Federated SPV Relay Mesh: An Open Protocol for Decentralised Transaction Relay on Bitcoin SV

zcooL

February 2026 (Revised: March 11, 2026 — v3: added indexing layer with tx confirmation tracking, Merkle proof storage, BSV-20 token indexing, content-addressed inscription storage, historical backfill, and price feed. v2: on-chain registry, stake bonds, peer scoring, cryptographic handshake, gossip protocol, protocol parsing. Reframed as open BSV developer infrastructure.)

Abstract

This paper presents a federated architecture for Simplified Payment Verification (SPV) relay nodes operating on the Bitcoin SV network. The system addresses the fundamental problem facing BSV application developers: the choice between running a full node (expensive, complex) and depending on centralised third-party APIs (single point of failure, someone else's uptime). The relay mesh provides a middle ground – lightweight bridge nodes that peer with each other over an open, decentralised protocol, using the BSV blockchain itself for peer discovery via an on-chain CBOR registry.

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; (5) a federation layer providing bridge-to-bridge peering with cryptographic identity, on-chain registration, stake bonds, and local peer scoring; (6) a supervision layer spanning all modules for crash resilience and self-healing; and (7) an indexing layer providing transaction confirmation tracking, Merkle proof storage, BSV-20 token accounting, content-addressed inscription storage, and historical backfill.

Bridge identity is tied to a BSV keypair. Registration is published on-chain via CBOR-encoded OP_RETURN transactions. Stake bonds lock satoshis to the bridge operator's own address as proof of BSV ownership. Peer scoring – computed locally by each bridge with no central authority – weights data accuracy (40%), uptime (30%), response time (20%), and stake age (10%). Peers below a threshold score are auto-disconnected.

The system operates in production across geographically distributed nodes, deployed in production for the Indelible platform (indelible.one) as a reference implementation. Each bridge connects to 250+ Bitcoin full nodes via native P2P peer exchange (`getaddr`/`addr`), maintains header chain synchronisation across 939,000+ blocks, and processes transaction broadcasts without dependence on any third-party API services. The protocol is open – any developer can run a bridge, register on-chain, and join the mesh.

---

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 is a material problem for any developer building on BSV. Every blockchain ecosystem faces the same tooling gap: Ethereum has Infura and Alchemy; Solana has Helius and QuickNode. These services provide API access to blockchain networks without running a full node. On BSV, the options are sparse. Developers either run a full node (requiring significant disk, bandwidth, and operational overhead) or depend on centralised APIs with opaque uptime guarantees and rate limits they do not control.

The relay mesh fills this gap. It provides BSV developers with a third option: run a lightweight bridge node that peers with other bridges, syncs headers, and relays transactions – all without storing the full blockchain. The protocol is open. Any developer can run a bridge, register it on-chain with a stake bond, and join the mesh. Discovery is decentralised: bridges find each other by scanning the BSV blockchain for registry transactions. Trust is local: each bridge scores its peers independently based on observed behaviour, with no central reputation authority.

The system was designed and first deployed for the Indelible platform (indelible.one), which stores AI conversation memory, encrypted files, and project archives permanently on the Bitcoin SV blockchain. The relay mesh proved that federated SPV infrastructure can serve production blockchain applications with sub-second latency and zero third-party dependencies. This paper describes the architecture, protocol, and production deployment as a reference for the broader BSV developer community.

---

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.

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 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 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 federation layer described in this paper. The current implementation realises several of the mechanisms Wright described: reputation scoring (Section 11.8), node admission via on-chain registration and stake bonds (Sections 7.2-7.3), and cryptographic identity verification (Section 7.5). The system uses a gossip-based peer discovery model rather than full Pastry DHT routing, with DHT routing deferred as future work for larger mesh sizes.

