Quantum Secure Messaging Protocol – QSMP 1.2

Revision 1.2c, June 6, 2023

John G. Underhill – john.underhill@protonmail.com

This document is an engineering level description of the QSMP 1.2 encrypted and authenticated network messaging protocols. There are two protocols specified in this standard, the SIMPLEX and DUPLEX forms of QSMP.

In its contents, a guide to implementing QSMP 1.2, an explanation of its design, links to a C reference implementation, as well as references to its component primitives and supporting documentation.

**Contents Page**

[Foreword](#Foreward) 2

[Figures](#Figures) 3

[Tables](#Tables) 4

1: [Introduction](#Introduction) 5

2: [Scope](#Scope) 7

3: [References](#References) 8

4: [Terms and definitions](#Terms_and_Definitions) 9

5: [Structures](#Structures) 10

6: [Duplex Operational Overview](#Operational_Overview) 17

7: [Simplex Operational Overview](#Operational_Overview) 28

7: [Duplex Formal Description](#Formal_Description) 35

8: [Simplex Formal Description](#Formal_Description) 41

8: [QSMP API](#QSMP_API) 45

9: [Design Decisions](#Design_Decisions) 56

# **Foreword**

This document is intended as the preliminary draft of a new standards proposal, and as a basis from which that standard can be implemented. We intend that this serves as an explanation of this new technology, and as a complete description of the protocol.

This document is the second revision of the specification of QSMP, further revisions may become necessary during the pursuit of a standard model, and revision numbers shall be incremented with changes to the specification. The reader is asked to consider only the most recent revision of this draft, as the authoritative expression of the QSMP specification.

Future revisions of this standards draft can be found at: https://github.com/Steppenwolfe65/QSMP

The author of this specification is John G. Underhill, and can be reached at john.underhill@protonmail.com

QSMP, the algorithm constituting the QSMP messaging protocol is patent pending, and is owned by John G. Underhill and Digital Freedom Defense Incorporated. The code described herein is copyrighted, and owned by John G. Underhill and Digital Freedom Defense Incorporated.

**Figures**

**Contents Page**

Figure 5.7: The QSMP packet structure. 14

Figure 6.1: QSMP Duplex connection request. 17

Figure 6.2: QSMP server connection response. 18

Figure 6.3: QSMP client exchange request. 20

Figure 6.4: QSMP server exchange response. 23

Figure 6.5: QSMP client establish request. 24

Figure 6.6: QSMP server establish response. 26

Figure 6.7: QSMP client establish request. 27

Figure 7.1: QSMP Duplex connection request. 28

Figure 7.2: QSMP server connection response. 29

Figure 6.3: QSMP client exchange request. 31

Figure 7.4: QSMP server exchange response. 33

Figure 7.5: QSMP client establish request. 34

# **Tables**

**Contents Page**

Table 5.1: The Protocol string choices in revision 2a. 10

Table 5.2: The client key structure. 10

Table 5.3: The server key structure. 11

Table 5.4: The keep alive state. 11

Table 5.5: The connection state structure. 12

Table 5.6: The Duplex client KEX state structure. 12

Table 5.7: The Duplex server KEX state structure. 13

Table 5.7: The Simplex client KEX state structure. 13

Table 5.8: The Simplex server KEX state structure. 13

Table 5.8: Packet header flag types. 15

Table 5.9: Error type messages. 16

Table 10.1a QSMP error strings. 45

Table 10.1b QSMP configuration string. 45

Table 10.1c QSMP packet structure. 45

Table 10.1d QSMP client key structure. 46

Table 10.1e QSMP keep alive state structure. 46

Table 10.1f QSMP configuration enumeration. 46

Table 10.1g QSMP errors enumeration. 47

Table 10.1h QSMP flags enumeration. 47

Table 10.1i QSMP constants. 49

Table 10.1j QSMP connection state structure. 49

Table 10.2a QSMP key structure. 51

Table 10.2b QSMP server state structure. 52

Table 10.3a QSMP client state structure. 55

# **1:** **Introduction**

There are numerous key exchange protocols in widespread use today, chief among these are mechanisms in secure networking protocols TLS, PGP, and SSH. These protocols define a means by which secret keys are exchanged between devices. A key exchange function is typically a part of a more complex scheme, one that incorporates authentication during and after the key exchange function, and establishes an encrypted tunnel between devices, using the shared secret to key symmetric ciphers, that encrypt and decrypt traffic flows. QSMP is a complete specification of a key exchange function, authentication mechanisms, and an encrypted tunnel.

Though existing schemes can be modified to incorporate quantum-strength primitives, QSMP breaks from these by creating an entirely new set of mechanisms, ones which have been designed for security and performance in the post-quantum security model. Rather than build on previous designs, given that a large migration to post-quantum strength cryptography is now inevitable, we decided to build something new, without the attached burden of backwards compatibility, and unnecessarily complexity due to artifacts in older protocols, versioning, and maintaining outdated APIs.

QSMP is a quantum secure messaging protocol, that employs state of the art asymmetric ciphers and signature schemes, and a post-quantum strength symmetric cipher. The current incarnation can use the Kyber, NTRU, or McEliece asymmetric ciphers, and the Dilithium, Falcon, or Sphincs+ signature schemes, the leading round-3 candidates in the NIST Post Quantum competition. It uses the authenticated symmetric stream cipher RCS, based on the wide-block Rijndael cipher, with increased rounds, a cryptographically strong key-schedule, and AEAD authentication using KMAC. QSMP was designed to be more flexible and more secure than the protocols it means to replace, and can be used in any context where strong post-quantum security is required in a networked communications stream.

There are two complete protocol specifications; SIMPLEX and DUPLEX.

The SIMPLEX protocol defines streamlined one-way authenticated key exchange. This exchange is a client/server unidirectional trust, whereby the client trusts the server, and asymmetric authentication consists of the server signing the public asymmetric key, and the client verifying the message with the server’s public asymmetric signature-verification key. The QSMP SIMPLEX protocol creates a 256-bit secure two-way encrypted network stream between the server and the client, in just two round trips. It is ideal in any use-case where an efficient and secure encrypted channel is required between a server and a client.

The DUPLEX protocol, is a bi-directional trust model, where two hosts authenticate each other, and exchange two shared secrets, which are combined to key a 512-bit secure encrypted communications stream. Both hosts in the exchange, have copies of the other hosts public signature-verification key, the hosts exchange signed public asymmetric cipher-keys, each create and exchange a shared secret and cipher-text, and combine those secrets to key 512-bit secure symmetric cipher instances, used to encrypt a network stream. The QSMP DUPLEX protocol is ideal for high-security post-quantum secure communications between remote hosts, and can be used in conjunction with the SIMPLEX protocol to register hosts on the network, distribute public signature keys, and connect hosts in a high-security environment.

The QSMP protocols are part of a larger framework currently being developed; the multi-party distributed cryptosystem (MPDC), which we believe will be the next generation of encrypted network communications systems. QSMP is also a very versatile and powerful set of tools, that can be applied in many different use-cases. We have designed these new protocols, to offer alternatives to older protocols, which are being retro-fitted with quantum secure algorithms, but are also aging software, that we do not feel offer the best performance or security as compared to these state-of-the-art protocols.

* 1. **Purpose**

The QSMP secure messaging protocol, utilized in conjunction with quantum secure asymmetric and symmetric cryptographic primitives, is used to create an encrypted and authenticated bi-directional communications channel. This specification presents a secure messaging protocol that creates encrypted communications channels, in such a way that:

1. The asymmetric cipher keys for both the send and receive channels, are ephemeral, and encapsulate shared secrets for one or both channels, that are also unique to each channel and session (forward secrecy).
2. The capture of the shared keys does not reveal any information about future sessions (predicative resistance).
3. That one-way or two-way associated trusts can be established using strong asymmetric and symmetric authentication.

# **2:** **Scope**

This document describes the QSMP secure messaging protocols, which are used to establish encrypted and authenticated communications between two hosts. This document describes the complete asymmetric key exchange, authentication, and the establishment of a secure network communications stream for both the QSMP SIMPLEX and DUPLEX protocols. This is a complete specification, describing the cryptographic primitives, the key derivation functions, and the complete client to server messaging protocols.

Test vectors and C reference code are available at https://github.com/Steppenwolfe65/QSMP

**2.1 Application**

This protocol is intended for institutions that implement secure communication channels used to encrypt and authenticate secret information exchanged between remote terminals.

The key exchange functions, authentication and encryption of messages, and message exchanges between terminals defined in this document must be considered as mandatory elements in the construction of an QSMP communications stream. Components that are not necessarily mandatory, but are the recommended settings or usage of the protocol shall be denoted by the key-words **SHOULD**. In circumstances where strict conformance to implementation procedures is required but not necessarily obvious, the key-word **SHALL** will be used to indicate compulsory compliance is required to conform to the specification.

## **3:** **References**

**3.1 Normative References**

The following documents serve as references for key components of QSMP:

1. NIST FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable Output Functions

2. NIST SP 800-185: Derived Functions cSHAKE, KMAC, TupleHash and ParallelHash

3. NIST SP 800-90A: Recommendation for Random Number Generation

4. NIST SP 800-108: Recommendation for Key Derivation using Pseudorandom Functions

5. NIST FIPS 197 The Advanced Encryption Standard

**3.2 Reference Links**

1. The QSMP C implementation: https://github.com/Steppenwolfe65/QSMP

2. The QSC Cryptographic library: https://github.com/Steppenwolfe65/QSC

3. The RCS authenticated stream cipher: https://github.com/Steppenwolfe65/RCS

4. The Keccak Code Package: https://github.com/XKCP/XKCP

5. NIST AES FIPS 197: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

## **4:** **Terms and Definitions**

**4.1 Kyber**

The Kyber asymmetric cipher and NIST Round 3 Post Quantum Competition candidate.

**4.2 McEliece**

The McEliece asymmetric cipher and NIST Round 3 Post Quantum Competition candidate.

**4.3 NTRU**

The NTRU asymmetric cipher and NIST Round 3 Post Quantum Competition candidate.

**4.4 Dilithium**

The Dilithium asymmetric signature scheme and NIST Round 3 Post Quantum Competition candidate.

**4.5 Falcon**

The Falcon asymmetric signature scheme and NIST Round 3 Post Quantum Competition candidate.

**4.6 SPHINCS+**

The SPHINCS+ asymmetric signature scheme and NIST Round 3 Post Quantum Competition candidate.

**4.7 RCS**

The Rijndael-256 Cryptographic Stream (RCS) authenticated symmetric stream cipher.

**4.8 SHA-3**

The SHA3 hash function NIST standard, as defined in the NIST standards document FIPS-202; SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.

**4.9 SHAKE**

The NIST standard Extended Output Function (XOF) defined in the SHA-3 standard publication FIPS-202; SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.

**4.10 KMAC**

The SHA3 derived Message Authentication Code generator (MAC) function defined in NIST special publication SP800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash.

## **5:** **Structures**

**5.1 Protocol string**

The protocol string is comprised of four unique components;

1. The asymmetric signature scheme string, including the security strength of the asymmetric signature scheme (s1, s3, s5), ex. dilithium-s3.
2. The asymmetric encapsulation cipher, including the security strength, ex. mceliece-s5.
3. The hash function family, SHA3.
4. The symmetric cipher, RCS.

