Quantum Secure Messaging Protocol

QSMP Integration Guide

Revision: 1.0
Date: October 15, 2025

Introduction

The **Quantum Secure Messaging Protocol** (QSMP) is QRCS's answer to post-quantum-secure messaging and tunnelling. It offers two variants; a **Simplex** mode for one-way trust and a **Duplex** mode for mutual authentication, both of which integrate post-quantum key exchange, digital signatures and authenticated encryption into a single protocol. QSMP supports the NIST-approved **Kyber** or **McEliece** key encapsulation mechanisms (KEMs) and **Dilithium** or **SPHINCS+** digital signatures, while its traffic is protected by the **RCS** wide-block stream cipher with **KMAC** authentication. QSMP avoids bolting PQ primitives onto legacy protocols; instead, it provides a deterministic design with a fixed packet header, explicit configuration strings and no version negotiation. These qualities make QSMP an attractive drop-in replacement for SSH, TLS tunnels, VPN overlays or custom RPC channels in environments that demand long-term confidentiality.

This guide explains how to integrate QSMP into payment networks, cloud services, SCADA systems and IoT devices. It covers protocol structure, API functions, key management, handshake procedures and practical implementation considerations. Wherever possible, it references the official executive summary and specification to justify design choices.

1 Protocol Overview

1.1 Modes of Operation

QSMP defines **two mutually exclusive handshake variants** to accommodate different trust models:

1. **Simplex (Client-Server) Mode:** The client trusts the server. During the handshake the server signs its public KEM key, and the client verifies the signature using a pre-shared server verification key. A single shared secret is exchanged, from which symmetric keys and nonces for both directions are derived. The result is a **256-bit** post-quantum secure channel established in just two round trips. Simplex is ideal for client-server applications

requiring scale: the QSMP server holds <4 kB of state per client and can manage hundreds of thousands of concurrent sessions.

2. **Duplex (Mutual Trust) Mode:** Both hosts authenticate each other by exchanging and verifying signed KEM public keys. Each side encapsulates a separate secret, and the two secrets are combined to derive independent transmit and receive keys. This yields **512-bit** symmetric keys and provides explicit mutual authentication. Duplex is suited for peer-to-peer connections, SCADA equipment, industrial control and environments where both ends hold persistent identities and require strong post-quantum security guarantees.

1.2 Cryptographic Primitives

- **Asymmetric KEMs:** Kyber or McEliece with configurable parameter sets. Both are IND-CCA secure and provide quantum-resistant encapsulation.
- **Signatures:** Dilithium or SPHINCS+; the server signs its public KEM key in Simplex, and both parties sign in Duplex.
- **Symmetric encryption:** QSMP uses the **RCS** cipher, a wide-block Rijndael variant with extra rounds and a cryptographically strong key schedule (cSHAKE). RCS operates as an authenticated stream cipher, using **KMAC** (or QMAC) as the MAC. The packet header is added to the additional authenticated data so that tampering or replay is detected.
- **Hash/KDF:** SHA3, SHAKE and KMAC are used throughout. A *session cookie* (sch) is computed by hashing the configuration string, key identifier and both parties' verification keys. The cookie binds the handshake to a unique context and is fed into the KDF to derive session keys.

1.3 Packet Structure and Header

Every QSMP packet starts with a **21-byte header:**

Field	Size (bytes)	Description
Flag	1	Indicates packet type (connect request/response, exchange request/response, establish verify, encrypted message, keep-alive, ratchet, etc.).
Message length	4	Length of payload in bytes.
Sequence number	8	Monotonic counter used to detect reordering and closing (0xFFFFFFFFUL terminates connection).

UTC time	8	Seconds since epoch when packet created. Used to reject stale/replayed packets (default time window is 60 s).

This header is serialized with the packet body using qsmp_packet_header_serialize() and deserialized on receipt via qsmp_packet_header_deserialize(). The function qsmp_packet_time_valid() checks whether the packet's timestamp is within the allowable window. When encrypting, the header bytes are passed as additional data to KMAC in the RCS AEAD so any change invalidates the MAC.