---

3. Architecture Overview

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

Layer	Module	Protocol	Port	Purpose
P2P	`bsv-peer.js`	TCP (Bitcoin)	8333	Wire-level Bitcoin protocol: binary message framing, checksum validation, handshake
SPV Client	`bsv-node-client.js`	TCP (Bitcoin)	8333	Peer management, transaction broadcast via `inv`/`getdata`/`tx`, transaction lookup
Header Sync	(integrated)	TCP (Bitcoin)	8333	Block header download, LevelDB dual-index storage (by height and hash)
Indexing	`persistent-store.js`	LevelDB	—	Tx confirmation tracking, Merkle proofs, BSV-20 tokens, CAS inscription storage, backfill
API	`status-server.js`	HTTP, WebSocket	9333	REST API, WebSocket interface, operator dashboard
Federation	`handshake.js`, `gossip.js`, `peer-scorer.js`, `registry`	WebSocket, BSV OP_RETURN	8333	On-chain registration, cryptographic peering, gossip discovery, local peer scoring
Supervision	(integrated)	—	—	Crash resilience, post-restart recovery

The layers are ordered by proximity to the Bitcoin network. The P2P layer handles the raw wire protocol. The SPV client manages peer connections and transaction operations. The header sync maintains the complete chain of block headers. The indexing layer tracks transaction lifecycle, stores Merkle proofs, indexes BSV-20 tokens, and manages content-addressed inscription storage.

The API layer faces outward to application clients. The federation layer provides bridge-to-bridge communication, identity, and trust. The supervision layer ensures crash recovery.

Each layer is self-contained. A bridge can operate without the federation layer – it simply has no peers and no mesh resilience. The federation layer can operate without the API layer – bridges peer and relay transactions without serving external clients. This modularity ensures that partial failures do not cascade.

The architecture is designed so any developer can run a bridge. The system is packaged as three npm packages (`@relay-federation/common`, `@relay-federation/registry`, `@relay-federation/bridge`) and installs with a single command: `npm install -g @relay-federation/bridge`.

---

4. P2P Layer

The P2P layer (`bsv-peer.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 (`bsv-node-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 that operate in sequence:

1. **DNS Seeds**: On startup, the bridge resolves DNS seed addresses from `seed.bitcoinsv.io`, `seed.satoshisvision.network`, and `seed.cascharia.com`. This typically yields 3-5 unique IP addresses for initial bootstrap.
2. **Native P2P Peer Exchange (`getaddr`/`addr`)**: After completing the handshake with each initial peer, the bridge sends a `getaddr` message – the Bitcoin protocol's built-in peer discovery mechanism. Each connected node responds with an `addr` message containing the IP addresses and ports of all nodes in its address book. The bridge parses these `addr` messages (30-byte entries: 4 bytes timestamp, 8 bytes services, 16 bytes IPv4-mapped IPv6 address, 2 bytes port), filters for IPv4 nodes on port 8333, and connects to each new peer. Those new peers also respond to `getaddr` with their own address books, creating a cascading discovery effect that rapidly maps the entire reachable network.

This two-stage approach – DNS bootstrap followed by native P2P peer exchange – eliminates all third-party API dependencies for peer discovery. The `getaddr`/`addr` approach uses the peer exchange protocol Nakamoto built into Bitcoin itself, discovers significantly more peers (250+ versus 12-14 with API-based approaches), and operates with zero external dependencies.

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.

This procedure is repeated to all connected peers. In the current deployment, transactions are broadcast to 250+ peers simultaneously, reaching the vast majority of the reachable BSV network.

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.

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 clients.

---

6. Header Synchronisation Layer

The header synchronisation layer 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.

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 939,000+ 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. Federation Layer

The federation layer replaces the static peer lists and shared-secret authentication of earlier relay mesh designs with an open, decentralised protocol. Any bridge can join by registering on-chain. Trust is earned through observed behaviour, not granted by configuration.

7.1 Purpose

The federation layer handles three problems:

1. **Discovery**: How does a new bridge find existing bridges to peer with?
2. **Identity**: How does a bridge prove it is who it claims to be?
3. **Trust**: How does a bridge decide which peers to keep and which to disconnect?

The solutions are: on-chain registry for discovery, cryptographic handshake for identity, and local peer scoring for trust.