The protocol string is used during the initial protocol negotiation to identify the protocol settings of the client and server. The client and server must support a common parameter set to establish a connection.

|  |  |  |  |
| --- | --- | --- | --- |
| Signature Scheme | Asymmetric Cipher | HASH Function | Symmetric Cipher |
| Dilithium | Kyber | SHA3 | RCS |
| Dilithium | McEliece | SHA3 | RCS |
| Dilithium | NTRU | SHA3 | RCS |
| Falcon | Kyber | SHA3 | RCS |
| Falcon | NTRU | SHA3 | RCS |
| Falcon | McEliece | SHA3 | RCS |
| Sphincs+ | McEliece | SHA3 | RCS |

Table 5.1: The Protocol string choices in revision 2a.

Note that the table above does not indicate all possible algorithm combinations. Future revisions will include other algorithms not listed, or different combinations of asymmetric cipher and signature scheme not noted here.

**5.2 Client Key**

The client key is an internal structure that stores the signature verification key and related variables, including the public-key expiration time, the protocol string, the public signature verification key, and the key identity array.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Data Type | Bit Length | Function |
| Expiration | Uint64 | 64 | Validity check |
| Configuration | Uint8 array | 320 | Protocol check |
| Key ID | Uint8 array | 128 | Identification |
| Verification Key | Uint8 array | Variable | Authentication |

Table 5.2: The client key structure.

The expiration parameter is a 64-bit unsigned integer that holds the seconds from the last epoch (01/01/1900) to the time the key remains valid. This value is checked during the initialization of the client, and if the key has expired, the connection attempt is halted, and the client must retain a new public key from the server.

The configuration parameter contains the protocol string associated with the signature verification public-key, the asymmetric cipher, the hash family, and the symmetric cipher. This value is checked during initialization, and if the protocol string does not match the on both hosts, the connection is aborted.

The key identity array is a 16-byte array that uniquely identifies a public verification key. This identifier can be used to match the key on a server.

The public key, is the public asymmetric signature verification key. This key can be distributed to clients, posted to a website, or distributed in any way public or private. It can also be signed using X.509 to create a ‘chain of trust’, in an extension to this protocol. It is used to verify the signature of an asymmetric encapsulation key, sent to the client during the key exchange.

**5.3 Server Key**

The server key is identical to the client key except for one additional parameter, the asymmetric signing key. It contains both the signature schemes verification and secret signing keys, along with the expiration, configuration, and key identity parameters.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| Expiration | Uint64 | 64 | Validity check |
| Configuration | Uint8 array | 320 | Protocol check |
| Key ID | Uint8 array | 128 | Identification |
| Verification Key | Uint8 array | Variable | Authentication |
| Signing Key | Uint8 array | Variable | Authenticating |

Table 5.3: The server key structure.

**5.4 Keep Alive State**

QSMP uses an internal keep-alive loop function. The server sends the client a keep alive packet, every QSMP\_KEEPALIVE\_TIMEOUT interval, with a default of 300 seconds.

The client echoes this keep alive back to the server to acknowledge receipt, proving it is still connected to the server. If the keep alive is not answered within the keep alive time-out period, the server will send a **bad keep alive** error message to the client, tear down the connection, and dispose of the server state.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Data Type | Bit Length | Function |
| Expiration Time | Uint64 | 64 | Validity check |
| Packet Sequence | Uint64 | 64 | Protocol check |
| Received Status | Bool | 8 | Status |

Table 5.4: The keep alive state.

**5.5 Connection State**

The connection state is an internal structure that contains all the variables required by the QSMP operations. This includes the Duplex symmetric ratchet key structure, packet counters and flag, and send and receive channels symmetric cipher states.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| Target | Socket struct | 664 | Validity check |
| Cipher Send State | Structure | Variable | Symmetric Encryption |
| Cipher Receive State | Structure | Variable | Symmetric Decryption |
| Receive Sequence | Uint64 | 64 | Packet Verification |
| Send Sequence | Uint64 | 64 | Packet Verification |
| Connection Instance | Uint32 | 32 | Identification |
| KEX Flag | Uint8 | 8 | KEX State Flag |
| Ratchet Key | Uint8 array | 512 | Symmetric Ratchet |
| PkHash | Uint8 array | 256 | Authentication |
| Session Token | Uint8 array | 256 | Authentication |
| ExFlag | Uint8 | 8 | Protocol Check |

Table 5.5: The connection state structure.

**5.6 Duplex Client KEX State**

The Duplex client state structure stores the asymmetric cipher and signature keys used during the key exchange execution.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| Key ID | Uint8 array | 128 | Key Identification |
| Session Token | Uint8 array | 512 | Verification |
| Private Cipher Key | Uint8 array | Variable | Asymmetric Encryption |
| Public Cipher Key | Uint8 array | Variable | Asymmetric Encryption |
| Remote Verification Key | Uint8 array | Variable | Asymmetric Authentication |
| Signature Key | Uint8 array | Variable | Asymmetric Authentication |
| Shared Secret | Uint8 array | 256 | Symmetric Key |
| Verification Key | Uint8 array | Variable | Asymmetric Authentication |
| Expiration | Uint64 | 64 | Verification |

Table 5.6: The Duplex client KEX state structure.

**5.7 Duplex Server KEX State**

The Duplex server state structure stores the asymmetric cipher and signature keys used during the key exchange execution.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| Key ID | Uint8 array | 128 | Key Identification |
| Session Token | Uint8 array | 512 | Verification |
| Private Cipher Key | Uint8 array | Variable | Asymmetric Encryption |
| Public Cipher Key | Uint8 array | Variable | Asymmetric Encryption |
| Remote Verification Key | Uint8 array | Variable | Asymmetric Authentication |
| Signature Key | Uint8 array | Variable | Asymmetric Authentication |
| Shared Secret | Uint8 array | 256 | Symmetric Key |
| Verification Key | Uint8 array | Variable | Asymmetric Authentication |
| Expiration | Uint64 | 64 | Verification |
| Key Query Callback | Uint64 | 64 | Function Pointer |

Table 5.7: The Duplex server KEX state structure.

**5.7 Simplex Client KEX State**

The Simplex client state structure stores the asymmetric cipher and signature keys used during the key exchange execution.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| Key ID | Uint8 array | 128 | Key Identification |
| Session Token | Uint8 array | 512 | Verification |
| Remote Verification Key | Uint8 array | Variable | Asymmetric Authentication |
| Signature Key | Uint8 array | Variable | Asymmetric Authentication |
| Shared Secret | Uint8 array | 256 | Symmetric Key |
| Verification Key | Uint8 array | Variable | Asymmetric Authentication |
| Expiration | Uint64 | 64 | Verification |

Table 5.7: The Simplex client KEX state structure.

**5.8 Simplex Server KEX State**

The Simplex server state structure stores the asymmetric cipher and signature keys used during the key exchange execution.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| Key ID | Uint8 array | 128 | Key Identification |
| Session Token | Uint8 array | 512 | Verification |
| Private Cipher Key | Uint8 array | Variable | Asymmetric Encryption |
| Public Cipher Key | Uint8 array | Variable | Asymmetric Encryption |
| Signature Key | Uint8 array | Variable | Asymmetric Authentication |
| Shared Secret | Uint8 array | 256 | Symmetric Key |
| Verification Key | Uint8 array | Variable | Asymmetric Authentication |
| Expiration | Uint64 | 64 | Verification |

Table 5.8: The Simplex server KEX state structure.

**5.7 QSMP Packet Header**

The QSMP packet header is 9 bytes in length, and contains:

1. The **Packet Flag**, the type of message contained in the packet; this can be any one of the key-exchange stage flags, a message, or an error flag.
2. The **Packet Sequence**, this indicates the sequence number of the packet exchange.
3. The **Message Size**, this is the size in bytes of the message payload.

The message is a variable sized array, up to QSMP\_MESSAGE\_MAX in size.

|  |  |  |
| --- | --- | --- |
| **Packet Flag**  **1 byte** | **Packet Sequence**  **8 bytes** | **Message Size**  **4 bytes** |
| **Message**  **Variable Size** | | |

Figure 5.7: The QSMP packet structure.

This packet structure is used for both the key exchange protocol, and the communications stream.

**5.8 Flag Types**

The following are a preliminary list of packet flag types used by QSMP:

|  |  |  |
| --- | --- | --- |
| Flag Name | Numerical Value | Flag Purpose |
| None | 0x00 | No flag was specified, the default value. |
| Connect Request | 0x01 | The key-exchange client connection request flag. |
| Connect Response | 0x02 | The key-exchange server connection response flag. |
| Connection Terminated | 0x03 | The connection is to be terminated. |
| Encrypted Message | 0x04 | The message has been encrypted by the communications stream. |
| Exchange Request | 0x07 | The key-exchange client exchange request flag. |
| Exchange Response | 0x08 | The key-exchange server exchange response flag. |
| Establish Request | 0x09 | The key- exchange client establish request flag. |
| Establish Response | 0x0A | The key- exchange server establish response flag. |
| Keep Alive Request | 0x0B | The packet contains a keep alive request. |
| Keep Alive Response | 0x0C | The packet contains a keep alive response. |
| Remote Connected | 0x0D | The remote host has terminated the connection. |
| Remote Terminated | 0x0E | The remote host has terminated the connection. |
| Session Established | 0x0F | The session is in the established state. |
| Establish Verify | 0x10 | The session is in the verify state. |
| Unrecognized Protocol | 0x11 | The protocol string is not recognized |
| Aymmetric Ratchet Request | 0x12 | The packet contains an asymmetric ratchet request. |
| Aymmetric Ratchet Response | 0x13 | The packet contains an asymmetric ratchet response. |
| Symmetric Ratchet Request | 0x14 | The packet contains a symmetric ratchet request. |
| Error Condition | 0xFF | The connection experienced an error. |

Table 5.8: Packet header flag types.

**5.9 Error Types**

The following are a preliminary list of error messages used by QSMP:

|  |  |  |
| --- | --- | --- |
| Error Name | Numerical Value | Description |
| None | 0x00 | No error condition was detected. |
| Authentication Failure | 0x01 | The symmetric cipher had an authentication failure. |
| Bad Keep Alive | 0x02 | The keep alive check failed. |
| Channel Down | 0x03 | The communications channel has failed. |
| Connection Failure | 0x04 | The device could not make a connection to the remote host. |
| Connect Failure | 0x05 | The transmission failed at the KEX connection phase. |
| Decapsulation Failure | 0x06 | The asymmetric cipher failed to decapsulate the shared secret. |
| Establish Failure | 0x07 | The transmission failed at the KEX establish phase. |
| Exstart Failure | 0x08 | The transmission failed at the KEX exstart phase. |
| Exchange Failure | 0x09 | The transmission failed at the KEX exchange phase. |
| Hash Invalid | 0x0A | The public-key hash is invalid. |
| Invalid Input | 0x0B | The expected input was invalid. |
| Invalid Request | 0x0C | The packet flag was unexpected. |
| Keep Alive Expired | 0x0D | The keep alive has expired with no response. |
| Key Expired | 0x0E | The QSMP public key has expired. |
| Key Unrecognized | 0x0F | The key identity is unrecognized. |
| Packet Un-Sequenced | 0x10 | The packet was received out of sequence. |
| Random Failure | 0x11 | The random generator has failed. |
| Receive Failure | 0x12 | The receiver failed at the network layer. |
| Transmit Failure | 0x13 | The transmitter failed at the network layer. |
| Verify Failure | 0x14 | The expected data could not be verified. |
| Unknown Protocol | 0x15 | The protocol string was not recognized. |
| Listener Failure | 0x16 | The listener function failed to initialize. |
| Accept Failure | 0x17 | The socket accept function returned an error. |
| Hosts Exceeded | 0x18 | The server has run out of socket connections. |
| Allocation Failure | 0x19 | The server has run out of memory. |
| Decryption Failure | 0x1A | The decryption authentication has failed. |
| Ratchet Failure | 0x1C | The ratchet operation has failed. |

Table 5.9: Error type messages.

