made in TXRX research team

It's a reply to @pipermerriam's Explorations into using DHT+SkipGraph for chain and state data retrieval. Piper has been gradually researching this topic starting with The Data Availability Problem under Stateless Ethereum post and several other related articles. Thanks for pushing it forward! There are several issues connected with sharded state and routing (lookup and discovery of current state and history objects) is one of them and should be solved by Discovery protocol. However, I want to oppose the way it's attempted to be answered.

Why it's important

Current state of Ethereum is already huge and continues to grow. Even non-archive nodes require 300Gb of space to be in sync and this number is getting bigger and bigger every day. Though Ethereum 2.0 is going to solve part of the problems with its shards, Ethereum 1.0 is going to be single shard in it and the accumulated user base will be the guarantee of "Shard 0" further expanding. Even if other shards will keep small size for a while after creation, the future is the same for all: shard size is very big to keep it's whole state and historical data on disk by regular users.

Yes, we could move to the point where core blockchain p2p consists only of full nodes maintained by Consensys, Infura, Ethereum Foundation and few other well known authorities in order to back their services, but it's not a way blockchain should look like. Decentralization is the crucial feature of Ethereum and all other non-private blockchains. When you require to have free 300Gb (and growing) SSD drive space and an hour to reach a recent state after overnight of downtime, you restrict nodes to professional members. And the more you raise the bar, the bigger share of professional members you have in the network.

We are not using 3rd party service to download torrent, because torrents are p2p network, and you just start torrent client and use it directly, but we are using 3rd party services to send Ethereum transaction with some middleware and decreased security of the underlying layer, because client requirements are overwhelming for our laptops. And SP (State Providers) doesn't solve this issue, because SP are professional members who need proper hardware to run service and expect return of their investments in hardware and bandwidth. Which means its deeper move to the blockchain core consisting of only professional members. Yeah, it's better than having several big companies keeping state, because it could attract smaller funds, but it's already not the kind of decentralization we may expect from a public blockchain.

What could be wrong with stateless

The ability to make clients with limited hardware capabilities be full-fledged members of a network is not only a handy UX feature, but it's a base decentralization requirement. We already have just 7,000 peers in the Ethereum Mainnet network while torrent clients are used by dozens of millions users without middleware. And it's not disputed that the number of active Ethereum users is way bigger than 7,000.

While Ethereum 2.0 offers a sharding approach, state network API is not yet created and some goals are challenging and not solved yet, as we want validators to be stateless. Even on this side, we still don't have a history and state storage scaling solution within one shard. And as we expect that Ethereum 1.0 is going to be one single shard we are still with the same issue: only professional members could be able to follow it due to storage and bandwidth requirements. Though Ethereum 2.0 makes historical data needed only for professional members and not needed for the sync with introduction of weak subjectivity, the current state is still valuable for all.

Here we come to the idea of partial state clients. They are not explicitly defined when we split members to stateful or full nodes (part of them could be State Providers) and stateless clients, they are part of stateless clients in this classification, but they could be a core part of the network without state providers in absence of functioning incentivization solution. If we get millions of peers like in torrents, they could serve another millions of fully stateless clients without concerns, improve decentralization and give users a better UX.

What could be wrong with Skip Graph?

Client recommendation to use SSD is motivated by size of the database, big number of keys accessed during block import or fast sync, when preparing answers for DevP2P network queries serving as full node or via RPC. While local key retrieval costs dozens of microseconds, any network roundtrip is measured in dozens or hundreds of milliseconds, meaning 1000-10000 times difference. Which leads to a thought that while Skip Graphs could be on the best side of O

complexity notation with O(log n)

lookups, it could not maintain standard client use cases mentioned above. Same could be relevant to techniques like Beam Sync.

Solving a task of distributed state storage we have some similarities with DHT data storage systems and IPFS, but, in fact we have one piece of data and we want to split it among a number of participants, which is a big difference from original DHT and IPFS goals. We don't have different files, any files submitted by users, instead we have one canonical history+state, it's big and growing and we want fastest access to any part of it. And if we ignore this fact we are going to get a less efficient system than it could be.

Both Skip Graph

and Beam Sync

require a number of lookups of the same order like local lookup of a full node in its database while increasing time of each lookup 1000x-10000x

folds due to network nature. What could be done to significantly improve this order of things?

- we could move part of lookup to the local storage and it’s crucial, what part of lookup could be performed locally and what is done in the network. Ideally we want to get O(1)

for the network part. It’s ok even if local part complexity has grown with it compared to network only alternatives

- we should reduce dependency of network lookups which could lead to greater parallelism and batch lookups of any size

Another issue with any DHT based approach is uniform distribution, which leads to absence of popularity and size balance features, so any key/value pair, no matter of its size and popularity, has the same presence in the network, and any attempts to change this leads to high complexity of the underlying protocol. It’s obviously an issue, when we have parts of unpopular shards on one set of nodes, and something like CryptoKitties on another. We think that balance issues should not be a part of protocol due to its complexity and fuzzy logic, instead it should be part of the UX of the client, and it could advise the client what part of state to choose to get desired participation in network life. Protocol could motivate clients to choose more popular parts of data by side feature with distributed DL/UL ratio or other similar mechanics, but core protocol should work even if each user picks a random part of state. Naturally users want to pick the pieces they need, and they cannot service a certain piece of data without incentivization.