7.2 On-Chain Registry

Bridge registration is published on-chain using CBOR-encoded OP_RETURN transactions. The protocol prefix `indelible.bridge-registry` identifies registry transactions in the OP_RETURN data.

A registration transaction has two outputs:

1. **OP_RETURN output** (0 satoshis): Contains the protocol prefix followed by a CBOR-encoded payload with the following fields:

Field	Type	Description
`action`	string	`"register"` or `"deregister"`
`endpoint`	string	WebSocket endpoint (e.g., `wss://bridge.example.com:8333`)
`pubkey`	Uint8Array (33 bytes)	Compressed public key – the bridge's identity
`capabilities`	string[]	Subset of: `tx_relay`, `header_sync`, `broadcast`, `address_history`
`versions`	string[]	Supported protocol versions (e.g., `["1.0"]`)
`network_version`	string	Current network version
`stake_txid`	Uint8Array (32 bytes)	Transaction ID of the stake bond
`mesh_id`	string	Mesh identifier (e.g., `"70016"`)
`timestamp`	number	Unix timestamp in seconds

2. **Dust output** (100 satoshis): Sent to the beacon address (`1KhH4VshyN8PnzxbTSjiojcQbbABNSZyzR`), a deterministic address derived from SHA-256 of the protocol prefix. This makes all registry transactions discoverable via a single address history query.

Deregistration uses the same protocol prefix with `action: "deregister"`, the bridge's pubkey, a reason string, and a timestamp. The latest transaction for a given pubkey wins, supporting endpoint updates and re-registration.

7.3 Stake Bonds

Before registering, a bridge must create a stake bond – a transaction that locks a minimum of 1,000,000 satoshis (~0.01 BSV) to the bridge operator's own address. The stake bond transaction ID is included in the registration payload.

Parameter	Value
Minimum	1,000,000 sats (~0.01 BSV)
Purpose	Proof of BSV ownership, Sybil deterrence
Scoring weight	10% (stake_age factor in peer scoring)
Recovery	Deregister to unlock

The stake is not a payment – it is a security deposit locked to the operator's own address. The economic cost of registering fake bridges provides Sybil deterrence. Higher stakes do not buy significant advantage in scoring (only 10% weight); the real defense against bad actors is data accuracy scoring (40% weight).

Operators can voluntarily stake more than the minimum for a slightly higher trust score, but the marginal benefit is logarithmic – doubling the stake adds minimal score improvement.

7.4 Peer Discovery

Bridges discover each other through three mechanisms:

1. **Seed peers**: On startup, the bridge connects to peers listed in its `seedPeers` configuration. These are known bridges with verified pubkeys.
2. **Gossip protocol**: Once connected to at least one peer, the bridge's gossip manager requests peer lists (`getpeers` message) and broadcasts signed announcements (`announce` message). Announcements include the bridge's pubkey, endpoint, mesh ID, and timestamp, signed with the bridge's private key. Peers verify the signature and propagate valid announcements to their own peers, creating a flooding discovery mechanism.
3. **On-chain beacon watcher**: The bridge monitors the beacon address for new registration transactions. When a new registration arrives on-chain, the bridge extracts the CBOR payload, verifies the registration fields, and adds the peer to its known peer set.

The gossip protocol provides real-time discovery (seconds), while the on-chain registry provides permanent, auditable discovery (minutes to hours). Both mechanisms filter by `mesh_id` – bridges only peer within the same mesh.

7.5 Cryptographic Handshake

Every bridge-to-bridge connection begins with a mutual authentication handshake using BSV keypairs. The protocol requires three messages:

```
1. Initiator → Responder: { type: "hello", pubkey, nonce, versions, endpoint }
2. Responder → Initiator: { type: "challenge_response", pubkey, nonce, signature,
selected_version }
3. Initiator → Responder: { type: "verify", signature }
```

Step 1: The initiator sends a hello containing its compressed public key, a 32-byte random nonce, its supported protocol versions, and its endpoint.