# **6:** **Duplex Protocol Operational Overview**

During initialization, clients generate an asymmetric signature scheme key-pair, the private key that the clients use to sign a key exchange, and the public key, which is distributed to other hosts, and contains the asymmetric signature verification key, along with the key identity array, protocol configuration string, and key expiration date.

The public/private signature keys are generated by the client, and serve as their root authentication keys. The public verification keys can be distributed to other clients through a trusted third party, like a server running a directory service. These client verification keys can be signed by that server, and by auxiliary authentication servers as an extension of this protocol.

The clients in the exchange are designated as either a server, which accepts the network connection request, or a client which requests the connection.

The client initiates a connection, which if the key is valid and known to the server, initializes a key exchange. Asymmetric cipher keys are authenticated and exchanged between the client and server, which generate a pair of shared secrets, used to key symmetric cipher instances, in the transmit and receive directions of a duplexed communications channel.

Any error during the key exchange or during the communications operations, causes the client or server to send an error message to the other host, disconnect, and tear down the session. This includes checks for message synchronization, expected size of sent and received messages during the key exchange, authentication failures, and internal errors raised by cryptographic or network functions used by the key exchange and communications stream.

* 1. **Connection Request**

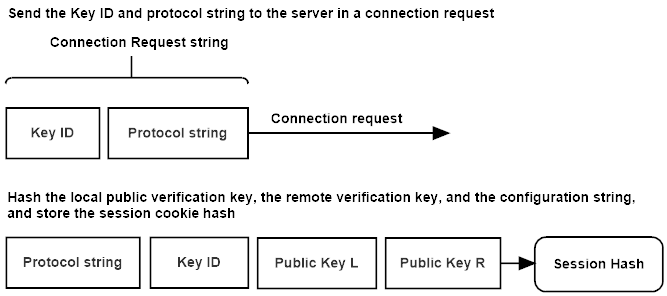
****

Figure 6.1: QSMP Duplex connection request.

The client initializes a key exchange operation, by sending the server a **connect request** packet. The packet message contains the client’s key identification array, and the protocol configuration string. The client stores a hash of the protocol string, and the client and the server’s asymmetric signature verification key in the session cookie **sch** state value, for use later in the key exchange.

* 1. **Connection Response**

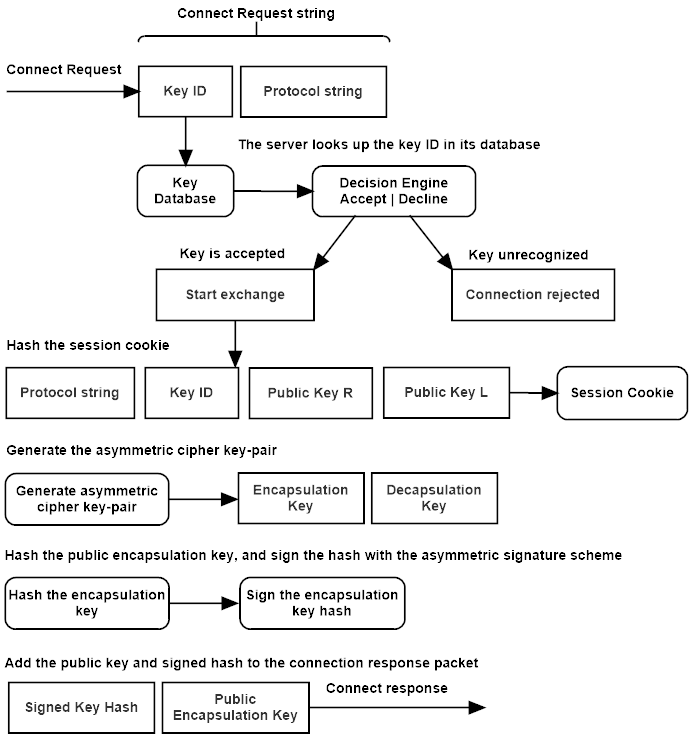
****

Figure 6.2: QSMP server connection response.

The server checks its database for a key matching the key identification array sent by the client in the **connect request** message. The server then compares the configuration string contained in the message against its own protocol string for a match. The server also verifies the key’s expiration time, and if all fields are valid, loads the key into state. If the protocol configuration strings do not match, the server will send an **unknown protocol** error to the client and close the connection. If the client’s key has expired, the server will send a **key expired** error message. If the key is not known to the server, the server sends a **key unrecognized** error message to the client. In any of these failures occur, the server closes the connection and logs the event. The client closes the connection, and passes the error up to the user interface software, that can initiate actions or inform the user of the cause of the failure.

The server hashes the key ID array, the client’s verification key, and its signature verification key, and stores the hash in its session cookie state value *sch*, for use as a unique session cookie.

The server generates a public/private asymmetric cipher key-pair. The server hashes the public key, and signs the hash with the asymmetric signature scheme’s private signing key. The client has a copy of the server’s verification key, that will be used to verify this signature. The server stores the private asymmetric cipher key temporarily in its state.

The server adds the public asymmetric encapsulation key, and the public keys signed hash, to the **connect response** message, and sends it to the client.

* 1. **Exchange Request**

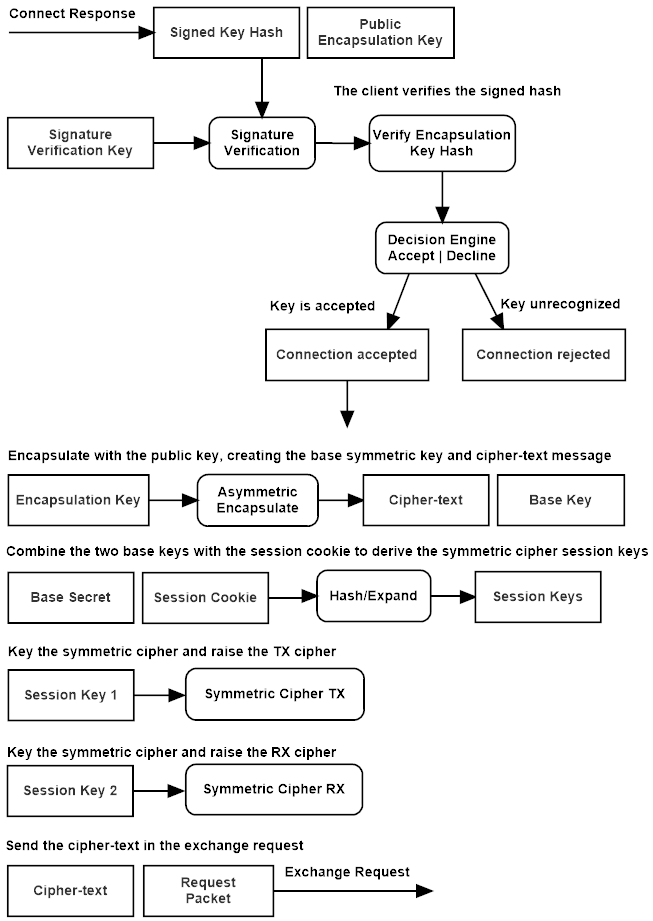
****

Figure 6.3: QSMP client exchange request.

The client uses the server’s public signature verification key to check the signature on the asymmetric encapsulation key’s hash, that was sent along with the asymmetric ciphers encapsulation key in the **connect response** message. If the signature is verified, the asymmetric cipher key is hashed, and that hash is compared to the signed hash contained in the servers connect response message. If the signature verification fails, the client sends an **authentication failure** message and terminates the connection, likewise if the hash check fails, the client sends a **hash invalid** error message.

The client uses the asymmetric cipher key to encapsulate a base *shared secret*, producing a cipher-text that will be sent to the server, and used to generate a shared secret value. The shared secret is stored, and will be used to derive session keys in a later step.

The client generates an asymmetric encryption key-pair, stores the private key, and hashes the public key and cipher-text, then uses its private signing key to sign the hash.

The asymmetric cipher-text, public encryption key, and the signed hash are added to the **exchange request** packet, and sent to the server.

* 1. **Exchange Response**

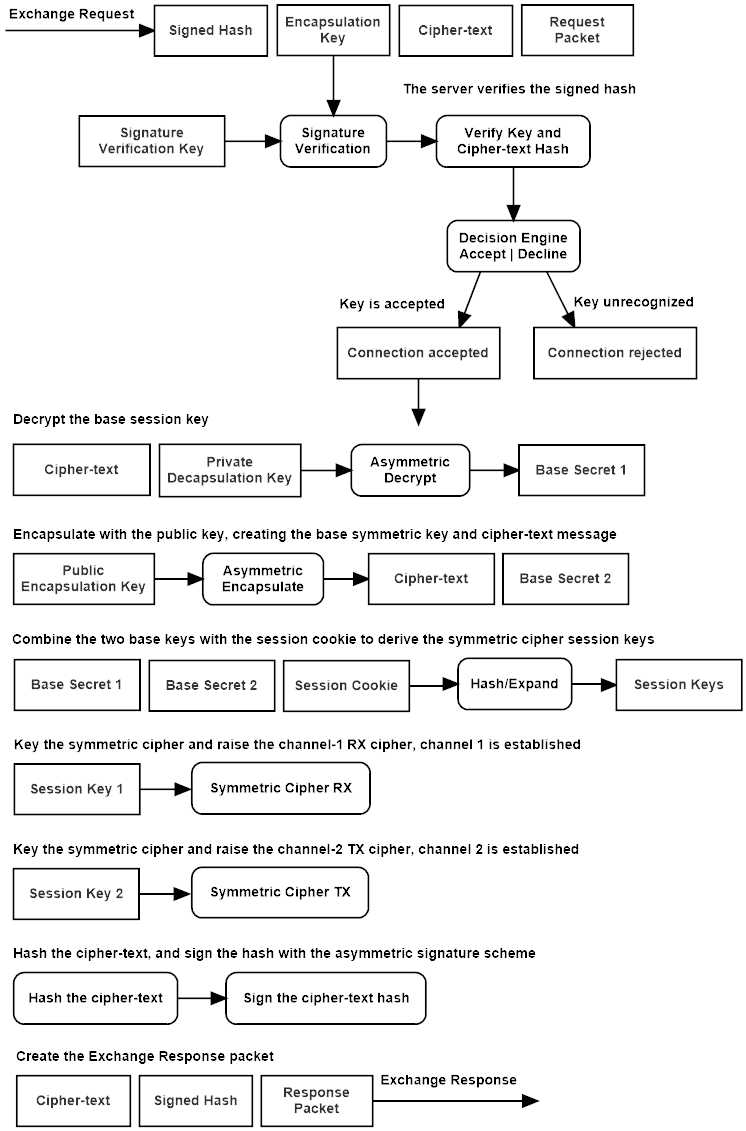
****

Figure 6.4: QSMP server exchange response.

The server verifies the signature of the hash of the cipher-text and key sent by the client in the **exchange request** packet using the client’s public asymmetric verification key, and then hashes the public key and cipher-text and verifies the hash for equivalence to the one contained in the signed hash. The server then uses the stored asymmetric cipher private key to decapsulate the first shared secret. The server then uses the public key sent by the client to generate a new shared secret and encapsulate it in cipher-text.

The two shared secrets and the session cookie are combined and used to derive the two symmetric session keys. The symmetric cipher instances are keyed with the session keys, raising both the transmit and receive channels of the encrypted tunnel.

The cipher-text is hashed, the hash is signed by the server’s private asymmetric signature key, and these are sent back to the client in an **exchange response** packet.

