Multi Party Domain Cryptosystem - Interior – MPDC-I

Revision 1b, October 26, 2024

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

This document is an engineering level description of the MDPC authenticated network domain crypto-system. This document describes the interior network protocol MPDC-I, a multi-party cryptographic key exchange and network security system.

**Contents Page**

[Foreword](#Foreward) 2

1: [Introduction](#Introduction) 2

2: [Scope](#Scope) 4

3: [References](#References) 5

4: [Terms and Definitions](#Terms_and_Definitions) 7

5: [Protocol Description](#Protocol_Description) 10

6: [Mathematical Description](#Mathematical_Description) 25

7: [Security Analysis](#Security_Analysis) 50

8: [Application Scenarios](#Application_Scenarios) 56

9: [MPDC Cryptanalysis](#MPDC_Cryptanalysis) 59

10: [Internal Functions](#Internal_Functions) 65

# **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 first revision of the specification of MPDC-I, further revisions may become necessary during the pursuit of a standard model, and revision numbers shall be incremented with changes to the specification. The reader is asked to consider only the most recent revision of this draft, as the authoritative expression of the MPDC-I specification.

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

MPDC-I, the algorithm constituting the MPDC-I domain crypto-system is patent pending, and is owned by John G. Underhill and the QRCS Corporation.

# **1.** **Introduction**

MPDC-I is a multi-party key exchange and network security system. It distributes the security of a key exchange between a server and a client across multiple devices. Network ‘agents’ contribute a portion of pseudo-random material to client-server session keys.

On an interior network, servers and clients exchange a shared secret with each agent on the network using an authenticated asymmetric key exchange. The secret is kept for the lifetime of the devices certificate, and used to generate a unique key-stream to encrypt a small amount of pseudo-random data. This data is called a ‘key fragment’. Fragments are combined and hashed to form the primary session keys used between the server and the client to initialize an encrypted tunnel. In this way, agents on the network that have been authenticated to both the server and client, can inject entropy into a key exchange through multiple independent cryptographic processes.

There can be any number of agents on a network, and each one has a certificate signed by the root domain server. Any attack that utilizes impersonation or ‘man-in-the-middle’ strategies, would need to simultaneously impersonate multiple network devices. The number of agent servers that contribute entropy to a client-server key exchange is unlimited, the generation of a key fragment and fragment encryption use computationally ‘cheap’ symmetric cryptography, and can scale so that even the most sophisticated impersonation attacks are practically impossible.

Unlike other multi-party key exchange schemes being considered, which use expensive classical asymmetric cryptographic schemes, MPDC-I explores an asymmetric/symmetric post-quantum secure cryptographic hybrid, that can provide the security, as well as the scalability and computational economy necessary if a system of this kind is to be considered for wide-scale adoption.

**Problem Description:**

In modern cryptographic systems, the security of key exchanges is increasingly threatened by advanced classical and emerging quantum attack vectors, many of which exploit weaknesses in randomness generation. Multi-party key exchange protocols that incorporate multiple independent sources of entropy provide a robust defense against these threats. By leveraging contributions from diverse entropy providers, such as hardware RNGs, network entropy beacons, and distributed nodes, the protocol ensures high-quality, unbiased randomness. This approach mitigates risks associated with single-point entropy failures, state recovery attacks, and entropy manipulation, significantly enhancing the unpredictability of the shared key.

Such enhanced key exchanges are crucial for post-quantum security, as quantum adversaries can exploit weak or deterministic entropy with powerful algorithms like Grover’s and Shor’s. By distributing the entropy contributions, the attack surface is widened, making it infeasible for a quantum attacker to compromise the entire pool of randomness. Furthermore, the inclusion of multiple entropy sources provides resilience against side-channel attacks, precomputation attacks, impersonation attacks, and replay attempts, making the scheme well-suited for secure communications in critical infrastructure, federated applications, and next-generation decentralized systems. As the threat landscape evolves, integrating multiple dedicated sources of entropy into key exchange protocols will be vital for ensuring long-term, quantum-resistant security.

**Design Requirements:**

The distributed security system is computationally economical, with functions in the primary key exchange and tunnel being performed solely by symmetric cryptography.

That asymmetric functions be constrained to network control messaging, and device registration and initialization.

Certificates are used as a means to authenticate devices and the messages they produce during device initialization and network operations. Each device generates its own asymmetric signature key-pair, and retains the secret signing key. Each device uses the signature verification key to create a certificate which must be signed by the root security server, the trust anchor for the domain.

The network must be scalable, expensive asymmetric operations must be constrained to registration and key exchange with participating devices, after which operations become administrative, and devices use the minimal network and hardware resources to function.

The system must be designed to be a form of authenticated key distribution with no tolerance for failure. Any failure in the exchange between nodes in the scheme, whether it be authentication or the distribution of keys, packet values, or symmetric or asymmetric authentication failure, causes the failure of the exchange, and the collapse of the circuit.

* 1. **Purpose**

MPDC-I provides a distributed security provisioning across multiple autonomous devices.

The MPDC-I crypto-system, has been designed in such a way that:

1. The keying material used in the exchange is distributed across multiple autonomous devices, strongly mitigating the threat of MITM attacks.
2. Uses an advanced authentication system, across multiple core devices, and a hierarchal certificate scheme for authentication.
3. That the model must be scalable, computationally efficient, and provide strong security guarantees against a wide range of classical and quantum attacks.

# **2.** **Scope**

This document describes the MPDC-I (Multi Party Domain Crypto-system - Interior Network) protocol, which is used to establish an encrypted and authenticated duplexed communications stream between a server and a host. The protocol is described in this document, and references to the example C implementation are available, including specific settings and software components necessary to its design.

The MPDC-I protocol is part of the MPDC protocol set; the interior protocol manages security at a domain level, whereas the MPDC-E protocol is the exterior protocol, that connects MPDC networks, authenticates internal and external certificates using a distributed trust model, and facilitates ‘key injection’ across trusted domains. The MPDC-E protocol is being developed, and will appear as a future publication with a separate protocol definition.

**2.1 Application**

The MPDC-I protocol is intended for institutions that implement secure communication streams used to encrypt and authenticate secret information exchanged between a server and a host.

The network design, key exchange functions, authentication and encryption of messages, and control message exchanges between devices defined in this document must be considered as mandatory elements in the construction of an MPDC-I network. Components that are not necessarily mandatory, but are the recommended settings or usage of the protocol will be denoted by the key-words **SHOULD**. In circumstances where strict conformance to implementation procedures is required but not necessarily obvious, the key-word **SHALL** will be used to indicate compulsory compliance is required to conform to the specification, likewise warnings indicating changes to the specification that are prohibited will be notated with **SHALL NOT**.

## **3.** **References**

**3.1 Normative References**

**3.1.1 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>

**3.1.2 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>

**3.1.3 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>

**3.1.4 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>

**3.1.5 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>

**3.1.6 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>

**3.1.7 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>

* 1. **Multi Party Cryptographic References**

**3.2.1 Threshold Cryptography by Yvo Desmedt (1994)**  
Introduces threshold cryptography for secure, distributed cryptographic operations.  
<https://onlinelibrary.wiley.com/doi/10.1002/ett.4460050407>

**3.2.2 Secure Computation with Minimal Interaction by Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner (2012)**  
Proposes protocols for secure two-party computation with minimal interaction.  
<https://eprint.iacr.org/2013/552.pdf>

**3.2.3 Efficient Secure Two-Party Computation Using Symmetric Cut-and-Choose by Wenliang Du and Mikhail Atallah (2001)**  
Presents an efficient protocol for secure two-party computation using cut-and-choose.

**3.2.4 SPDZ: An Efficient MPC Protocol for Dishonest Majority by Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias (2012)**  
Describes the SPDZ protocol for efficient multi-party computation with dishonest majority.  
<https://eprint.iacr.org/2011/535.pdf>

**3.2.5 Overdrive: Making SPDZ Great Again by Marcel Keller, Emmanuela Orsini, and Peter Scholl (2018)**  
Presents optimizations to SPDZ for improved efficiency and practicality.  
<https://eprint.iacr.org/2017/1230.pdf>

**3.3 Standards and Initiatives**

**3.3.1 NISTIR 8214A: Towards NIST Standards for Threshold Schemes for Cryptographic Primitives: A Preliminary Roadmap**  
Provides a roadmap towards NIST standards for threshold cryptography schemes.  
<https://csrc.nist.gov/publications/detail/nistir/8214a/final>

**3.3.2 ISO/IEC 11770-5:2011: Information technology, Security techniques, and Key management, Part 5: Group key management**  
Defines procedures for key management in secure group communications.  
<https://www.iso.org/standard/54527.html>

**3.3.3 IETF RFC 9380: The Messaging Layer Security (MLS) Protocol**  
Specifies the MLS protocol for secure and scalable group communication.  
<https://datatracker.ietf.org/doc/rfc9380/>

**3.3.4 IEEE P1363.3: Standard for Identity-Based Cryptographic Techniques using Pairings**  
Defines identity-based cryptographic techniques leveraging pairings.  
<https://standards.ieee.org/standard/1363_3-2013.html>

**3.3.5 ISO/IEC 15946 Series: Cryptographic Techniques Based on Elliptic Curves**  
Specifies cryptographic techniques based on elliptic curve algorithms.  
<https://www.iso.org/standard/56026.html>

## **4.****Terms and Definitions**

**4.1 Cryptographic Primitives**

**4.1.1 Kyber**

The Kyber asymmetric cipher and NIST Post Quantum Competition winner.

**4.1.2 McEliece**

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

**4.1.3 Dilithium**

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

**4.1.5 SPHINCS+**

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

**4.1.6 RCS**

The wide-block Rijndael hybrid authenticated symmetric stream cipher.

**4.1.7 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.

**4.1.8 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.

**4.1.9 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.

**4.2 Network References**

**4.2.1 Bandwidth**

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

**4.2.2 Byte**

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

**4.2.3 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.

**4.2.4 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.

**4.2.5 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.

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

**4.2.7 IP Address**

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

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

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

**4.2.10 LAN (Local Area Network)**

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

**4.2.11 Latency**

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

**4.2.12 Network Topology**

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

**4.2.13 Packet**

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

**4.2.14 Protocol**

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

**4.2.15 TCP/IP (Transmission Control Protocol/Internet Protocol)**

A suite of communication protocols used to interconnect network devices on the internet.

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

**4.2.17 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.

**4.2.18 VLAN (Virtual Local Area Network)**

A logical grouping of network devices that appear to be on the same LAN regardless of their physical location.

**4.2.19 VPN (Virtual Private Network)**

Creates a secure network connection over a public network such as the internet.

# **Protocol Description**

The Multi-Party Domain Cryptosystem – Interior Protocol (MPDC-I) is a cryptographic protocol designed to facilitate secure communication between entities in a domain. It leverages a combination of public-key cryptography, symmetric cryptography, and certificate management to establish an encrypted tunnel between participating devices. MPDC-I is engineered with both classical and quantum-resistant security in mind, utilizing robust cryptographic primitives to ensure confidentiality, integrity, and authentication of an encrypted communications stream.

**5.1 Objectives**

**The primary objectives of MPDC-I are:**

1. **Establish Secure Communication Channels:** Use public-key cryptography, certificate management, and entropy injection to create secure communications channels between participating devices.
2. **Ensure Forward Secrecy and Post-Quantum Resistance:** Provide security against both classical and quantum attacks in key exchange operations.
3. **Flexibility and Scalability:** Adapt to various network environments, including IoT, enterprise, and critical infrastructure.
4. **Prevent Common Cryptographic Attacks:** Defend against man-in-the-middle (MITM), replay, and key compromise attacks while ensuring integrity and authenticity.
5. **Provide a scalable and efficient MPC scheme:** Create an Multi Party Cryptographic scheme that is highly scalable, relatively lightweight, and computationally efficient.

**5.2 Key Components and Their Roles**

**MPDC operates with five key devices:**

1. **Client**: Initiates communication and key exchanges with the MPDC enabled Application Server.
2. **MAS (MPDC Application Server)**: Acts as the server managing communications with Clients. The MAS is an application server on the local network, this can be a file server, database server, or any other type of network resource used by Clients that requires a secure connection.
3. **Agent**: A trusted network device that injects entropy into the key exchange process.
4. **DLA (Domain List Agent)**: The network authority managing device registration and certificate validation.
5. **RDS (Root Domain Security)**: The root authority responsible for signing and managing device certificates.

**5.2.1 Client**

**Role**: An end-user network device that initiates secure communication with the MAS.

**Functions**:

* Generates a certificate and stores the secret signing key.
* The certificate is signed by the RDS, directly or by proxy through the DLA.
* Exchanges *master fragment keys* (*mfk*) with Agents and MAS servers, to facilitate fragment key encryption.
* Combines *key fragments* provided by the Agents along with the MAS fragment key, which are used to derive a set of secure session keys.
* Encrypts and decrypts messages using the session keys in a duplexed encrypted and authenticated tunnel.

**5.2.2 MAS (Application Server)**

**Role**: Central server managing secure communications with Clients.

**Functions**:

* Generates a certificate and stores the secret signing key.
* The certificate is signed by the RDS, directly or by proxy through the DLA.
* Validates Client certificates using the RDS root certificate.
* Communicates with the Agents to obtain key fragments.
* Derives the session key to securely interact with the Client.
* Encrypts and decrypts messages using the session keys in a duplexed encrypted and authenticated tunnel.

**5.2.3 Agent**

**Role**: Provides additional entropy to the key exchange process.

**Functions**:

* Generates a certificate and stores the secret signing key.
* The certificate is signed by the RDS, directly or by proxy through the DLA.
* Generates key fragments (entropy) for session key generation.
* Securely transmits key fragments to both the MAS and Client.
* Enhances the randomness and security of the session key.

**5.2.4 DLA (Domain List Agent)**

**Role**: Manages device registration and certificate validation.

**Functions**:

* Generates a certificate and stores the secret signing key.
* The certificate is signed by the RDS.
* Validates device certificates against the RDS certificate.
* Maintains a master list of trusted devices (network topology).
* Distributes certificates and updates to devices.
* Manages device certificate revocation and resignation.
* Handles topological queries from network devices.

**5.2.5 RDS (Root Domain Security Server)**

**Role**: Acts as the certificate authority for the network.

**Functions**:

* Generates and manages the root certificate (trust anchor).
* Signs device certificates to verify identity and authenticity.
* Can connect to the DLA enabling a certificate signing proxy function.

**5.3 Network Initialization**

**5.3.1 Root Server Initialization**

**Root Certificate Generation**:

* The RDS generates its signature key-pair (public/private keys).
* Creates a public root certificate containing its signature verification key, serial number, issuer, configuration set, version, and expiration period.
* Securely stores the private key used for signing.

**5.3.2 DLA Initialization**

**Certificate Generation**:

* The DLA generates its signature key-pair.
* Creates a public certificate and stores the secret signing key.
* The RDS signs the DLA's certificate, establishing it as a trusted entity.

**Network Management**:

* The DLA begins managing device registrations and maintaining the network topology.

**5.3.3 Device Initialization (Agent, Client, MAS)**

**Certificate Generation and Signing**:

* Each device generates its own signature key-pair and certificate.
* Certificates are signed by the RDS directly or by proxy via the DLA.
* The RDS signs each device's certificate, establishing trust.

**Registration with DLA**:

* Devices register with the DLA, which validates their certificates.
* Devices are added to the network topology maintained by the DLA.
* Devices build partial copies of the topology, with knowledge of only the devices with which they interact.

**5.3.4 MAS and Agent Integration**

**MAS Integration**:

* The MAS contacts the DLA to join the network.
* Receives a list of available Agents from the DLA.
* Establishes secure channels with each Agent through an asymmetric key exchange that exchanges *master fragment keys*.

**Agent Integration**:

* Agents exchange *master fragment keys* with Clients and MAS devices using an asymmetric key exchange.
* Agents establish secure communication with the MAS and Clients, using the *mfk* keys to encrypt key fragments.

**5.3.5 Client Integration**

**Client Integration**:

* The Client joins the network by registering with the DLA.
* Receives a list of available Agent and MAS servers.
* Exchanges certificates with Agent and MAS servers.
* Exchanges *mfk* keys with Agents and MAS servers.

**5.4 Network Initialization**

MPDC network devices are initialized in a sequence:

1. RDS – Trust anchor
2. DLA – Network management
3. Agents – Entropy provider
4. MAS – Application server
5. Clients – End user device

The root security server (RDS) signs the certificate of each device, either directly or once the DLA is initialized, through the DLA proxy signing feature.

Each device generates its own asymmetric signature verification/signing keypair. The public signature verification key is a member of the certificate that each device generates independently. Certificates and signing keys are the sole responsibility of the device itself, and only the originating device has knowledge of the secret signing key.

The master fragment encryption keys *mfk*, shared between the devices, is used to derive fragment encryption keys (*efk*), ephemeral keys which encrypt key fragments exchanged between devices.

When the certificate expiration time is exceeded, the *mfk* becomes invalid and a new certificate and master fragment key must be exchanged.

The maximum expiration time set in a certificate must not exceed the root certificate expiration time.

