

Freenet

A Peer-to-Peer Platform for Real-Time Decentralized Applications

Idempotent Commutative Monoids and Adaptive Small-World Routing

Ian Clarke

ian@freenet.org

<https://freenet.org>

May 25, 2026

Abstract

Freenet is a peer-to-peer platform for building real-time decentralized applications without global consensus. Its core abstraction is a *contract*: a WebAssembly module that defines both the public state of an application and the merge semantics used to reconcile replicas. Rather than imposing a single consistency model, the platform requires each contract to expose an idempotent commutative merge operation (equivalently, a join-semilattice) and an application-defined summary/delta protocol for efficient synchronization.

Each contract has a location on a one-dimensional ring derived from the hash of its code and parameters, and is replicated on peers near that location. Requests are routed using a per-neighbor, distance-indexed performance model rather than topological distance alone. For private state, the platform provides *delegates*: local WebAssembly agents that hold secrets on the user's device and interact with contracts and other software through attributed messages.

This architecture targets decentralized applications such as group chat, collaborative documents, social feeds, and reputation systems, where availability and application-defined convergence are preferable to global linearizability. The paper describes the architecture as currently implemented in the freenet-core reference runtime, includes a 24-hour snapshot of routing behaviour on the deployed network, and is explicit about which mechanisms are deployed, which remain experimental, and which are open problems.

1 The Problem

The modern web is built on infrastructure owned by a small number of companies. Group chat, social feeds, collaborative documents, identity, and reputation all run on it. So does the power to moderate, surveil, and remove the people who use it. This paper describes a peer-to-peer platform aimed at providing the same kinds of services without the same single points of control.

Two broad families of decentralized systems have been deployed at meaningful scale, and neither is a good substrate for interactive applications.

Content-addressed storage. The original Freenet (Clarke et al., 2001) and IPFS (Benet, 2014) provide name-by-hash access to immutable data. Mutable state is whatever the application chooses to layer on top. This is a poor fit for any system whose primary action is mutation: chat rooms, document editing, social feeds. In practice the application reinvents a consistency story per use case, often inconsistently across clients.

Blockchains. Ethereum (Wood, 2014) and successor smart-contract platforms provide programmable mutable state, but their consistency model is global linearizable consensus. Every state transition is ordered with every other transition, and replicated by every participating node. The latency, throughput, and replication cost that follow are reasonable for value-transfer and authoritative ledgers, but they are too expensive for the interactive applications that constitute the bulk of the web.

The gap between these is large. A chat room does not need a global linear order on every message; it needs replicas that converge to the same content. A reputation system does not need every reader to download every signed record; it needs efficient summary-driven sync. A collaborative document does not need consensus on each keystroke; it needs an algebra of merge that the application defines.

This paper describes a peer-to-peer platform built around that observation. Applications carry their own merge semantics, expressed as sandboxed WebAssembly. The platform handles routing, replication, synchronization, transport, and sandboxing generically over whatever algebra the application supplies, alongside a separate primitive (the delegate) for private state and sensitive operations that should not be replicated. The result targets a class of decentralized applications that has historically been awkward to build on either content-addressed or blockchain substrates.

2 Design Thesis

The platform’s central bet is to separate *what merges* from *how it propagates*. The application defines the algebra of its state: an idempotent, commutative merge operation on byte-string state, packaged as a WebAssembly module. The platform defines the rest: where state lives on the network, how peers find each other, how they exchange differences efficiently, how updates fan out to subscribers, how secrets are held locally. The platform is generic over the algebra; the application carries the lattice.

This design choice is what distinguishes the system from both ends of the existing decentralized spectrum.

Against fixed-lattice CRDTs. Conventional Conflict-free Replicated Data Type (Shapiro et al., 2011) frameworks expose a closed catalogue of lattices (counters, sets, registers, sequences) to applications. Each application chooses from the menu. The platform here generalizes the menu: a contract supplies its own merge function, with the validity invariants and tiebreak rules its workload requires. The trade is that the runtime cannot statically verify the algebraic properties: a contract whose “merge” is not associative will not converge. The contract is content-addressed, however, so any deviation is attributable to a specific module that users can choose to use or avoid.

Against global consensus. Blockchain platforms order all state transitions globally. That is more than most decentralized applications need. The system here provides per-contract eventual consistency: each contract’s state converges across its own replicas without requiring agreement with the rest of the network on the order in which its updates were applied. This makes the interactive, conversational class of applications natural, at the deliberate cost of being unsuitable for applications that require a single global order.

Public state and private state are different problems. Real applications have both: a chat room is public, the user’s signing key is private. The platform makes the distinction first-class. Contracts hold the public, replicated state. Delegates hold the private, device-local

state. A clean interface between them, with attributed messages crossing a platform-enforced trust boundary, replaces the usual story of “each application figures out where to keep its secrets,” which has not worked well in practice.

The remainder of the paper describes the primitives that follow from this thesis (Section 3), how updates move through the network (Section 4), how data is located (Section 5), how trust is established and what it does not cover (Section 6), the current implementation status and open problems (Section 7), and the position in related work (Section 8).

3 Core Primitives

This section names the artifacts that appear in the rest of the paper and fixes notation.

3.1 Peers and the Ring

A *peer* is an instance of the Freenet Core running on some host. Each peer p has a *location* $\ell(p) \in [0, 1)$. Locations live on a one-dimensional ring with wrap-around distance

$$d(x, y) = \min\{|x - y|, 1 - |x - y|\}.$$

A peer’s location is derived deterministically from its external network address; the exact derivation and its security implications are discussed in Section 6. Each peer maintains a set of *neighbor* connections, bounded between configurable minimum and maximum sizes (typically 25 and 200). Neighbor relations are reorganized over time by the topology rule described in Section 5.

3.2 Contracts

A *contract* \mathcal{C} is a pair (code, params), where code is a WebAssembly module conforming to the platform’s contract interface and params is an opaque byte string fixed at instantiation. The contract’s *key* is

$$k(\mathcal{C}) = H(H(\text{code}) \parallel \text{params})$$

for a cryptographic hash H . The two-stage form lets a client that already holds $H(\text{code})$ derive the key for a new contract from new parameters without re-hashing the full WebAssembly module, which may be several megabytes. The contract’s *location* $\ell(\mathcal{C}) \in [0, 1)$ is derived from $k(\mathcal{C})$ in the same way as a peer location. Each contract has a *state* $\sigma \in \mathcal{S}_{\mathcal{C}}$, replicated across peers near $\ell(\mathcal{C})$ on the ring. The contract’s code defines $\mathcal{S}_{\mathcal{C}}$, the validity predicate $\text{valid}_{\mathcal{C}}$, the merge operation $\oplus_{\mathcal{C}}$, and the summary/delta synchronization functions (all detailed in Section 4). Contracts are public: their state is readable by any peer that can route to $\ell(\mathcal{C})$.

3.3 Delegates

A *delegate* is a WebAssembly module that runs on a single user’s device, inside the local Freenet Core. It holds private *secret state* (sandboxed by the core) and responds to messages from user interfaces, contracts, and other delegates. The core attaches a verified sender identifier to every inbound message, so a delegate can apply policy based on sender identity in addition to message content. Delegates are local; they do not appear on the ring. The trust boundary they enforce and the cross-device replication story are described in Section 6.

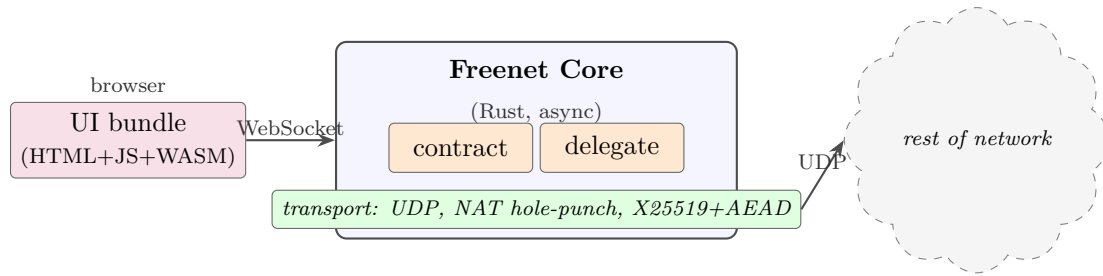


Figure 1: What runs in and around a single peer. The Freenet Core is a Rust process that hosts WebAssembly *contracts* (whose state is public and replicated across peers near the contract’s ring location) and *delegates* (whose state is local to the device and never leaves it). A *user interface* is an HTML/JS/WASM bundle delivered as contract state and run in the browser; it talks to its local Core over a WebSocket. The Core’s transport speaks UDP with NAT hole-punching to neighbors on the rest of the network.