1.4 Key and State Objects

QSMP provides C structures to encapsulate persistent keys and per-session state:

- **qsmp_server_signature_key:** Holds the server's private signing key (sigkey), corresponding verification key (verkey), a 48-byte configuration string (config), 16-byte key ID (keyid) and expiration time. This structure is used to sign KEM public keys in both modes.
- **qsmp_client_verification_key:** Contains the client's public verification key (verkey), configuration string, key ID and expiration time. Clients embed the server's verification key and their own to compute session cookies.
- **qsmp_network_packet:** Represents a packet with fields flag, msglen, sequence, utctime and pointer pmessage.
- **qsmp_connection_state:** Maintains per-connection state: transmit and receive cipher contexts (txcpr/rxcpr), transmit/receive sequence numbers (txseq/rxseq), ratchet seed (rtcs), socket descriptor and flags for mode and role. This object is passed to most API calls.

2 API Overview

This section summarizes the exported functions defined in qsmp.h, client.h and server.h. They allow applications to generate keys, configure connections, perform handshakes and exchange encrypted messages.

2.1 Key Management

Function	*Purpose*
qsmp_generate_keypair(pubkey, prikey, keyid)	Generates a new server signature key pair (prikey of type qsmp_server_signature_key and pubkey of type qsmp_client_verification_key).

	The 16-byte keyid uniquely identifies the key to peers.
qsmp_signature_key_serialize(serk, prikey) / qsmp_signature_key_deserialize(prikey, serk)	Convert a secret signing key structure to/from a byte array for storage or transmission.
qsmp_public_key_encode(enck, enclen, pubkey) / qsmp_public_key_decode(pubkey, enck, enclen)	Encode/decode a public verification key to/from a string.
qsmp_public_key_compare(a, b)	Compare two public keys for equality.
qsmp_public_key_encoding_size()	Returns the size of an encoded key.

2.2 Client Functions (client.h)

To initiate or accept connections, applications use the client API:

- **Simplex connections:**
  - qsmp_client_simplex_connect_ipv4(pubk, address, port, send_func, receive_callback) – Connect to a server over IPv4. pubk is the server's verification key; send_func transmits a packet and receive_callback handles inbound packets.
  - qsmp_client_simplex_connect_ipv6(...) – Same as above but for IPv6.
  - qsmp_client_simplex_listen_ipv4(prikey, send_func, receive_callback) / qsmp_client_simplex_listen_ipv6(...) – Start a Simplex server (the host that clients connect to) using the server signature key.
- **Duplex connections:**
  - qsmp_client_duplex_connect_ipv4(kset, rverkey, address, port, send_func, receive_callback) – Connect to a peer in Duplex mode. kset is the local private signature key; rverkey is the remote party's verification key; callbacks handle I/O.
  - qsmp_client_duplex_connect_ipv6(...) – Same for IPv6.
  - qsmp_client_duplex_listen_ipv4(kset, send_func, receive_callback, key_query) / qsmp_client_duplex_listen_ipv6(...) – Accept a single host-to-host connection in Duplex mode. key_query is a callback that receives a key ID from the remote and returns the associated verification key.

2.3 Server Functions (server.h)

Applications requiring a multi-threaded server that handles many QSMP clients use functions declared in server.h:

- **qsmp_server_start_ipv4(source_socket, prikey, receive_callback, disconnect_callback):** Start a multi-threaded server on an IPv4 socket. source_socket should be bound and listening. The function spawns threads to accept clients; each thread uses receive_callback to process incoming messages and disconnect_callback to free resources when a client disconnects.
- **qsmp_server_start_ipv6(...):** Same for IPv6.
- **qsmp_server_broadcast(message, msglen):** Broadcast a message to all connected clients.
- **qsmp_server_pause()**, **qsmp_server_resume()**, **qsmp_server_quit():** Control the server's run state.

