Distributed Data Types --- Implementations and Their Scaling Walls

Scope

The research collection covers distributed data management foundations: consistency models, replication protocols, partitioning strategies, CRDTs, storage engines. This collection examines specific data structures and protocols that were designed for distributed use --- what they store, how they replicate, what their designers expected, and what happens in practice at scale.

Each document traces a concrete implementation: the data structure, its properties, its scaling characteristics, and the point at which those characteristics become problems. The focus is on generic algorithmic properties --- the structural reasons systems hit walls, not the specific parameter choices that could have been tuned differently.

Related Documents

This collection extends and complements several existing research documents:

- CRDTs and Merkle DAGs --- Section 9 covers Radicle's use of git as a CRDT foundation; Section 8 covers Merkle-CRDTs
- Blob and Large Object Storage --- Section 3 covers content-addressed storage fundamentals, CID structure, and chunking
- Distributed Chat Systems --- Section 2 covers Matrix's event DAG and federation protocol in depth
- Consistency and Ordering --- Section 7 covers Matrix's event graph ordering model

Documents

Document	Subject	Core Question
01 --- Git as Distributed Primitive	Content-addressed object store with mutable refs	What happens when you build collaboration systems on an append-only DAG designed for source code?
02 --- Federation and Feeds	SSB's append-only feeds, Matrix's event DAG, ActivityPub's inbox model	How do feed-based and event-based replication protocols behave as networks grow?
03 --- Content-Addressed Networks	IPFS CIDs, Kademlia DHT, Bitswap block exchange	What is the cost of decoupling content identity from content location?
04 --- Ethereum State Trie	The authenticated state trie	What happens when an ever-growing authenticated data structure has no mechanism for state removal?

Connecting Thread

Every system here chose a data structure that provides strong guarantees --- integrity, deduplication, fork detection, authenticated state. In each case, those guarantees come with structural costs that compound as the system scales. The interesting question is not whether the designers made mistakes, but which costs are inherent to the data structure and which are artifacts of a particular implementation.

Git as Distributed Primitive

Git's core is a content-addressed object store: an append-only, immutable collection of objects identified by the hash of their contents. On top of this sits a thin layer of mutable pointers (refs) that give human-readable names to specific commits. This combination --- immutable data with mutable pointers --- has made git the foundation for systems well beyond source control.

---

1. The Object Model

Git stores four object types: blobs, trees, commits, and tags. Every object is stored with a header of the form `<type> <size>\0<content>`, and the object's identity is the SHA-1 hash of this entire byte sequence (header included). The hash is the object's address in the store. Two objects with the same content produce the same hash and are stored once.

Blobs hold file contents --- raw bytes with no filename, no path, no metadata. A 10 KB source file and an identical 10 KB copy in a different directory share one blob object.

Trees map names to objects. Each entry in a tree contains a file mode (permissions), a filename, and the hash of the object it points to --- either a blob (for files) or another tree (for subdirectories). Trees are themselves content-addressed: two directories with identical contents produce identical tree hashes regardless of where they appear in the repository.

Commits tie everything together. A commit object contains: the hash of a tree (the complete snapshot of the working directory at that point), zero or more parent commit hashes (zero for the initial commit, one for ordinary commits, two or more for merges), author and committer identity with timestamps, and a message. The parent references form a directed acyclic graph --- the commit DAG.

Tags are annotated markers pointing to any object (usually a commit), containing the tagger's identity, a message, and optionally a GPG signature.

The distribution property follows directly from content addressing. Any two repositories that contain an object with the same hash contain the exact same bytes. When repositories synchronize, they exchange object hashes, identify what's missing, and transfer only the missing objects. Deduplication is free and requires no coordination --- the hash is the proof of identity.

Git is transitioning from SHA-1 (160-bit, with a practical collision demonstrated in 2017 at 2^63 operations) to SHA-256, introduced experimentally in Git 2.29 (2020). The transition is backward-compatible for blobs (which don't reference other objects) but changes commit and tree hashes (which embed object references).

---

2. Pack Files and Delta Compression

The content-addressed model stores every version of every file as a complete blob. A 50 KB source file modified 200 times produces 200 blobs totaling 10 MB, even if each edit changed a few lines. This is the "loose object" representation --- simple, correct, and wasteful.

Pack files solve this. Git groups objects and computes binary deltas between similar ones, storing the result as a single compressed file. The packing heuristic sorts objects by type (different types never delta against each other), then by filename, then by size. Objects adjacent in this sorted order become delta candidates --- the intuition is that files with the same name across commits are likely to be similar.

The delta format is not part of the logical data model. A pack file contains the same objects as the loose representation; the deltas are a physical optimization. Any object can be reconstructed by applying its delta chain back to a base object that is stored in full.

Delta chains can be deep --- deltas of deltas. Git limits chain depth (default: 50 levels, maximum: 4095) and search window (default: 10 candidate objects, aggressive mode: 250). Deeper chains produce smaller

pack files but increase the cost of reconstructing any single object, since reconstruction requires applying the full chain. The tradeoff is storage size versus random access latency.

Pack compression uses zlib deflate, with the compression level configurable via `pack.compression` (default: zlib's own default). The combination of delta compression and zlib typically achieves substantial reduction --- a repository with many similar text files may see 10:1 or better compression ratios in pack files versus loose objects.

One consequence: repacking is an offline optimization that can be expensive. `git gc` and `git repack` rebuild pack files, re-sorting and re-computing deltas. For large repositories, this can take minutes to hours and consume significant memory (configurable via `pack.windowMemory`, default limit: 8 GB, multiplied by thread count).

---

3. Refs: Mutability on Top of Immutability

The object store is append-only and content-addressed. Nothing in it is mutable --- you cannot change an object without changing its hash, which makes it a different object. But repositories need mutable branch names, so git adds refs: named pointers stored outside the content-addressed store.

