P2POS Overview

P2POS is a **peer-to-peer OS overlay** built on libp2p. It provides a small set of OS primitives (directory, messaging, storage, accounting, compute, and event-sourcing) and composes them into higher-level features like collaboration and app runtime state.

This documentation summarizes what is currently implemented, the architecture, and the problems this project aims to solve.

Key building blocks:

- **libp2p overlay** for peer routing and messaging
- **Helia (IPFS)** for content-addressed storage
- **OrbitDB** for collaboration state
- **App runtime** for event sourcing + SQL projection
- **SOLID-compatible Pod API** for data access patterns

Problem Statement

Modern apps often rely on centralized services for identity, storage, collaboration, and compute. This creates:

- **Single points of failure** and control
- **Opaque data flows** and limited portability
- **High operational costs** for always-on infrastructure
- **Weak offline/edge support** due to cloud dependencies

Why the Internet Itself Feels Broken

In an agentic-AI world, the user is still a **client of a mixture of services**:

- Data is **dissiminated** across company databases
- Algorithms are **fragmented** across company AI systems
- UX is **fragmented** across company entry points

This produces **Fragmented data + Fragmented algorithms + Fragmented UX**.
The user still needs to use each company's agents, and each company offers its own single "outcome" entry point.

The deeper issue: **the existential nature of the internet is peer-to-peer, not client-server**.
The Web shifted power to servers, placing companies at the center and users at the margin. Other users became "resources behind servers," not peers.

Repairing the Internet's Peer-to-Peer Nature

The **P2POS** vision tries to repair the internet by recreating peer-to-peer over the existing network:

- **User data** stays with the user (SOLID pods)
- **User algorithms** run for the user (Pias/App runtime)
- **User UX** is owned by the user (Pias/App UI)

Companies still exist, but as **moral-person peers** interacting with human peers.

Outcome vs. Chain of Choice

When a user asks a company for an “outcome,” the company returns **its** outcome for the user.
That replaces the user’s **chain of choice** with the company’s chain of choice.

Instead, the user should ask **their own server** for the outcome.
Their server should discover **all peers** (companies and humans), propose possible chains of action, and let **their own algorithm** select the best path **for them**, using **their own data**.

P2POS aims to provide a **peer-to-peer OS overlay** that makes it practical to build apps with:

- **Local-first data** that can sync across peers
- **Content-addressed storage** and verifiable data flows
- **Composability** via small OS primitives instead of monoliths
- **Optional infrastructure** (relays, bootstrap) rather than hard dependency

Architecture Summary

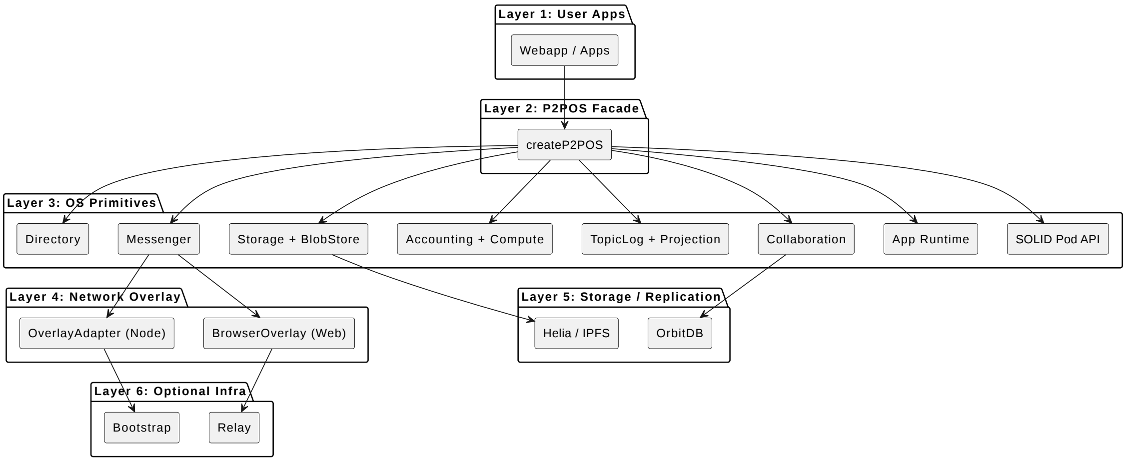
P2POS is layered around a small OS-like core and a libp2p overlay. Each layer is replaceable in tests or different environments.

High-level stack

1. **Apps / Webapp**
   - The test webapp (`webapp/main.ts`) exercises the OS primitives.
2. **P2POS facade**
   - `createP2POS()` wires overlay, messenger, Helia, OrbitDB, and OS services.
3. **OS primitives**
   - Directory, Storage, Accounting, Remote Compute, Messenger
   - TopicLog + Projection (event sourcing + SQL view)
   - Collaboration (OrbitDB key-value per file)
   - SOLID Pod API wrapper
4. **Overlay / Networking**
   - `OverlayAdapter` (Node/TCP)
   - `BrowserOverlay` (WebRTC + WebSockets + relay)
   - gossipsub pubsub for OrbitDB sync
5. **Helia / IPFS**
   - Content-addressed DAG-JSON + raw blocks
6. **Runtime storage**
   - In-memory by default; can be upgraded to persistent stores

Layered Implementation View

This is a more explicit layer map that ties code to each level.



Layer 1: User Apps

- `webapp/main.ts` (test UI)
- Any custom app using `createP2POS`

Layer 2: P2POS Facade

- `createP2POS()` in `src/p2pos.ts`
- Wires overlay + messenger + Helia + OrbitDB + OS primitives

Layer 3: OS Primitives

- Directory: `src/os/directory.ts`, `src/os/libp2p-directory.ts`
- Messenger: `src/os/messenger.ts`
- Storage + BlobStore: `src/os/storage.ts`, `src/os/blobstore.ts`
- Accounting + Compute: `src/os/accounting.ts`, `src/os/remote-compute.ts`
- TopicLog + Projection: `src/os/topic-log.ts`, `src/os/projection.ts`
- Collaboration: `src/os/collaboration.ts`
- App runtime: `src/os/app.ts`
- SOLID Pod API: `src/solid/pod.ts`

Layer 4: Network Overlay

- Node overlay: `src/overlay/adapter.ts`
- Browser overlay: `src/overlay/browser-adapter.ts`
- Protocol: `/p2pos/1.0.0`

Layer 5: Storage / Replication

- Helia + DAG-JSON: `src/os/helia.ts`
- OrbitDB (collab KV): `src/os/orbitdb.ts`

Layer 6: Infrastructure (optional)

- Relay server: `signalling/run.ts`
- Bootstrap server: `bootstrap/server.ts`

Data flow (app events)

1. App calls `node.app.open(appId)` and `append(event)`.
2. Event is written to the **TopicLog** and projected into **SQL**.
3. Envelope is stored in **Helia**; peers receive the CID via Messenger.
4. Peers `catchUp()` to fetch, verify, and project events.

Collaboration flow (files)

1. Each file path maps to an **OrbitDB key-value** database.
2. Updates publish via libp2p pubsub.
3. A messenger-based fallback can deliver updates even if pubsub is sparse.

What Is Implemented

Core

- **P2POS facade** (`createP2POS`)
- **Overlay**
  - Node: TCP + DHT
  - Browser: WebRTC + WebSockets + circuit relay
- **Directory**
  - In-memory (tests)
  - Relay-backed directory for libp2p mode
- **Messenger**
  - Send/receive envelopes over overlay
- **Storage**
  - In-memory storage primitive
  - BlobStore backed by Helia (raw blocks)
- **Accounting**
  - In-memory ledger
- **Remote compute**
  - In-process executor (mock)

Event Sourcing + SQL

- **TopicLog**
  - Signed event log
  - Encryption support (AES-GCM)
- **Projection store**
  - sql.js (SQLite in browser/Node)
  - In-memory fallback
- **App runtime**
  - `append`, `catchUp`, `query`, `exec`, `putBlob`, `getBlob`

Collaboration

- **File collaboration**

  - OrbitDB KeyValue per file
  - Pubsub sync with messenger fallback

SOLID

- **Pod API**
  - `StoragePod` wrapper for pod-style get/set
  - WebID directory integration

Tooling

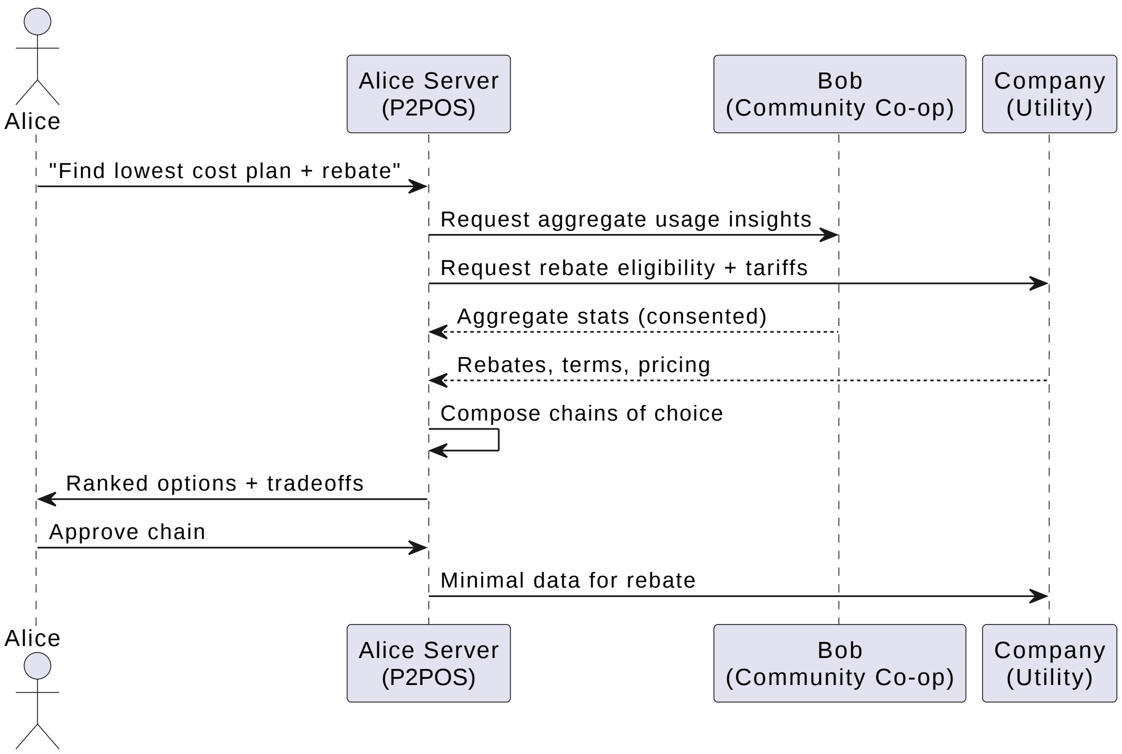
- **Relay + bootstrap servers**
- **E2E tests**
  - Alice/Bob libp2p connectivity
  - Collaboration sync
  - SQL sync (app events)
- **CLI utility**
  - `fetch:collab` to read collab files as a peer

Use Cases

This section shows three end-to-end use cases with Alice, Bob, and a Company. Each flow keeps the user in control and treats companies as peers rather than gatekeepers.