3.4 User Interfaces

A *user interface* is a web frontend (HTML, JavaScript, WebAssembly) delivered by a contract whose state encodes the application bundle. User interfaces run in the browser and communicate with the local core over a WebSocket API. They reach contracts and delegates only through that local core. They are not a distinct kind of network-visible artifact: from the platform’s perspective, the bundle is just contract state with a particular MIME type.

3.5 Operations

A small set of operations connect these artifacts. We describe them informally here; their detailed lifecycles appear in Section 5.

PUT

Publish a contract: insert or update its state at the peers near $\ell(\mathcal{C})$.

GET

Retrieve a contract’s current state by key.

UPDATE

Submit an update to a contract. The update is merged into existing replicas using the contract’s merge function and propagated outward through the subscription tree.

SUBSCRIBE

Register interest in future state changes to a contract.

CONNECT

Establish a peering relationship with the network as a new node.

PUT, GET, UPDATE, and SUBSCRIBE all route over the ring; their destination is $\ell(\mathcal{C})$. CONNECT is the bootstrap operation by which a fresh peer joins the topology.

4 How Updates Move

This section describes the algebra each contract carries, the synchronization protocol that propagates it efficiently across replicas, and the subscription tree along which updates flow.

4.1 Contract State as an Idempotent Commutative Monoid

Definition 1 (Contract state algebra). *A contract \mathcal{C} defines:*

1. a state space $\mathcal{S}_{\mathcal{C}}$ (an abstract set; the wire encoding is a byte string and the contract code defines the encoding);
2. a validity predicate $\text{valid}_{\mathcal{C}} : \mathcal{S}_{\mathcal{C}} \rightarrow \{\perp, \top\}$;
3. an identity element $e_{\mathcal{C}} \in \mathcal{S}_{\mathcal{C}}$ with $\text{valid}_{\mathcal{C}}(e_{\mathcal{C}}) = \top$;
4. a binary merge operation $\oplus_{\mathcal{C}} : \mathcal{S}_{\mathcal{C}} \times \mathcal{S}_{\mathcal{C}} \rightarrow \mathcal{S}_{\mathcal{C}}$, which is associative, commutative, and idempotent, with $e_{\mathcal{C}}$ as identity.

An idempotent commutative monoid is equivalently a join-semilattice; the partial order is $a \sqsubseteq b \iff a \oplus_{\mathcal{C}} b = b$. This is precisely the algebraic structure of a state-based CRDT (Shapiro et al., 2011). We use the monoid presentation because it foregrounds the operational property the platform relies on: peers can merge each other’s state at any time, in any order, with any multiplicity, and arrive at the same result.

The validity predicate is checked at the boundary. A peer that receives a candidate state σ with $\text{valid}_{\mathcal{C}}(\sigma) = \perp$ rejects it and never merges it into its local replica. Validity typically encodes integrity invariants beyond what the type system enforces, for example “every record in the log carries a valid signature by an authorized key.”

An *update* is itself an element of $\mathcal{S}_{\mathcal{C}}$. When a peer holding state σ receives an update u , it computes $\sigma' = \sigma \oplus_{\mathcal{C}} u$, subject to validity. There is no separate notion of operation: the algebra of state is the algebra of change.

Remark (Convergence). *For a finite multiset of valid updates $u_1, \dots, u_n \in \mathcal{S}_{\mathcal{C}}$ and a starting state σ_0 , the merged result $\sigma_0 \oplus_{\mathcal{C}} u_{\pi(1)} \oplus_{\mathcal{C}} \dots \oplus_{\mathcal{C}} u_{\pi(n)}$ is independent of the permutation π by associativity and commutativity, and idempotence makes duplicate delivery harmless. Together these are the sufficient conditions for the Strong Eventual Consistency theorem of Shapiro et al. (Shapiro et al., 2011): peers that have delivered the same set of valid updates converge regardless of order or repetition. The platform’s specific contribution here is not the convergence theorem but the choice to let each contract supply its own monoid, with the consequences described in the rest of this section.*

4.2 Worked Examples

Monotone counter. $\mathcal{S} = \mathbb{N}$, $\oplus = \max$, $e = 0$. The partial order is the usual \leq .

Last-writer-wins register. $\mathcal{S} = V \times \mathbb{N} \times H$, where V is the value type, \mathbb{N} a Lamport-style timestamp, and H a tiebreaker hash. Merge is lexicographic max on the (timestamp, hash) pair.

Signed log. \mathcal{S} is a finite set of records, each record carrying a signature by an authorized key. $\oplus = \cup$, $e = \emptyset$. Validity rejects any state containing a record whose signature does not verify against a key listed in params.

Observed-remove set. $\mathcal{S} = (E, T)$ where E is a set of (elem, id) pairs and T a set of ids that have been tombstoned. \oplus is pairwise union; the materialized set is $\{\text{elem} : (\text{elem}, \text{id}) \in E, \text{id} \notin T\}$.

A bounded variant of the signed log (say, “keep the most recent N records”) is straightforward but requires care: the eviction function must be a deterministic function of the merged set, not

of the order in which records arrived, or idempotence is lost. We treat this as a contract-author obligation rather than a platform-enforced property.

4.3 Where This Sits in the Design Space

Application-defined merge is not itself a new idea: Bayou (Terry et al., 1995) introduced application-supplied merge procedures in 1995, and CouchDB, Riak, and Git’s merge drivers all expose conflict resolution to application code. What is new here is the specific combination of constraints under which the application supplies it.

Compared to Bayou’s lineage of arbitrary procedural merge, contracts are more constrained: the merge code is content-addressed (so the merge identity binds to the data identity and cannot be silently switched), executes in a sandbox under fuel and memory bounds (because untrusted peers run it), and must satisfy the idempotent-commutative-monoid algebra (so the platform can make a convergence promise without inspecting the code). The platform gives up Bayou’s flexibility in exchange for the convergence guarantee and the peer-to-peer trust model.

Compared to fixed-lattice CRDT frameworks (Automerge (Automerge contributors, 2026), Y.js (Y.js contributors, 2026), Antidote (Akkoorath et al., 2016)), contracts are more flexible: the application supplies its own lattice rather than composing from a closed catalogue. We have found this matters in practice. The consistency requirements of a chat room, a reputation contract, and a public file feed are different enough that a fixed catalogue forces several of them to either pay another’s cost or be unimplementable. The price of the flexibility is that the platform cannot statically verify the algebraic properties; a contract that violates them will fail to converge, and the runtime detects this as a downstream symptom rather than as a structural error.

The platform’s position is therefore a middle path between the two: more flexible than the fixed-lattice frameworks, more constrained than Bayou. The summary/delta interface developed in the next two subsections extends the same logic from the merge function itself to the synchronization protocol, which is the part of the design we view as the strongest engineering generalization.

4.4 Summary/Delta Synchronization

Pairwise merge is sufficient for convergence but, on its own, would require peers to ship full state when reconciling. For non-trivial contracts this is prohibitive. The platform therefore extends the contract interface with three additional functions that allow peers to exchange only the difference between their states.

Definition 2 (Sync interface). *In addition to the algebra of Definition 1, a contract provides*

$$\begin{aligned} \text{summarize}_c &: \mathcal{S}_c \rightarrow \Sigma_c, \\ \text{getDelta}_c &: \mathcal{S}_c \times \Sigma_c \rightarrow \Delta_c, \\ \text{applyDelta}_c &: \mathcal{S}_c \times \Delta_c \rightarrow \mathcal{S}_c, \end{aligned}$$

where Σ_c and Δ_c are contract-defined byte-string types.

The summary type Σ_c is intended to be much smaller than full state. For a signed log it might be the set of record identifiers, or a Merkle root over them; for a counter it might be the counter’s value; for a large observed-remove set it might be a Bloom filter, or a pair of Bloom filters over E and T .