When a certificate is signed by the root, the certificate is hashed, and the hash is signed by the root signing key. The root signed hash is added to the child certificate, as well as the root certificate serial number, and if the user defined expiration time exceeds that of the root, the expiration time is set to the root’s expiration time. No device certificate can have an expiration time that exceeds the root certificate’s expiration time. Once the root certificate has expired, devices on the network must renew their certificates, and rejoin the network.

Each exchange in MPDC, whether it is a network message, part of the key exchange, or traffic on the encrypted tunnel, all of these functions use a ***packet* *valid-time*** feature. This adds the UTC time in **seconds** to the packet header at the point of packet creation. If the time in the packet valid-time parameter received by the remote host exceeds the packet valid-time field by the **packet time threshold** (60 seconds by default), the message is deemed invalid and the circuit is torn down.

The packet creation timestamp and packet sequence number are added to the signature hash on network messages, where the packet message is hashed along with the valid-time timestamp and the packet sequence number, then signed by the devices asymmetric signing key.

During the tunnelling phase, the sequence number and packet creation time (*st*) are added to the *additional data* function of the symmetric cipher MAC used to encrypt and authenticate messages in the encrypted tunnel (the AEAD authenticated stream cipher RCS). In this way, message replay attacks are strongly mitigated, and all MPDC messaging is protected from attack schemes that use packet header tampering, message alteration, or re-transmission of packet data.

**5.4.1 Root certificate creation**

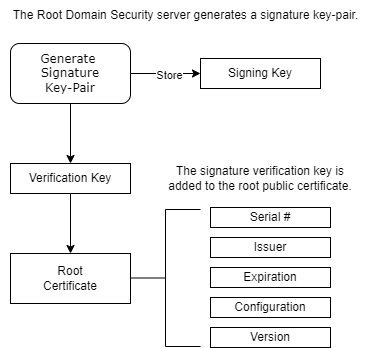


Figure 5.4.1 Root certificate generation.

The RDS generates a signature key-pair, stores the secret signing key, and adds the public signature verification key to the root certificate. The root certificate is made up of the following fields:

* The **signature** **verification key**, used to verify a root signature.
* The **issuer string**, identifies the certificate identity and formal name.
* The **serial number**, a unique 128-bit string used to identify the certificate.
* The **expiration time**, the valid *to* and *from* times, the time period during which the certificate is valid.
* The **configuration set** name, identifies the cryptographic primitives used by the key exchange from a set.
* The **version number**, the MPDC protocol version number.

The serial number and issuer fields identify the certificate and the originating device.

The expiration time is the starting time and expiration time of the certificate in UTC seconds from the epoch. All certificates signed by the root, expire when the root expires.

The algorithm set name identifies which cryptographic set is used in the implementation, this can be the combination of asymmetric cipher and signature scheme families; Kyber-Dilithium, McEliece-Dilithium, and McEliece-SPHINCS+, further subdivided by the parameter sets used by each cipher and signature scheme.

The version number ensures that local and remote versions are synchronized.

The RDS root certificate is distributed to every device on the network and installed during device initialization, cached by those devices and used to authenticate certificates signed by the root domain security server.

**5.4.2 DLA Initialization**

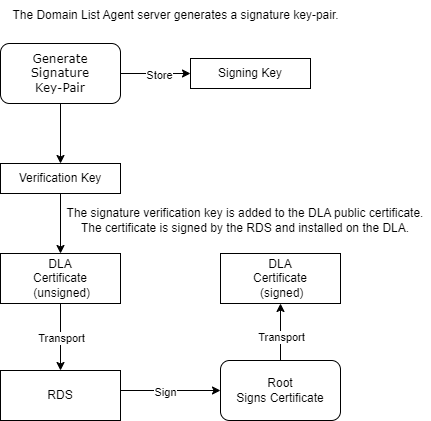


Figure 5.4.2a DLA certificate initialization

The DLA and all other child certificates have two additional parameters to the root certificate, the signature parameter which holds a copy of the RDS signed hash of the child certificate, and the root certificate serial number parameter.

Child certificate parameters:

* The **certificate signature**, generated by hashing the certificate, and signing the hash with the RDS signature key.
* The **root serial number** of the RDS server that signed this certificate.
* The **signature** **verification key**, used to verify a message signed by the corresponding signing key.
* The **issuer string**, identifies the certificate’s origin identity and formal readable network name.
* The **serial number**, a unique 128-bit string used to identify the certificate.
* The **expiration time**, the valid *to* and *from* times, the time period during which the certificate is valid.
* The **configuration set** name, identifies the cryptographic primitives used by the key exchange from a set.
* The **version number**, the MPDC protocol version number.

Once the DLA certificate has been signed by the RDS server, the DLA server can be brought online and is ready to handle registration requests and other administrative duties.

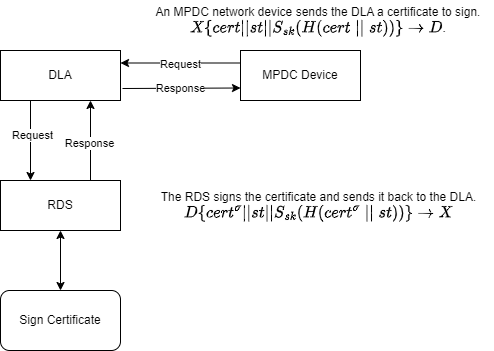


Figure 5.4.2b DLA proxy signing

The DLA certificate can be loaded onto the RDS to enable the proxy signing feature.

The RDS server is deliberately isolated, it has only one message capability, and this is to remotely sign a certificate as requested *only* by the DLA server. The DLA can act as a proxy for the signing of device certificates, allowing the isolation of the root server from other network devices. The RDS stores the DLA certificate, and can only accept signing requests that have been issued and signed by the DLA.

**5.4.3 Agent Initialization**

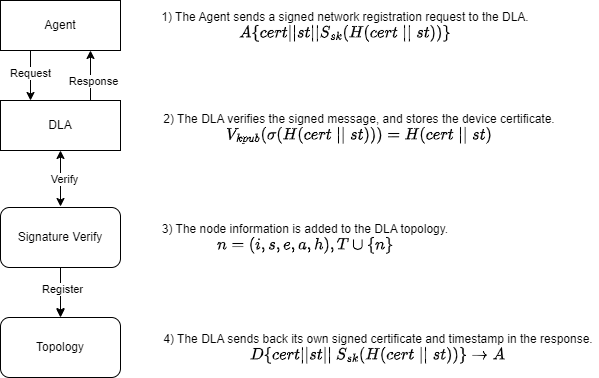


Figure 5.4.3 Agent network registration.

The Agent sends the DLA a registration request. The DLA verifies the signature field of the Agent certificate using the RDS public certificate. A hash of the certificate is compared to the signature hash to validate the certificate.

The registration request message has been signed by the Agent signing key, the message is authenticated, a message hash is generated and compared to the signature hash.

If the Agent certificate and the message have been validated, the certificate is stored, and the certificate is used to populate a topological node structure, which is added to the DLA topological database.

**5.4.4 MAS Initialization**

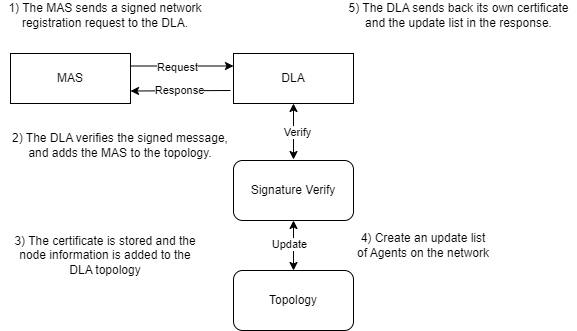


Figure 5.4.4a MAS network registration.

The MAS sends the DLA a registration request. The DLA verifies the signature field of the Agent certificate using the RDS public certificate. A hash of the certificate is compared to the signature hash to validate the certificate.

The registration request message has been signed by the MAS signing key, the message is authenticated, a message hash is generated and compared to the signature hash.

If the MAS certificate and the message have been validated, and the timestamp and sequence number are correct, the certificate is stored, and the certificate is used to populate a topological node structure, which is added to the DLA topological database.

The DLA assembles an update list for the MAS. The list contains the node information for every Agent on the network. The topological node contains all of the information that the MAS requires to contact and verify the Agent; IP address, the certificate serial number, issuer name, certificate expiration time, and the certificate hash. When the MAS contacts these devices and receives their certificates, the certificate hash value in the topological node is compared to a hash of the certificate. The serial number, issuer name, expiration time, and IP address must all match the values in the topological node received from the DLA.

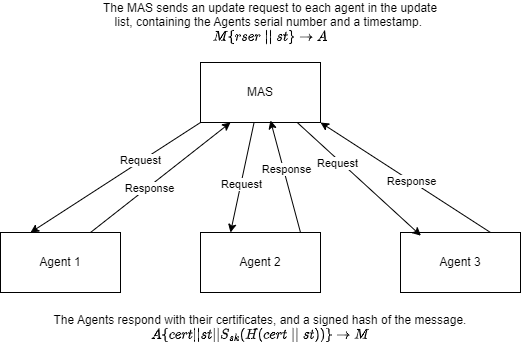


Figure 5.4.4b MAS agent update.

The MAS contacts each of the Agents in the DLA update list, and exchanges certificates. The certificates signatures are verified, and the certificate is hashed and compared to the signature hash. The verified certificates are stored on the MAS, and topological node entries for each Agent are added to the MAS topological database.

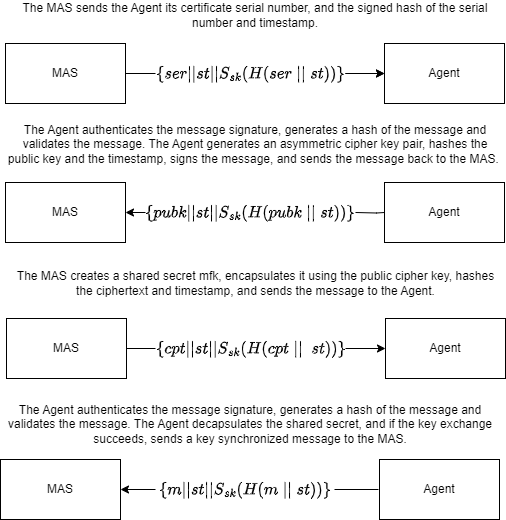


Figure 5.4.4c MAS to agent mfk exchange.

The MAS then sends each Agent a signed *mfk* *key exchange* request. The Agent generates an asymmetric cipher key-pair and timestamp, signs the public key and timestamp, and sends it to the MAS.

The MAS verifies the signed key and timestamp and encapsulates a shared secret, hashes the ciphertext and signs the hash. The signed ciphertext is sent back to the Agent, which verifies the signed hash, and decapsulates the shared secret.

The shared secret *mfk* key, is associated with the device certificate serial number of the relative remote device in an internal list, and stored on the Agent and MAS.

**5.4.5 Client Initialization**

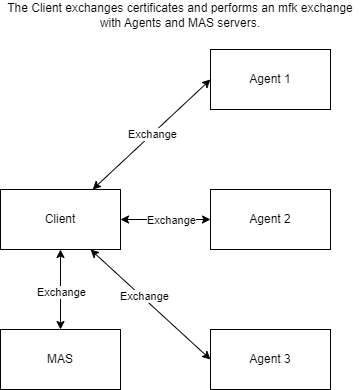


Figure 5.4.5 Client to MAS and Agent certificate and mfk exchange.

The Client registration undergoes an identical exchange of certificates and ***mfk*** keys with each Agent. The update list the DLA prepares for a Client also contains a list of MAS servers. The Client contacts each MAS server in the update list and exchanges certificates and master fragment keys.

Once the Clients have been initialized, the network is considered *synchronized*, and ready for encrypted tunnel connections between Clients and MPDC application servers.

**5.5 Key Exchange and Encrypted Tunnel**

The Client initiates a key exchange with an MPDC application server (MAS). The MAS server and the Client have previously exchanged master fragment keys, which are used to derive fragment encryption key (*efk*) used to encrypt pseudo-random fragments keys.

**5.5.1 Fragment Collection Request**

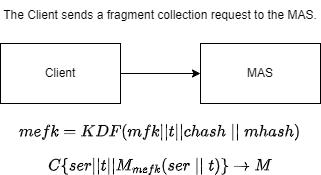


Figure 5.5.1 Client to MAS connection request.

The Client creates a fragment collection request, the Client-to-MAS *mfk* key, a random token, and the Client and MAS certificate hashes are used to key a KDF (cSHAKE), and create a MAC key. The serial number and token are added to the message, and a MAC code is created by hashing the message and key.

**5.5.2 MAS Fragment Request**

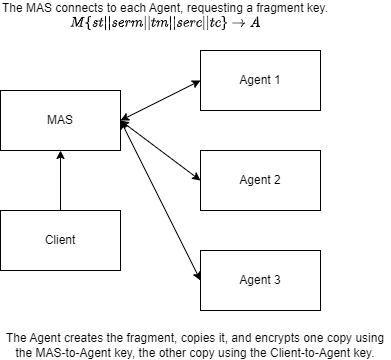


Figure 5.5.2 Agent to MAS agent response.

The MAS connects to each Agent in the topology, and requests a fragment key.

The Agents respond with a fragment key pairing, one copy encrypted with a fragment encryption key (*efk*) derived using the *mfk* shared between the MAS-to-Agent, the other copy encrypted with an *efk* key derived using the Client-to-Agent *mfk*.

**5.5.3 MAS Key Generation**



Figure 5.5.3 MAS to Client fragment transfer.

The MAS server decrypts its copy of the fragment received from each Agent, and generates a MAS-to-Client fragment key. The MAS encrypts the MAS-to-Client fragment using a fragment encryption key (*efk*) derived from the shared Client-to-MAS *mfk*, and bundles this key with the Agent fragment keys that were encrypted using Client-to-Agent *efk* derived from the *mfk* keys corresponding to each of the Agent responders.

The MAS sends the Client the encrypted fragment key set.

The MAS combines the fragments as input to a key derivation function (cSHAKE), and generates the MAS-to-Client session keys. The symmetric cipher (RCS) receive and transmit cipher instances are initialized, the tunnel is raised and ready to transmit data.

**5.5.4 Tunnel Establishment**

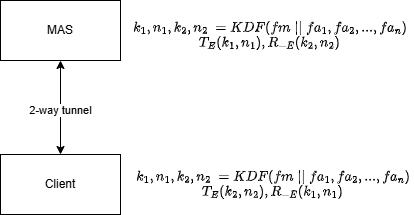


Figure 5.5.4 MAS to Client tunnel establishment.

The Client receives the encrypted fragment key bundle from the MAS. The Client derives the *efk* keys and decrypts the Agent key fragments, and derives the Client-to-MAS *efk* and decrypts the Client-to-MAS key fragment. The fragments are added to the KDF (cSHAKE) which generates the session keys for the transmit and receive channels of the encrypted tunnel. The symmetric cipher (RCS) instances are initialized, and the tunnel interfaces are raised and ready to transmit data.

**6.** **Mathematical Description**

MPDC uses various messages between devices to accomplish network tasks.

The DLA handles network control messaging, including certificate revocation, network convergence, certificate announcements, topological queries, registration and resignation messages.

Messages are also passed between Agents, MAS servers, and Clients, such as certificate updates, master fragment key exchanges, and fragment collection.

All messages are signed using the senders secret asymmetric signing key, and are verified by the receiving device using the senders’ certificate. This not only guarantees the authenticity of the sender, but a packet creation time and sequence number are included in the message hash that is signed by the originating device, protecting the message from replay attacks.

This section contains a list of message functions used by MPDC-I, and their mathematical descriptions.

**6.1 Announce Broadcast**

**Overview:**

Network announce is an administrative event broadcast from the DLA. The DLA announces a new Agent to nodes on the network. It broadcasts the new agent’s certificate, which is signed by the root, and signs the message with the DLA signing key. The receiving device validates the DLA signature and message hash, validates the root signature and parameters of the certificate, and checks that the packet timestamp is within the valid-time threshold. If the message is validated, the receiver adds the new device to its topology list, stores the certificate, and initiates a *master fragment key* exchange, trading shared secrets with the remote Agent device.

**API:**

* *mpdc\_network\_announce\_broadcast()*
* *mpdc\_network\_announce\_response()*

**Applies to:**

* Client
* DLA
* MAS

**Mathematical Description:**

**Let:**

* *CD*σ ​ be the root signed certificate of device *D*.
* H be the hash function.
* H*CS*σ be the signed certificate and timestamp hash.
* *Kpri* be the private signing key.
* *Kpub* be the signature verification key.
* *σ* be an asymmetric signature.
* *Sign* be the asymmetric signing function.
* *st* be the sequence number and valid-time timestamp.
* Verify be the asymmetric signature verification function.

The broadcast includes the certificate and the signature:

H*CS*σ= Sign*dlaKpri​​*(H(*CD* σ*​* || *st*))

Broadcast(*CD*σ)*=* (*CD* σ*​* ∥H*CS*σ)

Devices receiving this broadcast will verify the signature using the DLA’s public verification key:

Verify*dlaKpub*(H*CS*σ)*=* H(*CD*σ || *st*)

If the broadcast message is validated, the certificate is added to the devices certificate store, and the device exchanges a *master fragment key* with the new Agent.

