TLDR

: given a base layer blockchain that offers slow asynchronous cross-shard transactions, we can emulate fast asynchronous cross-shard transactions as a layer on top.

This is done by storing “conditional states”, for example if Bob has 50 coins on shard B, and Alice sends 20 coins to Bob from shard A, but shard B does not yet know the state of shard A and so cannot fully authenticate the transfer, Bob’s account state temporarily becomes “70 coins if the transfer from Alice is genuine, else 50 coins”. Clients that have the ability to authenticate shard A and shard B can be sure of the “finality” of the transfer (ie. the fact that Bob’s account state will

eventually resolve to 70 coins once the transfer can be verified inside the chain) almost immediately, and so their wallets can simply act like Bob already has the 70 coins.

Furthermore, Bob has the ability to send the received coins to Charlie almost immediately, though Charlie’s wallet would also need to be able to authenticate shard A to verify the payment.

Assumptions

- We assume an underlying blockchain where asynchronous cross-shard calls are done with “receipts”

(eg. which ethereum 2.0 phase 2 is). That is, to send coins from shard A to shard B, the transaction on shard A “destroys” the coins but saves a record containing the destination shard, address and value sent. The record can be proven with a Merkle branch terminating in the state root of shard A,. After some delay each shard learns the state roots of all other shards, at which point the receipt can be included into shard B, its validity verified, and the coins “destroyed” in shard A can be “recovered” in shard B (this mechanism can be generalized beyond sending coins to asynchronous messages in general).

- To simplify things, instead of considering separate state roots for each shard, we talk about global state roots

, which are root hashes of the state roots for each shard (ie. merkle_root([get_state_root(shard=0), ... , get_state_root(shard=SHARD_COUNT-1)])

). We assume the beacon chain computes and saves these global state roots

with a delay or roughly 1-2 epochs in the normal case but potentially longer (they can be computed from crosslink data once crosslinks for all shards are submitted); once the beacon chain has calculates global state roots up to some height, all shards have access to these roots.

- These exists some “optimistic” way of discovering probable global receipt roots more quickly

than the in-protocol discovery delay; this mechanism need only work “most of the time”. Most crudely, this could be a prediction-market mechanism where the side that stakes the most funds on some root wins, but it also could be some kind of light-client design, where shards discover roots of other shards via a committee mechanism and after log(n) steps shard 0 discovers the global state root, which can then be verified by any other shard.

- Each shard maintains a list expected_roots

, which equals the global state roots in the beacon chain plus predicted newer state roots. This list can change in two ways: (i) a new predicted root can be added to the end, and (ii) the last N roots can be replaced if the prediction mechanism changes its mind about some of the unconfirmed roots.

Confirmed roots: blue, predicted but not confirmed roots: yellow. Over time new optimistic roots get added (and sometimes removed), and the confirmed roots follow from behind.

The mechanism

- For every account

, we store multiple states

(for a simple currency, a “state” might be balance and nonce; for a smart contract system it can be more complicated). Each state has one yes dependency

and zero or more no dependencies

. A dependency is a (slot, root)

pair, effectively saying “the global root at slot X is Y” . A state is called “active” if its yes dependency is in the expected_roots

list, and none of its no dependencies are.

- A transaction must specify a dependency, and it is only valid if its dependency is (i) in the expected_roots

list, (ii) equal to or newer than the yes dependencies of all the accounts it touches, and (iii) all account states it touches are active.

- For each account that the transaction touches:

- If the transaction’s dependency simply equals the yes dependency of the account, then the state of the account is simply edited in place

- If the transaction’s dependency is newer, then the state is edited and the dependency is updated to equal the newer dependency, but an “alternate state

” is created with the yes dependency equaling that of the old state, the no dependency list being that of the no state plus the dependency of the transaction, and the state being the same as the old state.

- If the transaction’s dependency simply equals the yes dependency of the account, then the state of the account is simply edited in place
- If the transaction’s dependency is newer, then the state is edited and the dependency is updated to equal the newer dependency, but an “alternate state

” is created with the yes dependency equaling that of the old state, the no dependency list being that of the no state plus the dependency of the transaction, and the state being the same as the old state.