* 1. **Establish Request**

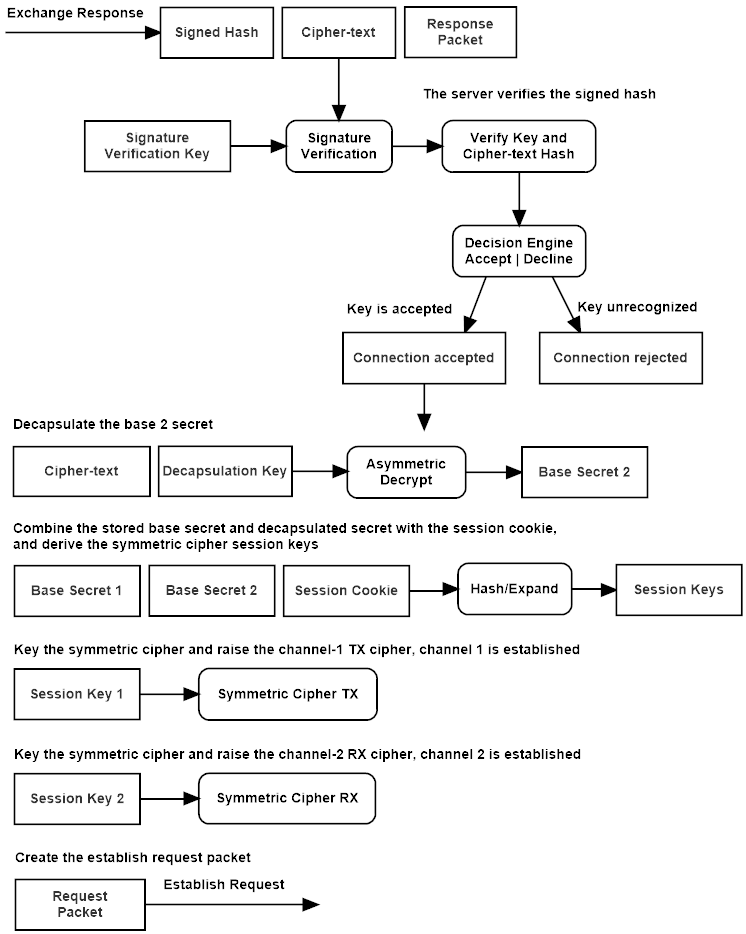
****

Figure 6.5: QSMP client establish request.

The client receives the exchange response packet from the server with the asymmetric cipher-text and the signed hash of the cipher-text. The client verifies the hash of cipher-text using the asymmetric signature scheme and the server’s public verification key. It then verifies the hash by hashing the cipher-text and comparing the signed hash with the newly hashed value.

The client decapsulates the shared secret from the cipher-text, adds the stored shared secret and the session cookie, and derives the symmetric session keys. The client keys both the transmit and receive cipher instances and raises both channels of the encrypted tunnel.

* 1. **Asymmetric Ratchet**

The Asymmetric ratchet injects new entropy into the encrypted stream, by re-keying the symmetric ciphers using a hash of the original key, combined with new keying material obtained through an asymmetric key exchange. When the primary KEX is run, extablishing the encrypted tunnel, the primary session keys for the receive and transmit channels are hashed, and this hash is stored in a persistant ratchet state. When the asymmetric ratchet function is called, the initiator creates a new asymmetric cipher key pair, and sends the public key over the encrypted tunnel to the remote host. The public key is first hashed, and the hash of the public key is signed by the initiators private signature key.

The receiving host verifies the signature, using the copy of the remote hosts public signature verification key. If the signature hash of the public cipher key is verified, the host hashes the public key it received and compares it to the signed hash for verification. If the signature and hash are correct, the host uses the public key to generate a new secret and cipher-text. The secret is hashed along with the hash of the original session key to derive a new set of session keys, which are used to re-key the transmit and receive channels symmetric cipher instances.

The receiver then hashes the cipher-text and signs the hash with their private signature key, and send the signed hash and cipher-text back to the initiator. The initiator verifies the signature of the hash using its copy of the remote hosts public signature verification key, then compares the hash to a hash it creates of the cipher-text. If verified the cipher-text is decrypted, and the shared secret is hashed along with the hash of the initial session key to create the new symmetric ciphers transmit and receive channels session keys.

The persistant ratchet key is then updated by hashing the new session keys, to be used the next time the ratchet mechanism is called. Either host, sender or receiver may call the asymmetric ratchet function. The function can be called from the hosting software, for example; after a byte limit is surpassed, when a new session is created if the keys are persistant, or even after each message in the exchange. This mechanism provides both forward security, in that an adversary can not derive previous keys from the current key, and predictive resistance, in that future keys can not be derived from the current state alone.

* 1. **Symmetric Ratchet**

The symmetric ratchet sends a random token over the encrypted channel. This key is hashed along with the hash of the initial session keys kept in the persistant ratchet state. The initiator generates a random token, and sends it through the encrypted tunnel. The initiator hashes this token along with the ratchet key, a hash of the currently used session keys, to derive a new set of session keys which are used to re-key the receive and transmit channels symmetric cipher instances. The receiver decrypts the random token, hashes it with the ratchet key and derives the new session keys. This mechanism provides forward security, in that knowledge of the current state alone is not enough to derive previous session keys.

* 1. **Establish Response**

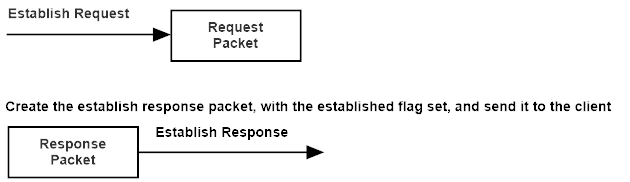
****

Figure 6.6: QSMP server establish response.

The server sends a message with either the establish response flag set, indicating the encrypted tunnel has been raised, or with an error flag set, which will tear down the connection on both ends.

* 1. **Establish Verify**

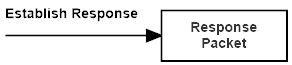
****

Figure 6.7: QSMP client establish request.

The client receives the establish response packet, verifying the tunnel is now in operation. If an error has occurred, the error flag will be set, causing the connection to be torn down.

# **7: Simplex Protocol Operational Overview**

The Simplex exchange is a one-way trust client/server model key exchange. With the client trusting the server, and a single shared secret exchanged. It was designed to be fast and lightweight, and provides strong 256-bit post-quantum security.

This exchange can be applied in many different use-cases, such as registration of clients on a network, cloud storage, secure communications in a hub-and-spoke model, commodity trading and electronic currency exchange; anywhere an encrypted tunnel using strong quantum-safe cryptography is required.

The server is a multi-threaded communications platform, that produces a uniquely keyed encrypted tunnel for each client. The total state for each client is less than 4 kilobytes, and a single server instance can manage potentially hundreds of thousands of simultaneous connections. The cipher encapsulation keys used during the key exchange are ephemeral, and unique to every key exchange.

The server shares a public signature verification key with clients. And uses this key to verify the public cipher encapsulation key to the client. This public signature verification key can be distributed to clients through a registration event, or embedded in the client software, or shared through some other secure means. The verification key can be signed by an outside authority in a PKI implementation, or by other shared trust models like PGP, or the distributed trust model these key exchanges were written for, MPDC.

* 1. **Connection Request**

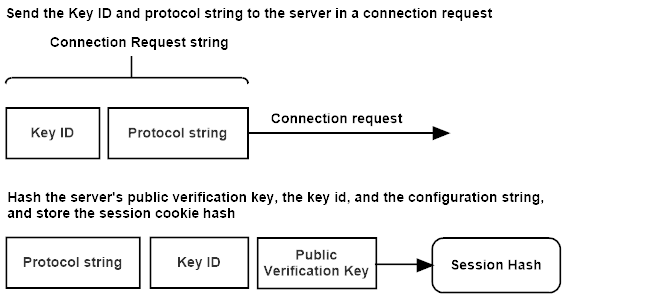
****

Figure 7.1: QSMP Duplex connection request.

The client initializes a key exchange operation, by sending the server a **connect request** packet. The packet message contains the client’s key identification array, and the protocol configuration string. The client hashes the configuration string, the key identification array, and the signature verification key, and stores the hash in its session cookie state value *sch*, for use as a unique session cookie.

* 1. **Connection Response**

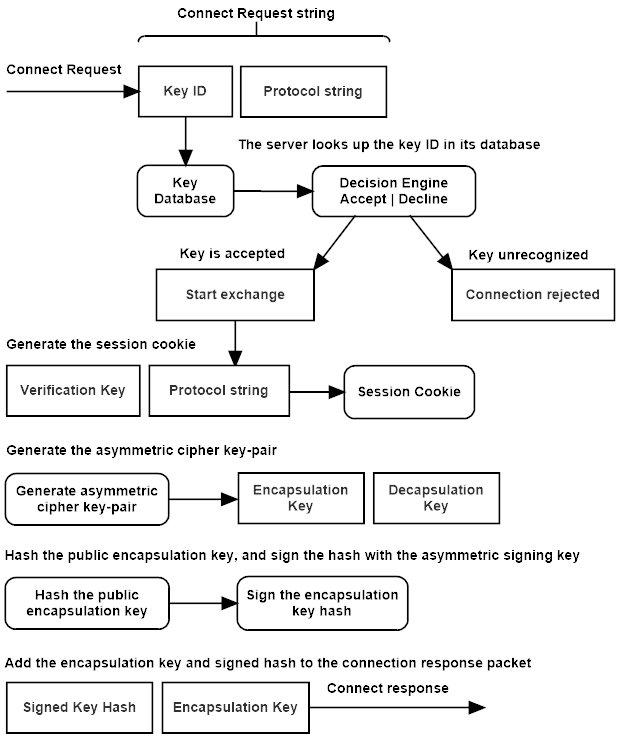
****

Figure 7.2: QSMP server connection response.

The server checks its database for a key matching the key identification array sent by the client in the **connect request** message. The server then compares the configuration string contained in the message against its own protocol string for a match. The server also verifies the key’s expiration time, and if all fields are valid, loads the key into state. If the protocol configuration strings do not match, the server will send an **unknown protocol** error to the client and close the connection. If the client’s key has expired, the server will send a **key expired** error message. If the key is not known to the server, the server sends a **key unrecognized** error message to the client. In any of these failures occur, the server closes the connection and logs the event. The client closes the connection, and passes the error up to the user interface software, that can initiate actions or inform the user of the cause of the failure.

The server hashes the configuration string, the key identification array, and the signature verification key, and stores the hash in its session cookie state value *sch*, for use as a unique session cookie.

The server generates a public/private asymmetric cipher key-pair. The server hashes the public encapsulation key, and signs the hash with the asymmetric signature scheme’s private signing key. The client has a copy of the server’s verification key, that will be used to verify this signature. The server stores the private asymmetric cipher key temporarily in its state.

The server adds the public asymmetric encapsulation key, and the public keys signed hash, to the **connect response** message, and sends it to the client.