**Proof of Security:**

**Correctness**: The broadcast is correctly signed by the DLA, ensuring that any recipient can verify the signature using DLA public key. The verification succeeds if and only if *σ*(H(*CD* || *st* )) was produced using the DLA private key, providing assurance of authenticity.

**Proof**: Given the definition of digital signatures:

σ(H(*CD*σ|| *st*)) = Sign*Kpri​​*(H(*CD*σ*​* || *st*))

The verification function computes:

Verify*Kpub*(σ(H(*CD*​σ || *st*))) = H(*CD*σ​ || *st*)

Since Verify*Kpub* is the inverse of Sign*Kpri*, the signature is valid if it was signed by the matching private key.

**Integrity**: Since H(*CD*σ || *st*) is hashed and signed, any change to the certificate or signature would cause the verification to fail. The hash function used (e.g., SHAKE) is collision-resistant, ensuring that an attacker cannot forge *CD*​σ or σ(H(*CD*σ || *st*)).

**Replay Protection**: A timestamp and sequence number are included in the hash (*st*) and checked to ensure it is within a specified valid timeframe, so that broadcasts cannot be reused maliciously.

**6.2 Converge Broadcast**

**Overview:**

Network convergence is an administrative event called from the DLA. Each MAS server and Agent on the network is sent a copy of their topological node database entry. The serialized node entry for the remote device is hashed along with a timestamp and sequence number, and the hash is signed by the DLA and sent to the device.

The signature is verified by the device using the DLA's public certificate, the local node entry is serialized and hashed, and compared with the signed hash. If the hashes match, the entry in the DLA topological database is *synchronized*, if the entries do not match, the device serializes the current topological database entry and the certificate, signs them with the current signature key, which is signed by the root (RDS), and sends it back to the DLA. The DLA verifies the new certificate using the RDS certificate. The old entry is purged, a new topological entry is added to the database, and the new certificate is stored.

\* Note that the proper procedure after a certificate update on a MAS or Agent, is to resign from the network, and then rejoin with a new certificate.

**API:**

* *mpdc\_network\_converge\_request()*
* *mpdc\_network\_converge\_response()*

**Applies to:**

* Agent
* DLA
* MAS

**Mathematical Description:**

**Let:**

* H*TS*σ be the signed hash of H(*TD* || *st*) signed using *D’*s private key.
* H be the hash function.
* *Kpri* be the secret signing key.
* *Kpub* be the signature verification key.
* *Sign* be the asymmetric signing function.
* *st* be the sequence number and valid-time timestamp.
* *TD* ​ be the topological node of device *D*.
* Verify be the asymmetric signature verification function.

**The converge broadcast request:**

The DLA creates the converge request using the remote device’s topological node, hashed with the timestamp and signed.

H*TS* σ = Sign*dlaKpri​​*(H(*TD​* || *st*))

Request(*TD*​) = (*TD*​ ∥ H*TS*σ)

The device verifies the DLA’s signature.

Verify*dlaKpub*(H*TS*σ) = H(*TD* || *st*)

**The converge response:**

The responding device signs the response message, and sends it to the DLA.

*σ* = Sign*respKpri​​*(H(*TD​* || *st*))

Response(*TR*​) = (*TR* ​∥ σ)

**Proof of Security:**

**Correctness**: The response is only generated if the request is valid. Both the request and the response signatures are verified using the public key of the respective device.

**Proof**: The request signature is:

σ(H(*TD*​ || *st*)) = Sign*Kpri*(H(*TD*​ || *st*))

Upon receiving the request, the recipient checks the validity of the signature using:

Verify*Kpub*(σ(H(*TD*​ || *st*))) = H(*TD*​ || *st*)

If the signature verification passes, the recipient knows the request is authentic. The node structure sent by the DLA, containing information about the remote device including certificate serial number, issuer, and expiration *to* and *from* times, is verified by the receiving device. If the node values match, the receiver signs its serialized node structure along with the timestamp and sequence number, and sends it back to the DLA as confirmation that the topology is aligned. If the values do not match, or the authentication or message is invalid, the receiver sends back an error message. If the DLA receives an error, or the connection times out, the remote node is removed from the DLA’s topology, and the device’s certificate is revoked, removing it from the topology list of every device on the network.

**Integrity**: The hash H(*TD* || *st*) ensures that the certificate cannot be altered. Any tampering will result in a failed signature verification.

**Replay Protection**: A timestamp and sequence number are included in the hash H(*TD*​ || *ts*) and checked to ensure that broadcasts cannot be reused maliciously.

**6.3 FKey Request**

**Overview:**

The FKey request is reserved for MPDC-E, it is used when the Inter Domain Gateway (IDG) is requesting a fragment key for a device on a remote network, as part of the cross-domain trusted entropy ‘borrowing’ that can be configured between trusted domains.

**API:**

* *mpdc\_network\_fkey\_request()*
* *mpdc\_network\_fkey\_response()*

**Applies to:**

* Agent
* IDG

**Mathematical Description:**

The key fragment request and response ensure secure transmission of key fragments.

**Let:**

* *CD*σ ​ be the root signed certificate of the requesting device.
* *FD*​ be the key fragment requested.
* *H* be the hash function (cSHAKE).
* H*CS*σ be the signed certificate and timestamp hash.
* *Sign* be the asymmetric signing function.
* *st* be the sequence number and valid-time timestamp.
* Verify be the asymmetric signature verification function.

**The FKey request:**

H*CS*σ = Sign*idgKpri​​*(H(*CD*σ*​* || *st*))

Request(*CD*​) = (*CD*​σ ∥ H*CS*σ)

**The FKey response:**

Verify*idgKpub*(H*CS*σ) = H(*FD* || *st*)

H*FS*σ = Sign*idgKpri​​*(H(*FD**​* || *st*))

Response(*FD*​) = (*FD* ​∥ H*FS*σ)

This ensures the integrity and authenticity of the key fragment *FD*​.

**Proof of Security:**

**Confidentiality**: The key fragment *FD* ​ is securely transmitted and signed, ensuring that it cannot be intercepted or modified.

**Proof**: The key fragment is signed using the sender's private key:

σ(H(*FD* || *st*)) = SignKpri(H(*FD* || *st*))

Upon receiving the fragment, the requesting device verifies:

Verify*Kpub*(σ(H(*FD* || *st*))) = H(*FD* || *st*)

This ensures the key fragment's integrity and authenticity.

**Replay Protection**: A timestamp and sequence number are included in the hash H(*FD*​ || *ts*) and checked to ensure that broadcasts cannot be reused maliciously.

**6.4 Fragment Collection (Primary Client-to-MAS Tunnel)**

**Overview**

The process begins when a Client sends a fragment key collection request to the MAS (MPDC enabled Application Server). This involves multiple symmetric-based key exchanges and cryptographic operations between the Client, MAS, and network Agents. The objective is to securely gather and validate key fragments from the Agents, derive shared session keys, and establish an encrypted tunnel between the Client and the MAS.

**API:**

* *mpdc\_network\_fragment\_collection\_request()*
* *mpdc\_network\_fragment\_collection\_response()*
* *mpdc\_network\_fragment\_query\_response()*

**Applies to:**

* Agent
* Client
* MAS

**Step-by-Step Description**

**Client Request to MAS**:

The Client initiates the fragment collection by sending a request to the MAS. This request includes:

* The Client's certificate serial number.
* A random token generated by the Client.

The MAS generates its own random token, and sends queries to every Agent in the topology.

**Fragment Queries to Agents**:

The MAS queries all Agents in the network by sending a fragment key request. Each query includes:

* The Client's certificate serial number and random token.
* The MAS's certificate serial number and random token.

If any Agent fails to respond or returns an error, the entire session is terminated.

**Agent Dual Token Encryption**:

The random token is copied and encrypted twice, one copy for the MAS the other for the Client:

* One copy is encrypted using a fragment encryption key (*efk*) derived from the MAS-to-Agent *mfk* fragment encryption session key.
* The other copy is encrypted using an *efk* derived from the Client-to-Agent *mfk* fragment encryption session key.

The MAS decrypts its copy of the fragment, and forwards the Client-to-Agent encrypted copies to the Client as a set. This fragment key-set includes a fragment generated by the MAS server and encrypted with the Client-to-MAS *mfk* derived fragment encryption key.

**Deriving Session Keys**:

The MAS decrypts the fragment keys it has received from the network Agents. The MAS generates a fragment key, and adds this key, and the decrypted Agent fragment keys to a KDF, which generates the session keys used to initialize symmetric cipher instances (RCS) for the transmit and receive channels of the Client-to-MAS tunnel.

The Client performs the same operations, decrypting the MAS fragment key, the Agent fragment keys, and using a KDF to derive the symmetric cipher session keys.

**Mathematical Description:**

**Let:**

* *cf* be the encrypted fragment key.
* E/-E be the encryption and decryption function.
* *efk* be the fragment encryption key.
* *frag* be the key fragment.
* KDF be the key derivation function (cSHAKE).
* *lhash* be the hash of the local certificate.
* M be the MAC function.
* *mfk* be the master fragment key, a shared master secret between two devices.
* *rhash* be the hash of the remote certificate.
* *ser* be the device certificate serial number.
* *tok* be a random session token.
* *st* be the timestamp and sequence number.

The Client calculates the Client-to-MAS *fragment encryption key* (*efk*). The token is randomly generated, added with the Client-to-MAS shared *master fragment key* (*mfk*), the MAS certificate hash, and the Client certificate hash. The KDF generates a key used to initialize the MAC function, which MACs the request message.

= KDF(|| *rhashmas* || *lhashclient*|| *tokclient*)

The Client sends the fragment collection request to the MAS containing its serial number and the random token. The serial number and token are MAC’d using the derived fragment encryption key to initialize the MAC function. The *STD* message is the client’s certificate serial number, the Client generated random token, and a MAC ***tag*** derived from the message and key. The packet creation timestamp and sequence number are also added to the MAC.

*tag* = M(*ser* || *tok* || *st* )

Request(*STD*) = (*ser* || *tok* || *tag* )

The MAS calculates the Client-to-MAS fragment encryption key, and checks the message MAC. If the MAC validates the message, and the timestamp and sequence number are correct, both Client and MAS have calculated their session fragment keys. If the MAC fails the MAS sends the Client an error message and the circuit is torn down.

M(*ser* || *tok* || *st* ) = *tag* ⟺ True

The MAS connects to each agent in its topological map, and requests a key fragment. The MAS *RM* request is composed of the MAS certificate serial number and random token, and the Client serial number and token is the *RC* state.

*RM* = (*sermas*|| *tokmas*)

*RC* = (*serclient* || *tokclient*)

The request is the pair of serial numbers and unique tokens for both MAS and Client, and the message MAC tag, derived from the message and key. The message is sent out to each Agent on the network, if the Agent is non-responding or returns an error, the key exchange is aborted.

*tag* = M(*RM* || *RC* || st)

Request(*RM, RC*) = (*RM* || *RC* || *tag*)

Where each Ai is an agent server in the topology:

Request(*RM, RC*) ⟶{A1​, A2​, …, An​}

Each Agent generates a random fragment key, makes a copy, and encrypts them both, the first copy is encrypted using the MAS-to-Agent fragment encryption key, the second using the Client-to-Agent fragment encryption key.

The first *efk* is derived from the MAS-to-Agent *mfk*, the MAS random token, and the Agent and MAS certificate hashes. This fragment key Encrypts the MAS copy of the key fragment. The creates two keys, the first is the fragment encryption key, the second is the key used to MAC the entire message, which will be verified by the MAS.

= KDF(|| *rhashmas* || *lhashagent*|| *tokmas*)

*cf1* = E(*frag*)

The second fragment is encrypted using the Client-to-Agent derived key.

= KDF(|| *rhashclient* || *lhashagent*|| *tokclient*)

*cf2*= E(*frag*)

The ciphertext from both encrypted key sets are MAC’d and the MAC ***tag*** is added to the message. The MAC key is the second half of the (512-bit size) *efk* key.

tag = M(*cf1* || *cf2* || *st* )

Agent(*cf1* || *cf2* || *tag* )⟶MAS.

The MAS verifies the mac tag against the ciphertext, the sequence number and timestamp, and if they are correct, decrypts its portion of the key-set.

M(*cf1* || *cf2* || *hdr* ) = *tag* ⟺ True

The MAS copies the encrypted Client fragment keys and Agent serial numbers to a key-set. Once the MAS has collected keys from every Agent, it sends the set of encrypted Client-to-Agent keys back to the client, with each fragment encrypted with the respective Client-to-Agent *efk*.

**Where:**

* *fset* is the set of agent fragment keys.
* *as* is the agent serial number.
* *fc* is the encrypted fragment key.

*fset* = { F1(*as1* || *fc1*), F2(*as2*|| *fc2*), ..., Fn(*asn* || *fcn*) }

The encrypted key-set is sent to the Client, where like on the MAS, the serial number is used to look up the corresponding Agent *mfk*, derive the fragment encryption key, and verify and decrypt the fragment key sent by each agent, along with a key fragment shared between the MAS and the Client.

For each *i* ∈ { 1, 2, …, *n* }, *fi* ​= -E(*cfi* )

All fragments are added to the hash to create a set of session keys used between the MAS and the Client to establish an encrypted tunnel. The fragment keys are added to the KDF input, including the fragment generated by the MAS for the Client:

*k1, k2, n1, n2*= KDF( *f1*, *f2*, ... *fn*)

This generates the session keys for the transmit and receive channels used to create a bi-directional encrypted tunnel between the MAS and the Client. The symmetric cipher instances (RCS) used to encrypt data on the receive and transmit channels of the encrypted tunnel are initialized on both the Client and MAS, and the tunnel interfaces are raised and ready to transmit data.

Session =

**Proof of Security:**

**Correctness**: Client and MAS servers share a secret exchanged during the *master fragment key* exchange. The Client and MAS also share unique secrets with every Agent on the network. This 256-bit secret key is combined with a random session token, and hashes of the local certificate and the remote certificate. The hash result is a fragment encryption key:

*efk* = KDF(*mfk* || *rhash* || *lhash*|| *tok* )

The combination of certificate hashes will be unique between devices, this along with the random token which acts as a session nonce, ensures that every session derives unique fragment encryption keys. This *efk* is XOR’d with the *key fragment*; a 256-bit pseudo-random string generated by each agent.

*cf* = E*efk*(*frag* )

**Proof**: Given a cryptographically strong key derivation function, specifically cSHAKE, the mixing of these inputs will produce a key-stream that is highly diffused and unique to each session. That key-stream mixed with the random fragment (XOR) will produce output that is indistinguishable from random, and highly resistant to differential analysis techniques.

Key fragments are input into the KDF, along with the Client-to-MAS session key. The KDF outputs keys and nonces for the two symmetric cipher instances, that will be the transmit and receive channels of the encrypted tunnel between the Client and the MAS.

*k1, k2, n1, n2*= KDF( *f1*, *f2*, ... *fn* )

Utilizing key fragments from Agents on the network, hardens the security of the server-to-client exchange. The injection of entropy into the key derivation, extends the mathematical hardness of differential analysis. By distributing the generation of the key across multiple autonomous devices on the network, impersonation, replay, and man-in-the-middle attacks become more problematic in proportion to the number of devices contributing to key generation.

**6.5 Incremental Update**

**Overview:**

The incremental update functions retrieve a devices certificate. When a device joins the network, the DLA sends a list of resources available for that device. When a MAS joins the network the DLA sends it a list of network Agents, when a Client joins the DLA sends a list of Agents and MAS servers.

The Client and MAS synchronize with devices on the list sent by the DLA, creating a topological database. The topology is a local list containing information about resources that the device uses on the network. A topological node is an element in the list that contains important information like the nodes IP address, issuer, expiration time, certificate hash and serial number. This information is used to connect to the device, request its certificate, verify the certificate, and interact with the device on the network.

The **mpdc\_topology\_node\_state** structure defines the state information for a device within the MPDC topology. This includes details like network address, certificate information, and the device's designation.

Once the device has obtained the certificate and added the node to its topology, the device can exchange a shared secret between devices using the master fragment key (*mfk*) asymmetric key exchange.

During network registration, the Client and MAS device receive a list of resources they will use on the network.

The Client or MAS queries each node on this list, requesting the devices public certificate. The requestor uses the remote devices serial number *SD* as the request message.

**API:**

* *mpdc\_network\_mfk\_exchange\_request()*
* *mpdc\_network\_mfk\_exchange\_response()*

**Applies to:**

* Agent
* Client
* MAS

**Let:**

* *CD*σ ​ be the root signed certificate of device *D*.
* *st*  be the sequence number and valid-time timestamp.
* *σ* be the asymmetric signature.
* H be the hash function.
* H*CS*σ be the signed certificate and timestamp hash.
* *Kpri* be the private signing key.
* *Kpub* be the signature verification key.
* *serD* be the requested certificate serial number.
* *Sign* be the asymmetric signing function.
* *st* be the sequence number and packet creation timestamp.
* Verify be the asymmetric signature verification function.

