Federated SPV Relay Mesh: A Layered Architecture for Scalable Transaction Relay on Bitcoin SV

zcooL

February 2026 (Revised: February 27, 2026 — added Section 9, Supervision & Self-Healing; restructured layers to match production code)

Abstract

This paper presents a five-layer federated architecture for Simplified Payment Verification (SPV) relay nodes operating on the Bitcoin SV network. The system, designed and deployed for the Indelible platform (indelible.one), addresses the fundamental isolation problem of SPV nodes by decomposing SPV functionality into specialised modules with mesh networking and supervision layers between the Bitcoin peer-to-peer protocol and the application-facing API. The architecture comprises: (1) a P2P layer speaking native Bitcoin protocol version 70016 over TCP; (2) an SPV client layer managing peer connections, transaction broadcast via `inv`/`getdata`/`tx`, and transaction lookup via `getdata MSG_TX`; (3) a header synchronisation layer downloading and storing all block headers in a dual-indexed LevelDB database; (4) an API layer serving client applications through REST and WebSocket interfaces; and (5) a mesh layer providing bridge-to-bridge federation with on-demand transaction lookup, health monitoring, and loop prevention. A supervision layer spans all modules, providing crash resilience and self-healing, grounded in Nakamoto's principle that "nodes can leave and rejoin the network at will" [1]. The system operates in production across five geographically distributed nodes, maintaining header chain synchronisation across 938,000+ blocks, achieving inter-bridge mesh latencies of 3-49ms, and processing transaction broadcasts without dependence on third-party API services. No bloom filters are employed, as `NODE_BLOOM` is disabled by default on BSV full nodes. Instead, the system uses direct `getdata MSG_TX` requests and the proper `inv`/`getdata`/`tx` broadcast protocol as specified in the original Bitcoin whitepaper.

---

1. Introduction

Satoshi Nakamoto described Simplified Payment Verification in Section 8 of the Bitcoin whitepaper [1] as a method by which a node "can verify payments without running a full network node" by keeping "a copy of the block headers of the longest proof-of-work chain" and obtaining "the Merkle branch linking the transaction to the block it's timestamped in." This design permits lightweight clients to verify transaction inclusion with mathematical certainty without storing the full blockchain.

In practice, however, most SPV implementations operate as isolated clients. Each connects independently to one or a small number of Bitcoin full nodes, downloads its own copy of the header chain, and has no awareness of other SPV infrastructure. If the connected full node does not have a particular transaction in its mempool, the SPV client has no recourse. If the full node is slow or temporarily unreachable, the client experiences degraded service with no failover path.

This isolation becomes a material problem when SPV infrastructure must serve a production application. Indelible (indelible.one) stores AI conversation memory, encrypted files, and project archives permanently on the Bitcoin SV blockchain. Every save operation requires a transaction to be constructed, broadcast, and confirmed. The system's MCP (Model Context Protocol) server, thin CLI, and web application all depend on reliable, low-latency access to the Bitcoin network. A single SPV node connecting to a handful of full nodes is insufficient for this workload.

This paper describes the federated relay mesh architecture built to solve this problem. The system decomposes SPV functionality into five distinct layers – P2P wire protocol, SPV client, header synchronisation, API, and bridge mesh – with a supervision layer providing crash resilience and self-healing across all modules. The result is a network of SPV bridge nodes that cooperate to provide resilient, low-latency transaction relay without requiring any node to store the full blockchain.

---

2. Background

2.1 Simplified Payment Verification

The Bitcoin whitepaper [1] describes SPV in Section 8:

> "It is possible to verify payments without running a full network node. A user only needs to keep a copy of the block headers of the longest proof-of-work chain, which he can get by querying network nodes until he's convinced he has the longest chain, and obtain the Merkle branch linking the transaction to the block it's timestamped in."

This establishes two core requirements for SPV: maintenance of the header chain and the ability to verify Merkle proofs. Section 7 [1] explains the data structure that makes this possible – the Merkle tree. Each block header contains a Merkle root that commits to every transaction in the block. A Merkle branch (or proof) is a logarithmic-size path from a leaf transaction to the root, allowing verification without downloading the full block.

Section 2 [1] defines the transaction model itself: "We define an electronic coin as a chain of digital signatures." Each transaction takes inputs from previous transaction outputs and produces new outputs, forming a directed acyclic graph of ownership. In the Indelible system, each save operation creates a new link in this chain, with the payload encrypted and embedded in transaction outputs.

2.2 The Broadcast Protocol

Section 5 [1] specifies the network protocol: "New transactions are broadcast to all nodes." In the Bitcoin peer-to-peer protocol, this broadcast follows a specific sequence. A node announces it possesses a transaction by sending an `inv` (inventory) message containing the transaction identifier. Interested peers respond with a `getdata` message requesting the full transaction. The originating node then sends the complete `tx` message.

This three-step `inv`/`getdata`/`tx` handshake is essential. Earlier iterations of the Indelible relay system attempted to push raw `tx` messages directly to peers without the preceding `inv` announcement. Bitcoin full nodes consistently ignored these unsolicited transactions. The protocol requires the announcement-request-delivery flow; any deviation results in transactions never reaching the mining network.