A ref is a file containing a single hash. `refs/heads/main` might contain `a1b2c3d...`, meaning "main currently points to commit a1b2c3d." Updating a branch means overwriting this file with a new hash. This is the one mutable operation in git, and the source of most coordination problems in multi-writer scenarios.

Ref updates are not content-addressed, not deduplicated, and not self-verifying. Two users pushing different commits to the same branch create a conflict that git resolves with a simple rule: the push succeeds only if the update is a fast-forward (the new commit is a descendant of the current one). Non-fast-forward updates require force-pushing, which discards history.

The reflog records local ref mutation history --- every time a ref changed, what it pointed to before and after. Reflogs are local only (not replicated), expire after 90 days by default (30 days for unreachable

entries), and are deleted by garbage collection. They are a safety net, not a distributed protocol.

The traditional ref storage (one file per ref in `.git/refs/`) scales poorly. Repositories with thousands of refs (monorepos, CI bots creating per-build branches) hit filesystem overhead on every operation. The packed-refs format consolidates refs into a single file, but updating any ref requires rewriting the entire file. The reftable format (upstreamed in Git 2.45.0 via GitLab) uses a binary format with prefix compression and binary search --- 51% faster lookups (5.4 usec vs 11.2 usec in benchmarks) and incremental updates without full rewrites.

---

4. Systems Built on Git

Several systems have extended git's data model beyond source code, using the content-addressed store as infrastructure for distributed collaboration.

Radicle

Radicle is peer-to-peer code collaboration built directly on git's object store. Each peer has a cryptographic identity (Ed25519 key pair) used as a namespace within the repository. Git's native namespace feature isolates each peer's refs: one physical repository holds all forks, with each peer's contributions under their namespace. Objects shared across forks are deduplicated automatically by the content-addressed store.

Replication uses a gossip protocol with three message types: node announcements (identity and address), inventory announcements (which repositories a node hosts), and reference announcements (updates to specific refs). Metadata propagates via gossip; actual git objects transfer over the git protocol. Discovery relies on bootstrap nodes (e.g., `iris.radicle.xyz`) --- new peers must connect to a known node to find others.

Radicle extends git with Collaborative Objects (COBs): issues, patches, and code review stored as CRDT-based data structures within the git repository itself. Each COB is a DAG of commits under

`refs/cobs/` (e.g., `refs/cobs/xyz.radicle.issue/<id>`), disjoint from code branches. Modifications from different peers merge automatically via deterministic replay.

What works: offline-first operation, verified history, deduplication across forks, no central authority for code hosting. What breaks: discovery depends on bootstrap nodes (no global directory), connecting to a peer requires knowing their Node ID in advance (Noise XK handshake), and the network remains small (~2,000 repositories, ~200 nodes online weekly as of late 2024).

Fossil

Fossil is an integrated SCM created by the SQLite project, storing everything in a single SQLite database file. Beyond version control, the repository contains tickets, wiki pages, forum posts, technotes, and a built-in web UI. A checkout is a single file; deployment is a single 6-8 MB binary.

The integration provides cross-referencing that git's ecosystem achieves only through external tooling: commit messages can link to tickets, wiki pages reference commits, forum posts cite specific changes. Fossil preserves branch names permanently in the database (git loses branch metadata after merge) and can efficiently query forward history via SQL (finding all descendants of a commit, which requires expensive DAG traversal in git).

What breaks: the ecosystem. Fossil uses SHA3-256 (incompatible with git), has no CI/CD integration, no equivalent to GitHub's pull request workflow, and limited third-party tooling. Fossil is designed for small teams --- SQLite itself is ~0.12 MLOC maintained by a handful of developers. The tradeoffs that make it excellent for that use case (single-file deployment, integrated everything, minimal operational complexity) make it impractical for projects that depend on the git ecosystem.

git-bug

git-bug stores distributed bug tracking data as git objects under `refs/bugs/<id>`. Each bug is a chain of operation objects (create, comment, label, assign) using Lamport timestamps for deterministic ordering. The CRDT-based model means bugs replicate with the repository --- `git push` and `git pull` synchronize bug state alongside code.

The operation model avoids merge conflicts by construction: concurrent modifications from different remotes are automatically merged via timestamp ordering. There is no manual conflict resolution for bug state.

What breaks: computing the current state of a bug requires replaying all operations from the beginning. Garbage collection can delete bug objects if they become unreachable from any ref. And the UI is separate from git's --- standard git tooling (GitHub, GitLab, IDE integrations) does not display git-bug data.

---

5. Hard Limits

Repository size. Git clones the full history by default. The Linux kernel repository is ~4.6 GB with 1.1M+ commits; cloning it takes approximately an hour. Shallow clones (`--depth N`) reduce transfer size but break `git bisect` and operations that depend on full history. Partial clones (`--filter=blob:none`) reduce the Linux kernel clone from 4.6 GB to 1.4 GB and from 20 minutes to 4.5 minutes, but lazily fetch blobs on demand, adding latency to operations that access file contents.

Ref advertisement. On `git fetch` , the server sends the names and hashes of all refs. For repositories with thousands of refs (monorepos, CI bots), this is O(refs) bandwidth and CPU on every fetch, before any objects transfer. The reftable format helps server-side, but the wire protocol still transmits the full ref list. Microsoft's Windows repository (3.5 million files) required a custom Git implementation (GVFS/VFS for Git) to make operations practical.

