Prerequisite: Delayed state execution, finality and cross-chain operations

If we are to go with a model where state execution is a separate process that trails behind collation selection, then the next question is: how do

clients figure out what the state is? Super-full nodes can always execute every collation, but every other type of node needs some kind of light client protocol.

Here is one simple proposal. Allow anyone with ETH in any shard to deposit their ETH (with a 4 month lockup period), and at certain points (eg. once every Casper epoch) give depositors the ability to make claims about the state at some given height. These claims can be published into the blockchain. The claims would be of the form [height, shard, state_root, signature]

. From the point of view of a node executing the state, a correct claim is given some reward proportional to the deposit (eg. corresponding to an interest rate of 5%), and a false claim means the claimer is penalized.

This is the simple “cryptoeconomic light client” design; anyone who sees a claim knows that either (i) the claim is true, or (ii) in the real state

, the depositor that made the false claim loses their deposit. It could be potentially used as a light client design even for Ethereum 1.0 today, though with a layer of indirection as the EVM only has access to block hashes, and so the state root would need to be confirmed by including the block header, checking its hash, and then extracting the state root.

However, when used with sharding it has a flaw: if too few nodes are super-full nodes (ie. nodes with O(c2

) computing power that process the entire chain), then if a 51% attack on the state calculation layer takes place (which is quite feasible, since every shard has separate state calculation deposits), then it may not matter

what happens to the deposits in the real state, because no one will know what the real state is (or at least there will be so few actors that do that they could collude to pretend it’s something other than what it actually is).

Interactive verification

Hence, what we actually need is something like an interactive verification protocol, where if any substantial disagreement at any particular point takes place the client calculates the state transition at that point themselves, so that even in the presence of a 90% attack on any single shard a client can still figure out the correct state. The following is a proposal for doing that.

We will start off by making a series of changes to the above cryptoeconomic light client protocol that we introduced above. First, we switch from claims being [height, shard, state_root, signature]

to [height, collation_hash, shard, state_root, signature]

. A claim that specifies the wrong collation_hash for a given height does not receive any reward or penalty; only a claim with the correct hash

but the wrong state root

is penalized. This has three effects:

1. It means that it’s the client’s responsibility to determine what the correct chain is. The chain of reasoning for a client would be “my chain selection process tells me that the collation at height H is 0x1234, and I have 357000 ETH worth of claims showing that if the collation at height H is 0x1234 then the state root is 0x5678, and so I believe that the state root at height H is 0x5678”.

2. It means that state executors do not have to assume “reorg risk”; because their claims are now conditional on a particular history, and state execution is a deterministic process, they can only lose money if they deliberately provide the wrong state root.

3. As a consequence of (2), it becomes safe to have very harsh penalties for claimers giving bad answers. Specifically, we can make the penalty be the entire deposit, or follow the Casper approach where if the portion of claimers that make incorrect claims is p, then the claimer loses portion 12 * p of their deposit, or 100% if p >= 1/12.

Now, we change the claims to [height, shard, prev_state_root, collation_hash, post_state)_root, signature]

- making claims also dependent on the previous

state root. Clients can then make estimates of the state by treating claims as a graph, where states are vertices, collation hashes are edges and the total deposit size of claims is the weight; to evaluate the correct state, clients would start at the genesis, with the head state being the genesis state, and then follow a GHOST-like protocol where at each stage, clients select the child state root with the most claims staked on it, conditional on the prev state root being the current head state.

That is, clients would follow a chain of reasoning like:

- I know the pre-state root at height 0 is 0x1234 because that's the genesis state, which is the protocol params
- I know the block hashes at heights 0, 1 and 3 are 0xabcd, 0xabce and 0xabcf.
- There's 520000 ETH staked on the claim that if the pre-state root is 0x1234 and the block hash at height 0 is 0xabcd, then the state root at height 0 is 0x5678
- There's 437000 ETH staked on the claim that if the post-state root at height 0 is 0x5678, and the block hash at height 1 is 0xabce, then the post-state root at height 1 is 0x9012.
- There's 619000 ETH staked on the claim that if the post-state root at height 1 is 0x9012, and the block hash at height 2 is 0xabcf, then the post-state root at height 2 is 0x3456.
- Hence, I know that either (i) the post-state root at height 2 actually is 0x3456, or (ii) at least 437000 ETH will be lost.

The purpose of this is to make state claims "bite-sized", in the sense that the correctness of any individual claim can be evaluated by re-checking the execution of only one block.

We now add the interactivity. If a client sees that, for some given (shard, prev state root, collation hash) there are two or more conflicting answers being given, with more than some threshold if disagreement (say, 10%), it doesn't believe any

claim until it receives a proof of execution - the full collation data plus witnesses, which the client can re-execute to determine what the actual

post-state root of the execution, conditional on the prev state, is. The client software can treat the proof of execution as being like a conditional bet, but with a weight of infinity.