1) Neighborhood Energy Co-op + Rebate

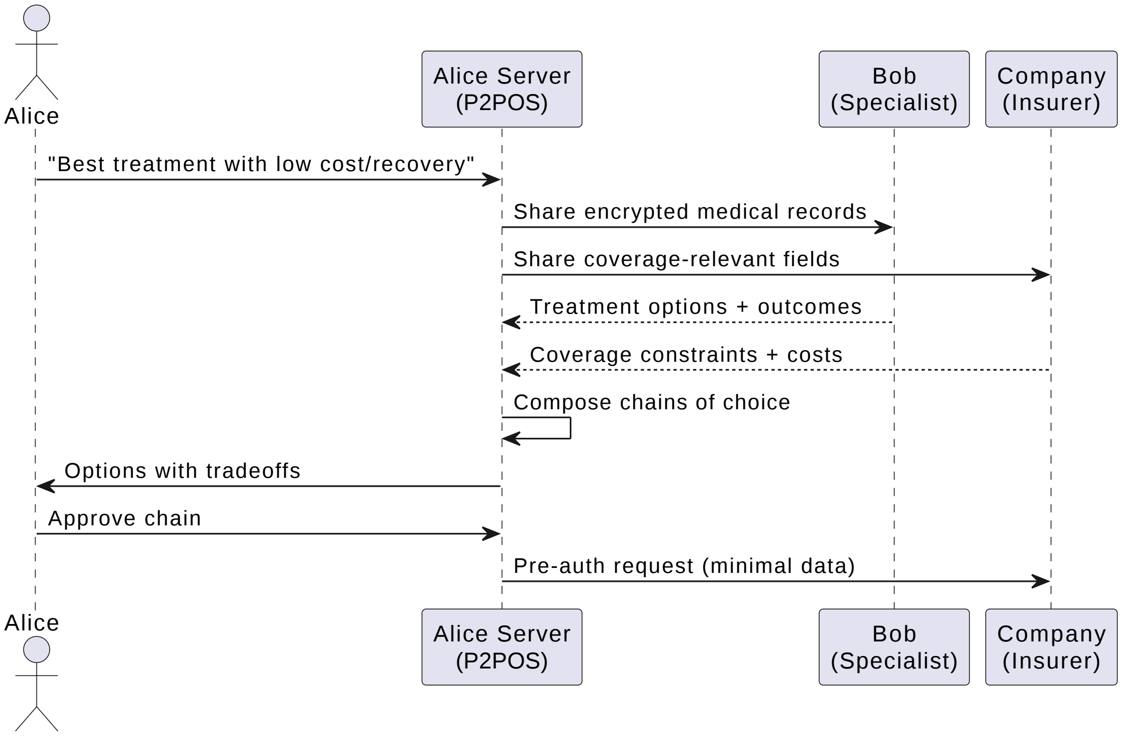
Alice wants to reduce her electricity bill. Bob runs a local co-op. The Company is the utility.



Outcome: Alice chooses a chain based on her data and preferences, not the utility's default outcome.

2) Medical Second Opinion + Insurance Approval

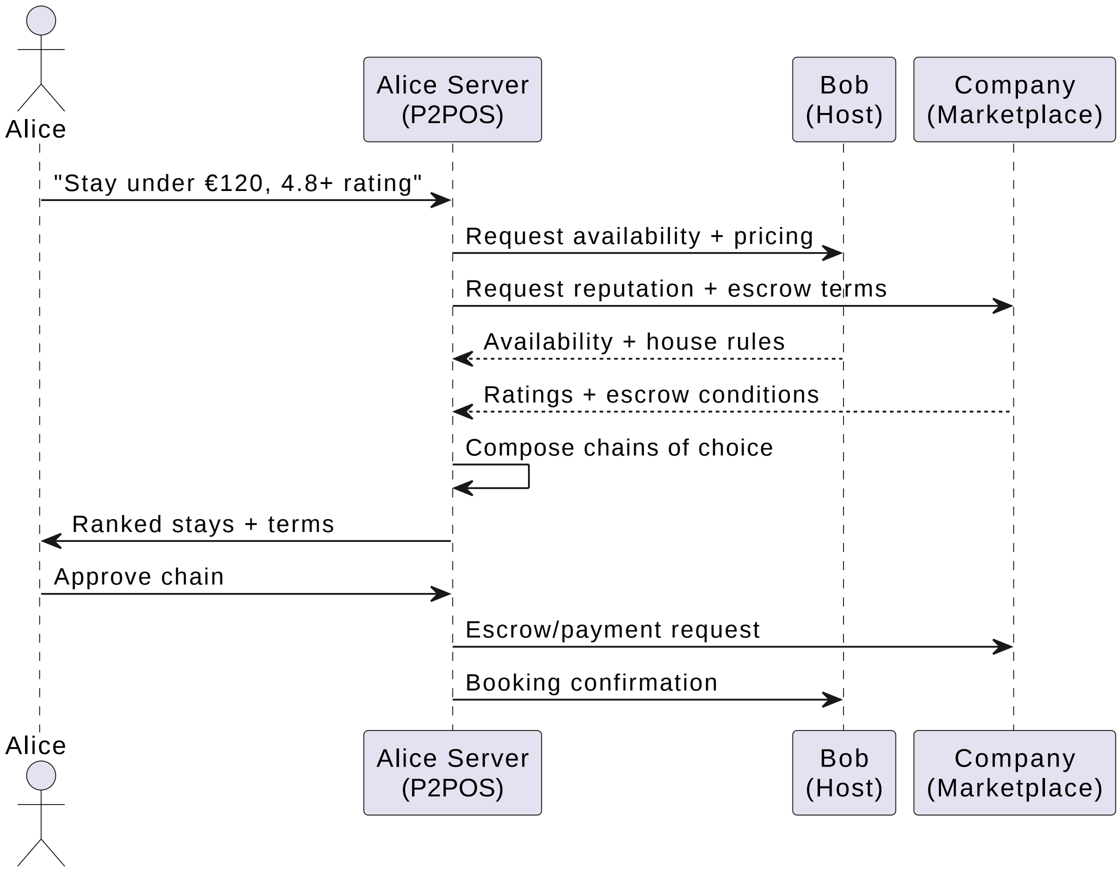
Alice seeks treatment options. Bob is a specialist. The Company is the insurer.



Outcome: Alice gets multiple peer-validated options and chooses what fits her life.

3) P2P Travel Booking (Airbnb-like)

Alice wants a place to stay. Bob is a host. The Company provides escrow and reputation.



Outcome: Bob and Company are peers; Alice's server orchestrates based on her data and intent.

Alternatives and Positioning

This section summarizes how P2POS compares to similar projects discussed in this doc set.

Projects in the same problem space

- **Solid (Pods)**: personal data pods + user-controlled app access
- **DXOS**: local-first app platform with CRDT replication
- **Holochain**: agent-centric p2p app framework with built-in validation
- **OrbitDB**: p2p database on IPFS/libp2p
- **SSB**: offline-first append-only feeds with gossip replication

Why P2POS has value

P2POS is **not a full application framework**; it is a set of **OS-like primitives**:

- Directory, messaging, storage, accounting, compute
- TopicLog + SQL projection for event sourcing
- Collaboration via OrbitDB
- SOLID-style data access patterns

This makes it easier to **embed in existing web systems** without adopting a new runtime.

Compared to Holochain

Holochain strengths

- Built-in validation model and agent-centric rules
- Strong framework conventions for p2p apps

P2POS strengths

- Interoperable with the existing **libp2p/Helia/OrbitDB/SOLID** ecosystem

- Familiar JS/TS developer tooling (npm, Vite, Playwright)
- Small composable primitives rather than a monolithic runtime

DevX + Adoption

If your priority is **rapid adoption**, P2POS has an advantage because it:

- Uses familiar web tooling and libraries
- Works with optional infrastructure (relays/bootstraps) rather than new runtimes
- Keeps the API surface small (`node.app.open`, `node.collaboration.openFile`, `node.messenger.send`)

"Is it just security?"

It is **not only security**. The main trade-offs are:

- **Validation model** (framework-level rules vs policy checks)
- **Interoperability** with the broader p2p/web ecosystem
- **Product shape** (primitive-based layer vs full framework)

P2POS is a pragmatic path to peer-to-peer adoption while keeping the web stack intact.

Pains and Goals

Pains We Aim To Solve

- **Centralization**: reduce reliance on a single server or database
- **Opaque data ownership**: make data verifiable and portable
- **Poor offline behavior**: allow local-first actions with eventual sync
- **Rigid architecture**: provide composable primitives instead of a monolith

Current Gaps / Known Trade-offs

- **Offline collaboration conflict resolution** is last-write-wins (full text overwrite).
- **Persistence in the browser** is in-memory by default (IndexedDB not wired).
- **Structured SQL exec sync** is currently manual via app events, not automatic.
- **Reliability depends on relay availability** in browser mode.

Near-term Goals

- Add **persistent browser storage** (IndexedDB for Helia and OrbitDB)
- Improve **conflict resolution** in collaboration (CRDT or operation log)
- Expand **observability** (logs, tracing, sync health)
- Hardening of **bootstrap/discovery** in heterogeneous networks

Roadmap

This roadmap outlines the next major development priorities for P2POS. Each item includes rationale, complexity assessment, architectural approach, and an implementation prompt.

Priority Items

Priority	Item	Complexity	Recommended OSS	Status
1	Persistent Browser Storage	Medium	`idb`	Planned
2	Observability	Medium	`pino` + OpenTelemetry	Planned
3	Access Control / Permissions	High	UCAN	Planned
4	Encryption Key Management	High	`@noble/curves`	Planned
5	E2E Test Coverage	Low-Medium	Playwright (existing)	Planned
6	API Documentation	Low	TypeDoc	Planned

Open Source Recommendations Summary

Before diving into each feature, here's a summary of recommended open source projects that can significantly reduce implementation complexity:

Feature	Library	Weekly Downloads	Dependents	Last Audit	Robu
Storage	`idb`	11M+	1,200+	N/A (simple wrapper)	**Excel**
Logging	`pino`	18M+	8,000+	N/A	**Excel**
Tracing	`@opentelemetry/api`	29M+	2,500+	CNCF Project	**Excel**
Access Control	`@ipld/dag-ucan`	~10K	50+	Fission-backed	**Good**
Crypto	`@noble/curves`	9.8M+	1,200+	Cure53, Trail of Bits, Kudelski	**Excel**
Crypto	`@noble/hashes`	15M+	2,000+	Same audits	**Excel**
Docs	`typedoc`	2.6M+	1,000+	N/A	**Excel**

Robustness criteria:

- **Excellent**: >1M downloads/week, >1000 dependents, active maintenance, audited (if security-critical)
- **Good**: >10K downloads/week, >50 dependents, active maintenance
- **Fair**: Active but newer project, smaller community

Implementation Order

Recommended sequence based on dependencies:

Suggested phases:

1. **Phase 1**: Persistent Storage + E2E Tests (foundation)
2. **Phase 2**: Observability (debugging support for subsequent work)
3. **Phase 3**: Key Management + Access Control (security, can be parallel)
4. **Phase 4**: API Documentation (ongoing, finalize after features)

Complexity Reduction Summary

Using the recommended open source libraries significantly reduces implementation complexity:

Feature	Original Complexity	With OSS	Reduction	Notes
Persistent Storage	Medium	**Low-Medium**	30%	`idb` handles IndexedDB quirks
Observability	Medium	**Low**	50%	`pino` + OTel are drop-in
Access Control	High	**Medium**	50%	UCAN handles signing/delegation
Key Management	High	**Medium-High**	40%	@noble handles crypto
E2E Tests	Low-Medium	**Low-Medium**	0%	Already using Playwright
API Docs	Low	**Low**	0%	TypeDoc is straightforward

Overall project impact:

- Original estimated effort: 14-18 weeks
- With OSS adoption: **10-13 weeks**
- Time saved: ~4-5 weeks

Risk reduction:

- Crypto bugs: **Eliminated** (audited @noble libraries)
- IndexedDB edge cases: **Minimized** (battle-tested `idb`)
- Logging performance: **Optimized** (pino is fastest)
- Standards compliance: **Guaranteed** (OpenTelemetry, UCAN specs)

Libraries to Install

```
# Storage
npm install idb

# Observability
npm install pino pino-pretty
npm install @opentelemetry/api  # optional, for distributed tracing

# Access Control (when implementing)
npm install @ipld/dag-ucan

# Crypto (likely already have @noble via libp2p)
npm install @noble/curves @noble/hashes

# Documentation (dev dependency)
npm install -D typedoc typedoc-plugin-markdown
```

Note: Many of these may already be transitive dependencies via libp2p/Helia. Check your lock file before adding.