2.4 Packet Processing

The following functions operate on qsmp_network_packet and qsmp_connection_state structures:

- **qsmp_header_create(packet, flag, sequence, msglen):** Populate a packet header and set its timestamp.
- **qsmp_header_validate(cns, packet, kexflag, pktflag, sequence, msglen):** Validate a packet's header and timestamp, checking the expected flag, sequence and length.
- **qsmp_packet_encrypt(cns, packetout, message, msglen):** Encrypt a plaintext message and place it in a packet.
- **qsmp_packet_decrypt(cns, message, msglen, packetin):** Decrypt a packet and return the plaintext.
- **qsmp_packet_error_message(packet, error):** Populate a packet with an error code.
- **qsmp_packet_clear(packet):** Zero a packet's memory.
- **qsmp_packet_header_serialize(packet, header)** and **qsmp_packet_header_deserialize(header, packet):** Convert between a header structure and a 21-byte array.
- **qsmp_packet_time_valid(packet):** Verify the packet's timestamp falls within the configured time window.

2.5 Ratchet Operations

QSMP supports optional ratcheting of session keys to enhance forward secrecy after the connection has been established. When compiled with QSMP_ASYMMETRIC_RATCHET defined, the client can request an **asymmetric key ratchet**; otherwise, a symmetric ratchet is available in all builds.

qsmp_duplex_send_symmetric_ratchet_request(cns): In Duplex mode, send a request to update the symmetric keys derived from the existing ratchet seed. The remote party responds automatically.

3 Key Exchange Workflow

3.1 Simplex Handshake Sequence

The Simplex protocol uses a **client-server** model. The server holds a signature key pair and distributes its verification key to clients. The handshake proceeds through three message pairs:

1. **Connect Request:** The client sends a packet containing the server's key ID and the configuration string (cfg), then signs a hash of these values along with the serialized header (sh). The signature shm authenticates the request. The session cookie sch = H(cfg || kid || pvk) is computed and stored. The server verifies the signature using the client's verification key (embedded in the request).
2. **Connect Response:** The server validates the packet header and ensures the timestamp and sequence are correct. It then generates a fresh KEM key pair (using Kyber or McEliece), hashes the public key with the header, signs this hash using its signature key, and returns the public KEM key and the signature to the client. The server computes its own session cookie in the same way.
3. **Exchange Request:** Upon receiving the server's response, the client verifies the signature, encapsulates a shared secret sec using the server's KEM public key and the configuration string, derives two symmetric keys (for tx and rx) by combining sec with the session cookie via SHAKE, and sends the ciphertext to the server.
4. **Exchange Response:** The server decapsulates the shared secret from the ciphertext, combines it with its session cookie to derive the same symmetric keys and nonces, and initializes the RCS cipher contexts. The tunnel is now active. Both parties verify the tunnel in an **Establish Verify** message; any error leads to a session teardown.

The Simplex handshake achieves a two-way encrypted channel in just two round trips while maintaining 256-bit post-quantum security.

3.2 Duplex Handshake Sequence

In Duplex mode, both ends possess signature keys and verification keys. A typical sequence is:

1. **Connect Request:** The initiator (client) sends its configuration string and key ID. It computes a session cookie sch = H(cfg || kid || pvka || pvkb) using both verification keys.
2. **Connect Response:** The responder (server) validates the header and signature, computes the same sch, generates its own KEM key pair and signs the hash of the public key and header. It sends the public key and signature back to the initiator.
3. **Exchange Request:** The initiator verifies the signature, generates a second KEM key pair, encapsulates a secret to the responder's KEM public key and signs the hash. It sends its public key, signature and the ciphertext.
4. **Exchange Response:** The responder verifies the initiator's signature, encapsulates a secret to the initiator's KEM public key and signs the hash. It sends the ciphertext and signature back.
5. **Establish:** Both parties decapsulate the peer's ciphertext, combine the two shared secrets with the session cookie to **derive independent transmit and receive keys**, and initialize the RCS cipher states. Each side now has 512-bit keys for both directions and sends an "establish verify" message.

Duplex connections provide mutual authentication and can optionally initiate **key ratchet requests** after channel establishment via qsmp_duplex_send_symmetric_ratchet_request().

