# Prysm Transport

A Comprehensive Analysis of the Networking and Transport Architecture of the Prysm P2P Encrypted Messenger

`xmreur`

Version 2.0 — July 2026

**Abstract**

Prysm is a Tor-only peer-to-peer messenger. This document specifies the complete transport and networking architecture: the Tor hidden service substrate, the dual HTTP/WebSocket protocol stack, message routing and delivery semantics, peer connection management with automatic dialer election, the binary framing protocol for high-bandwidth channels, and the retry and backoff mechanisms that guarantee resilient communication over the Tor network. Every message is routed through a carefully designed transport selection layer that dynamically chooses between HTTP and WebSocket based on peer capability, connection state, and power constraints.

**Keywords:** Tor, WebSocket, HTTP, peer-to-peer networking, hidden service, SOCKS5, onion routing, transport selection

# Contents

# 1 Introduction

Prysm operates as a fully decentralized peer-to-peer network where every client is simultaneously a server. Communication occurs exclusively over the Tor network: each client runs a Tor hidden service that listens for incoming connections, and connects to peers' hidden services for outgoing communication. There is no central infrastructure, no DHT, and no rendezvous server.

The transport layer is responsible for reliable message delivery between peers identified by their `.onion` addresses. It provides two distinct transport mechanisms—HTTP and WebSocket—with an intelligent selection layer that chooses the optimal transport per peer based on capability discovery, connection state, power management, and message urgency. A persistent WebSocket layer enables real-time communication for messaging, typing indicators, file transfers, and voice calls.

# 2 Tor Infrastructure

## 2.1 Process Lifecycle

The Tor process is managed by `TorManager` (`lib/util/tor_service.dart`), a singleton that handles the complete lifecycle. On desktop platforms, it launches a Tor binary with a custom configuration:

Listing 1: Desktop Torrc Configuration

```
ControlPort 9051
SocksPort 9050
DataDirectory <dataDir>
HashedControlPassword <hash>
CookieAuthentication 0
HiddenServiceDir <dataDir>/hidden_service/
HiddenServicePort 80 127.0.0.1:8080
```

The hidden service maps port 80 to the local HTTP server on port 8080. The SOCKS proxy on port 9050 provides outbound connectivity. The control port on 9051 enables circuit management and health monitoring.

## 2.2 Lifecycle States

The Tor lifecycle is modeled as a state machine with four states:

$$\texttt{stopped} \rightarrow \texttt{restarting} \rightarrow \texttt{bootstrapping} \rightarrow \texttt{ready}$$

The `TorLifecycleNotifier` broadcasts state transitions. Outbound traffic is gated by `TorRuntimeGate`, which blocks all networking when Tor is not in the `ready` state.

## 2.3 Health Supervision

Desktop platforms employ a `TorSupervisor` that periodically evaluates health via the control port and SOCKS proxy:

- Bootstrap progress queried via `GETINFO status/bootstrap-phase`
- Network liveness via `GETINFO network-liveness`
- SOCKS port responsiveness tested by attempting a connection

After 2 consecutive health failures, Tor is automatically restarted with exponential backoff (minimum 60 s between restarts, maximum 3 restarts per 30-minute window). If the rate limit is exceeded, the system enters `needsAttention` mode requiring user intervention.

### 2.4 Circuit Management

Outbound traffic can trigger circuit rotation via `SIGNAL NEWNYM` on the Tor control port:

- Circuit errors detected during delivery attempts trigger `refreshCircuit()`
- A 1.5-second delay follows the NEWNYM signal to allow circuit establishment
- This mitigates exit-node-level failures and provides a degree of anonymity set rotation

### 2.5 Platform-Specific Adaptation

**Desktop (Linux/Windows/macOS)** : Native Tor binary launched as a child process; control via control port; PipeWire for audio capture on Linux.

**Android** : Delegates to the `prysm_tor` platform plugin, which manages the Tor daemon via the Orbot/Tor Android abstraction.

**iOS** : Uses the Tor framework via platform channel.

## 3 Local HTTP Server

The local HTTP server (`PrysmServer`) listens on `127.0.0.1:8080` and is the entry point for all Tor-routed inbound traffic. It is built on the `shelf` middleware framework.