Persistent Browser Storage

Recommended Open Source: `idb`

Metric	Value
Package	`idb` (https://www.npmjs.com/package/idb)
Weekly Downloads	11,139,848
Dependents	1,203
Bundle Size	~1.19kB (brotli)
Author	Jake Archibald (Google Chrome team)
Last Published	8 months ago
License	ISC

Why this library:

- Tiny wrapper over native IndexedDB with Promise-based API
- Full TypeScript support with generics for type-safe stores
- Maintained by a Google Chrome developer (Jake Archibald)
- Used by major projects (11M+ weekly downloads)
- Zero dependencies
- Handles IndexedDB quirks (transaction auto-close, upgrade handling)

Robustness assessment: Excellent

- The library is essentially a thin Promise wrapper, so there's minimal surface area for bugs
- Extremely well-tested through massive adoption
- No security-critical code (just async wrappers)
- You won't be debugging this library

Alternative considered:

- `localforage`: More abstraction but larger bundle, less TypeScript support
- Raw IndexedDB: Works but verbose and error-prone

Why It Is Needed

Currently, browser storage is in-memory by default. This means:

- **Data loss on refresh**: Users lose all local state when closing or refreshing the browser tab.
- **No true local-first**: Without persistence, "local-first" is only valid for a single session.
- **Repeated sync overhead**: Every session requires full re-sync from peers.
- **Poor offline UX**: Users cannot return to their data when offline.

Persistent storage is foundational for the P2POS vision of user-owned data that survives across sessions and works offline.

Complexity: Medium

Factors:

- Helia and OrbitDB both support pluggable storage backends
- IndexedDB is well-supported but has async/quota quirks
- Need to handle storage migrations and corruption gracefully
- Must coordinate persistence across multiple subsystems (Helia blocks, OrbitDB logs, TopicLog events, SQL projections)

Estimated scope: 2-3 weeks of focused work

Potential Architecture

Key components using `idb`:

1. **StorageCoordinator**: Orchestrates initialization, uses `idb.openDB()` for each store with version tracking.

2. **HeliaBlockstore**: Implements Blockstore interface using `idb` - stores CID → Uint8Array.

3. **OrbitDBStorage**: Uses `idb` for operation logs per database address.

4. **TopicLogStore**: Uses `idb` with compound index on (topic, sequence) for range queries.

5. **SQLiteVFS**: Uses OPFS directly (better perf), no `idb` needed here.

Storage schema using `idb` DBSchema:

```
import { DBSchema } from 'idb';

interface P2POSSchema extends DBSchema {
  'helia-blocks': {
    key: string;        // CID string
    value: Uint8Array;  // Block data
  };
  'orbitdb-entries': {
    key: string;        // hash
    value: OrbitDBEntry;
    indexes: { 'by-db': string };  // dbAddress
  };
  'topiclog-events': {
    key: [string, number];  // [topic, sequence]
    value: SignedEvent;
    indexes: { 'by-topic': string };
  };
  'meta': {
    key: string;
    value: unknown;
  };
}
```

Implementation Prompt

```
Implement persistent browser storage for P2POS using the `idb` library.

Context:
- P2POS uses Helia for content-addressed storage, OrbitDB for
collaboration,
  TopicLog for event sourcing, and sql.js for SQL projections.
- Currently all storage is in-memory and lost on page refresh.
- The codebase uses TypeScript with Vite for browser builds.

Dependencies to install:
  npm install idb
  npm install -D fake-indexeddb  # for testing

Key `idb` APIs to use:
- openDB<Schema>(name, version, { upgrade }) - open/create database
- db.get(storeName, key) - get single value
- db.getAll(storeName) - get all values
- db.put(storeName, value, key?) - insert/update
- db.delete(storeName, key) - delete
- db.transaction(storeNames, mode) - for batch operations
- tx.store.index(indexName).getAll(query) - index queries

Requirements:
1. Define TypeScript schema in src/storage/schema.ts:
   - Use idb's DBSchema interface for type safety
   - Define stores: helia-blocks, orbitdb-entries, topiclog-events, meta
   - Define indexes for efficient queries

2. Create StorageCoordinator in src/storage/coordinator.ts:
   import { openDB, IDBPDatabase } from 'idb';
   import { P2POSSchema } from './schema';

   - Use openDB<P2POSSchema>('p2pos', version, { upgrade })
   - Handle schema migrations in upgrade callback
   - Provide init(): Promise<void> and close(): void
   - Use navigator.storage.estimate() for quota monitoring

3. Implement HeliaBlockstore in src/storage/helia-idb.ts:
   - Implement Blockstore interface using the shared IDBPDatabase
   - put(cid, block): await db.put('helia-blocks', block,
cid.toString())
   - get(cid): await db.get('helia-blocks', cid.toString())
   - has(cid): (await db.get(...)) !== undefined
   - delete(cid): await db.delete('helia-blocks', cid.toString())

4. Implement OrbitDB storage in src/storage/orbitdb-idb.ts:
   - Store operation logs with dbAddress as index
   - Use transactions for batch writes during sync

5. Implement TopicLogStore in src/storage/topiclog-idb.ts:
   - Use compound key [topic, sequence] for ordering
   - Range query: db.getAll('topiclog-events', IDBKeyRange.bound([topic,
fromSeq], [topic, Infinity]))

6. Configure sql.js with OPFS persistence:
   - Check: 'createSyncAccessHandle' in FileSystemFileHandle.prototype
```

```
   - Use sql.js OPFS VFS when available
   - Fallback: serialize DB to idb on changes

7. Update createP2POS() options:
   interface P2POSOptions {
     persistent?: boolean;  // default: false
     storageName?: string;  // default: 'p2pos'
   }

8. Write tests using fake-indexeddb:
   import 'fake-indexeddb/auto';
   // Now idb uses fake implementation

   - Test CRUD operations
   - Test migration from v1 to v2
   - Test quota warning emission

Example usage:
  import { openDB } from 'idb';
  import { P2POSSchema } from './schema';

  const db = await openDB<P2POSSchema>('p2pos', 1, {
    upgrade(db, oldVersion, newVersion, tx) {
      if (oldVersion < 1) {
        db.createObjectStore('helia-blocks');
        const eventStore = db.createObjectStore('topiclog-events');
        eventStore.createIndex('by-topic', 0);  // index on first
element of key
      }
    }
  });

Existing code references:
- src/p2pos.ts: createP2POS factory
- src/os/blobstore.ts: current in-memory BlobStore
- src/os/topic-log.ts: TopicLog implementation
- src/os/orbitdb.ts: OrbitDB setup
```

Observability

Recommended Open Source: `pino` + `@opentelemetry/api`

Logging: `pino`

Metric	Value
Package	`pino` (https://www.npmjs.com/package/pino)
Weekly Downloads	18,655,233
Dependents	8,049
Author	Matteo Collina (Node.js TSC member)
Last Published	4 days ago
License	MIT

Why pino:

- Fastest JSON logger for Node.js (5x faster than alternatives)
- Structured JSON output by default
- Works in both Node.js and browser
- Child loggers with context inheritance
- Transport system for async log processing
- `pino-pretty` for development formatting
- Maintained by Node.js Technical Steering Committee member

Tracing: `@opentelemetry/api`

Metric	Value
Package	`@opentelemetry/api` (https://www.npmjs.com/package/@opentelemetry/api)
Weekly Downloads	29,203,085
Dependents	2,559
Backing	CNCF (Cloud Native Computing Foundation)
Last Published	Stable (1.9.0)
License	Apache-2.0

Why OpenTelemetry:

- Industry standard for distributed tracing
- Zero dependencies in API package
- No-op by default (safe to include without SDK)
- Works in Node.js and browser
- Vendor-neutral: export to Jaeger, Zipkin, or any backend
- Massive ecosystem and tooling support

Robustness assessment: Excellent

- Both libraries are among the most downloaded on npm
- pino is maintained by Node.js core contributors
- OpenTelemetry is a CNCF graduated project
- You won't be debugging these libraries

Recommended approach:

- Use `pino` for structured logging (simple, fast)
- Use `@opentelemetry/api` for distributed tracing (standard, optional)
- Create thin wrappers to unify the interface for P2POS

Why It Is Needed

Distributed systems are inherently hard to debug. Without observability:

- **Silent failures**: Sync issues, dropped messages, or slow peers go unnoticed.
- **Hard to reproduce bugs**: Timing-dependent issues are nearly impossible to trace.
- **No health visibility**: Users and developers can't tell if the system is working correctly.
- **Poor operational readiness**: Can't deploy with confidence without metrics.

Observability is essential for moving from "demo" to "production-ready."

Complexity: Medium

Factors:

- Need structured logging without bloating bundle size
- Tracing distributed operations across peers requires correlation IDs
- Sync health metrics need careful definition
- Must work in both Node and browser environments

Estimated scope: 2 weeks

Potential Architecture

Key components using `pino` and `@opentelemetry/api`:

1. **P2POSLogger**: Thin wrapper around pino with P2POS-specific context (peerId, component).

2. **P2POSTracer**: Wrapper around OpenTelemetry API - no-op if SDK not installed.

3. **Metrics**: Simple counters/gauges using OpenTelemetry Metrics API (optional).

