution engine. We suppose that a node is aware of the state of one shard (without loss of generality, shard A) at height N-1, and the state roots of all other shards at height N-1, and the correct block hashes at height N (and the full block for shard A), and its job is to compute the state of shard A and learn the state roots of all other shards at height N. Notice that we assume that correct block hashes are pre-provided; if any shard reverts then the execution process will need to revert, but the ordering of data on all other shards would be preserved.

Green is known state, grey is unknown state, yellow are blocks, squares are roots, circles are full data.

[

%5BD(n-1)%7Bbg%3Agreen%7D%5D%20-%3E%20%5BD(n)%5D%2C%5BBlock%20D(n)%7Bbg%3Ayellow%7D%5D%20-%3E%20%5BD(n)%5D%2C%5BC(n-1)%7Bbg%3Agreen%7D%5D%20-%3E%20%5BC(n)%5D%2C%5BBlock%20C(n)%7Bbg%3Ayellow%7D%5D%20-%3E%20%5BC(n)%5D%2C%5BB(n-1)%7Bbg%3Agreen%7D%5D%20-%3E%20%5BB(n)%5D%2C%5BBlock%20B(n)%7Bbg%3Ayellow%7D%5D%20-%3E%20%5BB(n)%5D%2C%5BA(n-1)%7Bbg%3Agreen%3Bshape%3Acircle%7D%5D%20-%3E%20%5BA(n)%7Bshape%3Acircle%7D%5D%2C%5BBlock%20A(n)%7Bbg%3Ayellow%3Bshape%3Acircle%7D%5D%20-%3E%20%5BA(n)%5D

1020×378

](https://yuml.me/diagram/scruffy/class/%5BD(n-1)%7Bbg%3Agreen%7D%5D%20-%3E%20%5BD(n)%5D%2C%5BBlock%20D(n)%7Bbg%3Ayellow%7D%5D%20-%3E%20%5BD(n)%5D%2C%5BC(n-1)%7Bbg%3Agreen%7D%5D%20-%3E%20%5BC(n)%5D%2C%5BBlock%20C(n)%7Bbg%3Ayellow%7D%5D%20-%3E%20%5BC(n)%5D%2C%5BB(n-1)%7Bbg%3Agreen%7D%5D%20-%3E%20%5BB(n)%5D%2C%5BBlock%20B(n)%7Bbg%3Ayellow%7D%5D%20-%3E%20%5BB(n)%5D%2C%5BA(n-1)%7Bbg%3Agreen%3Bshape%3Acircle%7D%5D%20-%3E%20%5BA(n)%7Bshape%3Acircle%7D%5D%2C%5BBlock%20A(n)%7Bbg%3Ayellow%3Bshape%3Acircle%7D%5D%20-%3E%20%5BA(n)%5D)

We define a block as containing a sparse Merkle tree (or Patricia tree or any similar key-value structure) of transactions, mapping address => tx

. Each transaction is itself a bundle [shard[1], address[1], shard[2], address[2] ... shard[n], address[n], data]

. For a transaction tx

in any shard to get executed, it must meet the following condition: for all (shard, address) pairs specified in the tx, accessing the key-value tree of block N of the given shard at the given address should return the transaction. That is, if the transaction specifies [A, 123, A, 485, B, 769, data]

, then for data

to get executed the transaction must appear at position 123 of block N in shard A, position 485 of block N in shard A and position 769 of block N in shard B.

Note that this requirement makes it impossible to have two transactions at the same height that affect the same account. This is by design. Otherwise, it would be possible to have towers of transactions that depend on each other, requiring clients to recursively download very large sets of transactions from other shards to verify a given transaction within their own shard. One possible compromise would be to allow unlimited transactions within

a shard that do not even need to specify addresses, but that are always executed after

the cross-shard transactions.

A client can implement this model as follows:

1. Download block N on shard A.
2. Collect all “foreign references” (that is, address references in transactions in shard A that come from other shards). For each foreign reference, ask the network for a Merkle branch for the state of the associated address from height N-1, and the block at the given position at height N.
3. For all transactions in shard A, verify that the references are consistent; that is, for every reference (s_id, addr)

in a transaction T

(foreign or local), verify that the value of the block of shard s_id

at position addr

is also T

, using the data acquired in stage 2 for foreign references. If it is not, throw the transaction out.

1. For every transaction that passed step (3), execute it, using the state data acquired in stage 2.
2. Use cryptoeconomic claims, ZK-SNARKs, or any other mechanism to gather an opinion about the state roots of other shards at height N.

To keep heights perfectly synchronized, we can use slot number instead of height; if a given slot number on some shard is missing, we treat that as an empty block. Note that this algorithm requires two rounds of network communication per height: one to fetch foreign Merkle branches, and another to gather claims about state roots of other shards.