Ethereum 2.0 Mauve Paper

Because mauve has the most RAM...

Over the past decade, projects such as Bitcoin, Namecoin and Ethereum have shown the power of cryptoeconomic consensus networks to bring about the next stage in evolution of decentralized systems, potentially expanding their scope from simply providing for the storage of data and messaging services to managing the "back-end" of arbitrary stateful applications. Proposed and implemented applications for such systems range from globally accessible cheap payment systems to financial contracts, prediction markets, registration of identity and real world property ownership, building more secure certificate authority systems and even keeping track of the movement of manufactured goods through the supply chain.

However, there remain serious efficiency concerns with the technological underpinnings of such systems. Because every full node in the network must maintain the entire state of the system and process every transaction, the network can never be more powerful than a single computer. The consensus mechanism most often used in existing systems, proof of work, consumes a very large amount of electricity in order to operate; the largest instance of such a consensus, Bitcoin, in fact has been shown to consume as much electricity as the entire country of Ireland.

The leading solution to these problems is a consensus algorithm based on the combination of proof of stake and sharding. Proof of stake can be thought of as a kind of "virtual mining": whereas in proof of work, users can spend real-world dollars to buy real computers which expend electricity and stochastically produce blocks at a rate roughly proportional to the cost expended, in proof of stake, users spend real-world dollars to buy virtual coins inside the system, and then use an in-protocol mechanism to convert the virtual coins into virtual computers, which are simulated by the protocol to stochastically produce blocks at a rate roughly proportional to the cost expanded - replicating the exact same effect but without the electricity consumption. Sharding is a scalability solution in which nodes from a global validator set (in our case created through proof of stake bonding) are randomly assigned to specific "shards", where each shard processes transactions in different parts of the state in parallel.

Constants and Targets

We set (or target):

- `NUM_THREADS`: 80
- `BLOCK_TIME`: 4 seconds (aiming on the less ambitious side to reduce overhead)
- `EPOCH_LENGTH`: 12 hours, ie. 10800 "ticks"
- `NUM_VALIDATORS`: `NUM_THREADS * 50`, ie. ~4000
- `ASYNC_DELAY`: 1 minute, ie. 15 "ticks"

Threads

We consider the system as consisting of `NUM_THREADS` threads, where each thread behaves like a proof-of-stake blockchain with parameters in the spirit of this and this document. Blocks in each thread do not contain a state root (although they do contain a randao seed); they simply point to the previous block in that thread. However, there is a key difference between this design and the single-threaded proof of stake chain such as NXT and DPOS: a special thread, thread 0, contains the `CASPER` contract which contains a global validator set, and the next validator to create a block on any thread can be computed from (i) the randao seed in the previous seed, (ii) the `CASPER` state from block number `BLKNUMER - (BLKNUMBER % EPOCH_LENGTH) - EPOCH_LENGTH` (ie. the start of the previous epoch). The `CASPER` contract should contain logic for bonding, unbonding, etc.

The way that validators are computed is as follows. Suppose that `CASPER` contains `NUM_VALIDATORS` validators, all with equal deposit sizes (for simplicity). At the start of each epoch, we use a randao to determine a global seed `RNG_OUTPUT`. We randomly select `VALIDATORS_PER_THREAD = NUM_VALIDATORS / NUM_THREADS` validators for a given thread `thread_id` by taking `sha3(RNG_OUTPUT, thread_id, j) % NUM_VALIDATORS` for `0 <= j < VALIDATORS_PER_THREAD`. We then determine the specific validator for a block in thread `thread_id` with randao

`LOCAL_RNG_OUTPUT` by taking `sha3(RNG_OUTPUT, thread_id, LOCAL_RNG_OUTPUT % VALIDATORS_PER_THREAD)`. The purpose of this two-layered scheme is to select a subset of validators for each thread during each epoch, and allow this subset to remain for that epoch but be reshuffled every epoch.

Note that the above alone is sufficient to specify a scalable blockchain protocol. However, it is missing two important components: light client support, and cross-shard communication. The two are related: if it were not for the need for cross-shard communication, we could just bring back light client support by making every block contain a state root. However, for safety reasons we do not do this, and instead have a separate state root mechanism.

Light clients and the Finality Cycle

Inside of the `CASPER` contract in shard 0, we add a finality cycle. However, instead of doing this just for one thread, we allow bets on the state roots of all threads. In every place where a 32-byte state root was required, an array of `32 * NUM_THREADS` bytes representing `NUM_THREADS` hashes is used instead, and rewards and penalties are separately computed for each thread. Note that rewards and penalties should not be computed instantly; instead, the data should be saved, and rewards computed in the epoch after the current epoch.

The load on shard 0 will be as follows:

- `32 * NUM_THREADS` bytes of data every `BLOCK_TIME` seconds of bandwidth (ie. **640** bytes per second under above parameters)
- `32 * NUM_THREADS * EPOCH_LENGTH` bytes of data in the state (ie. **43.2 million** bytes under above parameters)

Note that the process of actually computing rewards can be spread out across blocks, not done all at once, to prevent the system from being overloaded with spikes of activity. Note also that EIP 97 is crucial for making this work efficiently.

Note that there is a simple strategy for clients willing to store and process every thread to act in this context: they can compute up-to-date state roots of every single thread. However, most clients will not be willing to do this; instead, they will be full clients on a few threads and light clients on most other threads. In this case, they can simply rely on other validators' judgements; if absolutely everyone does this and this only, then the betting cycle will essentially be a SchellingCoin. This also works and provides medium security. A stronger strategy would be to use light clients plus proofs of invalidity: if too many people complain that invalid state roots are being produced for a particular thread at some particular time (this can be done by checking for, say, a >10% disagreement threshold, or some social scheme, or a combination of both), then and only then the client switches to full validation for that shard for that time slot.

When joining a shard, validators need to download the state from the shard; this can be done from the state root, much like fast syncing works today. Alternatively, light clients may download just-in-time, downloading commonly accessed state first, and then downloading more rarely used contracts only if they prove to be needed to execute a particular transaction.

Cross-shard communication

Cross-shard communication works as follows. We create a `ETHLOG` opcode (with two arguments `to`, `value`), which creates a log whose stored topic is the empty string (note that's the empty string, not 32 zero bytes, a traditional log cannot do this), and whose data is a 64-byte string containing the destination and the value. We also create an opcode `PREPARELOG`, which takes as an argument a shard, a block number and a log ID number (ie. the number of logs in that block before that block, so the first log has id 0, the second has id 1, etc), and saves a record in the storage saying that this log is being requested and during what block. Finally, we create an opcode `GETLOG`, which gets a log which has been prepared at least `ASYNC_DELAY` blocks ago, stores a record in the state saying that the log has been consumed, and places the log data into a target array. If the log has the empty string as a topic, this also transfers the ether to the recipient.

Note that importing a proof from one shard into the other shard is not required; the light clients are expected to do this individually. The async delay reduces the likelihood that a reorganization in one shard will require

an intensive reorganization of the entire state, and the requirement to prepare ensures that light clients have time to download the receipt from the other shard, so that clients that are full clients on some shards and light clients on other shards can still process the state with full efficiency without waiting for synchronous network requests in the middle of a round of transaction processing.

Acknowledgements

Special thanks to:

- Gavin Wood, for the "Chain Fibers" proposal
- Vlad Zamfir, for ongoing research into proof of stake and sharding, and particuarly the realization that state computation can be separated from block production
- Martin Becze, for ongoing research and consultation