Prysm Architecture

A Complete System Architecture Document
of the Prysm P2P Encrypted Messenger

xmreur

Version 2.0 — July 2026

Abstract

This document presents the complete system architecture of Prysm, a Tor-only peer-to-peer encrypted messenger. It integrates the cryptographic, transport, database, and calling subsystems into a unified description of the application. We describe the layered architecture, module interactions, data flow across all subsystems, the startup and shutdown sequence, and the security properties that emerge from the composition of components. All protocol versions refer to v2 (current as of Prysm 0.3+).

Keywords: system architecture, peer-to-peer, Tor, end-to-end encryption, Flutter, layered design

Contents

1 System Overview

Prysm is a fully decentralized, Tor-only encrypted messenger built with Flutter for cross-platform deployment. Every node in the network runs identical software and functions as both client and server. There is no central infrastructure of any kind.

The architecture follows a layered design with five horizontal layers and three cross-cutting concerns:

Layer / Concern	Responsibility
Presentation	Flutter UI, state management, routing
Application	Chat logic, group management, file transfer
Cryptographic	Identity, ratchet, AEAD, key derivation
Transport	Tor, HTTP, WebSocket, routing, retry
Persistence	SQLite, blob store, secure storage
Tor Infrastructure	Process lifecycle, circuit management
Call System	Signaling, audio pipeline, Opus codec
Key Management	Lifecycle, prekey pool, migration

2 Layer Architecture

2.1 Presentation Layer

The presentation layer is a standard Flutter widget tree with the following structure:

- `MainScreen` with bottom navigation (Chats, Calls, Settings)
- Per-conversation screens for direct and group chats
- Call overlay, shown as a full-screen widget during active calls
- Crypto migration screen (shown once during v1 $\rightarrow$ v2 upgrade)
- Settings screens for appearance, privacy, relay configuration

State management uses `ChangeNotifier` with `ListenableBuilder` for reactive UI updates. Key state holders include `ChatListNotifier`, `CallManager`, `TorLifecycleNotifier`, and per-conversation message streams.

2.2 Application Layer

The application layer orchestrates business logic:

- **Chat Service**: Message compose, encrypt, send pipeline; inbound decrypt, store, notify
- **Group Service**: Member management, key distribution, control message processing
- **File Transfer Service**: Chunked upload/download, progress tracking
- **Notification Service**: Local notifications for inbound messages and calls
- **Settings Service**: Persistence and reactive access to user preferences

2.3 Cryptographic Layer

Detailed in the companion document *Prysm Cryptography*. Key components:

- `IdentityKeyPair`: Ed25519 + X25519 key pair with fingerprint binding
- `CryptoKdf`: Argon2id for passphrase KDF, HKDF-SHA256 for protocol KDF
- `CryptoAead`: AES-256-GCM and ChaCha20-Poly1305 wrappers
- `CryptoWire`: DH-AEAD message encryption with ephemeral X25519
- `RatchetService`: Double Ratchet protocol with X3DH session bootstrap
- `GroupCryptoV2`: Sender-key group encryption with Ed25519 signing

2.4 Transport Layer

Detailed in the companion document *Prysm Transport*. Key components:

- `TorManager`: Tor process lifecycle, hidden service, SOCKS proxy
- `TransportProvider`: Dual transport selection (HTTP / WebSocket)
- `WsConnectionManager`: Persistent WebSocket per peer
- `PrysmServer`: Local HTTP server for inbound Tor traffic
- `InboundMessageRouter`: Validation, dispatch, profile fetching

2.5 Persistence Layer

Detailed in the companion document *Prysm Database*. Key components:

- `chat_app.db`: Users, groups, sessions, preferences
- `messages.db`: Messages, reactions, read receipts
- `pending_messages.db`: Outbound delivery queue
- `message_blobs/`: Oversized payload files
- `CryptoKeyStore`: Secure storage for keys

3 Module Interaction Diagrams

3.1 Message Send Flow

Algorithm 1 Complete Outbound Message Pipeline

Require: plaintext, recipient onion
1: COMPOSE(*plaintext*)
2: ENCRYPT(*plaintext*, *recipient*) via RatchetService.encryptBytes()
3: PERSIST(*ciphertext*) via MessagesDb.insertMessage() (status: pending)
4: QUEUE(*ciphertext*) via PendingMessageDbHelper.insertPendingMessage()
5: DELIVER(*ciphertext*, *recipient*) via TransportProvider.postMessageOrFallback()
6: **if** ack received **then**
7: UPDATESTATUS(*sent*)
8: REMOVEPENDING()
9: **else**
10: RETRY() with exponential backoff

3.2 Message Receive Flow

Algorithm 2 Complete Inbound Message Pipeline

Require: ciphertext envelope, sender onion

1: VALIDATE(*envelope*) — check fields, type, sender
2: ACK() — return optimistic confirmation
3: FETCHPROFILE(*sender*) if unknown (async via Tor)
4: DECRYPT(*envelope*) via RatchetService.decryptBytes() or CryptoWire
5: PERSIST(*plaintext*) via MessagesDb.insertInboundMessage()
6: NOTIFY(*sender*, *preview*) — UI stream + notification
7: PROCESSSIDEEFFECTS() — typing, reactions, read receipts

