hostnought: Community Data That Survives Its Infrastructure

A Serverless Application Substrate on Self-Repairing Browser Swarms

Jared Myron Arthur Leonard

Working Paper — implemented and deployed · August 2026

Abstract

Every web application rests on infrastructure somebody keeps paying for: an application server, a database, and an organization willing to renew both. Withdraw any of them and the data goes too. This happens constantly, and rarely for interesting reasons. Maintainers lose interest. Companies get acquired. Cards on file expire. An archive a community spent a decade building goes quiet for reasons that have nothing to do with whether anyone still wants it.

hostnought moves the data. The application ships as static files from an object store, and everything durable lives in the browsers of the people using it, cut into erasure-coded fragments and repaired as those browsers come and go. Underneath is a rendezvous-placed repair loop that runs without coordination: every peer works out where fragments ought to be, notices which are missing, and elects a repairer from gossiped state alone. Durability then costs a community roughly what recruiting moderators costs it, and that cost does not grow with the number of users.

Above the storage layer, state is a pure fold over a signed event log, so an application amounts to a set of schemas, a set of reducers, and optionally some WebAssembly. Applications are admitted at runtime by a founder-council ratification chain, which means shipping one changes nothing about the substrate.

The system runs. Storage holds 60% hourly churn for 48 simulated hours without losing a segment, connection count no longer tracks community size, and the compute layer caught every dishonest worker we planted in it. We report measured costs for the hot paths, describe the five places the system broke under load and what each fix bought, and set the work against twenty years of peer-to-peer systems that wanted much the same thing.

> **Status.** This is a living working paper: it tracks the deployed system and is updated in place as the system evolves. Corrections are made where they matter, marked as corrections, rather than hidden in a changelog. Everything quantitative is measured on the deployed implementation unless it is explicitly labelled analytical.

1 Introduction

A community's archive belongs to whoever pays the hosting bill. That is not a claim about law. It is a claim about location: the bytes sit on machines the community does not control, and when those machines stop, the archive stops with them.

The failure is mundane and relentless. A maintainer drifts away. A company is acquired and the product is sunset. A card expires while its owner is unreachable. A jurisdiction objects. The proximate cause varies; the outcome rarely does, and the people who built the thing usually find out afterwards.

hostnought inverts the arrangement. The application is a static bundle. The data lives in the browsers of people who care about it, sliced by erasure coding and repaired continuously as peers arrive and leave. There is nothing to seize, subpoena, bill, or switch off, because there is no server. The claim we defend is narrow: **community data can survive the loss of its infrastructure**, because the community's own devices carry it, verify it cryptographically, and can rebuild it from any surviving quorum.

1.1 Contributions

- **A serverless application substrate** (§3–5). The whole application is static files, and all state lives in an erasure-coded browser swarm over an append-only log of signed, content-addressed events.
- **The rendezvous-placed erasure repair loop** (§4). This is the piece we think is genuinely new. Each peer derives ideal fragment placement, detects deficits, and elects repairers from gossip, with no directory and no coordinator anywhere in the path. Earlier browser peer-to-peer systems replicated whole objects or leaned on designated pinning. Erasure coding's durability advantage over replication is long established [1], and OceanStore proposed continuously repaired erasure-coded archival storage across untrusted machines a quarter-century ago [2], but on provisioned infrastructure. None of the browser systems ran continuous erasure repair across peers this ephemeral.
- **A degree-capped neighbour overlay** (§5.3). It separates the roster of who exists from the link set of who is connected, which holds connection count flat as a community grows. Signalling payloads are encrypted under a room-derived key, so relays carry ciphertext rather than session descriptions.
- **A deterministic event-fold application model** (§6) that generalizes the substrate past any one application. Five demonstrate it: a forum, a distributed search index, private messaging, a wiki, and a guestbook written outside the codebase entirely.
- **Governed runtime application loading** (§6.5). Application code is admitted by a ratification chain mirroring the bootloader's own, so shipping an application means uploading a bundle and collecting a signature, and that application ends up governed exactly as tightly as the substrate.
- **An enforced validity contract for applications** (§6.6): fold purity and junk tolerance, checked by a validator the ratification tooling will not skip.
- **A verifiable compute layer** (§5.2) that runs deterministic WebAssembly across elected peers and accepts results by hash majority. Execution is bounded three ways, by a terminable worker, a per-epoch budget, and explicit opt-in.
- **Measurements, not estimates** (§7, §10): chaos results, microbenchmarks for every hot path, and the five scale limits we hit.

1.2 Non-goals and the honest claim

Worth being blunt. hostnought is slower than a server, more expensive at small scale, and considerably more complicated. A five-dollar virtual machine beats the entire swarm on every conventional axis, and we do not expect that to change. It provides no anonymity against a global passive adversary. It cannot guarantee deletion against peers who decline to cooperate. It is not bot-free; the compute layer runs untrusted code as a matter of design.

What it offers instead is one property, novel we think in combination: community data can survive the loss of its infrastructure, because the community's own devices carry it, verify it, and can rebuild it from any surviving quorum. Judge the design on that.

2 System Model and Assumptions

Participants. A community of users, each running the application in one or more browsers. Casual peers come and go, and will happily let the browser evict their storage. Anchors opt into persistent storage and stay available across sessions. Anchors get no special authority: they are untrusted, interchangeable, verifiable, and hold nothing custodially.

Infrastructure. One static object store serves the application bundle, a bootstrap pointer, and optionally a cold archive. It runs no application logic. The cold archive deserves a precise sentence, because it is where an operator's role would grow if it existed: an archive blob is a segment header plus k fragments — full reconstruction — and writing one requires bucket credentials no peer holds, so archiving is an explicit operator act, never a swarm behaviour. Archive blobs are content-addressed and re-verified end to end on open, so an archive can deny service but never inject content, which means *anyone* may host one and peers lose nothing by treating it as untrusted. The deployed swarm runs archiveless by policy: the founders' bucket serves code and ratification metadata and holds no user content; durability rests on the peers and anchors it was designed to rest on. The operator's only measurement is request counts at its own front door, logged with the client address excluded at the source — browser family is recorded for device-mix statistics, an address never is; activity inside the swarm is public to its members by design, and the operator learns it the way any member does. We treat it as honest but replaceable, and if it is seized or blocked, mirrors take over (§9). Peer introduction needs a signalling facility. The design originally treated it as a dumb pipe that could nonetheless see connection metadata; the implementation gives it less than that (§5.3): payloads are encrypted under a key derived from the room identifier and published under a derived topic, so a relay handles opaque ciphertext.

