Quantum Secure Tunnelling Protocol

QSTP Technology Integration Guide

Revision: 1.0
Date: October 14, 2025

Introduction

The **Quantum Secure Tunnelling Protocol (QSTP)** is a post-quantum replacement for today's VPN/TLS tunnels. Developed by **Quantum-Resistant Cryptographic Solutions (QRCS)**, QSTP introduces a **three-party, root-anchored trust model** consisting of a **Root Domain Server (RDS)**, an **Application Server (QAS)** and a **Client**. The protocol replaces classical RSA/ECC with **post-quantum key-encapsulation mechanisms (KEMs)** such as **Kyber** or **McEliece**, digital signatures such as **Dilithium** or **SPHINCS+**, and a **wide-block symmetric cipher** (RCS) integrated with **cSHAKE/KMAC** for authenticated encryption. Each session derives unique keys from an authenticated certificate chain and nonces, timestamps and sequence numbers to protect against replay, tampering and downgrade attacks. QSTP is implemented with QRCS's MISRA-compliant QSC library and supports high-throughput, multi-threaded operation with a per-connection state <4 kB, making it suitable for enterprise VPNs, payment networks, SCADA/OT systems and IoT devices.

This guide provides practical instructions on integrating QSTP into real-world systems. It draws on the QSTP Executive Summary, Specification and open-source code to explain the protocol architecture, API, key management and integration patterns across payment networks, cloud services, industrial control systems and IoT devices.

Protocol Overview

Architecture and Trust Model

QSTP operates under a **one-way trust model** anchored by a **Root Domain Server (RDS)**. The RDS issues and signs certificates for application servers (QAS). Clients trust the RDS and verify the server's certificate chain during the handshake. The roles are:

- **Root Domain Server (RDS):** Acts as certificate authority and trust anchor. It generates a root signing key, signs server certificates, exports root certificates for distribution, and can revoke or renew certificates.

- **Application Server (QAS):** Hosts the secure tunnel. It possesses a **server signature key** and an **ephemeral KEM keypair**. During the handshake it signs its public KEM key, sends the certificate chain to the client and processes client key-exchange packets. The server's multi-threaded implementation supports hundreds of thousands of concurrent tunnels with <4 kB state per connection.
- **Client:** Connects to the server, verifies the certificate chain against the root certificate, encapsulates a shared secret using the server's public KEM key, and derives symmetric session keys. In QSTP there is no certificate chain from the client, only one-way trust towards the server.

Cryptographic Primitives

- **Asymmetric KEMs:** Kyber or McEliece provide IND-CCA key exchange resistant to quantum attacks. Macros in qstp.h map qstp_cipher_generate_keypair, qstp_cipher_encapsulate and qstp_cipher_decapsulate to the chosen KEM implementation.
- **Digital Signatures:** Dilithium or SPHINCS+ authenticate certificates and signed KEM public keys. Macros map qstp_signature_generate_keypair, qstp_signature_sign and qstp_signature_verify to the chosen signature scheme.
- **Symmetric Encryption:** RCS is a wide-block authenticated cipher using Rijndael rounds with a cSHAKE-derived key schedule and KMAC authentication. It encrypts data and authenticates packet headers in one pass. Functions qstp_encrypt_packet and qstp_decrypt_packet perform AEAD encryption and decryption on network packets.
- **Hash and KDF:** SHAKE/cSHAKE and KMAC derive session keys and compute packet MACs.

Handshake and Session Keys

QSTP's key exchange comprises multiple message types encoded by **packet flags** (qstp_flags), such as qstp_flag_connect_request, qstp_flag_connect_response, qstp_flag_exchange_request and qstp_flag_exchange_response. A typical client/server exchange proceeds as follows:

1. **Connect Request/Response:** The client sends a connect_request containing its supported protocol configuration. The server responds with a connect_response containing its certificate, algorithm identifiers and an authenticated challenge.
2. **Exchange Request/Response:** The client verifies the server certificate against the root certificate and sends an exchange_request encapsulating a shared secret using the

server's public KEM key. The server decapsulates the secret and responds with an exchange_response signed using its signature key. Both sides derive preliminary symmetric keys from the shared secret.