2.3 The NODE_BLOOM Problem

BIP37 [2] introduced bloom filters as a mechanism for SPV clients to receive only transactions matching a particular pattern. An SPV client would send a `filterload` message containing a bloom filter, and the connected full node would apply this filter to incoming transactions and blocks, sending only matches.

On Bitcoin SV, `NODE_BLOOM` is disabled by default (`DEFAULT_PEERBLOOMFILTERS=false`). Sending a `filterload` message to a BSV full node triggers an immediate Misbehaving penalty of 100, resulting in a 24-hour ban from that peer. This is not a bug; it is a deliberate design decision reflecting the view that bloom filters introduce privacy leaks and impose computational burden on full nodes.

This means that the entire class of SPV implementations relying on bloom filters is inoperable on BSV. An alternative approach is required: direct `getdata` requests using `MSG_TX` (type 1) to fetch specific transactions by their identifier.

2.4 Overlay Networks

Wright's paper on overlay network architectures [3] (Section 3, "Network Architecture") describes a structured approach to node-to-node communication layered above the base Bitcoin protocol. The paper details Pastry DHT (Distributed Hash Table) routing for efficient message delivery, reputation scoring for peer quality assessment, formal node admission protocols, the S-net overlay for structured communication, and mechanisms for eclipse attack resistance.

This work provided the conceptual blueprint for the mesh layer described in this paper. While the current implementation uses a simpler fully-connected model with API key authentication rather than full Pastry DHT routing, the underlying principle is the same: overlay networks between SPV nodes can solve coordination problems that the base Bitcoin P2P layer was not designed to address.

---

3. Architecture Overview

The system is organised into five distinct layers, each with a dedicated module, protocol, and purpose.

Layer	Module	Protocol	Port	Purpose
P2P	`p2p.js`	TCP (Bitcoin)	8333	Wire-level Bitcoin protocol: binary message framing, checksum validation, handshake
SPV Client	`spv-client.js`	TCP (Bitcoin)	8333	Peer management, transaction broadcast via `inv`/`getdata`/`tx`, transaction lookup
Header Sync	`header-sync.js`	TCP (Bitcoin)	8333	Block header download, LevelDB dual-index storage (by height and hash)
API	`server-spv.js`	HTTP, WebSocket	8080	REST API and WebSocket interface serving clients, rate limiting, CORS, caching
Mesh	`bridge-mesh.js`	HTTP	8080	Bridge-to-bridge federation: health monitoring, on-demand tx forwarding, loop prevention

A supervision layer spans all modules, providing crash resilience and post-restart recovery (Section 9).

The layers are ordered by proximity to the Bitcoin network. The P2P layer handles the raw wire protocol. The SPV client layer manages peer connections and transaction operations. The header sync layer maintains the complete chain of block headers. The API layer faces outward to application clients. The mesh layer provides bridge-to-bridge communication for resilience.

Each layer is self-contained. The P2P layer can operate without the mesh layer. The SPV client can function with a single peer or many. The header sync can run independently as a standalone tool. This modularity ensures that partial failures do not cascade – if the mesh layer loses connectivity to peer bridges, the P2P layer continues to interact with the Bitcoin network directly. The supervision layer ensures that when failures do cause a process termination, the bridge recovers and resynchronises automatically.

4. P2P Layer

The P2P layer (`p2p.js`) implements the Bitcoin peer-to-peer protocol at the binary wire level.

4.1 Protocol Implementation

Messages are framed with the BSV mainnet magic bytes (`0xe3e1f3e8`), followed by a 12-byte null-padded command string, a 4-byte little-endian payload length, a 4-byte checksum (the first four bytes of the double-SHA256 of the payload), and the payload itself.

The system uses protocol version 70016, the current BSV protocol version that includes support for large messages, and identifies itself with user agent `/Bitcoin SV:1.1.0/`. These values must match expectations of BSV full nodes; non-standard user agents or protocol versions can result in connection rejection.

4.2 Connection Handshake

Connection establishment follows the Bitcoin protocol handshake:

1. The bridge opens a TCP connection to a full node on port 8333.
2. The bridge sends a `version` message containing its protocol version, services bitmask (set to 0, `NODE_NONE`, as the bridge is a pure SPV client), timestamp, and current block height.
3. The full node responds with its own `version` message, including its best block height.
4. The bridge sends a `verack` (version acknowledgement) message.
5. The bridge immediately sends a `protoconf` message advertising its maximum receive payload size (2MB), as required by protocol version 70016 and above.
6. The full node may send `authch` (authentication challenge) messages related to MinerID. These are ignored silently. Per the BSV node source code, connections proceed even if authentication is not completed. Non-mining SPV clients have no MinerID key and cannot sign authentication challenges.

4.3 Message Handling

The P2P layer handles all standard Bitcoin protocol messages: `version`, `verack`, `ping`/`pong`, `headers`, `inv`, `tx`, `notfound`, `getdata`, `reject`, `merkleblock`, `protoconf`, and `authch`. Unrecognised messages are logged and ignored. The layer emits events for each message type, allowing upper layers to register handlers without coupling to the wire protocol.

