Quantum Secure Messaging Protocol – QSMP 1.3

Revision 1.3a, December 08, 2024

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

This document is an engineering level description of the QSMP 1.3 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, an explanation of its design, as well as references to its component primitives and supporting documentation.

Contents **Page**

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 third 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 implementation of the QSMP specification.

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

Figures

Contents

Page

Tables

Contents

1: Introduction

Key exchange protocols are foundational components in secure networking today, with prominent examples found in protocols like TLS, PGP, and SSH. These protocols establish methods for securely exchanging secret keys between devices. Typically, a key exchange function is part of a more extensive process that includes authentication both during and after the key exchange, and establishes an encrypted tunnel that uses the shared secret to secure traffic flows using symmetric ciphers.

QSMP provides a comprehensive framework that encompasses key exchange, authentication mechanisms, and encrypted tunnel creation. While existing protocols can be modified to incorporate quantum-resistant cryptographic primitives, QSMP takes a different approach by designing an entirely new set of mechanisms tailored for performance and security in a post-quantum environment. Recognizing the inevitable transition to post-quantum cryptography, QSMP was built from the ground up, avoiding the constraints of backward compatibility and the complexity associated with older protocol artifacts, versioning, and legacy APIs.

As a quantum-secure messaging protocol, QSMP leverages state-of-the-art asymmetric encryption and signature schemes alongside a post-quantum strength symmetric cipher. The current version supports Kyber or McEliece as asymmetric ciphers, with Dilithium or Sphincs+ as signature schemes.

For symmetric encryption, QSMP employs the RCS authenticated stream cipher, which is based on the wide-block (256-bit state size) version of the Rijndael cipher. This cipher features increased rounds, a cryptographically secure key schedule, and AEAD (Authenticated Encryption with Associated Data) authentication using KMAC. Designed to be both flexible and post-quantum secure, QSMP was designed to surpass the protocols it intends to replace and is suitable for any application that demands strong post-quantum security in an encrypted messaging scheme.

QSMP includes two protocol variants: SIMPLEX and DUPLEX.

SIMPLEX Protocol: The SIMPLEX protocol is a streamlined one-way-trust authenticated key exchange designed for client-server communications. In this unidirectional trust model, the client trusts the server. The server authenticates by signing its public asymmetric cipher key, which the client then verifies using the server’s public signature-verification key. This protocol creates a 256-bit secure, duplexed encrypted tunnel between the server and client in just two round trips, making it ideal for applications requiring efficient, post-quantum secure encrypted channels between a client and server.

DUPLEX Protocol: The DUPLEX protocol supports a bidirectional trust model, where two hosts mutually authenticate and exchange shared secrets. Each host possesses the other's public signature-verification key. They exchange signed public asymmetric cipher keys, signed ciphertext, create individual shared secrets, and combine these secrets to create a 512-bit secure encrypted communication stream. The QSMP DUPLEX protocol is designed for high-security, post-quantum communication between remote hosts. It can also work in conjunction with

SIMPLEX to facilitate host registration, distribute public signature keys, and establish secure communications within high-security environments.

The protocols within QSMP are versatile and adaptable to various use cases, offering modern alternatives to aging cryptographic protocols that are merely being retrofitted with quantum-resistant algorithms.

1.1 Purpose

Ephemeral Asymmetric Cipher Keys: The protocol ensures that asymmetric cipher keys are used for just a single transfer of shared secrets, encapsulating shared secrets unique to each session. This approach provides strong forward secrecy, guaranteeing that the compromise of current asymmetric cipher keys does not affect the security of previous sessions.

Predictive Resistance: The capture of any shared keys within a session reveals no information about future sessions, preventing adversaries from predicting or deriving future keys based on past communications.

One-Way or Two-Way Authentication: QSMP supports both one-way and two-way authenticated trust models, utilizing robust asymmetric and symmetric authentication methods to establish secure and verifiable communication channels between parties.

Post-Quantum Security: The protocol is designed to be resistant to quantum attacks by using quantum-safe cryptographic primitives such as Kyber and McEliece for key exchanges, and Dilithium or Sphincs+ for digital signatures, ensuring long-term data security.

Scalable Encryption: QSMP utilizes the RCS stream cipher with AEAD authentication using KMAC, which is based on the wide-block Rijndael cipher with increased rounds and a cryptographically strong key schedule (cSHAKE). This ensures scalability and adaptability for high-throughput environments, leveraging embedded CPU level AES-NI instructions, and maintaining robust encryption with superior quantum-level security.

Flexible Protocol Variants: QSMP provides two protocol variants, SIMPLEX and DUPLEX, to cater to different communication needs. SIMPLEX supports a streamlined one-way authenticated key exchange ideal for client-server exchanges, while DUPLEX offers a bidirectional trust model suitable for high-security communication between hosts.

Efficient Key Exchange: The SIMPLEX protocol is optimized to establish secure communication channels with minimal round trips, reducing latency and computational overhead in the key exchange process. SIMPLEX achieves a two-way encrypted tunnel in just two round trips.

Comprehensive Anti-Attack Strategy: The key exchange incorporates many different defenses against packet and message tampering, impersonation, replay, memory overflow, authentication, and cryptographic attacks against the key exchange and encrypted tunnel.

Interoperability: QSMP can be seamlessly integrated into existing network architectures and communication systems, offering an upgrade path to quantum-resistant security without the need to overhaul legacy infrastructure.

Multi-Layered Cryptographic Security: The protocol combines multiple layers of cryptographic techniques, including digital signatures, asymmetric encryption, and authenticated symmetric encryption, to provide comprehensive protection against a wide range of threats, including CCA, CPA, man-in-the-middle (MITM) attacks, and replay attacks.

Robust Error Handling: QSMP includes well-defined error detection and correction mechanisms to ensure communication integrity, with clear error messages and automated session tear-down procedures in the event of protocol violations or security issues.

Future-Proof Design: The protocol is designed with the modular flexibility to adopt new cryptographic algorithms and techniques as they emerge, ensuring that it remains secure against evolving threats and advancements in cryptographic research for many years to come.

These features make QSMP a highly secure, efficient, and future-ready protocol for establishing encrypted communication channels in environments that require strong post-quantum security.

2: Scope

This document provides a comprehensive description of the QSMP secure messaging protocols, focusing on establishing encrypted and authenticated communication channels between two hosts. It outlines the complete processes involved in key exchange, message authentication, and the establishment of secure communication tunnels using both the QSMP SIMPLEX and DUPLEX protocols.

2.1 Application

QSMP is designed primarily for institutions and organizations that require secure communication channels to handle sensitive information exchanged between remote devices. It is ideally suited for sectors where data confidentiality, integrity, and authenticity are paramount, including financial institutions, government agencies, defense contractors, and enterprises managing critical infrastructure.

The protocol is versatile enough to be applied in various settings, such as secure messaging, VPNs, and other network communication systems where robust encryption and authentication are essential. QSMP's design ensures that even if the cryptographic landscape changes due to advancements in quantum computing, its security framework remains resilient and flexible.

Mandatory Protocol Components:

The key exchange, message authentication, and encryption functions defined in this document are integral to the construction of a QSMP communication stream. These components MUST be implemented to ensure secure operations and protocol compliance.

Use of Keywords for Compliance:

- **SHOULD**: Indicates best practices or recommended settings that are not compulsory but are strongly advised for optimal performance and security.
- **SHALL**: Denotes mandatory requirements that must be followed to ensure full compliance with the QSMP protocol. Deviations from these guidelines result in non-conformity and may compromise the protocol's effectiveness.

2.2 Protocol Flexibility and Use Cases

QSMP is engineered to be highly adaptable, supporting various deployment scenarios ranging from simple client-server architectures to more complex multi-party distributed systems. This flexibility makes it ideal for cloud-based infrastructures, secure messaging applications, VPNs, and IoT networks that demand high-performance encryption and authentication.

Key use cases for QSMP include:

- **Institutional Communications**: Securely encrypting and authenticating sensitive data exchanges between financial institutions, government agencies, and corporate networks.

- **Internet of Things (IoT)**: Enabling secure communication for connected devices that require lightweight, efficient, and scalable encryption protocols to protect data integrity.
- **Secure Messaging Platforms**: Providing end-to-end encryption for messaging services that need to resist both classical and quantum attacks.

The protocol's ability to integrate with existing network infrastructure without requiring extensive modifications ensures that organizations can transition to post-quantum security seamlessly while maintaining high levels of operational efficiency.

2.3 Compliance and Interoperability

The QSMP protocol is designed to maintain strict compliance with its core cryptographic principles and industry standards while ensuring interoperability with other secure communication frameworks. To guarantee that different QSMP implementations can interact securely, adherence to the standards outlined in this document is crucial.

To facilitate future upgrades and adaptations, QSMP is structured to support modular cryptographic components. This approach allows for the addition of new cryptographic primitives or the enhancement of existing ones without disrupting the overall architecture. As new advancements in cryptographic techniques emerge, QSMP can be easily updated to include these innovations, maintaining its position as a state-of-the-art security protocol.

Key elements of compliance:

- **Interoperability Standards**: QSMP is developed to work seamlessly with other post-quantum cryptographic standards, ensuring that its communication channels can operate in diverse network environments.
- **Modular Design**: The protocol's flexible design allows for straightforward upgrades, facilitating the incorporation of future cryptographic advancements with minimal impact on existing deployments.
- **Powerful Building Block**: QSMP is an ideal choice as the core encrypted tunneling and networking components when constructing new protocols. It uses a simple interface that wraps the complex interactions governing a key exchange, authentication, and an encrypted tunnel. A sophisticated multi-threaded networked server can be implemented with just a few function calls, and a client implementation is similarly simple to employ, with a simple intuitive API, that makes the complex operations beneath them transparent to the implementor. New protocols can easily be built on top of QSMP. The two variants lend to the client-server model and peer-to-peer models of network communications, providing a ‘black box’ interface for constructing more complex protocols on top of QSMP.

2.4 Recommendations for Secure Implementation

In addition to outlining the core requirements for QSMP's secure communications model, this document provides best practice recommendations to enhance implementation security, performance, and reliability:

- **Regular Cryptographic Updates**: Institutions are advised to keep updated on developments in post-quantum cryptography and to update their cryptographic algorithms to maintain compliance with industry standards.
- **Security Audits and Assessments**: Routine security assessments should be conducted to identify potential vulnerabilities in the protocol implementation and to apply necessary mitigations.
- **Infrastructure Optimization**: It is recommended to configure network infrastructure in a way that supports QSMP's low-latency, high-throughput capabilities, ensuring that performance remains consistent even under heavy loads.

These guidelines aim to help organizations maximize QSMP's security potential, ensuring that their communication channels remain secure against both current and future threats.

2.5 Document Organization

This document is structured to provide a detailed, logical flow of information about the QSMP protocol's operation and implementation. It includes the following key sections:

- **Cryptographic Primitives**: Detailed explanations of the mathematical algorithms that form the foundation of QSMP's encryption and authentication processes.
- **Key Exchange Mechanisms**: Comprehensive breakdowns of how session keys are established securely through QSMP's SIMPLEX and DUPLEX protocols.
- **Message Authentication**: Detailed descriptions of the techniques used to verify the authenticity and integrity of messages exchanged within QSMP communications.
- **Protocol Flows**: Visual diagrams and cryptographic pseudocode outlining the message flows for both SIMPLEX and DUPLEX, illustrating the secure handshake processes and key exchanges.
- **Error Handling and Fault Tolerance**: Guidelines on how to manage protocol errors and disruptions while maintaining secure and stable communication channels.
- **Implementation Examples**: Practical examples, code snippets, and detailed use cases demonstrating the integration of QSMP in various application contexts.

3.Terms and Definitions

3.1 Cryptographic Primitives

3.1.1 Kyber

The Kyber asymmetric cipher and NIST Post Quantum Competition winner.

3.1.2 McEliece

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

3.1.3 Dilithium

The Dilithium asymmetric signature scheme and NIST Post Quantum Competition winner.

3.1.5 SPHINCS+

The SPHINCS+ asymmetric signature scheme and NIST Post Quantum Competition winner.

3.1.6 RCS

The wide-block Rijndael hybrid authenticated symmetric stream cipher.

3.1.7 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.

3.1.8 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.

3.1.9 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.

3.2 Network References

3.2.1 Bandwidth

The maximum rate of data transfer across a given path, measured in bits per second (bps).

3.2.2 Byte

Eight bits of data, represented as an unsigned integer ranged 0-255.

3.2.3 Certificate

A digital certificate, a structure that contains a signature verification key, expiration time, and serial number and other identifying information. A certificate is used to verify the authenticity of a message signed with an asymmetric signature scheme.

3.2.4 Domain

A virtual grouping of devices under the same authoritative control that shares resources between members. Domains are not constrained to an IP subnet or physical location but are a virtual group of devices, with server resources typically under the control of a network administrator, and clients accessing those resources from different networks or locations.

3.2.5 Duplex

The ability of a communication system to transmit and receive data; half-duplex allows one direction at a time, while full-duplex allows simultaneous two-way communication.

3.2.6 Gateway: A network point that acts as an entrance to another network, often connecting a local network to the internet.

3.2.7 IP Address

A unique numerical label assigned to each device connected to a network that uses the Internet Protocol for communication.

3.2.8 IPv4 (Internet Protocol version 4): The fourth version of the Internet Protocol, using 32-bit addresses to identify devices on a network.

3.2.9 IPv6 (Internet Protocol version 6): The most recent version of the Internet Protocol, using 128-bit addresses to overcome IPv4 address exhaustion.

3.2.10 LAN (Local Area Network)

A network that connects computers within a limited area such as a residence, school, or office building.

3.2.11 Latency

The time it takes for a data packet to move from source to destination, affecting the speed and performance of a network.