For the two-step protocol below to be useful, the contract author is required to ensure that `getDelta` produces something that, when applied, dominates the other peer’s state in the partial order:

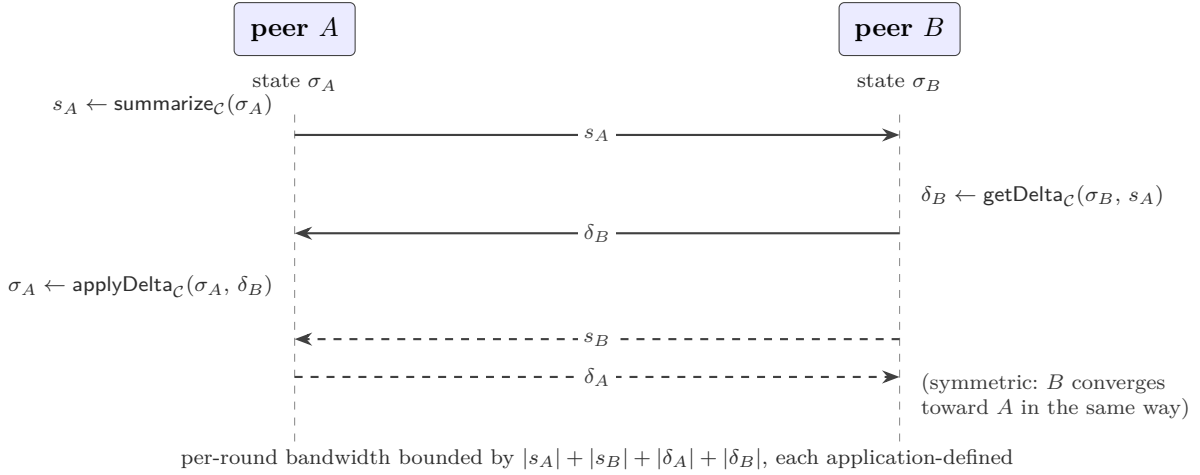


Figure 2: The two-step summary/delta sync protocol. Peer A sends a small summary s_A of its state. Peer B uses s_A to compute a *delta* relative to its own state and ships only the difference back, which A applies under the contract’s merge. The exchange is symmetric (dashed); each direction runs over the same transport. The contract supplies summarize_C , getDelta_C , and applyDelta_C , so the protocol is generic over whatever state representation the contract uses.

Property 1 (Sync soundness). *For all $\sigma_A, \sigma_B \in \mathcal{S}_C$, let $s = \text{summarize}_C(\sigma_B)$ and $\delta = \text{getDelta}_C(\sigma_A, s)$. Then*

$$\text{applyDelta}_C(\sigma_B, \delta) \oplus_C \sigma_A = \text{applyDelta}_C(\sigma_B, \delta).$$

Equivalently, applying δ at σ_B brings σ_B at least as far up the lattice as σ_A would have if merged in full. The platform does not verify this property; the contract is responsible for it.

4.5 The Two-Step Sync Protocol

Two peers A and B reconcile their states for a contract C as follows:

1. A computes $s_A = \text{summarize}_C(\sigma_A)$ and sends s_A to B.
2. B computes $\delta_B = \text{getDelta}_C(\sigma_B, s_A)$ and sends δ_B to A. A applies the delta: $\sigma_A \leftarrow \text{applyDelta}_C(\sigma_A, \delta_B)$.

The protocol is symmetric: B runs the same exchange in the other direction. In practice the two directions are interleaved over a single connection. Bandwidth per reconciliation round is bounded by $|s_A| + |s_B| + |\delta_A| + |\delta_B|$, each of which is application-defined. Contracts that use approximate summaries may require multiple rounds before replicas fully converge (see the next subsection), in which case the total bandwidth is the sum of these per-round bounds.

4.6 Malformed Summaries and Malicious Deltas

The honest-but-stale case is the easy one. Two cases require more care.

Approximate summaries. A summary type such as a Bloom filter has false positives: B may believe A already has a record that in fact A does not. In the bare protocol, the record is then not included in the delta and the two replicas remain divergent. Contracts that use

approximate summaries must therefore either fall back to a more precise second round when divergence is suspected, or accept the false-positive rate as part of their convergence story, relying on subsequent reconciliation rounds (driven by later updates, subscription renewals, or explicit re-sync) to eventually repair the divergence. The platform offers no built-in fallback; the contract designs the round structure. Diagnosing when the round structure is buggy or insufficient is one of the open problems noted in Section 7.

Malicious peers. A peer that supplies a deliberately malformed summary, or returns a delta that violates the validity predicate, is treated like any other source of invalid state. The receiver applies $\text{valid}_{\mathcal{C}}$ to the merged result; if the result fails validity, the delta is rejected. A more subtle attack is for a peer to supply a delta that is valid but withholding (claiming to be up to date when it is not). The protocol cannot distinguish such a peer from one that has simply not yet received the missing updates; the replication factor across multiple peers near $\ell(\mathcal{C})$ is the structural defense, not the sync protocol itself.

4.7 Subscription Trees

Once a contract is published, peers near $\ell(\mathcal{C})$ host its state and additional peers register as subscribers. A subscription forms a soft-state link from the subscriber back toward the contract’s home location; collectively these links form a tree rooted near $\ell(\mathcal{C})$. An update arriving at any node of the tree is merged into local state and propagated outward along the tree, with each pairwise hop using the summary/delta protocol above.

Because $\oplus_{\mathcal{C}}$ is commutative, simultaneous updates injected at multiple points of the tree do not require coordination. They propagate along the edges, meet at internal nodes, and merge; the order in which different updates arrive at a given node does not affect its final state. Subscriptions are lease-based: in the deployed configuration a lease lasts 8 minutes and is renewed every 2 minutes by a background task. A subscriber whose lease expires must re-subscribe. The leasing makes the tree self-healing: orphaned branches die out within a small multiple of the lease period.

4.8 Relationship to Other Mechanisms

Versus operation-based CRDTs. An operation-based CRDT propagates individual operations and relies on a causal-broadcast layer to ensure exactly-once, causally ordered delivery. The protocol here propagates state and tolerates arbitrary loss, reordering, and duplication; the merge function does the work that causal broadcast would otherwise have to.

Versus Merkle anti-entropy. Distributed key-value stores commonly use Merkle-tree anti-entropy (DeCandia et al., 2007) to discover differences between replicas. That is a special case of the protocol here in which the summary is a hash tree and the delta is the subtree the other replica is missing. A contract whose data structure is amenable to a more compact summary (a list of identifiers, a counter value) pays a single round trip rather than $O(\log n)$ of them.

Versus rsync. `rsync` (Tridgell, 1999) reconciles files by exchanging block-level checksums. It is generic but oblivious to application structure. Summary/delta is the same shape of protocol lifted to a level where the application defines the granularity.

Versus delta-state CRDTs. The closest formal analogue is the delta-state CRDT literature (Almeida et al., 2018), which ships only changes between replicas of a fixed-lattice CRDT. The

differences here are that the lattice and the delta-encoding are both supplied by the contract, and the result is embedded in a small-world routing layer rather than a fully-connected replica set.

5 How Data Is Found

A request for contract \mathcal{C} originates at some peer and must reach a peer that hosts \mathcal{C} 's state. This section describes the ring topology, the rule that constructs it, the routing logic that uses it, and the bootstrap by which a new peer joins. The section closes with a 24-hour measurement of routing path lengths on the live network.

5.1 Locations and the Ring

A peer's location $\ell(p)$ is computed by hashing its external network address into $[0, 1)$. A contract's location $\ell(\mathcal{C})$ is similarly derived from its content-addressed key. Distance is the wrap-around metric of Section 3.

This is the simplest workable arrangement, and its trade-offs are worth being explicit about. Locations are stable across restarts that preserve the peer's address but change when a peer is rehomed, when NAT mappings shift, or when a peer is reachable on multiple addresses. An adversary in control of many addresses (a cloud subnet, an IPv6 range, a small botnet) can grind locations and concentrate them in chosen regions of the ring. The platform does not attempt to solve Sybil resistance at this layer; defenses are application-level (signed contributions, reputation contracts, capacity-limited delegation) and are discussed in Section 6.

5.2 Why Small-World Routing

The platform's routing strategy traces to Kleinberg's small-world result (Kleinberg, 2000). Place N peers on a ring (or any metric space). Give each peer a constant number of *short-range* links to its immediate neighbors on the ring, plus a small number of *long-range* links sampled from a distribution that scales as $1/d$ in the ring distance d . Kleinberg proved that a node who knows only its own neighbors can route a message to any target in $O(\log^2 N)$ hops by greedy forwarding: at each step, send the message to whichever neighbor is closest to the target. No global knowledge, no flooding, no routing-table maintenance beyond keeping the link distribution roughly $1/d$.

The $1/d$ distribution is what makes the bound work. Too short and the long-range links don't help cover ground; too long and they overshoot the target. $1/d$ produces, at each scale, a constant probability of a useful long-range hop, so a request roughly halves its remaining distance at every step until the local neighborhood dominates.

The practical question is how the $1/d$ distribution gets there. Kleinberg's analysis assumed it as a given. Symphony (Manku et al., 2003) sampled the long-range links from a harmonic distribution at join time and kept them. The platform here takes a different tack: each peer continuously *steers* its own neighborhood toward the $1/d$ shape, by choosing the destination of each new CONNECT request from the under-represented region of its current connection distribution (*gap-targeting*, Section 5), and by preferring incoming connection candidates that close large gaps in its existing distribution (*Kleinberg-style score*, also Section 5). The active steering is what lets the network preserve the $1/d$ shape under churn; this is the mechanism we describe in the rest of the section.