5. SPV Client Layer

The SPV client layer (`spv-client.js`) manages peer connections and transaction operations. It consumes events from the P2P layer and orchestrates multi-peer communication.

5.1 Peer Discovery

Peers are discovered through two mechanisms, in order of preference:

1. **WhatsOnChain Peer API**: The bridge queries `https://api.whatsonchain.com/v1/bsv/main/peer/info` for a current list of connected BSV nodes. Only outbound peers on port 8333

with protocol version 70015 or higher are selected, as these are known to accept incoming connections.

2. **DNS Seeds**: If the API returns insufficient peers, the bridge resolves DNS seed addresses from `seed.bitcoinsv.io`, `seed.satoshisvision.network`, and `seed.cascharia.com`.

Discovered peers are shuffled randomly to distribute connection load. The bridge attempts to maintain a configurable number of concurrent peer connections (default: 3) and runs a peer maintenance cycle every 30 seconds to ping connected peers, detect disconnections, and reconnect as needed.

5.2 Transaction Broadcasting

Transaction broadcast follows the `inv`/`getdata`/`tx` flow described in Section 5 of the Bitcoin whitepaper [1]. The SPV client:

1. Computes the transaction identifier (double-SHA256 of the raw bytes, reversed).
2. Stores the raw transaction in a pending broadcasts map.
3. Sends an `inv` message announcing the transaction to all connected peers.
4. When peers respond with `getdata`, the P2P layer serves the full `tx` message from the pending map.

This procedure is repeated to all connected peers. In the current deployment, transactions are broadcast to 12-14 peers simultaneously.

5.3 Transaction Fetching

Transactions are fetched using `getdata` with inventory type `MSG_TX` (1). The SPV client constructs an inventory vector containing the desired transaction identifier, sends the `getdata` message, and waits for the corresponding `tx` response. If the remote peer does not possess the transaction, it responds with a `notfound` message. The SPV client handles `notfound` explicitly, resolving the request as not-found immediately rather than waiting for a 10-second timeout. This optimisation was added after observing that many peers respond to unknown transaction requests with `notfound` rather than silence.

5.4 Inventory Handling

When a full node announces new transactions or blocks via `inv`, the SPV client responds with `getdata` to request the full data. Block inventory triggers a header re-synchronisation to update the local chain tip.

5.5 Address Watching

The SPV client maintains a set of watched addresses. When a transaction arrives (via `getdata` response or peer announcement), the client decodes it and checks whether any output script contains the hash160 of a watched address. Matching transactions are stored in a local LevelDB and emit events to subscribed WebSocket clients.

---

6. Header Synchronisation Layer

The header synchronisation layer (`header-sync.js`) downloads and stores all BSV block headers, as specified in Section 8 of the Bitcoin whitepaper [1].

6.1 Header Download

Headers are requested using the `getheaders` message with a block locator – a list of known block hashes at exponentially decreasing heights, always including the genesis block hash. This locator allows the remote peer to find the common point in the chain and send subsequent headers. Each response contains up to 2,000 headers.

The header sync layer preferentially uses live peers from the SPV client's active connections. If no live peers are available, it falls back to DNS seed discovery and connects independently.

6.2 Storage

Headers are stored in LevelDB with dual indexing: by height (`header:{height}`) and by hash (`height:{hash}`). This permits both sequential traversal and random access by block hash, the latter being essential for Merkle proof verification.

At the time of writing, the header chain spans 938,093 blocks. Initial synchronisation of this chain on a 1GB VPS requires the `--max-old-space-size=768` Node.js option to avoid out-of-memory conditions during the download of approximately 75MB of header data.

6.3 Merkle Proof Verification

As described in Section 7 of the Bitcoin whitepaper [1], each block header contains a Merkle root that commits to every transaction in the block. The header sync layer provides Merkle proof verification: given a transaction hash, a Merkle branch, a transaction index, and a block hash, the layer computes the Merkle root from the proof and compares it against the stored block header's Merkle root. A match proves the transaction's inclusion in the block with mathematical certainty.

---

7. Mesh Layer

7.1 Purpose

The mesh layer handles the case where a bridge needs a transaction it does not have locally. This situation arises when:

- A bridge was temporarily offline when the transaction was broadcast.
- The transaction was broadcast through a different bridge and has not yet propagated via the Bitcoin P2P network.
- The transaction is old and has been evicted from the pending transaction store.
- The transaction was broadcast by an external party and was never relayed through the Indelible bridge network.

In these cases, the mesh layer provides an on-demand lookup mechanism: "I do not have this transaction. Does any of my peers?"

7.2 Health Monitoring

Each bridge maintains a list of peer bridge URLs. Every 30 seconds, the mesh module pings each peer's `/health` endpoint. A peer is marked as healthy if it responds with `{"status": "ok"}` within the configured timeout (5 seconds). The response time is recorded as the peer's latency.

When the mesh needs to query peers, it sorts them by latency (fastest first) and queries only healthy peers. This ensures that requests are routed to the most responsive available node.