4. **SyncHealth**: Aggregates metrics into health status.

pino output format (automatic):

```json
{
  "level": 30,
  "time": 1706784600000,
  "pid": 1234,
  "hostname": "browser",
  "component": "messenger",
  "peerId": "12D3Koo...",
  "msg": "envelope sent",
  "to": "12D3Koo...",
  "size": 1024
}
```

Implementation Prompt

```
Implement an observability layer for P2POS using pino for logging and
@opentelemetry/api for distributed tracing.

Context:
- P2POS is a peer-to-peer system with Messenger, TopicLog,
Collaboration, and Overlay components.
- Debugging distributed sync issues is currently difficult.
- The system runs in both Node.js and browser environments.

Dependencies to install:
  npm install pino pino-pretty
  npm install @opentelemetry/api
  # Optional SDK (users can install if they want tracing):
  # npm install @opentelemetry/sdk-trace-base @opentelemetry/sdk-trace-
web

Key pino APIs:
- pino(options) - create root logger
- logger.child({ component }) - create scoped child logger
- logger.info/warn/error/debug(obj, msg) - structured logging
- pino.destination() - custom output destination

Key OpenTelemetry APIs:
- trace.getTracer(name, version) - get tracer instance
- tracer.startSpan(name, options, context) - create span
- span.setAttribute(key, value) - add attributes
- span.end() - finish span
- context.with(ctx, fn) - run fn with context
- propagation.inject/extract() - for message headers

Requirements:
1. Create P2POSLogger in src/observability/logger.ts:
   import pino from 'pino';

   const rootLogger = pino({
     level: process.env.LOG_LEVEL || 'info',
     browser: { asObject: true },  // browser compatibility
     transport: process.env.NODE_ENV === 'development'
       ? { target: 'pino-pretty' }
       : undefined
   });

   export function createLogger(component: string, peerId?: string) {
     return rootLogger.child({ component, peerId });
   }

   // Usage: const log = createLogger('messenger', node.peerId);
   //        log.info({ to: peer, size: 1024 }, 'envelope sent');

2. Create P2POSTracer in src/observability/tracer.ts:
   import { trace, context, SpanStatusCode } from '@opentelemetry/api';

   const tracer = trace.getTracer('p2pos', '1.0.0');

   export function startSpan<T>(name: string, fn: (span: Span) =>
Promise<T>): Promise<T> {
```

```
      return tracer.startActiveSpan(name, async (span) => {
        try {
          const result = await fn(span);
          span.setStatus({ code: SpanStatusCode.OK });
          return result;
        } catch (err) {
          span.setStatus({ code: SpanStatusCode.ERROR, message:
err.message });
          throw err;
        } finally {
          span.end();
        }
      });
    }

    // For message propagation
    export function injectTraceContext(carrier: Record<string, string>) {
      propagation.inject(context.active(), carrier);
    }

    export function extractTraceContext(carrier: Record<string, string>)
{
      return propagation.extract(context.active(), carrier);
    }

3. Create Metrics in src/observability/metrics.ts:
    // Simple implementation (OTel Metrics API is optional)
    class SimpleMetrics {
      private counters = new Map<string, number>();
      private gauges = new Map<string, number>();

      inc(name: string, value = 1) {
        this.counters.set(name, (this.counters.get(name) || 0) + value);
      }
      set(name: string, value: number) {
        this.gauges.set(name, value);
      }
      snapshot() {
        return { counters: Object.fromEntries(this.counters), gauges:
Object.fromEntries(this.gauges) };
      }
    }

    export const metrics = new SimpleMetrics();
    // Usage: metrics.inc('p2pos_messages_sent_total');
    //        metrics.set('p2pos_peers_connected', 5);

4. Create SyncHealth in src/observability/sync-health.ts:
    export type HealthStatus = 'healthy' | 'degraded' | 'unhealthy';

    export class SyncHealth {
      private peerStates = new Map<string, { lastSync: number; lag:
number }>();

      updatePeer(peerId: string, lastSync: number, lag: number) { ... }
      removePeer(peerId: string) { ... }
```

```
    getStatus(): HealthStatus {
      const now = Date.now();
      const stale = [...this.peerStates.values()].filter(p => now -
p.lastSync > 30000);
      if (stale.length === this.peerStates.size) return 'unhealthy';
      if (stale.length > 0) return 'degraded';
      return 'healthy';
    }
  }

5. Instrument Messenger (example):
   // In src/os/messenger.ts
   import { createLogger } from '../observability/logger';
   import { startSpan, injectTraceContext } from
'../observability/tracer';
   import { metrics } from '../observability/metrics';

   const log = createLogger('messenger');

   async send(to: PeerId, envelope: Envelope) {
     return startSpan('messenger.send', async (span) => {
       span.setAttribute('peer.to', to.toString());
       span.setAttribute('envelope.size', envelope.data.length);

       // Inject trace context into envelope metadata
       const carrier: Record<string, string> = {};
       injectTraceContext(carrier);
       envelope.metadata = { ...envelope.metadata, ...carrier };

       log.info({ to: to.toString(), size: envelope.data.length },
'sending envelope');
       metrics.inc('p2pos_messages_sent_total');

       await this.overlay.send(to, envelope);
     });
   }

6. Browser debug hook:
   if (typeof window !== 'undefined') {
     (window as any).__P2POS_DEBUG__ = {
       getMetrics: () => metrics.snapshot(),
       getLogs: () => logBuffer.slice(-100),
       getHealth: () => syncHealth.getStatus(),
     };
   }

7. Configuration:
   interface ObservabilityOptions {
     level?: 'debug' | 'info' | 'warn' | 'error';
     tracing?: boolean;   // default: true (no-op if no SDK)
     metrics?: boolean;   // default: true
   }

   createP2POS({ observability: { level: 'debug' } });

8. Testing:
   import pino from 'pino';
```

```
  test('logger outputs structured JSON', () => {
    const logs: any[] = [];
    const log = pino({ level: 'info' }, pino.destination({ write: (s)
=> logs.push(JSON.parse(s)) }));
    log.info({ foo: 'bar' }, 'test message');
    expect(logs[0]).toMatchObject({ level: 30, msg: 'test message',
foo: 'bar' });
  });

Existing code references:
- src/os/messenger.ts: message send/receive
- src/os/topic-log.ts: event append/catchUp
- src/os/collaboration.ts: file operations
- src/overlay/adapter.ts: peer connections
```

Access Control / Permissions

Recommended Open Source: UCAN (User Controlled Authorization Networks)

Metric	Value
Specification	ucan.xyz (https://ucan.xyz/)
JS Package	`@ipld/dag-ucan` or `ucans`
Weekly Downloads	~10K (growing)
Backing	Fission, IPFS ecosystem
Spec Version	0.10.0
License	Apache-2.0 / MIT

Why UCAN:

- **Designed for P2P**: UCAN was specifically created for decentralized authorization
- **Cryptographically signed**: All capabilities are signed and verifiable offline
- **Delegation built-in**: Native support for capability delegation chains
- **Time-bounded**: Built-in expiration support
- **IPFS ecosystem alignment**: Used by Fission, integrates with IPLD/libp2p
- **No central authority**: Perfect fit for P2POS philosophy

Key UCAN concepts that map to P2POS needs:

- **Issuer/Audience**: Maps to grantor/grantee in P2POS
- **Capabilities**: Maps to resource + action patterns
- **Proofs**: Enables delegation chains
- **Expiration**: Built-in time-bounding

Robustness assessment: Good

- Specification is well-documented and stable

- Used in production by Fission (web3.storage)
- Smaller ecosystem than enterprise solutions, but growing
- Active development and community
- May need to contribute fixes if edge cases arise

Complexity reduction:

- UCAN handles the hard parts: signing, verification, delegation chains, expiration
- P2POS only needs to: define resource patterns, integrate with existing components
- Estimated complexity reduction: **50-60%** (from High to Medium)

Alternatives considered:

- **Custom capability system**: More work, reinventing the wheel
- **OAuth/OIDC**: Requires central server, not P2P-compatible
- **Macaroons**: Similar concept but less ecosystem support for JS/P2P

Caveat:

- UCAN ecosystem is newer than enterprise auth solutions
- You may encounter edge cases not covered by existing libraries
- Plan for some custom code around the edges

Why It Is Needed

The documented use cases (medical records, energy data, travel booking) all involve:

- **Selective data sharing**: Alice shares specific fields with Bob but not everything.
- **Peer authorization**: Only authorized peers should read/write certain resources.
- **Revocable access**: Permissions must be changeable over time.
- **Auditability**: Know who accessed what and when.

Without access control, P2POS cannot support real-world privacy-sensitive applications.

Complexity: High

Factors:

- Decentralized access control is fundamentally harder than centralized
- Need to define a capability/policy model that works offline
- Must integrate with encryption (item 4) for enforcement
- Revocation in a distributed system is an open research problem
- SOLID compatibility adds constraints

Estimated scope: 4-6 weeks

Potential Architecture

UCAN capability mapping for P2POS:

```
// P2POS resource types mapped to UCAN capabilities
// UCAN uses URI-style capability identifiers

// Topic access
{ with: 'p2pos://topic/app:todo:*', can: 'topic/read' }
{ with: 'p2pos://topic/app:todo:*', can: 'topic/write' }

// File access
{ with: 'p2pos://file/medical/**', can: 'file/read' }
{ with: 'p2pos://file/medical/**', can: 'file/write' }

// Pod access
{ with: 'p2pos://pod/profile/public/*', can: 'pod/read' }
```

UCAN token structure (handled by library):

```
// The library handles all of this - you just call issue()
interface UCANToken {
  header: { alg: 'EdDSA', typ: 'JWT', ucv: '0.10.0' },
  payload: {
    iss: string,      // Issuer DID (your peerId as did:key)
    aud: string,      // Audience DID (recipient peerId)
    exp: number,      // Expiration timestamp
    att: Capability[],// Capabilities granted
    prf: CID[],       // Proof chain (previous UCANs)
    fct: Fact[],      // Optional facts/context
  },
  signature: Uint8Array
}
```

Revocation approach (using UCAN patterns):

1. **Short expiration** (default 1 hour) - natural revocation
2. **Revocation topic** - publish CIDs of revoked UCANs to `p2pos://revocations`
3. **On verify** - check revocation topic before accepting