3.3 Session Bootstrap Flow

Algorithm 3 X3DH + Double Ratchet Session Establishment

Require: Peer onion address

1: Fetch peer profile via TransportProvider (includes prekey bundle)
2: Verify prekey bundle signature against peer's Ed25519 public key
3: Generate ephemeral X25519 key pair (ek, epk)
4: Compute $Đ_1 \parallel Đ_2 \parallel Đ_3 \parallel Đ_4$ (X3DH shared secret)
5: Initialize *RatchetSession* as initiator with shared material
6: Encrypt first message with derived message key (counter 0)
7: Attach handshake data (epk, $usedOTPK$) to ciphertext envelope
8: Persist session to *session_state* table
9: Deliver ciphertext with handshake

4 Startup Sequence

The application startup follows a strict dependency order:

Algorithm 4 Application Startup Sequence

1: INITIALIZEFLUTTER() — engine, plugins, platform channels
2: LOADSETTINGS() from SharedPreferences
3: INITIALIZEDATABASES() — open chat_app.db, messages.db, pending_messages.db
4: RUNMIGRATIONS() — apply pending schema migrations
5: CHECKCRYPTOGENERATION()
6: **if** legacy v1 keys detected **then**
7: Show CryptoMigrationScreen
8: **else if** passphrase not set **then**
9: Show onboarding / key generation flow
10: **else**
11: STARTTOR() — launch Tor process, await bootstrap
12: STARTHTTPSERVER() — bind to 127.0.0.1:8080
13: UNLOCKIDENTITY(passphrase) — decrypt and load keys
14: INITIALIZETRANSPORT() — configure TransportProvider
15: STARTWSMANAGER() — begin connection maintenance cycle
16: LOADCONVERSATIONS() — build chat list from DB
17: SHOWMAINSCREEN()

5 Critical Paths

5.1 Authentication and Key Unlock

1. User enters passphrase
2. Argon2id KDF: 64 MiB memory, 3 iterations, 16-byte salt
3. AES-256-GCM decrypt encrypted identity JSON
4. Load `IdentityKeyPair` (Ed25519 + X25519) into memory
5. Load prekey bundle (signed prekey + one-time pool)
6. All crypto operations now available
7. On lock: clear in-memory keys; encrypted data remains on disk

5.2 Offline Message Queue

When a peer is unreachable:

1. Message stays in `pending_messages.db` with initial delivery attempt
2. `TorDelivery.withTorRetry()` performs up to 3 attempts with exponential backoff
3. Message remains in `pending_messages.db` indefinitely
4. On next `WsConnectionManager` maintenance cycle, delivery is re-attempted
5. Sync-hints sent to peer to trigger reconnection

5.3 Group Key Rotation

When group membership changes:

1. New epoch key generated via `GroupCryptoV2.generateGroupKey()`
2. Key wrapped per existing member via DH-AEAD: `CryptoWire.wrapKeyForPeer()`
3. Control payload distributed as `control-wrap-1` envelope
4. Sender index reset per member
5. Old epoch key discarded

6 Security Architecture

6.1 Defense in Depth

Prysm employs multiple layers of protection:

- **Transport**: Tor onion routing provides anonymity and transport encryption
- **End-to-end**: Double Ratchet with X3DH provides forward secrecy
- **At rest**: Argon2id-derived AES-256-GCM within OS secure storage
- **Authentication**: Ed25519 signatures on prekey bundles and group messages
- **Replay protection**: Monotonic counters in ratchet and sender-key protocols

6.2 Trust Model

Component	Trust Assumption
Tor network	Honest majority of directory authorities and relays
Device OS	Secure boot, verified boot, sandboxing
Dart VM	Memory safety, no buffer overflows
Crypto primitives	Platform-native implementations (BoringSSL, CommonCrypto)
User passphrase	Sufficient entropy (≥ 12 characters)

6.3 Threat Model Summary

Threat	Mitigation
Network eavesdropping	Tor + end-to-end encryption
Impersonation	Fingerprint verification (QR codes)
Key compromise	Forward secrecy, KCI resistance
Traffic analysis	Tor, uniform message sizes
Offline brute force	Argon2id (64 MiB)
Physical device theft	Passphrase + OS secure storage
Metadata exposure	Minimal metadata in protocol

7 Directory Structure

```
lib/
  crypto/           -- Cryptographic primitives and protocols
    ratchet/        -- Double Ratchet, X3DH, session store
  transport/        -- HTTP/WS transports, routing
  server/           -- Local HTTP server, inbound routing
  client/           -- Tor-aware networking clients
  services/         -- Application services
    call/           -- Call signaling, audio pipeline
  util/             -- Tor management, database helpers
  database/         -- SQLite schema and queries
  models/           -- Data models
  screens/          -- UI pages
    call/           -- Call overlay UI
  theme/            -- Visual theming
  ui/               -- Reusable UI components
```

8 Configuration Reference

Setting	Default	Description
HTTP port	8080	Local server port
SOCKS port	9050	Tor SOCKS5 proxy
Control port	9051	Tor control port
Data directory	App docs/prysm	All databases, blobs, Tor data
Unlock type	passphrase	pin (6 digits) or passphrase (≥ 12 chars)
Max connection peers	10	Concurrent WS connections
Offline retry	3 attempts	Per-message delivery retries

9 Future Work

- **Relay nodes**: Optional store-and-forward relays for always-on delivery
- **Multi-device**: Sync session state and message history across devices
- **DHT**: Decentralized peer discovery without out-of-band contact sharing

- **Formal verification**: Machine-checked proofs of protocol correctness
- **Post-quantum**: Hybrid key encapsulation (X25519Kyber) for transport