7.3 Query Mechanism

When the bridge's local lookup chain (described in Section 10) reaches the mesh layer, the mesh module constructs the same API path that the client originally requested (e.g., `/api/tx/{txid}`) and appends `?nomesh=1` as a query parameter.

The `nomesh=1` parameter is the loop prevention mechanism. When a bridge receives an API request with this parameter, it skips its own mesh lookup step. This prevents infinite forwarding: Bridge A asks Bridge B, Bridge B does not have it, Bridge B would normally ask its peers (including Bridge A), but the `nomesh=1` flag stops the recursion at one hop.

All healthy peers are queried in parallel. The first non-null, non-error response is returned to the caller. If no peer returns a valid response, the mesh returns null and the request falls through to a 404.

7.4 Network Topology

The current deployment comprises five nodes configured in a fully-connected mesh:

- Three nodes in Chicago
- One node in Dallas
- One node in Atlanta (enterprise tier, 4GB RAM)

Each node is configured with the URLs of all other nodes via the `MESH_PEERS` environment variable (excluding itself). On startup, the mesh module begins health-checking all configured peers.

---

8. API Layer

8.1 REST API

The API layer exposes a set of HTTP endpoints for Indelible clients:

- `GET /api/tx/:txid` – Retrieve a transaction by identifier (JSON format including decoded inputs and outputs)
- `GET /api/tx/:txid/hex` – Retrieve the raw transaction hex
- `POST /api/broadcast` – Broadcast a raw transaction to the Bitcoin network
- `GET /api/address/:address/balance` – Query address balance
- `GET /api/address/:address/unspent` – Query unspent transaction outputs (UTXOs)
- `GET /api/address/:address/history` – Query transaction history for an address
- `GET /api/mesh/status` – Report mesh peer health and latency
- `GET /api/woc-fallbacks` – Monitor remaining third-party API dependencies

- `GET /health` – System health including header height, peer count, and uptime

8.2 WebSocket API

Clients may also connect via WebSocket for real-time interaction. The WebSocket interface supports the same operations as the REST API, plus real-time subscriptions:

- `watchAddress` / `unwatchAddress` – Subscribe to transaction notifications for a specific address
- `broadcast` – Broadcast a transaction and receive the identifier
- `verifyTx` – Verify a transaction's inclusion in a block using the stored Merkle proof
- `getHeaderByHeight` / `getHeaderByHash` – Query specific block headers

When a watched address receives a transaction, the bridge pushes a notification to all subscribed WebSocket clients with the transaction identifier, raw hex, and confirmation status.

8.3 Security

The API layer implements several security measures:

- **API Key Authentication**: Server-to-server calls (e.g., from the Indelible Express backend to the SPV bridge) require a valid `X-API-Key` header. Requests with a valid API key bypass rate limiting and are granted permissive CORS.
- **Origin Whitelisting**: Browser requests are subject to CORS enforcement. Only requests from whitelisted origins (including `indelible.one` and configured development URLs) receive the `Access-Control-Allow-Origin` header. Requests from unrecognised origins are silently rejected by the browser's CORS policy.
- **Rate Limiting**: Requests without a valid API key are rate-limited to 100 requests per minute per IP address. Exceeding this limit returns HTTP 429 with a `Retry-After` header.

---

9. Supervision & Self-Healing

9.1 Purpose

The preceding five layers address the operational concerns of a running relay bridge: speaking the Bitcoin wire protocol (P2P), managing peer connections and transaction operations (SPV client), maintaining the header chain (header sync), serving client applications (API), and recovering missed transactions from peer bridges on demand (mesh). These layers assume the bridge process is alive.

The Bitcoin whitepaper addresses a different concern. Nakamoto states in the Abstract [1]:

> "Nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain as proof of what happened while they were gone."

Section 5 [1] elaborates on the tolerance built into the network protocol:

> "Block broadcasts are also tolerant of dropped messages. If a node does not receive a block, it will request it when it receives the next block and realizes it missed one."

And Section 12 [1] reiterates the principle:

> "Nodes can leave and rejoin the network at will, accepting the proof-of-work chain as proof of what happened while they were gone."

Bitcoin's design assumes three things about nodes: they crash, they recover, and they catch up. The first four layers of this architecture handled none of these. A bridge that crashed stayed dead. A bridge that restarted had no mechanism to verify it had recovered correctly. The supervision layer addresses this gap.

9.2 The Promise Double-Reject Problem

During a production audit of the relay mesh in February 2026, two bridges crashed simultaneously. The root cause was a class of bug present in three of the five operational layers: the promise double-reject.

In Node.js, a Promise executor that calls `reject()` more than once produces an unhandled rejection on the second call. The first rejection is caught by the `.catch()` handler. The second, having no handler, propagates as an `unhandledRejection` event. If this event has no listener, Node.js terminates the process.

The bug pattern was identical across all affected layers. Network operations – TCP connections, HTTP requests, header downloads – used Promises with both a timeout handler and an error/close handler. When a connection timed out and then closed (or closed and then timed out), both handlers called `reject()`. The first succeeded; the second killed the process.