Step 2: The responder verifies the initiator's pubkey against the set of registered pubkeys (if available). It selects the highest mutually supported protocol version. It signs the initiator's nonce with its own private key (proving identity) and sends a challenge_response containing its own pubkey, a new nonce, the signature, and the selected version.

Step 3: The initiator verifies the responder's signature against the responder's pubkey and the original nonce. If valid, it signs the responder's nonce and sends a verify message.

The responder verifies the final signature. If valid, the connection is established with mutual authentication complete. Both sides have proven possession of the private keys corresponding to their claimed public keys.

Connections that do not complete the handshake within 10 seconds are dropped.

7.6 Real-Time Transaction Relay

Once peered, bridges relay transactions to each other in real time over persistent WebSocket connections. When a transaction is broadcast through one bridge (via its API or P2P layer), the bridge propagates it to all connected peer bridges immediately. This provides push-based propagation – peer bridges receive the transaction without waiting for the Bitcoin P2P network to propagate it naturally.

7.7 Header Sharing

Bridges share block headers with peers as they arrive. When a bridge receives a new header from the Bitcoin P2P network, it propagates the header to its peer bridges. This ensures that all bridges in the mesh maintain synchronised header chains, even if individual bridges have intermittent Bitcoin P2P connectivity.

---

8. API Layer

8.1 REST API

The API layer exposes HTTP endpoints for bridge operators and client applications:

- `GET /status` – Bridge status: pubkey, mesh ID, uptime, peers, headers, mempool
- `GET /tx/:txid` – Retrieve a transaction by identifier with full protocol parsing (see Section 12)
- `GET /tx/:txid/status` – Transaction lifecycle state (mempool, confirmed, orphaned, dropped)
- `GET /proof/:txid` – Merkle proof for confirmed transactions with block context
- `GET /mempool` – Current mempool with parsed outputs and protocol badges
- `GET /inscriptions` – Query indexed inscriptions by mime type, address, or time range
- `GET /inscription/:txid/:vout/content` – Serve raw inscription content with CAS resolution
- `GET /price` – Live BSV/USD exchange rate with 60-second cache
- `GET /tokens` – List all deployed BSV-20 tokens indexed by the bridge

- `GET /token/:tick` – Deploy info for a specific BSV-20 token
- `GET /token/:tick/balance/:scriptHash` – Token balance for an owner by script hash
- `GET /apps` – Health status of applications running on the bridge
- `POST /broadcast` – Broadcast a raw transaction to the Bitcoin network

8.2 Operator Dashboard

Each bridge runs a local HTTP server (default port 9333) with a glassmorphism dashboard. The dashboard provides five tabs:

- **Overview**: Bridge stats, peer connections, header height, wallet balance
- **Mempool**: Real-time transaction list with protocol badges and decoded data
- **Tx Explorer**: Transaction lookup with full protocol parsing
- **Inscriptions**: Browser for on-chain inscriptions with mime type and address filtering
- **Apps**: Health checks, SSL status, and latency monitoring for applications running on the bridge

Operator access is authenticated via a per-bridge `statusSecret` generated during initialisation.

8.3 Security

The API layer implements layered security:

- **Operator authentication**: Dashboard access requires the bridge's `statusSecret`.
- **Bridge-to-bridge authentication**: Peer bridges authenticate via the cryptographic handshake (Section 7.5), not API keys. A bridge proves its identity by signing nonces with its BSV private key.
- **Rate limiting**: Unauthenticated requests are rate-limited.
- **CORS**: Configurable origin whitelisting for browser-based access.

---

9. Supervision & Self-Healing

9.1 Purpose

The Bitcoin whitepaper addresses crash tolerance directly. 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:

> "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."

The supervision layer implements this principle for the relay mesh.