Adversaries. Malicious peers serving corrupted data, lying about their inventory, or trying to deanonymize others. Sybil adversaries. An adversary who seizes or blocks the static host. A supply-chain compromise. And, for anonymity claims specifically, a local passive adversary running honeypot peers. We exclude the global passive adversary, and say so wherever it matters.

Churn. Peer availability is diurnal: a substantial concurrent population at peak, a small one at trough, plus a monthly rate of permanent departure.

Trust. No peer is trusted for correctness. Correctness comes either from cryptography (signatures, content addresses, Merkle proofs) or from redundancy (erasure coding, repeated computation). Availability and durability are trusted only probabilistically, under the models above.

3 Data Model

3.1 Events

Every action is a signed, content-addressed, immutable event carrying an identifier (BLAKE3 [3] over the canonical body), an application namespace, a type, an author public key, an optional parent, a scope, an advisory timestamp, a body, and an Ed25519 signature [4]. Canonical encoding is deterministic CBOR [5]. The hash covers the encoding, the signature covers the hash.

Nothing is authoritative. Any peer may serve any event, and the signature plus the content address let a receiver check it without trusting whoever sent it. Events belonging to applications a peer has never heard of are stored and repaired anyway but never folded, because peers are storage-neutral carriers.

Device subkeys (§9) attach their delegation certificate outside the canonical body. Attaching or reattaching a certificate therefore leaves the event identifier untouched, and an event hashes identically whether a master key or a device key authored it.

3.2 Folds

All derived state is a pure fold over the event log: an `init`, plus a `step(state, event)` reducer, registered under `(application, name, version)`. Every peer replays the same events in the same total order, timestamp ascending with identifier breaking ties, and arrives at the same state. There is no database and no migration, and write conflicts cannot happen, because nothing is written. A total order over immutable events also sidesteps the merge problem CRDTs exist to solve [6]; applications get convergence from the order rather than from algebraic constraints on their state.

Revocation is the one thing the engine handles for itself, ahead of every fold. A `device_revoke` signed by a device's master suppresses that device's events falling after the revocation in total order. "After" is a property of the order rather than of any single event, which is precisely why it cannot live inside signature verification.

3.3 Segments and Erasure Coding

Events are batched into immutable segments, sealed at 256 KB or 24 hours, whichever arrives first. The pipeline runs canonical ordering, then deterministic CBOR, then compression (native `deflate-raw`, falling back to `lz-string`, with the choice recorded in the header so mixed swarms interoperate), then Reed–Solomon [7] at k=8, n=20, widening to n=40 inside the hot window. A segment's identifier is the Merkle root of its event identifiers, so it is addressed by what it contains rather than by how it happened to be stored.

Verification happens at two levels, for a reason. Carriers check a fragment against the header's fragment root without holding any events at all, which is what lets them carry data for applications they do not run. Readers check everything on open: each signature, then the event Merkle root against the segment identifier. A forged header can survive carriage. It cannot survive an open.

Blob segments carry raw bytes instead of events and are addressed by the BLAKE3 of those bytes. Their reason for existing is compute output (§5.2): a job result's hash *is* the segment identifier of its stored output, which makes result references self-certifying with no extra machinery. Since replicated workers have to produce byte-identical segments, the blob codec is pinned rather than negotiated per browser, and a runtime lacking native compression declines to seal instead of emitting a variant nobody else can reproduce.

3.4 Placement

Placement is rendezvous (highest-random-weight) hashing [8], weighted by demonstrated persistence:

```
score(peer, slot) = uptimeWeight(peer) × hash(segmentID ‖ slot ‖ peerID)
```

The top-scoring peer for a slot is its ideal holder. Every peer derives the same map from gossip alone, so there is no directory and no distributed hash table in the Kademlia tradition [9]. Uptime weight measures the fraction of a recent epoch window a peer's heartbeats covered, which rewards persistence rather than mere presence. Anchors take three slots per segment where a casual peer takes one.

4 The Repair Loop

A correction, kept visible. An earlier draft of this paper claimed the repair loop runs in service workers. It does not, and it never did. Service workers have no access to WebRTC, and a repair loop that cannot talk to peers is not a repair loop. In the implementation it runs in the leader tab, elected across a browser's tabs by the Web Locks API (§5.1), which owns the mesh, the repair loop, the job scheduler, and every IndexedDB write. We leave the correction in the text rather than silently rewording it: a paper about a verifiable system should keep its own claims auditable.

4.1 Gossiped State

Peers gossip in the epidemic style [10]: heartbeats carrying an epoch number and a capability advertisement, jittered by ±20 seconds so presence patterns blur, and every five minutes a Bloom-filter [11] inventory advertisement. Merging those filters yields an estimated live fragment count for each segment.

4.2 Deficit Detection and Repairer Election

A segment holding fewer than 12 of 20 fragments, or 24 of 40 for hot segments, needs repair. Both thresholds sit above k deliberately, to absorb Bloom false positives. The repairer is whichever online holder has the lowest `hash(segmentID ‖ epoch ‖ peerID)`, with a small redundancy factor.

The election turns on a distinction that is easy to miss. A *received* claim scoring lower than your own proves some real peer is already working. A locally computed hash proves nothing of the kind, because the peer it names may be long gone. Epochs run ten minutes.

4.3 Repair Execution

The elected repairer pulls any k fragments, reconstructs the segment, re-encodes the missing indices, and pushes them to their ideal holders. Holders verify before storing, so an uninvited push costs nothing. Holders also re-push under-replicated fragments to their assigned holders without decoding, which spreads custody around and stops full-custody repairers from becoming places where a lot of data can die at once. Sampled `WANT` probes audit claimed custody. A peer that cannot answer for a fragment it advertised loses its Bloom filter, so phantom fragments get repaired rather than believed.