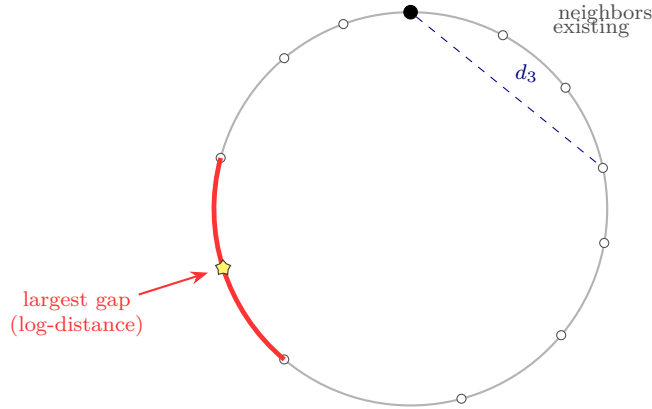


Figure 3: Gap-targeting from a single peer’s view. The peer measures ring distance d_i to each existing neighbor (one such d_i shown dashed), identifies the largest gap in log-distance space (red arc), and chooses its midpoint as the next **CONNECT** destination (star). The peer thus pulls its own connection distribution toward $1/d$ as it accumulates neighbors.

5.3 Topology Construction

The small-world topology is built by the **CONNECT** operation. A joiner chooses a *desired location* on the ring, sends a **CONNECT** message routed greedily toward that location, and the peer that terminates the route decides whether to accept. Both ends of this exchange are more involved than the original “accept only at the terminus” framing suggests, and both are actively shaping the topology toward Kleinberg’s $1/d$ ideal (Kleinberg, 2000) rather than relying on it to emerge.

Desired location. A peer with fewer than K_0 existing connections targets its own location, with random jitter applied after consecutive acceptance failures so that subsequent attempts probe different ring regions. Joiner locations are derived from network-address prefixes (Section 6), so distinct joiners hit distinct early targets and bootstrap traffic naturally spreads across the gateway’s neighborhood. A peer with at least K_0 connections instead computes a *gap target*: it measures the ring distance from itself to each existing neighbor, identifies the largest gap in that distribution in log-distance space, and targets the gap’s midpoint. Gap-targeting is the active mechanism by which each peer pulls its own connection distribution toward $1/d$ as it accumulates neighbors: the locations under-represented by current connections are exactly the locations the next **CONNECT** targets. The deployed thresholds are $K_0 = 3$ for the initial-join path and $K_0 = 5$ for the steady-state maintenance loop; the slightly higher maintenance threshold lets the local connection distribution settle before refresh **CONNECT**s start gap-targeting.

Acceptance. A peer that receives a **CONNECT** applies a chain of decisions. If it can forward strictly closer to the target, it forwards; this is the original terminus rule. A peer within a small fraction of the ring of the target (the deployed band is 5%) may additionally accept with a proximity-scaled probability while still forwarding, giving the joiner more than one nearby connection per request and softening the strict-terminus discipline. When a request reaches its terminus, the receiver’s choice depends on its current connection count: well below a small threshold it accepts unconditionally to unblock bootstrap; below the configured minimum-connections target it filters by a Kleinberg-style score on the candidate’s location relative to its existing neighbors, with the acceptance floor sliding from generous (early) to selective (near full); at or above the minimum it admits only the best-scoring candidate from a rolling window of recent arrivals. Above the maximum it rejects. When the terminal receiver rejects, the **CONNECT** may also be re-forwarded *uphill* (away from the target, up to a small budget of

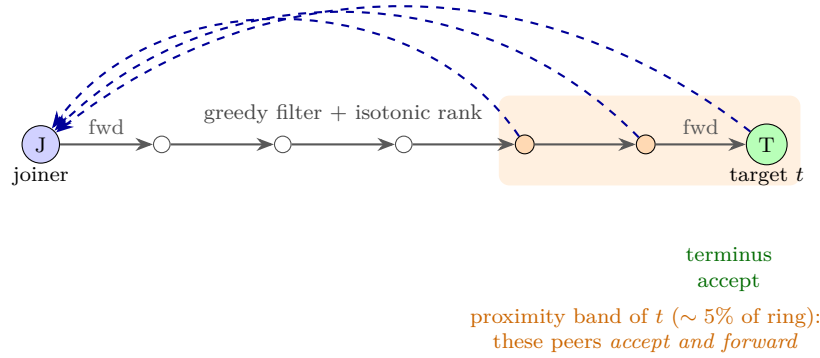


Figure 4: A single CONNECT from joiner J to its desired location t . Each forwarding hop applies a hard greedy filter (candidates strictly farther from t than the current peer are dropped) and then ranks the survivors with an isotonic-regression estimator trained on past forwarding outcomes; the request is sent to the best-scoring eligible neighbor (solid arrows). Once the route enters the proximity band of t (the deployed band is $\sim 5\%$ of the ring, shown shaded), each receiver may *additionally* accept the joiner with a proximity-scaled probability while still forwarding the request onward (dashed return edges). The terminus applies its acceptance chain (§5); a rejected terminus may re-forward the request uphill for a small budget of hops. Net result: one CONNECT request can establish multiple connections close to t , not just one at the terminus.

additional hops, deployed value 8) so that a joiner whose precise target lies behind a saturated peer can still be picked up by a nearby non-saturated one.

Where the long-range edges come from. The long-range component of the connection distribution is the joint output of gap-targeting and Kleinberg-score acceptance, not a side effect of ordinary request traffic. A peer with mostly-local neighbors has its largest log-distance gap at long range, so the next CONNECT it issues lands at a long-range terminus. The receiver, in turn, prefers candidates whose locations close large gaps in its own neighbor distribution. The combination pulls each peer’s neighborhood toward $1/d$ from both ends.

Bounded fan-out. Admission control bounds the connection count between configurable minimum and maximum (deployed defaults 25 and 200).

5.4 Greedy Forwarding

A request from peer p to target location t proceeds as follows. Let $N(p)$ be p ’s current set of neighbors. If some neighbor $q \in N(p)$ is strictly closer to t on the ring than p itself, forward the request to the closest such q ; otherwise p is a terminus for the request and serves it locally.

Each request carries a *hops-to-live* (HTL) counter that decrements on each forward. The deployed maximum is 10. Above a configurable threshold (HTL > 7 by default) the next hop is chosen by random walk rather than greedily. The fallback ensures the request leaves the immediate neighborhood of a peer that happens to be very near t but has not cached the contract yet, and contributes path diversity that helps the topology mix.

5.5 Adaptive Routing

Greedy-by-distance assumes the topologically closest neighbor is also the operationally fastest path. In a real peer-to-peer network this is wrong often enough to matter: neighbors vary by an order of magnitude or more in bandwidth, RTT, and uptime. The router therefore picks the next hop not by ring distance alone but by predicted total cost.

For each neighbor q , the local peer maintains a history of routing events of the form (d, τ, b, f) , where d is the ring distance from q to the request’s target, τ is the observed time-to-first-byte, b is the observed transfer rate, and f is a failure indicator. From this history three estimators are fit:

$$\hat{\tau}_q(d), \quad \hat{p}_q(d), \quad \hat{b}_q(d),$$

the predicted latency, failure probability, and throughput as functions of distance-to-target. Each is fit by *isotonic regression* (Barlow et al., 1972; de Leeuw et al., 2009), the standard non-parametric estimator for a monotone function.

The choice of isotonic regression encodes a structural prior we are willing to commit to (latency and failure are non-decreasing in distance-to-target, on average) without committing to a parametric family for the shape of the curve. The pool-adjacent-violators algorithm fits each curve in time logarithmic in history size; old events age out so the model tracks current conditions.

Given a target t for a new request, the router selects the neighbor that minimizes an expected-cost score combining the three estimators. A neighbor with too little history falls back to greedy-by-distance until enough events have accumulated. Predictions between fitted points use linear interpolation, and beyond the training-data domain the model extrapolates linearly.

Where the adaptive layer applies. For ordinary request traffic (GET, PUT, SUBSCRIBE, UPDATE) the adaptive estimator *replaces* the greedy rule of the previous subsection once the local model has enough data: the router picks by predicted cost, not by ring distance. CONNECT is different. CONNECT additionally retains a hard greedy filter (only neighbors strictly closer to the target than the current peer are eligible) and uses a separate isotonic estimator trained on Bernoulli accept/reject outcomes to rank among those eligible candidates. The hard greedy filter keeps joiner traffic moving downhill so the topology-construction logic in Section 5 converges; ordinary requests do not need that guarantee and benefit from the broader candidate set.