3.2.12 Network Topology

The arrangement of different elements (links, nodes) of a computer network, including physical and logical aspects.

3.2.13 Packet

A unit of data transmitted over a network, containing both control information and user data.

3.2.14 Protocol

A set of rules governing the exchange or transmission of data between devices.

3.2.15 TCP/IP (Transmission Control Protocol/Internet Protocol)
A suite of communication protocols used to interconnect network devices on the internet.

3.2.16 Throughput: The actual rate at which data is successfully transferred over a communication channel.

3.2.17 UDP (User Datagram Protocol)
A communication protocol that offers a limited amount of service when messages are exchanged between computers in a network that uses the Internet Protocol.

3.2.18 VLAN (Virtual Local Area Network)
A logical grouping of network devices that appear to be on the same LAN regardless of their physical location.

3.2.19 VPN (Virtual Private Network)
Creates a secure network connection over a public network such as the internet.

3.3 Normative References

The following documents serve as references for cryptographic components used by QSTP:

3.3.1 FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable Output Functions: This standard specifies the SHA-3 family of hash functions, including SHAKE extendable-output functions. https://doi.org/10.6028/NIST.FIPS.202

3.3.2 FIPS 203: Module-Lattice-Based Key Encapsulation Mechanism (ML-KEM): This standard specifies ML-KEM, a key encapsulation mechanism designed to be secure against quantum computer attacks. https://doi.org/10.6028/NIST.FIPS.203

3.3.3 FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA): This standard specifies ML-DSA, a set of algorithms for generating and verifying digital signatures, believed to be secure even against adversaries with quantum computing capabilities. https://doi.org/10.6028/NIST.FIPS.204

3.3.4 NIST SP 800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash: This publication specifies four SHA-3-derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. https://doi.org/10.6028/NIST.SP.800-185

3.3.5 NIST SP 800-90A Rev. 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators: This publication provides recommendations for the generation of random numbers using deterministic random bit generators. https://doi.org/10.6028/NIST.SP.800-90Ar1

3.3.6 NIST SP 800-108: Recommendation for Key Derivation Using Pseudorandom Functions: This publication offers recommendations for key derivation using pseudorandom functions. https://doi.org/10.6028/NIST.SP.800-108

3.3.7 FIPS 197: The Advanced Encryption Standard (AES): This standard specifies the Advanced Encryption Standard (AES), a symmetric block cipher used widely across the globe. https://doi.org/10.6028/NIST.FIPS.197

4: Cryptographic Primitives

QSMP relies on a set of cryptographic primitives designed to provide resilience against both classical and quantum-based attacks. The following sections detail the specific cryptographic algorithms and mechanisms that form the foundation of QSMP's encryption, key exchange, and authentication processes.

4.1 Asymmetric Cryptographic Primitives

QSMP employs post-quantum secure asymmetric algorithms to ensure the integrity and confidentiality of key exchanges, as well as to facilitate digital signatures and asymmetric key exchanges. The primary asymmetric primitives used are:

- **Kyber**: An IND-CCA secure lattice-based key encapsulation mechanism that provides secure and efficient key exchange resistant to quantum attacks. Kyber is valued for its balance between computational speed and cryptographic strength, making it suitable for scenarios requiring rapid key generation and exchange.
- **McEliece**: A code-based cryptosystem that remains one of the most established and trusted post-quantum algorithms. It leverages the difficulty of decoding general linear codes, offering a high level of security even against advanced quantum decryption techniques.
- **Dilithium**: A lattice-based digital signature scheme based on that of the underlying MLWE and MSIS problems, that offers fast signing and verification while maintaining strong security guarantees against quantum attacks.
- **Sphincs+**: A stateless hash-based signature scheme, which provides long-term security without reliance on specific problem structures, making it robust against future advancements in cryptographic research.

These asymmetric primitives are selected for their proven resilience against quantum cryptanalysis, ensuring that QSMP's key exchange and signature operations remain secure in the face of evolving computational threats.

4.2 Symmetric Cryptographic Primitives

QSMP's symmetric encryption employs the **Rijndael Cryptographic Stream (RCS)**, a stream cipher adapted from the Rijndael (AES) symmetric cipher to meet post-quantum security needs. Key features of the RCS cipher include:

- **Wide-Block Cipher Design**: RCS extends the original AES design with a focus on increasing the block size (from 128 to 256 bits) and number of transformation rounds (from 14 to 21 for a 256-bit key, and 30 rounds for a 512-bit key), thereby enhancing its resistance to differential and linear cryptanalysis.
- **Enhanced Key Schedule**: The key schedule in RCS is cryptographically strong using Keccak (cSHAKE), ensuring that derived keys are resistant to known attacks, including algebraic-based and differential attacks. RCS replaces Rijndael’s cryptographically-weak key schedule, with a tweakable post-quantum secure key expansion function.

- **Authenticated Encryption with Associated Data (AEAD)**: RCS integrates with KMAC (Keccak-based Message Authentication Code) to provide both encryption and message authentication in a single operation. This approach ensures that data integrity is maintained alongside confidentiality. Additional data can be added to the MAC function, to ensure the integrity of non-encrypted messaging components such as packet headers.

The RCS stream cipher's design is optimized for high-performance environments, making it suitable for low-latency applications that require secure and efficient data encryption. It leverages AVX/AVX2/AVX512 intrinsic functions, and AES-NI instructions embedded in modern CPUs.

4.3 Hash Functions and Key Derivation

Hash functions and key derivation functions (KDFs) are essential to QSMP's ability to transform raw cryptographic data into secure keys and hashes. The following primitives are used:

- **SHA-3**: SHA-3 serves as QSMP's primary hash function, providing secure, collision-resistant hashing capabilities.
- **SHAKE**: QSMP employs the Keccak SHAKE XOF function for deriving symmetric keys from shared secrets. This ensures that each session key is uniquely generated and unpredictable, enhancing the protocol's security against key reuse attacks.
- **KMAC**: The SHA-3 keyed hashing function (MAC), part of the SHA-3 family of post-quantum resistant hashing functions.

These cryptographic primitives ensure that QSMP's key management processes remain secure, even in scenarios involving high-risk adversaries and quantum-capable threats.

5. Protocol Components and State Structures

5.1 Protocol String

The protocol string in QSMP is composed of four key components, each representing a specific cryptographic element used in the secure communication process:

1. **Asymmetric Signature Scheme**: Specifies the signature scheme along with its security strength (e.g., s1, s3, s5) from low to high. Example: dilithium-s3 correlates to the NIST *level 3* security designation (192 bits of post-quantum security).
2. **Asymmetric Encapsulation Cipher**: Defines the asymmetric encryption algorithm and its security strength. Example: mceliece-s5.
3. **Hash Function Family**: The designated hash function used within the protocol, which is set as SHA3.
4. **Symmetric Cipher**: The symmetric cipher used for data encryption, set as the authenticated stream cipher RCS.

The protocol string plays a crucial role during the initial negotiation phase to ensure that both the client and server agree on a common set of cryptographic parameters. If the client and server do not support the same protocol settings, a secure connection cannot be established.

Signature Scheme	Asymmetric Cipher	HASH Function	Symmetric Cipher
Dilithium	Kyber	SHA3	RCS
Dilithium	McEliece	SHA3	RCS
Sphincs+	McEliece	SHA3	RCS

Table 5.1: The Protocol string choices in revision QSMP 1.3a.

5.2 Client Key Structure

The client key is a publicly exportable structure that contains the signature verification key and associated metadata. It includes parameters such as the key expiration time, protocol string, public signature verification key, and 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.

- **Expiration**: A 64-bit unsigned integer indicating the number of seconds since the epoch (01/01/1900) until the UTC time when the key remains valid. If the key has expired, the client must request a new public key from the server.

- **Configuration**: Contains the protocol string that defines the cryptographic parameters. If the protocol string on both hosts does not match, the connection is aborted.
- **Key ID**: A unique identifier for the public verification key, facilitating quick reference on the server.
- **Verification Key**: The public asymmetric signature verification key used for authenticating asymmetric encapsulation keys and data during the key exchange.

The client key can be distributed openly or could be enveloped and signed using X.509 certificates to create a chain of trust, enhancing its security in diverse environments.

5.3 Server Key Structure

The server key is a private (secret) key retained by the server. It contains all elements of the client key plus an additional parameter, the asymmetric signing key.

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	Signing

Table 5.3: The server key structure.

The inclusion of the signing key in the server key structure allows the server to sign messages during the key exchange, ensuring that data exchanges are authenticated and trusted.

5.4 Keep Alive State

QSMP SIMPLEX uses an internal keep-alive mechanism to maintain active connections. The server periodically sends a keep-alive packet to the client, which the client must acknowledge within the defined interval.

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.

If the server does not receive a response within the timeout period, it logs a keep-alive error and terminates the connection to prevent stale sessions.

5.5 Connection State

The internal connection state structure stores the critical data required by QSMP operations, including cipher states, sequence counters, and the ratchet key.

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.

This data structure ensures secure handling of connection parameters, packet sequencing, and cryptographic states during active communication sessions.

5.6 Duplex Client KEX State

The Duplex client key exchange (KEX) state holds information about asymmetric and symmetric keys during the key exchange process.

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.

This state ensures that all required keys and tokens are securely managed throughout the key exchange process.

5.7 Duplex Server KEX State

The Duplex server KEX state structure mirrors the client state, with additional functionality for handling server-specific key queries.

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.

5.8 Simplex Client KEX State

The Simplex protocol's client and server state structures focus on one-way authentication, storing essential key exchange data:

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.

5.9 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.

5.10 QSMP Packet Header

The QSMP packet header is 21 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 flag, or an error flag.
2. The **Packet Sequence**, this indicates the sequence number of the packet in the exchange.
3. The **Message Size**, this is the size in bytes of the message payload.
4. The **UTC time**, the time the packet was created, used in an anti-replay attack mechanism.

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**	**UTC Time** **8 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.

5.11 Flag Types

The following is a 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
Asymmetric Ratchet Request	0x12	The packet contains an asymmetric ratchet request.
Asymmetric 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.

5.12 Error Types

The following is a 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 the device initialization phase, clients generate an asymmetric signature key-pair. This pair consists of a private key, which the client uses to sign messages in the key exchange, and a public key, which is shared with other hosts and used to verify a message signature. The public key contains the asymmetric signature verification key, a key identity array, the protocol configuration string, and the key expiration date.

These public/private signature keys, generated by the clients, function as the primary authentication keys. The public verification keys can be distributed to other clients via a trusted intermediary, such as a server using a directory service, to ensure secure distribution.

Within the Duplex protocol, participating clients are assigned roles during the connection stage as either a *listener*, which accepts network connection requests, or a *sender*, which initiates the connection request, but a device can be both, initiating or accepting a connection. For the purposes of providing clarity, the listener shall be described in this process as the *server*, and the sender as the *client*.

The sender begins the connection process, and if the listener recognizes the sender's key-id as valid, the key exchange sequence is initiated. During this exchange, the asymmetric cipher keys and ciphertext are signed, verified, and mutually exchanged between the sender and listener. This process results in the generation of a pair of shared secrets, which are used to key symmetric cipher instances for both transmitting and receiving data in a set of secure communication channels.

If an error occurs during the key exchange the affected sender or listener immediately sends an error message to the other host, disconnects, and terminates the session. Error handling includes checks for message synchronization, timing, expected message size during the key exchange, authentication failures, packet expiration, and any internal errors triggered by cryptographic or network operations integral to the key exchange and communication flow.

6.1 Connection Request

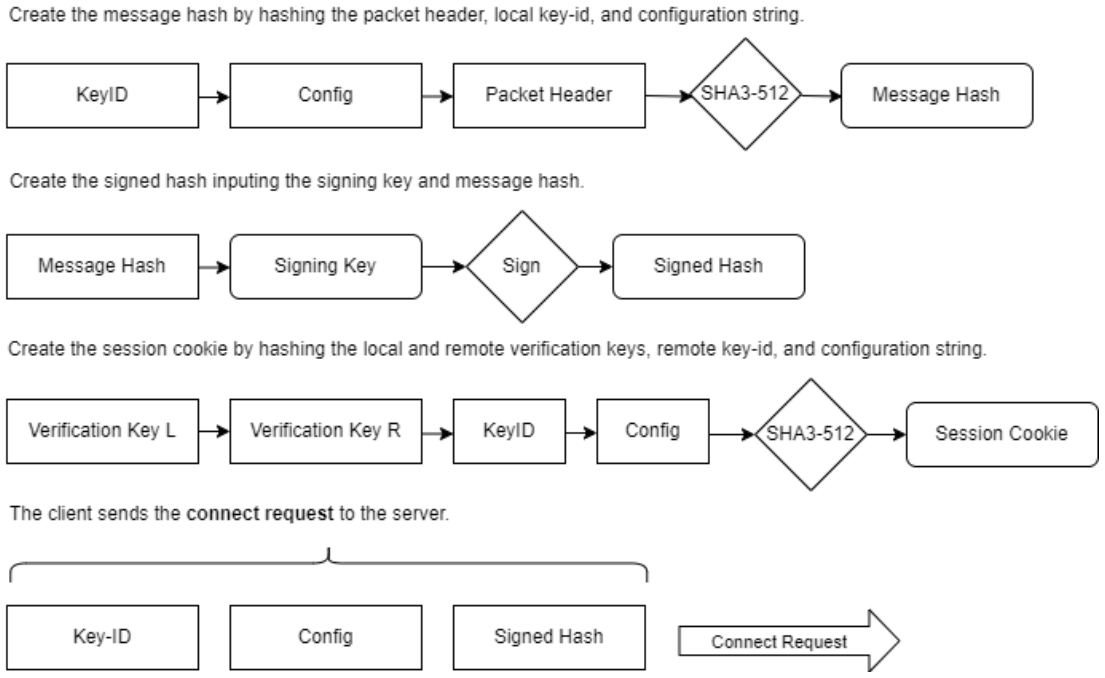


Figure 6.1: QSMP Duplex connection request.