Large binary files. Git's delta compression works by finding byte-level similarities between objects sorted by filename and size. Binary files (images, compiled assets, datasets) rarely share byte sequences across versions. Git LFS works around this by replacing binary content with small pointer files in the repository; the actual blobs are stored on a separate HTTP server. The pointer files are content-addressed (part of the Merkle tree); the actual blobs are not. This breaks git's self-contained integrity model --- a clone without LFS access produces a repository full of pointer files instead of content. Every client must install the LFS extension; misconfiguration produces silent data loss.

Merge semantics. Git's merge machinery (the ORT algorithm, successor to "recursive") performs three-way merge on line-oriented text. It has no awareness of code structure --- functions, classes, or syntactic boundaries. A benchmark of merge scenarios found 48% false conflicts (independent changes touching the same line). Extending git to non-text data (CRDT state, database records, structured configuration) requires custom merge drivers, which most tooling ignores. Binary files cannot be automatically merged at all; git takes the entire file from one side.

These limits share a pattern: they are consequences of design decisions that work well for git's original use case (moderate-sized source code repositories with text files) and degrade as the use case expands. The content-addressed model, the full-history clone, the line-based merge --- each is a strength in context and a wall outside it.

Federation and Feeds

Three protocols represent distinct approaches to distributing social data: SSB's signed append-only feed, Matrix's event DAG, and ActivityPub's inbox/outbox model. Each chooses a different data structure for replication, and each hits different scaling walls.

---

1. SSB: The Append-Only Feed

Secure Scuttlebutt represents each identity as a single append-only log --- a hash-linked chain of signed JSON messages. Every message contains: the author's public key, a sequence number (monotonically increasing), the hash of the previous message, a timestamp, and the content payload. The previous-message hash creates a Merkle chain: verifying message N requires message N-1's hash, which requires N-2's hash, back to the genesis message.

Messages are capped at 16,384 bytes (16 KB). The average message in practice is ~660 bytes. Content payloads are limited to 8 KB of text. Blobs (images, files) are stored separately with a default replication limit of 5 MB per blob.

This structure provides strong guarantees. A feed cannot be forked without detection (two messages with the same sequence number but different hashes are proof of misbehavior). Messages cannot be reordered or removed without breaking the hash chain. Any peer holding a prefix of the feed can verify its integrity without trusting the author's current server --- there is no server.

Replication: Epidemic Broadcast Trees

Replication uses Epidemic Broadcast Trees (EBT), loosely based on the Plumtree protocol. Peers maintain a vector clock of `(feed_id -> latest_sequence_number)` for every feed they track. On

connection, peers exchange vector clocks and send only the messages the other peer is missing.

The algorithm operates in two modes. In active mode, peers broadcast full messages eagerly. In lazy mode, peers exchange only vector clock entries, deferring message transfer until explicitly requested. The negotiation determines who sends what to whom: if peer A has feed F at sequence 50 and peer B has feed F at sequence 42, A sends messages 43-50 to B.

The bandwidth complexity is O(messages + updated_feeds) --- proportional to the number of messages actually transferred plus the overhead of vector clock exchange. This is efficient when peers share most of the same feeds. The problem surfaces when they don't: each vector clock element is ~60 bytes, and a peer following thousands of feeds sends the full vector clock on every connection. Early SSB implementations saw peers exchanging over half a megabyte of vector clock data per connection, even when no messages needed to be sent.

In a 10,000-peer network with each peer making one random connection per interval, a message reaches all peers within 8 rounds of gossip. Complete dissemination requires 5-10 connections per peer. But the cost of broadcasting scales with connection count: in a 1,000-peer network, K=2 connections produces 2.98x message overhead (2,981 messages to deliver one), K=5 produces 8.87x overhead, and K=10 produces 18.5x overhead.

Scaling Walls

Feed size. A feed is append-only and immutable by design. A user who posts 10 messages per day accumulates ~3,650 messages per year. With average message sizes of 660 bytes, a single feed reaches ~2.4 MB per year. With larger messages (1-4 KB), this grows to 3-15 MB per year. This is manageable per-feed, but a peer following 500 feeds replicates 1.2-7.5 GB per year, and must process the full history of any new feed they subscribe to. There is no "shallow follow." The default replication strategy is all-or-nothing: you get the complete feed or nothing.

Social graph density. By default, peers replicate feeds from friends and friends-of-friends (two hops on the social graph). With 5 random peers replicating two hops out, approximately 75% of friends-of-friends are covered by at least one peer. But sparse overlap between peers wastes bandwidth: two peers with few shared feeds still pay the full vector clock exchange cost.

The onboarding problem. SSB has no servers, so new users need a "pub" --- an always-online peer that follows the new user and serves as a relay for their messages. Pubs became de facto infrastructure: users generate invite codes that command pubs to follow their friends. The result is the centralization SSB was designed to avoid. Anyone can run a pub, and the protocol works without them given full IPv6 connectivity, but in practice the network depends on a small number of well-known pubs. The ecosystem has been moving toward "room servers" --- connection relays that facilitate peer introductions without hosting feed data --- but the pattern persists.

Partial replication. The ssb-subset-replication specification attempts to solve the full-feed problem by allowing peers to request subsets of messages (filtered by author, type, or other criteria). A query language (ssb-ql-0) supports basic filters: author feed ID, message type, and a privacy flag. Pagination parameters allow incremental sync. The tradeoff: partial replication cannot validate complete hash chains. Receiving a subset of a feed means trusting that the sender has not omitted messages, which sacrifices the integrity guarantee that makes SSB's feed structure valuable. The specification remains in design phase.

---

2. Building on Matrix's Event DAG

The research collection covers Matrix's federation protocol, event DAG structure, and state resolution algorithm in depth (see Consistency and Ordering Section 7 and Distributed Chat Systems Section 2). This section focuses on what an engineer encounters when building on top of Matrix --- using it as infrastructure rather than as a chat system.