Implementation Prompt

```
Implement capability-based access control for P2POS using UCAN (User
Controlled Authorization Networks).

Context:
- P2POS needs to support selective data sharing between peers.
- Use cases include sharing medical records, energy data, and financial
info.
- The system must work in a decentralized, offline-capable environment.
- UCAN provides the cryptographic foundation; P2POS defines capability
semantics.

Dependencies to install:
  npm install @ipld/dag-ucan
  npm install @ucans/core  # alternative implementation
  # Note: @ipld/dag-ucan is more actively maintained

Key UCAN APIs (@ipld/dag-ucan):
- UCAN.issue({ issuer, audience, capabilities, expiration, proofs })
- UCAN.verify(token, { audience })
- UCAN.parse(tokenString)
- UCAN.encode(token) / UCAN.decode(bytes)

Requirements:
1. Define P2POS capability semantics in src/access/capabilities.ts:
   // Define resource URI patterns
   const RESOURCE_PATTERNS = {
     topic: (pattern: string) => `p2pos://topic/${pattern}`,
     file: (path: string) => `p2pos://file${path}`,
     pod: (path: string) => `p2pos://pod${path}`,
   };

   // Define actions per resource type
   type TopicAction = 'topic/read' | 'topic/write' | 'topic/admin';
   type FileAction = 'file/read' | 'file/write' | 'file/delete';
   type PodAction = 'pod/read' | 'pod/write';

   // Helper to create capability
   export function capability(resource: string, action: string) {
     return { with: resource, can: action };
   }

2. Create UCANStore in src/access/ucan-store.ts:
   import * as UCAN from '@ipld/dag-ucan';
   import { openDB } from 'idb';

   class UCANStore {
     private db: IDBPDatabase<UCANSchema>;

     // Store UCAN tokens we've received (grants to us)
     async storeReceived(token: UCAN.UCAN): Promise<void> {
       const encoded = UCAN.encode(token);
       const cid = await computeCID(encoded);
       await this.db.put('received-ucans', { cid, token: encoded, ...
});
     }
```

```
      // Store UCAN tokens we've issued (grants from us)
      async storeIssued(token: UCAN.UCAN): Promise<void> { ... }

      // Find UCANs that grant a specific capability
      async findByCapability(resource: string, action: string):
Promise<UCAN.UCAN[]> {
        // Use index to find matching UCANs
        // Check glob patterns (e.g., 'p2pos://topic/app:*' matches
'p2pos://topic/app:todo')
      }
    }

3. Create UCANVerifier in src/access/ucan-verifier.ts:
   import * as UCAN from '@ipld/dag-ucan';

   class UCANVerifier {
     constructor(
       private store: UCANStore,
       private revocationTopic: string = 'p2pos://revocations'
     ) {}

     async verify(
       token: UCAN.UCAN | string,
       resource: string,
       action: string
     ): Promise<{ valid: boolean; reason?: string }> {
       const ucan = typeof token === 'string' ? UCAN.parse(token) :
token;

       // 1. Verify signature
       const signatureValid = await UCAN.verify(ucan);
       if (!signatureValid) return { valid: false, reason:
'invalid_signature' };

       // 2. Check expiration
       if (ucan.payload.exp < Date.now() / 1000) {
         return { valid: false, reason: 'expired' };
       }

       // 3. Check audience matches our DID
       if (ucan.payload.aud !== this.myDID) {
         return { valid: false, reason: 'wrong_audience' };
       }

       // 4. Check capability grants the requested action
       const hasCapability = ucan.payload.att.some(cap =>
         matchesResource(cap.with, resource) && cap.can === action
       );
       if (!hasCapability) return { valid: false, reason:
'insufficient_capability' };

       // 5. Check revocation list
       const isRevoked = await this.checkRevocation(ucan);
       if (isRevoked) return { valid: false, reason: 'revoked' };

       // 6. Verify proof chain (delegation)
       for (const proofCid of ucan.payload.prf) {
```

```
          const proofUcan = await this.store.getByCid(proofCid);
          const proofValid = await this.verify(proofUcan, resource,
action);
          if (!proofValid.valid) return { valid: false, reason:
'invalid_proof_chain' };
        }

        return { valid: true };
      }
    }

4. Create capability management API in src/access/manager.ts:
   import * as UCAN from '@ipld/dag-ucan';
   import { ed25519 } from '@noble/curves/ed25519';

   class AccessManager {
     constructor(
       private issuerKey: { secretKey: Uint8Array; publicKey: Uint8Array
},
       private store: UCANStore,
       private verifier: UCANVerifier
     ) {}

     // Issue a new UCAN granting capabilities to another peer
     async grant(
       audience: string,  // recipient's DID (did:key:...)
       capabilities: Array<{ resource: string; action: string }>,
       options?: { expiration?: number; proofs?: CID[] }
     ): Promise<UCAN.UCAN> {
       const token = await UCAN.issue({
         issuer: this.issuerKey,
         audience,
         capabilities: capabilities.map(c => ({ with: c.resource, can:
c.action })),
         expiration: options?.expiration ?? Math.floor(Date.now() /
1000) + 3600,
         proofs: options?.proofs ?? [],
       });

       await this.store.storeIssued(token);
       return token;
     }

     // Delegate a capability we have to another peer
     async delegate(
       originalUcan: UCAN.UCAN,
       newAudience: string,
       subset?: Array<{ resource: string; action: string }>
     ): Promise<UCAN.UCAN> {
       const capabilities = subset ?? originalUcan.payload.att;
       const proofCid = await computeCID(UCAN.encode(originalUcan));

       return this.grant(newAudience, capabilities, { proofs: [proofCid]
});
     }

     // Check if a peer has capability for an action
```

```
    async check(
      token: UCAN.UCAN | string,
      resource: string,
      action: string
    ): Promise<boolean> {
      const result = await this.verifier.verify(token, resource,
action);
      return result.valid;
    }
  }

5. Create AccessLog in src/access/audit-log.ts:
   class AccessLog {
     async log(entry: {
       timestamp: number;
       peer: string;
       resource: string;
       action: string;
       granted: boolean;
       reason?: string;
       ucanCid?: string;
     }): Promise<void> {
       await this.db.add('access-log', entry);
     }

     async query(filter: { peer?: string; resource?: string; since?:
number }) { ... }
   }

6. Integrate with TopicLog (example):
   // In src/os/topic-log.ts
   async append(event: Event, ucan: UCAN.UCAN): Promise<void> {
     const resource = `p2pos://topic/${this.topicId}`;
     const canWrite = await this.accessManager.check(ucan, resource,
'topic/write');

     if (!canWrite) {
       await this.accessLog.log({ ..., granted: false, reason:
'insufficient_capability' });
       throw new AccessDeniedError('Cannot write to topic');
     }

     await this.accessLog.log({ ..., granted: true });
     // ... proceed with append
   }

7. Integrate with Messenger (attach UCAN to envelopes):
   interface Envelope {
     data: Uint8Array;
     metadata: {
       ucan?: string;  // Encoded UCAN token for authorization
       // ... other fields
     };
   }

   // When sending, attach relevant UCAN
   async send(to: PeerId, data: Uint8Array, capability: { resource:
```

```
string; action: string }) {
     const ucan = await this.store.findByCapability(capability.resource,
capability.action);
     const envelope = { data, metadata: { ucan: UCAN.encode(ucan[0]) }
};
     await this.overlay.send(to, envelope);
   }

8. Add to P2POS facade:
   interface P2POS {
     access: {
       grant(audience: string, capabilities: Cap[], options?: Options):
Promise<UCAN>;
       delegate(ucan: UCAN, newAudience: string): Promise<UCAN>;
       check(ucan: UCAN | string, resource: string, action: string):
Promise<boolean>;
       revoke(ucanCid: string): Promise<void>;
       audit: {
         query(filter: AuditFilter): Promise<AuditEntry[]>;
       };
     };
   }

9. Revocation via TopicLog:
   // Publish revocation to well-known topic
   async revoke(ucanCid: string): Promise<void> {
     await this.topicLog.append(
       { type: 'revocation', ucanCid, timestamp: Date.now() },
       'p2pos://revocations'
     );
   }

10. Testing:
    test('grant and verify UCAN', async () => {
      const alice = await createTestPeer();
      const bob = await createTestPeer();

      // Alice grants Bob read access to her todos
      const ucan = await alice.access.grant(bob.did, [
        { resource: 'p2pos://topic/app:todo:*', action: 'topic/read' }
      ]);

      // Bob can verify and use the capability
      const valid = await bob.access.check(ucan,
'p2pos://topic/app:todo:123', 'topic/read');
      expect(valid).toBe(true);

      // Bob cannot write (not granted)
      const canWrite = await bob.access.check(ucan,
'p2pos://topic/app:todo:123', 'topic/write');
      expect(canWrite).toBe(false);
    });

Existing code references:
- src/os/topic-log.ts: event access points
- src/os/collaboration.ts: file access points
- src/solid/pod.ts: pod resource access
```

```
- src/os/messenger.ts: envelope handling

UCAN Resources:
- Spec: https://ucan.xyz/
- @ipld/dag-ucan: https://github.com/ipld/js-dag-ucan
- Tutorial: https://ucan.xyz/getting-started/
```

Encryption Key Management

Recommended Open Source: `@noble/curves` + `@noble/hashes`

Cryptographic Primitives: `@noble/curves`

Metric	Value
Package	`@noble/curves` (https://www.npmjs.com/package/@noble/curves)
Weekly Downloads	9,854,973
Dependents	1,197
Author	Paul Miller
Last Published	4 months ago
License	MIT
Security Audits	**3 independent audits**

Security Audits (Critical for crypto libraries):

1. **Cure53 (Sep 2024)** - v1.6.0
    - Scope: ed25519, ed448, bls12-381, bn254, hash-to-curve, low-level primitives
    - Funded by OpenSats
    - Audit PDF available (https://cure53.de/audit-report_noble-crypto-libs.pdf)
2. **Kudelski Security (Sep 2023)** - v1.2.0
    - Scope: StarkNet integration, curve, modular, poseidon, weierstrass
    - Funded by Starkware
3. **Trail of Bits (Feb 2023)** - v0.7.3
    - Scope: Abstract modules, secp256k1
    - Funded by Ryan Shea

Why @noble/curves:

- **Audited 3 times** by independent security firms
- Zero dependencies (only @noble/hashes)
- Pure JS implementation (no native bindings to debug)
- Supports all needed algorithms: X25519 (ECDH), Ed25519 (signing), AES-GCM
- Tree-shakeable: only include what you use
- Works in Node.js and browser
- Used by major crypto projects (Ethereum, Bitcoin libraries)

Hashing: `@noble/hashes`

Metric	Value
Package	`@noble/hashes` (https://www.npmjs.com/package/@noble/hashes)
Weekly Downloads	15,000,000+
Dependents	2,000+
Audited	Same audits as @noble/curves

Provides:

- SHA-256, SHA-512 (for signatures)
- HKDF (for key derivation)
- HMAC (for MACs)
- All needed for P2POS key management

Robustness assessment: Excellent

- This is the gold standard for JS cryptographic libraries
- Multiple independent security audits
- Massive adoption (used by ethers.js, viem, etc.)
- You absolutely won't be debugging cryptographic primitives
- Pure JS means no native compilation issues

What you still need to build:

- Key storage layer (use `idb` from item 1)
- Group key management protocol
- Key exchange orchestration
- Integration with access control (UCAN)

Complexity reduction:

- Crypto primitives are 100% handled by audited library
- Remaining work: protocol design and integration
- Estimated complexity reduction: **40%** (from High to Medium-High)

Why It Is Needed

TopicLog already supports AES-GCM encryption, but the docs don't cover:

- **Key exchange**: How do peers securely share encryption keys?
- **Key rotation**: How do you change keys without losing access to old data?
- **Group encryption**: How do multiple authorized peers decrypt the same content?
- **Key recovery**: What happens if a peer loses their keys?

Without proper key management, encryption is either unusable or insecure.