1) The client initiates the key exchange operation by sending a connection request packet to the server. This packet includes the server's key identification array, the protocol configuration string, and a signed hash of the message including the serialized packet header.
2) The packet header fields are verified by the server; message size, sequence number, flag, and the timestamp, all of which are added to the message hash, the message hash is signed, and this guarantees protection from replay attacks.
3) The client generates a hash of the protocol string, along with both the client's and server's asymmetric signature verification keys, and the remote key ID, and stores this information in the session cookie (*sch*) state value for later use in the key exchange. This ensures that the correct verification keys and cryptographic parameters are referenced throughout the key exchange process.

6.2 Connection Response

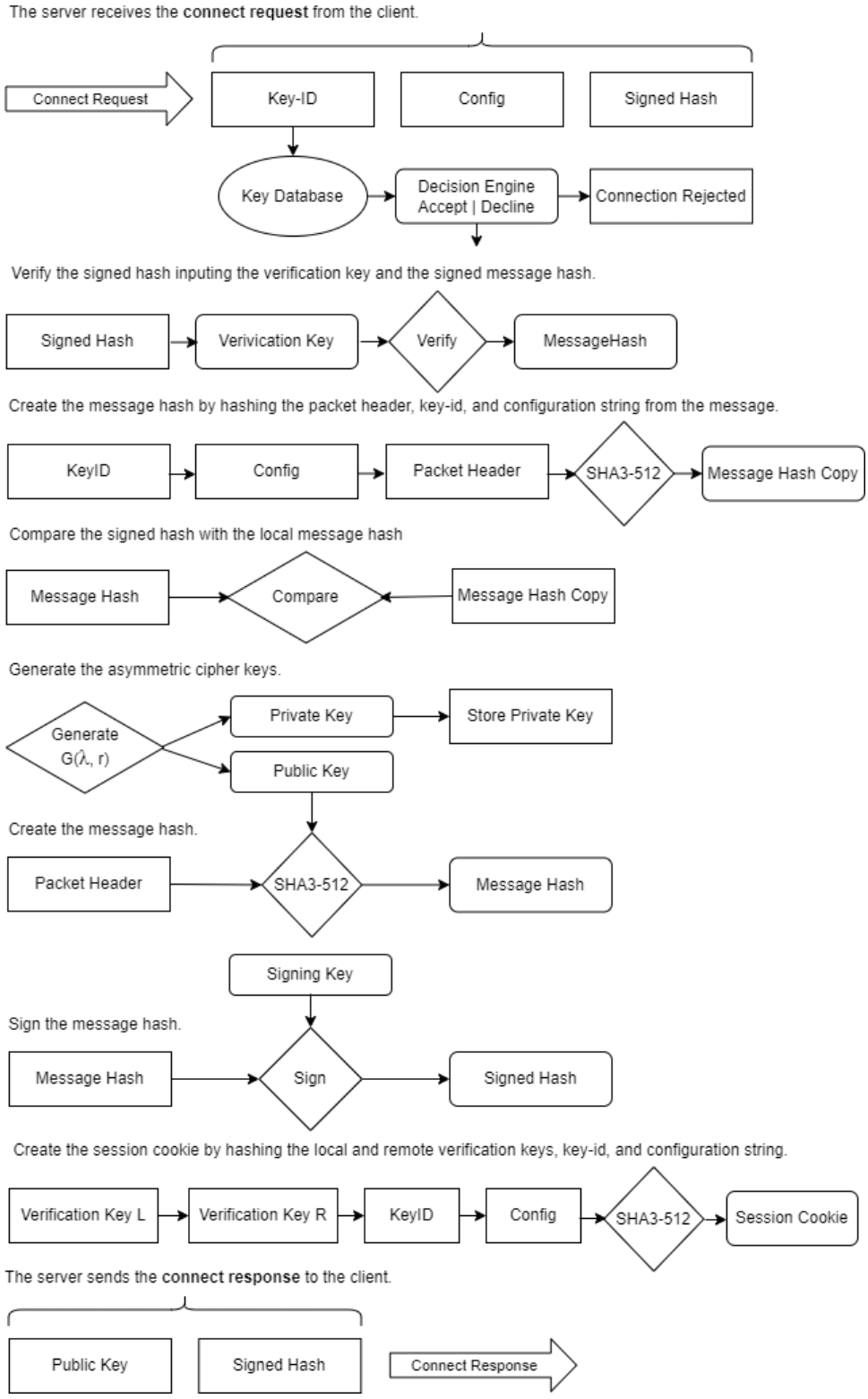


Figure 6.2: QSMP server connection response.

1) The server inspects the packet header for the correct flag, sequence number, expected message size, and that the packet valid-time has not expired.
2) The server checks its database for a key matching the key identification array sent by the client in the connect request message. The verification key is retrieved or the exchange is aborted. If the key is not known to the server, the server sends a *key unrecognized* error message to the client.
3) The server compares the configuration string contained in the message against its own protocol string for a match. If the protocol configuration strings do not match, the server will send an *unknown protocol* error to the client and close the connection.
4) The server verifies the key's expiration time, and if all fields are valid, loads the key into state. If the client's key has expired, the server will send a *key expired* error message.
5) The server checks the signature of the client's message hash using the client's signature verification key.
6) If the signature is authenticated, the server hashes the key-id, the config string, and the serialized *connect request* packet header, and compares this hash to the signed hash it received from the client for equivalence, as the final stage of verification of the message. In any protocol failure occurs, the server notifies the client, closes the connection, and logs the event, and the client is expected to close the connection, and pass the error up to the user interface software, that can initiate actions or inform the user of the cause of the failure.
7) The server generates a public/private asymmetric cipher key-pair.
8) The server hashes the public encapsulation key and the serialized *connect response* packet header, and signs the hash with its asymmetric signature signing key. The client has a copy of the asymmetric signature verification key, that will be used to verify this signature.
9) The server stores the private asymmetric cipher key temporarily in its state.
10) The server hashes the key-id array, the configuration string, and the local and remote copies of the signature verification keys, and stores the hash in its session cookie state value *sch*, for use as a session cookie.
11) The server adds the public asymmetric encapsulation key, and the public key's signed hash, to the *connect response* message, and sends it to the client.

6.3 Exchange Request

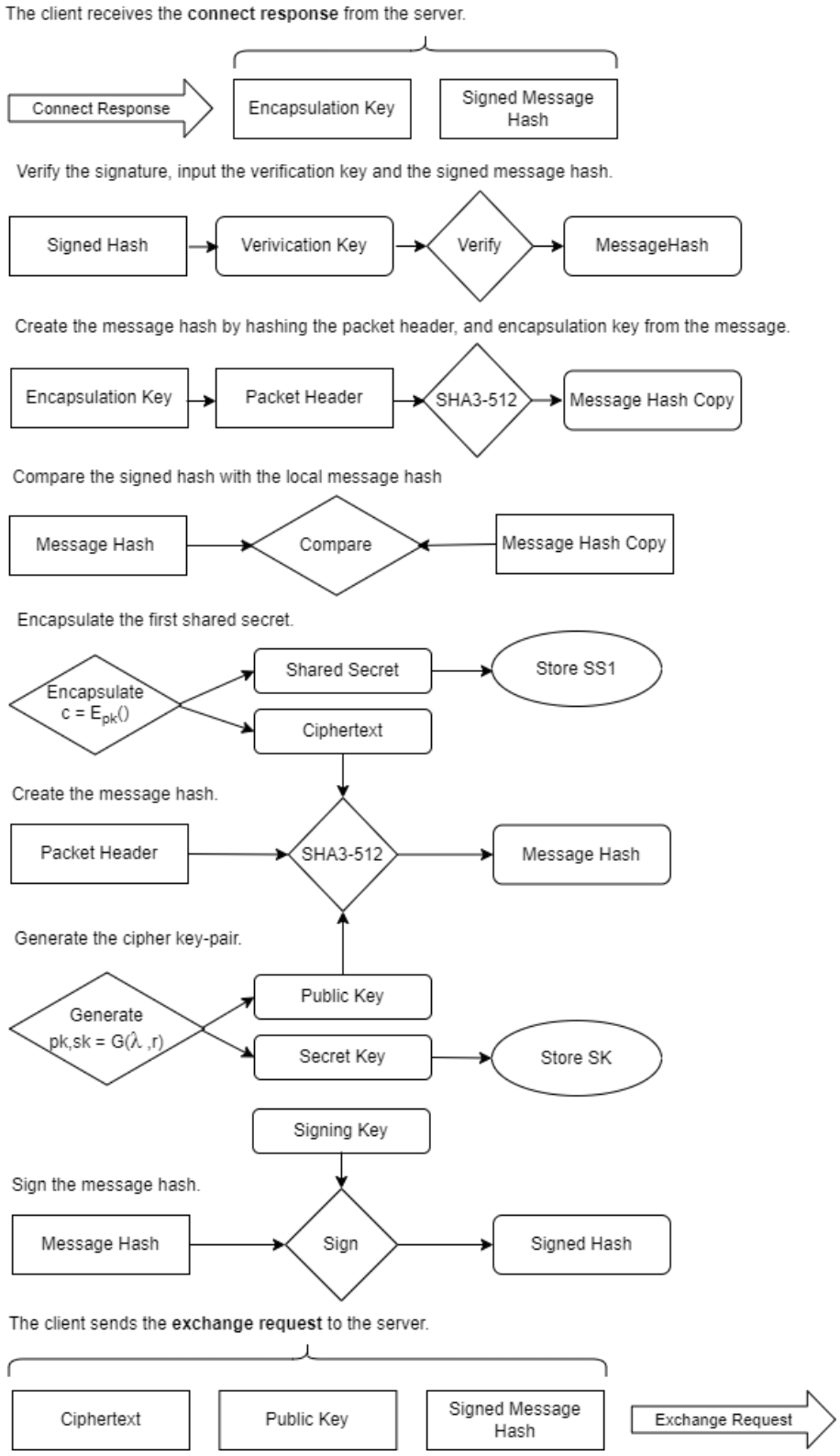


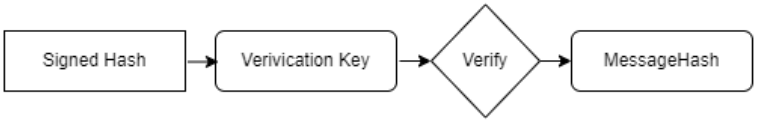
Figure 6.3: QSMP client exchange request.

1) The client inspects the *connect response* packet header for the correct flag, sequence number, expected message size, and that the valid-time has not expired.
2) The client uses the server's public signature verification key to validate the signature on the message hash.
3) If the signature verification is successful, the client hashes the asymmetric cipher key and *connection response* packet header and compares this hash to the signed hash received from the server. If the signature verification fails, the client sends an *authentication failure* message to the server and terminates the connection. Similarly, if the hash comparison fails, the client sends a *hash invalid* error message and closes the connection.
4) Once the signature and hash have been successfully authenticated, the client uses the asymmetric cipher key to encapsulate a shared secret, generating a ciphertext that will be sent to the server. This ciphertext will be used by the server to decapsulate the shared secret, which the client securely stores for later use in deriving the session keys.
5) The client generates a new asymmetric encapsulation key-pair, and stores the private key. The client hashes the public key, the ciphertext, and the serialized *exchange request* packet header, signing the hash using its private signing key.
6) The client adds the asymmetric ciphertext, the public encryption key, and the signed hash to the exchange request packet, which is sent to the server to continue the key exchange process.

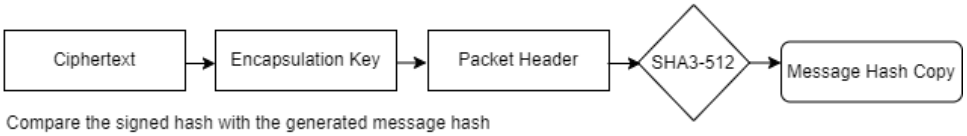
6.4 Exchange Response

The server receives the **exchange request** from the client.

Verify the signature, input the verification key and the signed message hash.

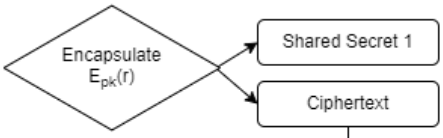


Create the message hash copy by hashing the packet header, ciphertext, and encapsulation key.

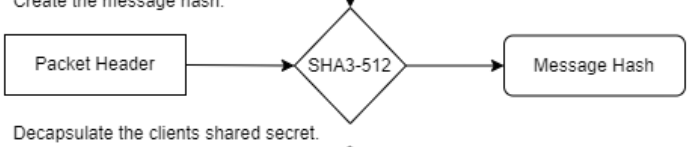


Compare the signed hash with the generated message hash

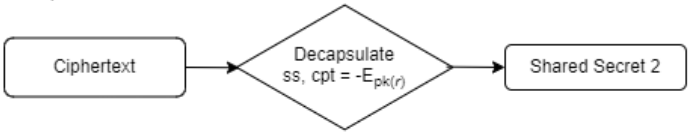
Encapsulate the shared secret in ciphertext.



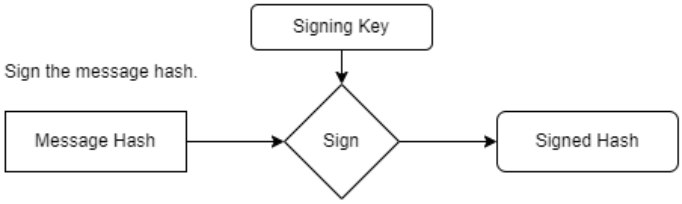
Create the message hash.