* 1. **Exchange Request**

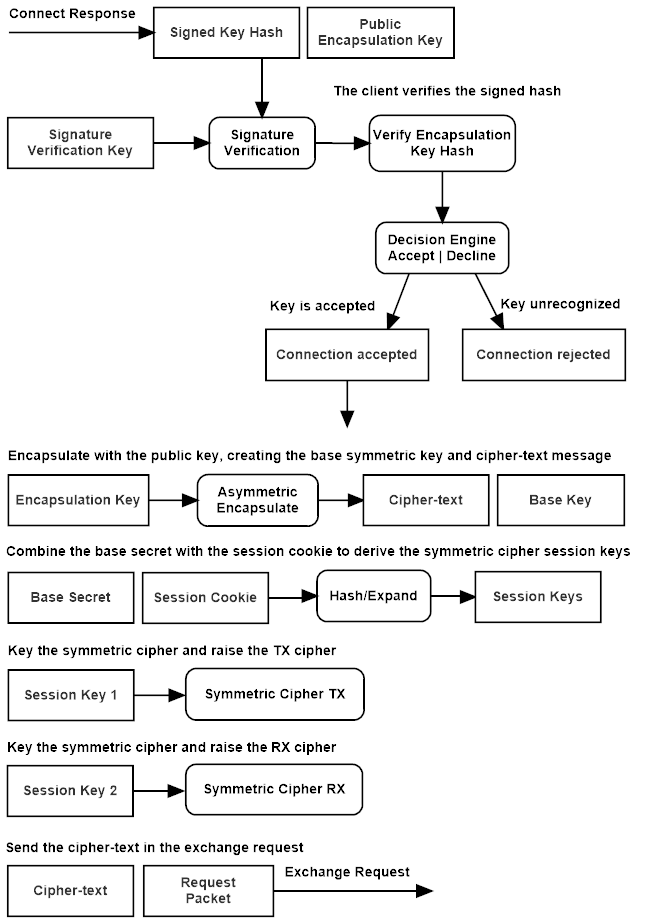
****

Figure 6.3: QSMP client exchange request.

The client uses the server’s public signature verification key to check the signature on the asymmetric encapsulation key’s hash, that was sent along with the asymmetric ciphers encapsulation key in the **connect response** message. If the signature is verified, the asymmetric cipher key is hashed, and that hash is compared to the signed hash contained in the servers connect response message. If the signature verification fails, the client sends an **authentication failure** message and terminates the connection, likewise if the hash check fails, the client sends a **hash invalid** error message.

The client uses the asymmetric cipher key to encapsulate a base *shared secret*, producing a cipher-text that will be sent to the server, and used to generate the session keys.

The asymmetric cipher-text is added to the **exchange request** packet, and sent to the server.

* 1. **Exchange Response**

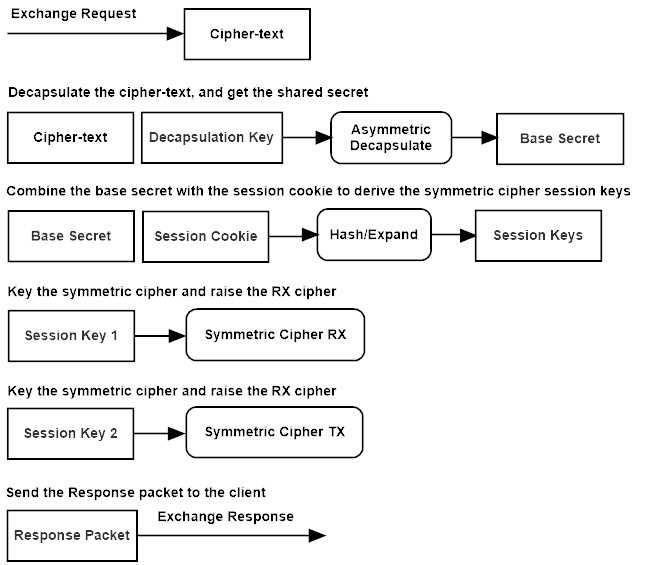
****

Figure 7.4: QSMP server exchange response.

The server uses the stored asymmetric cipher private key to decapsulate the shared secret from the cipher-text contained in the **exchange request**.

The shared secret and the session cookie are combined and used to derive the two symmetric session keys. The symmetric cipher instances are keyed with the session keys, raising both the transmit and receive channels of the encrypted tunnel.

The server sets the session established flag and sends it to the client in the **exchange response** packet.

* 1. **Establish Verify**

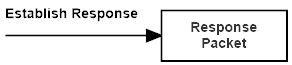
****

Figure 7.5: QSMP client establish request.

The client receives the establish response packet, verifying the tunnel is now in operation. If an error has occurred, the error flag will be set, causing the connection to be torn down.

# **QSMP Duplex Formal Description**

**Legend:**

🡨🡪 -Assignment operators

**:=, !=, ?=** -Equality operators; equals, not equals, evaluate

**C** -The client host, initiates the exchange

**S** -The server host, listens for a connection

**AG** -The asymmetric cipher key generator function

**-AEsk** -The asymmetric decapsulation function and secret key

**AEpk** -The asymmetric encapsulation function and public key

**ASsk** -Sign data with the secret signature key

**AVpk** -Verify a signature the public verification key

**cfg** -The protocol configuration string

**cond,** -A conditional statement

**cprrx** -A receive channels symmetric cipher instance

**cprtx** -A transmit channels symmetric cipher instance

**cpt** -The symmetric ciphers cipher-text

**cpta** -The asymmetric ciphers cipher-text

**-Ek** -The symmetric decryption function and key

**Ek** -The symmetric encryption function and key

**H** -The hash function (SHA3)

**k,mk** -A symmetric cipher or MAC key

**KDF** -The key expansion function (SHAKE)

**kid** -The public keys unique identity array

**Mmk** -The MAC function and key (KMAC)

**pk,sk** -Asymmetric public and secret keys

**pvk** -Public signature verification key

**sch** -A hash of the configuration string and and asymmetric verification-keys

**sec** -The shared secret derived from asymmetric encapsulation and decapsulation

**spkh** -The signed hash of the asymmetric public encapsulation-key

**The Duplex Key Exchange Sequence**

**Preamble:**

The duplex key exchange and tunnel is designed as a client-to-client architecture. Each client has a unique signature verification key that is shared with other clients. This key can be exchanged using a host lookup system, whereby a client queries a server that keeps a database of clients on its network. The server checks the request for authorization, and returns information about the client, including this public key. The key can then be cached on the client for as long as the key expiration time is valid. The server can also act as a point of authentication, and can sign client verification keys. Because there must be one node that initiates a connection in the exchange, and one that accepts, the initiator is considered the client in the key exchange, and the receiver of the request is considered to be the server.

The client first checks the expiration date on the servers public key, if the key is invalid, it queries the server for a new signature verification key. This mechanism should be a re-authentication session with the server, in which the new key is distributed over an encrypted channel, and the client verifies this key certificate, using whichever authentication authority or scheme has been implemented by the server and client software. For example, the servers public verification key can be signed by PGP or MPDC, or through the standard PKI, and verified through that mechanism.

When a client wants to connect to a remote host, it checks its public key cache, for a valid public signature verification key for that host. If the client does not have a copy of a valid key, it queries the server in an address lookup message exchange, that sends the remote hosts signed public verification key, along with some contact details for the remote host. The server should implement access safeguards in a connection protocol, that controls access between hosts on the network, i.e. a contact request mechanism.

Once both hosts have copies of each other’s signature verification keys, and those keys have been authenticated, the clients are considered in a pre-connection ready state.

**8.1 Connect Request:**

The client sends a connection request with its configuration string, and signature key identity string.

The key identity (kid) is a multi-part 16-byte address and key identification array, used to match the intended target to the corresponding key. The configuration string defines the cryptographic protocol set being used, these must be identical.

The client stores a hash of the configuration string, the key identity, and both of the public asymmetric signature verification-keys, which is used as a session cookie during the exchange.

**sch 🡨 H(cfg || kid || pvka || pvkb)**

The client sends the key identity string, and the configuration string to the server.

**C{ kid, cfg } 🡪 S**

**8.2 Connect Response:**

The server responds with either an error message, or a connect response packet. Any error during the key exchange will generate an error-packet sent to the remote host, which will trigger a tear down of the exchange, and the network connection on both sides.

The server first checks that it has the requested asymmetric signature verification key corresponding to that host using the key-identity array, then verifies that it has a compatible protocol configuration. The server stores a hash of the configuration string, key identity, and both public signature verification-keys, to create the public key hash, which is used as a session cookie.

**sch 🡨 H(cfg || kid || pvka || pvkb)**

The server then generates an asymmetric encryption key-pair, stores the private key, hashes the public encapsulation key, and then signs the hash of the public encapsulation key using the asymmetric signature key. The public signature verification key can itself be signed by a ‘chain of trust’ model, like X.509, using a signature verification extension to this protocol.

**pk,sk 🡨 AG(cfg)**

**pkh 🡨 H(pk)**

**spkh 🡨 ASsk(pkh)**

The server sends a connect response message containing a signed hash of the public asymmetric encapsulation-key, and a copy of that key.

**S{ spkh, pk } 🡪 C**

**8.3 Exchange Request:**

The client verifies the signature of the hash, then generates its own hash of the public key, and compares it with the one contained in the message. If the hash matches, the client uses the public-key to encapsulate a shared secret. If the hash does not match, the key exchange is aborted.

**cond 🡨 AVpk(H(pk)) := (true ?= pk : 0)**

**cpta, seca 🡨 AEpk(seca)**

The client stores the shared secret (**seca**), which along with a second shared secret and the session cookie, will be used to generate the session keys.

The client generates an asymmetric encryption key-pair, stores the private key, hashes the public encapsulation key and cipher-text, and then signs the hash using its asymmetric signature key.

**pk, sk 🡨 AG(cfg)**

**kch 🡨 H(pk || cpta)**

**skch 🡨 ASsk(kch)**

The client sends a response message containing the signed hash of its public asymmetric encapsulation-key and cipher-text, and a copy of the cipher-text and encapsulation key.

**C{ cpta, pk, skch } 🡪 S**

**8.4 Exchange Response:**

The server verifies the signature of the hash, then generates its own hash of the public key and cipher-text, and compares it with the one contained in the message. If the hash matches, the server uses the public-key to decapsulate the shared secret. If the hash comparison fails, the key exchange is aborted.

**cond 🡨 AVpk(H(pk || cpta)) = (true ?= cph : 0)**

The server decapsulates the second shared-secret, and stores the secret (**seca**).

**seca 🡨 -AEsk(cpta)**

The server generates a cipher-text and the second shared secret (**secb**) using the clients public encapsulation key.

**cptb, secb 🡨 AEpk(secb)**

The server combines both secrets and the session cookie to create two session keys, and two unique nonce, one for each channel of the communications stream.

**k1, k2, n1, n2 🡨 Exp(seca, secb, sch)**

The receive and transmit channel ciphers are initialized.

**cprrx(k1,n1)**

**cprtx(k2,n2)**

An optional tweak value can be added to the ciphers initialization function. The tweak is mixed with the key using the internal KDF function. The tweak can be a tertiary key provided by the server, or a hash of multiple keys from a list of trusted key holders.

**t 🡨 H(s1, s2, ..., sn)**

**cpr(k,n,t)**

The server then hashes the cipher-text, and signs the hash.

**cpth 🡨 H(cptb)**

**scph 🡨 ASsk(cpth)**

The server sends the signed hash of the cipher-text, and the cipher-text to the client.

**S{ scph, cptb } 🡪 C**

**8.5 Establish Request:**

The client verifies the signature of the hash, then generates its own hash of the cipher-text, and compares it with the one contained in the message. If the hash matches, the client decapsulates the shared secret (**secb**). If the hash comparison fails, the key exchange is aborted.

**cond 🡨 AVpk(H(cptb)) = (true ?= cptb : 0)**

**secb 🡨 -AEsk(cptb)**

The client combines both secrets and the session cookie to create the session keys, and two unique nonce, one for each channel of the communications stream.