Complexity: High

Factors:

- Cryptographic protocols require careful implementation
- Key exchange in P2P without a server is non-trivial
- Group key management (add/remove members) is complex
- Must integrate tightly with access control (item 3)
- Forward secrecy and post-compromise security are desirable

Estimated scope: 4-6 weeks

Potential Architecture

Key hierarchy using @noble:

```
import { x25519 } from '@noble/curves/ed25519';
import { ed25519 } from '@noble/curves/ed25519';
import { hkdf } from '@noble/hashes/hkdf';
import { sha256 } from '@noble/hashes/sha256';
import { randomBytes } from '@noble/hashes/utils';

// Identity Key (Ed25519) - for signing and peer identity
const identityPrivate = randomBytes(32);
const identityPublic = ed25519.getPublicKey(identityPrivate);

// Exchange Key (X25519) - for ECDH key agreement
const exchangePrivate = randomBytes(32);
const exchangePublic = x25519.getPublicKey(exchangePrivate);

// Derived keys via HKDF
const sharedSecret = x25519.getSharedSecret(myPrivate, peerPublic);
const sessionKey = hkdf(sha256, sharedSecret, salt, 'p2pos-session',
32);
```

Group key management using @noble:

```
// Create group: generate random secret, encrypt to each member
const groupSecret = randomBytes(32);
for (const member of members) {
  const sharedWithMember = x25519.getSharedSecret(myExchangePrivate,
member.exchangePublic);
  const wrappingKey = hkdf(sha256, sharedWithMember, 'group-wrap',
'p2pos', 32);
  const encryptedGroupSecret = aesGcmEncrypt(wrappingKey, groupSecret);
  // Send encryptedGroupSecret to member
}

// Derive topic key from group secret
const topicKey = hkdf(sha256, groupSecret, topicId, `p2pos-
topic-${epoch}`, 32);
```

Implementation Prompt

```
Implement encryption key management for P2POS using @noble/curves and
@noble/hashes.

Context:
- TopicLog supports AES-GCM encryption but key exchange is not
implemented.
- P2POS needs secure key exchange between peers and group key
management.
- The system must work offline and in a decentralized manner.
- Integration with the access control system (UCAN) is important.

Dependencies to install:
  npm install @noble/curves @noble/hashes
  # Note: these may already be transitive deps via libp2p

Key @noble/curves APIs:
- import { x25519, ed25519 } from '@noble/curves/ed25519'
- x25519.getPublicKey(privateKey) - derive X25519 public key
- x25519.getSharedSecret(myPrivate, theirPublic) - ECDH shared secret
- ed25519.getPublicKey(privateKey) - derive Ed25519 public key
- ed25519.sign(message, privateKey) - sign message
- ed25519.verify(signature, message, publicKey) - verify signature

Key @noble/hashes APIs:
- import { hkdf } from '@noble/hashes/hkdf'
- import { sha256 } from '@noble/hashes/sha256'
- import { randomBytes } from '@noble/hashes/utils'
- hkdf(hash, inputKey, salt, info, length) - derive key material
- randomBytes(length) - cryptographically secure random bytes

Requirements:
1. Create KeyStore in src/crypto/keystore.ts:
   import { x25519, ed25519 } from '@noble/curves/ed25519';
   import { randomBytes } from '@noble/hashes/utils';
   import { openDB } from 'idb';

   interface StoredKeys {
     identity: { private: Uint8Array; public: Uint8Array };
     exchange: { private: Uint8Array; public: Uint8Array };
     derivedKeys: Map<string, Uint8Array>;  // keyId -> key
   }

   class KeyStore {
     private keys: StoredKeys | null = null;

     // Generate new key pairs
     async generate(): Promise<void> {
       const identityPrivate = randomBytes(32);
       const exchangePrivate = randomBytes(32);

       this.keys = {
         identity: {
           private: identityPrivate,
           public: ed25519.getPublicKey(identityPrivate),
         },
         exchange: {
```

```
            private: exchangePrivate,
            public: x25519.getPublicKey(exchangePrivate),
          },
          derivedKeys: new Map(),
        };

        await this.persist();
      }

      // Persist to IndexedDB (encrypted with password-derived key)
      async persist(): Promise<void> {
        // Use Web Crypto to encrypt keys before storing
        const db = await openDB('p2pos-keys', 1, { ... });
        // Encrypt this.keys with storage key derived from password
        await db.put('keys', encryptedKeys, 'main');
      }

      getIdentityPublic(): Uint8Array { return
this.keys!.identity.public; }
      getExchangePublic(): Uint8Array { return
this.keys!.exchange.public; }
    }

2. Create KeyExchange in src/crypto/key-exchange.ts:
   import { x25519 } from '@noble/curves/ed25519';
   import { hkdf } from '@noble/hashes/hkdf';
   import { sha256 } from '@noble/hashes/sha256';

   class KeyExchange {
     private sessionKeys = new Map<string, Uint8Array>();

     constructor(private keyStore: KeyStore) {}

     // Derive shared session key with a peer
     async deriveSessionKey(peerExchangePublic: Uint8Array):
Promise<Uint8Array> {
       const peerId = bytesToHex(peerExchangePublic);

       // Check cache
       if (this.sessionKeys.has(peerId)) {
         return this.sessionKeys.get(peerId)!;
       }

       // X25519 ECDH
       const sharedSecret = x25519.getSharedSecret(
         this.keyStore.getExchangePrivate(),
         peerExchangePublic
       );

       // Derive session key via HKDF
       const sessionKey = hkdf(
         sha256,
         sharedSecret,
         'p2pos-session-salt',  // salt
         'p2pos-session-key',   // info
         32                      // output length
       );
```

```
        this.sessionKeys.set(peerId, sessionKey);
        return sessionKey;
      }
    }

3. Create GroupKeyManager in src/crypto/group-keys.ts:
   import { randomBytes } from '@noble/hashes/utils';
   import { hkdf } from '@noble/hashes/hkdf';
   import { sha256 } from '@noble/hashes/sha256';

   interface Group {
     id: string;
     secret: Uint8Array;
     epoch: number;
     members: Set<string>;  // member DIDs
   }

   class GroupKeyManager {
     private groups = new Map<string, Group>();

     constructor(
       private keyStore: KeyStore,
       private keyExchange: KeyExchange,
       private messenger: Messenger
     ) {}

     // Create a new group with initial members
     async createGroup(memberPublicKeys: Uint8Array[]): Promise<string>
{
       const groupId = bytesToHex(randomBytes(16));
       const groupSecret = randomBytes(32);

       const group: Group = {
         id: groupId,
         secret: groupSecret,
         epoch: 0,
         members: new Set(),
       };

       // Encrypt group secret to each member
       for (const memberPub of memberPublicKeys) {
         const memberId = bytesToHex(memberPub);
         group.members.add(memberId);

         const wrappedSecret = await
this.wrapSecretForMember(groupSecret, memberPub);
         await this.messenger.send(memberId, {
           type: 'group-invite',
           groupId,
           epoch: 0,
           wrappedSecret,
         });
       }

       this.groups.set(groupId, group);
       return groupId;
```

```
    }

    // Wrap group secret for a specific member
    private async wrapSecretForMember(secret: Uint8Array, memberPub:
Uint8Array): Promise<Uint8Array> {
      const sessionKey = await
this.keyExchange.deriveSessionKey(memberPub);
      return aesGcmEncrypt(sessionKey, secret);  // Use Web Crypto
    }

    // Get the current topic key for encryption
    getTopicKey(groupId: string, topicId: string): Uint8Array {
      const group = this.groups.get(groupId)!;
      return hkdf(
        sha256,
        group.secret,
        topicId,
        `p2pos-topic-epoch-${group.epoch}`,
        32
      );
    }

    // Rotate group key (e.g., when removing a member)
    async rotateKey(groupId: string): Promise<void> {
      const group = this.groups.get(groupId)!;
      group.secret = randomBytes(32);
      group.epoch += 1;

      // Re-distribute to all remaining members
      for (const memberId of group.members) {
        // ... send new wrapped secret
      }
    }
  }

4. Create encryption utilities in src/crypto/aes.ts:
  // Use Web Crypto API for AES-GCM (audited browser implementation)

  export async function aesGcmEncrypt(key: Uint8Array, plaintext:
Uint8Array): Promise<Uint8Array> {
    const iv = crypto.getRandomValues(new Uint8Array(12));
    const cryptoKey = await crypto.subtle.importKey('raw', key, 'AES-
GCM', false, ['encrypt']);
    const ciphertext = await crypto.subtle.encrypt({ name: 'AES-GCM',
iv }, cryptoKey, plaintext);

    // Return iv + ciphertext
    const result = new Uint8Array(iv.length + ciphertext.byteLength);
    result.set(iv);
    result.set(new Uint8Array(ciphertext), iv.length);
    return result;
  }

  export async function aesGcmDecrypt(key: Uint8Array, data:
Uint8Array): Promise<Uint8Array> {
    const iv = data.slice(0, 12);
    const ciphertext = data.slice(12);
```

```
    const cryptoKey = await crypto.subtle.importKey('raw', key, 'AES-
GCM', false, ['decrypt']);
    const plaintext = await crypto.subtle.decrypt({ name: 'AES-GCM', iv
}, cryptoKey, ciphertext);
    return new Uint8Array(plaintext);
  }

5. Integrate with TopicLog (example):
  // In src/os/topic-log.ts
  async append(event: unknown, groupId: string): Promise<void> {
    const topicKey = this.groupKeyManager.getTopicKey(groupId,
this.topicId);
    const plaintext = new TextEncoder().encode(JSON.stringify(event));
    const ciphertext = await aesGcmEncrypt(topicKey, plaintext);

    const envelope = {
      groupId,
      epoch: this.groupKeyManager.getEpoch(groupId),
      ciphertext,
    };

    await this.store(envelope);
  }

6. Add to P2POS facade:
  interface P2POS {
    keys: {
      getPublicKey(): { identity: Uint8Array; exchange: Uint8Array };
      exchangeWith(peerExchangePublic: Uint8Array):
Promise<Uint8Array>;
      createGroup(members: Uint8Array[]): Promise<string>;
      addToGroup(groupId: string, memberPublic: Uint8Array):
Promise<void>;
      removeFromGroup(groupId: string, memberId: string):
Promise<void>;
      backup(password: string): Promise<Uint8Array>;  // encrypted
backup
      restore(backup: Uint8Array, password: string): Promise<void>;
    };
  }

7. Key backup/restore:
  import { scrypt } from '@noble/hashes/scrypt';

  async function backup(password: string): Promise<Uint8Array> {
    const salt = randomBytes(16);
    const derivedKey = scrypt(password, salt, { N: 2**17, r: 8, p: 1,
dkLen: 32 });
    const encrypted = await aesGcmEncrypt(derivedKey, serializedKeys);
    return concat(salt, encrypted);
  }

8. Testing:
  import { x25519 } from '@noble/curves/ed25519';
  import { randomBytes } from '@noble/hashes/utils';

  test('X25519 ECDH produces same shared secret', () => {
```

```
    const alicePrivate = randomBytes(32);
    const alicePublic = x25519.getPublicKey(alicePrivate);
    const bobPrivate = randomBytes(32);
    const bobPublic = x25519.getPublicKey(bobPrivate);

    const aliceShared = x25519.getSharedSecret(alicePrivate,
bobPublic);
    const bobShared = x25519.getSharedSecret(bobPrivate, alicePublic);

    expect(aliceShared).toEqual(bobShared);
  });

  test('encrypt/decrypt roundtrip', async () => {
    const key = randomBytes(32);
    const plaintext = new TextEncoder().encode('hello world');
    const ciphertext = await aesGcmEncrypt(key, plaintext);
    const decrypted = await aesGcmDecrypt(key, ciphertext);
    expect(decrypted).toEqual(plaintext);
  });

Existing code references:
- src/os/topic-log.ts: existing AES-GCM encryption
- src/os/messenger.ts: envelope handling
- src/overlay/adapter.ts: peer identity (libp2p peer ID)

Security notes:
- @noble/curves handles constant-time operations for crypto
- Web Crypto API is used for AES-GCM (browser-native, audited)
- Keys are stored encrypted in IndexedDB
- Group secrets are never transmitted in plaintext
```

E2E Test Coverage

Recommended Open Source: Playwright (already in use)

Metric	Value
Package	`@playwright/test` (https://www.npmjs.com/package/@playwright/test)
Weekly Downloads	8,000,000+
Backing	Microsoft
License	Apache-2.0

P2POS already uses Playwright - the recommendation is to leverage its advanced features:

Network simulation capabilities (built-in):

```
// Route interception for simulating network conditions
await page.route('**/*', route => {
  // Simulate latency
  await new Promise(r => setTimeout(r, 2000));
  route.continue();
});

// Offline simulation
await context.setOffline(true);

// Abort requests (simulate failures)
await page.route('**/api/**', route => route.abort());
```

Why Playwright is sufficient:

- Built-in network interception and mocking
- Offline mode simulation
- Multi-browser, multi-context support (perfect for Alice/Bob scenarios)
- Request/response modification

- WebSocket interception (for libp2p testing)
- Parallel test execution

Robustness assessment: Excellent

- Microsoft-backed, actively maintained
- Used by thousands of projects
- Excellent documentation
- You won't be debugging Playwright

Additional test utilities to consider:

Package	Purpose	Downloads
`fake-indexeddb`	Mock IndexedDB for storage tests	500K+
`msw`	Mock Service Worker for API mocking	3M+

What you need to build:

- Test harness for multi-peer orchestration
- Custom assertions for eventual consistency
- Test data factories

Complexity: Low-Medium (leveraging existing tooling)

Why It Is Needed

The current E2E tests cover basic Alice/Bob connectivity, collaboration sync, and SQL sync. However:

- **Edge cases are untested**: Network partitions, slow peers, message reordering.
- **Failure modes are unknown**: What happens when relay goes down? When peers disagree?
- **Confidence is low**: Hard to refactor or add features without breaking things.
- **Regression risk**: Bugs fixed once may return without comprehensive tests.