9.2 The Promise Double-Reject Problem

During a production audit, multiple bridges crashed simultaneously. The root cause was a class of bug present across multiple 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 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 fix uses a settled flag 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'));
});
```

A process-level `unhandledRejection` handler provides a safety net for any unforeseen double-rejects, logging the error and preventing process termination.

9.3 Recovery After Restart

When a bridge restarts, it re-establishes all layers:

1. **P2P reconnection**: Re-establishes TCP connections to Bitcoin full nodes via peer discovery.
2. **Header chain re-synchronisation**: Requests headers from its last known height using `getheaders` with a block locator.
3. **Federation re-peering**: Reconnects to seed peers, resumes gossip announcements, and re-establishes authenticated WebSocket connections.
4. **Peer scoring reset**: Peers receive a default stake age on reconnection; scoring resumes with fresh uptime and latency measurements.

---

10. Transaction Lifecycle

10.1 Client Initiation

A client application – whether a web application, CLI tool, or API integration – constructs a transaction, signs it using `@bsv/sdk`, and submits the raw transaction hex to a bridge's `/broadcast` endpoint.

10.2 Broadcast

The API layer passes the raw transaction hex to the SPV client's broadcast method. The SPV client computes the transaction identifier, stores the raw transaction, sends an `inv` message to all connected Bitcoin full nodes, and serves the full `tx` message when peers respond with `getdata`.

The federation layer simultaneously relays the transaction to all connected peer bridges via WebSocket, ensuring mesh-wide propagation.

10.3 Client Lookup

When a client requests a transaction, the bridge follows a deterministic lookup chain:

1. **Local store**: LevelDB lookup by txid.
2. **P2P `getdata`**: A `getdata MSG_TX` request to connected Bitcoin full nodes. `notfound` responses are handled immediately.
3. **Peer bridges**: WebSocket relay from peer bridges that may have the transaction.
4. **404**: If no source produces the transaction.

10.4 Confirmation

The bridge's periodic header synchronisation detects new blocks. 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.

Every transaction the bridge sees is tracked through a lifecycle state machine with four states: `mempool` (seen but unconfirmed), `confirmed` (proven in best chain with Merkle proof), `orphaned` (was confirmed but block disconnected by reorg), and `dropped` (mempool expiry after 14 days). The `txStatus` sublevel stores the authoritative state; the `txBlock` sublevel stores block placement with a reverse index (`block!<blockHash>!tx!<txid>`) that enables efficient rollback during chain reorganisations.

When a reorg occurs, the bridge identifies disconnected blocks via the reverse index, marks all affected transactions as orphaned, and re-enqueues them for confirmation against the new best chain. All rollback operations execute in a single atomic LevelDB `batch()` write to prevent inconsistent state.

Clients query confirmation state via `GET /tx/:txid/status` and retrieve Merkle proofs via `GET /proof/:txid`.

---

11. Security Considerations

11.1 Authentication and Access Control

The system employs layered authentication:

- **Operator secret**: Per-bridge authentication for dashboard access.
- **Cryptographic handshake**: Bridge-to-bridge authentication via mutual nonce signing (Section 7.5).
- **Rate limiting**: Per-IP throttling for unauthenticated requests.

11.2 Data Encryption

Applications using the relay mesh can encrypt payloads before broadcast. The relay mesh itself is transport-agnostic – it relays transactions without inspecting their content.

11.3 Loop Prevention

Gossip announcements include a deduplication mechanism: each announcement is identified by `pubkeyHex:timestamp`, and seen announcements are stored in a dedup set. Bridges do not re-broadcast announcements they have already seen, preventing infinite propagation loops.

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. The system avoids this entirely by never using bloom filters. All transaction lookups use direct `getdata MSG_TX` with explicit transaction identifiers.

11.5 Protocol Compliance

The system maintains strict compliance with the Bitcoin P2P protocol:

- Protocol version 70016 with `protoconf` message 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.
- Silent handling of `authch` messages from mining nodes.
- Relay flag set to 0 (SPV client does not relay).

11.6 Stake Bond Anti-Sybil

Registering a bridge requires locking a minimum of 1,000,000 satoshis in a stake bond. This economic cost prevents trivial mass registration of fake bridges. The stake bond transaction ID is included in the on-chain registration payload, making it publicly verifiable.

Stake age is factored into peer scoring (Section 11.8): longer-held stakes contribute to a higher trust score. This creates an ongoing cost for maintaining fake identities – the attacker's capital remains locked for the duration.

11.7 Peer Scoring

Every bridge scores its peers locally. There is no centralised reputation authority.