**The incremental update request:**

The device sends an incremental update request with the remote device certificate serial number.

Request(*SD*) = (*serD*)

The responding device sends the serialized certificate, and a hash of the certificate and the packet headers valid-time timestamp and sequence number, signed with its secret signing key.

**The incremental update response:**

H*CS*σ = Sign*respKpri​​*(H(*CD*σ*​* || *st*))

Response(*CD*σ​) = (*CD*σ ​∥ H*CS*σ)

The certificate signature is verified and a hash of the certificate is compared to the signed hash, and the hash contained in the topological node entry. The certificate hash must match the hash stored in the node information sent by the DLA. If the certificate is validated, it is added to the devices certificate store.

Verify*respKpub*(H*CS*σ) = H(*CD*σ || *st*)

**Proof of Security:**

**Correctness**: The response is only generated if the request is valid and the serial number in the request matches the responder’s certificate serial number. The responder’s certificate is verified by the requestor using the root public certificate. The response message signature is verified using the received public key of the respective device.

**Proof**: The response signature is:

σ(H(*CD*​σ || *st*)) = SignKpri​​(H(*CD*σ​ || *st*))

Upon receiving the request, the recipient checks the validity of the certificates’ signature using:

Verify*rootKpub*(σ(H(*CD*​))) = H(*CD*​)

The response message including the responder’s certificate and valid-time timestamp are then verified using the validated responder’s certificate.

Verify*devKpub*(σ(H(*CD*σ || *st*))) = H(*CD*σ || *st*)

If the root signature verification passes, the certificate is authentic. The certificate is then used to authenticate that the message is valid and sent by the responding device. If any of these checks fail; root signature, responder message signature, hashes, sequence, packet creation valid-time, or the certificate hash comparison with the node hash value sent by the DLA, the certificate is rejected.

**Integrity**: The hash H(*CD*σ || *st*) ensures that the certificate cannot be altered. Any tampering will result in a failed signature verification.

**Replay Protection**: A timestamp and sequence number are included in the hash H(*CD*​ || *ts*) and checked to ensure that the requests cannot be reused maliciously.

**6.5 Master Fragment Key Exchange**

**Overview:**

The master fragment key exchange, is an authenticated asymmetric key exchange, where a shared secret is exchanged between devices. A Client and a MAS exchange *master fragment keys* (***mfk***), and both Client and MAS exchange master fragment keys with Agent servers.

**API:**

* *mpdc\_network\_mfk\_exchange\_request()*
* *mpdc\_network\_mfk\_exchange\_response()*

**Applies to:**

* Agent
* Client
* MAS

**Mathematical Description:**

**Let:**

* *CD*σ ​ be the root signed certificate of device *D*.
* *ct* be the asymmetric cipher-text.
* Enc be the asymmetric encapsulation function.
* Dec be the asymmetric decapsulation function.
* H be the hash function.
* H*CS*σ be the signed certificate and timestamp hash.
* H*ES*σ be the signed asymmetric ciphertext and timestamp hash.
* H*PS*σ be the signed public cipher key and timestamp hash.
* KGen be the asymmetric cipher key generation function.
* *Kpub* be the asymmetric signature public key.
* *Kpri* be the asymmetric signature private key.
* *mfk* be the master fragment key.
* *pk* be the asymmetric cipher public key.
* *sk* be the asymmetric cipher secret key.
* Sign is the asymmetric signing function.
* *ss* be the shared secret.
* Verify is the asymmetric verification function.

The requestor sends an exchange request to the device. The message contains the requestors serialized certificate, and a valid-time timestamp.

Note: Agents do not retain Client or MAS certificates.

H*CS*σ = Sign*reqtKpri*(H(*CD*σ || *st*))

Request(*CD*σ) = (*CD*σ || ​ H*CS*σ)

The responder verifies the certificates root signature.

Verify*rootKpub*(*CD*σ) = H(*CD*​)

The responder validates the requestors certificate and the valid-time timestamp are then verified using the validated responder’s certificate.

Verify*devKpub*(H*CS*σ) = H(*CD*σ || *st*)

The responder generates a keypair using the asymmetric cipher. It stores the private key, hashes and signs the public cipher key and valid-time timestamp, and sends it to the requestor.

*pk, sk =* KGen(*λ, r*)

H*PS*σ = Sign*respKpri*(H(*pk* || *st*))

Response(*pk*​) = (*pk*​ ∥ H*PS*σ)

The signed public key is sent to the requestor. The signature, hash, and timestamp are verified, and the requestor uses the public key to encapsulate a shared secret.

Verify*respKpub*(H*PS*σ) = H(pk || *st*)

*ct* = Enc*pk*(*ss*)

The shared secret is retained by the requestor and is the *master fragment key*. The ciphertext is hashed along with the valid-time timestamp, and the hash is signed by the requestors signing key.

H*ES*σ = Sign*reqtKpri*(H(*ct* || *st*))

Request(*ct*​) = (*ct*​ ∥ H*ES*σ)

The responder verifies the message hash using the requestors public verification key, then compares the hash against the hashed ciphertext and timestamp.

Verify*devKpub*(H*ES*σ) = H(*ct* || *st*)

If the ciphertext is validated, the ciphertext is decrypted using the responders private cipher key.

*ss* = Dec*sk*(*ct*)

**Proof of Security:**

**Correctness**: The key exchange consists of three steps:

* The requestor sends a signed hash of its certificate and timestamp to the responder.
* The responder signs a hash of the public cipher key and timestamp and sends it to the requestor.
* The requestor signs a copy of the ciphertext and timestamp and sends it to the responder.

**Proof**: Given the definition of digital signatures and the message *m*:

σ(H(m|| *st​*)) = Sign*Kpri​​*(H(m​ || *st*))

The verification function computes:

Verify*Kpub*(σ(H(m​ || *st*))) = H(m​ || *st*)

Since Verify*Kpub* is the inverse of Sign*Kpri*, the signature is valid if it was signed by the matching private key. The hash is generated from the message and compared to the signed hash for equality.

**Integrity**: Since H(m || *st*) is hashed and signed, any change to the certificate or signature would cause the verification to fail. The hash function used (e.g., SHAKE) is collision-resistant, ensuring that an attacker cannot forge *CD*​ or σ(H(m || *st*)).

**Replay Protection**: A timestamp and sequence number are included in the hash H(m​ || *ts*) and checked to ensure that broadcasts cannot be reused maliciously.

**6.6 Registration Request**

**Overview:**

An Agent registers with the DLA to join an MPDC network. The DLA verifies the agents certificate, then sends a copy of its own root-signed certificate, and adds the device to the topology.

**API:**

* *mpdc\_network\_register\_request()*
* *mpdc\_network\_register\_response()*

**Applies to:**

* Agent
* DLA

**Mathematical Description:**

**Let:**

* *CD*σ be the root signed device certificate.
* H be the hash function.
* H*CS*σ be the signed certificate and timestamp hash.
* *Kpri* be the private asymmetric signing key.
* *Kpub* be the public asymmetric verification key.
* Sign be the asymmetric signing function.
* σ be the signature.
* Verify be the asymmetric signature verification function.

The Agent requestor sends a *register request* to the device. The message contains the requestors serialized certificate, and a signed hash of the certificate and the valid-time timestamp.

**The registration request:**

H*CS*σ = Sign*agentKpri*(H(*CD*σ || *st*))

Request(*CD*​σ) = (*CD*σ ​ ∥ H*CS*σ)

The DLA responder validates the requestors certificate and the valid-time timestamp are then verified using the validated responder’s certificate.

Verify*rootKpub*(σ(H(*CD*​)) = H(*CD*​)

Verify*agentKpub*(H*CS*σ) = H(*CD*σ || *st*)

**The registration response:**

The DLA hashes and signs its certificate and valid-time timestamp and sends it to the Agent.

H*CS*σ = SigndlaKpri(H(*CD*σ || *st*))

Response(*CD*​σ) = (*CD*​σ ∥ H*CS*σ)

The Agent verifies and stores the DLA certificate, generates a topological node for the DLA, and is registered on the network.

Verify*rootKpub*(σ(H(*CD*​)) = H(*CD*​)

Verify*dlaKpub*(H*CS*σ) = H(*CD*σ || *st*)

**6.7 Register Update Request**

**Overview:**

When a Client or MAS registers with the DLA to join an MPDC network. The DLA verifies the devices certificate, then sends a list of topological nodes that are available for that device, a copy of its own root-signed certificate, and adds the device to the topology.

**API:**

* *mpdc\_network\_register\_update\_request()*
* *mpdc\_network\_register\_update\_response()*

**Applies to:**

* Client
* DLA
* MAS

**Mathematical Description:**

**Let:**

* *CD*σ be the root signed device certificate.
* H be the hash function.
* H*CS*σ be the signed certificate and timestamp hash.
* H*CLS*σ be the signed certificate, list, and timestamp hash.
* *Kpri* be the private asymmetric signing key.
* *Kpub* be the public asymmetric verification key.
* *list* be the list of nodes.
* Sign be the asymmetric signing function.
* σ be the signature.
* Verify be the asymmetric signature verification function.

The requestor sends a *register update request* to the DLA. The message contains the requestors serialized certificate, and a signed hash of the certificate and the valid-time timestamp.

**The registration update request:**

H*CS*σ = Sign*reqtKpri*(H(*CD*σ || *st*​))

Request(*CD*σ) = (*CD*σ​ ∥ H*CS*σ)

The DLA responder validates the requestors certificate root signature and the valid-time timestamp are then verified using the validated responder’s certificate.

Verify*rootKpub*(σ(H(*CD*​)) = H(*CD*​)

Verify*devKpub*(H*CS*σ) = H(*CD*σ || *st*)

The DLA generates a list of topological nodes for the device; MAS servers receive a list of Agent servers, and Clients receive the list of Agent and MAS servers.

The DLA hashes and signs the list, its certificate, and valid-time timestamp and sends it to the Agent.

**The registration update response:**

*list* = { D1, D2, … Dn } where D*i* is a topological node.

H*CLS*σ = Sign*dlaKpri*(H(*CD*σ || *list* || *st*))

Response(*CD*​σ || *list*) = (*CD*​σ || *list* ∥ H*CLS*σ)

The requestor verifies and stores the DLA certificate, generates a topological node for the DLA, and is registered on the network. The requestor adds the list of nodes to the topological list, and will synchronize certificates with each device using the *incremental update* function, and then exchange master fragment keys using the *master fragment key exchange*. Once the device has the certificate and master fragment key of each device, its topology is considered *synchronized*.

Verify*rootKpub*(σ(H(*CD*​)) = H(*CD*​)

Verify*dlaKpub*(H*CLS*σ) = H(*CD*σ || *list* || *st*)

**6.8 Remote Signing Request**

**Overview:**

The root domain security (RDS) server only has a single networked function. Remote signing allows *only* the DLA to connect to the RDS, to act as a proxy for certificate signing. The DLA can sign certificates for devices on the network by connecting to the RDS, and forwarding the certificate to be signed. The RDS has a copy of the DLA certificate, allowing it to verify the signing request message.

**API:**

* *mpdc\_network\_remote\_signing\_request()*
* *mpdc\_network\_remote\_signing\_response()*

**Applies to:**

* DLA
* RDS

**Mathematical Description:**

**Let:**

* *CD* be a device certificate.
* H be the hash function.
* H*CS*σ be the signed certificate and timestamp hash.
* *Kpri* be the private asymmetric signing key.
* *Kpub* be the public asymmetric verification key.
* Sign be the asymmetric signing function.
* σ be the signature.
* Verify be the asymmetric signature verification function.

The DLA sends a *remote signing request* to the RDS. The message contains the serialized certificate to be signed, and a signed hash of the certificate and the valid-time timestamp.

**The remote signing request:**

H*CS*σ = SigndlaKpri(H(*CD* || *st*))

Request(*CD*​) = (*CD*​ ∥ H*CS*σ)

The RDS validates the DLA’s remote signing request signature, the certificate hash, and the valid-time timestamp.

Verify*dlaKpub*(H*CS*σ) = H(*CD* || *st*)

The RDS signs the certificate, and sends it back to the DLA.

**The remote signing response:**

*CD*​σ = SignrootKpri(H(*CD*))

Response(*CD*​σ)

The DLA verifies the root signature, and can now forward the certificate to the network device.

Verify*rootKpub*(σ(H(*CD*​)) = H(*CD*​)

**6.9 Resign Request**

**Overview:**

A Client, MAS, or an Agent can resign from the network by sending a resign request to the DLA. The DLA sends out a revoke request broadcast removing the device’s certificate and nodal information from every node on the network.

**API:**

* *mpdc\_network\_resign\_request()*
* *mpdc\_network\_resign\_response()*

**Applies to:**

* Agent
* Client
* DLA
* MAS

**Mathematical Description:**

**Let:**

* H be the hash function.
* H*SS*σ be the signed serial number and timestamp hash.
* *Kpri* be the private asymmetric signing key.
* *Kpub* be the public asymmetric verification key.
* *list* be the list of nodes.
* Sign be the asymmetric signing function.
* σ be the signature.
* Verify be the asymmetric signature verification function.

The requestor sends a *resign request* to the DLA. The message contains the requestors certificate serial number, and a signed hash of the serial number and the valid-time timestamp.

**The resignation request:**

H*SS*σ = Sign*devKpri*(H(*SD* || *st*))

Request(*SD*​) = (*SD*​ ∥ H*SS*σ)

The DLA looks up the serial number in its topology, loads the device certificate and validates the signed message.

Verify*devKpub*(H*SS*σ) = H(*SD* || *st*)

The requesting device erases its topology, and must make a *register request* to the DLA to rejoin the network. The DLA sends a *revocation broadcast* to a subsect of relevant nodes on the network.

**6.10 Revoke Broadcast**

**Overview:**

The revocation request is a broadcast message that instructs nodes on the network that a certificate has been revoked, and that device is to be removed from the network. Network members that receive this message, delete the devices certificate and remove it from the local topological database.

**API:**

* *mpdc\_network\_revoke\_broadcast()*
* *mpdc\_network\_revoke\_response()*

**Applies to:**

* Agent
* Client
* DLA
* MAS

**Mathematical Description:**

**Let:**

* H be the hash function.
* H*SS*σ be the signed serial number and timestamp hash.
* *Kpri* be the private asymmetric signing key.
* *Kpub* be the public asymmetric verification key.
* *list* be the list of nodes.
* *SD* be the device certificate serial number.
* Sign be the asymmetric signing function.
* *st* be the sequence number and valid-time timestamp.
* σ be the signature.
* Verify be the asymmetric signature verification function.

The DLA sends a *revoke request* to a subset of nodes on the network depending on the device type being revoked:

* Agent revocations are sent to Client and MAS devices.
* MAS revocations are sent to Agent and Client devices.
* Client revocations are sent to Agent and MAS devices.

The revocation message contains a signed copy of the device certificate serial number to be revoked.

H*SS*σ = Sign*dlaKpri*(H(*SD* || *st*))

Request(*SD*​) = (*SD*​ ∥ H*SS*σ)

The DLA sends the revocation out to a list of devices.

L = { D1, D2, …, Dn }

For each *i* ∈ L = Broadcast(Li, (*SD*​ ∥ σ))

**6.11 Topological Query Request**

**Overview:**

The Client-requestor sends the hashed and signed issuer string of a remote Client node and the local certificate serial number to the DLA.

Clients are not updated with each other’s certificates during network registration. This is meant to scope topology information to the smallest number of nodes required for a given device. Clients can connect to other Clients, by querying the DLA for a remote Clients node information. The Client sends the DLA the remote Client’s network (issuer) name, and the DLA returns that Client’s topological node information to the requestor.

The Client sends its serial number, and the remote nodes issuer string, which is composed of the network name, device name, and certificate extension. The query interface takes only the device name, which is resolved to the issuer string for the request. The DLA uses the certificate serial number to load the requestors certificate, and verify the signature. The requesting Client receives the remote Clients node information, and uses it to synchronize certificates, and exchange master fragment keys.

**API:**

* *mpdc\_network\_topological\_status\_request()*
* *mpdc\_network\_topological\_status\_response()*

**Applies to:**

* Client
* DLA

**Mathematical Description:**

**Let:**

* H be the hash function.
* H*SRS*σ be the signed serial number, issuer name, and timestamp hash.
* H*NS*σ be the node and timestamp hash.
* *ID* be the issuer string query.
* *Kpri* be the private asymmetric signing key.
* *Kpub* be the public asymmetric verification key.
* *ND* be the serialized node.
* *RI* be the remote device issuer name.
* *SD* be the device serial number.
* Sign be the asymmetric signing function.
* *st* be the sequence number and valid-time timestamp.
* σ be the signature.
* Verify be the asymmetric signature verification function.

The Client sends a *topological query request* to the DLA. The message contains the requestors certificate number, the remote Client’s issuer name, and a signed hash of the serial number, issuer name, and the valid-time timestamp.