5.6 Lifecycles

The five operations of Section 3 are realized as transaction state machines coordinated by the local core. Each is assigned a transaction id, tracked through completion, and supports timeout, retry, and composite parent/child relationships.

CONNECT. As above. A successful CONNECT establishes a single neighbor relationship. A joining peer issues an initial sequence of CONNECT operations to populate its first K_0 neighbors using own-location targeting; once it has enough connections for gap analysis, all subsequent CONNECT operations use gap-target locations. Topology growth and refresh are driven by additional CONNECT operations issued from a maintenance loop, not as a side effect of ordinary request traffic.

PUT. A PUT carries a contract \mathcal{C} and an initial state σ_0 . The originating peer routes the request toward $\ell(\mathcal{C})$. The first peer with capacity at or near $\ell(\mathcal{C})$ caches \mathcal{C} , checks $\text{valid}_{\mathcal{C}}(\sigma_0)$, and accepts the publish. Nearby peers receive the state through the same path and replicate it.

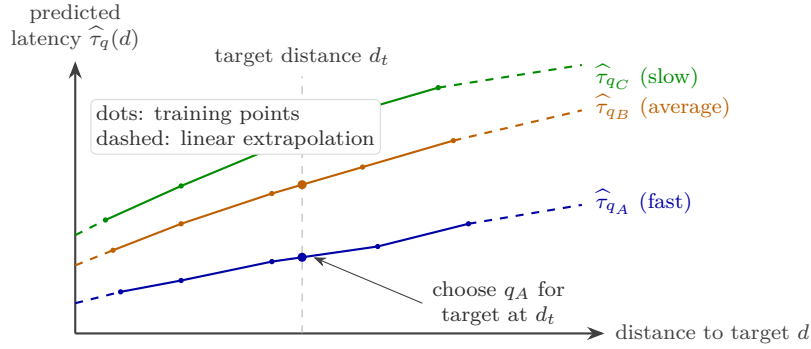


Figure 5: Adaptive routing. Each neighbor q carries a non-parametric, non-decreasing model $\hat{\tau}_q(d)$ of predicted latency as a function of ring distance to the target, fit by isotonic regression (pool-adjacent-violators) on the history of routing events. The fitted points (dots) are connected by linear interpolation; beyond the training-data domain the model extrapolates linearly (dashed). Two more such models per neighbor track predicted failure probability and throughput; one is shown for clarity. For a request to a target at distance d_t , the router reads each neighbor’s prediction at d_t and picks the neighbor that minimizes the combined expected cost. The chosen neighbor need not be the one topologically closest to the target.

GET. A GET carries a key k . The originating peer routes toward the corresponding location. Any peer along the path that currently hosts the contract may answer with its current state σ and parameters; the response travels back along the request path, optionally caching at intermediate peers.

The requester can verify on arrival that the served code and parameters hash to k . *State* is not directly verifiable from the key: the key is derived from code and params, not from state, and state is mutable. This is a meaningful difference from immutable content-addressed stores such as IPFS, where $\text{get}(k)$ returns a value whose own hash equals k . Readers carrying that mental model should note that the guarantee does not transfer to Freenet contracts. State is trusted only to the extent that the contract’s validity predicate accepts it; replication across several peers near $\ell(\mathcal{C})$ is the structural defense against a single peer serving a stale or doctored state.

UPDATE. An UPDATE carries a key k and a candidate state $u \in \mathcal{S}_{\mathcal{C}}$. It is routed toward $\ell(\mathcal{C})$. Each peer along the path that already hosts the contract merges u into its local copy subject to the validity check, then propagates the change outward through the subscription tree (Section 4).

SUBSCRIBE. A SUBSCRIBE establishes a lease on future updates. The subscriber routes the request toward $\ell(\mathcal{C})$ and the first hosting peer responds; intermediate peers along the path record the subscription. Subscriptions are leased and renewed periodically (Section 4).

5.7 A 24-Hour Snapshot of Routing on the Deployed Network

This subsection records a single observation of routing behaviour on the deployed network, as a sanity check rather than as a scaling study. We also include a snapshot of the connection-distance distribution (Figure 6), which is the most direct empirical check on the small-world topology the rest of this section constructs. The 24-hour window ends 2026-05-24. Network size in that window was 443 distinct active peers in the last 10 minutes, with 604 distinct peer identifiers seen across the full 24 hours (the difference attributable to churn). All peers run with telemetry enabled in the current deployment.

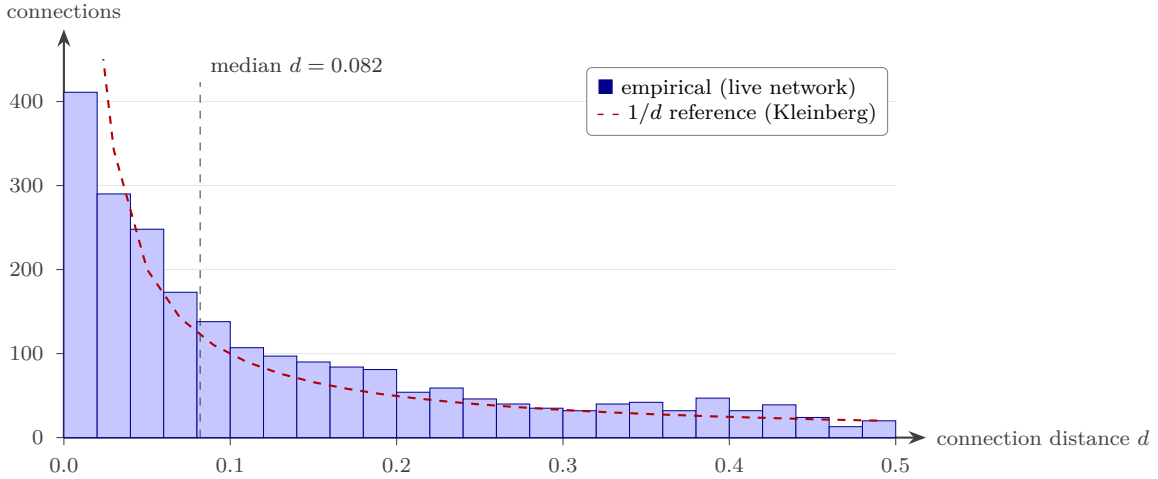


Figure 6: Distribution of per-connection ring distances on the deployed network. Each of the $n = 2274$ live connections across $N = 396$ active peers (snapshot taken 2026-05-25, source: the telemetry collector at `nova.locut.us:3133`) contributes one data point; bins are linear, width 0.02. The dashed curve is a Kleinberg $1/d$ reference normalized to the same total mass. The empirical distribution is monotonically decreasing in d at short range and stays above the $1/d$ reference in the long-range tail, consistent with the active gap-targeting and Kleinberg-style acceptance of Section 5 steering each peer’s neighborhood toward the $1/d$ shape Kleinberg’s analysis requires. Median connection distance is 0.082; the maximum half-ring distance is 0.5.

For each GET and PUT transaction we compute a *path length*: the number of distinct peers seen emitting the corresponding `*_request` event for that transaction id, which is the number of peers visited (hops + 1). Statistics over operations whose telemetry reached completion in the window:¹

Operation	n	mean	median	p95	p99
GET	1,302	6.84	7	11*	11*
PUT	158	6.01	7	10*	10*

**The deployed hops-to-live cap is 10, so any route that would otherwise visit 12 or more peers is dropped before it can appear in the table. The reported p95 and p99 for GET are therefore at the cap itself; they record where the tail is truncated, not where it actually lives. We are not directly observing the upper tail of the path-length distribution.*

Two further caveats are worth being explicit about. First, this is one observation at one network size. At $N \approx 443$, $\log_2 N \approx 8.8$; the observed median path length of 7 peers (6 hops) for GET is consistent with logarithmic-diameter routing, but a single data point at a single N cannot distinguish $O(\log n)$ from $O(\sqrt{n})$, $O(n^{0.4})$, or a constant. Second, the table does not compare against any baseline: it does not separate the contribution of the adaptive-routing layer (Section 5) from baseline greedy-by-distance, and it does not measure what the path length would be without the random-walk fallback at high HTL.