### 3.1 Route Table

| Method | Path | Function |
|---|---|---|
| `GET` | `/ws` | WebSocket upgrade |
| `POST` | `/message` | Receive chat messages |
| `GET` | `/public` | Public identity JSON |
| `GET` | `/profile` | User profile (optional `?requester=`) |
| `POST` | `/sync-hint` | Wake/sync hints |
| Any other | | 404 |

### 3.2 WebSocket Upgrade Handler

When a peer initiates a WebSocket connection, the server upgrades at `/ws` and creates an `InboundWsPeerLink`:

**Algorithm 1** Inbound WebSocket Acceptance

**Require:** Incoming `WebSocketChannel` from Tor

1: Create `InboundWsPeerLink` wrapping the channel
2: Subscribe to channel stream
3: Await first frame — must be `hello` handshake
4: Extract `payload.onion` from handshake
5: Check for duplicate link via `WsConnectionManager`
6: Run `shouldDialPolicy(localOnion, peerOnion)`
7: **if** peer should be dialer **then**
8: Reject connection (ours will dial)
9: **else**
10: Send our own `hello` handshake
11: Register link with `WsConnectionManager.registerInboundLink()`
12: Begin dispatching post-handshake frames

### 3.3 Message Processing Pipeline

Inbound HTTP messages are processed in two phases:

1. **Validation** (synchronous): checks required fields (`id`, `senderId`, `receiverId`, `message`, `type`, `timestamp`), validates types, verifies group membership.

2. **Processing** (asynchronous): routes by message type to the appropriate service (chat, group control, read receipt, reaction, message modification), stores in database, notifies UI, shows notifications.

The server returns an optimistic ack immediately, then processes asynchronously. Error acks are sent only for validation failures.

## 4 Dual Transport Architecture

### 4.1 Abstract Transport Interface

Both HTTP and WebSocket transports implement the `OutboundTransport` interface:

**Algorithm 2** `OutboundTransport` Interface

1: `outboundQueueDepth` — pending messages count
2: `lastSuccessForPeer(onion)` — timestamp of last success
3: `runForPeer(onion, operation, maxAttempts)` — retry wrapper
4: `getProfile(onion)` — fetch peer profile
5: `getPublic(onion)` — fetch peer public key
6: `postMessage(onion, payload)` — send message
7: `postJson(onion, path, payload)` — arbitrary JSON

### 4.2 Transport Selection (`TransportProvider`)

The `TransportProvider` singleton is the central routing hub. It maintains instances of both `TorHttpTransport` and `TorWebSocketTransport` (which wraps `WsConnectionManager`) and selects the optimal transport per operation using the `withPeer()` method.

#### 4.2.1 Transport Preferences

| Preference | Behavior |
|---|---|
| `wsPreferred` | Try WS connect (bounded 25 s budget); fall back to HTTP |
| `wsIfConnected` | Use WS only if already connected; HTTP immediately |
| `httpOnly` | Always HTTP — for wake hints, low-priority nudges |

### 4.2.2 Routing Algorithm

**Algorithm 3** `withPeer()` Routing Logic

**Require:** peerOnion, operation, preference

**if** preference == httpOnly **then**
    **return** HTTP transport
**else if** preference == wsPreferred **and** not connected **then**
    Attempt WS connect with 25 s budget
    **if** connect succeeds **then**
        **return** WS transport
**else if** preference == wsIfConnected **and** not connected **then**
    Schedule WS recovery for later
    **return** HTTP transport
**if** WS connected **then**
    Execute operation via WS transport
    **if** WS fails **then**
        Log error, optionally disconnect peer
        Fall through to HTTP
**return** HTTP transport

## 4.3 HTTP Transport (`TorHttpTransport`)

The HTTP transport creates a fresh `TorHttpClient` per request, configured to route through the Tor SOCKS5 proxy at `127.0.0.1:socksPort`. Requests target the peer's hidden service at `http://<peerOnion>:80/<path>`. All requests use `TorDelivery.withTorRetry()` which provides:

- Up to 3 retry attempts
- Exponential backoff: 800 ms, 2 s, 3 s
- Circuit error detection triggering `SIGNAL NEWNYM` with 1.5 s delay
- Retry on: `TimeoutException`, connection refused, connection reset, handshake failures, SOCKS errors

## 4.4 WebSocket Transport (`TorWebSocketTransport`)

Delegates directly to `WsConnectionManager`, which manages persistent connections. Operations are serialized per-peer via chained futures to maintain ordering.

# 5 WebSocket Protocol

## 5.1 Frame Format

All WebSocket messages follow the `WsFrame` structure:

$$\texttt{WsFrame} = \{\texttt{v} : 1, \quad \texttt{op} : \texttt{string}, \quad \texttt{id} : \texttt{string}\ \text{(optional)}, \quad \texttt{payload} : \{\}\ \text{(optional)}\}$$