Specific instances included:

- **P2P layer** (`p2p.js`): The `connect()` method used a timeout and a socket `close` event, both calling `reject()`. A slow peer that timed out and then closed the socket triggered a double rejection.
- **Header sync** (`header-sync.js`): The `headerPromise()` method registered event handlers for both timeout and error conditions. When a header download timed out, the timeout handler rejected the promise and the subsequent socket cleanup triggered the error handler, which rejected again.
- **Mesh layer** (`bridge-mesh.js`): The `_fetch()` and `_fetchRaw()` methods used both a timeout and an error handler on HTTP requests, producing the same double-reject pattern.

This bug class is particularly insidious because it only manifests under adverse network conditions – exactly when reliability matters most.

9.3 Process-Level Safety Net

The first line of defence is a process-level handler for unhandled rejections:

```
process.on('unhandledRejection', (reason, promise) => {
  console.error('[UNHANDLED REJECTION]', reason);
});
```

This handler catches any rejection that escapes all other handlers, logs the error, and prevents process termination. It is a safety net, not a solution – the underlying bugs must be fixed so that this handler never fires. However, its presence ensures that an unforeseen double-reject in any layer degrades to a logged warning rather than a crash.

9.4 Per-Layer Hardening

The production audit identified eight bugs across all five layers. Each was fixed with a targeted correction:

#	Layer	Module	Bug	Fix
1	API	`server-spv.js`	No `unhandledRejection` handler	Process-level safety net (Section 9.3)
2	P2P	`p2p.js`	`connect()` double-reject	Settled flag guards on `reject()` calls
3	Header Sync	`header-sync.js`	`headerPromise()` double-reject	Named handlers with mutual cleanup
4	P2P	`p2p.js`	Missing `notfound` handler	Explicit handler resolves lookup immediately (176ms vs 10s timeout)
5	API	`server-spv.js`	`wocFetch` no timeout	15-second timeout cap on all third-party API calls
6	Mesh	`bridge-mesh.js`	`_fetch`/ `_fetchRaw` double-reject	Settled flag pattern matching the P2P layer fix
7	API	`server-spv.js`	`pendingTxs` map never cleaned	5-minute sweep cycle with 1-hour TTL per entry
8	P2P	`p2p.js`	Raw `tx` push (unsolicited)	Correct `inv`/ `getdata`/`tx` protocol sequence per Section 5 [1]

The settled flag pattern (bugs 2 and 6) wraps the resolve/reject calls in a guard:

```
let settled = false;
const timeout = setTimeout(() => {
  if (settled) return;
  settled = true;
  reject(new Error('timeout'));
}, 10000);
socket.on('close', () => {
  if (settled) return;
  settled = true;
  reject(new Error('closed'));
});
```

This ensures that whichever event fires first completes the promise, and the second event is silently ignored.

The named handler pattern (bug 3) uses a different approach for cases where cleanup is required:

```
const onTimeout = () => {
  socket.removeListener('error', onError);
  reject(new Error('timeout'));
};
const onError = (err) => {
  clearTimeout(timer);
  reject(err);
};
```

Each handler removes the other before rejecting, ensuring mutual exclusion without a shared flag.

9.5 Recovery After Restart

When a bridge process restarts – whether from a crash, a deployment, or a manual restart – it must re-establish all five operational layers:

1. **P2P reconnection**: The P2P layer re-establishes TCP connections to Bitcoin full nodes via peer discovery (Section 5.1).
2. **Header chain re-synchronisation**: The header sync layer reconnects and requests headers from its last known height. The `getheaders` message with a block locator (Section 6.1) allows the peer to identify the divergence point and send only missing headers. A bridge that was down for minutes catches up in under a second; one that was down for hours may require several rounds of header batches.
3. **Mesh health resumption**: The mesh module resumes its 30-second health check cycle immediately on startup (Section 7.2). Peer bridges that were marked unhealthy during the outage are re-probed and restored to the healthy pool once they respond.
4. **Pending transaction recovery**: Transactions broadcast during the bridge's downtime are available through the mesh layer's on-demand lookup (Section 7.1). A client requesting a transaction that was broadcast while this bridge was offline triggers a mesh query, and any peer that has the transaction will serve it.

This recovery sequence implements Nakamoto's principle directly: the bridge "rejoins the network at will" and "accepts the proof-of-work chain as proof of what happened while it was gone" by re-synchronising its header chain from peers.

---

10. Transaction Lifecycle

This section traces the complete lifecycle of a transaction from client initiation to blockchain confirmation, illustrating how the layers interact.

10.1 Client Initiation

An Indelible client – whether the web application at indelible.one, the thin CLI, or an MCP-integrated AI agent – constructs a transaction. The transaction payload is encrypted using AES-256-GCM with a key derived from the user's WIF (Wallet Import Format) private key. The WIF never leaves the client device; only the encrypted payload is transmitted.

As described in Section 10 of the Bitcoin whitepaper [1], privacy is maintained by using separate keys for separate concerns. In the Indelible system, each user generates their own WIF locally. The server never possesses user keys and cannot decrypt user data.

