The Ethereum Data Availability Problem

This document covers what I have been thinking about as the data availability problem

. I will cover the basics of how data flows around the Ethereum network today, and why the network continues to function despite having what appears to be a fundamental design flaw. This flaw, lies in the combination of a basic network level reliance on the availability of full nodes from which data can be retrieved, combined with exposing APIs for on-demand state retrieval which can be used to create stateless clients. I assert that stateless clients will naturally trend towards the majority on the network, eventually degrading network performance for all nodes.

After establishing a concrete definition of the problem, I then cover some solutions that are unlikely to work, followed by a proposal which is intended to be a starting point from which we could iterate towards a proper solution.

Data availability in the current network.

In the current state of the Ethereum mainnet we have a single type of client, the full node.

We ignore the LES

protocol for simplicity, but it is covered a bit later in this document.

In the current network, all ETH

nodes are assumed to operate as full nodes and to make available the entirety of the chain history (headers, blocks & receipts) and the state data (accounts from the account trie and the individual contract storage tries).

The only mechanism available for a node to adverstise what data it makes available is the [Status

](https://github.com/ethereum/devp2p/blob/e54f53081a305c36a9251f14d77620462a657db0/caps/eth.md#status-0x00) message which supports advertising your best_hash

. This value indicates the latest block the client has. In theory, a client should not be asked for data that occurs later in the chain than the block referenced by best_hash

since that client has signaled that their view of the chain does not include that data.

The ETH

protocol is very forgiving allowing empty responses for the majority of data requests. Typically a client which does not have a piece of data will respond with an empty response when asked for data they do not have.

The current most common mechanism for a node to join the network is using “fast sync”. This involves fetching the full chain of headers, blocks, and receipts, and then directly pulling the entirety of the state trie for a recent block.

A client who is in the process of “fast” syncing must choose how to represent themselves on the network while their sync is in progress.

If Alice chooses to advertise to Bob that her best_hash

is the current head of the header chain she has synced, then Bob will not be able to know Alice is missing the state data for that block since she has only partically synced the chain state. This means when Bob sends Alice a [GetNodeData

](https://github.com/ethereum/devp2p/blob/e54f53081a305c36a9251f14d77620462a657db0/caps/eth.md#getnodedata-0x0d) request, Alice is unlikely to be able to provide a complete response.

If Alice instead decides to be conservative and wait until she finishes syncing, then she would choose to advertise to Bob that her best_hash

is the genesis hash. In this model, Bob’s view is that Alice is stuck at the genesis of the chain and has no chain data. In this case it is likely that Bob may disconnect from Alice, deeming her a “useless peer”, or simply never issue her any requests since Bob may assume she has none of the chain data.

These two cases are meant to demonstrate the lack of expressivity in this protocol. In reality, clients seem to have chosen to use slighly more advanced heuristics to determine whether to issue a request for data to a connected peer. The lack of expressivity of best_hash

results in it largely only being usable as an imperfect hint as to what data a peer might

be capable of providing.

With the current network dominated by Geth and Parity, this has not proven to be incredibly problematic. According to ethernodes.org

the mainnet network contains 8,125 nodes, 26% of which are in the process of syncing, and 74% which are fully synced.

So, despite not being able to reliably identify which nodes should have which data, in practice nodes can sync quite reliably. I attribute this to there being a majority of nodes which have all of the data meaning that simply randomly issuing requests to connected peers has a high probability of success.

A Partial Problem statement

It is my assertion that the reliability of this network is heavily dependent on the assumption that all nodes are either full nodes, or in the process of syncing to become a full node, with a very small minority of disfunctional nodes (such as a node that has malfunctioned and is no longer syncing and stuck at an old block). If the topology of the network were to change such that these assumptions were no longer correct, I believe that the process of syncing a full node will become increasingly difficult.

RESEARCH: play with the numbers to maybe provide some extra insight into how changing the distribution of full nodes, to nodes missing data would change sync times…

One way that this may come to pass is through broader adoption of the “beam” sync approach to client UX. A client that is syncing towards being a full node can be viewed as triggering some accumulation of “debt” on the network, as they are requesting more data than they are providing to the network. The network can handle some finite volume of this behavior. Under the assumption that all nodes eventually become full nodes, most nodes eventually pay back this network debt by providing data to other nodes.