Custom State Events

Matrix allows arbitrary state event types. A game application can define `com.example.game.score`; a collaborative editor can define `com.example.doc.cursor`. These custom events participate in the same state resolution as built-in events (membership, power levels, room configuration). State resolution v2 applies to all state events: conflicts are resolved by considering power levels, event ordering, and the auth chain. A custom state event must survive the same resolution logic as a membership event --- there is no way to opt out or define custom conflict resolution.

This is both powerful and constraining. Custom state travels with the room, replicates across federated servers, and benefits from Matrix's existing infrastructure. But the state resolution algorithm was designed for access control semantics (higher power levels take precedence, earlier events are preferred), which may not match the semantics of an application's custom data.

Computational Cost of State Resolution

State resolution v2 requires multiple passes over the event graph: separating conflicted from unconflicted state, processing control events via reverse topological sort, ordering normal events via mainline following, and reapplying events with auth checks. Room complexity scales with the number of state events and the depth of the auth chain.

State resolution v2.1 (MSC4297, deployed in Room Version 12 via Matrix Spec v1.16) addressed state reset attacks where a malicious homeserver could corrupt room state by sending crafted event sequences. The fix replays all events in conflicted state subgraphs (strongly connected components), adding computational cost for correctness.

For large federated rooms, the practical impact is visible: initial room joins can take 10-60 minutes as the joining server fetches events, validates auth chains, and computes state resolution across potentially thousands of events. Analysis by researchers at Karlsruhe Institute of Technology (ACM SACMAT 2020) examined Matrix's access control model and identified implementation issues in state resolution that were subsequently fixed in Synapse and Room Version 6.

The Linearization Problem

Matrix's event DAG provides partial order: concurrent events from different servers are not ordered relative to each other. Applications that need total order --- collaborative text editing, turn-based games, sequential transaction processing --- must linearize the DAG. Topological sorting produces a valid linearization, but multiple valid linearizations exist for the same DAG, and different servers may choose different ones.

This reintroduces the coordination problem that the DAG was designed to avoid. A collaborative editor built on Matrix must either accept that users on different servers may temporarily see different event

orderings, or implement a consensus layer on top of Matrix to agree on a single linearization --- at which point the DAG's tolerance for concurrent events provides no benefit.

Bridge Impedance Mismatches

Matrix bridges translate between Matrix's DAG-based event model and other systems' ordering models. The Matrix-IRC bridge logs into IRC as real clients and relays messages bidirectionally. Messages sent between IRC and Matrix may arrive out of order, and the order may vary by homeserver --- a direct consequence of Matrix's eventual consistency model interacting with IRC's total-order-per-channel model.

The mismatch goes beyond ordering. Access control synchronization between IRC channel modes (+i, +k) and Matrix room state requires careful mapping. Features that exist in one system but not the other (IRC op-only messages, Matrix reactions, Matrix threads) create semantic gaps that the bridge must either approximate or drop. Federation network issues cause delays, message repetition, and intermittent failures that surface as user-visible glitches in the bridged channel.

---

3. ActivityPub: Inbox Delivery at Scale

ActivityPub defines a simple model: actors have an inbox (receive) and an outbox (publish). When a user creates a post, their server delivers it by sending an HTTP POST with a JSON-LD Activity to every recipient's inbox. The `to`, `cc`, and `bcc` fields on the activity determine the recipients.

Fan-Out Arithmetic

Delivery is per-server, not per-follower. An account with 50,000 followers across 8,000 servers requires 8,000 HTTP POST requests, not 50,000 --- the receiving server routes to individual inboxes internally. Servers may advertise a `sharedInbox` endpoint to batch deliveries: 100 followers on the same server can be reached with a single POST to the shared inbox.

This is still significant. Each POST requires a connection, authentication (HTTP Signatures), JSON-LD parsing on the receiving end, and retry logic for failures (the spec says delivery SHOULD be asynchronous and SHOULD retry on network errors). Mastodon implements delivery via Sidekiq job queues with separate queues for incoming federation (pull), outgoing delivery (push), and internal processing (default). High-traffic instances report 100% CPU load in Redis during traffic spikes, and queue optimization requires splitting queues across separate processes and CPU cores.

Fan-out also affects content discovery. When a post goes viral, every instance that wants to display it must independently fetch OpenGraph metadata, user profiles, and media. Poorly optimized implementations create unintentional DDoS patterns as thousands of servers simultaneously request the same resources.

Single Server of Record

An actor's canonical state lives on their home server. The home server holds the actor's identity, follower list, post history, and cryptographic keys. If that server goes down, the actor becomes unreachable. There is no replication of actor state across the federation. Posts that were already delivered to other servers remain on those servers, but the original actor's profile, follower relationships, and future posts are unavailable until the home server returns.

ActivityPub does not natively support account migration. Implementations have built their own solutions (Mastodon's account migration redirects followers), but thread continuity breaks: replies reference the original post's URI on the home server, and if that server is offline, reply threads show gaps and outdated statistics.

The architecture assumes stable, long-lived servers --- the same assumption that email makes. The consequences are similar: a small number of well-resourced servers become the default choice for most users.

Relay Infrastructure

To reduce fan-out costs, relays aggregate and redistribute activities. `relay.fedi.buzz` connects to approximately 330 Mastodon instances' streaming APIs, serving ~3,500 unique relay followers. It holds

posts in memory (not on disk), checks whether content matches subscribers' interests (by tag, instance, or language), and queues matching posts for delivery. Other relays (e.g., `relay.mastodon.host` with 288 subscribed instances) serve similar aggregation functions.

Relays are centralization points. A relay failure affects content visibility across all its subscribers. A relay operator can filter, delay, or inspect traffic passing through their infrastructure. The relay model reproduces the architecture of content delivery networks --- a layer of intermediaries between publishers and consumers --- within a protocol designed to eliminate intermediaries.