Decapsulate the clients shared secret.



Combine the two shared secrets and the session cookie and generate the session keys.



The server sends the **exchange response** to the client.

Figure 6.4: QSMP server exchange response.

1) The server inspects the packet header for the correct flag, sequence number, expected message size, and that the valid-time has not expired.
2) The server verifies the signature of the hash of the message and packet header using the client's signature verification key.
3) The server hashes the public key, cipher-text, and serialized *exchange request* header, and verifies the hash for equivalence to the one contained in the signed hash.
4) The server uses the stored asymmetric cipher private key to decapsulate the first shared secret.
5) The server uses the public key sent by the client to generate a new shared secret and encapsulate it in ciphertext.
6) The two shared secrets and the session cookie are used to key a KDF, which derives the two symmetric session keys (*tx* and *rx*) on the server.
7) The symmetric cipher instances are keyed with the session keys, raising both the transmit and receive channels of the encrypted tunnel.
8) The cipher-text and exchange response header are 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.

6.5 Establish Request

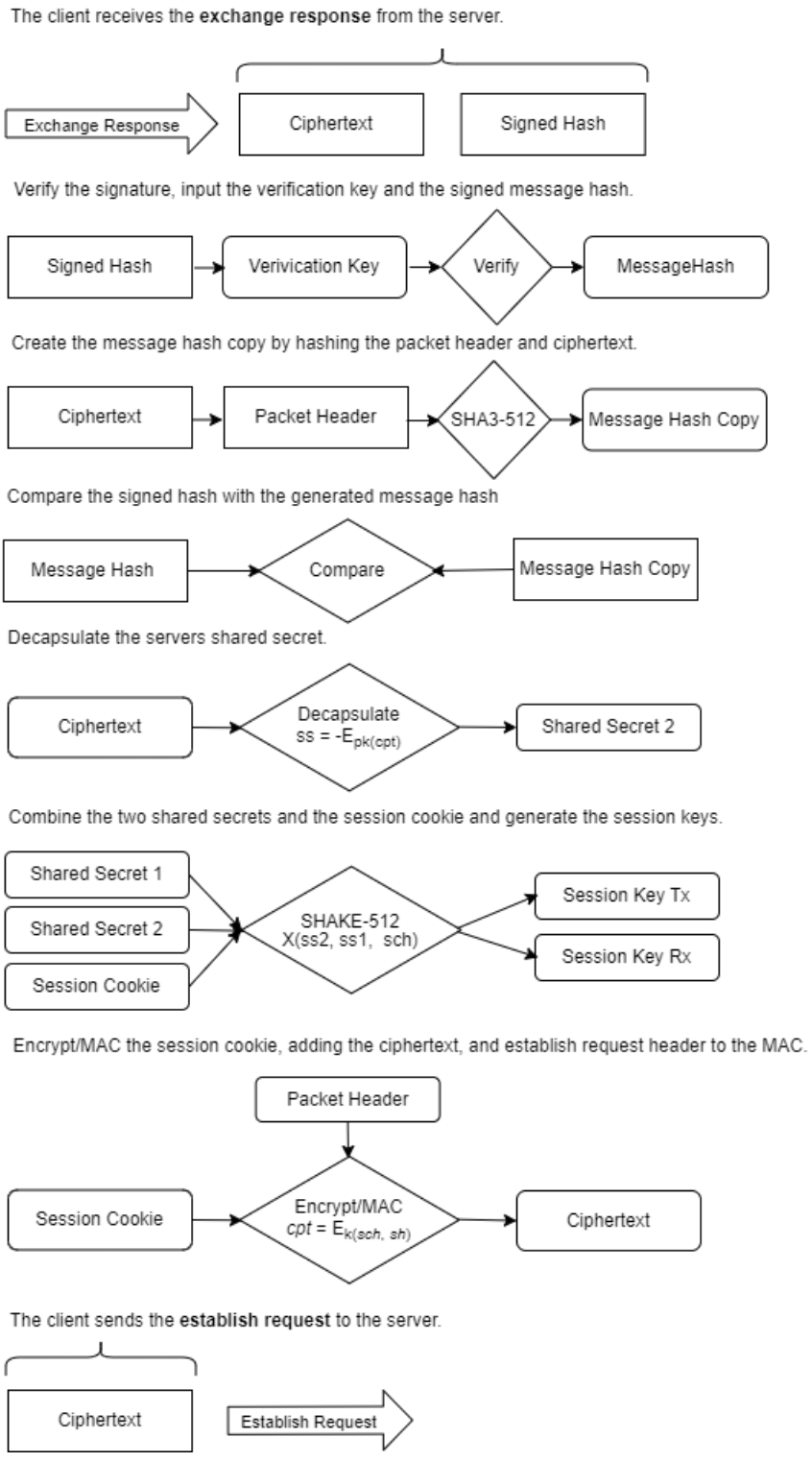


Figure 6.5: QSMP client establish request.

1) The client inspects the packet header for the correct flag, sequence number, expected message size, and that the valid-time has not expired.
2) The client extracts the asymmetric ciphertext and the signed hash of the ciphertext. It uses the server's public verification key to verify the signature on the hash, ensuring its authenticity.
3) The client hashes the ciphertext and the serialized *exchange response* header, and compares the generated hash with the signed hash. If the hashes match, the client confirms the integrity of the data.
4) The client decapsulates the shared secret from the ciphertext.
5) The client combines this shared secret with the previously stored shared secret and the session cookie to key a KDF which derives the (*rx* and *tx*) symmetric session keys.
6) The session keys are used to initialize the transmit and receive symmetric cipher instances, establishing both transmit and receive channels of the encrypted tunnel.
7) The client encrypts the session cookie with the *tx* instance of the symmetric ciphers, and adds the serialized *establish request* header to the additional data of the AEAD stream cipher (RCS).

6.6 Establish Response

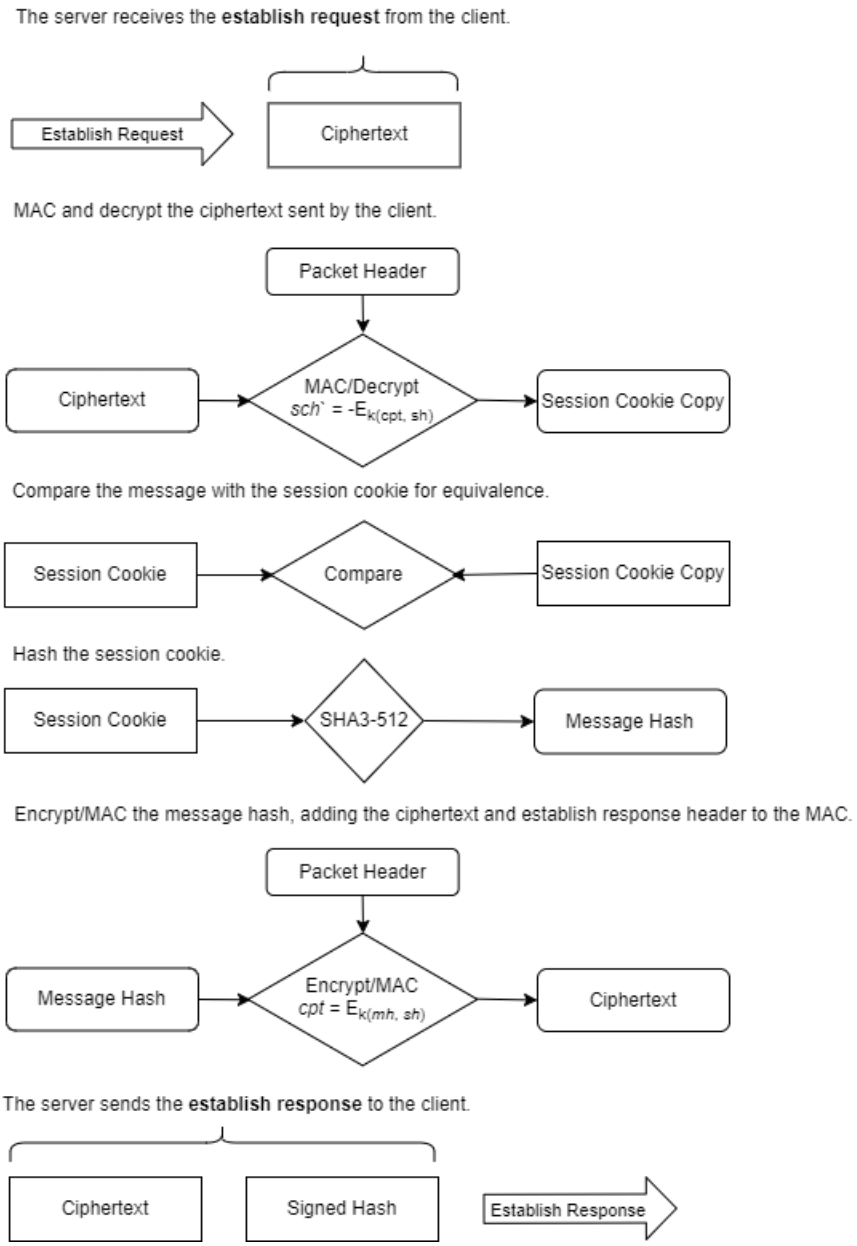


Figure 6.6: QSMP server establish response.

1) The server inspects the packet header for the correct flag, sequence number, expected message size, and that the valid-time has not expired.
2) The server decrypts the ciphertext using the *rx* cipher instance, adding the serialized *establish request* packet header as additional data.
3) The message is compared to the session cookie for equivalence. If the decryption succeeds and the message equals the session cookie, the session cookie is hashed.

4) The hashed session cookie is encrypted using the *tx* cipher instance, adding the serialized *establish response* packet header to the cipher MAC.
5) The message is sent to the client, and the tunnel interface is changed to the active state on the server.

6.7 Establish Verify

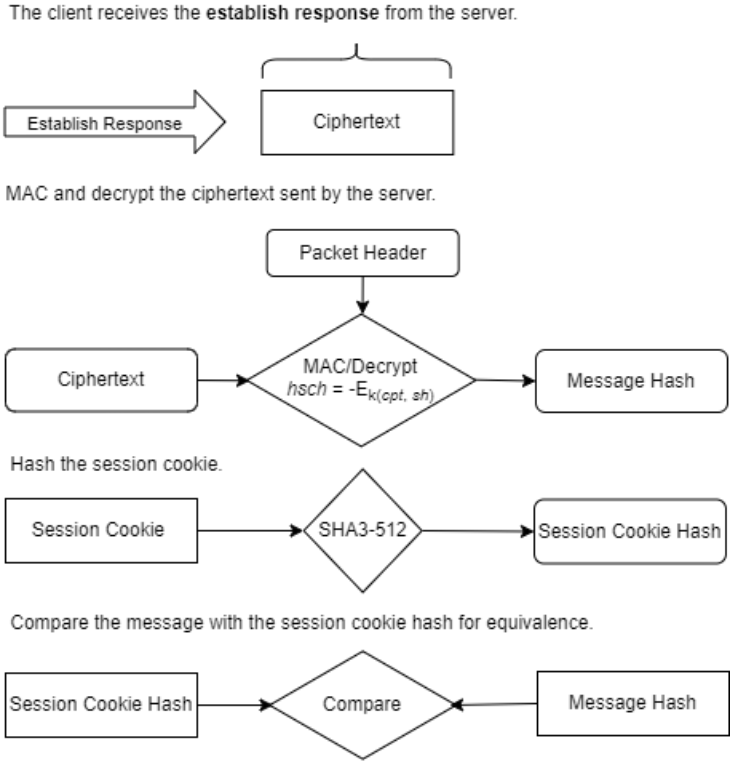


Figure 6.7: QSMP client establish request.

1) The client inspects the packet header for the correct flag, sequence number, expected message size, and that the valid-time has not expired.
2) The client decrypts the ciphertext using the *rx* instance of the cipher, adding the serialized *established request* packet header to the cipher MAC.
3) The client hashes the session cookie, and compares it to the decrypted message for equivalence.
4) If the hashes are equal, the tunnel state is changed to active, and the encrypted tunnel interfaces are now ready to process data.

6.8 Asymmetric Ratchet

The [optional] asymmetric ratchet mechanism in QSMP injects new entropy by periodically re-keying the symmetric ciphers. This process involves combining a hash of the original session key with new keying material obtained through an asymmetric key exchange.

When the primary key exchange operation establishes the encrypted tunnel, the primary session keys for both the receive and transmit channels are hashed, and this hash is stored in a persistent ratchet state. This ratchet state forms the foundation for future re-keying operations.

Upon invocation of the asymmetric ratchet function, the initiator generates a new asymmetric cipher key pair and transmits the public key to the remote host over the existing encrypted tunnel. Before transmission, the public key is hashed, and this hash is signed using the initiator's private signature key to ensure its authenticity.

The receiving host verifies the signature using the initiator's public signature verification key. If the signature is valid, the host hashes the received public key and compares it to the signed hash as an additional validation step. Upon successful verification, the host uses the verified public key to generate a new shared secret and corresponding ciphertext. This shared secret is then hashed, along with the hash of the original session key, and used to key the KDF and derive a new set of session keys, which are used to re-key the transmit and receive channels of the symmetric cipher instances.

The receiving host hashes the new ciphertext and signs the hash using its private signature key, sending this signed hash and ciphertext back to the initiator. The initiator then verifies the signature using its stored copy of the receiving host’s public signature verification key. If the signature and hash checks are successful, the initiator decrypts the ciphertext to obtain the shared secret, which is then added to a KDF with the initial session key hash to create new symmetric session keys for both transmit and receive channels.