3. **Establish Request/Response:** Optional extension to confirm handshake completion and signal readiness for data transfer (qstp_flag_establish_request/response).

After the handshake, both ends derive **two symmetric channels** (transmit and receive) using SHAKE from the shared secret and certificate hashes. Sequence numbers (rxseq, txseq) and a **session cookie** (hash of certificates and configuration) ensure packets are tied to the negotiated configuration and cannot be replayed or downgraded. Optional ratcheting injects fresh entropy after 2^{24} packets.

Packet Structure and Flags

Network packets are described by the qstp_network_packet structure, containing a **flag**, **message length**, **sequence number**, **UTC timestamp**, and a pointer to the encrypted message. Packets use various flags to represent handshake messages, encrypted payloads (qstp_flag_encrypted_message), keep-alive requests/responses, errors and termination signals. The header is serialized/deserialized with qstp_packet_header_serialize() and qstp_packet_header_deserialize() and validated with qstp_header_validate().

Root-Anchored Certificates

Certificates are defined by qstp_root_certificate and qstp_server_certificate structures. A server certificate contains the issuer string, root and server serial numbers, expiration times, algorithm configuration and the public verification key. The root certificate holds the root public key and algorithm identifiers. Certificates are signed by the root signature key (qstp_root_signature_key) and can be revoked via qstp_root_certificate_revoke(). Clients verify the server certificate using the root certificate before accepting the server's KEM key.

QSTP API Summary

The QSTP implementation in the QSC library exposes a set of C functions for root-key management, server operation, client connection and packet processing. The key structures and functions are summarized below; refer to the full header files for complete definitions.

Root Server Functions (root.h)

Function	*Purpose*

qstp_root_key_generate(kset, issuer, exp)	Generate a new **root signing key** with issuer name and validity period (days).
qstp_root_certificate_export(root, path)	Export a root certificate to a file for distribution to clients and servers.
qstp_root_sign_certificate(path, root, rootkey)	Sign a child certificate (server or client) and write it to disk.
qstp_root_certificate_revoke(rootkey, serial, address)	Send a certificate revocation request to a target server address.
qstp_root_certificate_print(root) / qstp_root_server_certificate_print(cert)	Print certificate details for debugging.

Server Functions (server.h)

Function	***Purpose***
qstp_server_key_generate(kset, issuer, exp)	Generate a **server signature key** for a QAS; includes issuer string and expiration days.
qstp_server_start_ipv4(source, kset, recv_cb, disc_cb) / qstp_server_start_ipv6(...)	Launch a multi-threaded server listener on a socket; uses kset to sign its KEM public key and handle key exchanges; callbacks process incoming messages and disconnections.
qstp_server_pause() / qstp_server_resume() / qstp_server_quit()	Control server operation; pause/resume new connections or shut down server.
qstp_server_expiration_check(kset)	Check if a server key has expired.

Client Functions (client.h)

Function	***Purpose***
qstp_client_connect_ipv4(root, cert, address, port, send_func, recv_cb) *qstp_client_connect_ipv6(...)*	Connect to a QSTP server using IPv4/IPv6; authenticate server by verifying its certificate against root; perform key exchange; return error code indicating success or failure. The

	send_func handles outgoing message loops and the receive_callback processes incoming data streams.