To give an analogy, suppose there is an account Alice with state “50 coins if the monkey is at least 4 feet tall but the bear is not European”, and we receive a transaction that gives Alice 10 coins with a dependency “the monkey is at least 5 feet tall”. The new state of the account becomes “60 coins if the monkey is at least 5 feet tall but the bear is not European” but we add an alternate state “50 coins if the monkey is at least 4 feet tall but the monkey is less than 5 feet tall and the bear is not European”.

- In the case that the main state is no longer active due to a reverted expected_roots

entry, there is a reorg

operation that can move an alternate state that is active into the “main state” position and the main state into an alternate state position.

For example, if it turns out that the monkey in the previous example is actually 4.5 feet tall, the newly created alternate state above in which Alice still has 50 coins can be reorged into the main state position and it would be active.

Example

To illustrate the above with another example, suppose there was an account Bob with 50 coins with a yes dependency (1, DEF) and no no dependencies (write this as {yes: (1, DEF), no: [], balance: 50}

, and suppose the expected_roots

list equals [ABC, DEF]. Now:

- A transaction comes in with dependency (1, DEF) sending Bob 10 coins. All that happens here is that Bob’s state becomes {yes: (1, DEF), no: [], balance: 60}
- A transaction comes in with dependency (2, MOO) sending Bob 10 coins. This gets rejected because (2, MOO)

is not in the expected_roots

- The expected_roots

extends to [ABC, DEF, GHI]

. A transaction comes in with dependency (2, GHI) sending Bob 20 coins. Bob’s state becomes {yes: (2, GHI), no: [], balance: 80}

but we add an alt-state {yes: (1, DEF), no: [(2, GHI)], balance: 60}

- (2, GHI)

gets reverted, and (2, GHJ)

gets added. Bob’s state is no longer “active”.

- Suppose (3, KLM)

gets added, and Charlie receives 1 coin. Charlie’s state is now {yes: (3, KLM), no: [], balance: 5}

- Suppose (4, NOP)

then gets added.

- If Bob wants to send 5 coins to Charlie, Bob can send a transaction to perform a reorg, which sets his state to {yes: (1, DEF), no: [(2, GHI)], balance: 60}

and his alt-states to [{yes: (2, GHI), no: [], balance: 80}]

, and then send a transaction with dependency (3, KLM)

(he could also make the dependency (4, NOP)

but this is not necessary). Now, Bob’s state becomes {yes: (3, KLM), no: [(2, GHI)], balance: 55}

with alt-states [{yes: (2, GHI), no: [], balance: 80}], {yes: (1, DEF), no: [(3, KLM)], balance: 60}]

, and Charlie’s state becomes {yes: (3, KLM), no: [], balance: 5}]

The above is a simplification; in practice, nonces and used-receipt bitfields also need to be put into these conditional data structures. Particularly, note that if transactions’ effects get reverted due to expected_roots

entries being reverted, the transactions can usually be replayed.

Implementation

Here is an implementation in python of a basic version dealing only with balance transfers:

github.com

ethereum/research/blob/f8470ef07ad179968bedcbefbe820eb970aac786/fast_cross_shard_execution/optimistic_dependency_test.py

import copy, os, random, binascii

zero_hash = '00000000'

def new_hash(): return binascii.hexlify(os.urandom(4)).decode('utf-8')

Account state record

class Account(): def **init**(self, yes_dep, no_deps, balance): # This state is conditional on this dependency being CORRECT self.yes_dep = yes_dep # This state is conditional on these dependencies being INCORRECT self.no_deps = no_deps # Account balance self.balance = balance

```
def __repr__(self):
    return "[yes_dep: %r, no_deps: %r, balance: %d]" % (self.yes_dep, self.no_deps, self.balance)
```

This file has been truncated. show original

Generalizations

- This technique can likely be extended to synchronous cross shard calls, though the challenges there are greater.
- This technique can also be used for fast inter-blockchain communication.
- This technique can be used to handle forms of delayed verification other than those at consensus layer. For example, if there is a prediction market on whether or not Bob won the election, and a community of users agrees that Bob did win the election, but the on-chain resolution game is delayed by 3 weeks for security reasons, then this technique can be used to allow “ETH if Bob wins the election” tokens to be used as ether within that community even before the consensus layer finalizes the operation.
- More granular expected root structures can be considered to reduce the inherent “fragility” in this mechanism where if even one shard reorgs, the expected global state root changes and so almost everything gets reverted.