The Centralization Pattern

Mastodon.social is by far the largest Mastodon server. Across the broader Fediverse (approximately 9 million Mastodon accounts, ~1.5 million active as of late 2024), the server size distribution is heavy-tailed: research has found that 96% of users join the largest 25% of instances. The distribution of connections (in-degree and out-degree) follows a lognormal pattern with a secondary power-law fit --- a small number of large servers dominate connectivity.

This is not a bug in ActivityPub; it is a structural consequence of the protocol's design. Federation requires running a server. Running a server requires technical skill, ongoing maintenance, and moderation labor. Users prefer servers that are stable, well-moderated, and socially connected. These preferences concentrate users on a small number of well-resourced servers, reproducing the centralization that federation was intended to prevent. The same pattern appears in email (Gmail), XMPP (jabber.org historically), and IRC (the largest networks).

Content-Addressed Networks

Content addressing separates what data is from where data is. A file's identity is the hash of its contents; anyone who has the file can verify it, and anyone who wants the file can request it by hash from any source. IPFS is the most prominent system built on this idea. Its architecture combines content-addressed storage (CIDs and Merkle DAGs), a distributed hash table for discovery (Kademlia), and a block exchange protocol (Bitswap). Each layer works well in isolation. The problems emerge at the seams.

---

1. The Content-Addressed Model

CIDs

A Content Identifier (CID) is a self-describing hash: it encodes the hash function used, the codec for interpreting the data, and the hash digest itself. CIDv1 uses a multicodec prefix to indicate how the data should be parsed (raw bytes, dag-pb, dag-cbor, etc.) and a multihash to specify the hash algorithm and digest length. This makes CIDs future-proof --- a CID created with SHA-256 and a CID created with BLAKE3 are both valid and distinguishable.

For an extended discussion of CID structure and content-addressed storage fundamentals, see Blob and Large Object Storage Section 3 (Content-Addressed Storage).

Merkle DAGs

Files larger than a single block (default chunk size: 256 KB) are split into chunks, each chunk is hashed to produce a CID, and the chunks are linked into a Merkle DAG. Directory structures are DAGs of DAGs: a directory node contains CIDs pointing to file nodes, which contain CIDs pointing to chunk

nodes. The root CID of a directory commits to the entire directory tree --- changing any byte in any file changes the root CID.

The promise: request any content by its CID, receive it from any peer that has it, verify on receipt that the content matches the CID. No trust in the source is required. The verification is the same operation as identifying the content.

---

2. Kademlia DHT: Content Discovery

Content addressing solves integrity but not discovery. Knowing a CID tells you what you want, not who has it. IPFS uses a Kademlia-based distributed hash table (DHT) to map CIDs to providers.

How It Works

Nodes in the DHT have IDs in the same keyspace as content (256-bit). The "distance" between two IDs is their XOR --- a metric that satisfies the triangle inequality and distributes evenly. Each node maintains a routing table of other nodes, organized into "k-buckets": for each bit position, a bucket holds up to K nodes at that XOR distance (K=20 in IPFS). The result is that each node knows many nearby nodes and progressively fewer distant ones.

Looking up a CID means iteratively querying nodes whose IDs are closer to the target. Each hop narrows the distance. With N nodes in the network, a lookup completes in O(log N) hops --- approximately 15-16 hops for a 40,000-node network, approximately 20 hops for 10 million nodes. Each hop is a full network round trip.

Provider Records

A node that has content publishes a "provider record" --- an announcement saying "I have CID X" --- to the K=20 nodes closest to X in the keyspace. These records have a TTL. The current defaults (revised from earlier 12h/24h values) are a 22-hour republish interval and a 48-hour expiration. If a provider stops

republishing, its records disappear and the content becomes undiscoverable through the DHT, even though the data still exists on the provider's disk.

The asymmetry: publishing a provider record is a single operation. Keeping content discoverable requires republishing every 22 hours, for every CID the node hosts. A node hosting 10,000 CIDs must perform 10,000 DHT publish operations per day. For nodes hosting large collections, the republishing overhead becomes the dominant network cost --- the revised intervals reduced this overhead by approximately 50% compared to the original 12-hour schedule, but the O(CIDs) scaling remains.

Measured Performance

Trautwein et al. (ACM SIGCOMM 2022) conducted the most comprehensive measurement study of the IPFS public network. Key findings:

- **DHT lookup latency**: The "cost of decentralization" --- comparing IPFS retrieval time against estimated HTTPS retrieval time --- shows a stretch factor below 2 for 80% of content retrievals from central Europe. In absolute terms, DHT lookups on the public network take seconds, not the sub-100ms latencies of DNS or CDN routing. Follow-up work (INFOCOM 2024) demonstrated that "optimistic provide" --- publishing records before the provide operation completes --- achieves sub-second PUT latencies in over 90% of operations from North America and central Europe.
- **Provider record reliability**: Over 70% of provider records in the DHT point to peers that are not reachable. The remaining peers are often reachable only via relay nodes. This is a direct consequence of churn.
- **Network churn**: A single IPFS client observed connections to 40,000-65,000 different peer IDs while maintaining only ~16,000 simultaneous connections, indicating high node turnover. Nodes go offline constantly. The DHT must continuously repair routing tables, and lookups frequently hit stale entries. Peers that exhibit daily periodic uptime (booted for a few hours, then shut down) can appear as "stable" peers because the provider record TTL exceeds their daily session length.

---

3. Bitswap: Block Exchange

Once a provider is found via the DHT, the actual data transfer uses Bitswap --- a block-level exchange protocol where peers negotiate which blocks to send using want-lists and have-lists.