Thorough E2E tests are essential for a distributed system where bugs are hard to reproduce.

Complexity: Low-Medium

Factors:

- Test infrastructure (Playwright) already exists
- Need to simulate network conditions (latency, partitions)
- Multi-peer scenarios require careful orchestration
- Tests must be deterministic despite async/distributed nature

Estimated scope: 1-2 weeks

Potential Architecture

Test categories:

1. **Connectivity**: Peer discovery, connection establishment, reconnection.
2. **Messaging**: Send/receive, large messages, message ordering.
3. **Collaboration**: File create/update/delete, conflict scenarios, sync convergence.
4. **Event Sync**: TopicLog append, catchUp, SQL projection consistency.
5. **Failure Modes**: Relay down, peer crash, network partition, storage full.

Test matrix:

Scenario	Alice	Bob	Relay	Expected
Basic sync	Browser	Browser	Up	Sync completes
Offline edit	Browser (offline)	Browser	Up	Syncs when online
Relay failure	Browser	Browser	Down	Falls back or queues
Concurrent edit	Browser	Browser	Up	Conflict resolved
Node + Browser	Node	Browser	Up	Cross-env sync works

Implementation Prompt

```
Expand E2E test coverage for P2POS using Playwright's built-in features
for
network simulation and multi-browser orchestration.

Context:
- P2POS has existing E2E tests using Playwright in tests/e2e/.
- Current tests cover basic Alice/Bob connectivity, collaboration sync,
and SQL sync.
- The system involves Browser peers, Node peers, and relay
infrastructure.
- Distributed systems have many failure modes that need testing.

Dependencies (already installed):
  @playwright/test

Additional dev dependencies:
  npm install -D fake-indexeddb  # for unit tests with IndexedDB

Key Playwright APIs for P2POS testing:
- page.route(pattern, handler) - intercept/modify network requests
- context.setOffline(true/false) - simulate offline mode
- browser.newContext() - create isolated browser contexts (peers)
- expect.poll(fn, options) - retry assertions for eventual consistency
- test.describe.parallel() - run tests in parallel
- page.evaluate(fn) - execute code in browser context

Requirements:
1. Create test utilities in tests/e2e/utils/harness.ts:
   import { test, expect, Browser, BrowserContext, Page } from
'@playwright/test';

   // Peer orchestrator using Playwright contexts
   class PeerOrchestrator {
     private contexts: Map<string, BrowserContext> = new Map();
     private pages: Map<string, Page> = new Map();

     async createPeer(browser: Browser, name: string): Promise<Page> {
       const context = await browser.newContext();
       const page = await context.newPage();
       await page.goto('/');  // Load P2POS app
       this.contexts.set(name, context);
       this.pages.set(name, page);
       return page;
     }

     async destroyPeer(name: string): Promise<void> {
       await this.contexts.get(name)?.close();
       this.contexts.delete(name);
       this.pages.delete(name);
     }

     getPeer(name: string): Page { return this.pages.get(name)!; }
   }

   // Network simulator using Playwright route interception
   class NetworkSimulator {
```

```
      async addLatency(page: Page, ms: number): Promise<void> {
        await page.route('**/*', async route => {
          await new Promise(r => setTimeout(r, ms));
          await route.continue();
        });
      }

      async dropRequests(page: Page, pattern: string): Promise<void> {
        await page.route(pattern, route => route.abort('failed'));
      }

      async simulateOffline(context: BrowserContext): Promise<void> {
        await context.setOffline(true);
      }

      async simulateOnline(context: BrowserContext): Promise<void> {
        await context.setOffline(false);
      }
    }

    // Eventual consistency assertions
    async function expectEventually<T>(
      fn: () => Promise<T>,
      matcher: (value: T) => boolean,
      options = { timeout: 10000, interval: 100 }
    ): Promise<T> {
      return expect.poll(fn, options).toSatisfy(matcher);
    }

 2. Connectivity tests in tests/e2e/connectivity.spec.ts:
    import { test, expect, Browser } from '@playwright/test';
    import { PeerOrchestrator, NetworkSimulator } from './utils/harness';

    test.describe('Connectivity', () => {
      let orchestrator: PeerOrchestrator;
      let netSim: NetworkSimulator;

      test.beforeAll(async ({ browser }) => {
        orchestrator = new PeerOrchestrator();
        netSim = new NetworkSimulator();
      });

      test.afterAll(async () => {
        // Cleanup all peers
      });

      test('two browsers connect via relay', async ({ browser }) => {
        const alice = await orchestrator.createPeer(browser, 'alice');
        const bob = await orchestrator.createPeer(browser, 'bob');

        // Get Alice's peer ID
        const alicePeerId = await alice.evaluate(() =>
window.p2pos.peerId);

        // Bob connects to Alice
        await bob.evaluate((peerId) => window.p2pos.connect(peerId),
alicePeerId);
```

```
    // Verify connection established
    await expect.poll(
      () => bob.evaluate(() => window.p2pos.connectedPeers.length),
      { timeout: 10000 }
    ).toBeGreaterThan(0);
  });

  test('reconnection after disconnect', async ({ browser }) => {
    const alice = await orchestrator.createPeer(browser, 'alice');
    const bob = await orchestrator.createPeer(browser, 'bob');

    // Connect
    await bob.evaluate((id) => window.p2pos.connect(id),
      await alice.evaluate(() => window.p2pos.peerId));
    await expect.poll(
      () => bob.evaluate(() => window.p2pos.connectedPeers.length)
    ).toBeGreaterThan(0);

    // Disconnect (simulate offline)
    const bobContext = orchestrator.contexts.get('bob')!;
    await netSim.simulateOffline(bobContext);

    // Wait for disconnect detection
    await expect.poll(
      () => bob.evaluate(() => window.p2pos.connectedPeers.length)
    ).toBe(0);

    // Reconnect
    await netSim.simulateOnline(bobContext);
    await expect.poll(
      () => bob.evaluate(() => window.p2pos.connectedPeers.length),
      { timeout: 15000 }
    ).toBeGreaterThan(0);
  });

  test('connection with high latency', async ({ browser }) => {
    const alice = await orchestrator.createPeer(browser, 'alice');
    const bob = await orchestrator.createPeer(browser, 'bob');

    // Add 3 second latency to Bob
    await netSim.addLatency(bob, 3000);

    // Should still connect (with longer timeout)
    await bob.evaluate((id) => window.p2pos.connect(id),
      await alice.evaluate(() => window.p2pos.peerId));
    await expect.poll(
      () => bob.evaluate(() => window.p2pos.connectedPeers.length),
      { timeout: 30000 }
    ).toBeGreaterThan(0);
  });
});

3. Collaboration tests in tests/e2e/collaboration.spec.ts:
test.describe('Collaboration', () => {
  test('concurrent edits by two peers', async ({ browser }) => {
    const alice = await orchestrator.createPeer(browser, 'alice');
```

```
    const bob = await orchestrator.createPeer(browser, 'bob');

    // Both open same file
    await alice.evaluate(() =>
      window.p2pos.collaboration.openFile('/shared/doc.txt'));
    await bob.evaluate(() =>
      window.p2pos.collaboration.openFile('/shared/doc.txt'));

    // Both edit simultaneously
    await Promise.all([
      alice.evaluate(() =>
        window.p2pos.collaboration.set('/shared/doc.txt', 'Alice
edit')),
      bob.evaluate(() =>
        window.p2pos.collaboration.set('/shared/doc.txt', 'Bob
edit')),
    ]);

    // Wait for sync, verify both have same content (conflict
resolved)
    await expect.poll(async () => {
      const aliceContent = await alice.evaluate(() =>
        window.p2pos.collaboration.get('/shared/doc.txt'));
      const bobContent = await bob.evaluate(() =>
        window.p2pos.collaboration.get('/shared/doc.txt'));
      return aliceContent === bobContent;
    }, { timeout: 10000 }).toBe(true);
  });

  test('sync after offline edit', async ({ browser }) => {
    const alice = await orchestrator.createPeer(browser, 'alice');
    const bob = await orchestrator.createPeer(browser, 'bob');
    const bobContext = orchestrator.contexts.get('bob')!;

    // Initial sync
    await alice.evaluate(() =>
      window.p2pos.collaboration.set('/shared/doc.txt', 'initial'));
    await expect.poll(() => bob.evaluate(() =>

window.p2pos.collaboration.get('/shared/doc.txt'))).toBe('initial');

    // Bob goes offline
    await netSim.simulateOffline(bobContext);

    // Alice edits while Bob is offline
    await alice.evaluate(() =>
      window.p2pos.collaboration.set('/shared/doc.txt', 'alice
update'));

    // Bob comes back online
    await netSim.simulateOnline(bobContext);

    // Verify Bob gets the update
    await expect.poll(
      () => bob.evaluate(() =>
        window.p2pos.collaboration.get('/shared/doc.txt')),
      { timeout: 15000 }
```

```
        ).toBe('alice update');
      });
    });

4. Failure mode tests in tests/e2e/failure-modes.spec.ts:
   test.describe('Failure Modes', () => {
     test('relay unavailable - graceful degradation', async ({ browser
}) => {
       const alice = await orchestrator.createPeer(browser, 'alice');

       // Block relay requests
       await alice.route('**/relay/**', route => route.abort('failed'));

       // Attempt operation
       const result = await alice.evaluate(async () => {
         try {
           await window.p2pos.connect('some-peer-id');
           return { success: true };
         } catch (e) {
           return { success: false, error: e.message };
         }
       });

       // Should fail gracefully, not crash
       expect(result.success).toBe(false);
       expect(result.error).toContain('relay');  // Meaningful error
     });

     test('IndexedDB quota exceeded', async ({ browser }) => {
       const alice = await orchestrator.createPeer(browser, 'alice');

       // Fill up storage (mock quota error)
       await alice.evaluate(() => {
         // Override IndexedDB to throw QuotaExceededError
         const original = indexedDB.open;
         indexedDB.open = () => {
           throw new DOMException('QuotaExceededError');
         };
       });

       const result = await alice.evaluate(async () => {
         try {
           await window.p2pos.storage.put('large-data', new
Uint8Array(1000000));
           return { success: true };
         } catch (e) {
           return { success: false, error: e.message };
         }
       });

       expect(result.success).toBe(false);
       // App should not crash, error should be catchable
     });
   });

5. Performance baseline in tests/e2e/performance.spec.ts:
   test.describe('Performance', () => {
```

```
    test('sync latency baseline', async ({ browser }) => {
      const alice = await orchestrator.createPeer(browser, 'alice');
      const bob = await orchestrator.createPeer(browser, 'bob');

      // Connect peers
      // ... setup ...

      // Measure sync time for 100 events
      const startTime = Date.now();
      await alice.evaluate(async () => {
        for (let i = 0; i < 100; i++) {
          await window.p2pos.app.append({ type: 'test', i });
        }
      });

      // Wait for Bob to catch up
      await expect.poll(
        () => bob.evaluate(() =>
          window.p2pos.app.query('SELECT COUNT(*) as n FROM events').n)
      ).toBe(100);

      const elapsed = Date.now() - startTime;
      console.log(`100 events synced in ${elapsed}ms`);

      // Fail if > 20% regression from baseline (e.g., 5000ms)
      expect(elapsed).toBeLessThan(5000 * 1.2);
    });
  });

6. Test configuration in playwright.config.ts:
   import { defineConfig } from '@playwright/test';

   export default defineConfig({
     testDir: './tests/e2e',
     fullyParallel: true,
     retries: process.env.CI ? 2 : 0,
     workers: process.env.CI ? 1 : undefined,  // Serial in CI for relay
     use: {
       baseURL: 'http://localhost:5173',
       trace: 'on-first-retry',
     },
     webServer: {
       command: 'npm run dev',
       port: 5173,
       reuseExistingServer: !process.env.CI,
     },
     projects: [
       { name: 'chromium', use: { browserName: 'chromium' } },
       { name: 'firefox', use: { browserName: 'firefox' } },
     ],
   });

Existing code references:
- tests/e2e/libp2p-alice-bob.spec.ts: existing connectivity test
- signalling/run.ts: relay server
- bootstrap/server.ts: bootstrap server
```

```
Playwright patterns used:
- expect.poll() for eventual consistency (retries until condition met)
- context.setOffline() for network partition simulation
- page.route() for latency injection and request blocking
- test.describe.parallel() for independent test parallelization
- browser.newContext() for isolated peer environments
```