A real test of the routing claims is a controlled simulation sweep across N using the deterministic harness described in Section 7, with and without the adaptive and random-walk

¹UPDATE (median path length 2, $n = 386$) and SUBSCRIBE (median 1, $n = 5,359$) propagate primarily through pre-existing subscription trees rather than fresh greedy routes, so their path lengths reflect tree depth from the subscriber to the nearest hosting peer, not routing-layer hop counts. They are excluded from the table here.

components. We treat that sweep as the appropriate follow-up empirical work; it is listed among the open problems in Section 7.

6 Security and Trust Boundaries

The platform has several layers, and each authenticates and trusts different things. This section is the consolidated story of what guarantees the platform actually provides, where the boundaries are, and what it does *not* attempt to defend against.

6.1 Transport: Hop-by-Hop Encryption and Authentication

A connection between two neighboring peers is established by an authenticated ephemeral Diffie-Hellman handshake (X25519). The handshake exchanges symmetric keys used to encrypt subsequent traffic with AEAD constructions: AES-128-GCM by default, with ChaCha20-Poly1305 as an alternative for platforms without AES hardware acceleration. The handshake itself is encrypted with the recipient’s long-term identity key, which doubles as a defense against unsolicited probes: a peer that has not received a properly-encrypted hello does not respond at all, making the port effectively invisible to a passive scanner.

This authentication is *hop-by-hop*. A request that traverses peers $A \rightarrow B \rightarrow C \rightarrow D$ is authenticated by A to B , by B to C , and by C to D . There is no transport-level guarantee that the message body at D is the body A originally sent: an intermediate peer can drop, reorder, or modify the payload before forwarding.

6.2 Application Layer: End-to-End Authentication via Validity Predicates

End-to-end integrity is the contract’s responsibility. A contract’s validity predicate valid_C runs at every peer that touches the state. Update messages typically include a signature by an application-authorized key over the update content; the validity predicate verifies the signature and rejects the update otherwise. Because valid_C is itself part of the contract’s content-addressed code, a peer cannot serve a state that fails validity without being caught by every receiving peer that runs the same code.

The combination is the platform’s substitute for a single end-to-end transport channel. Transport gives hop-by-hop confidentiality and authentication of neighbors. The contract gives end-to-end semantic integrity of state.

6.3 NAT Traversal and Bootstrap

Most peers operate behind consumer NAT and do not have a stable inbound address. The transport supports hole-punching: when two peers want to connect and at least one is behind a NAT, both peers send hello packets toward each other’s observed external addresses simultaneously (Ford et al., 2005). The introducer in this scheme, the peer that reports each side’s externally-observed address, is whichever peer routed the connection request between them.

For a brand-new peer joining the network the introducer is a gateway, which is required to have a public IP. For a peer that already has at least one connection, the introducer is one of its existing neighbors. Gateways are therefore load-bearing only during the initial join; once a peer is in the network, all subsequent connections are introduced by other peers, and the network continues to form new connections even if every gateway is offline.

A peer’s location is derived from this externally-observed address. The implications are discussed in the next subsection.

6.4 Location Assignment: A Known Sybil and Grinding Surface

A peer's location is computed by hashing its externally-observed network address into $[0, 1)$, but the hash is taken over the address *prefix* rather than the full address: the high 24 bits of an IPv4 address (so all hosts in the same /24 share a location) and the high 48 bits of an IPv6 address (the typical site-level allocation). The intent is to raise the cost of location grinding: an adversary that wants to place peers at chosen ring locations needs control of multiple distinct prefixes, not merely many hosts inside one subnet.

This is a mitigation, not a defense. An attacker who can rent IP space across many /24s, draw on an IPv6 range from a friendly ISP, or vend VMs across distinct prefixes in a cloud account can still grind locations and concentrate peers in chosen regions of the ring. Within that region they can serve targeted contract requests with selected (or no) responses, and they can crowd out honest peers from acceptance under the topology rule.

The platform does not solve this at the location layer. Two architectural mitigations operate at higher levels.

First, the contract layer's validity predicates make state-doctoring detectable: a malicious peer that returns a state with an invalid signature is rejected by every honest receiver. The attack surface this leaves open is selectively serving stale (but still valid) state, not actively forging state.

Second, the platform expects identity, reputation, and contribution-control to be built as application-layer contracts (see Ghost Keys (The Freenet Project, 2026a) for an example of a contract-mediated anonymous identity scheme). A contract that posts authoritative information can require its writers to hold reputation tokens issued by another contract; an inbox contract can demand a small proof of work; a community forum can require introductions from existing members. The platform's role is to provide a substrate on which these mechanisms can be built, not to embed any particular one in the routing layer.

We treat the absence of a built-in Sybil defense as a deliberate scope decision, not an unintended gap. A platform-level scheme tends to either over-fit (assuming a specific economic or identity model that not all applications share) or under-deliver (a generic mitigation that any sufficiently-resourced adversary defeats).

A reasonable concern about deferring to application-layer contracts is that those contracts are themselves routed and replicated through the same vulnerable layer. An attacker with enough /24 prefixes to crowd a contract's location can both serve poisoned reads of that contract and prevent honest writes from reaching the canonical replicas. The contract layer's validity predicates catch outright forgery but not selective serving of stale-but-valid state. For the application-layer story to bootstrap, something has to be true that the platform alone does not currently provide: geographic diversity of honest peers near safety-critical contract locations, sufficient ring-distance separation among replicas of safety-critical contracts, or out-of-band trust anchors (a bundled root key, a release-managed seed list) for the few contracts that bootstrap the rest. The precise mechanism is open work.

6.5 Delegates: A Trust Boundary Around Secrets

Contracts hold public state. Applications also have private state (a user's signing keys, a private contact list, the decryption material for an end-to-end encrypted inbox) that should never be replicated and should be accessible only through controlled interfaces. The platform's primitive for this is the delegate.

A delegate is a WebAssembly module running inside the local Freenet Core on a user's device. It holds private secret state, sandboxed by the core. Other components (user interfaces,

contracts, other delegates) can interact with it only by sending messages. The core attaches a verified sender identifier to each inbound message, so the delegate’s policy can be “only respond to messages from UI X, originating from contract Y.”

The trust boundary is therefore platform-enforced rather than conventional. Traditional in-process encapsulation (a “private” field on an object) is honored only by code that chooses to honor it; the same address space affords access to memory, file system, and undocumented APIs. A delegate’s secret state is not addressable from any other code on the device. The only way to extract a secret from a delegate is to convince the delegate, on the basis of sender identity and message contents, that releasing the secret is allowed.

The boundary is enforced by the local Freenet Core, which is itself a piece of software running on the same host. A compromised or tampered Core can violate the boundary directly. Defending against this requires hardware-attested execution or TEE-style mechanisms that the platform does not currently provide; the trust model here assumes an uncompromised local Core.

Worked example: signing without exporting a key. A chat application’s UI needs to send signed messages. Without delegates, it must hold the private key. With delegates, the UI sends a (sign, payload) message to a key-manager delegate; the delegate verifies the sender, signs the payload, and returns the signature. The private key never crosses the boundary. A bug or compromise in the UI can produce unwanted signed payloads but cannot exfiltrate the key itself; the delegate is also free to apply rate limits or user confirmation prompts before signing.

Cross-device replication. A user typically runs the “same” delegate on multiple devices, and the platform’s intended pattern for keeping those instances in sync is a shared-secret contract: a contract that holds an encrypted, replicated representation of the shared private state, decryptable only by the delegates that hold the symmetric key. The contract’s merge function provides convergence; the delegate provides confidentiality. This pattern is only partially implemented at the time of writing (see Section 7); the encrypted-contract primitives exist but the end-to-end delegate-to-delegate flow is not yet wired up.

6.6 Resource Accounting and What It Doesn’t Cover

Each peer enforces user-configured limits on bandwidth (token-bucket rate limiting at the transport layer), storage (an LRU cache of hosted contracts with a per-peer budget), CPU, and memory (per-contract WebAssembly fuel and explicit memory bounds, enforced by Wasmtime (Bytecode Alliance, 2026)). Excessive consumption by a neighbor results in disconnection.

This protects the peer against a single misbehaving neighbor. It does not protect against coordinated traffic from many neighbors, against adversarial DoS that uses many cheap identities to fill local queues, or against a network-level abuse story that survives across peer identities. Defenses in that regime probably require some form of accounting that persists across identities, which is one of the explicit open problems in Section 7.

7 Implementation Status and Open Problems

This section describes the reference implementation, makes the implementation status of each major mechanism explicit, and lists the architectural questions we view as open.

7.1 Implementation

The Freenet Core is implemented in Rust as an asynchronous `tokio`-based runtime. Contracts and delegates execute inside Wasmtime (Bytecode Alliance, 2026); both run under per-instance fuel limits and explicit memory bounds so that a runaway or malicious module cannot exhaust the host. Contract state is persisted through a pluggable store layer with implementations on top of `redb` (Berner, 2026) and `SQLite`.