H*SRS*σ = Sign*clientKpri*(H(*SD* || *RI*|| *st*))

Request(*ID*​) = (*SD*​ || *RI* ∥ H*SRS*σ)

The DLA responder validates the requestors signature, and the valid-time timestamp.

Verify*clientKpub*(H*SRS*σ) = H(*ID* || *SD*​ || *st*)

The DLA looks up the node in the topological database using the issuer string, hashes and signs the node, and sends it back to the requestor.

H*NS*σ = Sign*dlaKpri*(H(*ND* || *st*​))

Response(*ND*​) = (*ND*​ ∥ H*NS*σ)

**6.12 Topological Status Request**

**Overview:**

The DLA sends a status request to the target Client, verifying it is online and available. It sends a signed copy of its certificate serial number in the message.

The remote Client receives the signed serial number for the remote node, verifies the hash, signature, and the serial number.

If the responder is available, it sends its signed serial number back to the DLA requestor.

The DLA verifies the message, and the function signals if the node is available for connect.

**API:**

* *mpdc\_network\_topological\_query\_request()*
* *mpdc\_network\_topological\_query\_response()*

**Applies to:**

* Client
* DLA

**Mathematical Description:**

**Let:**

* H be the hash function.
* H*SS*σ be the signed serial number and timestamp hash.
* *ID* be the issuer string query.
* *Kpri* be the private asymmetric signing key.
* *Kpub* be the public asymmetric verification key.
* *ND* be the serialized node.
* Sign be the asymmetric signing function.
* *st* be the sequence number and valid-time timestamp.
* σ be the signature.
* Verify be the asymmetric signature verification function.

The DLA sends a *topological status request* to the device. The message contains the DLA’s certificate serial number, and a signed hash of the serial number and the valid-time timestamp.

H*SS*σ = Sign*dlaKpri*(H(*SD* || *st* ))

Request(*ID*​) = (*SD*​ ∥ H*SS*σ)

The responder validates the DLA’s signature, serial number, and the valid-time timestamp.

Verify*dlaKpub*(H*SS*σ) = H(*SD*​ || *st*)

The responder then echoes back it’s signed certificate serial number to the DLA if it is available.

H*SS*σ = Sign*dlaKpri*(H(*SD* || *st* ​))

Response(*SD*​) = (*SD*​ ∥ H*SS*σ)

**7.** **Security Analysis**

MPDC is designed to provide robust security against a wide range of attacks, including classical and quantum threats. The protocol incorporates multiple layers of security measures to ensure confidentiality, integrity, authentication, and forward secrecy.

**Defense Against Classical Attacks**

**7.1 Man-in-the-Middle (MITM) Attacks**

**Threat**: An attacker intercepts and possibly alters communication between the Client and MAS, attempting to impersonate one or both parties.

**Defense Strategies**:

**Certificate Validation**:

* Both the Client and MAS use certificates signed by the RDS.
* Each party validates the other's certificate against the trusted root certificate.
* Any unauthorized certificate will fail validation.
* Impersonating a MAS would require impersonation of the entire Agent network, requiring each device’s signing key be compromised.

**Digital Signatures**:

* Public keys are accompanied by digital signatures.
* Signatures are verified using the sender's public key.
* Altered public keys result in failed signature verification.

**Mutual Authentication**:

* Both parties authenticate each other using their respective key pairs and certificates.
* Prevents unauthorized entities from joining the communication.

**7.2 Replay Attacks**

**Threat**: An attacker reuses valid data transmissions to deceive a system into unauthorized actions.

**Defense Strategies**:

**Nonces and Timestamps**:

* Incorporate unique nonces and timestamps in messages.
* Ensures each message is fresh and cannot be replayed.
* Messages with old timestamps or used nonces are rejected.

**Session Identifiers**:

* Unique session IDs associated with each communication session.
* Prevents mixing of messages from different sessions.

**7.3 Key Compromise Attacks**

**Threat**: Compromise of a private key could allow an attacker to decrypt communications or impersonate a device.

**Defense Strategies**:

**Multi-Party Key Contribution**:

* Session key derivation involves key fragments from the Agent network.
* Compromising a single private key is insufficient without the Agent's fragment.

**Regular Key Refresh**:

* Session keys are refreshed periodically based on certificate expiration time.
* Limits the window of opportunity for attackers.

**Forward Secrecy**:

* Past session keys remain secure even if current private keys are compromised.
* Session keys are not derived solely from long-term private/public keys.

**7.4 Entropy Injection and Randomness**

**Threat**: Attacks exploiting weak or predictable keys due to insufficient randomness.

**Defense Strategies**:

**Agent's Key Fragment**:

* Provides high-quality entropy from an independent source.
* Enhances randomness in session key generation.

**Multiple Entropy Sources**:

* Combines entropy from Client, MAS, and Agent.
* Reduces the risk associated with any single point of failure.

**Defense Against Quantum Attacks**

Quantum computing poses a significant threat to classical cryptographic algorithms. MPDC addresses this by integrating quantum-resistant cryptographic primitives.

**7.5 Post-Quantum Cryptography**

**Quantum Threat**: Quantum algorithms like Shor's algorithm can break RSA, ECC, and other classical public-key systems.

**Defense Strategies**:

**Quantum-Resistant Algorithms**:

* Use lattice-based cryptography (e.g., Kyber, Dilithium) for public-key operations.
* Alternatively use code-based asymmetric cipher McEliece, and hash based signatures (SPHINCS+).
* Resistant to attacks from quantum computers, with a wide range of security options and parameter sets to accommodate different expectations of long-term security requirements.

**Hash Functions**:

* Employ SHAKE (SHA-3 variant) for hashing operations.
* Provides security against quantum attacks due to its collision and pre-image resistance even in a quantum context.

**7.6 Entropy Injection and Randomness**

**Quantum Threat**: Quantum computers could potentially simulate or predict key generation processes with insufficient entropy.

**Defense Strategies**:

**Agent's Key Fragment**:

* Injects additional entropy not predictable by quantum algorithms.
* Enhances the unpredictability of the session key.

**Multi-Party Contribution**:

* Session key depends on inputs from multiple parties.
* Increases computational difficulty for quantum adversaries.

**Mathematical Proofs of Security**

**7.7 Correctness of Key Exchange**

**Shared Session Key**:

Both MAS and Client compute:

**Where:**

* *fset* is the set of fragment keys
* *as* is the agent serial number
* *fc* is the encrypted fragment key

Fragment set shared by MAS and Client:

*fset* = { F1(*as1* || *fc1*), F2(*as2*|| *fc2*), ..., Fn(*asn* || *fcn*) }

Each fragment is decrypted.

For each *i* ∈ { 1, 2, …, *n* }, *fi* ​= -E(*cfi* )

Fragments are hashed to create a set of session keys used between the MAS and the Client to establish an encrypted tunnel. The fragment keys are added to the KDF input, including the fragment generated by the MAS for the Client:

*k1, k2, n1, n2*= KDF( *f1*, *f2*, ... *fn* )

Generates the session keys transmit and receive channels used to create an encrypted tunnel between the MAS and the Client.

Session =

**Verification**:

* Since all inputs are the same and verified, both parties derive the same session key.
* The distribution of keying material across multiple autonomous devices, ensures tamper-proof key derivation.
* That Client and MAS use different keys to decrypt each fragment, ensures the key fragments are not tampered with during transport.
* Entropy injected from multiple devices with different source random generators, vastly increases the mathematical hardness of differential analysis of the keying material.

**7.8 Resistance to Attacks**

**Collision Resistance**:

* Hash function is collision-resistant.
* Computationally infeasible to find different inputs that produce the same hash output.

**Computational Difficulty**:

* Without access to the private keys and the Agent's key fragment, attackers cannot compute the session key.
* Quantum algorithms do not efficiently solve lattice-based (Kyber, Dilithium), code-based (McEliece), or hash-based (SPHINCS+) cryptographic problems used in MPDC.

**7.9 Forward Secrecy**

**Session-Specific Keys**:

* Each session generates a new, unique session key.
* Session keys are not stored long-term.

**Ephemeral Key Fragments**:

* Agent's key fragments are unique per session and discarded after use.
* Compromise of long-term keys does not affect past session keys.

**Attack Mitigation Strategies**

**7.10 Certificate Revocation**

**Certificate Revocation**:

* DLA can broadcast a revocation message to all affected devices.
* Devices remove the certificate and topological node from the database.

**Comparison with Other Protocols**

**Strengths of MPDC**

**Multi-Party Key Exchange**:

* Involves multiple entities, enhancing security through distribution of security and authentication.
* Agent's entropy injection strengthens randomness.

**Post-Quantum Readiness**:

* Incorporates quantum-resistant algorithms.
* Future-proof against advancements in quantum computing.

**Flexibility and Scalability**:

* Adaptable to various network sizes and configurations.
* Suitable for IoT, enterprise, and critical infrastructure.

**Conclusion**

MPDC offers a robust cryptographic protocol that addresses both current and emerging security threats. Its design emphasizes secure communication through multi-party key exchange, leveraging contributions from the Client, MAS, and Agent to establish a secure session key. By integrating quantum-resistant cryptographic primitives and comprehensive attack mitigation strategies, MPDC ensures long-term security and resilience against sophisticated attacks.

The protocol's flexibility and scalability make it suitable for a wide range of applications, from IoT devices to enterprise networks. While it introduces additional complexity and reliance on multiple entities, the enhanced security benefits outweigh these challenges in environments where security is paramount.

**8.** **Application Scenarios**

A multi-party key exchange scheme that incorporates multiple dedicated sources of entropy enhances security by utilizing distributed randomness to establish a shared key. This model can be particularly advantageous in environments where strong and unpredictable entropy is crucial to prevent attacks that exploit weak randomness or deterministic behavior. Here are potential use cases and applications of such a system:

**8.1 Enhanced Client-Server Key Exchange for Critical Infrastructure**

**Description**: In scenarios involving critical infrastructure (e.g., power grids, water treatment facilities, military, and state applications), secure client-server communication is paramount. A multi-party key exchange augmented with multiple dedicated sources of entropy can involve various components of the infrastructure contributing entropy to the key generation process.

**Use Case**: During the key exchange, the client and server gather entropy from geographically separated sensors or entropy sources. This approach reduces the risk of entropy failures, increases randomness, and mitigates single-point vulnerabilities that could be exploited by attackers.

**Benefits**:

* Greater resilience against entropy-based attacks, including side-channel attacks.
* Mitigates the risk of predictable keys, which is crucial in long-term infrastructure deployments.
* Adds strong resistance against impersonation and man-in-the-middle attacks.
* Increased security against both classical and quantum adversaries by ensuring high-quality randomness.

**8.2 Secure Multi-user Messaging Applications**

**Description**: Multi-party key exchange with multiple entropy sources can be utilized in secure messaging applications where a shared group key needs to be established. Instead of relying solely on client-provided randomness, each participant (or dedicated entropy provider) contributes entropy to the key agreement.

**Use Case**: In a secure group chat application, users connect through a central server. The server coordinates a key exchange where each client contributes entropy, as well as an independent entropy provider (e.g., a trusted hardware random number generator or an entropy service).

**Benefits**:

* Guarantees high-quality randomness for the group key, reducing the risk of key compromise.
* Provides robustness against compromised clients or entropy providers, as no single entity can control the entire randomness pool.
* Improves forward secrecy and deniability, essential for secure messaging applications like Signal or WhatsApp.

**8.3 Post-Quantum Secure Remote Shell Protocol**

**Description**: In a remote shell protocol (similar to SSH but quantum-secure), using a multi-party key exchange with multiple entropy sources enhances the security of the session key generation process. Entropy can be injected from both the client device, server device, and additional entropy nodes or agents on the network.

**Use Case**: During the key exchange, the client, server, and a distributed entropy agent (e.g., a hardware security module or remote entropy service) each provide contributions. The combined entropy is used to derive session keys, ensuring they are resistant to prediction or manipulation.

**Benefits**:

* Stronger resistance against entropy manipulation or degradation attacks.
* Enhanced post-quantum security, as the key generation process integrates randomness from multiple independent sources.
* Suitable for highly sensitive environments, such as financial trading platforms or military communication systems.

**8.4 Secure Federated Learning and Distributed Data Analysis**

**Description**: In federated learning, multiple data providers (e.g., hospitals, financial institutions) collaborate to train a machine learning model without sharing raw data. A secure multi-party key exchange with diverse entropy sources can protect the communication channels used to aggregate local model updates.

**Use Case**: Each data provider injects its own entropy into the key exchange, ensuring that the shared model aggregation keys are random and unpredictable. A central coordinator aggregates these updates securely using the derived keys.

**Benefits**:

* Prevents data inference attacks that could arise from weak key generation.
* Enhances data confidentiality by ensuring that the shared keys have strong, unbiased randomness.
* Provides robustness against compromised participants or entropy failures in a decentralized network.

**8.5 Quantum-secure Blockchain Consensus Protocols**

**Description**: In blockchain and distributed ledger systems, consensus mechanisms (e.g., Proof of Stake, Byzantine Fault Tolerance) require secure communication channels for node-to-node messaging. A multi-party key exchange using multiple entropy sources can ensure secure key generation even in the presence of malicious nodes.

**Use Case**: Nodes participating in the consensus inject entropy into the key exchange, along with a separate entropy provider (e.g., a random beacon or oracle service). The resulting shared key secures node-to-node communication and ensures the integrity of the consensus process.

**Benefits**:

* Increases the unpredictability of the shared key, making it resistant to manipulation by malicious nodes.
* Supports post-quantum security, protecting the blockchain against future quantum attacks.
* Improves the robustness of consensus mechanisms, reducing the risk of double-spending or consensus failure.

**Advantages of Using Multiple Dedicated Sources of Entropy**

1. **Reduced Risk of Entropy Attacks**:

By distributing the entropy contribution among multiple independent sources, the risk of a single point of entropy failure (e.g., faulty hardware RNG, compromised software RNG) is minimized.

1. **Mitigation of Bias and Predictability**:

Each entropy source may have different characteristics and potential biases. Combining contributions from diverse sources helps mitigate any inherent biases and increases the overall quality of randomness.

1. **Resilience Against Compromise**:

If one of the entropy sources is compromised or controlled by an attacker, the randomness provided by the other sources can still ensure the unpredictability of the key, making attacks significantly harder.

1. **Quantum Resistance**:

A robust and diverse entropy pool enhances the security of the key exchange against quantum adversaries, who might otherwise exploit deterministic patterns in key generation.

1. **Flexibility and Scalability**:

The approach can be adapted to various network configurations, including client-server, peer-to-peer, and decentralized systems, making it a versatile solution for modern cryptographic applications.

In conclusion, multi-party key exchanges that leverage multiple sources of entropy provide enhanced security, reliability, and quantum resistance, making them an essential component of next-generation cryptographic systems. These schemes address the increasing demand for secure and scalable communication protocols in distributed and decentralized environments.

**10. Cryptanalysis of MPDC-I**

**10.1 Threat Model and Target Properties**

We analyze MPDC-I under an **active, adaptive adversary 𝔄** that

* controls all network links (eavesdrop, drop, modify, replay, reorder);
* compromises at will any subset of long-term keys or certificates held by **Client C**, **MAS S**, **Agent Aᵢ**, **DLA D**, or **RDS R**;
* performs chosen-ciphertext queries to the IND-CCA KEM (Kyber / McEliece) and chosen-message queries to the EUF-CMA signature (Dilithium / SPHINCS+);
* enjoys unlimited classical computing power and, after protocol termination, a large-scale quantum computer.

**Security goals:**

|  |  |
| --- | --- |
| Goal | Symbolic requirement |
| Entity Authentication | C and S accept *iff* every certificate chain verifies to **RDS** and all message signatures/MACs validate. |
| Session-key Secrecy | The two tunnel keys (kᴿˣ, kᵀˣ) are indistinguishable from random to 𝔄. |
| Forward Secrecy (FS) | Compromise of any long-term key after tunnel teardown reveals no past session keys. |
| Predictive-Resistance (PR) | Compromise of client or MAS state *before* the next fragment refresh reveals no future keys. |
| Replay & Downgrade Resistance | All MPDC packets embed a sequence number and UTC timestamp inside the signed/MAC’d data. |
| Robustness | Any authentication failure aborts the entire handshake, as mandated in § 5 “Design Requirements”. |

**10.2 Security Analysis of the Interior Key-Exchange**

**10.2.1 Fragment-Collection Sub-protocol**

1. **MAS authenticity** — C accepts a fragment bundle only if every included Agent fragment is MAC’d under an efk that C can reconstruct from its unique *mfk* with that Agent and the MAS-supplied token (§ 5.5). Forgery ⇒ break KMAC (UF-CMA) or derive *mfk* without running the 3-pass authenticated KEM.
2. **Entropy injection** — Session-key input = (MAS frag ∥ A₁ frag ∥ ⋯ ∥ Aₙ frag). Each fragment is 256-bit uniformly random (SHAKE output) and XOR-masked under an independent efk. Unless *all* contributing Agents are compromised, the min-entropy of the concatenation remains ≥ 256 bits.
3. **IND-CPA / INT-CTXT** — The data tunnel uses RCS+KMAC AEAD (optionally AES-GSM) with per-packet {seq, utc} as AAD. Confidentiality reduces to PRF-security of RCS and MAC-unforgeability of KMAC.