```
score = 0.3 * uptime + 0.2 * response_time + 0.4 * data_accuracy + 0.1 *
stake_age
```

Factor	Weight	What It Measures	Normalisation
Uptime	0.3	Percentage reachable over rolling window	pongs / pings (1000-sample window)
Response time	0.2	Normalised inverse latency	1.0 at <= 100ms, 0.0 at >= 5000ms, linear between
Data accuracy	0.4	Percentage of relayed data that validates correctly	good / total (1000-sample window)
Stake age	0.1	How long the stake bond has existed	log2(days) / 10, capped at 1.0

Data accuracy is weighted highest (40%) because correct data is the primary function of a relay bridge. A bridge that relays invalid headers or transactions is worse than useless – it wastes bandwidth and can mislead clients.

Score thresholds: - Score < 0.3 ⟶ auto-disconnect - Score < 0.1 ⟶ 24-hour blacklist

Each bridge computes scores independently. No consensus is required. A bridge that behaves well from one peer's perspective but poorly from another's will have different scores on different bridges. This is by design – local scoring reflects local observations.

11.8 Pubkey Authentication

Bridge identity is tied to a BSV keypair generated during initialisation. The compressed public key is the bridge's permanent identity. The on-chain registration proves key ownership (the registration transaction is signed by the bridge's private key). The cryptographic handshake (Section 7.5) proves identity on every connection.

This prevents endpoint spoofing: an attacker cannot claim to be a registered bridge without possessing the corresponding private key.

11.9 Gossip Signature Verification

All gossip announcements are signed by the announcing bridge's private key. The signature covers the pubkey, endpoint, mesh ID, and timestamp. Receiving bridges verify the signature before accepting or propagating the announcement. This prevents impersonation – a bridge cannot announce itself as another bridge without possessing that bridge's private key.

Announcements older than 5 minutes or more than 30 seconds in the future are discarded, preventing replay attacks.

---

12. Protocol Parsing

The relay mesh parses BSV transactions beyond simple P2PKH. Every output is typed, protocol-detected, and returned as structured data via the API and dashboard.

12.1 Supported Output Types

Type	Detection	Parsed Fields
`p2pkh`	`76a914{20 bytes}88ac`	`hash160`
`op_return`	`6a...` or `006a...`	`data[]`, `protocol`, `parsed`
`ordinal`	Contains `0063036f7264`	`contentType`, `content`, `isBsv20`, `bsv20`
`p2sh`	`a914{20 bytes}87`	`scriptHash`
`multisig`	Ends with `ae`	`m`, `n`, `pubkeys[]`

12.2 Supported Protocols (inside OP_RETURN)

Protocol	Prefix Address	Parsed Fields
B://	`19HxigV4QyBv3tHpQVcUEQyq1pzZVdoAut`	`data`, `mimeType`, `encoding`, `filename`
BCAT	`15DHFxWZJT58f9nhyGnsRBqrgwK4W6h4Up`	`info`, `mimeType`, `charset`, `filename`, `chunkTxids[]`
BCAT-part	`1ChDHzdd1H4wSjgGMHyndZm6qxEDGjqpJL`	`data`
MAP	`1PuQa7K62MiKCtssSLKy1kh56WWU7MtUR5`	`action`, `pairs: { key: value }`
MetaNet	Magic bytes `6d657461`	`nodeAddress`, `parentTxid`
BSV-20	Ordinal with `application/bsv-20`	`bsv20: { p, op, tick/id, amt, ... }`

This parsing enables the dashboard to display protocol badges and decoded data for every transaction, transforming raw hex into human-readable information.

Additionally, every parsed output now includes a `scriptHash` field – the SHA256 hash of the output's locking script hex. This provides a universal owner identifier that works for P2PKH, P2PK, P2SH, bare scripts, and any locking script type, unlike address-based keying which only works for standard P2PKH outputs.

---

13. Indexing Layer

The indexing layer transforms the bridge from a stateless relay into a self-verifying data source. It stores transaction confirmation state, Merkle proofs, BSV-20 token balances, and inscription content – all anchored to the header chain for reorg safety.

13.1 Transaction Confirmation Tracking

Every transaction seen by the bridge is tracked through a lifecycle state machine (Section 10.4). The `txStatus` sublevel stores the authoritative state for each transaction. The `txBlock` sublevel stores block placement with Merkle proofs and a reverse index for efficient reorg rollback.