4.4 Repair Cost

Reed–Solomon repair pulls k fragments to rebuild one, an eightfold bandwidth tax intrinsic to the code. Three things blunt it. Repair is batched per segment rather than per fragment. The hot window's higher n makes repairs rarer exactly where activity is highest. And each peer works under a bandwidth budget with a deficit-priority queue. We track the repair-to-usage ratio as a health metric rather than burying it. Locally repairable codes cut exactly this cost in production storage systems [12] and remain future work.

5 Coordination, Compute, and Overlay

5.1 Tab Coordination

A user with five tabs open should count as one peer. Every tab races for a Web Locks lease. The winner owns the mesh, the repair loop, the job scheduler, and IndexedDB, and the losers proxy through a BroadcastChannel bus. The lease releases on its own when the leader's tab dies, so the next tab promotes in milliseconds and no heartbeat protocol is needed.

5.2 Verifiable Compute

A job is a pure WebAssembly [13] module with zero imports. That single constraint carries the entire argument about what a job can *do*. A module with no imports has no clock, no randomness, no network, no DOM, no storage, no syscalls. It reads input bytes and returns output bytes. It cannot exfiltrate anything and it cannot touch the machine underneath it. The runtime enforces this by handing the module nothing to import, and it rejects modules that declare unbounded or oversized memory or that break the ABI, which is `memory`, `alloc`, and `run`, with `run` returning a packed pointer and length.

Workers are elected as the lowest `hash(jobID ‖ epoch ‖ peerID)` among capable peers, up to the replication factor. Each runs the job and gossips a signed result. Acceptance is by hash majority,

and an elected worker that disagrees with the majority is slashed, forfeiting its accumulated uptime weight and its candidacy in later elections. One subtlety matters here. Each verifier must tally only peers elected under its *own* view of the roster. Otherwise an attacker grinds ephemeral identities offline until one wins, then stuffs the ballot without ever having participated.

What a job can *cost* is a different question, and one the design initially left open. Three bounds answer it:

1. **Terminable execution.** Compilation and validation happen on the caller's thread, where no job code runs. Execution happens in a disposable worker, a Web Worker in browsers and `worker_threads` in Node, under a five-second wall-clock deadline. A job that loops forever is stopped by killing the worker, which is the only thing that stops code that never yields, and its memory dies with it. We tested this against a hand-written infinite loop. It dies at 300 ms.
2. **A per-epoch budget.** A peer spends at most 30 seconds of wall clock per epoch on untrusted jobs. Past that it declines, and the job's other elected workers carry it.
3. **Opt-in.** A peer advertises compute capability only once its user has turned that on. Loading an application does not conscript a browser into running jobs. If nobody volunteers, jobs do not run and the features depending on them degrade, which we prefer to borrowing cycles nobody offered.

We bound wall clock rather than instructions. Fuel metering would need whole-module instrumentation and we have not done it. The cost of that choice is small: a spinning job dies at its deadline, produces no result, and expires exactly as though nobody had run it.

5.3 The Neighbour Overlay

The design originally assumed peers connect to one another and said nothing about which ones. The first implementation inherited a policy from its transport library, which connected to every member of the room. That imposes a ceiling. Browsers get unhappy somewhere past fifty WebRTC [14] connections, and flood gossip crosses $O(n^2)$ links on the way there.

The overlay now keeps apart two things that library had conflated. The roster is who exists, learned from signalling announcements. The link set is who we are actually connected to. A planner picks roughly eight neighbours by ranking *unordered pair* hashes, so both endpoints of a candidate link score it identically and target sets come out largely mutual. Soft and hard caps, at twelve and sixteen, with hysteresis between them, stop connections flapping every time the roster wobbles. Connection count is now independent of community size, and gossip floods across the overlay instead of across a clique. Simultaneous offers resolve by perfect negotiation, politeness assigned by identifier order so both sides agree about who yields.

Random graphs are connected with overwhelming probability once average degree clears a low threshold [15], so flood reachability survives a degree this small. We check it mechanically anyway (§7.3), because "with overwhelming probability" and "in our implementation" are different statements.

Signalling is a small client over public relays, using ephemeral signed events that relays never store. Payloads are encrypted under a key derived from the room identifier and tagged with a derived topic. What a relay learns, then, is that somebody published ciphertext under an opaque topic. Not who is joining what, and not the session descriptions.

5.4 Off-Main-Thread Execution

Signature verification and Reed–Solomon coding are the substrate's two CPU-bound hot paths, and both used to run on the thread that draws the interface. Batch verification and erasure coding

now run in a worker pool, and boot replay yields to the browser between fold chunks when the log is large. Every pooled path keeps a synchronous fallback, so the pool is an offload rather than a dependency, and a runtime without workers gets slower instead of broken.

5.5 Isolation Inside the Browser

It is worth stating what kind of machine this system actually runs on. A browser is an edge compute node we do not administer: an operating system we do not control, other tenants beside us, and no root anywhere. Everything in this section — coordination, compute, offload — therefore rests on isolation boundaries composed entirely from web primitives, and it is worth enumerating them as a set, because they fail as a set.

Between untrusted code and the machine. A compute job is a zero-import WASM module in a disposable worker with a hard deadline and an epoch budget (§5.2). The job cannot reach the machine; the machine can always kill the job. In edge-computing terms: a process with no syscall table and a mandatory kill switch.

Between worlds. A room is a full tenant, not a filter. Each room gets its own IndexedDB database, its own mesh room and derived signalling topic, its own tab-leadership lock, its own cross-tab channel, and its own identity. Two rooms in one browser share code and nothing else. The developer playground is not a special mode — it is simply a room, which means its isolation is enforced by the same mechanism as everything else rather than by a flag someone must remember to check.

Between the tabs of one user. A user with five tabs open must count as one peer, so the tabs of a room elect a leader through the Web Locks API (§5.1) and followers proxy through the room's channel. The browser is doing the job an init system does on a conventional edge node: one instance of the daemon, supervised, restarted on death — except the browser's lock manager provides all of it for free.