The persistent ratchet key state is updated by hashing these new session keys, preparing it for the next invocation of the ratchet mechanism. This process can be initiated by either the sender or the receiver, allowing either party to enhance the security of the communication stream.

The asymmetric ratchet can be triggered by the hosting software under various conditions, such as after a specific amount of data has been transmitted, when starting a new session with persistent keys, or even after each individual message exchange. This dynamic mechanism provides robust forward security, ensuring that an adversary cannot derive previous keys from the current key, and predictive resistance, preventing future keys from being derived from the current state alone.

6.9 Symmetric Ratchet

The [optional] symmetric ratchet mechanism in QSMP periodically updates the symmetric session keys using a randomly generated token. This process introduces new entropy into the encrypted stream and ensures that even if the current session keys are compromised, past keys remain secure.

The initiator of the symmetric ratchet generates a random token and transmits it over the encrypted channel. After sending the encrypted key, the initiator hashes this token together with the persistent ratchet key, which itself is a hash of the initial session key stored in the ratchet state. The combined hash of the random token and the ratchet key is added to the key derivation function and used to derive a new set of session keys. These derived session keys are used to re-key the symmetric cipher instances for both the transmit and receive channels, ensuring forward secrecy.

On the receiving end, the receiver decrypts the random token, then hashes it along with the ratchet key from its own persistent state. The resulting hash is used to key the KDF and derive the new symmetric session keys, which are then applied to re-key the transmit and receive channels.

This symmetric ratchet mechanism offers strong forward secrecy, it ensures that the knowledge of the current state alone is not sufficient to determine any of the previous session keys. This continuous re-keying process prevents attackers from gaining insight into past communications, even if they manage to compromise the current session key.

7: Simplex Protocol Operational Overview

The Simplex exchange is a one-way-trust client-server key exchange model in which the client trusts the server, and a single shared secret is securely exchanged between them. Designed for efficiency, the Simplex exchange is fast and lightweight, while providing 256-bit post-quantum security, ensuring protection against future quantum-based threats.

This protocol is versatile and can be used in a wide range of applications, such as client registration on networks, secure cloud storage, hub-and-spoke model communications, commodity trading, and electronic currency exchange—essentially, any scenario where an encrypted tunnel using strong quantum-safe cryptography is required.

The server in this model is built as a multi-threaded communications platform capable of generating a uniquely keyed encrypted tunnel for each connected client. With a lightweight state footprint of less than 4 kilobytes per client, a single server instance has the capability to handle potentially hundreds of thousands of simultaneous connections. The cipher encapsulation keys utilized during each key exchange are ephemeral and unique, ensuring that every key exchange remains secure and independent from previous key exchanges.

The server distributes a public signature verification key to its clients. This key is used to authenticate the server's public cipher encapsulation key during the key exchange process. The server's public verification key can be shared with clients through various secure methods, including during a registration event, pre-embedding in client software, or via other secure distribution channels.

7.1 Connection Request

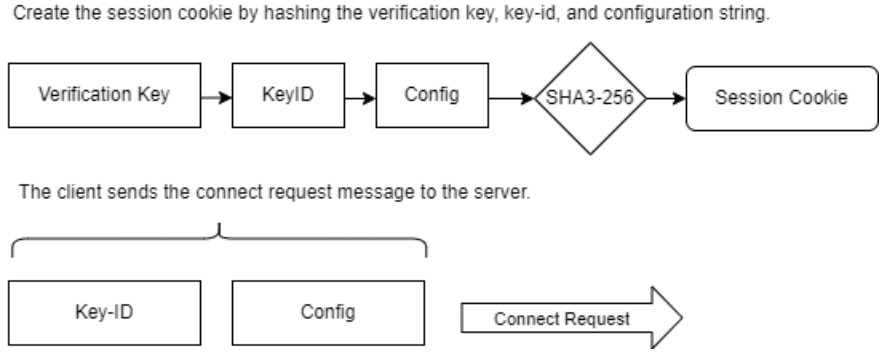


Figure 7.1: QSMP Simplex connection request.

1) The client begins the key exchange operation by sending a connect request packet to the server. This packet contains the server's key identification array and the protocol configuration string.
2) The client hashes the configuration string, the key identification array, and its signature verification key. This combined hash is stored in the session cookie state value (*sch*) and is used as a unique session identifier. This approach ensures that the session's cryptographic parameters are referenced and that the session state is uniquely identifiable.
3) The client adds the key-id and the configuration string, and sends the connection request to the server.

7.2 Connection Response

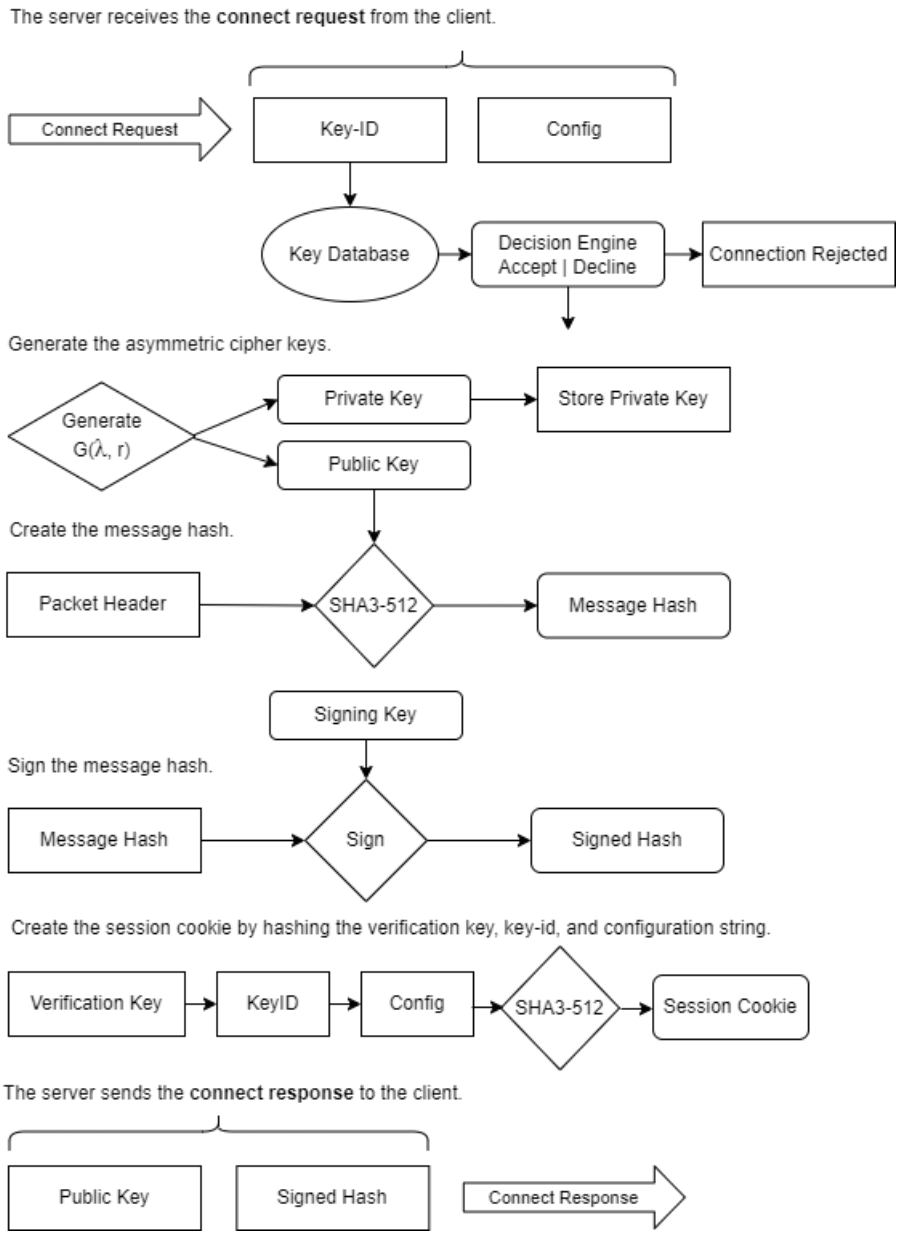


Figure 7.2: QSMP server connection response.

1) The server inspects the packet header for the correct flag, sequence number, expected message size, and that the valid-time has not expired.
2) The server checks its database for a key that matches the key identification array provided in the request. If the verification key is not found, the server sends an *unknown key* error message to the client, aborts the key exchange, logs the event, and tears down the session.
3) The server compares the protocol configuration string sent by the client with its own stored protocol string to ensure compatibility.

4) The server verifies the expiration time of the key. If all these fields are validated successfully, the server loads the key into its active state.
5) The server hashes the configuration string, the key identification array, and its signature verification key, and stores this combined hash in its session cookie state value (sch).
6) The server generates a new public/private asymmetric cipher key pair. It hashes the public encapsulation key and the serialized connection response packet header, and signs this hash with its private signing key.
7) The server adds the public asymmetric encapsulation key and the signed hash of the public key to the connect response message and sends it to the client to continue the key exchange process.

7.3 Exchange Request

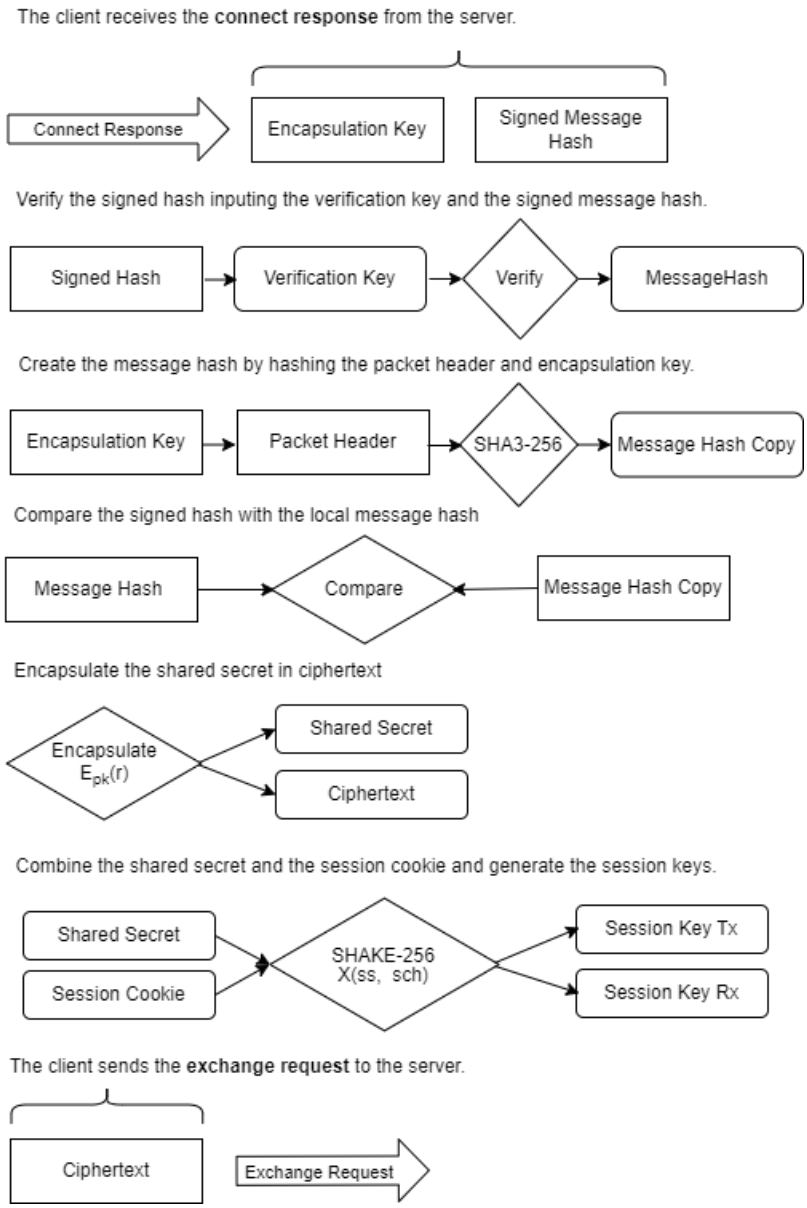


Figure 7.3: QSMP client exchange request.

1) The client inspects the packet header for the correct flag, sequence number, expected message size, and that the packet valid-time has not expired.
2) The client uses the server's signature verification key to verify the signature on the hash of the asymmetric encapsulation key and serialized packet header. If the signature verification fails, the client sends an *authentication failure* message and terminates the connection.
3) If the signature is successfully verified, the client hashes the asymmetric cipher key and serialized header, and compares this hash to the signed hash in the server's response message.

If the hash check fails, the client sends a *hash invalid* error message and closes the connection.

4) The client uses the asymmetric cipher key to encapsulate a *shared secret*, creating the ciphertext.
5) The shared secret is combined with the session cookie to key the KDF, which generates the symmetric cipher keys and nonces used to key the transmit and receive cipher instances.
6) The cipher *rx* and *tx* symmetric instances are initialized and ready to transmit and receive data.
7) The asymmetric ciphertext is added to the exchange request packet, which the client sends to the server.