Suppose a client implements beam sync without the background sync to fill in the full state. Such a client will continually take from the network by continually requesting data on-demand. If the population of the network is sufficiently dominated by such a client, then it would eventually surpass the overall networks ability to supply the demand created by these clients. It is unclear exactly how this would effect the network as a whole as well as individual clients, but my intuition is that both the offending beam syncing nodes and the network’s full nodes would suffer. The full nodes would be overwhelmed by the beam syncing node requests, preventing other nodes who are in the process of syncing from being able to sync. Similarly, the beam syncing nodes would suffer degraded performance as they are unable to find full nodes who are able to provide the necessary data.

- New nodes attempting to sync would have problems finding peers who have the data they need.
- The bad

“beam” syncing nodes would no longer function well because they would be unable to find nodes that have the needed data.

- Full nodes would be unable to serve the demand, likely degrading the performance of the node under the heavy request load.

My intuition is that this is a problem with the network architecture, rather than the nodes. While it is easy to view the “bad” beam syncing nodes as doing some irresponsible, we have chosen to operate on a permissionless network and thus we cannot reasonably expect all network participants to operate in the manner we deem appropriate. The only two options available to address this problem are to address it at the protocol level, or to attempt to address this at the client level. The client level fixes are likely to ultimately be a sort of “arms” race, so we’ll ignore that option and focus on the protocol aspect.

Enter “Stateless Ethereum”

“Beam” sync can be viewed as a highly inneficient stateless client. The mechanism by which it fetches state data on-demand via the GetNodeData

primative requires traversal of the state trie one node at a time. Naively this means issuing requests for single trie nodes to a peer as the client steps through EVM execution, encountering missing state data. Some basic optimizations like parrallel block processing and optimistic transaction lookahead processing provide the improvements that make this approach viable, but these improvements are tiny compared to proper “witnesses”.

The “Stateless Ethereum” roadmap is designed to modify the protocol to formally support stateless clients in the most efficient manner possible. This is likely to involve:

- Protocol changes to reduce witness sizes (binary tries, merklizing contrac tcode)
- Protocol changes to bound witness sizes (gas accounting for witness sizes)

Both of these improve the efficiencly of a stateless client, and thus should reduce the overall load that a stateless client places on the network. The best case for stateless client is small efficient witnesses for block verification. Alternatively any protocol that allows for syncing of the state also allows for on-demand construction of witnesses. Both of these approaches rely on full nodes to be data providers.

Current thinking is that a very small number of full nodes should be able to support providing witnesses to an arbitrarily large number of stateless nodes due to the ability to use gossip protocols to have stateless clients assist with the dissemination of witnesses. This is akin to bittorrent style swarming behavior.

However, for stateless clients to provide a useful user experience, they will need more than witnesses from the network. For stateless clients to be viable for a broad set of use cases, they need to be able to have reliable on-demand access to the full chain and state data.

Examples in the JSON-RPC APIs.

- eth_getBalance

: Needs proofs about account state

- eth_call

: Needs proofs about account state, contract state, and the contract bytecode.

- eth_getBlockByHash

: Request block and header data

- eth_getTransactionReceipt

: Request block, header, and receipt

While all of the data above can be requested via the ETH

devp2p protocol, it doesn’t seem like the full nodes on the network would be able to support the request load, not to mention the other forementioned negative effects that too many leeching peers would trigger.

Narrowing in towards a solution

At this stage you should have a decent picture of the problem. Stated concisely.

- Any protocol that supports syncing towards a full node appears to also support inneficient stateless clients
- A sufficiently high number of these stateless clients will overwhelm the full nodes.
- Any attempt to mitigate this at the client level is likely to be innefective against a motivated client developer since differentiating between a leeching node and a node which is in the process of a full sync is expected to be imperfect.
- Changes to the protocol to support stateless clients in the most efficient ways possible will only provide temporary mitigation.
- In the stateless paradigm we expect stateless nodes to far outnumber full nodes.