Cross-shard communication

We now add asynchronous cross shard communication. Now, the execution of a collation depends not just on the prev state and the collation, but also on receipts from CROSS_SHARD_DELAY (eg. 5) blocks ago. Receipts are generated by execution, and so we can simplify by assuming that receipts are part of the state. Hence, calculating the state of any shard at height H requires knowing the state root of every shard at height H-5. In order to preserve the "zero external risk" property of honest state execution claiming, we thus need to make the claims be conditional on the prior state of every

shard. We do this by creating an implicit meta_state_root

, which is the root hash of a Merkle tree of all the state roots; the claims become [height, shard, prev_state_root, prior_meta_state_root, collation_hash, post_state_root, signature]

.

Here is a sample code for how a client could run in this paradigm:

from ethereum.utils import sha3 NUM_SHARDS = 64 CROSS_SHARD_DELAY = 5 GENESIS_STATE = b'\x00' * 32

Database mock

class EphemDB(): self.kv = {}

```
def get(self, k):
    assert isinstance(k, bytes)
    return self.kv[k] if k in self.kv else None

def put(self, k, v):
    assert isinstance(k, bytes)
    if isinstance(v, int): v = str(v)
    if ininstance(v, str): v = v.encode('utf-8')
    self.kv[k] = v
```

Set the block hash of the given block height and shard

def set_blockindex(db, height, shard, blkhash): # Put hash db.put(b'block:%d-%d' % (height, shard), blkhash) # Set max score calc *for this shard* existing_max = int(db.get(b'max_score_calc:%d' % shard)) db.put(b'max_score_calc:%d' % shard, min(existing_max, height - 1)) # Set max height existing_max = int(db.get(b'max_height:%d' % shard)) db.set(b'max_height:%d' % shard, max(existing_max, height))

Get the block hash of the given block height and shard

def get_blockindex(db, height, shard): return db.get(b'block:%d-%d' % (height, shard))

Get the meta state root

def get_metastate_root(db, height): return db.get(b'merkle:%d-%d' % (height, 1))

Add a bet of the form prev state, blockhash, root -> post_state

with some given size

Note: an execution proof is a bet of infinite size

def add_bet(prev_state, blockhash, root, shard, post_state, size): # Add to the score for the post-state in this context bethash = sha3(prev_state + blockhash + root + post_state) cur_score = int(db.get(b'score:' + bethash)) or 0 db.put(b'score:' + bethash, cur_score + size) # Check the max score for this context; if the score for # the given post-state is higher, change the child pointer position_hash = sha3(prev_state + blockhash + root) cur_max_score = int(db.get(b'score:' + position_hash)) or 0 if cur_score + size > cur_max_score: db.put(b'score:' + position_hash, cur_score + size) db.put(b'winner:' + position_hash, post_state) existing_max = int(db.get(b'max_score_calc:%d' % shard)) db.put(b'max_score_calc:%d' % shard, min(existing_max, height + CROSS_SHARD_DELAY))

Assuming the state for height H-1 has been estimated, estimates

the state for height H

def calc_candidate_child(height, shard): prev_state = db.get(b'state:%d-%d' % (height - 1, shard)) \ if height else GENESIS_STATE blockhash = get_blockindex(height, shard) root = get_root(height - CROSS_SHARD_DELAY) \ if height >= CROSS_SHARD_DELAY else b'\x00' * 32 position_hash = sha3(prev_state + blockhash + root) old_state = db.get(b'state:%d-%d' % (height, shard)) new_state = db.get(b'winner:' + position_hash) if old_state != new_state: db.put(b'state:%d-%d' % (height, shard), new_state) db.put(b'merkle:%d-%d' % (height, NUM_SHARDS + shard), new_state) # Update a Merkle tree of state roots to compute the meta-state root i = (NUM_SHARDS + shard) // 2 while i: L = db.get(b'merkle:%d-%d' % (height, i*2)) or b'\x00' * 32 R = db.get(b'merkle:%d-%d' % (height, i*2+1)) or b'\x00' * 32 db.set(b'merkle:%d-%d' % (height, i), sha3(L + R)) i //= 2 # Annul all scores more than CROSS_SHARD_DELAY after this height existing_max = int(db.get(b'max_score_calc')) db.put(b'max_score_calc', min(existing_max, height + CROSS_SHARD_DELAY))

Recalculates state for the given shard for the appropriate range

def recalc_states(shard): range_start = min(int(db.get(b'max_score_calc')), int(db.get(b'max_score_calc:%d' % shard))) range_end = db.get(b'max_height:%d' % shard) for h in range(range_start, range_end): calc_candidate_child(h, shard) db.set(b'max_score_calc:%d' % shard, range_end)

2018.03.12 addendum:

The protocol as written may be inefficient, because if there is a very large number of executors on each shard, then there will be a very large number of messages that the chain needs to process every block. We can reduce this inefficiency in two ways:

1. Randomly sample, so only a specific 1% (or whatever other percentage; perhaps a fixed number, like 50) of validators are allowed to make a claim during each block. Have clients execute the collation themselves if >10% of the validators that vote disagree with the majority.

2. Have individual claims be over a few consecutive epochs, rather than a single epoch; this allows signatures to be reused.

Note that because claims are included in the chain as raw signed objects, a client can get the complete set of claims by just scanning the chain; additionally, note that a client can get claims about the state of block B from inside blocks that are descendants of B, even without knowing the state of those blocks yet, because the client can find the claims simply by scanning the chain.