The lesson, learned the honest way. Isolation is a property of every shared primitive, not of the obvious one. Storage was room-scoped from the first playground release. The leadership lock and the cross-tab channel were not — they were origin-global — and the consequences were exactly the two failures the scoping discipline exists to prevent: playground rooms ran permanently leaderless while any production tab was open (no mesh, no sealer, no compute — a starved tenant), and a playground tab could proxy its events to the production leader through the shared channel — the precise contamination path the storage scoping had closed, reopened one primitive over. Both were found within minutes of driving the first interactive compute job through the deployed client, and neither was findable by the unit suite, which never runs two rooms in one origin. The rule we now apply: enumerate the browser's origin-global namespaces — databases, locks, broadcast channels, caches — and scope every one by world, or the isolation story is the story of the one you forgot.

6 The Application Model

6.1 Applications as Folds

An application is a triple: event schemas, pure folds, and optionally verifiable jobs, plus whatever user interface it wants. Registering it wires the folds into the engine. Applications share the swarm, and peers store and repair fragments for applications they do not run. Deploying one needs no registration authority and no change to the substrate.

6.2 Application One: Forum

Threads, comments, votes with last-vote-wins per author, first-claim name badges, and moderation. Moderation applies labels under a per-user "respect moderation" toggle. Labels hide; they never delete. Turn the toggle off and the append-only content comes back. Board founders moderate, and may grant moderation to others.

6.3 Application Two: Search

Sealing a segment triggers an index job. Elected workers build an inverted index shard whose construction is bit-deterministic, with sorted terms, sorted postings and canonical encoding, so hash majority can verify it. Shards are stored as blob segments under rendezvous custody. Queries scatter to the peers computed to hold the relevant shards, and results are merged and ranked client-side with BM25 [16], which needs no global view because scoring happens over the postings that came back. The honest consequence is a delay: freshly posted content is not searchable until its batch seals.

6.4 Application Three: Private Messaging

Direct messages are ciphertext events that the swarm repairs without being able to read them. Each device derives an X25519 [17] encryption key, and each message is sealed under an ephemeral key of its own by ECDH, then HKDF [18], then AES-GCM. **There is no recipient field.** Established messages carry only a routing tag derived from the conversation secret and an epoch counter, so an observer sees that somebody messaged somebody. First contact bootstraps by trial decryption, which stays cheap at community scale.

One correction deserves writing down, because the mistake is easy to make and the symptom is baffling. An outbound message sealed to the *recipient's* key is unreadable by its own author. The sender's history therefore vanishes on reload, and a message to oneself is invisible. The fix is a second sealed copy, an *echo* addressed to the sender, which restores durable sender-side history while looking to any observer like one more indistinguishable ciphertext. The current limits are worth stating plainly. There is no forward secrecy, which a ratchet would fix and we have not built. Timing and existence metadata stay public even when content and participants do not.

6.5 Governed Runtime Application Loading

The promise was always that the substrate "admits others by the addition of a static bundle alone." Making that true meant answering a question the original design never posed. An application bundle runs with the same origin privileges as the substrate. It can read keys. Admitting arbitrary bundles is therefore indistinguishable from admitting arbitrary substrate code, and a promise to accept anything is a promise to be compromised eventually.

So application code is governed the way the substrate governs itself. Manifests are admitted by a ratification chain structurally identical to the bootloader's: quorum tallied per (sequence, version, parent) against the same founder council, a persisted anti-rollback floor, and fail-closed behaviour on fork or equivocation. The manifest travels inline and its canonical hash is the ratified version, so swapping a manifest under an otherwise valid signature fails. Bundles are fetched by content address, verified, then loaded at runtime. The resolver mirrors the bootloader's logic but sits one stage above it, which leaves the frozen first stage alone. In development, where there is no genesis and therefore no council, external loading is disabled outright rather than approximated.

The result is the property as promised. An application ships by uploading a bundle and collecting a council signature, with no substrate release, and it is governed exactly as tightly as the substrate.

6.6 The Application-Validity Contract

Two properties hold up a shared substrate, and convention enforces neither.

Folds must be pure and order-deterministic. The same events in any arrival order have to produce the same state, or peers diverge silently, which is the worst available way to diverge.

Folds must tolerate arbitrary input. Any peer can emit any body under any application namespace. A fold that throws on unexpected input is a remote crash button pointed at everyone running that application.

Both get checked mechanically before ratification. The validator loads a bundle against a stubbed host, replays a synthetic corpus of every declared type crossed with malformed bodies, runs that corpus forward and reversed through the real engine and compares the resulting states, and benchmarks fold throughput. Ratification tooling refuses bundles the validator rejects, which turns two requests in a document into a gate.

7 Evaluation

Point measurements say a system worked once; curves say where it stops working. Every figure in this section is generated by a scripted, seeded experiment, and the data behind each figure is committed next to the code that produced it, so the curves regenerate on any machine.

7.1 Durability and Availability

Under §2′s churn model, 40 concurrent peers at peak, three to five at trough, and roughly 10% permanent loss per month, per-fragment loss over a one-week repair window gives a per-segment death probability near 10^{-16} per window, and about 10^{-9} over a decade across ten thousand segments. The repair loop solves durability outright.

Availability at trough is a different question, and it reduces to whether $\lceil k/f \rceil$ anchors are online. With anchors taking three slots each, **three anchors carry the night.** Availability is therefore a social problem of about the size communities already solve when they find moderators, and it does not grow with the community.

7.2 Chaos Results, and Where the Cliff Is

The chaos suite kills 60% of peers every hour for 48 simulated hours, across several seeded trajectories. No segment ever dropped below k. Minimum live fragment counts stayed at or near full custody, 39 or 40 of 40, with a handful of repairs per trajectory. Injected malicious fragments are rejected by Merkle proof. A deliberately dishonest compute worker was caught by hash majority in every trial, and slashed on every node.

A single passing configuration says little about margin, so we swept the kill rate to find the edge.