7.4 Exchange Response

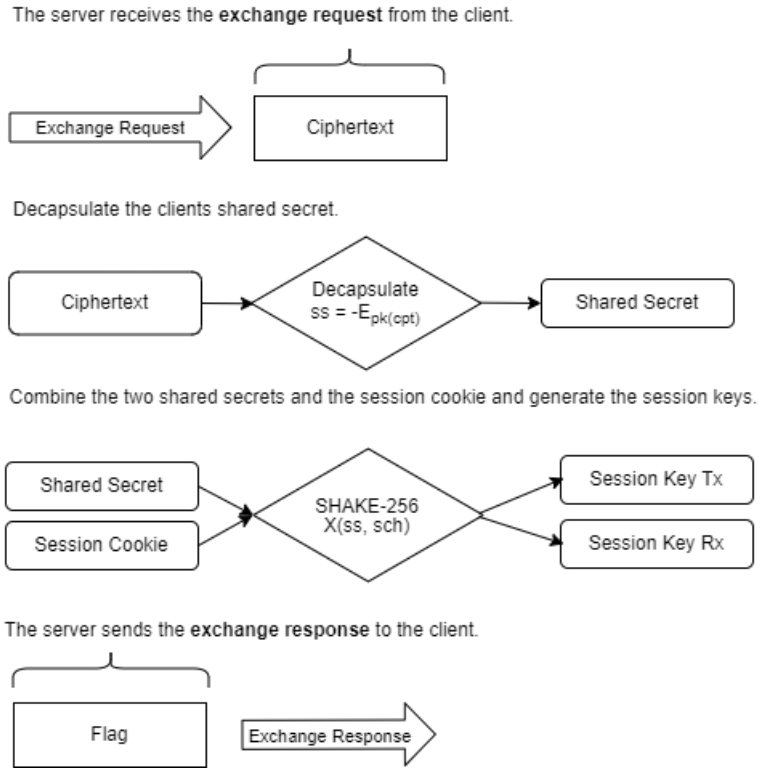


Figure 7.4: QSMP server exchange response.

1) The server inspects the packet header for the correct flag, sequence number, expected message size, and that the valid-time has not expired.
2) The server uses its stored asymmetric cipher private key to decapsulate the shared secret from the ciphertext.
3) The decapsulated shared secret is combined with the session cookie to derive the two symmetric session keys and nonces.
4) These derived session keys are used to initialize the symmetric cipher instances, activating both the transmit and receive channels of the encrypted tunnel.

7.5 Establish Verify

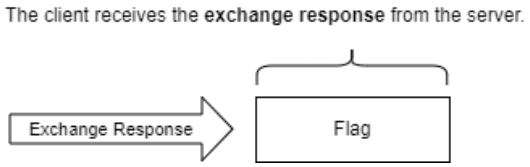


Figure 7.5: QSMP client establish request.

1) The client inspects the packet header for the correct flag, sequence number, expected message size, and that the valid-time has not expired.
2) The client verifies that the encrypted tunnel is now active and fully operational. If the packet contains an error flag, indicating that an issue occurred during the tunnel setup, the client immediately initiates a connection teardown.
3) The client **should** then handle the error according to its predefined procedures, ensuring the user or application is informed of the failure.

Mathematical Symbols

← ↔ →	-Assignment and direction symbols
:=, !=, ?=	-Equality operators; assign, not equals, evaluate
C	-The client host, initiates the exchange
S	-The server host, listens for a connection
G(λ, r)	-The asymmetric cipher key generation with parameter set and random source
$-E_{sk}$	-The asymmetric decapsulation function and secret key
E_{pk}	-The asymmetric encapsulation function and public key
S_{sk}	-Sign data with the secret signature key
V_{pk}	-Verify a signature the public verification key
cfg	-The protocol configuration string
cpr_{rx}	-A receive channels symmetric cipher instance
cpr_{tx}	-A transmit channels symmetric cipher instance
cpt	-The symmetric ciphers cipher-text
cpta	-The asymmetric ciphers cipher-text
$-E_k$	-The symmetric decryption function and key
E_k	-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
M_{mk}	-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

8 QSMP Duplex Formal Description

Duplex Key Exchange Sequence

Preamble:

The Duplex key exchange is designed to facilitate secure communication in a peer-to-peer architecture. Each client in the network has a unique signature verification key, which is shared with other clients to authenticate communications.

These signature verification keys can be exchanged using a host lookup system, where a client queries a server that maintains a database of clients on its network. Upon receiving a request, the server checks the client's authorization status and, if approved, returns information about the target client, including its public signature verification key. This key can then be cached on the querying client for as long as the key expiration time remains valid.

Additionally, the server could act as a central point of authentication by signing client verification keys, thereby enhancing the trustworthiness of key exchanges.

In the Duplex architecture, since one node must initiate the connection while the other must accept it, the initiator is designated as the *client*, and the recipient of the request is referred to as the *server* within the key exchange context.

8.1 Connect Request

The client initiates the connection by sending a *connection request* to the server, which includes its configuration string and signature verification-key identity string.

The key identity (kid) is a multi-part, 16-byte array that serves as both a device and key identification array. This identifier is used to link the intended target with its corresponding cryptographic key, ensuring that the correct signature verification key is used during the secure exchange.

The configuration string (*cfg*) specifies the set of cryptographic protocols being utilized in the key exchange process. For the exchange to proceed successfully, the configuration strings of both the client and server must match exactly, indicating they are using the same protocol parameters.

To maintain the integrity and state of the key exchange, the client generates a *session cookie* by hashing a combination of the configuration string, the key identity, and the public asymmetric signature verification keys from both the client and the server:

$sch \leftarrow \mathrm{H}(cfg \,||\, kid \,||\, pvka \,||\, pvkb)$

Where:

- *cfg* is the configuration string.

- *kid* is the key identity.
- *pvka* is the client's public verification key.
- *pvkb* is the server's public verification key.

This session cookie (*sch*) serves as a unique identifier for the session, ensuring secure reference to the cryptographic parameters throughout the key exchange.

The client serializes the *connection request* packet header (*sh*), including the protocol flag, message size, sequence number, and the timestamp. The serialized header is added to a hash along with the key id and configuration string. The client signs the hash with its asymmetric signing key, and adds this to the packet message along with the kid and cfg arrays.

$shm \leftarrow S_{sk}(H(kid \parallel cfg \parallel sh))$

The client then transmits the *connection request* to the server to begin the key exchange operation:

$C\{ kid \parallel cfg \parallel shm \} \rightarrow S$

8.2 Connect Response

The server processes the client's connection request and responds with either an error message or a connect response packet. If any error occurs during the key exchange, the server generates an error packet and sends it to the remote host, triggering a teardown of both the key exchange and the network connection on both ends.

Key Verification and Configuration Check

The server checks the *connect request* packet header, including the sequence number, message size, protocol flag, and valid-time timestamp. This check is done at each step of the exchange, verifying inbound packets for correctness of the expected flag, message size, creation time, and sequence number. The UTC timestamp is tested for a valid-time threshold; if the local time is different from the packet creation time by more than the threshold (default is 60 seconds) the packet is rejected, and the exchange is torn down. This mechanism protects the exchange from replay attacks and packet header tampering. Serialized packet headers are either added to the hash of a message and signed, or added to the *additional data* of the authenticated stream cipher (RCS) to guarantee authenticity.

The server verifies that it has the requested asymmetric signature verification key that matches the client's host using the key identity array (*kid*). It then checks that its protocol configuration is compatible with that of the client.

The server verifies the message signature, then hashes the message, which is compared to the hash signed by the client for equivalence.

Where:

- *shm* is the signed message hash received from the client.
- *hm* is the hashed message signed by the client.
- *hm`* is the message hashed by the server.
- *m* is the packet message : *kid* || *cfg* || *sh*

$V_{pk}(shm) \leftarrow (\text{true} \mathrel{?=} hm : 0)$

$hm` \leftarrow H(kid \,||\, cfg \,||\, sh)$

$hm` \mathrel{?=} hm : m : 0$

The server creates a session cookie by hashing the configuration string, the key identity, and both the public signature verification keys:

Where:

- *cfg* is the configuration string.
- *kid* is the key identity.
- *pvka* is the client's public verification key.
- *pvkb* is the server's public verification key.

$sch \leftarrow H(cfg \,||\, kid \,||\, pvkb \,||\, pvka)$

This hash acts as a unique session identifier for the exchange, and is added to the KDF as an input when the session keys are generated.

Asymmetric Key Generation and Signing

The server generates a new asymmetric cipher key pair and securely stores the private key. It then hashes the public encapsulation key and the serialized outbound packet header, and signs this hash using its private asymmetric signature key.

Key generation and signing steps are as follows:

Generate the public (*pk*) and private (*sk*) asymmetric encryption keys.

$pk, sk \leftarrow G(\lambda, r)$

Create a hash of the public key and serialized *connection response* packet header (*sh*).

$pkh \leftarrow H(pk \,||\, sh)$

Sign the hashed public key using the server's private signature key.

$spkh \leftarrow S_{sk}(pkh)$

The server then sends the *connection response* message to the client, which contains the signed hash of the public asymmetric encapsulation key (*spkh*) and a copy of the public key:

$S\{\, spkh \,||\, pk \,\} \rightarrow C$

8.3 Exchange Request

The client processes the *connect response* message from the server and proceeds with the next steps in the key exchange. This phase involves verifying the server's public key, encapsulating a shared secret, and authenticating the message.

Signature Verification and Hash Check

The client checks the *connect response* packet header, the flag, expected message size, the valid-time timestamp, and the sequence number.

The client verifies the server's signature on the hashed public key and serialized packet header. It then generates its own hash of the received public key and serialized header and compares it to the one included in the server's message. If the hashes match, the client proceeds with the key exchange. If the hashes do not match, the key exchange is aborted, and the session is terminated.

The client verifies the hash of the public key using the server's public verification key. If the hash is valid, the process continues; otherwise, the exchange fails.

$V_{pk}(H(pk \parallel sh)) \leftarrow (\text{true} \mathrel{?=} pk : 0)$

Once the public key is verified, the client uses it to encapsulate a shared secret. The client generates a ciphertext (*cpta*) and encapsulates the shared secret (*seca*) using the public key.

$cpta = E_{pk}(seca)$

The client stores the shared secret (*seca*), which will be combined with another shared secret and the session cookie to derive the symmetric session keys later in the exchange.

Asymmetric Key Generation and Signing

The client generates its own asymmetric encryption key pair and securely stores the private key. It then creates a hash of its public encapsulation key, the ciphertext and the serialized outbound packet header, and signs this hash using its private asymmetric signature key.

Key generation and signing steps:

Generate the client's public (*pk*) and private (*sk*) asymmetric encryption keys.

$pk, sk \leftarrow G(\lambda, r)$

Hash the client's public key and the ciphertext.

$kch \leftarrow H(pk \parallel cpta \parallel sh)$

Sign the hashed value using the client's private signature key.

$skch \leftarrow S_{sk}(kch)$

The client sends an exchange request message back to the server. This message contains the signed hash of its public asymmetric encapsulation key and ciphertext, the ciphertext itself, and a copy of the public encapsulation key:

$C\{\ cpta \parallel pk \parallel skch\ \} \rightarrow S$

8.4 Exchange Response

The server processes the exchange request from the client, verifying the integrity of the message, decapsulating the shared secret, and establishing the symmetric session keys for the secure communication channel.

Signature Verification and Hash Check

The server checks the *exchange request* packet header, the flag, expected message size, the valid-time timestamp and sequence number.

The server verifies the signature of the hash included in the client's message. It then generates its own hash of the client's public key and the ciphertext, comparing this hash with the one provided in the message signature. If the hashes match, the server continues with the key exchange; otherwise, the process is terminated, and the key exchange is aborted.

The server uses the client's public verification key to verify the hash of the public key, ciphertext and serialized exchange request packet header. If the verification is successful, the process continues; otherwise, the server halts the exchange.

$V_{pk}(H(pk \parallel cpta \parallel sh)) \leftarrow$ (true ?= pk || cpta : 0)

Shared Secret Decapsulation

The server decapsulates the first shared secret received from the client. The server uses its private asymmetric key to decapsulate the shared secret (*seca*) from the ciphertext (*cpta*) provided by the client.

$seca \leftarrow -E_{sk}(cpta)$:

This shared secret (*seca*) is securely stored for use in generating the session keys.

Generation of Second Shared Secret

The server generates a new ciphertext and a second shared secret using the client's public encapsulation key. The server generates a second ciphertext (*cptb*) and shared secret (*secb*) using the client's public key.

$cptb \leftarrow E_{pk}(secb)$

Session Key Derivation

The server combines the two shared secrets (*seca* and *secb*) with the session cookie (*sch*) to derive two symmetric session keys and two unique nonces, one for each communication channel. The key expansion function generates two symmetric keys (*k1*, *k2*) and two nonces (*n1*, *n2*) for the transmit and receive channels of the communication stream.

$k1, k2, n1, n2 \leftarrow KDF(seca, secb, sch)$

Cipher Initialization

The symmetric cipher instances for the receive and transmit channels are then initialized with the derived session keys and nonces.

Initializes the receive channel cipher with key *k1* and nonce *n1*.

$cpr_{rx}(k1, n1)$

Initializes the transmit channel cipher with key *k2* and nonce *n2*.

$cpr_{tx}(k2, n2)$

Hash and Signature of Ciphertext

To complete the exchange response, the server hashes the newly generated ciphertext and signs the hash to ensure its integrity and authenticity before sending it back to the client.

$cpth \leftarrow H(cptb \parallel sh)$

$scph \leftarrow S_{sk}(cpth)$

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

$S\{ cptb, scph \} \rightarrow C$

8.5 Establish Request

In the final phase of the key exchange process, the client completes the establishment of the encrypted communication channel by validating the received data, decapsulating the shared secret, and generating the symmetric session keys.

Signature Verification and Hash Check

The client checks the *exchange response* packet header, the flag, expected message size, the valid-time timestamp and sequence number.

The client verifies the server's signature on the hash of the ciphertext and serialized exchange response packet header. It generates its own hash of the ciphertext and compares it with the one provided by the server. If the hashes match, the client proceeds to decapsulate the shared secret; otherwise, the key exchange is aborted.