The client signs the transaction using `@bsv/sdk` and submits the raw transaction hex to a bridge's `/api/broadcast` endpoint (API layer).

10.2 Broadcast

The API layer receives the raw transaction hex and passes it to the SPV client's `broadcastTx` method. The SPV client:

1. Computes the transaction identifier (double-SHA256 of the raw bytes, reversed).
2. Stores the raw transaction in the pending broadcasts map.
3. Sends an `inv` message to all connected Bitcoin full nodes, initiating the `inv`/`getdata`/`tx` flow described in Section 5 of the whitepaper [1].
4. When peers respond with `getdata`, the P2P layer serves the full `tx` message.
5. Stores the transaction in the pending transactions map with a timestamp.

10.3 Client Lookup

When a client subsequently requests the transaction (e.g., to verify a save), the bridge follows a deterministic lookup chain:

1. **Local cache**: An in-memory TTL cache. If the transaction was recently served, it is returned immediately without any I/O.
2. **Pending transactions**: The in-memory map of recently broadcast but unconfirmed transactions.
3. **P2P `getdata`**: A `getdata MSG_TX` request is sent to a connected Bitcoin full node. If the node has the transaction in its mempool, it responds with the full `tx` message. If not, it responds with `notfound`, and the bridge proceeds immediately.
4. **Mesh peers**: The bridge's mesh module queries all healthy peer bridges via HTTP. If any peer has the transaction (in its own cache, pending map, or from its own P2P connections), it returns the data.
5. **404**: If no source produces the transaction, the bridge returns HTTP 404 to the client.

This five-step chain ensures that the most local and fastest source is tried first, with progressively wider queries as fallbacks.

10.4 Confirmation

Section 3 of the Bitcoin whitepaper [1] describes the timestamp server: "each timestamp includes the previous timestamp in its hash, forming a chain." When a miner includes the transaction in a block, the block header – containing the Merkle root that commits to the transaction – becomes part of the header chain.

The bridge's periodic header synchronisation (every 60 seconds) detects the new block. As described in Section 7 [1], the Merkle tree structure allows the bridge to verify that a transaction is included in a block by checking a Merkle branch against the block header's Merkle root, without downloading the full block.

Section 11 [1] provides the probabilistic analysis of confirmation security: with each subsequent block, the probability that an attacker could reverse the transaction decreases exponentially. The bridge reports the number of confirmations (current header height minus the transaction's block height plus one) to clients on verification requests.

10.5 UTXO Management

Section 9 of the Bitcoin whitepaper [1] describes combining and splitting value: "transactions contain multiple inputs and outputs. Normally there will be either a single input from a larger previous transaction or multiple inputs combining smaller amounts, and at most two outputs: one for the payment, and one returning the change."

The Indelible system manages UTXOs for back-to-back broadcast operations. When a client performs rapid successive saves, each transaction consumes the change output of the previous transaction. This UTXO chaining requires that the bridge track pending (unconfirmed) transaction outputs and make them available for subsequent operations before the previous transaction has confirmed. The pending transactions map serves this purpose.

---

11. Security Considerations

11.1 Authentication and Access Control

The system employs layered authentication:

- **API key**: Server-to-server authentication between the Indelible application server and the SPV bridge. Relay keys (`relay_sk_*`) are validated against the Indelible backend with caching and usage tracking.
- **Origin whitelisting**: Browser-based access control via CORS. Only requests from whitelisted origins receive CORS headers.
- **Rate limiting**: Per-IP throttling for unauthenticated requests (100 requests per minute). API key holders bypass rate limits.

11.2 Data Encryption

All user data stored on the blockchain is encrypted with AES-256-GCM using a key derived from the user's WIF. The encryption uses a 12-byte random initialisation vector (IV), produces a 16-byte authentication tag, and the encrypted output is formatted as `iv:tag:ciphertext` in

Base64 encoding. The authentication tag provides integrity verification – any modification to the ciphertext is detected during decryption. The user's WIF private key is the sole decryption key and never leaves the client.

11.3 Loop Prevention

Mesh queries use the `?nomesh=1` query parameter to prevent forwarding loops. When Bridge A queries Bridge B and includes `nomesh=1`, Bridge B performs its own local lookup (cache, pending, P2P) but does not query its own mesh peers. This limits mesh queries to a single hop, preventing circular dependencies.

11.4 The Bloom Filter Ban

As noted in Section 2.3, sending a `filterload` message to a BSV full node results in an immediate Misbehaving score of 100, which causes a 24-hour ban. The system avoids this entirely by never using bloom filters. All transaction lookups are performed via `getdata MSG_TX` with explicit transaction identifiers. This is both safer (no bloom filter privacy leaks) and simpler (no filter management).

11.5 Protocol Compliance

The system maintains strict compliance with the Bitcoin P2P protocol to avoid triggering misbehaviour penalties:

- Protocol version 70016 with `protoconf` message sent after handshake.
- User agent string matching known BSV client patterns.
- Proper `version`/`verack` handshake sequence.
- Correct checksum validation (double-SHA256) on all messages.
- Proper `pong` responses to `ping` messages for keepalive.
- Silent handling of `authch` messages from mining nodes.
- Relay flag set to 0 (SPV client does not relay).