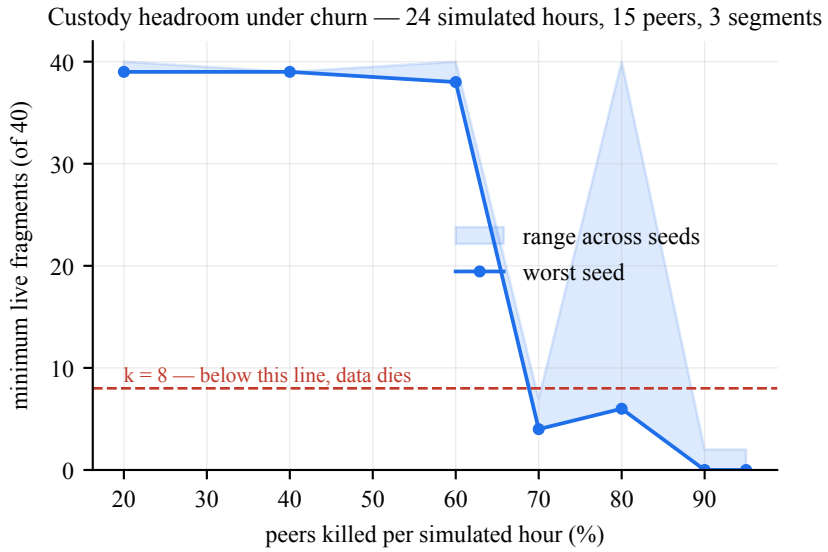

Figure 1: Custody headroom against hourly churn. Fifteen peers, three segments, one repair epoch per simulated hour, three seeds per point. The system holds nearly full custody through 60% hourly churn, and the survival boundary sits between 60% and 70%.

The cliff is sharp and it is not mysterious. Through 60% hourly churn the repair loop keeps custody within two fragments of full, and headroom barely degrades as churn rises — until repair itself becomes the casualty. At 70%, every seed lost data within 24 hours; at 90% and above, the recorded repair count is zero, because the elected repairer is dead before its collect completes. The governing ratio is peer lifetime against repair latency: once the expected survivor fraction of a repair epoch cannot deliver k fragments to anyone, no coding parameters save you. The boundary is also probabilistic rather than clean — one 80% trajectory survived at full custody while its sibling seeds died — which is exactly what a race between repair and death should look like. For the diurnal communities this system targets, 60% sustained hourly loss is already far beyond observed behaviour; the value of the sweep is knowing the failure mode on the other side is repair starvation, not coding weakness.

7.3 Overlay Connectivity

Flood reachability depends on the union of neighbour sets that peers choose independently, so we verify it rather than assume it — first as a test on a sixty-peer room, and then as a Monte Carlo sweep over room size and degree.

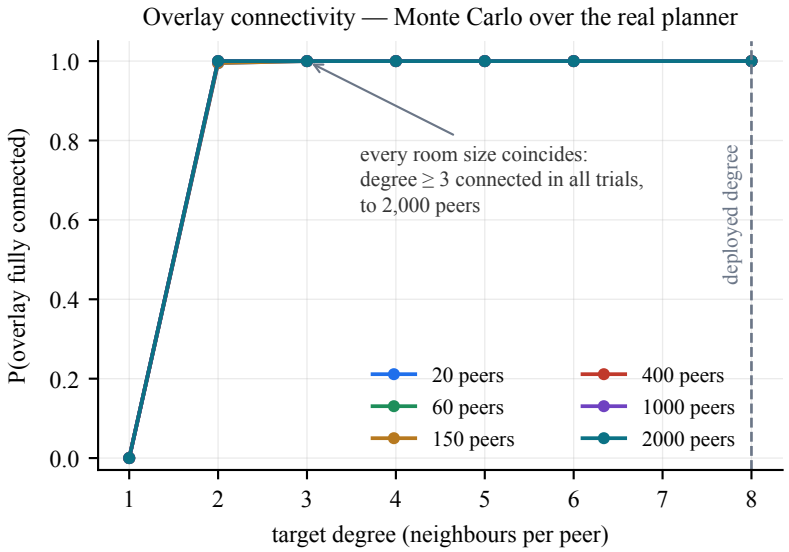

Figure 2: Probability that the union of independently chosen neighbour sets forms one connected component, by target degree, for rooms of 20 to 2,000 peers. Hundreds of seeded trials per point over the deployed planner's own ranking.

Degree 1 never connects — each peer's single favourite forms islands. Degree 2 already connects essentially every trial, and from degree 3 upward not one trial in any room size up to 2,000 peers produced a partition. The union helps more than intuition suggests: links are added from both endpoints' independent choices, so a peer's effective degree is roughly twice its target. The deployed degree of 8 is therefore not the connectivity requirement; it is margin for the things the sweep does not model — churn between plan and dial, NAT failures, and asymmetric reachability — purchased at a connection cost that no longer grows with the room.

7.4 Measured Costs

Microbenchmarks on a development machine, pure CPU. Browser figures are comparable or slightly worse.

Path	Cost
Ed25519 verification	0.80 ms/event (~1,250/s)
Fold insert, in-order	0.79 ms/event, flat in log size
Fold insert, out-of-order (20k log)	2.30 ms/event, growing with N
Fold insert, batched sync (20k log)	0.02–0.08 ms/event amortized
Seal 2,000 events (~430 KB, n=40)	54 ms
Reed–Solomon encode, 256 KB, n=40	16 ms (15 MB/s); n=20, 41 MB/s
Search ranking, 20k documents	13 ms
Search index build, 20k documents	78 ms (job-side)
Application fold throughput (example app)	$\sim 5\times10^{6}$ events/s
Example compute job (byte sum)	$\sim 3.3\times10^{4}$ runs/s, deterministic

7.5 Where It Broke

Measurement found five limits analysis had missed, in the order a real community reaches them.

Client-side fold recomputation. The messaging inbox refolded every message event whenever the log changed, including changes with nothing to do with messages. That is roughly a second per two thousand events, triggered by unrelated activity. An incremental fold now consumes only new events and permanently memoizes failed trial decryptions, on the grounds that a key does not turn up retroactively.