13.2 Content-Addressed Inscription Storage

Inscription content is stored in a content-addressed system keyed by SHA256 hash. Small payloads (< 4 KB) are stored inline in LevelDB; larger payloads are written to the filesystem at `data/content/<first2chars>/<hash>`. The inscription record stores only metadata (content hash, length, MIME type, location pointer), reducing LevelDB compaction pressure and enabling deduplication when the same content appears in multiple transactions.

Content is served via `GET /inscription/:txid/:vout/content` with immutable cache headers (`Cache-Control: public, max-age=31536000, immutable`). The serving path resolves content through the CAS layer, falling back to re-extraction from the raw transaction if the CAS entry is missing.

13.3 BSV-20 Token Indexing

The bridge indexes BSV-20 deploy and mint operations. Token indexing operates on **confirmed transactions only** – mempool token operations are excluded to prevent double-spend corruption of balances.

Token owners are identified by script hash (SHA256 of locking script hex) rather than address, supporting all output types. The state machine enforces BSV-20 rules: first deploy wins (chain-ordered by block height), tick normalisation to lowercase, per-mint amount limits, and supply cap enforcement via BigInt arithmetic.

Every token operation is applied as a single atomic LevelDB `batch()` write: token record update, balance credit, operation log entry, and idempotency marker. Operation keys use zero-padded heights for lexicographic ordering, ensuring deterministic replay. Idempotency markers (`applied!<txid>`) prevent duplicate processing across restarts and backfill reruns.

13.4 Historical Backfill

The `relay-bridge backfill` CLI command walks historical blocks from a configured start height to the chain tip. For each block, it fetches the transaction list from WhatsOnChain (one API call per block), then selectively fetches raw transaction data only for transactions matching interest filters (ordinal inscriptions, BSV-20 operations). This avoids fetching entire blocks while still indexing relevant historical data.

Backfill supports resume via a stored `meta.backfill_height` value. Rate limiting (350ms between API calls) prevents upstream throttling. Progress is logged every 100 blocks.

13.5 Price Feed

The bridge provides a live BSV/USD exchange rate via `GET /price`, cached in memory with a 60-second TTL. The price is sourced from WhatsOnChain's exchange rate endpoint. This enables applications to display fiat-denominated values without maintaining their own price feed infrastructure.

---

14. Performance

14.1 Production Deployment

The initial deployment comprises two federation bridge nodes:

Node	Location	RAM	BSV Peers	Header Height	Peer Score
bridge-alpha	Dallas	1GB	230-267	939,901	0.92
bridge-beta	New Jersey	1GB	230-267	939,901	0.92

Both bridges are registered on-chain on mesh `70016` with 1,000,000-satoshi stake bonds. The on-chain registry enables additional bridges to join the mesh without coordination – register, stake, connect.

14.2 Broadcast Performance

Transaction broadcasts reach 250+ Bitcoin full nodes simultaneously via the P2P layer, covering the vast majority of the reachable BSV network. Inter-bridge relay via WebSocket adds sub-second mesh-wide propagation.

14.3 Header Synchronisation

Initial synchronisation of the full 939,000+ block header chain takes approximately 15-30 minutes. Subsequent synchronisation completes in under one second per new block.

14.4 Federation Overhead

Operation	Cost
CBOR registration tx	~300 bytes OP_RETURN + 100 sats dust
Stake bond	1,000,000 sats (locked to operator's own address)
Cryptographic handshake	3 messages, <100ms
Gossip announcement	~200 bytes, every 60 seconds
Peer scoring	Computed locally, zero network cost

---

15. Related Work

15.1 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. Section 7 describes the Merkle tree structure. Section 2 establishes the transaction model. Section 12 [1] establishes crash tolerance.

15.2 Overlay Network Architectures