The client verifies the hash of the server's ciphertext (*cptb*) using the server's public verification key. If the verification is successful, the client continues; otherwise, it terminates the exchange.

$V_{pk}(H(cptb \parallel sh)) \leftarrow (\text{true} ?= cptb : 0)$:

Shared Secret Decapsulation

The client decapsulates the second shared secret from the ciphertext received from the server.

The client uses its private asymmetric key to decapsulate the second shared secret (*secb*) from the server's ciphertext (*cptb*).

$secb \leftarrow -E_{sk}(cptb)$

Session Key Derivation

The client combines both shared secrets (*seca* and *secb*) with the session cookie (*sch*) to generate the session keys and nonces for the secure communication channels.

The key derivation function produces two symmetric session keys (*k1* and *k2*) and two unique nonces (*n1* and *n2*) for the receive and transmit channels.

$k1, k2, n1, n2 = \text{KDF}(seca, secb, sch)$

Cipher Initialization

The client initializes the symmetric ciphers for both communication channels.

Initializes the receive channel cipher with key *k2* and nonce *n2*.

$cpr_{rx}(k2, n2)$

Initializes the transmit channel cipher with key *k1* and nonce *n1*.

$cpr_{tx}(k1, n1)$

Establish Request Message

Once the symmetric channels are successfully initialized, the client sends a copy of the session cookie through the encrypted tunnel to the server, signaling that both encrypted channels of the tunnel are now active and that the tunnel is in its operational state.

$hsch \leftarrow H(sch \parallel sh)$

The establish request packet header is serialized and added to the *additional data* of the transmit instance of the authenticated cipher (RCS). The session cookie is encrypted and sent to the server.

$cpt \leftarrow E_k(hsch, sh)$

In the event of an error during this process, the client sends an error message to the server, which causes the key exchange to abort and the connection to be terminated on both ends.

The client sends the establish request to the server, indicating the successful establishment of the encrypted tunnel.

C{ cpt } → S

8.6 Establish Response

Strictly speaking, this step is not mandatory. If an error occurs during the final stage of the key exchange and the session keys do not match between the hosts, the first message sent will fail symmetric authentication, causing the tunnel to close automatically. However, in the interest of good design and ensuring the secure establishment of the tunnel, the tunnel state should still be explicitly confirmed. This approach prevents the risk of allowing the cipher's MAC function to process messages when the tunnel has not been properly confirmed.

Server Response Verification

The server checks the *establish request* packet header, the flag, expected message size, the valid-time timestamp and sequence number.

The server adds the serialized *establish request* packet header to the *additional data* of the receive instance of the authenticated stream cipher, and decrypts the session cookie.

$hsch \leftarrow -E_k(cpt, sh)$

The decrypted session cookie is compared to the local session cookie for equivalence. If the hashes are equal, the server hashes the session cookie.

$hhsch \leftarrow H(hsch)$

The server adds the serialized *establish response* packet header to the additional data of the transmit cipher instance, and encrypts the hashed session cookie.

$cpt \leftarrow E_k(hhsch, sh)$

Once the server sends the establish response, it sets its internal state to "session established," signaling that the encrypted tunnel is fully operational and ready to process data transmissions.

$S\{ cpt \} \rightarrow C$

8.7 Establish Verify

In the final step of the key exchange sequence, the client verifies the status of the encrypted tunnel based on the server's response.

Client Verification

The client checks the *establish response* packet header, the flag, expected message size, the valid-time timestamp and sequence number.

If the flag does not indicate an establish response, the client identifies that the tunnel is in an error state as specified by the message. In such cases, the client initiates a teardown of the tunnel on both sides to ensure that no data is transmitted over an insecure connection.

Operational State

The client adds the serialized establish response packet header to the additional data of the receive instance of the authenticated stream cipher. The client decrypts the session cookie, hashes its own session cookie, and compares the two hashes for equivalence.

$hhsch \leftarrow -E_k(cpt, sh)$

$hhsch` \leftarrow H(hsch)$

$hhsch` \ ?= hhsch$ (true : 0):

If the two hashes are equal the encrypted tunnel is in the up state, and ready to transmit and receive data.

8.8 Transmission

During message transmission, either the client or server initiates the process of securely sending data over the encrypted tunnel. This involves encrypting the message, updating the message authentication code (MAC), and preparing the packet for secure delivery.

Message Serialization and Encryption

The transmitting host, whether it is the client or server, first serializes the packet header, which includes details such as the message size, timestamp, protocol flag, and sequence number. This serialized header is then added to the symmetric cipher's associated data parameter to ensure that it is securely integrated into the encryption process.

The host proceeds to encrypt the message using the RCS (Rijndael Cryptographic Stream) stream cipher’s Authenticated Encryption with Associated Data (AEAD) functions. The encryption process generates a ciphertext, which is then passed through the MAC function to produce a verification code.

The plaintext message (*m*) is encrypted using the symmetric encryption function (E_k) to generate the ciphertext (*cpt*).

$cpt \leftarrow \mathrm{E_k}(m)$

The MAC code (*mc*) is calculated by updating the MAC function with the serialized packet header (*sh*) and the ciphertext (*cpt*).

$mc \leftarrow \mathrm{M_{mk}}(sh, cpt)$

The MAC code is appended to the end of the ciphertext, ensuring that any tampering with the data during transmission will be detected.

Packet Decryption and Verification

Upon receiving the packet, the host deserializes the packet header and adds it to the MAC state, along with the received ciphertext. The host then finalizes the MAC computation and compares the output code with the MAC code appended to the ciphertext. If the codes match, the ciphertext is authenticated and can be safely decrypted.

If the MAC verification succeeds, the ciphertext (*cpt*) is decrypted back into the plaintext message (*m*).

$m \leftarrow \text{-}\mathrm{E_k}(cpt)$

The packet timestamp is compared to the *UTC* time, if the time is outside of a tolerance threshold, the packet is rejected and the session is torn down.

If the MAC check fails, the decryption function returns an empty message array and an error signal, indicating that the message was either corrupted or tampered with.

This process guarantees the integrity and confidentiality of the transmitted data, allowing the application to handle any errors in a controlled manner.

9 QSMP Simplex Formal Description

Simplex Key Exchange Sequence

Preamble

The Simplex key exchange sequence begins with the client verifying the validity of the server's public signature verification key. The client checks the expiration date of this key, and if it is found to be invalid or expired, the client initiates a re-authentication session with the server. During this session, a new key is distributed over an encrypted channel, and the client verifies the new key's certificate using the designated authentication authority or scheme implemented by the server and client software.

9.1 Connect Request

The client initiates the connection process by sending a *connection request* to the server that includes its configuration string and asymmetric public signature key identity.

Key Identity

The *key identity* (*kid*) is a multi-part, 16-byte array that acts as a public asymmetric verification key and device identification string. It is used to match the target server to its corresponding cryptographic key, ensuring that the correct key is used during the exchange.

Configuration String

The *configuration string* (*cfg*) specifies the cryptographic protocol set being used in the key exchange process. For the exchange to proceed successfully, the configuration strings used by both the client and server must match, indicating that they are using the same cryptographic parameters.

Session Cookie

To securely manage the state of the key exchange, the client generates a *session cookie* by hashing a combination of the configuration string, the key identity, and the server public asymmetric signature verification key:

$sch \leftarrow \mathrm{H}(cfg \,||\, kid \,||\, pvk)$

Where:

- *cfg* is the configuration string.
- *kid* is the key identity.
- *pvk* is the server's public signature verification key.

This session cookie (*sch*) serves as a unique identifier for the session, helping to ensure that the cryptographic parameters are consistently referenced throughout the exchange.

The client then sends the key identity string (*kid*) and the configuration string (*cfg*) to the server to initiate the connection:

C{ *kid*, *cfg* } → S

9.2 Connect Response

The server processes the client's connection request and responds with either an error message or a connect response packet. If any error occurs during the key exchange, the server generates an error packet and sends it to the remote host, which triggers a teardown of the session and network connection on both sides.

Key Verification and Protocol Check

The server begins by verifying that it has the appropriate asymmetric signature verification key that corresponds to the client's request, using the key-identity array (*kid*).

It then checks that its protocol configuration matches the one specified by the client. To securely manage the state of the exchange, the server creates a session cookie by hashing the configuration string, the key identity, and the public signature verification key:

sch ← H(*cfg* || *kid* || *pvk*)

Where:

- *cfg* is the configuration string.
- *kid* is the key identity.
- *pvk* is the server's public signature verification key.

This session cookie (*sch*) serves as a unique identifier for the session, helping maintain the integrity of the key exchange.

Asymmetric Key Generation and Signing

The server generates a new asymmetric encryption key pair and securely stores the private key. It hashes the public encapsulation key and the serialized connect response packet header, and signs

this hash using its private asymmetric signature key. The signature provides a cryptographic guarantee that the public asymmetric cipher key has not been tampered with during transmission.

Key generation and signing steps:

Generate the public (*pk*) and private (*sk*) asymmetric encryption keys.

$pk, sk \leftarrow G(\lambda, r)$

Create a hash of the public key and serialized *connect response* packet header (*sh*).

$pkh \leftarrow H(pk \parallel sh)$

Sign the hashed public key using the server's private signature key.

$spkh \leftarrow S_{sk}(pkh)$

The public signature verification key itself can be enveloped and signed using a 'chain of trust' model, such as X.509, to ensure further authentication through a signature verification extension to the protocol.

Server Response

The server sends a connect response message back to the client, containing the signed hash of the public asymmetric encapsulation key (*spkh*) and a copy of the public key itself:

$S\{ spkh, pk \} \rightarrow C$

9.3 Exchange Request

The client processes the server's connect response and initiates the next steps of the key exchange by verifying the received data, encapsulating a shared secret, and preparing the session keys.

Signature Verification and Hash Check

The client begins by verifying the signature of the hash using the server's public verification key. It then generates its own hash of the server's public key and compares it to the hash contained in the server's message. If the hashes match, the client proceeds to encapsulate the shared secret. If the hashes do not match, the key exchange is aborted.

The client uses the server's public verification key to check the hash of the public key. If the verification is successful, the process continues; otherwise, the key exchange fails.

$V_{pk}(H(pk)) \leftarrow (\text{true} ?= pk : 0)$

The public encapsulation key and connect response packet header are hashed, and the hash is compared with signed hash received from the server. Once the packet header and public key are verified, the client uses the server's public key to encapsulate a shared secret.

The client generates a ciphertext (*cpt*) and encapsulates the shared secret (*sec*) using the server's public key.

$cpt, sec \leftarrow E_{pk}(sec)$

The client combines the shared secret and the session cookie to derive the session keys and two unique nonces for the communication channels.

The Key Derivation Function (KDF) generates two session keys (*k1*, *k2*) and two nonces (*n1*, *n2*) using the shared secret (*sec*) and the session cookie (*sch*).

$k1, k2, n1, n2 \leftarrow \text{KDF}(sec \parallel sch)$:

Cipher Initialization

The receive and transmit channel ciphers are then initialized using the derived keys and nonces.

Initializes the receive channel cipher with key *k2* and nonce *n2*.

$cpr_{rx}(k2, n2)$

Initializes the transmit channel cipher with key *k1* and nonce *n1*.

$cpr_{tx}(k1, n1)$

Client Transmission

The client sends the ciphertext to the server as part of the exchange request.

The client transmits the encapsulated shared secret to the server.

$C\{ cpt \} \rightarrow S$

9.4 Exchange Response

The server processes the client's exchange request by decapsulating the shared secret, deriving the session keys, and confirming the secure communication channel.

Shared Secret Decapsulation

The server decapsulates the shared secret from the ciphertext received from the client.

The server uses its private asymmetric key to decapsulate the shared secret (*sec*) from the received ciphertext (*cpt*).

$sec \leftarrow -E_{sk}(cpt)$

Session Key Derivation

The server combines the decapsulated shared secret and the session cookie hash to derive two session keys and two unique nonces for the communication channels.

The Key Derivation Function (KDF) generates two symmetric session keys (*k1*, *k2*) and two nonces (*n1*, *n2*) using the shared secret (*sec*) and the session cookie (*sch*).

$k1, k2, n1, n2 \leftarrow \text{KDF}(sec \parallel sch)$

Cipher Initialization

The server initializes the symmetric ciphers for the receive and transmit channels.

Initializes the receive channel cipher with key *k1* and nonce *n1*.

$cpr_{rx}(k1, n1)$

Initializes the transmit channel cipher with key *k2* and nonce *n2*.

$cpr_{tx}(k2, n2)$

Server Response

The server sets the packet flag to "exchange response", indicating that the encrypted channels have been successfully established. It then sends this notification back to the client to confirm the secure communication channel.

The server sends an exchange response flag to the client, confirming that the secure tunnel is established.

$S\{f\} \rightarrow C$

The server updates its operational state to *session established*, indicating that it is now ready to securely process data over the encrypted channels.

9.5 Establish Verify

In the final step of the key exchange sequence, the client verifies the status of the encrypted tunnel based on the server's exchange response.

Client Verification

The client inspects the flag of the exchange response packet received from the server. If the flag indicates an error state, the client immediately tears down the tunnel to prevent any further data transmission. This ensures that no data is sent over an insecure or compromised connection.

If the flag does not indicate an error state, the client confirms that the tunnel is successfully established and in an operational state.

Operational State

Once the verification is complete and the tunnel is confirmed, the client updates its internal state to *session established*, indicating that the secure communication channels are fully operational. The client is now ready to process data over the encrypted tunnel.

9.6 Transmission

During the transmission phase, either the client or server sends messages over the established encrypted tunnel using the RCS stream cipher's MAC, AEAD (Authenticated Encryption with Associated Data), and encryption functions. This process ensures the integrity and confidentiality of the transmitted data.