The runtime is organized as a single central event loop, a `tokio::select!` over inbound transport packets, client API events, operation timers, and background tasks, that dispatches into per-operation state machines. The single-coordinator design simplifies reasoning about partial state but limits the throughput of a single peer to what one event-loop thread can sustain. So far this has not been the binding constraint in deployment.

Deterministic simulation testing. A peer-to-peer system is hard to test by ordinary means: many bugs in routing, congestion, or convergence appear only in specific multi-peer interleavings that are difficult to reproduce. The Core ships with a deterministic simulation harness built on Turmoil (Tokio Project, 2026): virtual time, a seeded global RNG, an in-memory socket implementation, and fault-injection knobs for loss, latency, partitions, and reorderings. Tests run the entire network in a single deterministic process and replay bit-for-bit from a seed. Most non-trivial routing, subscription, and contract-execution bugs we have found were first reproduced in simulation.

Content-addressed components. Contracts, delegates, and UI bundles are all addressed by hash. This pushes a lot of complexity off the platform: there is no version negotiation, no upgrade protocol, no need for the platform to understand application semantics during deployment. A new version is published under a new key; the old key continues to work as long as anyone hosts it.

7.2 Implementation Status of Each Mechanism

The following table is the current honest status of the mechanisms described in this paper. “Deployed” means present in the production runtime and exercised on the live network; “Implemented” means present in the runtime but not yet broadly exercised at scale; “Experimental” means available as a build-time option but not in production; “Open” means designed or partially designed but not implemented.

Mechanism	Status	Notes
Small-world ring topology	Deployed	Live network \approx 440 active peers
Greedy forwarding + HTL fallback	Deployed	HTL cap = 10, random walk above 7
Two-stage CONNECT acceptance	Deployed	Greedy forward, accept-and-forward in 5% band, Kleinberg-scored at terminus
Adaptive isotonic-regression routing	Deployed	Per-neighbor distance-indexed model
Summary/delta synchronization	Deployed	Per-contract, application-defined
Subscription trees with leases	Deployed	8-min lease, 2-min renewal
Contract WebAssembly execution	Deployed	Wasmtime, fuel + memory limits
Delegates (local, secret state)	Deployed	Used by River chat (The Freenet Project, 2026b)
Delegate cross-device sync via shared-secret contract	Partial	Encrypted-contract primitives exist; end-to-end delegate flow not wired up
NAT hole-punching transport	Deployed	UDP, X25519 + AEAD
Fixed-rate congestion control	Deployed	Production default
LEDBAT++ congestion control	Experimental	Did not meet our requirements in initial trials; not in production
BBR-style congestion control	Experimental	Build-time option, not the default
Deterministic simulation testing	Deployed	Turmoil-based (Tokio Project, 2026)
Per-peer resource limits (bandwidth, storage)	Deployed	User-configured budgets
Sybil-resistant peer identity / location	Open	Defenses are application-layer (see Section 6)
Network-level incentive / abuse layer	Open	Application-layer reputation contracts are the intended mechanism

The single biggest gap between this paper’s narrative and the deployed system is congestion control. The platform supports pluggable congestion controllers but the deployed default is a token-bucket fixed-rate scheme. LEDBAT++ and BBR variants exist in the codebase as alternatives, but neither has yet been adapted to the workload satisfactorily. Real congestion control across heterogeneous peer-to-peer links remains open work.

7.3 Where the Platform Fits

The combination of contract-defined merge, summary/delta sync, small-world routing, and delegates is well-matched to applications whose consistency requirements are conflict-mergeable in some application-defined sense. Group chat, collaborative documents, social feeds, reputation systems, asynchronous email-like inboxes, and identity directories are all in the well-fit class. A working chat implementation (The Freenet Project, 2026b) and an anonymous identity scheme (The Freenet Project, 2026a) are deployed against the live network.

7.4 Where the Platform Does Not Fit

Several application classes do not fit the model gracefully, and we want to be explicit about them.

Strictly linearizable transactions. A defense against double-spending fungible tokens requires that two attempts to spend the same balance cannot both succeed. The merge model

cannot express “one of these updates is rejected unconditionally”; merge is total. Concretely, fungible-token transfer with platform-level double-spend protection cannot be expressed by a single contract in the current model. Applications that need a strong-consistency primitive must either contain the inconsistency inside a contract whose validity predicate detects and rejects the second update (which forces the conflict into application code, with the loser’s state restored on detection) or layer an auxiliary consensus mechanism on top of the platform.

Abuse-resistant open posting. A purely-open contract is simple to write as a set under union and equally simple to flood. Practical contracts in this class restrict writes to signed records by approved keys, charge a proof-of-work fee per update, or require credentials issued by a reputation contract. The platform’s transport-layer rate limiting and per-neighbor resource accounting do provide a partial floor: a neighbor consuming excessive resources is disconnected, which caps the volume any single attacker can push through any single peer. That bound is useful but limited; it does not stop a distributed flood from many cheap identities, nor a slow-burn pattern that stays within per-peer limits. The platform has no application-level spam defense beyond that transport floor.

Coordinated DoS. Per-peer resource limits protect a peer against a single misbehaving neighbor, not against coordinated traffic from many neighbors using cheap identities. Defenses here probably require accounting that persists across identities, which is the open incentive layer noted above.

7.5 Open Problems

We treat the following as genuinely open and intend them as subjects of future work or empirical evaluation.

1. **Scaling characterization.** The single 24-hour data point in Section 5.7 is consistent with logarithmic-diameter routing at the deployed network size. A real characterization requires scanning N across orders of magnitude. We do not currently have the deployment to do this; controlled simulation studies are an interim alternative.
2. **Failure modes of contract-defined merge.** A contract whose merge is not actually associative or idempotent does not converge; the platform’s handling of this is currently to allow the divergence to be observable but not to take corrective action. Better diagnostics, and where feasible contract-level invariants checked at the platform layer, are open work.
3. **Congestion control.** As above. The current fixed-rate default is a deliberate choice not to ship something we cannot tune correctly, not an endorsement of fixed-rate as the right answer.
4. **Sybil-resistant identity.** The application-layer approach has examples (Ghost Keys) but no clean general story; this is the recurrent question for every decentralized platform, and we do not claim to have solved it.
5. **Incentive / abuse layer.** How a network as a whole resists coordinated denial of service without a centralized authority, while still allowing low-friction joining, is the work that determines whether the network grows past a small community.

8 Related Work

Distributed hash tables. Chord (Stoica et al., 2001), Pastry (Rowstron and Druschel, 2001), and Kademlia (Maymounkov and Mazières, 2002) established the structured-overlay paradigm the ring inherits. These systems organize peers by hashed identifiers in a ring or XOR-metric space and provide $O(\log n)$ lookup with carefully maintained routing tables. Their consistency model is fundamentally immutable: a put-then-get returns the published value. The platform here replaces the immutable cell with a mutable, contract-defined one and replaces explicit finger-table maintenance with the implicit topology construction of Section 5.

Content-addressed storage. The original Freenet (Clarke et al., 2001) introduced hash-based naming of immutable data, in the form of content-hash keys (CHKs), around 1999–2000. IPFS (Benet, 2014) later popularized the pattern as a deployment substrate for web content, with a peer-to-peer block exchange (Bitwap) underneath. Both are fundamentally immutable, with mutability provided externally (IPNS, smart contracts, or higher-level naming) and consistency left to applications. The choice here is to put mutability and consistency inside the platform via the contract abstraction; this is strictly more opinionated.

CRDTs. The theory of Conflict-free Replicated Data Types (Shapiro et al., 2011) is the algebraic foundation the contract layer builds on. The contract interface realizes the state-based (CvRDT) flavor with two extensions: the lattice is application-supplied rather than chosen from a catalogue, and the summary/delta protocol generalizes the implicit “ship the whole state” protocol of vanilla state-based CRDTs. Antidote and AntidoteDB (Akkoorath et al., 2016) are the closest comparison points from the database side, both being CRDT databases for the geo-distributed setting, but they expose a fixed CRDT vocabulary rather than letting applications define their own.

Bayou. Bayou (Terry et al., 1995) is the most direct historical precedent for contracts. Application-supplied merge procedures, executed on the replicas, reconciling concurrent writes that the storage layer cannot resolve generically: this is the same idea, in 1995, on trusted replica servers. The differences are forced by the peer-to-peer setting. Bayou’s merges were full procedures with database access; contracts are sandboxed WebAssembly subject to the idempotent-commutative-monoid algebra. Bayou’s merge identity was per-database configuration; contracts bind merge code to content-addressed keys. Bayou ran on a known set of trusted replicas; contracts run on whatever peers happen to host them. The narrower algebra and the content-addressed binding are the price the platform pays for being able to make a convergence promise without inspecting the merge code or trusting the replica.