Out-of-order insertion during sync. Backfill arrives carrying older timestamps than what is already loaded, which cost a full re-sort and fold replay per event. Batch insertion, sorting once and replaying once, took 2.30 ms per event down to 0.02–0.08 ms amortized. Sweeping log size shows the shape of the problem and of both fixes:

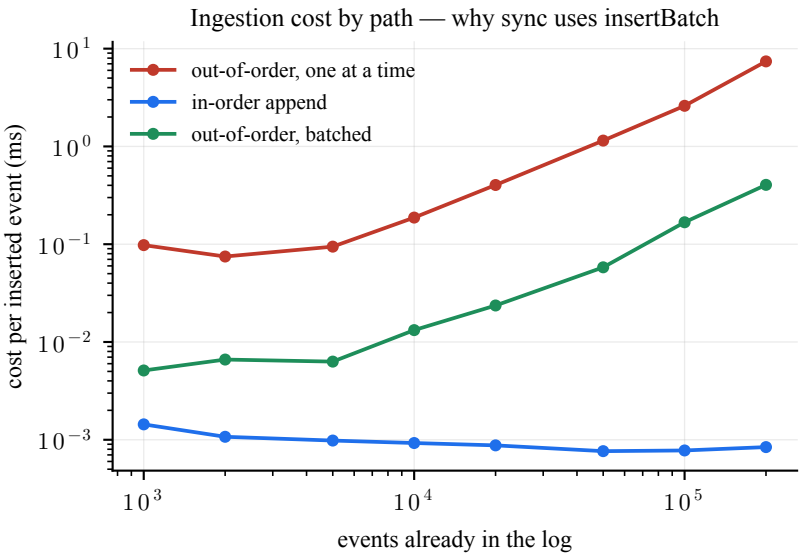

Figure 3: Per-event ingestion cost against log size, log-log. In-order appends hold flat at about a microsecond across two hundred-fold growth; out-of-order singles grow linearly with the log; batching recovers more than an order of magnitude. Eviction caps both rising curves at the resident tail.

The in-order path is flat at roughly a microsecond per event from one thousand to two hundred thousand events. The out-of-order single-insert path grows linearly and reaches 7.4 ms per event at two hundred thousand — over seven thousand times the in-order cost — and even batching, eighteen times cheaper there, still grows with the log. This is why eviction (below) matters twice: after freezing, out-of-order arrivals refold from the checkpoint, so both rising curves are capped at the resident tail size rather than the full history.

Verification-bound startup. Boot re-verified signatures it had already verified before writing them to disk. Local storage sits in the same trust domain as the code reading it, since an attacker who can write there can rewrite the bundle. Re-verification therefore bought nothing and cost seconds, so it is gone for the local store. Untrusted sources still verify, off-thread.

Connection ceiling. Fixed by the overlay in §5.3.

The fifth limit was memory: the log was held in RAM at roughly 50 MB per hundred thousand events. It is fixed by sealed-history eviction.

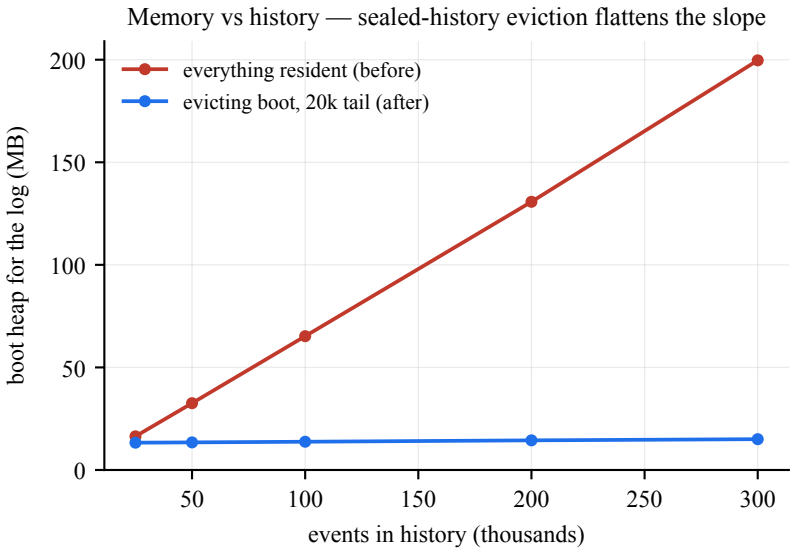

Figure 4: Boot heap attributable to the log, against history size, with everything resident versus an evicting boot retaining a twenty-thousand-event tail. The resident line grows with history; the evicting line stays flat at the tail.

The resident set is now a tail of the log; sealed history is folded into checkpointed base states, remembered in a 32-byte-per-id membership index, and dropped from RAM, with its bytes in IndexedDB and, by construction, reconstructible from swarm segments. Ordinary inserts are unchanged, out-of-order arrivals within the tail refold from the checkpoint at lower cost than the old full replay, and the rare event that lands behind the eviction horizon triggers a correct, streamed rebuild. At a hundred thousand events with a twenty-thousand-event tail, boot memory falls from 65 MB to 14 MB with fold states bit-identical; at three hundred thousand events, from 200 MB to 15 MB. Boot itself streams the log in pages, so memory is bounded by the tail at every point in the lifecycle.

8 Related Work

hostnought descends from a long line of attempts: Freenet [19], GNUnet [20], ZeroNet, Secure Scuttlebutt [21], Dat and Hypercore [22], browser IPFS [23]. All of them died, grew servers, or stayed niche. Scuttlebutt is the instructive one, because its architecture was right and its community was real, and it was killed anyway by storage bloat, missing partial replication, and onboarding that asked too much of newcomers.

Put crudely, hostnought is Scuttlebutt plus erasure coding plus browser capabilities that did not exist when these systems were designed: persistent storage, Web Locks, mature WebRTC, WebAuthn, WebAssembly. The repair loop attacks what killed Scuttlebutt, namely partial replication and unbounded per-peer storage cost. What it does not attack is enumerated in §11 rather than left implied.

9 Threat Model

Honeypot peers and deanonymization. WebRTC shows your IP address to every peer you connect to; that is what it is for. Network identity is therefore a rotating ephemeral key with no derivable relationship to signing keys, which splits participation from authorship into two

cryptographically separate roles. A honeypot learns that a peer at some address carries fragments. It never learns that the address authors particular posts. A peer's own new events also travel two or three hops as a stem before flooding, borrowing the Dandelion insight from Bitcoin's transaction relay [24], so whoever floods an event is usually not its author. The residual risk is real, and we state it in the interface rather than leave people to work it out: a global passive adversary correlating timing defeats all of this, and defeating that adversary takes cover traffic we do not have.