4 Implementation Steps

4.1 Key Provisioning

1. **Generate signature keys:** Use qsmp_generate_keypair() on each host that will act as a Simplex server or participate in Duplex connections. Persist the secret signing key (sigkey) in secure storage (e.g., HSM or encrypted file). The public verification key (verkey), configuration string and key ID should be distributed to peers. Keys should be rotated before the expiration time defined in the qsmp_server_signature_key structure.
2. **Define configuration strings:** Each QSMP configuration string encodes the chosen KEM (Kyber/McEliece), signature scheme (Dilithium/SPHINCS+), hash family and symmetric cipher. Both ends must use identical configuration strings; mismatches lead to qsmp_error_unknown_protocol.
3. **Embed key IDs and verification keys:** In Simplex mode, embed the server's verification key in client software or distribute it via secure registration. In Duplex mode, maintain a key store mapping key IDs to verification keys; implement a key_query() callback for

qsmp_client_duplex_listen_*() to fetch the peer's verification key based on the provided key ID.

4.2 Initializing a QSMP Connection

4.2.1 Simplex Client

```
// Prepare server verification key (pubk) and network address
qsmp_client_verification_key pubk;
// ... populate pubk.config, pubk.keyid, pubk.verkey and expiration


qsmp_connection_state* cns;
// allocate or zero cns


qsmp_errors ret = qsmp_client_simplex_connect_ipv4(
    &pubk,
    &server_ipv4,    // qsc_ipinfo_ipv4_address structure
    server_port,
    send_func,
    receive_callback);
if (ret != qsmp_error_none) {
    // handle connection error
}
```

During the handshake, the library invokes send_func(cns) to transmit packets (you should use your underlying network API to send qsmp_packet_to_stream()) and calls receive_callback(cns, data, len) for incoming packets. After the exchange completes, the connection state cns contains initialized txcpr and rxcpr cipher contexts. To encrypt a message, call qsmp_packet_encrypt(cns, &packet, message, msglen) and send the resulting packet. To decrypt, call qsmp_packet_decrypt(cns, plaintext, &plen, &packet) in the receive callback.

4.2.2 Simplex Server

Implement a server loop that accepts connections (e.g., using BSD sockets) and for each new client creates a qsmp_connection_state. The QSMP API offers two options:

1. **Single connection (client API):** Use qsmp_client_simplex_listen_ipv4(prikey, send_func, receive_callback) to listen for one client. This convenience API hides the socket setup and

handshake; after acceptance, you can call qsmp_packet_encrypt/decrypt on the returned connection state.

2. **Multi-threaded server:** Bind and listen on a socket, then call qsmp_server_start_ipv4(&source_socket, prikey, receive_callback, disconnect_callback). Each new client will be handled in a separate thread; messages are delivered via receive_callback, and you can send responses using qsmp_packet_encrypt().

4.2.3 Duplex Connections

For peer-to-peer connections, each host runs both client and server roles. To initiate a connection, call qsmp_client_duplex_connect_ipv4(kset, rverkey, address, port, send_func, receive_callback). To accept, use qsmp_client_duplex_listen_ipv4(kset, send_func, receive_callback, key_query). Once the handshake completes, both hosts derive separate transmit and receive keys and may call qsmp_duplex_send_symmetric_ratchet_request() to ratchet the symmetric keys at any time during the session.

4.3 Sending and Receiving Messages

The QSMP API leaves actual socket I/O to the application via callback functions:

1. **send_func(qsmp_connection_state *cns):** Called by QSMP whenever it has a packet ready to send (during handshake or in response to a ratchet). The application must serialize the qsmp_network_packet into a byte buffer using qsmp_packet_to_stream(packet, buffer) and write it to the network.

2. **receive_callback(qsmp_connection_state *cns, const uint8_t *data, size_t len):** The application should call qsmp_packet_header_deserialize() to parse the header, then call qsmp_packet_decrypt() to retrieve the plaintext. If the header flag is qsmp_flag_encrypted_message, deliver the plaintext to your upper layer; handle other flags (keep-alive, ratchet) accordingly.