We cannot support a large population of stateless nodes on the current ETH

protocol, nor can we do it on an improved ETH

protocol that supports witnesses, and we also cannot prevent stateless clients from being created.

It is also worth noting that stateless clients are likely to have a better UX than the current status quo (until they crush the network and then none of the clients work very well)

Solutions that probably wont work

Asymetric protocol participation

LES

seems to demonstrate that an asymetric protocol does not scale without incentives. For such a network we’ll refer to the ones who have the data as the providers

and the ones who need the data as the consumers

. Networks with this provider/consumer hierarchy seems to have a natural limit at which an imbalance in the ratio of providers to consumers causes the providers to be overwhelmed by the consumers and inherently suffers from a tragedy of the commons problem.

Incentives

A commonly presented solution for the provider/consumer model is to introduce (economic) incentives. This approach is problematic as it places a high barrier to entry on node operators and is non trivial to bootstrap (requiring people acquire currency and fund an account before even being able to sync a node). Given that one of the goals is for our network to be easy to join, requiring node operators to fund their clients will likely have an adverse effect on adoption.

Treating this as a discovery problem

One way to view the problem is to treat it as a discovery problem.

I previously outlined that the current mainnet would suffer if the assumption that all nodes are full nodes were to be sufficiently wrong, resulting in clients having increased difficulty in finding the data they need.

One might try to fix this by improving the expressiveness of the client's ability to specify what data it makes available. My intuition for this model is that it could serve to bolster the current network, allowing the network to continue to operate to a higher threshold of imbalance between nodes that have the data and nodes that do not. However, such a mechanism would also serve as a way for nodes to discriminate against less statefull nodes, resulting in stateless nodes having similar problems to the current issuse LES

nodes face.

This problem would likely be exhaserbated since we cannot rely on nodes to be honest, allowing stateless nodes to mascerade as stateful nodes. If the scaling mechanism of the network relys on the accuracy of this data, then my intuition says that we end up in the same situation that the network exists in today, where clients rely on imprecise heuristics to determine the usefullness of clients, resulting in this broadcast information being nothing more than a potentially helpful hint that still must be verified.

Node type definitions

Lets take a moment to define the different node types more precisely.

All node types track the header chain but potentially might not keep the full history. Note that the Syncing node and the Stateless node have roughly the same requirements.

Full Nodes:

Defined as a node which has the full state for their head block and some amount of historical chain data.

Note this is explicitely changing the expectation that a full node does not necessarily have all blocks and receipts

- gossip of new headers/blocks/witnesses
- gossip of pending transactions
- on demain retrieval of blocks, headers, receipts that they do not actively track

Note that the block/header/receipt retrieval is not needed if the node keeps the full history.

Syncing node

Defined as a node which has an incomplete copy of state for their head block and some amount of historical chain data. This node type keeps data, building towards a full copy of the state, and a full copy of the historical chain data for the historical chain data they choose to retain.

- gossip of new headers/blocks/witnesses
- gossip of pending transactions
- retrieve blocks, headers, receipts on demand.
- retrieve state on demand
- retrieve contract code on demand

Stateless node

Defined as a node which does not retain a full copy of the state or the chain data. The extreme version of this is retaining none of the data.

- gossip of new headers/blocks/witnesses
- gossip of pending transactions
- retrieve blocks, headers, receipts on demand.
- retrieve state on demand
- retrieve contract code on demand

Deriving network topology from client needs and expected rational behavior

There are multiple participants in the network who have economic incentives to run full nodes.

- Miners get paid to produce blocks, and need the full state to do so.
- Centralized data providers like “infura” have a business model built around having this data available for parties willing to pay for it.
- Businesses building blockchain based products need infrastructure to connect to the blockchain. While not all use cases will require the full chain state, many will.

I don’t believe there is reason to be concerned about “losing” the state. However, it is reasonable to expect these nodes to be selfish

as this behavior will be necessary for self preservation. Full nodes will be unable to support the demand for data that a large number of stateless nodes would create. It is also reasonable to expect there to be fewer full nodes on the network as hardware requirements increase and less hardware intensive clients become available.