Supply chain. The delivered bundle is the root of trust, which makes the object store a better target than the protocol. The mitigation is a two-stage boot. A small frozen bootloader learns the swarm's genesis descriptor by hash, verifies founder ratifications, tallies quorum per (sequence, version, parent), walks a contiguous chain, refuses anything below a persisted anti-rollback floor, and only then fetches and executes the verified second stage. Any error at all renders an error page and runs nothing. The honest limit: a browser cannot pin its own top-level page, so for a casual visitor the bootloader is monitorable rather than immutable, and a compromised council key defeats the chain until somebody rotates it.

Abuse content. A network that cannot shed abuse content does not deserve anyone's browser. Purge certificates, signed by a community-configured quorum, name target segments. Compliant peers drop the fragments and keep the certificate as an audit record. Peers that refuse cannot be forced, but placement, repair, and queries route around them. This breaks pure immutability, which we would rather do in the open than pretend otherwise.

Sybil attacks. Uptime weight is capped per address prefix and backed by sampled storage audits. Douceur's result stands: without a trusted authority or a cost function, full Sybil resistance is out of reach [25]. Proof-of-work on identity creation is the escape hatch, and we have not needed it yet.

Untrusted compute. Covered in §5.2: no imports, terminable execution, a per-epoch budget, opt-in participation, and an election that keeps an attacker from targeting a machine of their choosing.

10 Implementation and Deployment

This section began life as a plan. It is now a report.

The system is built and deployed, served as a static bundle behind a CDN, with a two-stage verified boot whose ratification chain has advanced through twenty-six signed releases. The test suite runs 180 tests, including the chaos matrix of §7.2.

Every milestone criterion the plan stated was met: a single-user application before any networking; ephemeral peer identity from the first networked release; reconstruction after deliberately killing peers; zero loss under sustained 60% hourly churn; cold start of full history from an archive plus one live peer (demonstrated in testing — the deployed swarm runs archiveless by policy, per §2); rejection of a tampered bundle; detection of a dishonest compute worker; and an application SDK the applications were then rebuilt against.

The final criterion, that a new application ships without modifying substrate code, is met in the strongest sense available to us. An externally authored application loads at runtime from a content address under council ratification (§6.5), rather than merely compiling alongside the others.

Two developer artifacts ship alongside the substrate. A workbench inside the application exposes the live registry, fold state, a signed event composer, and interactive compute submission — a person can hand the swarm a WASM module and input bytes and receive the hash-majority-verified result with no code — next to an isolated playground that is simply a room of its own (§5.5), so experiments cannot reach the production swarm. A command-line tool scaffolds applications

and compute jobs, runs a complete local swarm with the developer's application side-loaded, and applies §6.6′s validity contract before anything can be ratified. We distribute it as a checksum-verified tarball from the same origin as the bundle. A package registry would have been easier, but it would have added a second root of trust governed by somebody else, which is an odd thing to accept in a project whose whole argument is verifiability.

Observability is treated as a product surface rather than a debugging aid. A telemetry view reports peers, custody, repairs, compute votes, and the verified protocol messages crossing the wire, on the theory that a system whose security story is verifiability ought to let its users watch it work.

11 Limitations

hostnought is the wrong choice wherever a server is acceptable, which is most of the time. Its costs in bandwidth, latency, and complexity are worth paying only when infrastructure independence is itself the requirement.

Its durability guarantee is conditional on a community that keeps participating, and it offers nothing at all to one that disperses. Running archiveless (§2) makes that conditionality strict: with no operator-held copy, the trough hours rest entirely on anchors, which is the design's intent and also its cost. Its availability rests on a small anchor class volunteering to persist, which puts something with a server's availability profile back into the picture, albeit a disposable, verifiable and non-custodial one. Its anonymity measures raise the cost of deanonymization without defeating a global passive adversary. Its immutability is imperfect by choice. Its root of trust is still the delivered bundle and the council keys ratifying it. Mobile browsers remain mostly consumers rather than providers, which thins the base of peers carrying anything.

Implementation added three more. Compute bounds wall clock rather than instructions. Freshly authored content is not searchable until its batch seals. And the compute layer is only as useful as the number of people who opt into it, a design choice we would make again but which is not free. That boundary is where the claim stops. We would rather draw it here than have somebody discover it in production.

12 Conclusion

hostnought removes privileged server infrastructure by shipping the entire application as static files and keeping durable state in a self-repairing, erasure-coded swarm of ordinary browsers. Its central mechanism, a coordination-free rendezvous-placed repair loop running in an elected leader tab, gives archival-grade durability under browser-scale churn, with availability governed by a small and replaceable anchor class rather than by how many users there are. A degree-capped neighbour overlay keeps connection cost flat as a community grows. A ratification chain admits applications at runtime without handing them more trust than the substrate has. And a bounded, opt-in compute layer does verifiable work without conscripting the machines that do it.

None of this outperforms a server, and we have not tried to argue otherwise. The contribution is a property no widely deployed system offers: that a community's data can outlive the infrastructure that hosted it. Whether the communities who need that property will find it is not a question a design can settle. The twenty years of systems this one descends from suggest the need is real, and the browser capabilities it leans on have only recently made this shape of answer possible.

13 What Building It Taught Us

A design document records decisions; it rarely records why the alternatives lost. This section does, because several of these lessons cost us real time and each generalizes past this system. It grows as the system does.

Write claims against the implementation, not the intention. The service-worker error of §4 survived into print because the sentence was written from the design's vocabulary rather than checked against what the code could possibly do — service workers cannot open peer connections, so the claim was not just wrong but impossible. Every load-bearing sentence in this paper is now checked against the running system before it ships.