**k1, k2, n1, n2 🡨 KDF(seca, secb, sch)**

The receive and transmit channel ciphers are initialized.

**cprrx(k2,n2)**

**cprtx(k1,n1)**

An optional tweak value can be added to the cipher’s initialization function. This tweak is mixed with the key using the internal key derivation function. This tweak can be a tertiary key provided by the server, or a hash of multiple keys from a list of trusted key holders;

**t 🡨 H(s1, s2, ..., sn)**

**cpr(k, n, t)**

The client sends an empty message with the **establish request** flag, indicating that both encrypted channels of the tunnel have been raised, and that the tunnel is in the operational state. In the event of an error, the client sends an error message to the server, aborting the exchange and terminating the connection on both hosts.

**C{ f } 🡪 S**

**8.6 Establish Response:**

Strictly speaking, this step isn’t required. If something has gone wrong in the final stage of the key exchange and the keys don’t match between hosts, the first message sent will fail symmetric authentication, and close the tunnel. But in terms of good design, the tunnel state should be confirmed. There may also be consequences to allowing the cipher’s MAC function to run on some message when the tunnel has not been confirmed established. So out of caution, we send back the empty packet with the flag set to established.

The server checks the packet flag for the operational status of the client. If the flag is set to **establish request**, the server sends an empty message back to the client with the **establish response** flag set. Otherwise the tunnel is in an error state indicated in the message, and the tunnel is torn down on both sides. The server sets the operational state to **session established**, and is now ready to process data.

**S{ f } 🡪 C**

**8.7 Establish Verify:**

The client checks the flag of the **establish response** packet sent by the server. If the flag is set to **establish response**, the client tunnel is established and in an operational state. Otherwise the tunnel is in an error state indicated by the message, and the tunnel is torn down on both sides. The client sets the operational state to **session established**, and is now ready to process data.

**8.8 Transmission:**

The host, client or server, transmitting a message, first serializes the packet header and adds it to the symmetric ciphers associated data parameter. The host then encrypts the message, updates the MAC function with the cipher-text, and appends a MAC code to the end of the cipher-text. All of this is done by using the RCS stream cipher’s AEAD and encryption functions.

The serialized packet header, including the message size, protocol flag, and sequence number, is added to the MAC state through the additional-data parameter of the authenticated stream cipher RCS. This unique data is added to the MAC function with every packet, along with the encrypted cipher-text.

**cpt 🡨 Ek(m)**

**mc 🡨 Mmk(sh, cpt)**

The packet is decrypted by serializing the packet header and adding it to the MAC state along with the cipher-text, then finalizing the MAC and comparing the output code with the code appended to the cipher-text. If the code matches, the cipher-text is decrypted, and the message passed up to the application. If this check fails, the decryption function returns false with an empty message array, and the error must be handled by the application.

**m 🡨 -Ek(cpt) = true ? m : 0**

# **QSMP Simplex Formal Description**

**Legend:**

**:=, !=, ?=** -Equality operators; equals, not equals, evaluate

🡨🡪 -Assignment operators

**C** -The client host, initiates the exchange

**S** -The server host, listens for a connection

**AG** -The asymmetric cipher key generator function

**-AEsk** -The asymmetric decapsulation function and secret key

**AEpk** -The asymmetric encapsulation function and public key

**ASsk** -Sign data with the secret signature key

**AVpk** -Verify a signature the public verification key

**cfg** -The protocol configuration string

**cond,** -A conditional statement

**cprrx** -A receive channels symmetric cipher instance

**cprtx** -A transmit channels symmetric cipher instance

**cpt** -The symmetric ciphers cipher-text

**cpta** -The asymmetric ciphers cipher-text

**-Ek** -The symmetric decryption function and key

**Ek** -The symmetric encryption function and key

**H** -The hash function (SHA3)

**k,mk** -A symmetric key

**KDF** -The key expansion function (SHAKE)

**kid** -The public keys unique identity array

**Mmk** -The MAC function and key (KMAC)

**pk,sk** -Asymmetric public and secret keys

**pvk** -Public signature verification key

**sch** -A hash of the configuration string and and asymmetric verification-keys

**sec** -The shared secret derived from asymmetric encapsulation and decapsulation

**spkh** -The signed hash of the asymmetric public encapsulation-key

**The Simplex Key Exchange Sequence**

**Preamble:**

The client first checks the expiration date of the servers public key, if the key is invalid, it queries the server for a new signature verification key. This mechanism should be a re-authentication session with the server, in which the new key is distributed over an encrypted channel, and the client verifies this key certificate, using whichever authentication authority or scheme has been implemented by the server and client software. For example, the servers public verification key can be signed by PGP, or through the standard PKI, or in an MPDC network, and verified through that mechanism.

When a client wants to connect to a server, it checks its public key cache, to see if a valid signature verification key exists for the server. It then sends a connection request packet to the server, containing the protocol string, and key id.

**9.1 Connect Request:**

The client sends a connection request with its configuration string, and asymmetric public signature key identity.

The key identity (kid) is a multi-part 16-byte address and key identification array, used to match the intended target to the corresponding key. The configuration string defines the cryptographic protocol set being used, these must be identical.

The client stores a hash of the configuration string, the key id, and of the servers public asymmetric signature verification-key, which is used as a session cookie during the exchange.

**sch 🡨 H(cfg || kid || pvk)**

The client sends the key identity string, and the configuration string to the server.

**C{ kid, cfg } 🡪 S**

**9.2 Connect Response:**

The server responds with either an error message, or a response packet. Any error during the key exchange will generate an error-packet sent to the remote host, which will trigger a tear down of the session, and network connection on both sides.

The server first checks that it has the requested asymmetric signature verification key corresponding to that host using the key-identity array, then verifies that it has a compatible protocol configuration. The server stores a hash of the configuration string, key id, and the public signature verification-key, to create the session cookie hash.

**sch 🡨 H(cfg || kid || pvk)**

The server then generates an asymmetric encryption key-pair, stores the private key, hashes the public encapsulation key, and then signs the hash of the public encapsulation key using the asymmetric signature key. The public signature verification key can itself be signed by a ‘chain of trust’ model, like X.509, using a signature verification extension to this protocol.

**pk, sk 🡨 AG(cfg)**

**pkh 🡨 H(pk)**

**spkh 🡨 ASsk(pkh)**

The server sends a connect response message containing a signed hash of the public asymmetric encapsulation-key, and a copy of that key.

**S{ spkh, pk } 🡪 C**

**9.3 Exchange Request:**

The client verifies the signature of the hash, then generates its own hash of the public key, and compares it with the one contained in the message. If the hash matches, the client uses the public-key to encapsulate a shared secret.

**cond 🡨 AVpk(H(pk)) = (true ?= pk : 0)**

**cpt, sec 🡨 AEpk(sec)**

The client combines the secret and the session cookie to create the session keys, and two unique nonce, one key-nonce pair for each channel of the communications stream.

**k1, k2, n1, n2 🡨 KDF(sec, sch)**

The receive and transmit channel ciphers are initialized.

**cprrx(k2,n2)**

**cprtx(k1,n1)**

The client sends the cipher-text to the server.

**C{ cpt } 🡪 S**

**9.4 Exchange Response:**

The server decapsulates the shared-secret.

**sec 🡨 -AEsk(cpt)**

The server combines the shared secret and the session cookie hash to create two session keys, and two unique nonce, one key-nonce pair for each channel of the communications stream.

**k1, k2, n1, n2 🡨 KDF(sec, sch)**

The receive and transmit channel ciphers are initialized.

**cprrx(k1,n1)**

**cprtx(k2,n2)**

The server sets the packet flag to **exchange response**, indicating that the encrypted channels have been raised, and sends the notification to the client. The server sets the operational state to **session established**, and is now ready to process data.

**S{ f } 🡪 C**

**9.5 Establish Verify:**

The client checks the flag of the **exchange response** packet sent by the server. If the flag is set to indicate an error state, the tunnel is torn down on both sides, otherwise the client tunnel is established and in an operational state. The client sets the operational state to **session established**, and is now ready to process data.

**9.6 Transmission:**

The host, client or server, transmitting a message, first serializes the packet header and adds it to the symmetric ciphers associated data parameter. The host then encrypts the message, updates the MAC function with the cipher-text, and appends a MAC code to the end of the cipher-text. All of this is done by using the RCS stream cipher’s AEAD and encryption functions.

The serialized packet header, including the message size, protocol flag, and sequence number, is added to the MAC state through the additional-data parameter of the authenticated stream cipher RCS. This unique data is added to the MAC function with every packet, along with the encrypted cipher-text.

**cpt 🡨 Ek(m)**

**mc 🡨 Mmk(sh, cpt)**

The packet is decrypted by serializing the packet header and adding it to the MAC state along with the cipher-text, then finalizing the MAC and comparing the output code with the code appended to the cipher-text. If the code matches, the cipher-text is decrypted, and the message passed up to the application. If this check fails, the decryption function returns false with an empty message array, and the error must be handled by the application.

**m 🡨 -Ek(cpt) = true ? m : 0**

**10:** **QSMP API**

**10.1 Definitions and Shared API**

**Header:**

qsmp.h

**Description:**

The qsmp header contains shared constants, types, and structures, as well as function calls common to both the QSMP server and client implementations.

**Structures:**

The **QSMP\_ERROR\_STRINGS** is a static string-array containing QSMP error descriptions, used in the error reporting functionality.

|  |  |
| --- | --- |
| Data Set | Purpose |
| QSMP\_ERROR\_STRINGS | A string array of readable error descriptions. |

Table 10.1a QSMP error strings.

The **QSMP\_CONFIG\_STRING** is a static string containing the readable QSMP configuration string.

|  |  |
| --- | --- |
| Data Set | Purpose |
| QSMP\_CONFIG\_STRING | The QSMP configuration string. |

Table 10.1b QSMP configuration string.

The **qsmp\_packet** contains the QSMP packet structure.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| flag | Uint8 | 0x08 | The packet flag |
| msglen | Uint32 | 0x20 | The packets message length |
| sequence | Uint64 | 0x40 | The packet sequence number |
| message | Uint8 Array | Variable | The packets message data |

Table 10.1c QSMP packet structure.

The **qsmp\_client\_key** contains the QSMP client key state.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| expiration | Uint64 | 0x40 | The expiration time, in seconds from epoch |
| config | Uint8 Array | Variable | The primitive configuration string |
| keyid | Uint8 Array | Variable | The key identity string |
| verkey | Uint8 Array | Variable | The asymmetric signatures verification-key |

Table 10.1d QSMP client key structure.

The **qsmp\_keep\_alive\_state** contains the QSMP keep alive state.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| target | Struct | Variable | The target host socket structure |
| etime | Uint64 | 0x40 | The keep alive epoch time |
| seqctr | Uint64 | 0x40 | The keep alive packet sequence number |
| recd | Boolean | 0x08 | The keep alive response received status |

Table 10.1e QSMP keep alive state structure.

**Enumerations:**

The **qsmp\_configuration** enumeration defines the cryptographic primitive configuration.

|  |  |
| --- | --- |
| Enumeration | Purpose |
| qsmp\_configuration\_none | No configuration was specified |
| qsmp\_configuration\_sphincs\_mceliece | The Sphincs+ and McEliece configuration |
| qsmp\_configuration\_dilithium\_kyber | The Dilithium and Kyber configuration |
| qsmp\_configuration\_dilithium\_ntru | The Dilithium and NTRU configuration |
| qsmp\_configuration\_falcon\_kyber | The Falcon and Kyber configuration |
| qsmp\_configuration\_falcon\_ntru | The Falcon and NTRU configuration |

Table 10.1f QSMP configuration enumeration.