**10.2.2 Forward & Post-compromise Security**

* **FS**: All TKC entries and *mfk*-derived efk keys are wiped immediately after use; the surviving state on C and S contains only {kᴿˣ, kᵀˣ}. Compromise of signing keys or *mfk* after that point yields no information on past tunnels.
* **PR**: An attacker that snapshots C before the next fragment cycle cannot compute the upcoming efk because:

efknext​ = KDF(mfk\_Cai ​∥ HMAS​ ∥ Hai ​​∥ toknext​),

and **tokₙₑₓₜ** is generated by S *after* the snapshot. Breaking PR requires predicting a 256-bit nonce and defeating KDF-SHAKE.

**10.2.3 Replay, Reflection, Downgrade**

* Every network message binds seq‖utc inside the signature/MAC. Re- use fails because seq is strictly monotone per channel; utc must verify |Δ| ≤ τ (default 60 s).
* Protocol version and cryptographic *configuration-set* ID are hashed into every certificate and into the fragment-encryption KDF input, eliminating silent algorithm downgrade. MDPC Specification

**10.3 System-level Attack Surface**

|  |  |  |
| --- | --- | --- |
| Vector | Mitigation | Residual risk |
| Single-point CA failure (RDS) | Root certificate hard-expiry; proxy-signing via DLA; revocation broadcast (§ 6.10). | Short-lived (≤ 90 d) root epochs recommended. |
| Compromised Agent | Needs to corrupt *all* Agents contributing to a given handshake to bias the KDF; partial leakage only reduces entropy. | Diversify geography & HSM vendors of Agents. |
| Side-channel on mfk | mfk lives only in volatile RAM; implementations follow QSC constant-time style. | Harden HSM with DPA counter-measures. |
| Traffic analysis | Fixed-size packets; optional PAD frames under consideration (§ 5 “Design Requirements”). | Correlation on packet rate still possible. |

**10.4 Expanded Comparison with Representative Multi-Party Cryptography (MPC) Schemes**

The table below refines § 10.4 by contrasting **MPDC-I** with three well-known families of multi-party protocols, weighing them along five axes that matter in real deployments:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dimension | MPDC-I | MLS / TreeKEM<br>(RFC 9380, 2024) | Threshold-ECDH / TSS<br>(GG-18 / GG-20) | SPDZ-2k<br>(actively secure MPC) |
| Primary goal | Post-quantum client–server tunnel with entropy splitting across *n* Agents | Large-scale group messaging w. FS & PCS | Distributed signing / decryption without revealing key | General secure computation over arithmetic circuits |
| Cryptographic core | Kyber + McEliece KEM; Dilithium / SPHINCS+ sig.; RCS+KMAC AEAD | TreeKEM (X25519); Ed25519 sig.; HPKE-ChaCha20-Poly1305 | Elliptic-curve DKG; interactive zero-knowledge; Paillier | Packed secret-sharing; homomorphic MACs; OT + GMW |
| Post-quantum | ✓ (native) | ✗ (classical) | △ (research PQ-TSS variants) | △ (if using lattice OT) |
| Forward secrecy (FS) | ✓ fresh KEM keys each session | ✓ asym ratchet per epoch | ✓ (fresh nonce in DKG) | N/A (offline pre-processing) |
| Post-compromise security (PCS) | ✓ mandatory symmetric + optional asym ratchets | ✓ (Leaf & Group TreeKEM update) | ✗ (no built-in ratchet) | N/A |
| MitM surface | All handshake msgs signed by MAS & validated to RDS; Agents MAC their fragments → MitM must break EUF-CMA or IND-CCA | TreeKEM signatures on Update/Commit; HPKE authenticated channel; PKI root CA | Interactive proofs authenticated over mutually trusted channel → MitM blocked if one honest party | Requires authenticated OT channel; MitM breaks correctness if OT not authenticated |
| Handshake round-trips | 3 RTT (C↔D, C↔S, S↔Agents) | 2 RTT (init) + 1 RTT per epoch | 2–5 RTT depending on TSS variant | Dozens of OT & MAC rounds |
| Scalability | Linear in #Agents (n ≤ 8 typical) | Log₂ M members (balanced tree) | All-to-all or dealer → O(n²) | Quadratic comm. in party count |
| Online perf. at 256-bit sec. | ≈ 1.8 ms KEM + 2 KB traffic (n = 4) | ≈ 0.9 ms DH + 1 KB (M = 32) | 25–40 ms per party, 30 KB | > 100 ms, > 1 MB per gate |
| Replay / downgrade defense | seq‖utc in every signed/MAC’d field; cfg hash bound into KDF | Epoch + transcript hash | Depends on application layer | Depends on OT-auth |
| Indispensable trust | Root-signed RDS + majority of Agents honest | One honest member per epoch | One honest key-share holder | Honest majority (t < n/3) |
| Typical deployment | Fintech VPN, fleet mgmt., SCADA | Encrypted chat (Signal, Matrix) | Crypto-wallet, HSM quorum | Privacy-preserving analytics |

**Key Take-aways**

**Security strength**

* **Post-quantum assurance** – MPDC-I adopts **lattice + code-based KEMs** and **hash-based / lattice signatures**, whereas MLS and today’s production TSS still rely on classical elliptic curves.
* **MitM robustness** – MPDC-I signs *every* control packet (including Agent fragments) and embeds a **configuration-ID hash** in its KDF, removing downgrade vectors. MLS signs only *epoch commits*; intermediate handshake traffic is HPKE-authenticated but not globally transcript-bound, leaving room for exotic prefix attacks if the application layer forgets to enforce the transcript hash.
* **Resilience to single-party compromise** – Thanks to entropy splitting, MPDC-I preserves ≥ 256-bit min-entropy so long as *one* Agent remains honest; TreeKEM collapses to 128-bit if any member’s leaf secret leaks; most 2-of-3 TSS deployments lose the *entire* signing key if two shares collude.

**Performance**

* MPDC-I’s online cost grows **linearly** with the number of contributing Agents; in realistic setups (Agent quorum ≤ 8) it stays below 3 ms on commodity x86, only double MLS while delivering PQ security.
* Threshold-ECDH and SPDZ incur interactive, all-to-all rounds; latency dominates in WAN deployments and renders them unsuitable for “open tunnel in <10 ms targets.
* Offline, MPDC-I lets Agents pre-compute fragment caches, cutting MAS handshake CPU by ≈ 80 %.

**Man-in-the-Middle (MitM) exposure**

|  |  |  |
| --- | --- | --- |
| Scheme | Earliest point MitM can inject without being detected | Reason |
| MPDC-I | After MAS sends *FINISH* packet – but MAC/Sig check fails immediately | Every packet signed/MACed, seq# monotone |
| MLS TreeKEM | During **Welcome** if DS identity not pinned | DS leaf not signed by external CA |
| GG-20 TSS | After key-generation until signature aggregation – MitM can force abort but not forge | Messages authenticated per share |
| SPDZ-2k | Any OT channel if lacking TLS – will corrupt output silently | OT not integrity-protected by design |

**Summary**

MPDC-I strikes a **middle ground**:

* It offers *quantum-resistant, tunnel-oriented security* stronger than today’s MLS or TSS yet avoids the heavy-weight arithmetic of generic MPC.
* Its MitM surface is narrower than MLS because of full-transcript signatures and sequence-number binding.
* Performance stays practical (< 3 ms, < 3 KB) for the intended small-to-medium operator pools, whereas full MPC protocols remain orders of magnitude slower.

For infrastructures that already trust a root CA and can deploy 3–8 hardened Agents, MPDC-I delivers a uniquely strong, efficiently deployable alternative to classical group key-exchange or threshold-ECDH solutions.

**10.5 Recommendations**

1. **Root-key rotation & CRLite-style revocation** to cap RDS compromise impact.
2. **Length-hiding padding option** (PAD flag) to mitigate traffic-shape leakage.
3. **Public audit of RCS** wide-block cipher; provide AES-GCM fallback for FIPS zones.
4. **Automated Agent health checking**—MAS aborts handshake if fewer than *t* fragments arrive, where *t* is policy-configurable quorum.

**10.6 Conclusion**

Under standard assumptions (IND-CCA KEM, EUF-CMA signatures, PRF-secure SHAKE/KMAC, and honest-majority Agents), MPDC-I achieves entity authentication, strong session-key secrecy, forward & predictive security, and robustness against replay and downgrade attacks, even against quantum-equipped adversaries. Its entropy-splitting design offers a measurably higher security margin than single-source tunnels and positions MPDC-I as a practical, post-quantum-ready alternative to MLS or classic SSH/TLS in environments where centralized trust and scalable, lightweight operations are paramount.

**10.** **Internal Functions**

**10.1 MPDC Certificate API Documentation**

**10.1.1 Function:** mpdc\_certificate\_algorithm\_decode

**Purpose:** Decodes a protocol-set string into its enumerated form for internal use.

**Parameters:**

* name (Type: const char\*): A string representing the protocol-set.

**Returns:** mpdc\_configuration\_sets - The protocol-set enumerator corresponding to the provided string.

**10.1.2 Function:** mpdc\_certificate\_algorithm\_enabled

**Purpose:** Tests if a specific protocol-set is enabled on this system.

**Parameters:**

* conf (Type: mpdc\_configuration\_sets): The protocol-set enumerator.

**Returns:** bool - Returns true if the protocol-set is enabled.

**10.1.3 Function:** mpdc\_certificate\_algorithm\_encode

**Purpose:** Encodes the protocol-set enumerator to a string format.

**Parameters:**

* name (Type: char\*): The output protocol-set string.
* conf (Type: mpdc\_configuration\_sets): The protocol-set enumerator.

**Returns:** void

**10.1.4 Function:** mpdc\_certificate\_child\_are\_equal

**Purpose:** Compares two child certificates for equivalence.

**Parameters:**

* a (Type: const mpdc\_child\_certificate\*): The first certificate.
* b (Type: const mpdc\_child\_certificate\*): The second certificate.

**Returns:** bool - Returns true if the two certificates are equal.

**10.1.5 Function:** mpdc\_certificate\_child\_copy

**Purpose:** Copies data from one child certificate to another.

**Parameters:**

* output (Type: mpdc\_child\_certificate\*): The destination certificate for copied data.
* input (Type: const mpdc\_child\_certificate\*): The source certificate to copy.

**Returns:** void

**10.1.6 Function:** mpdc\_certificate\_child\_create

**Purpose:** Initializes a new child certificate with provided parameters.

**Parameters:**

* child (Type: mpdc\_child\_certificate\*): A pointer to the empty child certificate.
* pubkey (Type: const uint8\_t\*): A pointer to the public signature key (size: QSMP\_VERIFYKEY\_SIZE).
* expiration (Type: const mpdc\_certificate\_expiration\*): The certificate expiration time structure.
* address (Type: const char\*): The certificate IP address string.
* issuer (Type: const char\*): The certificate issuer string.
* designation (Type: mpdc\_network\_designations): The certificate designation type.

**Returns:** void

**10.1.7 Function:** mpdc\_certificate\_child\_decode

**Purpose:** Decodes a child certificate string into a certificate structure.

**Parameters:**

* child (Type: mpdc\_child\_certificate\*): A pointer to the child certificate to populate.
* enck (Type: const char[MPDC\_CHILD\_CERTIFICATE\_STRING\_SIZE]): The encoded key array.

**Returns:** bool - Returns true if the key decoded successfully.

**10.1.8 Function:** mpdc\_certificate\_child\_deserialize

**Purpose:** Deserializes a child certificate from a serialized input array into a structure.

**Parameters:**

* child (Type: mpdc\_child\_certificate\*): A pointer to the child certificate.
* input (Type: const uint8\_t\*): A pointer to the serialized certificate data.

**Returns:** void

**10.1.9 Function:** mpdc\_certificate\_child\_encode

**Purpose:** Encodes a child certificate into a readable string format.

**Parameters:**

* enck (Type: char[MPDC\_CHILD\_CERTIFICATE\_STRING\_SIZE]): The buffer to store the encoded certificate.
* child (Type: const mpdc\_child\_certificate\*): The certificate to encode.

**Returns:** size\_t - The size of the encoded certificate string.

**10.1.10 Function:** mpdc\_certificate\_child\_erase

**Purpose:** Deletes the data of a child certificate.

**Parameters:**

* child (Type: mpdc\_child\_certificate\*): A pointer to the child certificate to erase.

**Returns:** void

**10.1.11 Function:** mpdc\_certificate\_child\_file\_to\_struct

**Purpose:** Loads a child certificate from a file into a structure.

**Parameters:**

* fpath (Type: const char\*): The file path to the serialized certificate.
* child (Type: mpdc\_child\_certificate\*): A pointer to the child certificate structure to populate.

**Returns:** bool - Returns true on successful loading.

**10.1.12 Function:** mpdc\_certificate\_child\_hash

**Purpose:** Generates a hash of a child certificate.

**Parameters:**

* output (Type: uint8\_t\*): The output buffer for the hash (size: MPDC\_CERTIFICATE\_HASH\_SIZE).
* child (Type: const mpdc\_child\_certificate\*): A pointer to the child certificate to hash.

**Returns:** void

**10.1.13 Function:** mpdc\_certificate\_child\_is\_valid

**Purpose:** Checks if a child certificate has a valid format.

**Parameters:**

* child (Type: const mpdc\_child\_certificate\*): A pointer to the child certificate to validate.

**Returns:** bool - Returns true if the certificate format is valid.

**10.1.14 Function:** mpdc\_certificate\_child\_message\_verify

**Purpose:** Verifies a message signature using the child certificate.

**Parameters:**

* message (Type: uint8\_t\*): The buffer to store the verified message output.
* msglen (Type: size\_t\*): The length of the verified message.
* signature (Type: const uint8\_t\*): The signed message.
* siglen (Type: size\_t): The length of the signed message.
* child (Type: const mpdc\_child\_certificate\*): A pointer to the child certificate used for verification.

**Returns:** bool - Returns true if the message signature is verified.

**10.1.15 Function:** mpdc\_certificate\_child\_serialize

**Purpose:** Serializes a child certificate into a byte array.

**Parameters:**

* output (Type: uint8\_t\*): The array to receive the serialized certificate (size: MPDC\_CERTIFICATE\_CHILD\_SIZE).
* child (Type: const mpdc\_child\_certificate\*): The child certificate to serialize.

**Returns:** void

**10.1.16 Function:** mpdc\_certificate\_child\_struct\_to\_file

**Purpose:** Saves a child certificate structure to a file.

**Parameters:**

* fpath (Type: const char\*): The file path where the certificate will be saved.
* child (Type: const mpdc\_child\_certificate\*): A pointer to the child certificate structure to save.

**Returns:** bool - Returns true on successful saving.

**10.1.17 Function:** mpdc\_certificate\_designation\_decode

**Purpose:** Decodes a network designation string into its enumerated form.

**Parameters:**

* sdsg (Type: const char\*): The string representing the network designation.

**Returns:** mpdc\_network\_designations - The enumerated network designation.

**10.1.18 Function:** mpdc\_certificate\_designation\_encode

**Purpose:** Encodes a network designation enumerator to string format.

**Parameters:**

* sdsg (Type: char\*): The buffer to store the encoded network designation string.
* designation (Type: mpdc\_network\_designations): The network designation enumerator to encode.

**Returns:** size\_t - The size of the encoded string.

**10.1.19 Function:** mpdc\_certificate\_expiration\_set\_days

**Purpose:** Sets expiration days for a certificate.

**Parameters:**

* expiration (Type: mpdc\_certificate\_expiration\*): Pointer to the expiration structure to configure.
* start (Type: uint16\_t): Number of days before the certificate becomes valid.
* duration (Type: uint16\_t): Duration in days for the certificate validity.

**Returns:** void

**10.1.20 Function:** mpdc\_certificate\_expiration\_set\_seconds

**Purpose:** Sets expiration time in seconds for a certificate.

**Parameters:**

* expiration (Type: mpdc\_certificate\_expiration\*): Pointer to the expiration structure to configure.
* start (Type: uint64\_t): The starting second when the certificate is valid.
* period (Type: uint64\_t): Duration in seconds for the certificate validity.

**Returns:** void

**10.1.21 Function:** mpdc\_certificate\_expiration\_time\_verify

**Purpose:** Verifies if a certificate’s expiration time is valid against the current time.

**Parameters:**

* expiration (Type: const mpdc\_certificate\_expiration\*): A pointer to the expiration structure of the certificate.

**Returns:** bool - Returns true if the certificate has not expired.

**10.1.22 Function:** mpdc\_certificate\_message\_hash\_sign

**Purpose:** Hashes a message and generates a signature for the hash.

**Parameters:**

* signature (Type: uint8\_t\*): Buffer for storing the generated signature.
* sigkey (Type: const uint8\_t\*): The private signing key used for signing.
* message (Type: const uint8\_t\*): The message to sign.
* msglen (Type: size\_t): Length of the message.