Measurement finds different limits than analysis. Analysis produced the durability math of §7.1 and got it right. It missed all five of the limits in §7.5 — recomputation, insertion order, redundant verification, the connection ceiling, memory — because each lives in the gap between an algorithm and its data structures. The lesson is not that analysis is worthless; it is that the two tools find disjoint bugs, and a system evaluated only one way is half-evaluated.

Asymmetric cryptography breaks symmetric intuitions. Sealing a message to its recipient reads as obviously correct until the author reloads and their own words are gone (§6.4). Encryption to a key is a one-way door, and every party who needs to re-open a message needs their own ciphertext. The general habit: walk each flow as every participant, including yourself tomorrow.

Know where the trust boundary is, and stop paying rent inside it. Boot re-verified thousands of signatures that had already been verified before they were persisted — seconds of frozen startup buying nothing, because an attacker who can write our local storage can rewrite the code doing the checking (§7.5). Verification belongs at trust boundaries; inside one, it is theater with a CPU bill.

A plugin system is a supply chain. An application bundle runs with the substrate's origin privileges, so "load community apps" and "execute arbitrary code" are the same feature (§6.5). Once named, the answer was forced: application code must be governed exactly as tightly as the substrate governs itself, by the same ratification machinery. Any looser gate would have made the bootloader's rigor decorative.

Kill processes; don't meter instructions. For untrusted compute we bounded wall clock with a terminable worker rather than metering fuel (§5.2), because the only thing that stops code that never yields is ending its world, and a killed job is indistinguishable from a job nobody ran — the failure mode composes cleanly with the rest of the protocol. Choosing the enforcement mechanism first made the safety argument short.

Eviction is an ordering problem before it is a caching problem. The memory fix (§7.5) looked like "add an LRU" and was actually two invariants: only reconstructible (sealed) events may leave RAM, and the evicted region must be an exact prefix of the total order, or checkpointed fold states are unsound. The cache was an afternoon; the invariants were the design. Optimizations over ordered logs tend to have this shape.

Volunteers, not conscripts. Compute is off until a user turns it on, even though opt-in makes the compute layer weaker on paper (§5.2). A substrate that borrows cycles uninvited is indistinguishable from the malware it must defend against, and the trust cost of one such story would exceed the capacity value of every conscripted machine.

Distribution is part of the trust story. The developer tooling ships from the same origin as the platform, checksum-verified, rather than through a package registry (§10) — not because

registries are careless but because a second root of trust, governed by someone else, is a strange dependency for a system whose entire argument is that you should not have to trust infrastructure you cannot verify.

Isolation is per-primitive, not per-system. The browser hands an origin several independent global namespaces — databases, locks, broadcast channels, caches — and scoping one of them creates the *feeling* of isolation while the others quietly span worlds. We scoped storage by room on day one and still shipped a leadership lock and a cross-tab channel that were origin-global (§5.5), one of which reopened the exact contamination path the storage scoping had closed. The discipline that works is an enumeration, not an intuition: list every shared namespace the platform touches, and scope each one or write down why not.

Drive the deployed artifact; the suite cannot see whole bug classes. One session of operating the live client — submitting the first interactive compute job through the real UI — surfaced five defects that 187 green tests structurally could not: a UI control disabled by a framework's attribute-serialization quirk on custom elements, the two unscoped isolation primitives above, an acceptance rule demanding a majority of the *requested* replication rather than of the elected committee (so a room with one volunteer could never accept any job, the implementation violating the published protocol specification), and job inputs delivered as an encoding wrapper rather than the submitted bytes. Unit suites exercise modules under assumptions; the deployed artifact composes browser semantics, build tooling, and protocol under none. The fix for the acceptance rule was verified the only convincing way: a live job whose output was the arithmetically correct answer.

Test invariants, not schedulers. A churn test began failing not because custody was lost but because a repair timed out under a saturated test machine's scheduler, and the seeded trajectory then ground custody down deterministically. The invariant under test was "no segment below k with repairs allowed to run" — so the fix was to stop letting scheduler latency masquerade as protocol failure. Simulated-time tests should be generous with real time and strict about state.

References

1. H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. IPTPS*, 2002.
2. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. ASPLOS*, 2000.
3. J. O'Connor, J.-P. Aumasson, S. Neves, and Z. Wilcox-O'Hearn. BLAKE3: One function, fast everywhere. Specification, 2020.
4. D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2), 2012.
5. C. Bormann and P. Hoffman. Concise Binary Object Representation (CBOR). RFC 8949, IETF, 2020.
6. M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proc. SSS*, 2011.
7. I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2), 1960.
8. D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1), 1998.

9. P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. IPTPS*, 2002.
10. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. PODC*, 1987.
11. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), 1970.
12. C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In *Proc. USENIX ATC*, 2012.
13. A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proc. PLDI*, 2017.
14. H. Alvestrand. Overview: Real-Time Protocols for Browser-Based Applications. RFC 8825, IETF, 2021.
15. P. Erdős and A. Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5, 1960.
16. S. Robertson and H. Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4), 2009.
17. A. Langley, M. Hamburg, and S. Turner. Elliptic Curves for Security. RFC 7748, IETF, 2016.
18. H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, IETF, 2010.
19. I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. International Workshop on Designing Privacy Enhancing Technologies*, 2001.
20. K. Bennett and C. Grothoff. gap — practical anonymous networking. In *Proc. PET*, 2003.
21. D. Tarr, E. Lavoie, A. Meyer, and C. Tschudin. Secure Scuttlebutt: An identity-centric protocol for subjective and decentralized applications. In *Proc. ACM ICN*, 2019.
22. M. Ogden, K. McKelvey, and M. B. Madsen. Dat — distributed dataset synchronization and versioning. Whitepaper, 2017.
23. J. Benet. IPFS — content addressed, versioned, P2P file system. arXiv:1407.3561, 2014.
24. S. Bojja Venkatakrishnan, G. Fanti, and P. Viswanath. Dandelion: Redistributing anonymity in Bitcoin. In *Proc. ACM SIGMETRICS*, 2017.
25. J. R. Douceur. The Sybil attack. In *Proc. IPTPS*, 2002.

This draft describes a deployed system. Durability and availability figures are still analytical, under the churn model stated in §2. Everything else is measured, and the chaos results reproduce from the test suite.