### 5.2 Supported Operations

| Operation | Purpose |
|---|---|
| `message` | Chat message delivery |
| `read_update` | Read receipt |
| `reaction_update` | Emoji reaction |
| `message_modify` | Message edit |
| `typing_update` | Typing indicator |
| `sync-hint` | Wake/nudge signal |
| `call_offer` | Call initiation |
| `call_answer` | Call acceptance |
| `call_end` | Call termination |
| `call_mute` | Mute toggle |
| `file_transfer_begin` | File transfer start |
| `file_transfer_end` | File transfer complete |
| `file_transfer_begin_ack` | File transfer ack |
| `file_transfer_end_ack` | Transfer completion ack |
| `file_transfer_chunk_ack` | Per-chunk ack |
| `profile` | Profile request |
| `public` | Public key request |

### 5.3 Handshake Protocol

Connection establishment follows a two-message handshake:

1. Dialer sends `hello` frame: `{op: "hello", payload: {onion: "<ourOnion>"}}`
2. Acceptor responds with `hello` frame containing its own onion address
3. Both sides register the link and begin exchanging operational frames

### 5.4 Heartbeat Mechanism

Connected peers exchange `ping` / `pong` frames at intervals determined by `BatterySaverPolicy`. After 3 consecutive unanswered pings, the peer is disconnected (unless pinned).

### 5.5 Request/Response Pattern

Operations that require a response (e.g., `profile`, `public`) include a UUID `id` field. The responder sends a `response` frame with the matching `id`. Pending requests are tracked in a `Map<String, Completer>` keyed by UUID, with configurable timeouts.

## 6 WebSocket Connection Management

### 6.1 Link Architecture

The system maintains one full-duplex WebSocket link per peer. Two implementations exist:

**`OutboundWsPeerLink`** : Wraps `TorWebSocketClient` (the dialer side). Connects via Tor SOCKS, performs WebSocket upgrade, manages hello handshake.

**`InboundWsPeerLink`** : Wraps a `WebSocketChannel` accepted by the HTTP server (the acceptor side). Handles hello handshake verification, duplicate detection, and dial policy enforcement.

### 6.2 Connection Manager (`WsConnectionManager`)

A singleton managing all peer links with the following state per peer:

- `WsPeerLink` — the active connection
- Serialized connect attempt chains (`_connectChains`)
- Serialized request queues (`_requestChains`)
- Failure counters for exponential backoff
- Last-success timestamps
- Inbound-waiter completers for peer-expected connections
- Peer capability sets

### 6.3 Connection Lifecycle

**Algorithm 4** Connection Maintenance Cycle

```
Determine target peers from DB (recent conversations) + pinned set target peer
if connected then
    Send ping
    if 3 consecutive ping failures then
        Disconnect
else if should dial(localOnion, peerOnion) then
    ensureConnected() with exponential backoff

Remove stale links for non-target, non-pinned peers
```

### 6.4 Dialer Election

To prevent simultaneous connection attempts, the peer with the lexicographically smaller `.onion` address opens the outbound WebSocket:

$$\texttt{shouldDial}(L, P) \triangleq L.\texttt{compareTo}(P) < 0$$

If a peer receives an inbound connection but determines it should be the dialer, it rejects the connection and waits for its own outbound attempt.

### 6.5 Backoff Strategy

Connection failures use truncated exponential backoff:

$$t_{n+1} = \min(2 \cdot t_n,\ t_{\max}), \quad t_0 = 1 \text{ s}, \quad t_{\max} = 120 \text{ s}$$

The backoff timer resets on successful connection.

### 6.6 Binary Frame Routing

The `WsConnectionManager` exposes streams for binary frames:

- `binaryFramesFor(peerOnion)` — call audio frames
- `pushFramesFor(peerOnion)` — unsolicited JSON push frames

Binary frames are distinguished by magic byte: `0xA1` for call audio, `0xA2` for file transfer chunks.

# 7 Binary Framing Protocol

## 7.1 Call Audio Frame

Listing 2: CallAudioFrame Structure

```
Byte 0:       magic 0xA1
Bytes 1-4:    sessionId (uint32 big-endian)
Bytes 5-8:    seq (uint32 big-endian)
Bytes 9+:     encrypted audio payload (ChaCha20-Poly1305)
```

## 7.2 File Transfer Chunk Frame

Listing 3: FileTransferChunkFrame Structure

```
Byte 0:       magic 0xA2
Bytes 1-16:   transferId (UUID as 16 raw bytes)
Bytes 17-20:  chunkIndex (uint32 big-endian)
Bytes 21+:    encrypted chunk payload
```

# 8 File Transfer Protocol

Files larger than 1 MiB use chunked WebSocket transfer. The protocol proceeds as:

**Algorithm 5** Chunked File Transfer

**Require:** File metadata, encrypted chunks

1: Sender sends `file_transfer_begin`: transferId, metadata, wrappedKey, totalChunks
2: Receiver validates and replies `file_transfer_begin_ack`
3: **for** chunkIndex = 0 to totalChunks - 1 **do**
4: Sender sends binary `FileTransferChunkFrame`
5: Receiver decrypts chunk, sends `file_transfer_chunk_ack`
6: **if** no ack within timeout **then**
7: Retry chunk (max 3 attempts)
8: Sender sends `file_transfer_end`
9: Receiver reassembles, processes, sends `file_transfer_end_ack`

Default chunk size is 256 KiB. Transfers expire after 10 minutes of inactivity.