Simplified approach

Let’s address all types of objects state and historical data consists of:

- headers
- blocks
- transactions (part of block)
- transaction receipts
- witnesses
- state data

Looking at historical data where we could clearly define canonical chain, especially after being a shard of Ethereum 2.0 with finalization. Headers, blocks, transactions, transaction receipts and some types of witnesses could be referenced by block index number. There are two benefits from it:

- it doesn’t change with time, which means long running ads
- we don’t need dependant queries to download any missed part of state

The same properties could also empower state data if it’s organized in aSparse Merkle Tree where index is account address, state parts could be referenced by account ranges it includes, which translates to appropriate sub-tree. So instead of searching this data by hashes and traversing state by tree which is changing on each block, we expect users to follow current chain and state, we need a way to quickly verify it and we need a way to heal outdated parts after batch downloads.

Let’s solve it for a simple case to show it’s possible: split state to equal 64 parts, so any partial stateless client could store any of these 64 parts, one or several. We create one with indexes being Ethereum account addresses and values Ethereum accounts. And 1/64th means subtree with accounts from #00

- **03**

, 2/64th #04-

07

etc. In a standard Sparse Merkle Tree we could be sure that such subtrees will not migrate, are going to include accounts we want and will not include anything else. We skip how we could encourage block producers or other members to gossip witness data, of course, it's another side of the issue, but, say, it's solved, so, we have gossip with recent blocks and standard witnesses, roots for these 64 subtrees. When 3/64th of the world state is advertised it's not advertised by any state root, it's changing too often. Instead, it's advertised as 3/64th or by appropriate generalized index and we expect that it has recent state. Following the gossip and asking for 2 children from the root of subtree from such a node we could easily and fast clarify the freshness of its state part before downloading part of it.

[

732×720 7.82 KB

](https://ethresear.ch/uploads/default/original/2X/0/052cb5138cf00c813c8982f42e2ce015085023b1.png)

Main downside using Sparse Merkle Trees are their properties: speed, required storage size, witness size.@protolambda did a great job by gathering current research made of these kind of trees together here, and though caching could improve lookup in standard Sparse Merkle Tree speed up to log(n)

(see Efficient Sparse Merkle TreesCaching Strategies and Secure (Non-)Membership Proofs by Rasmus Dahlberg, Tobias Pulls, and Roel Peeters), Compact Sparse Merkle Trees has the smaller size and smaller proofs (seeCompact Sparse Merkle Trees by Faraz Haider). And when we are going to take into account all these properties, standard Sparse Merkle Trees are not the best candidates for state storage. But they guarantee stability of node positions, it's what we need in our approach: we get stable ad for our piece of data.

We are not changing the set of stored data very often. Yes, if it's a state we transition our part to the latest one (how to do it is not a part of this write-up), but it remains to be the same part of Sparse Merkle Tree. We could drop the part we are following or add a new one, this will change our ad, but current state changes should not modify it, it's a key idea. So we are going to advertise it by account address range as part of Sparse Merkle Tree, and it's possible to advertise state data in such a manner.

While splitting everything to 64 parts could be easily discovered and managed with ENR attributes, like it's done in Ethereum 2.0 with subnetworks, it's limited to 300 bytes (actually attributes part is about 150 bytes maximum), and advertisement of complex cuts are not so obvious. But 1/64 doesn't scale well, the state will grow with time, we may need to jump to 1/96 one day. Or if we talk about Ethereum 2.0, we may want to store parts from several shards on one node. And users are not going to host exactly 1/64. They should be either incentivized for it or be more flexible when choosing parts to host. They should be ok with sharing storage, but they need more freedom. What could help?

Adaptive Range Filters

We propose a Discovery advertisement system with 24 hours ad time. Ad is dropped only if peer goes offline, otherwise it's kept alive and only nodes which online/offline status is actively tracked by peer serving as media could advertise, so it's up to 200 ads on peer limited by 10 Kb size each, 2Mb of advertisements on each peer at maximum. It could be RLP with something like [headers bloom filter, blocks bloom filter, block with transactions bloom filter, block with receipts bloom filter, state account address ranges bloom filter]

, blocks are by index numbers in canonical chain, addresses are typical Ethereum addresses, but we need some approach to put ranges in bloom filter. And this approach exists: Adaptive Range Filters (see Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia by Karolina Alexiou, Donald Kossmann, Per-Ake Larson)

Bloom filters are a great technique to test whether a key is not in a set of keys or is very likely a part of a set. In a nutshell, Adapted Range Filters are for range queries what Bloom filters are for point queries. That is, an ARF can determine whether a set of keys does not contain any keys that are part of a specific range, or very likely has it.

Finally, peer sync strategy looks like this: download a set of Adaptive Range Filters for a big number of different peers, say 1,000, total 10Mb of download and build a set of range filter trees in memory. ARF support range queries, plus lookups could be easily parallelized to all these 1,000 trees. Likely for any piece of data it will find several locations, say, 10 peers, and could be further improved with techniques like swarming data transfer or similar, reaching a speed and efficiency of torrents. Next, needed pieces of data are downloaded from peers.

Also we add an endpoint to perform a search without downloading all ads, which locally performs a search in all adaptive range filter ads with low priority. Yeah, we still don't have fully featured stateless API without tx hash to block resolving and other methods but we got structure where partially state clients could be already useful for network, return proportional to get and fulfil some stateless clients queries. Also it could be a checkpoint to more functional stateless API without incentivization and State Providers.

What's next

We are going to make simulation to ensure Adaptive Range Filter performance and ability to solve current and future Eth1.x tasks:

- moving load from full nodes, we expect that full nodes will participate only in gossip after all

- ability of network with big share of partial state nodes to handle imbalances of storage allocation and popularity
- ability of network with big share of partial state nodes to execute all real world task for different type of clients
- ability of full nodes to recover from partial state nodes after all full nodes are down
- provide viable state and historical data store balancing strategy built over non-fuzzy stimulus