The Protocol

When a node wants a block, it adds the CID to its want-list and broadcasts it to connected peers. Peers with the block add it to their have-list and send it. Bitswap was designed with tit-for-tat incentives inspired by BitTorrent: peers that send blocks to you get priority when they request blocks from you. A ledger tracks the debt ratio between peer pairs.

Why the Incentives Don't Work

BitTorrent's tit-for-tat works because both sides have something the other wants --- they are downloading the same file concurrently and can exchange complementary pieces. IPFS's usage pattern is different: most nodes are consumers requesting content from a small number of providers. The typical interaction is asymmetric --- one side has content, the other wants it, and there is nothing to trade. The debt ratio mechanism has little practical effect. Implementation issues compounded this: the `MessageSent` function that was supposed to update the ledger was not being called in the go-bitswap implementation, and the ledger was described by developers as trying "too hard to be fair" in a way that creates a zero-sum game from a throughput standpoint.

Overhead

Bitswap operates at the block level. Each 256 KB chunk requires want-list/have-list negotiation, CID comparison, and protocol framing. For a 100 MB file split into ~400 blocks, this means hundreds of round trips of protocol messages before the transfer completes.

Contrast with HTTP: a CDN serves a 100 MB file in a single TCP connection with range requests, amortizing connection setup over the entire transfer. IPFS first discovers providers via the DHT (seconds), then negotiates via Bitswap (multiple round trips per block), then transfers block-by-block.

For the common case of "fetch a known file from a known location," IPFS adds latency at every step relative to HTTP.

---

4. What a Working System Would Require

The Core Tension

Content addressing provides integrity and deduplication. These properties are valuable and well-proven --- git, Nix, Docker, and many other systems use content-addressed storage successfully. The difference is that those systems separate content addressing (for integrity) from discovery and distribution (handled by other mechanisms).

Git uses content addressing for its object store but relies on explicit remote configuration and ref advertisement for discovery. Nix uses content-addressed store paths for deduplication but distributes packages through centralized binary caches with HTTP. Docker registries use content-addressed layers but serve them through a registry API. In each case, the content-addressing layer handles verification, and a separate, typically centralized or mirror-based, layer handles "where do I get this?"

IPFS conflates content addressing with discovery and distribution, using a single mechanism (the DHT) for both "what is this content?" and "where is this content?" The DHT answers the second question poorly because it was designed for the first.

The Pinning Economy

Filecoin attempted to solve the persistence problem by incentivizing storage with cryptocurrency. Storage providers (miners) commit storage capacity and earn tokens for proving they hold data. The proof-of-storage mechanism requires "sealing" data into sectors --- a computationally expensive process (~1.5 hours per 32 GB sector) that makes the data difficult to access. Retrieving sealed data requires "unsealing" (~3 hours per 32 GB sector on minimum hardware).

The result: miners optimize for proof-of-storage (which earns tokens) rather than for serving content (which does not). Data is verifiably stored but not necessarily retrievable in any practical timeframe. The system provides archival guarantees, not retrieval performance --- cold storage, not a CDN.

Separating the Layers

A content-addressed network that worked for general content distribution would need to separate its layers explicitly:

- **Content addressing** (integrity and deduplication): CIDs, Merkle DAGs, content verification. This layer works well. It is battle-tested across many systems.
- **Discovery** (mapping CIDs to locations): The DHT approach adds seconds of latency per lookup and suffers from churn. Pre-computed provider indexes, HTTP-based discovery endpoints, or DNS-based routing would provide orders-of-magnitude better latency. This is the layer that benefits from centralization or at least from stable infrastructure.
- **Retrieval** (transferring data): HTTP with content verification (download the file, check the hash) achieves the integrity guarantees of content addressing with the performance of existing CDN infrastructure. Block-level negotiation adds overhead without corresponding benefit for the common case.
- **Persistence** (keeping data available): Requires economic incentives or organizational commitment. Pinning services, mirrors, and institutional hosting are honest about being centralized. The question is not whether persistence requires coordination, but whether the coordination is transparent.

---

5. Hard Limits

DHT lookup: Minimum O(log N) network round trips. On the public IPFS network, practical latency is seconds for discovery. Optimistic provide techniques have improved PUT operations to sub-second in

favorable network conditions, but GET operations still depend on provider record freshness and peer reachability.

Provider records: 48-hour expiry, O(CIDs) republish cost per node every 22 hours. A node hosting 10,000 CIDs performs 10,000 DHT publish operations daily. Over 70% of provider records point to unreachable peers.

Bandwidth: Block-level negotiation (want-list/have-list per 256 KB chunk) adds per-block protocol overhead. For bulk transfers, this is strictly worse than HTTP range requests or BitTorrent's piece-level negotiation with larger piece sizes.

Naming: IPNS (InterPlanetary Name System) resolves mutable names via the DHT, inheriting all its latency and reliability problems. An IPNS record maps a peer ID to a CID, allowing content updates without changing the name. Resolution requires a DHT lookup plus record verification, adding seconds of latency on top of the content retrieval itself. IPNS records have their own TTL and caching semantics, creating a second layer of staleness on top of provider record staleness.

Ethereum State Trie

Ethereum stores the entire state of every account and every contract in a single authenticated data structure: the Modified Merkle Patricia Trie. The block header contains the root hash of this trie, which commits to the complete state of the network at that block height. This design enables compact proofs and light client verification. It also means the state can only grow.

---

1. The Authenticated State Trie

What It Stores

Every Ethereum account is a four-element structure: nonce (transaction counter), balance (in Wei), storage root (the root hash of a separate trie containing contract storage), and code hash (the hash of EVM bytecode, empty for externally owned accounts). The world state trie maps addresses to account structures. The path through the trie is the Keccak-256 hash of the Ethereum address; the value is the RLP-encoded account.