# 9 Inbound Frame Routing

## 9.1 Frame Router (`WsFrameRouter`)

Inbound WebSocket request frames are routed by operation type:

| Operation | Handler |
|---|---|
| `ping` | Respond with `pong` |
| Call ops | Dispatch to `CallSignalingNotifier` |
| `get_profile` | Build profile via `InboundMessageRouter` |
| `get_public` | Return public key JSON |
| `typing_update` | Dispatch to `TypingIndicatorNotifier` |
| `message`, side-channels | Validate, ack, async-process |
| `sync-hint` | Handle via `WakeHintService` |
| File transfer ops | Dispatch to `FileTransferHandler` |

### 9.2 Inbound Dispatcher (`WsInboundDispatcher`)

While `InboundWsPeerLink` processes frames on connections accepted by our server, `WsInboundDispatcher` processes push frames arriving on outbound connections (connections we initiated). Both converge on the same `InboundMessageRouter` for message processing.

## 10 Message Delivery Semantics

### 10.1 Outbound Flow

1. Application encrypts message (Double Ratchet or group key)
2. `insertMessage()` stores encrypted payload in `messages.db` with status `pending`
3. `PendingMessageDbHelper` queues in `pending_messages.db`
4. `TransportProvider.postMessageOrFallback()` selects transport:
   (a) Try WebSocket (with retry up to 3 attempts)
   (b) Fall back to HTTP via Tor SOCKS
   (c) If transport unconfigured, raw Tor HTTP
5. On peer ack, status updated to `sent`, pending row removed

### 10.2 Inbound Flow

1. Tor forwards request to local `PrysmServer` (HTTP or WS)
2. Message validated synchronously (field checks, type routing)
3. Optimistic ack returned to sender
4. Message processed asynchronously:
   - Decrypted, stored in `messages.db`
   - UI notified via stream
   - Notification shown if app is backgrounded
   - Auto-profile fetch for unknown senders

### 10.3 Side-Channel Messages

Read receipts, reactions, typing indicators, and message modifications bypass the full encrypt/decrypt pipeline and are sent as small JSON frames over WebSocket. They are delivered best-effort without persistent queuing.

## 11 Peer Discovery and Connectivity

Prysm has no global peer discovery mechanism. Peer identities are exchanged out of band:

- QR codes containing `prysm:v2:<onion>:<fingerprint>`
- Manual entry of `.onion` addresses
- Contact sharing between existing peers

Connection targets are determined from the local database: peers with recent conversations (queried from `messages.db`) plus explicitly pinned peers. The `WsConnectionManager` maintains these connections in the background, subject to power constraints (`BatterySaverPolicy.wakeHintMaxPeers`).

# 12 Security Considerations

## 12.1 Transport-Layer Security

- All traffic is routed through Tor, which provides onion encryption (TLS between each pair of Tor nodes)
- Hidden service connections are end-to-end encrypted at the Tor layer
- Application-layer encryption (Double Ratchet, AEAD) provides defense-in-depth

## 12.2 Denial of Service

- Inbound connections are validated via handshake before registration
- Duplicate connections are rejected
- Profile requests from blocked peers are redacted

## 12.3 Anonymity Considerations

- No IP addresses are ever exchanged or logged
- Circuit errors trigger NEWNYM to rotate the exit circuit
- Dialer election is deterministic (lexicographic) and reveals no additional information
- Timing side channels are partially mitigated via sync-hints and exponential backoff

# 13 Constants and Configuration

| Parameter | Value | Context |
|---|---|---|
| HTTP server port | 8080 | Local listening port |
| SOCKS proxy port | 9050 | Tor SOCKS5 |
| Control port | 9051 | Tor control |
| Max retry attempts | 3 | `TorDelivery` |
| Backoff sequence | 800 ms, 2 s, 3 s | HTTP retry |
| WS heartbeat interval | `BatterySaverPolicy` | Ping frequency |
| Max ping failures | 3 | Disconnect threshold |
| Connect backoff start | 1 s | Exponential backoff |
| Connect backoff max | 120 s | Backoff ceiling |
| Auto-restart max | 3 per 30 min | Supervisor rate limit |
| File chunk size | 256 KiB | Chunked transfer |
| File transfer timeout | 10 min | Inactivity expiry |
| Large file threshold | 1 MiB | Monolithic vs chunked |

SIGNED: `xmreur` DATE: July 2026