It is also important to note that from the perspective of a full node, differentiation between a node which is syncing towards becoming a full node and a stateless node mascerading as a syncing node is likely to be imprecise and an “arms race”. Thus, it seems reasonable to expect this need for a new way to access the chain and state data to be an issue for both stateless nodes and nodes which are syncing towards becoming a full node.

So, in this new network, we expect the topology to have the following properties.

- Full nodes restrict their resources primarily for other full nodes or potentially.
- All node types participate in the gossip protocols for new headers/blocks/witnesses

This has the following implications:

- all nodes should be interconnected for the gossip protocol.
- all nodes will need access to the chain data, but that full nodes are unlikely to be able to reliably provide it since some may choose to prune ancient chain cdata.
- both stateless and unsynced full nodes will need access to the state data.

A Possible Solution

What follows is the best idea I have to address the problems above. It is meant to be a starting off point and would require broad buy-in and much more research and development to be viable.

First, a protocol that is explicitely for gossip. All nodes would participate in this protocol as equals, assisting in gossiping the new chain data as it becomes available. We already have this embedded within the current DevP2P ETH

protocol, however in the current form it is not possible to participate in only this part of the protocol.

Second, a protocol where all nodes can participate as equals to assist in providing access to the full history of the chain data and the full state for recent blocks. While some nodes might have the full data set, most nodes would only have a piece of it (more on this below)

The term “protocol” is meant to be vauge. It could be a new DevP2P protocool, something on libp2p, etc. At this point I’m focused on defining the functionality we need after which we can bike shed over implementation details.

Distributed Storage Network of Ethereum Data

This is a concept to fill the need that all nodes will exhibit for reliable access to the full historical chain data and recent state data.

Data Types

This network will provide access to the following data, likely through something akin to the DevP2p ETH

protocols various GetThing/Thing

command pairs.

- Headers
- Block Bodies (uncles and transactions)
- Receipts
- Witnesses
- State Trie Data (both accounts and contract storage)
- Contract Code

In addition to this it may be beneficial if we can find ways to store the various reverse indices that clients typically construct (see section below about "extra functionality beyond the standard ETH protocol")

Basic Functionality Requirements

Finding Peers

Nodes need to be able to join the network and find peers that are participating in their chain. The recent "ForkID" seems to be the best candidate to efficiently categorize peers into subgroups where all nodes are operating on the same data set.

Data Ingress

The network needs a mechanism for new data to be made available. The forementioned gossip network is a provider of this data. A simple bridge between these networks might be adequate.

While it might be tempting to combine the gossip and state network, I believe this would be incorrect. A valid use case which needs gosip bup does not need any of the data retrieval APIs is a full node which also stores all of the historical chain data. Such a node has no need for the data retrieval APIs and should be allowed to limit participation to the gossip protocol.

Data Storage

I believe what I am describing below is just a DHT, maybe a special purpose one.

We need the network as a whole to store the entirety of the chain data and state data, while allowing individual network participants to only store a subset of this data. Nodes could either all store a fixed amount of data, or nodes could choose how much data they store. It is not yet clear what implications these choices might have.

There needs to be deterministic mechanism by which a node can determine "where" in the network a piece of data can be found. For example, if I need the block #1234

, a node should be able to determine with some likelihood which of the peers it is connected to is most likely to have that data. This implies some sort of function which maps each piece of data to a location in the network. Nodes would store data that is "close" to them in the network.

We may need some concept of radius, where a node stores and advertises the radius in which it stores data.

Nodes who are joining the network would also need a mechanism to "sync" the historical data for their portion of the network.

As new headers/blocks/receipts/witnesses propogate through the network as the chain progresses, nodes keep the ones that are their responsibility to store, and discard the rest.

Data Retrieval

One of the main differences to this network as opposed to the current DevP2P ETH

protocol is that we likely need routing. Data needs to be reliably retrievable, and we only expect any given node to have a subset of the data. For this reason, we can expect that there will be pieces of data that are not avialable from any of your connected peers, even for very well connected nodes in this network.