Contract storage lives in per-account sub-tries. Each contract has its own storage trie mapping 256-bit keys to 256-bit values, with the root hash of this sub-trie stored in the account's `storageRoot` field. A contract with 10,000 storage slots has a sub-trie with 10,000 leaves.

The Authentication Mechanism

The trie is a commitment structure. Every internal node stores hashes of its children. The root hash of the trie, stored in the block header, is cryptographically dependent on every leaf. Changing any account's balance, any contract's storage slot, or any nonce changes the root hash. The block hash includes the state root, making each block a commitment to the entire Ethereum state.

This enables Merkle proofs: given the state root, anyone can construct a proof that a specific account has a specific state. A light client can verify a claimed account balance by checking a proof path from the leaf to the root, without downloading the full state trie. The proof consists of the sibling nodes along the path --- the minimum information needed to recompute the root hash.

The Design Goal

The trie was chosen for one property: any node can produce a compact, trustless proof of any piece of state. This is the foundation of light client verification and cross-chain communication. A full node makes a claim ("account X has balance Y at block Z"); a light client verifies the claim against the published state root without trusting the full node.

---

2. The Growth Problem

State grows monotonically. Every new account, every new contract deployment, every new storage slot written by a contract adds nodes to the trie. The protocol provides no mechanism for removing state that is no longer accessed.

Growth Trajectory

- **2015 (launch)**: Near-zero state. Genesis block contained pre-allocated accounts.
- **2021**: Head state approximately 130 GB. Geth chaindata directory: 528 GB total (267 GB in historical block data).
- **2024**: Reth node measurement (March, Paradigm analysis): 245.5 GiB state on disk. Breakdown: accounts 14.1%, contract bytecodes 4.3%, contract storage 81.7%. State growing at approximately 50% per year.
- **2025**: Full node with default pruning requires 3+ TB total storage. Archive nodes (all historical states) range from 2-3.5 TB (Erigon, optimized) to 18-20 TB (Geth).

The contract storage component dominates: over 80% of state. And approximately 80% of stored state has not been accessed for over a year, yet validators must maintain it.

SELFDESTRUCT Deprecation

`SELFDESTRUCT` was the only opcode that could remove state. EIP-6780, deployed in the Dencun upgrade (mainnet March 13, 2024), restricted it: contracts can now only self-destruct in the same transaction as their creation. Calling `SELFDESTRUCT` on an existing contract sends its funds but leaves code, storage, and nonce intact.

The removal was necessary because `SELFDESTRUCT` broke composability. A contract could cease to exist mid-transaction, invalidating assumptions made by other contracts that held references to it. Combining `CREATE2` (deterministic address deployment) with `SELFDESTRUCT` allowed deployed code to change --- deploy, destroy, redeploy different code at the same address --- which undermined the immutability guarantee that other contracts depended on.

With `SELFDESTRUCT` effectively gone, the trie only grows. Every account, once created, persists indefinitely.

Write Amplification

Updating a single leaf in the trie requires recomputing hashes for every node on the path from the leaf to the root. The trie's depth grows logarithmically with the number of leaves, but each write touches every node along that path. A single storage slot update can trigger 25-50 random database reads to traverse the trie, plus hash recomputations at each level. Disk I/O represents approximately 70% of block execution overhead.

As the trie grows, paths get longer and the working set gets larger. `SLOAD` (reading a storage slot) is among the most expensive EVM opcodes, not because of computation, but because it requires traversing the trie with random database lookups.

No Caching-Friendly Access Pattern

Trie keys are Keccak-256 hashes of Ethereum addresses. The hash function distributes addresses uniformly across the keyspace. Sequential account accesses in contract execution --- calling contract A which calls contract B which reads storage from contract C --- map to random positions in the trie. There is no locality to exploit.

CPU caches and OS page caches are effective when access patterns have temporal or spatial locality. The state trie has neither. A random caching strategy achieves a hit ratio of `cache_size / total_state_size`. With state exceeding 200 GB and typical node memory at 16-64 GB, most accesses are cache misses that hit disk. This is not a tuning problem; it is a structural property of using a hash function to distribute keys.

---

3. No Pruning, No Sharding

Pruning

The current trie is needed to validate new blocks. Historical tries are needed for archive queries (e.g., "what was account X's balance at block 10,000,000?"). Pruning historical state saves storage but breaks archive access.

The pruning challenge is structural: states across different blocks share almost all their data. The trie deduplicates identical subtrees --- if only one account changed between blocks N and N+1, the two state tries share every node except those on the modified path. When arbitrarily many blocks reference the same trie nodes, determining whether a node is safe to delete (unreferenced by any retained state) becomes expensive. The storage difference between a pruned node (~90 GB historically, now in the hundreds of GB) and an archive node (multi-TB) reflects this: pruning saves an order of magnitude, but the pruned size is still large and growing.

Historical states can be reconstructed by replaying transactions from genesis. This allows aggressive pruning in principle, but replaying the full transaction history takes days to weeks on current hardware.

State Expiry

State expiry proposals (e.g., EIP-7736) would evict accounts untouched for approximately one year. Expired state would not be deleted but moved to a separate, less accessible store. Resurrecting an expired account would require providing a witness proof --- cryptographic evidence of the account's last known state.

This solves the growth problem but changes Ethereum's contract model. Currently, deploying a contract is a one-time operation: the contract exists forever, its storage persists forever, and other contracts can depend on its existence. State expiry breaks "deploy and forget." Contract developers and users must account for state potentially becoming inactive. Resurrection requires witnesses, which must be obtained from archive services or proof providers --- adding a dependency on infrastructure that may not exist when needed.

