Symmetric Authenticated Tunnelling Protocol

SATP Technology Integration Guide

Revision: 1.0
Date: October 23, 2025

1 Introduction and Scope

Symmetric Authenticated Tunneling Protocol (SATP) is a post-quantum, symmetric-only tunneling and authentication protocol designed to replace certificate-based VPN/TLS stacks. It eliminates public-key dependencies by deriving all material from SHA-3 family primitives (cSHAKE/KMAC), a single stream/AEAD cipher (RCS-256 by default), and a cost-based KDF for optional passphrase auth. The result is sub-millisecond handshakes on constrained devices, deterministic provisioning, and certificate-free operations (see the SATP summary and performance table).

This guide mirrors the **DKTP Integration Guide** structure you supplied, but translates each step to SATP's symmetric key-tree model and API.

2 Protocol Overview

2.1 Simplex Handshake (symmetric-only)

SATP uses a staged **Connect → Exchange → Establish** flow. Packets carry an authenticated header (flag, sequence, UTC time, payload size). Replay, ordering, and integrity are enforced before AEAD decryption. Core phases and header fields are defined in satp.h and the spec:

- **Connect Request/Response.** Client sends identity/config + nonce; server returns server ID/config + server nonce; both compute session hashes.
- **Exchange Request/Response.** Client sends a secret token (AEAD-protected) derived from device session hash + embedded device key. Server verifies and derives its receive key.
- **Establish Request/Response/Verify.** Client encrypts its key identity, server echoes it back, client verifies, session becomes **Established**.

Packet flags and timing/sequence rules are standardized (e.g., Encrypted Message, Keep Alive, Error) in the spec's tables.

2.2 API Summary

Key integration surfaces:

- **Client connect (IPv4/IPv6):**
  satp_client_connect_ipv4(satp_device_key*, addr4, port, send_func, receive_cb)
  satp_client_connect_ipv6(satp_device_key*, addr6, port, send_func, receive_cb) — performs KEX, initializes cipher states & sequences.
- **Client close / error:**
  satp_client_connection_close(cns, error); satp_client_send_error(sock, error).
- **Server (multi-threaded):**
  satp_server_start_ipv4(const satp_server_key*, receive_cb, disconnect_cb, authentication_cb) and IPv6 twin; plus pause/resume/quit/broadcast.
- **Connections pool (internal server):** initialize, add, index, next, reset, clear, dispose, self-test.
- **KEX internals (normally hidden):** satp_kex_client_key_exchange(...), satp_kex_server_key_exchange(...), and KEX state structs.

2.3 Choosing Parameter Sets

- **Cipher/AEAD:** Default is **RCS-256 + KMAC** (SATP_USE_RCS_ENCRYPTION enabled). Undefine to use **AES-256/GCM**.
- **Tag/nonce sizes:** With RCS: MACTAG=32, NONCE=32; with AES-GCM: MACTAG=16, NONCE=16.
- **Packet/header sizing:** HEADER=21B, default MTU=1500, MESSAGE_SIZE=1024.
- **Keepalive/timing:** KEEPALIVE_TIMEOUT=300s; enforce a narrow time window in packet validation for stronger anti-replay.

3 Key Management and Provisioning

SATP organizes secrets into a **root (master)** → **server (branch)** → **device (tree)** → **session** hierarchy:

- **Root (domain) key:** Kroot, expiration, domain id (offline, rarely changed).
- **Server (branch) key:** Kbr, sid, stc, expiration.
- **Device key-tree:** 1024 one-time leaves by default (SATP_KEY_TREE_COUNT) bound to kid; one leaf is consumed per session.

Provisioning steps (DKTP-style adapted to SATP):

1. **Generate Root & Server keys** offline (root → branch derivation); serialize for secure storage. (See SATP header for serialization constants and sizes.)
2. **Generate Device key-trees** per device under target server branch; embed the device's kid and the first unused leaf. Erase each leaf after use; persist the incremented key index atomically. (Single-use keys underpin forward secrecy.)
3. **Optional passphrase factor:** Pre-compute hardened passphrase hashes for the device and store server-side; use server helpers to generate/verify in the authentication_callback.

Domain identity/config strings: Adjust your deployment config constants (e.g., per line-of-business) similarly to how DKTP exposes a domain identity string; SATP's SATP_CONFIG_SIZE defines the config string length.

4 Integration into Payment Networks

4.1 Architecture

- **Client:** POS/ATM devices hold a **device key-tree** and kid (and optional passphrase hash).
- **Server:** Payment gateway holds **server branch key** and stc, validates device kid and passphrase (if required) via authentication_callback.

Transport uses your existing sockets; callbacks mirror the DKTP guide's send/receive flow, but all SATP tunnel operations are symmetric. (The DKTP guide section this mirrors is 4.x).

4.2 Integration Steps

1. **Provisioning:** Manufacture-time derivation of device key-trees from the server branch; inject kid and the initial leaf.
2. **Client connect:** Call satp_client_connect_ipv4/ipv6(...) with your send/receive callbacks; KEX derives duplex channel keys and sequence counters.
3. **Transmit:** Build packet → encrypt/authenticate → send; on receive, validate header (flag/seq/time/len) then decrypt. (Header format and flags in spec.)
4. **Rotation:** Each session **consumes one key-tree leaf**; device erases it and increments kid. Server rejects replays/out-of-order kid.