Routing is intended to make requests for data reliable. A node receiving a request for data that they do not have would forward that request on towards their peer(s) which are most likely to have the requested data, forwarding an eventual response back to the original requester.

Extra functionality beyond the standard ETH

protocol APIs

The following functionality is exposed by the standard JSON-RPC API, however, it requires a client to have a full picture of the chain history to serve responses. For example, the only way to retrieve transactions that are part of the canonical chain is to retrieve the block body of the block the transaction was included. The standard behavior of clients is to create a reverse index which allows clients to query the block number-or-hash for a given transaction hash which then allows retrieval of the transaction from the block body some clients may differ but the high level take away is the ETH

protocol has no mechanism for retrieving a transaction by hash.

- Ability to lookup the canonical block hash for a transaction referenced by the transaction hash.
- Ability to lookup the canonical block hash for a given block number.
- Ability to lookup the receipts for a given block hash

Network Design Principles

Homogenous

The networks should have a single homogenous node type. This should reduce asymetry between nodes.

- reduce incentives for nodes to be selfish by making all nodes usefull to all other nodes and making node behavior simple.
- reduce ability for nodes to discriminate by making all nodes behave in a similar manner.

Minimal hardware requirements

If we want broad participation then the hardware requirements for running a node should be as small as possible. I would propose using something like a raspberry pi as a possible baseline.

A starting point for hardware goals:

- ability to run the node with <500MB of available ram
- nodes can be ephemeral, operating purely from memory, persisting minimal data between runs, and not persisting any network data between runs.
- nodes can easily validate the content of the data they store against the header chain with a single CPU core.

Inherent load balancing

The network should contain a mix of full nodes, nodes with partial state, and nodes that only have ephemeral state. It would be ideal if the protocol could exhibit basic load balancing across the different participants.

For example, ephemeral nodes may be able to handle more of the routing, while full nodes can handle a larger quantity of the actual data retrieval.

Bittorrent swarming behavior when possible

We should aim for bittorrent style swarming behavior whenever possible. Some of this can be accomplished simply by having well documented conventions.

Example: Naive implementation of state sync is to walk the state trie from one side to the other, fetching all of the data via this simple iterative approach. We can alter this approach by using the block hash of every Nth

block to determine the path in the state trie where clients would start traversing the trie. This produces emergent behavior that all clients which are currently syncing the state will converge on requesting the same data which better utilizes the caches of the nodes serving the data as well as allowing partially synced nodes to more reliably be able to serve other partially synced nodes.

This approach should be able to be replicated across other node behaviors and the simple process of documenting “best practices” should go a long way.

Another option would be ensuring that requests are easy to cache which lets nodes which have recently routed a request to cache the result and return it on a subsequent request for the same or similar data.

Likely problems and issues that need to be thought about.

Header Chain Availability

Ideally, every participant of this network would store the full header chain, however, the storage requirements may be too steep for the types of nodes we expect to participate in this network.

Thus, we may need a mechanism to efficiently prove that a given header is part of the canonical chain, assuming that all nodes track some number of recent headers.

It may be ok

to require all participants to sync the entirety of the header chain and do some sort of data processing on it, but allow them to then discard the data. Can we do something fancy here?

TODO: Look into Eth2 research on how they do this for their light protocol. Something like a set of nested merkle tries which allow for a tuneable lightweight mechanism, though it appears it would require nodes to either trust someone to provide this data for them upon joining the network, or to fully process the canonical header chain to produce these merkle tries since they are not part of the core protocol.

Eclipse Attacks

A simple attack vector is to create multiple nodes which “eclipse” a portion of the network, giving the attacker control over the data for that portion of the network. They could then refuse to serve requests for that data.

DOS Attacks

We may need some mechanism to place soft limits or throttling on leeching peers. The ideal state of the network is to have a very large number of participants which are all both regularly requesting and serving data. We however should not rely on this naturally occuring since there is a natural tragedy of the commons problem that arrises from any node behavior which requests significantly more data than it serves.

Lost Data

It is more likely that this network could fully lose a piece of data. In this case the network needs a mechanism to heal itself. Intuition suggest a single benevolent full node could monitor the protocol for missing data and then provide that data.