4.4 Keep-Alive and Connection Closure

QSMP implements a built-in **keep-alive** mechanism. Clients must respond to qsmp_flag_keep_alive_request packets by returning a corresponding response. Connections that miss keep-alive acknowledgements for more than QSMP_KEEPALIVE_TIMEOUT (default 120 s) should be terminated with qsmp_connection_close(cns, qsmp_error_keepalive_timeout, true). When either side is finished, send an empty packet with sequence equal to QSMP_SEQUENCE_TERMINATOR to close the session.

5 Integration Scenarios

5.1 Payment Networks and FinTech

QSMP is ideal for replacing SSH or TLS tunnels in remote payment processing, ATM maintenance and bank-to-bank messaging. In Simplex mode, a central server holds the private signing key and distributes a verification key to all point-of-sale (POS) terminals. Each terminal connects using qsmp_client_simplex_connect_*(), ensuring that cardholder data is encrypted with 256-bit post-quantum security. The deterministic packet header (flag, sequence, timestamp) simplifies compliance auditing and provides inherent replay protection. Integrators should embed the server's verification key in the POS firmware and implement periodic key rotation when the signature key approaches expiration.

5.2 Cloud Platforms and Micro-Services

For SaaS or micro-service architectures, deploy a QSMP multi-threaded server using qsmp_server_start_ipv4(). Each micro-service holds a copy of the server's private signature key or obtains a delegated key via qsmp_signature_key_deserialize(). Clients (other services, API gateways or remote administrators) use qsmp_client_simplex_connect_ipv4() or qsmp_client_duplex_connect_ipv4() depending on whether mutual authentication is required. Because the server state per client is <4 kB, a single host can handle hundreds of thousands of persistent tunnels. Coupled with the optional asymmetric ratchet, keys can be rotated frequently without renegotiating the full handshake.

5.3 SCADA and Industrial Control

SCADA systems often require device-to-device communications with very low latency and deterministic behavior. QSMP's Duplex mode enables two controllers to mutually authenticate and derive separate transmit and receive keys, ensuring that compromise of one direction does not expose the other. The minimal handshake and fixed header format allow QSMP packets to fit inside existing SCADA frames or to be carried over UDP. When integrating, assign each controller a persistent signature key pair and deploy a key-store or directory service so peers can look up verification keys via the key_query() callback in qsmp_client_duplex_listen_*().

5.4 IoT Devices and Edge Sensors

Resource-constrained IoT devices benefit from QSMP's small state, constant-time operations and avoidance of certificate parsing. In Simplex mode, each device stores only the server's verification key and uses qsmp_client_simplex_connect_*() to establish a secure tunnel for telemetry or command/control. The RCS cipher's wide-block design combined with KMAC authentication provides protection against bit-flipping and length-extension attacks with

negligible overhead. The 60-s default packet time window may be reduced for private networks with synchronized clocks.

6 Security and Operational Considerations

1. **Key rotation:** Keys include an expiration field. Rotate keys and configuration strings before they expire to avoid qsmp_error_key_expired and to maintain forward secrecy.
2. **Time synchronization:** Maintain a reliable time source (e.g., NTP) because packets with timestamps outside the QSMP_PACKET_TIME_THRESHOLD are rejected.
3. **Configuration enforcement:** Reject connections with mismatched configuration strings to avoid downgrade attacks.
4. **Error handling:** Use qsmp_error_to_string() to convert error codes to human-readable messages. For fatal errors, call qsmp_connection_close(cns, err, true) to tear down the session and notify the peer.
5. **Performance:** QSMP uses constant-time cryptographic operations and small per-session state. In high-throughput servers, allocate connection pools sized according to QSMP_CONNECTIONS_MAX (default 50 k).

7 Conclusion

QSMP provides a modern, unified approach to quantum-safe messaging. Its two handshake variants support both client-server and peer-to-peer trust models; its robust cryptographic primitives ensure 256 or 512 bit security; and its deterministic packet structure with built-in time validation simplifies implementation and auditability. Through the API functions described above, developers can easily embed QSMP into payment systems, cloud services, SCADA controllers and IoT devices. By adhering to the recommended practices for key provisioning, handshake management and packet processing, integrators will be well-positioned to offer long-term confidentiality and integrity in the post-quantum era.