Wright [3] describes overlay networks built atop the Bitcoin peer-to-peer layer, with attention to structured routing (Pastry DHT), reputation scoring, and eclipse attack resistance. The federation layer described in this paper implements several of the mechanisms Wright described: reputation scoring (peer scorer), node admission (on-chain registry with stake bonds), and cryptographic identity verification (handshake protocol).

15.3 Electrum and Similar SPV Systems

Electrum-style SPV systems use a client-server model where dedicated indexing servers serve SPV clients over a custom protocol. While effective, this model introduces a trusted third party. The relay mesh eliminates this dependency – bridges connect directly to Bitcoin full nodes and maintain their own header chains.

15.4 Centralised Node Providers

Ethereum has Infura and Alchemy. Solana has Helius and QuickNode. These services provide API access to blockchain networks without running a full node. They solve the same problem the relay mesh solves – but through centralisation. One company controls access, pricing, and uptime. The relay mesh achieves the same convenience with decentralised infrastructure. Every developer runs their own bridge.

---

16. Future Work

16.1 Merkle Tree Pruning

Section 7 of the Bitcoin whitepaper [1] describes reclaiming disk space through Merkle tree pruning. Implementing this would allow long-running bridges to discard old transaction data while retaining verification capability.

16.2 Advanced UTXO Management

Pre-splitting UTXOs into parallel chains would enable concurrent broadcast operations without contention, relevant for high-throughput applications.

16.3 Pastry DHT Routing

Replacing the current gossip-based discovery with DHT-based routing would reduce the O(n) message cost of announcement flooding to O(log n), enabling the mesh to scale to hundreds of bridges.

16.4 Process Supervision

Integration with `systemd` would provide automatic restart on crash, making the "rejoin the network at will" principle fully automatic.

16.5 Cross-Mesh Isolation

The `mesh_id` field in the registration schema supports multiple independent meshes. Cross-mesh peering policies and routing are deferred until demand materialises.

16.6 Index Service

When the number of registered bridges exceeds ~50, chain scanning for registry transactions may become slow. An optional index service could cache registry state for faster bootstrap.

16.7 BSV-20 Transfer Tracking

The current token indexing supports deploy and mint operations. Transfer tracking – following token ownership as inscriptions move between outputs – requires building a UTXO graph for ordinal positions. This is deferred to a future phase.

16.8 Full Block Fetching

Historical backfill currently relies on WhatsOnChain for block transaction lists. Adding `MSG_BLOCK` support behind a feature flag would enable fully self-sovereign backfill with no third-party dependency, at the cost of increased bandwidth and storage.

---

17. Conclusion

The federated SPV relay mesh transforms isolated SPV nodes into an open, self-governing network. Any BSV developer can run a bridge, register on-chain, and join the mesh – no permission required, no central authority, no API keys. The protocol uses BSV at every layer: identity (one keypair per bridge), registry (CBOR-encoded OP_RETURN transactions), security (stake bonds locked to the operator's own address), discovery (beacon address scanning and gossip), and trust (local peer scoring based on observed behaviour).

The system is deployed in production with two geographically distributed bridges serving the Indelible platform as a reference implementation. 383 tests verify the protocol implementation. The npm packages are published and install with a single command.

BSV has the technical capability – big blocks, cheap fees, native OP_RETURN, powerful script. What it has lacked is developer infrastructure. The relay mesh fills the gap between "run a full node" and "depend on someone else's API."

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

Relay Federation – Open BSV infrastructure, secured by proof of work.

https://github.com/zcoolz/relay-federation