Anti-entropy and delta sync. Dynamo (DeCandia et al., 2007) and its descendants used Merkle anti-entropy to reconcile replicas. Delta-state CRDTs (Almeida et al., 2018) formalize a state-based protocol that ships only changes between replicas, very much in the same shape as Section 4; the differences here are that the lattice and the delta encoding are both supplied by the contract, and the result is embedded in a small-world routing layer rather than a fully-connected replica set.

Local-first software and CRDT libraries. The local-first software movement (Kleppmann et al., 2019) argues for application-level convergence on user devices over server-mediated consistency. Automerge (Automerge contributors, 2026) and Y.js (Y.js contributors, 2026) are the canonical embodiments, offering CRDT data structures (JSON-like trees, sequences,

counters) that applications compose into peer-to-peer apps over whatever transport they choose. The platform here shares the local-first goal but inverts the boundary: rather than offering a closed catalogue of CRDT types that applications compose from, it lets each contract supply its own algebra and its own synchronization protocol. The trade is between two design points: library-based CRDT systems offer rigorous convergence for any program built on the library's primitives; contracts let applications go beyond those primitives at the cost of carrying their own correctness obligation.

Secure Scuttlebutt. SSB (The SSB community, 2026) is the peer-to-peer system whose primary abstraction is closest in spirit to a single Freenet contract: a per-user append-only signed log. SSB chose one structure (the log) and one replication protocol (gossip on demand by feed identifier); contracts let each application choose. An SSB feed is straightforwardly expressible as a contract whose merge is set-union over signed records, and many SSB applications (private messaging, blogging, social feeds) are recognizably the same applications the contract abstraction targets.

Hypercore. The Hypercore protocol family (Holepunch, 2026) provides append-only signed logs replicated peer-to-peer, with higher-level data structures (Hyperbee, Hyperdrive) built on top. As in SSB the abstraction is fixed at the log level; as in contracts the cryptographic identity is content-addressed. The closest analogue to a contract is a Hyperbee whose merge semantics are baked into the library; the closest analogue to a delegate is a local-only key-managed agent acting over Hypercore.

Small-world routing. Kleinberg's 2000 result (Kleinberg, 2000) is the theoretical underpinning. The original Freenet (Clarke et al., 2001) predated and informally anticipated some of it. Symphony (Manku et al., 2003) introduced explicit sampling from a harmonic distribution to build $1/d$ long-range links on a ring overlay. The platform here uses neither Symphony's static sample nor pure greedy acceptance. Instead it actively converges toward the $1/d$ distribution at each peer: the joiner's *gap target* chooses the under-represented region of the joiner's own connection distribution as the next CONNECT destination, and the receiver applies a Kleinberg-style score filter that prefers candidates closing large gaps in the receiver's neighbor distribution. To our knowledge the specific combination has not been analyzed in the structured-overlay literature.

Performance-aware routing. Adaptive routing in peer-to-peer overlays has a long history. Isotonic regression specifically has been used as a calibration tool in machine learning (de Leeuw et al., 2009); as far as we are aware, it has not previously been applied as the per-neighbor performance model in a structured overlay's forwarding decision.

Blockchain platforms. Ethereum (Wood, 2014) and successor smart-contract platforms provide programmable consistency, paid for by a globally-linearized replicated state machine. The platform here relaxes the global linearization in favor of per-contract eventual consistency with application-defined merge. The trade is that a different class of applications is natural: the interactive, conversational ones, at the cost of being unsuitable for applications that fundamentally require a single global transaction order.

Federated systems. Matrix (The Matrix.org Foundation, 2026) is the closest non-peer-to-peer system in application surface. It is server-based and federated, and its state-resolution algorithm reconciles divergent room states by a deterministic application-defined rule. The platform here

pushes the same idea into a fully peer-to-peer setting and generalizes the merge to arbitrary idempotent commutative monoids.

The original Freenet. The original Freenet (Clarke et al., 2001) was anonymous, content-addressed, immutable file storage from 2001. Its routing heuristic (forward to the peer whose previous successful routes most resemble the requested key) was an early precursor to small-world greedy routing, but it operated over a routing table that had no explicit notion of location, and the network’s data model was a single immutable blob per key. The platform described here preserves the small-world intuition and the content-addressed naming and replaces almost everything else: explicit locations, contract-defined mutable state, anonymity moved out of the core into application-layer protocols, and first-class delegates and user interfaces alongside contracts. The shared name reflects shared lineage, not shared design.

9 Conclusion

The architecture in this paper rests on a small number of design commitments. Applications define the algebra of their state, packaged as sandboxed WebAssembly, rather than choosing from a fixed catalogue of CRDTs. Synchronization is parameterized by that same algebra through a contract-supplied summary/delta protocol, so the wire cost of reconciliation tracks application semantics rather than raw state size. The network underneath is a small-world ring whose long-range link distribution is actively shaped by gap-based CONNECT targeting and Kleinberg-style acceptance scoring, and whose forwarding decisions are informed by a per-neighbor performance model rather than topological distance alone. Private state and sensitive operations live in delegates: actor-style agents that hold secrets behind a platform-enforced trust boundary, leaving public state to contracts.

This set of commitments targets a class of decentralized applications that has been awkward to build on either content-addressed-immutable or globally-linearizable substrates: group chat, collaborative documents, social feeds, reputation systems. We have been explicit about which mechanisms are deployed, which remain experimental, and which are open problems. The next iteration of this work should address those open problems, and the asymptotic scaling claims in particular, with measurements rather than design.

Acknowledgements

Special thanks to Ignacio Duarte Gomez and Hector Alberto Santos Rodriguez for their substantial engineering contributions to Freenet, and to Steven Starr for his invaluable support with outreach, fundraising, and strategic guidance.

References

- Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 405–414, 2016.
- Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162–173, 2018.
- Automerge contributors. Automerge: A CRDT library for building local-first applications, 2026. <https://automerge.org/>, accessed 2026-05-24.

- Richard E. Barlow, D. J. Bartholomew, J. M. Bremner, and H. D. Brunk. *Statistical Inference Under Order Restrictions: The Theory and Application of Isotonic Regression*. Wiley, 1972.
- Juan Benet. IPFS — content addressed, versioned, P2P file system, 2014. arXiv:1407.3561.
- Christopher Berner. redb: An embedded key-value database in pure Rust, 2026. <https://github.com/cberner/redb>, accessed 2026-05-24.
- Bytecode Alliance. Wasmtime: A standalone runtime for WebAssembly, 2026. <https://wasmtime.dev/>, accessed 2026-05-24.
- Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66. Springer, 2001.
- Jan de Leeuw, Kurt Hornik, and Patrick Mair. Isotone optimization in R: Pool-adjacent-violators algorithm (PAVA) and active set methods. *Journal of Statistical Software*, 32(5):1–24, 2009.
- Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.
- Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *USENIX Annual Technical Conference*, pages 179–192, 2005.
- Holepunch. Hypercore Protocol: A streaming, append-only, signed log for peer-to-peer applications, 2026. <https://holepunch.to/>, accessed 2026-05-24.
- Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing (STOC)*, pages 163–170, 2000.
- Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: You own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 154–178. ACM, 2019.
- Gurmeet Singh Manku, Mayank Bawa, and Prabhakar Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 127–140, 2003.
- Petar Maymounkov and David Mazières. Kademia: A peer-to-peer information system based on the XOR metric. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 53–65. Springer, 2002.
- Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.
- Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 386–400. Springer, 2011.
- Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, pages 149–160, 2001.

Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 172–182, 1995.

The Freenet Project. Ghost Keys: Anonymous, verifiable identities on Freenet, 2026a. <https://freenet.org/ghostkey/>, accessed 2026-05-24.

The Freenet Project. River: A decentralized group chat on Freenet, 2026b. <https://github.com/freenet/river>, accessed 2026-05-24.

The Matrix.org Foundation. The Matrix specification, 2026. <https://spec.matrix.org/>, accessed 2026-05-24.

The SSB community. Secure Scuttlebutt: A protocol for decentralized social applications, 2026. <https://scuttlebutt.nz/>, accessed 2026-05-24.

Tokio Project. Turmoil: A framework for testing distributed systems, 2026. <https://docs.rs/turmoil>, accessed 2026-05-24.

Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, 1999.

Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014. Ethereum Project Yellow Paper.

Y.js contributors. Y.js: Shared data types for building collaborative software, 2026. <https://yjs.dev/>, accessed 2026-05-24.