Connection and Packet Functions (qstp.h)

Function	*Purpose*
qstp_connection_close(cns, err, notify)	Gracefully close a connection and optionally notify peer.
qstp_connection_state_dispose(cns)	Zero a connection state after use.
qstp_encrypt_packet(cns, packetout, message, msglen)	Encrypt a plaintext message into a QSTP network packet (includes header creation and MAC).
qstp_decrypt_packet(cns, message, msglen, packetin)	Decrypt a received packet and recover the plaintext.
qstp_header_create(packetout, flag, seq, msglen)	Fill packet header with flag, sequence and message length, and set creation time.
qstp_header_validate(cns, packetin, flag, seq, msglen)	Validate packet header, sequence and timestamp for replay protection.
qstp_packet_header_serialize(packet, buf) / qstp_packet_header_deserialize(buf, packet)	Convert a packet header to/from byte array.
qstp_error_to_string(err) / qstp_get_error_description(msg)	Map error codes or log messages to human-readable strings.

Data Structures and Enumerations

- **qstp_connection_state:** holds the socket, cipher states (rxcpr, txcpr), sequence numbers and flags for an active connection.
- **qstp_network_packet:** represents a network packet's header fields and points to the encrypted message buffer.
- **qstp_server_certificate / qstp_root_certificate:** hold signed certificate data, issuer, serial numbers, expiration time, algorithm configuration and public verification key.

- **qstp_flags:** enumerates packet types: connect, exchange, establish, encrypted message, keep-alive, connection terminate, error and others.
- **qstp_messages:** enumerates log messages for diagnostics.
- **qstp_configuration_sets:** enumerates supported cryptographic parameter sets; functions convert between strings and enums (qstp_configuration_from_string / qstp_configuration_to_string).

Key Management and Certificate Provisioning

Generating Root Keys and Certificates

1. **Generate Root Key:** Use qstp_root_key_generate(&rootkey, issuer, exp_days) to create a root signature key with a specified issuer name and validity period. Store the root key securely (e.g., within a Hardware Security Module) and note its expiration.
2. **Export Root Certificate:** Populate a qstp_root_certificate root; structure with the public root information and export it to a file using qstp_root_certificate_export(&root, path). Distribute this file to clients and servers. Clients use it as the trust anchor and servers include its serial number and hash when generating their certificates.
3. **Sign Server Certificates:** For each application server, generate a server signature key using qstp_server_key_generate(&serverkey, issuer, exp_days). Then call qstp_root_sign_certificate(server_cert_path, &root, rootkey) to sign the server certificate and embed the server's public verification key. Distribute the signed server certificate alongside the root certificate.
4. **Key Expiration and Revocation:** Implement certificate rotation policies. Use qstp_server_expiration_check() to test server key validity. If a certificate is compromised or expired, call qstp_root_certificate_revoke(rootkey, serial, server_address) to revoke it.

Generating Server Keys and Certificates

- **Server Signature Key:** Use qstp_server_key_generate() to generate the server's long-term signature key with issuer and expiration parameters.
- **Server KEM Keypair:** At runtime, the server generates an *ephemeral* KEM keypair (Kyber or McEliece) using qstp_cipher_generate_keypair() (mapped to qsc_kyber_generate_keypair). The server signs the public KEM key with its signature key and includes it in the handshake.

- **Server Certificate File:** The server certificate (signed by the root) contains the server's verification key (verkey) and other metadata. The server loads this certificate at startup and provides it to clients during the handshake.

Client Key Verification and Session Setup

- **Trust Anchor:** The client loads the root certificate from a trusted location and passes it to qstp_client_connect_ipv4() or qstp_client_connect_ipv6() as the root parameter. It also loads the server certificate file (path provided out of band) and passes it as cert.
- **Handshake Execution:** qstp_client_connect_ipv4() performs the handshake: verifying the server certificate signature chain, verifying the signed KEM key, encapsulating a shared secret using the server's KEM public key, and deriving symmetric keys. If successful, it returns qstp_error_none and yields a qstp_connection_state through the callback functions.
- **Callbacks:** Provide a send_func callback to manage the client's outgoing message queue and a receive_callback to process decrypted messages. On the server side, supply a receive_callback and disconnect_callback to qstp_server_start_ipv4().