---

12. Performance

12.1 Production Deployment

The system operates across five VPS nodes:

Node	Location	RAM	Peers	Header Height	Mesh Latency
Bridge 1	Dallas	1GB	12-14	938,093	–
Bridge 2	Chicago	1GB	12-14	938,093	3-12ms
Bridge 3	Chicago	1GB	12-14	938,093	5-15ms
Bridge 4	Chicago	1GB	12-14	938,093	5-15ms
Bridge 5	Atlanta	4GB	12-14	938,093	20-35ms

Header heights are synchronised across all nodes. The Chicago-to-Chicago mesh latency is consistently under 15ms. The Chicago-to-Dallas and Chicago-to-Atlanta latencies reflect the physical network distances but remain well within acceptable bounds. Bridge 5 operates as an enterprise-tier node with 4GB RAM, providing additional capacity for high-throughput operations.

12.2 Mesh Health

The mesh health check runs every 30 seconds. In the production deployment, the failure rate across all nodes is zero – all configured peers consistently report healthy status. The mesh module sorts peers by latency, ensuring that the fastest responder is always queried first during on-demand lookups.

12.3 Broadcast Performance

Transaction broadcasts reach 12-14 Bitcoin full nodes simultaneously via the P2P layer. The total time from client submission to full network propagation is dominated by the TCP round-trip time to the Bitcoin full nodes, typically 50-200ms.

12.4 Header Synchronisation

Initial header synchronisation of the full 938,000+ block header chain takes approximately 15-30 minutes depending on the peer's throughput. Subsequent synchronisation (catching up to the chain tip) typically involves one or two `getheaders` round-trips per new block and completes in under one second. The periodic synchronisation interval of 60 seconds ensures that the bridge is never more than approximately one block behind the network tip.

---

13. Related Work

13.1 The Bitcoin Whitepaper

This system is a direct implementation of the principles described by Nakamoto [1]. Section 8 provides the theoretical foundation for SPV. Section 5 defines the broadcast protocol and establishes the network's tolerance of dropped messages. Section 7 describes the Merkle tree structure that makes lightweight verification possible. Section 2 establishes the transaction model. Section 3 establishes the timestamping property that gives blockchain data its permanence. Section 9 governs the UTXO management required for efficient operation. The Abstract and Section 12 establish the crash tolerance principle – "nodes can leave and rejoin the network at will" – which grounds the supervision and self-healing layer described in Section 9 of this paper.

The system departs from the whitepaper only where the current state of the BSV network requires it – specifically, the inability to use bloom filters due to `NODE_BLOOM` being disabled, which necessitates the direct `getdata MSG_TX` approach.

13.2 Overlay Network Architectures

Wright [3] describes overlay networks built atop the Bitcoin peer-to-peer layer, with particular attention to structured routing (Pastry DHT), reputation scoring, and eclipse attack resistance. Section 3 of that paper outlines an architecture where specialised nodes form an overlay network with formal admission protocols and quality-of-service guarantees.

The mesh layer described in this paper implements a simplified version of this overlay concept. The current deployment uses a static peer list and API key authentication rather than DHT routing and reputation scoring. However, the modular architecture accommodates future evolution toward the more sophisticated mechanisms described in Wright's work (see Section 14, Future Work).

13.3 Electrum and Similar SPV Systems

Electrum-style SPV systems (ElectrumX, Fulcrum, etc.) use a client-server model where dedicated indexing servers maintain full address-to-transaction indexes and serve SPV clients over a custom protocol. While effective, this model introduces a trusted third party – the indexing server operator.

The Indelible relay mesh eliminates this dependency for transaction broadcast and lookup. The bridges connect directly to Bitcoin full nodes and maintain their own header chains. Third-party API fallbacks (e.g., WhatsOnChain) exist only for address-based indexing operations (balance, UTXO queries) that require full blockchain scanning, and these fallbacks are tracked with the explicit goal of eventual elimination.

---

14. Future Work

14.1 Relay Peering Layer

The current mesh layer provides on-demand transaction lookup between bridges via HTTP. A relay peering layer would add proactive, real-time transaction propagation via authenticated WebSocket connections. When a transaction is broadcast through one bridge, the relay peering layer would push it to all peer bridges immediately, eliminating the delay of waiting for the Bitcoin P2P network to propagate the transaction naturally. This layer would operate on a dedicated port with HMAC-SHA256 authentication, deduplication caching, and automatic reconnection with exponential backoff. Combined with the existing mesh layer, it would provide both push (relay peering) and pull (mesh) bridge-to-bridge communication.

14.2 Merkle Tree Pruning

Section 7 of the Bitcoin whitepaper [1] describes reclaiming disk space through Merkle tree pruning: "Once the latest transaction in a coin is buried under enough blocks, the spent transactions before it can be discarded to save disk space." Implementing this in the bridge would allow long-running nodes to discard old transaction data while retaining the ability to verify recent transactions. The header chain, which is essential for SPV verification, is never pruned.