The interaction with existing contracts is the hard part. Thousands of deployed contracts assume that storage slots, once written, persist indefinitely. Introducing state expiry requires either grandfathering existing contracts (limiting the benefit) or accepting that some deployed contracts will break.

Why Sharding Is Structurally Impossible

The state trie cannot be partitioned across nodes without breaking the root hash commitment. Computing the state root requires access to the complete trie. A transaction can read or write any account's storage (a contract can call any other contract), so a validator cannot know in advance which portion of state a transaction will access. Every validator needs access to the full trie to validate any block.

This is the fundamental barrier to state sharding in Ethereum. Research into approaches like Layered Merkle Patricia Tries has explored separating the trie structure from the commitment, but these require multiple root hashes --- incompatible with Ethereum's single-state-root model.

The contrast with conventional databases is instructive. A B-tree database can separate the authenticated commitment (e.g., a Merkle tree over key ranges, computed lazily) from the data layout (B-tree pages optimized for sequential access). Reads are fast because the data layout has locality. Proofs are fast because the commitment structure is computed over the data. Ethereum's trie fuses authentication and

data access into one structure: the path you traverse to read a value is the same path you hash to compute a proof. This makes proofs compact but makes reads random.

---

4. The Centralization Consequence

Hardware Escalation

Running a full node in 2015 required a laptop. By 2025, the minimum requirements are: 8 CPU cores (16 threads recommended), 16 GB RAM (64 GB recommended for stable operation), 4-8 TB NVMe SSD (HDDs are insufficient --- the random access pattern requires NVMe latency), and 300-500 Mbps bandwidth (1 Gbps recommended). This is workstation or server-class hardware.

The growth is continuous. Each year, the state grows, the history grows, and the hardware requirements increase. The set of people who can practically run a full node shrinks.

The Infrastructure Dependency

When running a full node is impractical for most users, they connect through RPC providers. Infura and Alchemy handle an estimated 90% of Ethereum RPC traffic (combined). The top 5 infrastructure providers captured 48% of 2024 blockchain infrastructure revenues. These are centralized companies providing API access to Ethereum state --- the protocol has no mechanism for decentralized RPC access.

The global Blockchain Nodes-as-a-Service market was approximately $135 million in 2024, projected to reach $318 million by 2032. This market exists because the state trie makes self-sovereign access to Ethereum state expensive.

Validator Centralization

Proof-of-stake validators must access state to validate blocks. The state size advantages well-resourced operators who can maintain high-performance infrastructure. Staking pool concentration as of 2025: Lido

holds 24-25% of all staked ETH (~8.7 million ETH), Coinbase holds 12-22% (~1.8 million ETH), Kraken holds ~16% of the centralized staking market (~1.3 million ETH), and Binance holds ~8.4%.

The top 3-4 entities control a large share of validation, and individual stakers increasingly delegate to pools rather than running their own infrastructure. The state trie's resource requirements are one factor (among several, including capital requirements and technical complexity) that pushes validation toward professionalized operators.

---

5. Proposed Fixes and Their Tradeoffs

Verkle Trees

Verkle trees replace the hash-based trie with a vector-commitment-based structure. Keys are 32-byte elements (31-byte "stem" + 1-byte "suffix"), and each node has 256 children, creating a flatter tree. Proofs use polynomial commitments instead of hash sibling paths.

The improvement: proof sizes drop from approximately 1 KB per account (Merkle) to approximately 150-200 bytes per account (Verkle). For full blocks accessing thousands of accounts, total witness size drops from ~18 MB to kilobytes. This is the enabling technology for stateless clients.

The tradeoff: Verkle trees require newer cryptographic assumptions (inner product arguments on Pedersen commitments, relying on the discrete logarithm assumption) compared to hash-based Merkle proofs. Deployment requires a hard fork and migration of the entire state trie. The Kaustinen testnet has been running since 2024, with multiple client implementations (Geth, Nethermind, Besu, and others) in progress. Mainnet deployment is projected for late 2025 or 2026, pending testnet stabilization. Verkle trees were not included in the Pectra upgrade (May 2025).

Stateless Clients

Instead of storing the full state, stateless clients receive state proofs (witnesses) with each block. The block proposer includes a proof for every state access the block's transactions perform. Validators verify

the proofs against the state root without maintaining local state.

The tradeoff: stateless validation shifts the state burden from validators to block proposers. Someone must still maintain the full state trie to generate witnesses. This creates a two-tier system: lightweight stateless validators (which can run on modest hardware) and full state maintainers (block builders, MEV searchers, RPC providers) who are necessarily well-resourced. Statelessness does not eliminate centralization; it changes which participants must be centralized.

State Expiry (EIP-4444, EIP-7736)

EIP-4444 (history expiry) is already implemented: execution clients stop serving historical block data older than one year, reducing disk requirements by 300-500 GB. Historical data remains accessible through archive services or peer-to-peer retrieval (Portal Network).

EIP-7736 (leaf-level state expiry in Verkle trees) proposes expiring state that hasn't been accessed within approximately one year. Expired accounts remain in the trie but are marked inactive; resurrection requires a witness proof. This breaks the assumption that deployed contracts persist forever without maintenance --- a fundamental change to Ethereum's contract model.

The Common Thread

Every proposed fix trades one form of complexity for another. Verkle trees trade cryptographic assumptions and a migration effort for smaller proofs. Stateless clients trade centralized proof generation for decentralized validation. State expiry trades the "deploy and forget" contract model for bounded state growth. The state trie's properties --- authenticated, ever-growing, random-access, globally committed --- are deeply embedded in how transactions interact with state. Changing the data structure means changing the execution model, and changing the execution model means contending with thousands of deployed contracts that depend on the current semantics.