Implementation Sequence

This section outlines a typical integration sequence for QSTP. Adjust the flow to suit your application architecture.

1. Initialize Root Infrastructure

1. **Generate root keys:** Use qstp_root_key_generate() to create the root signing key and export the root certificate with qstp_root_certificate_export().
2. **Generate server signature keys:** For each QAS instance, generate a server signature key using qstp_server_key_generate(). Sign the server certificate using qstp_root_sign_certificate().
3. **Distribute root certificate:** Provide clients with the root certificate file and ensure they verify server certificates against it.

2. Server Integration

1. **Load server certificate and key:** At startup, load the signed server certificate into a qstp_server_certificate structure and load the server signature key into qstp_server_signature_key.

2. **Start listener:** Create a listener socket (IPv4 or IPv6). Call qstp_server_start_ipv4(socket, &server_key, recv_cb, disc_cb) or the IPv6 variant to start the multi-threaded server. The listener accepts connections and spawns threads to handle each client.
3. **Handle key exchange:** When a new client connects, the server's callback functions use internal routines (see server.c) to perform the handshake: send connect response, decapsulate client secret, send signed exchange response and derive symmetric keys. The callback receives decrypted messages via recv_cb and processes keep-alive and termination events.
4. **Manage connections:** Use qstp_connections_size() and related functions (defined in connections.h) to manage active connection states. When a connection terminates, call qstp_connection_state_dispose() to reset resources.
5. **Pause/Resume/Shutdown:** Use qstp_server_pause(), qstp_server_resume() and qstp_server_quit() to control server availability.

3. Client Integration

1. **Load certificates:** Load the root certificate into a qstp_root_certificate structure and the server certificate into a qstp_server_certificate structure (e.g., read from disk and decode using qstp_root_certificate_decode(), details in the specification). Validate that the server certificate is signed by the root.
2. **Connect to server:** Create a network address structure (IPv4 or IPv6) and call qstp_client_connect_ipv4(root, cert, &address, port, send_func, receive_callback). The function returns a qstp_errors code indicating the connection result. On success, a secure tunnel is established and subsequent packets can be encrypted and decrypted.
3. **Send messages:** To send application data, call qstp_encrypt_packet(&conn_state, &packet, message, msglen) to produce a network packet then send it over the socket. The send_func callback can handle this loop using qsc_socket_send() and should assign the appropriate sequence number, flag (qstp_flag_encrypted_message) and handle retransmissions.
4. **Receive messages:** The receive_callback processes incoming packets. Use qstp_decrypt_packet(&conn_state, buffer, &msglen, &packet) to decrypt the payload and deliver plaintext to the application. Check header validity with qstp_header_validate() to ensure correct flag, sequence and timing.
5. **Keep-Alive and Termination:** Implement keep-alive messages by periodically sending a qstp_flag_keep_alive_request flag and awaiting the corresponding response. If a timeout

occurs, close the connection using qstp_connection_close(&conn_state, qstp_error_keepalive_expired, true). For clean shutdown, send a qstp_flag_connection_terminate and dispose state.

4. Packet Handling Pattern

Below is a high-level pseudocode pattern illustrating client message flow:

```
// Connect
err = qstp_client_connect_ipv4(&root_cert, &server_cert, &server_addr, QSTP_PORT,
                               send_func, receive_callback);
if (err != qstp_error_none) {
    // handle error
}


// Prepare a message
qstp_network_packet pkt;
qstp_header_create(&pkt, qstp_flag_encrypted_message, state.txseq++, msglen);
qstp_encrypt_packet(&state, &pkt, message, msglen);
qstp_packet_header_serialize(&pkt, buffer);
// send buffer + encrypted payload via socket


// In receive callback:
qstp_packet_header_deserialize(buffer, &pkt);
if (qstp_header_validate(&state, &pkt, expected_flag, expected_seq, pkt.msglen) ==
qstp_error_none) {
    qstp_decrypt_packet(&state, plaintext, &msglen, &pkt);
    // process plaintext
}
```