Message Serialization and Encryption

The transmitting host (client or server) starts by serializing the packet header, which includes critical details such as the message size, timestamp, protocol flag, and sequence number. This serialized header is then added to the symmetric cipher’s associated data parameter, which adds metadata authentication to the encryption process.

The message encryption process is as follows:

1. **Encrypt the Message**: The plaintext message is encrypted using the symmetric encryption function of the RCS stream cipher. The symmetric encryption function (E_k) is applied to the plaintext message (m) to produce the ciphertext (cpt).
   $cpt \leftarrow E_k(m)$

2. **Update the MAC State**: The serialized packet header is added to the MAC (Message Authentication Code) state through the additional-data parameter of the RCS cipher.

The MAC function (M_{mk}) is updated with the serialized packet header (*sh*) and the ciphertext (*cpt*) to produce the MAC code (*mc*).
$mc \leftarrow M_{mk}(sh, cpt)$

3. **Append the MAC Code**: The MAC code is appended to the end of the ciphertext, ensuring that any tampering with the data during transmission will be detected.

Packet Decryption and Verification

Upon receiving the packet, the recipient host deserializes the packet header and adds it to the MAC state along with the received ciphertext. The MAC computation is then finalized and compared with the MAC code that was appended to the ciphertext. The packet timestamp is compared to the *UTC* time, if the time is outside of a tolerance threshold, the packet is rejected and the session is torn down.

1. **Generate the MAC Code:** Add the serialized packet header to the cipher AEAD. Add the ciphertext and generate the MAC code.
   $mc` \leftarrow M_{mk}(sh, cpt)$
   Compare the MAC tag copy with the MAC tag appended to the ciphertext.
   $mc` ?= mc$
   If the MAC check fails, indicating potential data tampering or corruption, the decryption function returns an empty message array and an error status. The application **shall** handle this error accordingly.

2. **Decrypt the Ciphertext:** If the MAC code matches, the ciphertext is considered authenticated, and the message is decrypted.
   The ciphertext (*cpt*) is decrypted back into the plaintext message (*m*) if the MAC verification succeeds.
   $m \leftarrow -E_k(cpt)$

This process ensures that the transmitted data remains confidential and tamper-evident, providing both encryption and authentication to protect the integrity of the communication. Any errors during decryption signal an immediate response to prevent the further exchange of potentially compromised data.

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
utctime	Uint64	0x40	The UTC packet creation time
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.

Functions:

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_encrypt_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.

Functions:

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 10.3 QSMP client state structure.

Functions

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)
```

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: Security Analysis

QSMP is designed to protect against a range of threats, including both classical and quantum attacks. This section provides a detailed analysis of its security features, highlights potential attack vectors, and describes how the protocol mitigates these risks.

11.1 Post-Quantum Cryptography

QSMP utilizes cryptographic primitives that are specifically designed to withstand the computational power of quantum computers. It implements the asymmetric ciphers Kyber and McEliece, and the signature schemes Dilithium and Sphincs+. These algorithms are recommended by the NIST Post-Quantum Cryptography standardization process in the case of the signature schemes, and the Kyber cipher. We added McEliece, the third round candidate for its excellent long-term security potential.

- **Quantum Resistance:** The cryptographic algorithms employed in QSMP are chosen to resist attacks by quantum computers, particularly Shor's algorithm, which can break classical encryption methods like RSA and ECC.
- **Algorithm Flexibility:** QSMP's design supports multiple post-quantum algorithms, which enhances its adaptability. In the event that a vulnerability is discovered in one algorithm, the protocol can easily switch to another secure alternative.

11.2 Forward Secrecy and Predictive Resistance

QSMP incorporates robust mechanisms to ensure that past and future communications remain secure, even if a key is compromised.

- **Forward Secrecy:** The use of ephemeral asymmetric keys in both SIMPLEX and DUPLEX exchanges ensures that each session's key is independent of previous keys. This means that even if an attacker gains access to the current session's keys, they cannot decrypt any previous session data.
- **Predictive Resistance:** The asymmetric ratcheting mechanism ensures that future a state cannot be derived from the current key state. This prevents attackers from calculating or predicting future session keys, even if they have access to the current keys on a long running tunnel implementation.

11.3 Ratcheting Mechanism

QSMP uses both asymmetric and symmetric ratcheting to inject new entropy into the encrypted communication channel, enhancing its security.

- **Asymmetric Ratchet:** Periodically re-keys the symmetric encryption using new asymmetric key exchanges, providing a higher level of security. This approach ensures that even if an attacker gains access to one set of keys, they cannot compute the next set of keys.

- **Symmetric Ratchet:** Provides a lightweight method for continuously re-keying the encryption stream based on hash values derived from current session keys. This technique is efficient and allows for quick recovery if a key compromise is detected.

11.4 Man-in-the-Middle (MITM) Attack Mitigation

QSMP implements strong authentication techniques to counter MITM attacks.

- **Digital Signatures:** Messages are signed using the sender's private key, and these signatures are verified by the recipient using the sender's public key. This method prevents attackers from tampering with or spoofing messages.
- **Public Key Authentication:** The protocol's use of signature verification keys ensures that attackers cannot impersonate another party, as they cannot forge the digital signatures without the corresponding private key.

11.5 Replay Attack Prevention

To prevent replay attacks, QSMP employs a valid-time timestamp, nonce values, and sequence numbers for each message.

- **Timestamp:** Each packet during the key exchange and during tunnel operation has a low resolution (seconds) timestamp added to the packet header. The packet header itself is added to signature hashes during the key exchange, and to the additional data field of the symmetric cipher during tunnel exchanges. This guarantees that the packet header has not been altered, and that the packet cannot exceed a timeout threshold (60 seconds by default) or the packet is discarded.
- **Sequence Numbers:** Sequence numbers are included with each message to prevent replay attacks, ensuring that old messages cannot be resent to gain unauthorized access. These are added to the MAC AAD input of the sessions symmetric stream cipher (RCS).
- **Message Size:** During the key exchange, the packet flag and message size are checked on each message, if the message size is not exactly what is expected by that stage of the exchange, the key exchange is aborted and the connection is torn down.

11.6 Resistance to Key Compromise

QSMP's use of ephemeral keys and ratcheting provides resilience against key compromise, ensuring that a breach of one key does not affect other sessions.

- **Ephemeral Key Generation:** Each session generates a new ephemeral key pair, meaning that even if one session's key is compromised, it does not compromise other sessions.
- **Key Expiry and Replacement:** Keys have defined expiration times, which are checked upon every use, prompting regular re-authentication and generation of new keys, which mitigates risks associated with long-term key reuse.

11.7 Error Handling and Security Considerations

Effective error handling is a critical part of QSMP's security strategy, ensuring minimal information is leaked to attackers.

- **Error Codes and Logging:** Error messages are intentionally limited to only a nondescript packet flag value to prevent attackers from gaining insights into the protocol's state or operation, which could aid in crafting more sophisticated attacks.
- **Session Tear-Down on Error:** When a critical error is detected (e.g., signature mismatch or decryption failure), the session is immediately terminated, logged, and further communication is halted to prevent exploitation.

Potential Attack Vectors and Mitigations

Attack Vector	Description	QSMP Mitigation
Quantum Attacks	Exploitation of quantum algorithms to break classical cryptosystems.	Uses post-quantum algorithms like Kyber, McEliece, SPHINCS+, and Dilithium.
Man-in-the-Middle (MITM)	An attacker intercepts and manipulates communication between two parties.	Digital signatures and public key verification are used.
Replay Attacks	Re-sending a previously captured message to gain unauthorized access.	Session tokens, timestamps, and sequence numbers prevent replays.
Key Compromise	Access to encryption keys by adversaries due to theft or malware.	Ephemeral keys and symmetric and asymmetric key ratcheting provide forward secrecy.
Side-Channel Attacks	Attacks exploiting information from hardware or software leakage.	Components are written to be timing neutral, including support functions, asymmetric, and symmetric primitives.
Cryptographic Downgrade Attacks	Forcing the use of weaker cryptographic algorithms.	Protocol negotiation ensures only post-quantum algorithms are used. The key exchanges themselves do not support handshake negotiations.
Denial-of-Service (DoS) Attacks	Overloading the server with requests to disrupt communications.	Error handling and session timeout mechanisms mitigate this risk.

Summary

The security architecture of QSMP is robust and forward-looking, integrating post-quantum cryptography, ratcheting techniques, strong authentication, and comprehensive error handling to guard against a wide range of classical and quantum threats. By employing these techniques, QSMP ensures that it remains resilient in the face of evolving attack vectors and advances in computational capabilities.

12: Design Decisions

QSMP has been carefully crafted with several strategic design choices to ensure robust security and adaptability for future developments in cryptography and network protocols. This section outlines the rationale behind the key design decisions that shape QSMP's implementation.

12.1 Networking Protocol Considerations

While the accompanying example code for QSMP is built upon the Transport Control Protocol (TCP), it is important to note that the choice of networking protocol is considered to operate at a layer beneath the QSMP protocol itself. QSMP could utilize TCP, UDP, or even a custom IP stack to transport packets.

- **Layered Flexibility:** The flexibility to use different transport protocols allows for future enhancements that may include custom IP stack implementations with features such as windowing controls, packet buffers, and other advanced networking controls tailored to specific use cases.
- **Example Simplicity:** The current implementation using TCP was intentionally kept simple to provide clarity for those studying or adopting QSMP. This choice aligns with common practices in many widely-used VPN software implementations, which also use TCP to avoid the complexities associated with custom IP stacks.

12.2 Protocol Negotiation

QSMP intentionally omits protocol negotiation for a number of reasons, despite the relative ease of implementation.

- **Security Integrity:** Protocol negotiation is often misused to reduce the security level of communications to the lowest common denominator, which undermines the integrity of the protocol suite. By avoiding this, QSMP maintains a consistent security posture across all implementations.
- **Current Algorithm Support:** QSMP supports three asymmetric configurations: Dilithium-Kyber, Dilithium-McEliece, and SphincsPlus-McEliece. The parameter sets corresponding to NIST 128, 192, and 256-bit security (S1, S3, and S5) are implemented, allowing for granular security controls via the different parameter sets. Although additional asymmetric algorithms sets may be added in the future, the current structure already offers robust security without the need for protocol negotiation.

12.3 Signature Chaining

QSMP does not implement signature chaining directly but allows for this functionality to be integrated via secondary protocols such as X.509.

- **Optional Feature:** Signature chaining is not a core feature of QSMP since the protocol is designed as a standalone secure tunneling system. However, for implementations where

additional layers of authentication are required, signature chaining can be added using existing standards. This would be done by enveloping the QSMP verification key in a secondary authentication scheme like X.509.
- **Integration with Existing Systems:** Public keys can be distributed using X.509 or other "web of trust" mechanisms. This additional authentication step can provide greater assurance in key-exchange processes when necessary.

12.4 Compact Packet Headers

QSMP's packet headers were designed to be highly efficient, significantly smaller than standard SSH-2 protocol headers, with a size of just 21 bytes.

- **Optimization for Efficiency:** By eliminating unnecessary fields and limiting integer sizes to ranges that reflect realistic use cases, the protocol reduces overhead and increases efficiency for real-time applications like SSH.
- **Scalable Design:** The use of a single byte for flags and a 32-bit unsigned integer for the message size parameter ensures scalability without exceeding the requirements of most use cases, where payload sizes typically remain below 4 GB. This could however be easily changed with just a couple small adjustments to the QSMP header file.

12.5 Dual-Channel Communication System

QSMP employs a two-channel communication system, with each channel independently keyed, to maximize security.

- **512-bit Secure Tunnel:** The duplex mode creates a 512-bit secure encrypted tunnel. Combined with an aggressive choice of asymmetric protocols and parameter sets, this will likely never be broken.
- **Separate Key Generation:** Each host independently generates the keys for the channels it uses to transmit data. This practice is a significant enhancement over protocols that use a single shared secret for both transmit and receive channels.
- **Avoiding Security Shortcuts:** While some protocols opt for single key exchanges to simplify operations, QSMP deliberately avoids these shortcuts to ensure the highest level of security.

12.6 Post-Quantum Authenticated Stream Cipher (RCS)

QSMP uses the RCS (Rijndael Cryptographic Stream) cipher, which is based on a wide-block transformation (256-bit) of the Rijndael cipher (AES) and includes several enhancements.

- **Advanced Features:** RCS utilizes cSHAKE for round key generation, KMAC for authentication, and features 22 transformation rounds compared to the AES-256 14 rounds. It also has a 512-bit secure option used by the duplex mode, utilizing 30 rounds of encryption
- **Forward-Thinking Security:** The decision to use RCS over more established ciphers like AES or ChaCha reflects a proactive stance toward post-quantum security. QSMP

chooses stronger cryptographic primitives now to mitigate future risks posed by advances in quantum computing and future cryptanalytic breakthroughs.

12.7 Long-Term Security Vision

QSMP is designed with the future of computing technology in mind, anticipating the significant advances that are almost certain to arise.

- **Future-Proofing:** The protocol prioritizes long-term security by employing cryptographic techniques that are resilient not just against current threats but also those that may emerge as quantum computing evolves.
- **Focus on Strong Security:** QSMP's design aims to keep sensitive data secure for decades to come, ensuring that it remains robust in the face of unknown future developments in cryptography and computer capabilities.

Summary of Design Principles

The decisions that guide QSMP's design emphasize security, adaptability, and efficiency. By focusing on post-quantum cryptographic techniques, dual-channel communication, and streamlined packet structures, QSMP aims to provide the highest possible security in a protocol that is both versatile and forward-compatible. The absence of protocol negotiation and the choice to employ RCS over older ciphers highlight QSMP’s commitment to proactive defense against emerging threats, ensuring that it remains a resilient solution for secure messaging in the quantum era.