The **qsmp\_errors** enumeration is a list of the QSMP error code values.

|  |  |
| --- | --- |
| Enumeration | Purpose |
| qsmp\_error\_none | No error was detected |
| qsmp\_error\_authentication\_failure | The symmetric cipher had an authentication failure |
| qsmp\_error\_bad\_keep\_alive | The keep alive check failed |
| qsmp\_error\_channel\_down | The communications channel has failed |
| qsmp\_error\_connection\_failure | The device could not make a connection to the remote host |
| qsmp\_error\_connect\_failure | The transmission failed at the KEX connection phase |
| qsmp\_error\_decapsulation\_failure | The asymmetric cipher failed to decapsulate the shared secret |
| qsmp\_error\_establish\_failure | The transmission failed at the KEX establish phase |
| qsmp\_error\_exstart\_failure | The transmission failed at the KEX exstart phase |
| qsmp\_error\_exchange\_failure | The transmission failed at the KEX exchange phase |
| qsmp\_error\_hash\_invalid | The public-key hash is invalid |
| qsmp\_error\_invalid\_input | The expected input was invalid |
| qsmp\_error\_invalid\_request | The packet flag was unexpected |
| qsmp\_error\_keep\_alive\_expired | The keep alive has expired with no response |
| qsmp\_error\_key\_expired | The QSMP public key has expired |
| qsmp\_error\_key\_unrecognized | The key identity is unrecognized |
| qsmp\_error\_packet\_unsequenced | The packet was received out of sequence |
| qsmp\_error\_random\_failure | The random generator has failed |
| qsmp\_error\_receive\_failure | The receiver failed at the network layer |
| qsmp\_error\_transmit\_failure | The transmitter failed at the network layer |
| qsmp\_error\_verify\_failure | The expected data could not be verified |
| qsmp\_error\_unknown\_protocol | The protocol string was not recognized |
| qsmp\_error\_accept\_fail | The socket accept function returned an error |
| qsmp\_error\_hosts\_exceeded | The server has run out of socket connections |
| qsmp\_error\_memory\_allocation | The server has run out of memory |
| qsmp\_error\_decryption\_ | The decryption authentication has failed |
| qsmp\_error\_keepalive\_timeout | The decryption authentication has failed |
| qsmp\_error\_ratchet\_fail | The ratchet operation has failed |

Table 10.1g QSMP errors enumeration.

The **qsmp\_flags** enum contains the QSMP packet flags.

|  |  |
| --- | --- |
| Enumeration | Purpose |
| qsmp\_flag\_none | No flag was specified |
| qsmp\_flag\_connect\_request | The QSMP key-exchange client connection request flag |
| qsmp\_flag\_connect\_response | The QSMP key-exchange server connection response flag |
| qsmp\_flag\_connection\_terminate | The connection is to be terminated |
| qsmp\_flag\_encrypted\_message | The message has been encrypted flag |
| qsmp\_flag\_exstart\_request | The QSMP key-exchange client exstart request flag |
| qsmp\_flag\_exstart\_response | The QSMP key-exchange server exstart response flag |
| qsmp\_flag\_exchange\_request | The QSMP key-exchange client exchange request flag |
| qsmp\_flag\_exchange\_response | The QSMP key-exchange server exchange response flag |
| qsmp\_flag\_establish\_request | The QSMP key-exchange client establish request flag |
| qsmp\_flag\_establish\_response | The QSMP key-exchange server establish response flag |
| qsmp\_flag\_keep\_alive\_request | The packet contains a keep alive request |
| qsmp\_flag\_remote\_connected | The remote host is connected flag |
| qsmp\_flag\_remote\_terminated | The remote host has terminated the connection |
| qsmp\_flag\_session\_established | The exchange is in the established state |
| qsmp\_flag\_session\_establish\_verify | The exchange is in the established verify state |
| qsmp\_flag\_unrecognized\_protocol | The protocol string is not recognized |
| qsmp\_flag\_asymmetric\_ratchet\_request | The host has received an asymmetric key ratchet request |
| qsmp\_flag\_symmetric\_ratchet\_request | The host has received a symmetric key ratchet request |
| qsmp\_flag\_transfer\_request | The host has received a transfer request |
| qsmp\_flag\_error\_condition | The connection experienced an error |

Table 10.1h QSMP flags enumeration.

**Constants:**

|  |  |  |
| --- | --- | --- |
| Constant Name | Value | Purpose |
| QSMP\_CONFIG\_DILITHIUM\_KYBER | N/A | Sets the asymmetric cryptographic primitive-set to Dilithium/Kyber |
| QSMP\_CONFIG\_DILITHIUM\_MCELIECE | N/A | Sets the asymmetric cryptographic primitive-set to Dilithium/McEliece |
| QSMP\_CONFIG\_DILITHIUM\_NTRU | N/A | Sets the asymmetric cryptographic primitive-set to Dilithium/NTRU |
| QSMP\_CONFIG\_SPHINCS\_MCELIECE | N/A | Sets the asymmetric cryptographic primitive-set to Sphincs+/McEliece |
| QSMP\_SERVER\_PORT | 0x1315 | The default server port address |
| QSMP\_CONFIG\_SIZE | 0x30 | The size of the protocol configuration string |
| QSMP\_CONFIG\_STRING | Variable | The QSMP cryptographic primitive configuration string |
| QSMP\_CIPHERTEXT\_SIZE | Variable | The byte size of the asymmetric cipher-text array |
| QSMP\_PRIVATEKEY\_SIZE | Variable | The byte size of the asymmetric cipher private-key array |
| QSMP\_PUBLICKEY\_SIZE | Variable | The byte size of the asymmetric cipher public-key array |
| QSMP\_SIGNKEY\_SIZE | Variable | The byte size of the asymmetric signature signing-key array |
| QSMP\_VERIFYKEY\_SIZE | Variable | The byte size of the asymmetric signature verification-key array |
| QSMP\_SIGNATURE\_SIZE | Variable | The byte size of the asymmetric signature array |
| QSMP\_PUBKEY\_ENCODING\_SIZE | Variable | The byte size of the encoded QSMP public-key |
| QSMP\_PUBKEY\_STRING\_SIZE | Variable | The string size of the serialized QSMP client-key structure |
| QSMP\_HASH\_SIZE | 0x20 | The size of the hash function output |
| QSMP\_HEADER\_SIZE | 0x13 | The QSMP packet header size |
| QSMP\_KEEPALIVE\_STRING | 0x14 | The keep alive string size |
| QSMP\_KEEPALIVE\_TIMEOUT | 0x18750 | The keep alive timeout in milliseconds (5 minutes) |
| QSMP\_KEYID\_SIZE | 0x10 | The QSMP key identity size |
| QSMP\_MACKEY\_SIZE | 0x20 |  |
| QSMP\_MACTAG\_SIZE | 0x20 | The size of the mac function output |
| QSMP\_SRVID\_SIZE | 0x08 | The QSMP server identity size |
| QSMP\_TIMESTAMP\_SIZE | 0x08 | The key expiration timestamp size |
| QSMP\_MESSAGE\_MAX | Variable | The maximum message size used during the key exchange |
| QSMP\_PKCODE\_SIZE | 0x20 | The size of the session token hash |
| QSMP\_PUBKEY\_DURATION\_DAYS | 0x223 | The number of days a public key remains valid |
| QSMP\_PUBKEY\_DURATION\_SECONDS | Variable | The number of seconds a public key remains valid |
| QSMP\_PUBKEY\_LINE\_LENGTH | 0x40 | The line length of the printed QSMP public key |
| QSMP\_SECRET\_SIZE | 0x20 | The size of the shared secret for each channel |
| QSMP\_SIGKEY\_ENCODED\_SIZE | Variable | The secret signature key size |
| QSMP\_SEQUENCE\_TERMINATOR | 0xFFFFFFFF | The sequence number of a packet that closes a connection |
| QSMP\_CONNECT\_REQUEST\_SIZE | Variable | The key-exchange connect stage request packet size |
| QSMP\_EXSTART\_REQUEST\_SIZE | Variable | The key-exchange exstart stage request packet size |
| QSMP\_EXCHANGE\_REQUEST\_SIZE | Variable | The key-exchange exchange stage request packet size |
| QSMP\_ESTABLISH\_REQUEST\_SIZE | Variable | The key-exchange establish stage request packet size |
| QSMP\_CONNECT\_RESPONSE\_SIZE | Variable | The key-exchange connect stage response packet size |
| QSMP\_EXCHANGE\_RESPONSE\_SIZE | Variable | The key-exchange exchange stage response packet size |
| QSMP\_ESTABLISH\_RESPONSE\_SIZE | Variable | The key-exchange establish stage response packet size |

Table 10.1i QSMP constants.

The **qsmp\_connection\_state** contains the QSMP connection state.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| target | Struct | 0x440 | The target host socket structure |
| rxcpr | Struct | Variable | The receive channel cipher state |
| txcpr | Struct | Variable | The transmit channel cipher state |
| rxseq | Uint64 | 0x40 | The receive channels packet sequence number |
| txseq | Uint64 | 0x40 | The transmit channels packet sequence number |
| instance | Uint32 | 0x20 | The connections instance count |
| exflag | Uint8 | 0x08 | The KEX position flag |
| rtcs | Uint8 | 0x40 | The ratchet key |
| receiver | bool | 0x08 | The hosts receiver status |
| mode | enum | 0x08 | The QSMP mode |

Table 10.1j QSMP connection state structure.

**API:**

**Asymmetric Ratchet**

Run the asymmetric ratchet and update the session keys (duplex mode).

void qsmp\_duplex\_send\_asymmetric\_ratchet\_request (qsmp\_connection\_state\* cns)

**Symmetric Ratchet**

Run the symmetric ratchet and update the session keys (duplex mode).

void qsmp\_duplex\_send\_symmetric\_ratchet\_request (qsmp\_connection\_state\* cns)

**Connection Close**

Close the network connection between hosts.

void qsmp\_connection\_close(qsmp\_connection\_state\* cns, qsmp\_errors err, bool notify)

**Decode Public Key**

Decode a public key string and populate a client key structure.

void qsmp\_decode\_public\_key(qsmp\_client\_key\* pubk, const char enck[QSMP\_PUBKEY\_STRING\_SIZE])

**Encode Public Key**

Encode a public key structure and copy to a string.

void qsmp\_encode\_public\_key(char enck[QSMP\_PUBKEY\_STRING\_SIZE]), const qsmp\_client\_key\* pubk)

**Deserialize Signature Key**

Decode a secret signature key structure and copy to an array.

void qsmp\_deserialize\_signature\_key(qsmp\_server\_key\* prik, const uint8\_t serk[QSMP\_SIGKEY\_ENCODED\_SIZE])

**Serialize Signature Key**

Encode a secret key structure and copy to a string.

void qsmp\_serialize\_signature\_key(uint8\_t serk[QSMP\_SIGKEY\_ENCODED\_SIZE]), const qsmp\_server\_key\* prik)

**Connection Dispose**

Reset the connection state.

void qsmp\_connection\_close(qsmp\_connection\_state\* cns)

**Decrypt Packet**

Decrypt a message and copy it to the message output.

qsmp\_errors qsmp\_decrypt\_packet(qsmp\_connection\_state\* cns, uint8\_t\* message, size\_t\* msglen, const qsmp\_packet\* packetin)

**Encrypt Packet**

Encrypt a message and copy it to a packet.

qsmp\_errors qsmp\_decrypt\_packet(qsmp\_connection\_state\* cns, qsmp\_packet\* packetout, const uint8\_t\* message, size\_t\* msglen)