Integration Scenarios

Payment Networks and Financial Services

QSTP can replace TLS/IPsec in inter-bank channels and payment networks. Deploy a dedicated QSTP root authority within the network operations center to issue server certificates for payment gateways. Payment terminals connect via QSTP tunnels to the payment switch; HKDS or SKDP may be layered over QSTP to provide hierarchical key derivation for transaction keys. Ensure the

server's certificate includes regulatory information (PCI DSS) and configure the protocol using qstp_configuration_from_string("dilithium-kyber") to meet FIPS 203/204/202 compliance.

Cloud Platforms and Enterprise Services

For cloud-native micro-services, QSTP offers a certificate-authenticated tunnel analogous to a quantum-secure VPN. Root and server certificates can be managed via a private PKI integrated into the cloud provider's secret management. Use the multi-threaded server functions to run a QSTP service that fronts micro-services, then replace TLS termination with QSTP. Clients (e.g., mobile apps or service daemons) embed the root certificate and perform the handshake via IPv6. QSTP's small state and high throughput allow thousands of simultaneous connections per node.

Critical Infrastructure and SCADA/OT

Industrial control systems require deterministic, low-latency tunnels with strong authentication. QSTP's root-anchored design prevents rogue devices from connecting and its packet timestamps ensure replay protection. Deploy QSTP servers at SCADA masters and configure clients on remote RTUs/PLCs. Use qstp_server_pause() to suspend new connections during maintenance and qstp_server_resume() to resume. Implement keep-alive intervals that align with SCADA polling cycles; a 4 kB connection state fits within typical embedded controller memory budgets.

IoT and Edge Devices

Lightweight endpoints such as sensors and edge gateways can integrate QSTP thanks to its constant-time cryptography and minimal state. Provision the root certificate in firmware and obtain server certificates from a device-owner root. Use the IPv6 client API to connect to the gateway. Sequence numbers and timestamps embedded in packets enable out-of-order detection across unreliable networks.

Security and Operational Best Practices

1. **Certificate Lifecycle Management:** Enforce short validity periods and rotate root and server keys before expiration. Use revocation to remove compromised certificates.
2. **Downgrade Immunity:** Always hash protocol configuration strings into the session cookie; verify that both parties agreed on the same parameter set (qstp_configuration_sets).
3. **Replay Protection:** Validate packet timestamps and sequence numbers using qstp_header_validate() and discard packets outside the Δt window.

4. **Memory and Thread Safety:** Dispose of connection state with qstp_connection_state_dispose() after termination to avoid residual key material. Use the multi-threaded server API and asynchronous primitives in QSC (e.g., qsc_async_mutex_lock_ex()) to coordinate state across threads as demonstrated in the reference server code.
5. **Performance Tuning:** Choose cryptographic parameter sets based on performance requirements—Kyber/Dilithium provide fast operations, while McEliece/SPHINCS+ offer diversity. The configuration macros (e.g., QSTP_CONFIG_DILITHIUM_KYBER) set the default parameters.
6. **Interoperability:** QSTP can operate over IPv4 or IPv6; ensure consistent address representation across client and server. For integration with existing PKI, export certificates in standard DER/PEM formats and wrap QSTP key exchange within existing PKIX workflows.

Conclusion

The Quantum Secure Tunnelling Protocol offers a complete, production-ready replacement for TLS, IPsec and SSH, providing forward secrecy and quantum resistance through a root-anchored, certificate-authenticated design. Its C API integrates seamlessly with existing networking stacks, and the protocol's flexible configuration enables tuning for performance and compliance. By following the integration steps in this guide, generating keys and certificates, configuring server and client functions, managing packet headers and encryption, and adopting best practices; developers can deploy QSTP across payment networks, cloud services, industrial control systems and IoT devices. QSTP not only secures communications for today's threats but establishes a trust fabric resilient against future quantum adversaries.