4.3 Operational Considerations

- **Latency:** SATP handshake and data path are constant-time symmetric ops, yielding the low latencies noted in the summary.
- **HA / Load-balancing:** Use stickiness per device kid during KEX; post-establish, the server maps packet headers (seq/time/flag) to the right satp_connection_state. Manage capacity with the connections pool helpers.

5 Integration into Cloud Platforms

5.1 Use Cases

- **Service-mesh RPC** (replace mTLS): lower CPU and no certificates.
- **Tenant VPNs / inter-DC links** using SATP tunnels.
- **SaaS API protection** over SATP channels. (Parallels DKTP §5.1 items.)

5.2 Integration Steps

- **Key distribution service:** Issue branch/device keys and kid via your KMS/secret store, similar to DKTP's guidance, but supplying SATP device key-trees.
- **Sidecar pattern:** Sidecars initiate satp_client_connect_* to peers and pass plaintext to the app via UDS/shared memory.
- **High concurrency:** Use satp_connections_* to pre-size, add, index, and recycle connection states; run the included self-test in CI.
- **LB & routing:** Ensure packets from a given connection land on the same backend instance across KEX; after establishment, header fields plus instance IDs in your pool enable correct routing. (Mirrors DKTP §5.2.)

6 Integration into SCADA and Industrial Control

6.1 Deployment Architecture

- **Field devices** act as SATP clients; each has a device key-tree.
- **Control center** runs the SATP server with the matching branch key; optional passphrase factor per device group. (Mirrors DKTP §6.)

6.2 Integration Steps

- **Offline provisioning** of device key-trees; choose longer epochs in isolated networks. (Adapted from DKTP key-provisioning advice.)
- **Connection management:** Devices call satp_client_connect_ipv4/ipv6; control centers start the multi-threaded server and register callbacks (receive, disconnect, auth).
- **Fieldbus encapsulation:** Wrap Modbus/DNP3 frames in SATP packets; validate header (seq/time/flag) prior to decrypt to meet determinism.
- **Keepalive/watchdog:** Respect KEEPALIVE_TIMEOUT=300s; use authenticated keep-alive requests and drop on expired time window.

7 Integration into IoT Devices

7.1 Integration Guidelines

- **Footprint:** Symmetric-only SATP minimizes flash/RAM vs PQ-KEM tunnels; see summary table.
- **Compile-time options:** Keep RCS enabled (bigger tag/nonce) or select AES-GCM for FIPS pathways.
- **Key storage:** Store device key-trees and kid in secure flash/SE; erase consumed leaves immediately; persist index atomically.
- **Network stack:** Use non-blocking sockets; supply send_func/receive_callback exactly as in the client API.
- **Firmware updates:** Ship signed payloads inside SATP; the tunnel provides AEAD-level integrity in addition to update signatures (pattern mirrors DKTP §7.1).

8 Security Best Practices

- **Single-use keys:** Enforce one leaf per session; on reconnect, a fresh leaf must be used to preserve PFS.
- **Passphrase hardening:** When a human/passphrase factor is used, generate/verify with the server helpers and only store hardened hashes.
- **Replay/downgrade protection:** Strictly check header flag/seq/time/len before decryption; reject on time skew or sequence violations; use the spec’s flag registry.

- **Timeouts:** Enforce KEEPALIVE_TIMEOUT; treat keep-alive failure as connection loss; drop early on invalid time windows.
- **Capacity & isolation:** Size the connection pool to OS FD limits; monitor availability/full status via satp_connections_available()/full().

9 Quick Start (both roles)

Server (multi-threaded)

1. Load **server branch key** and register callbacks: satp_server_start_ipv4(&skey, on_receive, on_disconnect, on_auth) (IPv6 variant available).
2. Optionally: satp_server_broadcast(...), satp_server_pause/resume/quit() for lifecycle control.
3. Initialize the **connections pool** early: satp_connections_initialize(init_count, max); use *_add/next/index/reset during operation.

Client

1. Load **device key-tree** and current kid.
2. Connect: satp_client_connect_ipv4(&dkey, &addr, SATP_SERVER_PORT, send_func, receive_cb) (IPv6 variant available).
3. Send/receive using the packet API in your callbacks; close with satp_client_connection_close(cns, err).

10 Appendix: Packet & Flags (Field Reference)

- **Header = 21 bytes:** flag (1) | seq (8) | UTC (8) | len (4); message follows (≤ SATP_MESSAGE_MAX).
- **Flags:** Connect Request/Response, Encrypted Message, Auth Request/Response/Verify, Keep Alive, Session Established, Error, etc.

11 Conclusion

Adopting SATP delivers certificate-free, quantum-resilient tunnels with deterministic provisioning and dramatically lower resource budgets than asymmetric or PQ-KEM stacks. The integration path mirrors your DKTP rollout playbooks — but with simpler key hierarchy operations, single-use device leaves, and a smaller, symmetric-only codebase. For payments, cloud service meshes, SCADA, and IoT, the steps above are sufficient to bring up **connect** → **exchange** → **establish** flows and operate at scale using the provided server, client, KEX, and connection-pool APIs.