**Returns:** size\_t - The size of the generated signature.

**10.1.23 Function:** mpdc\_certificate\_root\_compare

**Purpose:** Compares two root certificates for equivalence.

**Parameters:**

* a (Type: const mpdc\_root\_certificate\*): The first root certificate.
* b (Type: const mpdc\_root\_certificate\*): The second root certificate.

**Returns:** bool - Returns true if the two root certificates are equal.

**10.1.24 Function:** mpdc\_certificate\_root\_create

**Purpose:** Creates a new root certificate with specified parameters.

**Parameters:**

* root (Type: mpdc\_root\_certificate\*): Pointer to the root certificate structure.
* pubkey (Type: const uint8\_t\*): Public key for the certificate.
* expiration (Type: const mpdc\_certificate\_expiration\*): Certificate expiration time structure.
* issuer (Type: const char\*): Issuer name string.

**Returns:** void

**10.1.25 Function:** mpdc\_certificate\_root\_decode

**Purpose:** Decodes a root certificate from an encoded string.

**Parameters:**

* root (Type: mpdc\_root\_certificate\*): Pointer to the root certificate structure to populate.
* enck (Type: const char\*): Encoded string representing the certificate.

**Returns:** bool - Returns true if decoding is successful.

**10.1.26 Function:** mpdc\_certificate\_root\_deserialize

**Purpose:** Deserializes a root certificate from a byte array.

**Parameters:**

* root (Type: mpdc\_root\_certificate\*): Pointer to the root certificate to populate.
* input (Type: const uint8\_t\*): Input array containing the serialized certificate data.

**Returns:** void

**10.1.27 Function:** mpdc\_certificate\_root\_encode

**Purpose:** Encodes a root certificate into a readable string format.

**Parameters:**

* enck (Type: char\*): Buffer to store the encoded certificate.
* root (Type: const mpdc\_root\_certificate\*): Root certificate to encode.

**Returns:** size\_t - The size of the encoded certificate string.

**10.1.28 Function:** mpdc\_certificate\_root\_erase

**Purpose:** Deletes data from a root certificate.

**Parameters:**

* root (Type: mpdc\_root\_certificate\*): Pointer to the root certificate to erase.

**Returns:** void

**10.1.30 Function:** mpdc\_certificate\_root\_file\_to\_struct

**Purpose:** Loads a root certificate from a file into a structure.

**Parameters:**

* fpath (Type: const char\*): Path to the file containing the serialized certificate.
* root (Type: mpdc\_root\_certificate\*): Pointer to the root certificate structure to populate.

**Returns:** bool - Returns true on successful loading.

**10.1.31 Function:** mpdc\_certificate\_root\_hash

**Purpose:** Generates a hash of a root certificate.

**Parameters:**

* output (Type: uint8\_t\*): Buffer to store the hash (size: MPDC\_CERTIFICATE\_HASH\_SIZE).
* root (Type: const mpdc\_root\_certificate\*): Pointer to the root certificate to hash.

**Returns:** void

**10.1.32 Function:** mpdc\_certificate\_root\_is\_valid

**Purpose:** Validates the format and structure of a root certificate.

**Parameters:**

* root (Type: const mpdc\_root\_certificate\*): Pointer to the root certificate to validate.

**Returns:** bool - Returns true if the root certificate is valid.

**10.1.33 Function:** mpdc\_certificate\_root\_serialize

**Purpose:** Serializes a root certificate into a byte array.

**Parameters:**

* output (Type: uint8\_t\*): Array to receive the serialized certificate (size: MPDC\_CERTIFICATE\_ROOT\_SIZE).
* root (Type: const mpdc\_root\_certificate\*): Pointer to the root certificate to serialize.

**Returns:** void

**10.1.34 Function:** mpdc\_certificate\_root\_sign

**Purpose:** Signs a child certificate with the root certificate’s signing key.

**Parameters:**

* child (Type: mpdc\_child\_certificate\*): Pointer to the child certificate to sign.
* root (Type: const mpdc\_root\_certificate\*): Pointer to the root certificate used for signing.
* rsigkey (Type: const uint8\_t\*): Pointer to the root signing key (QSMP\_SIGKEY\_ENCODED\_SIZE).

**Returns:** size\_t - The size of the signed certificate.

**10.1.35 Function:** mpdc\_certificate\_root\_signature\_verify

**Purpose:** Verifies a child certificate’s signature using the root certificate.

**Parameters:**

* child (Type: const mpdc\_child\_certificate\*): Pointer to the child certificate being verified.
* root (Type: const mpdc\_root\_certificate\*): Pointer to the root certificate for verification.

**Returns:** bool - Returns true if the signature is verified successfully.

**10.1.37 Function:** mpdc\_certificate\_root\_struct\_to\_file

**Purpose:** Saves a root certificate structure to a file.

**Parameters:**

* fpath (Type: const char\*): Path to the file where the certificate will be saved.
* root (Type: const mpdc\_root\_certificate\*): Pointer to the root certificate structure to save.

**Returns:** bool - Returns true on successful saving.

**10.1.38 Function:** mpdc\_certificate\_signature\_generate\_keypair

**Purpose:** Generates an asymmetric key-pair for signing and verification.

**Parameters:**

* keypair (Type: mpdc\_signature\_keypair\*): Pointer to a container that will hold the generated key-pair.

**Returns:** void

**10.1.39 Function:** mpdc\_certificate\_signature\_hash\_verify

**Purpose:** Verifies a signature over a hashed message using the child certificate.

**Parameters:**

* signature (Type: const uint8\_t\*): Pointer to the signed hash.
* siglen (Type: size\_t): Length of the signed hash.
* message (Type: const uint8\_t\*): Pointer to the message hash.
* msglen (Type: size\_t): Length of the message hash.
* lcert (Type: const mpdc\_child\_certificate\*): Pointer to the certificate used for verification.

**Returns:** bool - Returns true if the signature is verified successfully.

**10.1.40 Function:** mpdc\_certificate\_signature\_sign\_message

**Purpose:** Signs a message using an asymmetric private key.

**Parameters:**

* signature (Type: uint8\_t\*): Array to store the generated signature (MPDC\_ASYMMETRIC\_SIGNATURE\_SIZE).
* message (Type: const uint8\_t\*): The message to be signed.
* msglen (Type: size\_t): Length of the message.
* prikey (Type: const uint8\_t\*): Private key used for signing.

**Returns:** size\_t - The length of the generated signature.

**10.1.41 Function:** mpdc\_certificate\_signature\_verify\_message

**Purpose:** Verifies a signed message using an asymmetric public key.

**Parameters:**

* message (Type: const uint8\_t\*): The message to verify.
* msglen (Type: size\_t): Length of the message.
* signature (Type: const uint8\_t\*): The signature to verify.
* siglen (Type: size\_t): Length of the signature.
* pubkey (Type: const uint8\_t\*): Public key used for verification.

**Returns:** bool - Returns true if the message is verified successfully.

**10.2 Crypto.h**

**10.2.1 Function:** mpdc\_crypto\_decrypt\_stream

**Purpose:** Decrypts a stream of bytes.

**Parameters:**

* output (Type: uint8\_t\*): Array receiving the decrypted plain text.
* seed (Type: const uint8\_t\*): Secret seed array (MPDC\_CRYPTO\_SEED\_SIZE).
* input (Type: const uint8\_t\*): The encrypted input array.
* length (Type: size\_t): Number of bytes to decrypt.

**Returns:** bool - Returns true on success.

**10.2.2 Function:** mpdc\_crypto\_encrypt\_stream

**Purpose:** Encrypts a stream of bytes.

**Parameters:**

* output (Type: uint8\_t\*): Array receiving the encrypted cipher text.
* seed (Type: const uint8\_t\*): Secret seed array (MPDC\_CRYPTO\_SEED\_SIZE).
* input (Type: const uint8\_t\*): Plain text input array.
* length (Type: size\_t): Number of bytes to encrypt.

**Returns:** void

**10.2.3 Function:** mpdc\_crypto\_generate\_application\_keychain

**Purpose:** Generates a secure key chain for application use.

**Parameters:**

* seed (Type: uint8\_t\*): Output array for the secret seed.
* seedlen (Type: size\_t): Length of the seed array.
* password (Type: const char\*): Password array.
* passlen (Type: size\_t): Byte length of the password array.
* username (Type: const char\*): Computer's user name.
* userlen (Type: size\_t): Byte length of the user name array.

**Returns:** void

**10.2.4 Function:** mpdc\_crypto\_generate\_application\_salt

**Purpose:** Generates a unique application salt using OS sources.

**Parameters:**

* output (Type: uint8\_t\*): Array for the secret salt.
* outlen (Type: size\_t): Length of the salt array.

**Returns:** void

**10.2.5 Function:** mpdc\_crypto\_generate\_hash\_code

**Purpose:** Hashes a message and writes it to an output array.

**Parameters:**

* output (Type: char\*): Output array to receive the hash.
* message (Type: const char\*): Pointer to the message array.
* msglen (Type: size\_t): Length of the message array.

**Returns:** void

**10.2.6 Function:** mpdc\_crypto\_generate\_mac\_code

**Purpose:** Generates a message authentication code (MAC) for a message and writes it to an output array.

**Parameters:**

* output (Type: char\*): Output array to receive the MAC.
* outlen (Type: size\_t): Byte length of the output array.
* message (Type: const char\*): Pointer to the message array.
* msglen (Type: size\_t): Length of the message array.
* key (Type: const char\*): Pointer to the key array.
* keylen (Type: size\_t): Length of the key array.

**Returns:** void

**10.2.7 Function:** mpdc\_crypto\_hash\_password

**Purpose:** Hashes a password and user name and writes it to an output array.

**Parameters:**

* output (Type: char\*): Output array to receive the hash.
* outlen (Type: size\_t): Byte length of the output array.
* username (Type: const char\*): Computer's user name.
* userlen (Type: size\_t): Byte length of the user name array.
* password (Type: const char\*): Password array.
* passlen (Type: size\_t): Byte length of the password array.

**Returns:** void

**10.2.8 Function:** mpdc\_crypto\_password\_minimum\_check

**Purpose:** Checks if a password meets a minimum security threshold.

**Parameters:**

* password (Type: const char\*): Password array.
* passlen (Type: size\_t): Byte length of the password array.

**Returns:** bool - Returns true if the password meets minimum requirements.

**10.2.9 Function:** mpdc\_crypto\_password\_verify

**Purpose:** Hashes a password and user name and compares it to a stored hash value.

**Parameters:**

* username (Type: const char\*): Computer's user name.
* userlen (Type: size\_t): Byte length of the user name array.
* password (Type: const char\*): Password array.
* passlen (Type: size\_t): Byte length of the password array.
* hash (Type: const char\*): Hash array for comparison.
* hashlen (Type: size\_t): Byte length of the hash array.

**Returns:** bool - Returns true if the password and user name hash matches the stored value.

**10.2.10 Function:** mpdc\_crypto\_secure\_memory\_allocate

**Purpose:** Allocates a block of secure memory.

**Parameters:**

* length (Type: size\_t): Byte length of the memory block to allocate.

**Returns:** uint8\_t\* - Pointer to the allocated memory or NULL if allocation fails.

**10.2.11 Function:** mpdc\_crypto\_secure\_memory\_deallocate

**Purpose:** Releases an allocated block of secure memory.

**Parameters:**

* block (Type: uint8\_t\*): Pointer to the memory block to deallocate.
* length (Type: size\_t): Byte length of the allocated memory block.

**Returns:** void

**10.3 API Documentation MPDC.h**

**10.3.1 Constants**

**MPDC\_NETWORK\_CLIENT\_CONNECT**  
Enables client-to-client encrypted tunnels.

**MPDC\_NETWORK\_MFK\_HASH\_CYCLED**  
Enables MFK key cycling (default).

**MPDC\_NETWORK\_PROTOCOL\_IPV6**  
Indicates that MPDC is using the IPv6 networking stack.

**MPDC\_EXTENDED\_SESSION\_SECURITY**  
Enables 512-bit security on session tunnels.

**MPDC\_ASYMMETRIC\_CIPHERTEXT\_SIZE**  
Defines the byte size of the asymmetric cipher-text array.  
**Value**: QSC\_KYBER\_CIPHERTEXT\_SIZE (variable, depending on the selected cipher).

**MPDC\_ASYMMETRIC\_PRIVATE\_KEY\_SIZE**  
Defines the byte size of the asymmetric cipher private key array.  
**Value**: QSC\_KYBER\_PRIVATEKEY\_SIZE (variable, depending on the selected cipher).

**MPDC\_ASYMMETRIC\_PUBLIC\_KEY\_SIZE**  
Defines the byte size of the asymmetric cipher public key array.  
**Value**: QSC\_KYBER\_PUBLICKEY\_SIZE (variable, depending on the selected cipher).

**MPDC\_ASYMMETRIC\_SIGNATURE\_SIZE**  
Defines the byte size of the asymmetric signature array.  
**Value**: QSC\_DILITHIUM\_SIGNATURE\_SIZE (variable, depending on the selected signature algorithm).

**MPDC\_ASYMMETRIC\_SIGNING\_KEY\_SIZE**  
Defines the byte size of the asymmetric signing key array.  
**Value**: QSC\_DILITHIUM\_PRIVATEKEY\_SIZE (variable, depending on the selected signature algorithm).

**MPDC\_ASYMMETRIC\_VERIFICATION\_KEY\_SIZE**  
Defines the byte size of the asymmetric verification key array.  
**Value**: QSC\_DILITHIUM\_PUBLICKEY\_SIZE (variable, depending on the selected signature algorithm).

**MPDC\_ACTIVE\_VERSION**  
Defines the active version of MPDC.  
**Value**: 1

**MPDC\_ACTIVE\_VERSION\_SIZE**  
Defines the size of the MPDC active version.  
**Value**: 2

**MPDC\_APPLICATION\_AGENT\_PORT**  
Defines the default port number for the Agent.  
**Value**: 37766

**MPDC\_AGENT\_FULL\_TRUST**  
Defines the full trust designation number.  
**Value**: 1000001

**MPDC\_AGENT\_MINIMUM\_TRUST**  
Defines the minimum trust designation number.  
**Value**: 1

**MPDC\_AGENT\_NAME\_MAX\_SIZE**  
Defines the maximum agent name string length in characters. The last character must be a string terminator.  
**Value**: 256

**MPDC\_AGENT\_TWOWAY\_TRUST**  
Defines the two-way trust designation number.  
**Value**: 1000002

**MPDC\_APPLICATION\_CLIENT\_PORT**  
Defines the default port number for the MPDC Client.  
**Value**: 37761

**MPDC\_APPLICATION\_DLA\_PORT**  
Defines the default port number for the DLA.  
**Value**: 37762

**MPDC\_APPLICATION\_IDG\_PORT**  
Defines the default port number for the MPDC IDG.  
**Value**: 37763

**MPDC\_APPLICATION\_RDS\_PORT**  
Defines the default port number for the RDS.  
**Value**: 37764

**MPDC\_APPLICATION\_MAS\_PORT**  
Defines the default port number for the MAS.  
**Value**: 37765

**MPDC\_CANONICAL\_NAME\_MINIMUM\_SIZE**  
Defines the minimum size for a canonical name.  
**Value**: 3

**MPDC\_CERTIFICATE\_ADDRESS\_SIZE**  
Defines the maximum IP address length.  
**Value**: 22

**MPDC\_CERTIFICATE\_ALGORITHM\_SIZE**  
Defines the algorithm type size.  
**Value**: 1

**MPDC\_CERTIFICATE\_DEFAULT\_PERIOD**  
Defines the default certificate validity period in seconds.  
**Value**: 365 \* 24 \* 60 \* 60 \* 1000 (1 year)

**MPDC\_CERTIFICATE\_DESIGNATION\_SIZE**  
Defines the size of the child certificate designation field.  
**Value**: 1

**MPDC\_CERTIFICATE\_EXPIRATION\_SIZE**  
Defines the certificate expiration date length.  
**Value**: 16

**MPDC\_CERTIFICATE\_HASH\_SIZE**  
Defines the size of the certificate hash in bytes.  
**Value**: 32

**MPDC\_CERTIFICATE\_ISSUER\_SIZE**  
Defines the maximum certificate issuer string length. The last character must be a string terminator.  
**Value**: 256

**MPDC\_CERTIFICATE\_LINE\_LENGTH**  
Defines the line length of the printed MPDC certificate.  
**Value**: 64

**MPDC\_CERTIFICATE\_MAXIMUM\_PERIOD**  
Defines the maximum certificate validity period in seconds.  
**Value**: MPDC\_CERTIFICATE\_DEFAULT\_PERIOD \* 2

**MPDC\_CERTIFICATE\_MINIMUM\_PERIOD**  
Defines the minimum certificate validity period in seconds.  
**Value**: 24 \* 60 \* 60 \* 1000 (1 day)

**MPDC\_CERTIFICATE\_SERIAL\_SIZE**  
Defines the certificate serial number field length.  
**Value**: 16

**MPDC\_CERTIFICATE\_HINT\_SIZE**  
Defines the size of the topological hint.  
**Value**: MPDC\_CERTIFICATE\_HASH\_SIZE + MPDC\_CERTIFICATE\_SERIAL\_SIZE

**MPDC\_CERTIFICATE\_SIGNED\_HASH\_SIZE**  
Defines the size of the signature and hash field in a certificate.  
**Value**: MPDC\_ASYMMETRIC\_SIGNATURE\_SIZE + MPDC\_CERTIFICATE\_HASH\_SIZE

**MPDC\_CERTIFICATE\_VERSION\_SIZE**  
Defines the version ID size.  
**Value**: 1

**MPDC\_CERTIFICATE\_CHILD\_SIZE**  
Defines the length of a child certificate.  
**Value**: Calculated based on various field sizes.