**Generate Key Pair**

Generate a QSMP key-pair; generates the public and private asymmetric signature keys.

void qsmp\_generate\_keypair(qsmp\_client\_key\* pubkey, qsmp\_server\_key\* prikey, const uint8\_t keyid[QSMP\_KEYID\_SIZE])

**Packet Clear**

Clear a packet's state, resetting the structure to zero.

void qsmp\_packet\_clear(qsmp\_packet\* packet)

**Error To String**

Return a pointer to a string description of an error code.

const char\* qsmp\_error\_to\_string(qsmp\_errors error)

**Error Message**

Populate a packet structure with an error message.

void qsmp\_packet\_error\_message(qsmp\_packet\* packet, qsmp\_errors error)

**Header Deserialize**

Deserialize a byte array to a packet header.

void qsmp\_packet\_header\_deserialize(const uint8\_t\* header, qsmp\_packet\* packet)

**Header Serialize**

Serialize a packet header to a byte array.

void qsmp\_packet\_header\_serialize(const qsmp\_packet\* packet, uint8\_t\* header)

**Log Error**

Log the message, socket error, and string description.

void qsmp\_log\_error(const qsmp\_messages emsg, qsc\_socket\_exceptions err, const char\* msg)

**Log Message**

Log the message.

void qsmp\_log\_message(const qsmp\_messages emsg)

**Log Write**

Log the message, and string description.

void qsmp\_log\_write(const qsmp\_messages emsg, const char\* msg)

**Packet Clear**

Clear a packet's state .

size\_t qsmp\_packet\_clear(const qsmp\_packet\* packet)

**Packet To Stream**

Serialize a packet to a byte array.

size\_t qsmp\_packet\_to\_stream(const qsmp\_packet\* packet, uint8\_t\* pstream)

**Stream To Packet**

Deserialize a byte array to a packet.

void qsmp\_stream\_to\_packet(const uint8\_t\* pstream, qsmp\_packet\* packet)

**10.2 Server API**

**Header:**

qsmpserver.h

**Description:**

Functions used to implement the QSMP server.

**Structures:**

The **qsmp\_server\_key** contains the QSMP server key structure.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| expiration | Uint64 | 0x40 | The expiration time, in seconds from epoch |
| config | Uint8 Array | 0x180 | The primitive configuration string |
| keyid | Uint8 Array | 0x80 | The key identity string |
| sigkey | Uint8 Array | Variable | The asymmetric signature signing-key |
| verkey | Uint8 Array | Variable | The asymmetric signature verification-key |

Table 10.2a QSMP key structure.

**API:**

**Broadcast**

Broadcast a message to all connected hosts.

void qsmp\_server\_broadcast(const uint8\_t\* message, size\_t msglen)

**Pause**

Pause the server, suspending new joins.

void qsmp\_server\_pause()

**Quit**

Quit the server, closing all connections.

void qsmp\_server\_quit()

**Resume**

Resume the server listener function from a paused state.

void qsmp\_server\_resume()

**Listen IPv4**

Run the IPv4 networked key exchange function. Returns the connected socket and the QSMP server connection state.

qsmp\_errors qsmp\_server\_listen\_ipv4(qsmp\_server\_key\* prik, void (\*receive\_callback)( qsmp\_ server\_connection\_state\*, const char\*, size\_t)

**Listen IPv6**

Run the IPv6 networked key exchange function. Returns the connected socket and the QSMP server state.

qsmp\_errors qsmp\_server\_listen\_ipv6(qsmp\_server\_key\* prik, void (\*receive\_callback)( qsmp\_ server\_connection\_state\*, const char\*, size\_t)

**10.3 Client API**

**Header:**

qsmpclient.h

**Description:**

Functions used to implement the QSMP client.

**Structures:**

The **qsmp\_kex\_client\_state** contains the QSMP server state structure.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| rxcpr | RCS state | Variable | The receive channel cipher state |
| txcpr | RCS state | Variable | The transmit channel cipher state |
| config | Uint8 Array | 0x180 | The primitive configuration string |
| keyid | Uint8 Array | 0x80 | The key identity string |
| pkhash | Uint8 Array | 0x20 | The session token hash |
| prikey | Uint8 Array | Variable | The asymmetric cipher private key |
| pubkey | Uint8 Array | Variable | The asymmetric cipher public key |
| mackey | Uint8 Array | 0x20 | The intermediate mac key |
| token | Uint8 Array | 0x100 | The session token |
| verkey | Uint8 Array | Variable | The asymmetric signature verification-key |
| exflag | enum | qsmp\_flags | The KEX position flag |
| expiration | Uint64 | 0x40 | The expiration time, in seconds from epoch |
| rxseq | Uint64 | 0x40 | The receive channels packet sequence number |
| txseq | Uint64 | 0x40 | The transmit channels packet sequence number |

Table 10.3a QSMP client state structure.

**API:**

**Decode Public Key**

Decode a public key string and populate a client key structure.

bool qsmp\_client\_decode\_public\_key(qsmp\_client\_key\* clientkey, const char input[QSMP\_PUBKEY\_STRING\_SIZE])

**Decode Public Key**

Decode a public key string and populate a client key structure.

bool qsmp\_client\_decode\_public\_key(qsmp\_client\_key\* clientkey, const char input[QSMP\_PUBKEY\_STRING\_SIZE])

**Send Error**

Send an error code to the remote host.

void qsmp\_client\_send\_error(const qsc\_socket\* sock, qsmp\_errors error)

**Send Error**

Run the IPv4 networked key exchange function. Returns the connected socket and the QSMP server state.

qsmp\_errors qsmp\_client\_connect\_ipv4(qsmp\_kex\_client\_state\* ctx, qsc\_socket\* sock, const qsmp\_client\_key\* ckey, const qsc\_ipinfo\_ipv4\_address\* address, uint16\_t port)

**Connect IPv4**

Run the IPv4 networked key exchange function. Returns the connected socket and the QSMP client state.

qsmp\_errors qsmp\_client\_connect\_ipv4(qsmp\_kex\_client\_state\* ctx, qsc\_socket\* sock, const qsmp\_client\_key\* ckey, const qsc\_ipinfo\_ipv4\_address\* address, uint16\_t port)

**Connect IPv6**

Run the IPv6 networked key exchange function. Returns the connected socket and the QSMP client state.

qsmp\_errors qsmp\_client\_connect\_ipv6(qsmp\_kex\_client\_state\* ctx, qsc\_socket\* sock, const qsmp\_client\_key\* ckey, const qsc\_ipinfo\_ipv6\_address\* address, uint16\_t port)

**Connection Close**

Close the remote session and dispose of resources.

void qsmp\_client\_connection\_close(qsmp\_kex\_client\_state\* ctx, const qsc\_socket\* sock, qsmp\_errors error)

**Decrypt Packet**

Decrypt a message and copy it to the message output.

qsmp\_errors qsmp\_client\_decrypt\_packet(qsmp\_kex\_client\_state\* ctx, const qsmp\_packet\* packetin, uint8\_t\* message, size\_t\* msglen)

**Encrypt Packet**

Encrypt a message and build an output packet.

qsmp\_errors qsmp\_client\_encrypt\_packet(qsmp\_kex\_client\_state\* ctx, const uint8\_t\* message, size\_t msglen, qsmp\_packet\* packetout)

# **11:** **Design Decisions**

QSMP is built upon the Transport Control Protocol (TCP) in the accompanying example code, but networking protocol choices should be considered as operating at a layer beneath the QSMP protocol. QSMP is an authenticated key exchange and secure communication protocol, it may use TCP, UDP, or a custom IP stack to transport packets. Future revisions of the protocol implementation may use a custom IP stack, implement windowing controls, packet buffers, and other custom networking controls as best suits implementation-specific requirements. However, many widely used VPN software implementations currently use TCP, and forego the complexities of a custom-built IP stack, and keeping the implementation relatively simple and straight-forward was a chief goal of the example project, to lend clarity to a somewhat complex network security protocol. Future implementation could introduce a more complex networking implementation, one that offers more granularity in the network application of the QSMP protocol.

QSMP does not currently use protocol negotiation, this is for several reasons. Though trivial to implement, and that QSMP currently has several implementation choices, protocol negotiation is too often misused to ‘dumb down’ a security scheme to the cheapest possible combination of security protocols. It also adds extra messaging overhead to the key negotiation. QSMP currently supports six asymmetric configurations: Dilithium-Kyber, Dilithium-McEliece, Dilithium-NTRU, Falcon-NTRU, Falcon-McEliece, and SphincsPlus-McEliece. More parameter sets can be added in the future, and other asymmetric primitives may also be added, but the benefit of adding protocol negotiation is limited, and not necessary in most of the intended implementation use-cases.

QSMP does not implement signature chaining directly, but this is a feature that can either be added, or implemented using a secondary protocol implementation like X.509. It is not though, a specific feature of the design, as QSMP is primarily intended as a standalone secure messaging protocol. We do believe that in cases where this extra layer of authentication is warranted by the implementation, that signature chaining can be a means to add some extra assurance to the key-exchange authentication. Public keys can be distributed in X.509 format or other ‘web of trust’ mechanism, and the authentication chain checked in an extra step, with the primary public key extracted and passed to the QSMP client.

QSMP packet headers were designed to be compact, less than half the size of the standard SSH-2 protocol at just 13 bytes. Unnecessary fields are omitted, and integer sizes are kept within ranges of reasonable expected use, such as flags taking up just a single byte, and the message data size parameter a 32-bit integer. In a custom IP stack implementation, this can translate to a small overall packet header size, making applications of the protocol that send small amounts of data in ‘real time’ processing applications such as Telnet, more efficient. An application should never require more than 255 flag members, and a payload size should never exceed 4 gigabytes of message data in a single packet payload, so we feel these are more conservative and realistic uses of packet header space.

We use a 2-channel communications system, with each channel keyed separately, this is to fulfill the purpose of what this protocol represents; a high-security, post-quantum, communications protocol. Other VPN implementations use a single shared secret, derived to key transmit and receive symmetric ciphers on both sides of the communications stream. They take security ‘shortcuts’, preferring small gains in performance over the overall security of the design. We feel that in order to be truly secure, each host, client and server, must generate the key for the channel that they transmit data over, anything less is a compromise to the security in the design. Further, we use two independent authenticated key exchanges, using post-quantum asymmetric ciphers to encapsulate two unique shared secrets. We are well aware of ‘shortcuts’ used by other protocols that could reduce this to a single asymmetric key exchange, but do not feel that this offers the best possible security guarantee, and the goal of QSMP, is to provide strong, uncompromising security to a communications stream.

We use a post-quantum authenticated stream cipher; RCS. This cipher’s transform is based on the wide-block form of the Rijndael cipher (AES), with a hash-based key schedule and strong authentication using KMAC. We believe authenticated stream ciphers are the future of symmetric ciphers, and that this cipher which uses cSHAKE to generate round keys, KMAC for authentication, and increased transformation rounds from 14 with AES, to 22 with RCS, provides a realistic post-quantum symmetric security. There are those that would urge us to use AES or ChaCha for another twenty years, until it is proven beyond doubt that they have been broken, but we do not think this is wise, as the powerful agencies that work relentlessly towards breaking the worlds cryptography, do not publish those discoveries in scientific journals, so we choose to use stronger primitives now as a precaution, and a better guarantee of true long-term security in the coming quantum age.

QSMP was designed to be secure, not just in the present day, but in a future which promises incredible advances in computing technology, advances that can not now be fully known, and must not be underestimated. It is designed for the purpose of keeping sensitive data safe, now and for decades to come.