14.3 Advanced UTXO Management

Section 9 [1] describes combining and splitting value. The current system handles UTXO chaining for sequential broadcasts, but a more sophisticated UTXO management system could pre-split UTXOs into parallel chains, enabling concurrent broadcast operations without contention. This is relevant for high-throughput Indelible users performing many saves in rapid succession.

14.4 Pastry DHT Routing

Wright's overlay network paper [3] describes Pastry DHT routing for efficient message delivery in overlay networks. Replacing the current static seed list with DHT-based routing would allow the relay mesh to scale beyond the current model of fully connected peers. In a DHT model, each bridge would maintain a routing table and forward relay messages to the appropriate subset of peers, reducing the O(n) message cost of the current broadcast model to O(log n).

14.5 Reputation Scoring

Wright [3] describes reputation scoring where peers are rated based on their reliability, latency, and adherence to protocol. The mesh layer already tracks peer latency and failure counts. Formalising this into a reputation score that influences peer selection priority and, potentially, triggers automatic exclusion of consistently unreliable peers would improve the system's resilience.

14.6 Node Admission Protocols

The current mesh layer uses API key authentication – any node with a valid key is admitted. As the network grows, a formal admission protocol as described in Wright [3] would provide finer-grained access control. This could involve cryptographic identity verification, stake-based admission, or a web-of-trust model where existing nodes vouch for new entrants.

14.7 Process Supervision

The current supervision layer (Section 9) hardens the bridge process against crashes but does not automatically restart a terminated process. Integration with operating system process managers – such as `systemd` on Linux – would provide automatic restart on crash, ensuring that a bridge returns to service without manual intervention. This would complete the "rejoin the network at will" principle from Nakamoto [1] by making the rejoining automatic rather than operator-initiated.

14.8 Degraded Health Reporting

When a bridge restarts after a crash, it currently reports healthy as soon as its HTTP server is listening, even if its header chain is still synchronising or its mesh connections have not yet re-established. A degraded health state – reported via the `/health` endpoint and propagated to mesh peers – would allow clients and load balancers to route traffic away from a recovering bridge until it has fully caught up. This is analogous to Nakamoto's observation that a node must "accept the proof-of-work chain" before it can participate fully [1].

14.9 Structured Context Snapshots

AI coding assistants operate within finite context windows. When a context window fills, the assistant compacts its memory, losing structured working state – active task lists, implementation plans, code change traces, and git branch context. The Indelible system now captures this structured context alongside conversation messages before compaction occurs.

The pre-compaction hook extracts metadata from the Claude Code transcript (session identifier, git branch, active plan slug) and reads the current task state from the assistant's todo persistence layer. This structured context is attached to the session object as a `structured_context` field, encrypted alongside the conversation messages, and committed to the blockchain in the same transaction.

On restoration after compaction, the structured context is rendered before the conversation history: git branch, active plan name, and a task checklist with completion status. The assistant receives not only what was discussed but what was being done – enabling it to resume multi-step work without re-derivation.

This represents a shift from storing conversation logs to storing working state. The blockchain payload is no longer just a record of what was said; it is a snapshot of an active development session, including its task decomposition and progress. Delta saves ensure that each subsequent save captures the latest structured context, and the delta merge logic carries `structured_context` forward from the most recent delta to the merged session.

---

15. Conclusion

The federated SPV relay mesh described in this paper transforms isolated SPV nodes into a cooperating, self-healing network. By decomposing SPV functionality into five distinct layers – P2P wire protocol, SPV client, header synchronisation, API, and bridge mesh – with supervision spanning all modules, the system achieves resilient, low-latency transaction relay without requiring any node to store the full blockchain.

The architecture is grounded in the principles of the Bitcoin whitepaper [1]: SPV verification via header chains and Merkle proofs (Section 8), the `inv`/`getdata`/`tx` broadcast protocol (Section 5), the chain of digital signatures securing each transaction (Section 2), the timestamp server providing provable ordering (Section 3), the privacy model of user-controlled keys (Section 10), and the crash tolerance principle that "nodes can leave and rejoin the network at will" (Abstract, Section 12).

The overlay network concept draws from Wright's work on structured peer-to-peer communication [3], adapting the principles of node-to-node coordination to the specific requirements of an SPV relay federation.

The system operates in production serving the Indelible platform (indelible.one), where it handles the storage of AI conversation memory, encrypted file archives, and project codebases on the Bitcoin SV blockchain. Five geographically distributed nodes maintain synchronised header chains across 938,000+ blocks, achieve inter-bridge mesh latencies of 3-49ms, and broadcast transactions to the Bitcoin network without dependence on third-party API services.

The code is not theoretical. It is deployed, running, and processing real transactions on the Bitcoin SV mainnet.

References

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008. Available: https://bitcoin.org/bitcoin.pdf

[2] M. Hearn, M. Corallo, “BIP 37: Connection Bloom filtering,” 2012. Available: https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki

[3] C. S. Wright, “Overlay Network Architecture for Bitcoin Scaling,” SSRN Electronic Journal, SSRN-6277825, 2025. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6277825

Indelible – Permanent memory, secured by proof of work.

https://indelible.one