API Documentation

Recommended Open Source: TypeDoc

Metric	Value
Package	`typedoc` (https://www.npmjs.com/package/typedoc)
Weekly Downloads	2,684,701
Dependents	1,044
Last Published	13 days ago
License	Apache-2.0

Why TypeDoc:

- **Native TypeScript support**: Extracts types directly from TS source
- **JSDoc integration**: Combines code comments with type information
- **Customizable themes**: Can match Docusaurus styling
- **Markdown output**: Can generate MD files for Docusaurus integration
- **Monorepo support**: Works with workspaces
- **Active maintenance**: Regular updates

Robustness assessment: Excellent

- De facto standard for TypeScript documentation
- 2.6M+ weekly downloads
- Used by major TypeScript projects
- You won't be debugging TypeDoc

Integration with Docusaurus:

```
# Generate API docs as Markdown
npx typedoc --plugin typedoc-plugin-markdown --out docs/api src/index.ts
```

Recommended plugins:

Plugin	Purpose
`typedoc-plugin-markdown`	Output as Markdown for Docusaurus
`typedoc-plugin-merge-modules`	Cleaner module organization

What you need to write:

- JSDoc comments on public APIs
- Getting started guides
- Examples and tutorials

Complexity: Low (mostly documentation effort, tooling is mature)

Why It Is Needed

The alternatives doc mentions P2POS has a small API surface (`node.app.open`, `node.collaboration.openFile`, `node.messenger.send`), which is good for adoption. However:

- **No formal docs**: Developers must read source code to understand the API.
- **Unclear contracts**: What are the parameters? Return types? Error cases?
- **Missing examples**: How do common tasks look in practice?
- **Onboarding friction**: New contributors spend time figuring out basics.

Good API documentation is essential for developer adoption and contribution.

Complexity: Low

Factors:

- API surface is intentionally small
- Can generate docs from TypeScript types
- Examples can be extracted from existing tests
- Mainly a documentation effort, not code changes

Estimated scope: 1 week

Potential Architecture

Documentation structure:

```
docs/
├── intro.md                    # Existing
├── architecture.md             # Existing
├── api/
│   ├── overview.md             # API design principles
│   ├── p2pos.md                # createP2POS options and facade
│   ├── app.md                  # node.app API
│   ├── collaboration.md        # node.collaboration API
│   ├── messenger.md            # node.messenger API
│   ├── storage.md              # node.storage API
│   ├── directory.md            # node.directory API
│   └── pod.md                  # node.pod API (SOLID)
├── guides/
│   ├── getting-started.md      # Quick start
│   ├── browser-setup.md        # Browser-specific setup
│   ├── node-setup.md           # Node.js setup
│   └── building-an-app.md      # Tutorial
├── examples/
│   ├── todo-app.md             # Simple example
│   ├── collaborative-editor.md # Collaboration example
│   └── encrypted-notes.md      # Encryption example
└── roadmap/                    # This document
```

Implementation Prompt

```
Create comprehensive API documentation for P2POS using TypeDoc and
Docusaurus.

Context:
- P2POS has a small, intentional API surface for ease of adoption.
- The project uses Docusaurus for documentation (already set up).
- TypeScript source has type definitions that can be leveraged.
- Examples exist in tests but aren't documented.

Dependencies to install:
  npm install -D typedoc typedoc-plugin-markdown

TypeDoc configuration (typedoc.json):
{
  "entryPoints": ["src/index.ts"],
  "out": "docs-site/docs/api",
  "plugin": ["typedoc-plugin-markdown"],
  "readme": "none",
  "githubPages": false,
  "excludePrivate": true,
  "excludeInternal": true,
  "categorizeByGroup": true,
  "categoryOrder": ["P2POS", "App", "Collaboration", "Messenger",
"Storage", "*"]
}

Requirements:
1. Add JSDoc comments to all public APIs in src/:
   Example JSDoc format for TypeDoc:

   /**
    * Creates a new P2POS node instance.
    *
    * @remarks
    * This is the main entry point for using P2POS. The returned object
    * provides access to all P2POS subsystems.
    *
    * @param options - Configuration options for the node
    * @returns A promise that resolves to a P2POS node instance
    *
    * @throws ConfigError if options are invalid
    *
    * @example
    * import { createP2POS } from 'p2pos';
    *
    * const node = await createP2POS({
    *   persistent: true,
    *   observability: { level: 'info' }
    * });
    *
    * // Use the node
    * await node.messenger.send(peerId, { data: 'hello' });
    *
    * // Clean up
    * await node.stop();
    *
```

```
     * @category P2POS
     * @public
     */
   export async function createP2POS(options?: P2POSOptions):
Promise<P2POS> {
     // ...
   }

   Files to document:
   - src/p2pos.ts: createP2POS function and P2POSOptions interface
   - src/os/app.ts: App class with @category App
   - src/os/collaboration.ts: Collaboration class with @category
Collaboration
   - src/os/messenger.ts: Messenger class with @category Messenger
   - src/os/storage.ts: Storage/BlobStore with @category Storage
   - src/os/directory.ts: Directory with @category Directory
   - src/solid/pod.ts: Pod with @category SOLID

2. Generate API docs with TypeDoc:
   # Add to package.json scripts
   "scripts": {
     "docs:api": "typedoc",
     "docs:build": "npm run docs:api && cd docs-site && npm run build"
   }

   # Run generation
   npm run docs:api

   # Output structure (generated by typedoc-plugin-markdown):
   docs-site/docs/api/
   ├── README.md           # Module overview
   ├── classes/
   │   ├── App.md
   │   ├── Collaboration.md
   │   ├── Messenger.md
   │   └── ...
   ├── interfaces/
   │   ├── P2POSOptions.md
   │   ├── Envelope.md
   │   └── ...
   └── functions/
       └── createP2POS.md

3. Create manual guide pages in docs/guides/:

   # docs/guides/getting-started.md
   ---
   title: Getting Started
   sidebar_position: 1
   ---

   ## Installation

   npm install p2pos

   ## Quick Start
```

```
   import { createP2POS } from 'p2pos';

   // Create a node
   const node = await createP2POS();

   // Your peer ID (share this with others)
   console.log('My peer ID:', node.peerId);

   // Send a message to another peer
   await node.messenger.send(otherPeerId, {
     type: 'greeting',
     message: 'Hello from P2POS!'
   });

   // Listen for messages
   node.messenger.onMessage((from, envelope) => {
     console.log(`Message from ${from}:`, envelope);
   });

   ## Next Steps

   - Browser Setup - Configure for web apps
   - Building an App - Tutorial
   - API Reference - Full API documentation

4. Create example pages in docs/examples/:

   # docs/examples/todo-app.md
   ---
   title: Todo App Example
   ---

   This example shows how to build a syncing todo app with P2POS.

   import { createP2POS } from 'p2pos';

   const node = await createP2POS({ persistent: true });
   const app = await node.app.open('todo-app');

   // Define projection schema
   await app.exec(`
     CREATE TABLE IF NOT EXISTS todos (
       id TEXT PRIMARY KEY,
       text TEXT,
       done INTEGER DEFAULT 0
     )
   `);

   // Add a todo (event sourcing)
   async function addTodo(text: string) {
     await app.append({
       type: 'todo.add',
       id: crypto.randomUUID(),
       text,
       timestamp: Date.now()
     });
   }
```

```
   // Query todos
   async function getTodos() {
     return app.query('SELECT * FROM todos ORDER BY id');
   }

   // Sync with peers
   await app.catchUp();

5. Update docs-site/sidebars.js:
   const sidebars = {
     docsSidebar: [
       'intro',
       'problem-statement',
       'architecture',
       'implemented',
       {
         type: 'category',
         label: 'API Reference',
         link: { type: 'doc', id: 'api/README' },
         items: [
           'api/functions/createP2POS',
           'api/classes/App',
           'api/classes/Collaboration',
           'api/classes/Messenger',
           'api/classes/Storage',
         ],
       },
       {
         type: 'category',
         label: 'Guides',
         items: [
           'guides/getting-started',
           'guides/browser-setup',
           'guides/node-setup',
           'guides/building-an-app',
         ],
       },
       {
         type: 'category',
         label: 'Examples',
         items: [
           'examples/todo-app',
           'examples/collaborative-editor',
           'examples/encrypted-notes',
         ],
       },
       'use-cases',
       'alternatives',
       'pains-and-goals',
       {
         type: 'category',
         label: 'Roadmap',
         items: [
           'roadmap/index',
           'roadmap/persistent-storage',
           'roadmap/observability',
```

```
            'roadmap/access-control',
            'roadmap/key-management',
            'roadmap/e2e-tests',
            'roadmap/api-documentation',
          ],
        },
      ],
    };

6. TypeDoc + Docusaurus tips:
   - Use @category tags to group related APIs
   - Use @link tags for cross-references (e.g., @link Messenger.send)
   - Use @see tags for related APIs
   - Include runnable @example blocks
   - Mark internal APIs with @internal (excluded from docs)
   - Use @deprecated for deprecated APIs with migration path

7. Verify generated docs by running npm run docs:api and previewing with
   cd docs-site && npm start

Existing code references:
- docs-site/docusaurus.config.js: Docusaurus config
- docs-site/sidebars.js: Sidebar configuration
- src/p2pos.ts: Main entry point
- webapp/main.ts: Example usage

Documentation style:
- Concise but complete
- Code examples for every API
- Error cases documented
- Progressive disclosure (simple first, advanced later)
```