Summary

tldr; use Kademlia, with keys restricted to on-chain commitments and values restricted to a maximum size, to store things like app UIs.

Motivation

Many ethereum use cases require more data than fits on chain. Currently, most use centralized hosting for that kind of data.

This is limiting for many use cases and sacrifices censorship resistance. Also means that app developers have to pay for hosting / API access, a cost that scales with growth.

Here’s a sketch of an idea to allow an app or browser to efficiently serve data peer-to-peer, as long as the data is reasonable size and committed to on-chain. This lets us avoid some shortcomings of past attempts at p2p data distribution.

Example use cases

For example, the source code (and even the ABI) for smart contracts is data stored offchain. Chain stores compiled bytecode only. Currently, if you want the source or ABI you go to Etherscan.

Say you’re making a wallet, and you want to show users what their transaction does before they click “Confirm”. Current wallets are bad at this; Metamask shows just the contract address and the raw data bytes. If the ABI was available, a wallet could show you the name of the contract and the function you’re calling.

Another use case: contract UIs like app.uniswap.org. This proposal would let dapp developers commit to a UI (say, HTML+JS bundle) on chain, giving uncensorable interfaces and a globally transparent app publishing mechanism.

Proposal

Kademlia is a DHT protocol. It provides an elegant solution to data availability, where every node stores data that is “close to it” in an ID space.

This approach sits between two extremes:

- on chain, every node stores all the data. this means that either adding data is very expensive, or running a node is very expensive, or both. see The Limits to Blockchain Scalability

(sharding will mitigate this somewhat, but storing data on-chain will remain expensive.)

- in kademlia/DHT, every node stores up to a fixed amount of data: say, the k keys closest to them in ID space. a large network (n nodes) provides efficient key-value lookup to O(n) total keys.

- in swarming protocols such as Bittorrent and IPFS, every node stores only the data on an opt-in basis. by default they serve only what a user has chosen to download. this leads to frequent data unavailability, the “0 seeds” problem.

DHTs have two big limitations that prevent them from being used directly to store large data. First is spam. Plain kademlia is susceptible to spamming and sybil attacks. We mitigate this by having each node enforce a maximum size and verify that data they’re storing matches an on-chain commitment, making spam expensive.

A second reason DHTs traditionally store small values is to limit bandwidth consumption for nodes storing popular keys.

We can mitigate this by using a swarming protocol (say, IPFS) to store values, while maintaining the rule that nodes store data for keys close to them.

So if you’re creating an installable dapp / wallet / browser, you can put a category of on-chain attested data you care about into a DHT + swarm.

Concrete example

You’re building a web3 browser. Browser bundles an ipfs node, using the standard ipfs DHT.

Browser stores app UIs (HTML+JS) that are committed on chain using ENS records (= IPFS link in a txt record). Specifically, the browser automatically seeds up to 1000 ipfs objects closest to the node in ID space

(using your ipfs node id) and with a size under 2MB = maximum 2GB disk consumption.

As long as the browser has a reasonable number of active users, this should provide robust data availability for objects meeting those requirements.

–

A developer publishing a dapp has an easy process with no hosting costs/limitations. They publish a smart contract on chain; compile the UI, seed to IPFS; and finally publish the IPFS link to their ENS domain name. Shortly after, a number of browser nodes see the link on-chain and automatically download and seed it. The original publisher can go offline now–their app is published on chain and available p2p to all other browser users.

Speed

Swarming protocols (eg Bittorrent, IPFS) have high latencies in part because of the need to find new peers for every object you want to download. To find /ipfs/xyz, or a bittorrent magnet://xyz, you follow roughly these steps:

- Look up xyz in a DHT. This requires a few hops.
- The value in the DHT is a list of peers seeding the file.
- Connect to those

peers. Ask them for parts of the file and for additional peers.

Here we shorten it to:

- Look up xyz in an DHT. This requires a few hops.
- As you get close to xyz in the DHT, the DHT nodes are also seeders

. You ask them for parts of the file and for peers.

We could shorten further by having each node download+seed some number of frequently requested files. So for popular files the browser would find seeds + download without even having to do a full DHT lookup.

This improves app latency / page load time.