**MPDC\_CERTIFICATE\_IDG\_SIZE**  
Defines the length of an IDG certificate.  
**Value**: Calculated based on various field sizes.

**MPDC\_CERTIFICATE\_ROOT\_SIZE**  
Defines the length of a root certificate.  
**Value**: Calculated based on various field sizes.

**MPDC\_CRYPTO\_SYMMETRIC\_KEY\_SIZE**  
Defines the byte length of the symmetric cipher key.  
**Value**: 32

**MPDC\_CRYPTO\_SYMMETRIC\_NONCE\_SIZE**  
Defines the byte length of the symmetric cipher nonce.  
**Value**: 32

**MPDC\_CRYPTO\_SEED\_SIZE**  
Defines the seed array byte size.  
**Value**: 64

**MPDC\_CRYPTO\_SYMMETRIC\_TOKEN\_SIZE**  
Defines the byte length of the token.  
**Value**: 32

**MPDC\_CRYPTO\_SYMMETRIC\_HASH\_SIZE**  
Defines the hash function output byte size.  
**Value**: 32

**MPDC\_CRYPTO\_SYMMETRIC\_MAC\_SIZE**  
Defines the MAC function output byte size.  
**Value**: 32 or 64 if MPDC\_EXTENDED\_SESSION\_SECURITY is enabled.

**MPDC\_CRYPTO\_SYMMETRIC\_SECRET\_SIZE**  
Defines the shared secret byte size.  
**Value**: 32

**MPDC\_CRYPTO\_SYMMETRIC\_SESSION\_KEY\_SIZE**  
Defines the session key security size.  
**Value**: 32 or 64 if MPDC\_EXTENDED\_SESSION\_SECURITY is enabled.

**MPDC\_DLA\_CONVERGENCE\_INTERVAL**  
Defines the interval between agent convergence checks in seconds.  
**Value**: 86400 (24 hours)

**MPDC\_DLA\_IP\_MAX**  
Defines the maximum IP address length.  
**Value**: 65

**MPDC\_DLA\_PENALTY\_MAX**  
Defines the maximum unreachable penalty before the DLA is deemed unreliable.  
**Value**: 256

**MPDC\_DLA\_REDUCTION\_INTERVAL**  
Defines the time in milliseconds before a penalty is reduced for a flapping DLA.  
**Value**: 1000000

**MPDC\_DLA\_UPDATE\_WAIT\_TIME**  
Defines the interval in seconds between full topology updates.  
**Value**: 604800 (7 days)

**MPDC\_ERROR\_STRING\_DEPTH**  
Defines the number of error strings.  
**Value**: 26

**MPDC\_ERROR\_STRING\_WIDTH**  
Defines the maximum size in characters of an error string.  
**Value**: 128

**MPDC\_MESSAGE\_MAX\_SIZE**  
Defines the maximum message size, including maximum signature and certificate sizes.  
**Value**: 1400000

**MPDC\_MFK\_EXPIRATION\_PERIOD**  
Defines the MFK validity period in seconds.  
**Value**: 5184000 (60 days)

**MPDC\_MINIMUM\_PATH\_LENGTH**  
Defines the minimum file path length.  
**Value**: 9

**MPDC\_NETWORK\_CONNECTION\_MTU**  
Defines the MPDC packet buffer size.  
**Value**: 1500

**MPDC\_NETWORK\_DOMAIN\_NAME\_MAX\_SIZE**  
Defines the maximum domain name length in characters. The last character must be a string terminator.  
**Value**: 256

**MPDC\_NETWORK\_MAX\_AGENTS**  
Defines the maximum number of agent connections in a network.  
**Value**: 1000000

**MPDC\_NETWORK\_NODE\_ID\_SIZE**  
Defines the node identification string length.  
**Value**: 16

**MPDC\_PERIOD\_DAY\_TO\_SECONDS**  
Defines the number of seconds in a day.  
**Value**: 86400

**MPDC\_SOCKET\_TERMINATOR\_SIZE**  
Defines the packet delimiter byte size.  
**Value**: 1

**MPDC\_PACKET\_ERROR\_SIZE**  
Defines the packet error message byte size.  
**Value**: 1

**MPDC\_PACKET\_HEADER\_SIZE**  
Defines the MPDC packet header size.  
**Value**: 22

**MPDC\_PACKET\_SUBHEADER\_SIZE**  
Defines the MPDC packet sub-header size.  
**Value**: 16

**MPDC\_PACKET\_SEQUENCE\_TERMINATOR**  
Defines the sequence number of a packet that closes a connection.  
**Value**: 0xFFFFFFFFUL

**MPDC\_PACKET\_TIME\_SIZE**  
Defines the byte size of the serialized packet time parameter.  
**Value**: 8

**MPDC\_PACKET\_TIME\_THRESHOLD**  
Defines the maximum number of seconds a packet is valid.  
**Value**: 600 (default; can be modified)

**MPDC\_NETWORK\_TERMINATION\_MESSAGE\_SIZE**  
Defines the network termination message size.  
**Value**: 1

**MPDC\_NETWORK\_TERMINATION\_PACKET\_SIZE**  
Defines the network termination packet size, including the header and termination message.  
**Value**: MPDC\_PACKET\_HEADER\_SIZE + MPDC\_NETWORK\_TERMINATION\_MESSAGE\_SIZE

**Enums**

**10.3.2 mpdc\_configuration\_sets**

|  |  |
| --- | --- |
| Name | Description |
| mpdc\_configuration\_set\_none | No algorithm identifier is set. |
| mpdc\_configuration\_set\_dilithium1\_kyber1\_rcs256\_shake256 | The Dilithium-S1/Kyber-S1/RCS-256/SHAKE-256 algorithm set. |
| mpdc\_configuration\_set\_dilithium3\_kyber3\_rcs256\_shake256 | The Dilithium-S3/Kyber-S3/RCS-256/SHAKE-256 algorithm set. |
| mpdc\_configuration\_set\_dilithium5\_kyber5\_rcs256\_shake256 | The Dilithium-S5/Kyber-S5/RCS-256/SHAKE-256 algorithm set. |
| mpdc\_configuration\_set\_dilithium5\_kyber6\_rcs512\_shake512 | The Dilithium-S5/Kyber-S6/RCS-256/SHAKE-256 algorithm set. |
| mpdc\_configuration\_set\_sphincsplus1f\_mceliece1\_rcs256\_shake256 | The SPHINCS+-S1F/McEliece-S1/RCS-256/SHAKE-256 algorithm set. |
| mpdc\_configuration\_set\_sphincsplus1s\_mceliece1\_rcs256\_shake256 | The SPHINCS+-S1S/McEliece-S1/RCS-256/SHAKE-256 algorithm set. |
| mpdc\_configuration\_set\_sphincsplus3f\_mceliece3\_rcs256\_shake256 | The SPHINCS+-S3F/McEliece-S3/RCS-256/SHAKE-256 algorithm set. |
| mpdc\_configuration\_set\_sphincsplus3s\_mceliece3\_rcs256\_shake256 | The SPHINCS+-S3S/McEliece-S3/RCS-256/SHAKE-256 algorithm set. |
| mpdc\_configuration\_set\_sphincsplus5f\_mceliece5\_rcs256\_shake256 | The SPHINCS+-S5F/McEliece-S5a/RCS-256/SHAKE-256 algorithm set. |
| mpdc\_configuration\_set\_sphincsplus5s\_mceliece5\_rcs256\_shake256 | The SPHINCS+-S5S/McEliece-S5a/RCS-256/SHAKE-256 algorithm set. |
| mpdc\_configuration\_set\_sphincsplus5f\_mceliece6\_rcs256\_shake256 | The SPHINCS+-S5F/McEliece-S5b/RCS-256/SHAKE-256 algorithm set. |
| mpdc\_configuration\_set\_sphincsplus5s\_mceliece6\_rcs256\_shake256 | The SPHINCS+-S5S/McEliece-S5b/RCS-256/SHAKE-256 algorithm set. |
| mpdc\_configuration\_set\_sphincsplus5f\_mceliece7\_rcs256\_shake256 | The SPHINCS+-S5F/McEliece-S5c/RCS-256/SHAKE-256 algorithm set. |
| mpdc\_configuration\_set\_sphincsplus5s\_mceliece7\_rcs256\_shake256 | The SPHINCS+-S5S/McEliece-S5c/RCS-256/SHAKE-256 algorithm set. |

**10.3.3 mpdc\_network\_designations**

|  |  |
| --- | --- |
| Name | Description |
| mpdc\_network\_designation\_none | No designation was selected. |
| mpdc\_network\_designation\_agent | The device is an agent. |
| mpdc\_network\_designation\_client | The device is a client. |
| mpdc\_network\_designation\_dla | The device is the DLA. |
| mpdc\_network\_designation\_idg | The device is an inter-domain gateway. |
| mpdc\_network\_designation\_mas | The device is a server. |
| mpdc\_network\_designation\_remote | The device is a remote agent. |
| mpdc\_network\_designation\_rds | The device is an RDS security server. |
| mpdc\_network\_designation\_revoked | The device has been revoked. |
| mpdc\_network\_designation\_all | Every server and client device on the network. |

**10.3.4 mpdc\_network\_errors**

|  |  |
| --- | --- |
| Name | Description |
| mpdc\_network\_error\_none | No error was detected. |
| mpdc\_network\_error\_accept\_fail | The socket accept function returned an error. |
| mpdc\_network\_error\_auth\_failure | The cipher authentication has failed. |
| mpdc\_network\_error\_bad\_keep\_alive | The keep alive check failed. |
| mpdc\_network\_error\_channel\_down | The communications channel has failed. |
| mpdc\_network\_error\_connection\_failure | The device could not make a connection to the remote host. |
| mpdc\_network\_error\_decryption\_failure | The decryption authentication has failed. |
| mpdc\_network\_error\_establish\_failure | The transmission failed at the kex establish phase. |
| mpdc\_network\_error\_general\_failure | The connection experienced an unexpected error. |
| mpdc\_network\_error\_hosts\_exceeded | The server has run out of socket connections. |
| mpdc\_network\_error\_identity\_unknown | The identity could not be verified. |
| mpdc\_network\_error\_invalid\_input | The input is invalid. |
| mpdc\_network\_error\_invalid\_request | The request is invalid. |
| mpdc\_network\_error\_keep\_alive\_expired | The keep alive has expired with no response. |
| mpdc\_network\_error\_keep\_alive\_timeout | The keepalive failure counter has exceeded the maximum limit. |
| mpdc\_network\_error\_kex\_auth\_failure | The kex authentication has failed. |
| mpdc\_network\_error\_key\_not\_recognized | The key-id is not recognized. |
| mpdc\_network\_error\_key\_has\_expired | The certificate has expired. |
| mpdc\_network\_error\_listener\_fail | The listener function failed to initialize. |
| mpdc\_network\_error\_memory\_allocation | The server has run out of memory. |
| mpdc\_network\_error\_packet\_unsequenced | The packet was received out of sequence. |
| mpdc\_network\_error\_random\_failure | The random generator experienced a failure. |
| mpdc\_network\_error\_ratchet\_fail | The ratchet operation has failed. |
| mpdc\_network\_error\_receive\_failure | The receiver failed at the network layer. |
| mpdc\_network\_error\_transmit\_failure | The transmitter failed at the network layer. |
| mpdc\_network\_error\_unknown\_protocol | The protocol version is unknown. |
| mpdc\_network\_error\_unsequenced | The packet was received out of sequence. |
| mpdc\_network\_error\_verify\_failure | The expected data could not be verified. |

**10.3.5 mpdc\_network\_flags**

|  |  |
| --- | --- |
| Name | Description |
| mpdc\_network\_flag\_none | No flag was selected. |
| mpdc\_network\_flag\_connection\_terminate\_request | The packet contains a connection termination message. |
| mpdc\_network\_flag\_error\_condition | The connection experienced an error message. |
| mpdc\_network\_flag\_fragment\_collection\_request | The packet contains a server fragment collection request message. |
| mpdc\_network\_flag\_fragment\_collection\_response | The packet contains an agent fragment collection response message. |
| mpdc\_network\_flag\_fragment\_request | The packet contains a server fragment key request message. |
| mpdc\_network\_flag\_fragment\_response | The packet contains an agent fragment key response message. |
| mpdc\_network\_flag\_fragment\_query\_request | The packet contains a server fragment key query request message. |
| mpdc\_network\_flag\_fragment\_query\_response | The packet contains an agent fragment key query response message. |
| mpdc\_network\_flag\_incremental\_update\_request | The packet contains an incremental update request message. |
| mpdc\_network\_flag\_incremental\_update\_response | The packet contains an incremental update response message. |
| mpdc\_network\_flag\_register\_request | The packet contains a join request message. |
| mpdc\_network\_flag\_register\_response | The packet contains a join response message. |
| mpdc\_network\_flag\_register\_update\_request | The packet contains a join update request message. |
| mpdc\_network\_flag\_register\_update\_response | The packet contains a join update response |

**10.3.6 mpdc\_network\_flags enumeration documentation**

|  |  |
| --- | --- |
| Name | Description |
| mpdc\_network\_flag\_register\_update\_response | The packet contains a join update response message. |
| mpdc\_network\_flag\_keep\_alive\_request | The packet contains a keep alive request. |
| mpdc\_network\_flag\_keep\_alive\_response | The packet contains a keep alive response. |
| mpdc\_network\_flag\_mfk\_establish | The packet contains a server master fragment key establish message. |
| mpdc\_network\_flag\_mfk\_request | The packet contains a server master fragment key request message. |
| mpdc\_network\_flag\_mfk\_response | The packet contains a client MFK exchange response message. |
| mpdc\_network\_flag\_mfk\_verify | The packet contains a server master fragment key verify message. |
| mpdc\_network\_flag\_network\_announce\_broadcast | The packet contains a topology announce broadcast. |
| mpdc\_network\_flag\_network\_converge\_request | The packet contains a network convergence request message. |
| mpdc\_network\_flag\_network\_converge\_response | The packet contains a network convergence response message. |
| mpdc\_network\_flag\_network\_converge\_update | The packet contains a network convergence update message. |
| mpdc\_network\_flag\_network\_resign\_request | The packet contains a network resignation request message. |
| mpdc\_network\_flag\_network\_resign\_response | The packet contains a network resignation response message. |
| mpdc\_network\_flag\_network\_revocation\_broadcast | The packet contains a certificate revocation broadcast. |
| mpdc\_network\_flag\_network\_signature\_request | The packet contains a certificate signing request message. |
| mpdc\_network\_flag\_system\_error\_condition | The packet contains an error condition message. |
| mpdc\_network\_flag\_tunnel\_connection\_terminate | The packet contains a socket close message. |
| mpdc\_network\_flag\_tunnel\_encrypted\_message | The packet contains an encrypted message. |
| mpdc\_network\_flag\_tunnel\_session\_established | The exchange is in the established state. |
| mpdc\_network\_flag\_tunnel\_transfer\_request | Reserved - The host has received a transfer request. |
| mpdc\_network\_flag\_topology\_query\_request | The packet contains a topology query request message. |
| mpdc\_network\_flag\_topology\_query\_response | The packet contains a topology query response message. |
| mpdc\_network\_flag\_topology\_status\_request | The packet contains a topology status request message. |
| mpdc\_network\_flag\_topology\_status\_response | The packet contains a topology status response message. |
| mpdc\_network\_flag\_topology\_status\_available | The packet contains a topology status available message. |
| mpdc\_network\_flag\_topology\_status\_synchronized | The packet contains a topology status synchronized message. |
| mpdc\_network\_flag\_topology\_status\_unavailable | The packet contains a topology status unavailable message. |
| mpdc\_network\_flag\_network\_remote\_signing\_request | The packet contains a remote signing request message. |
| mpdc\_network\_flag\_network\_remote\_signing\_response | The packet contains a remote signing response message. |

**10.3.7 mpdc\_protocol\_errors**

|  |  |
| --- | --- |
| Name | Description |
| mpdc\_protocol\_error\_none | No error was detected. |
| mpdc\_protocol\_error\_authentication\_failure | The symmetric cipher had an authentication failure. |
| mpdc\_protocol\_error\_certificate\_not\_found | The node certificate could not be found. |
| mpdc\_protocol\_error\_channel\_down | The communications channel has failed. |
| mpdc\_protocol\_error\_connection\_failure | The device could not make a connection to the remote host. |
| mpdc\_protocol\_error\_connect\_failure | The transmission failed at the KEX connection phase. |
| mpdc\_protocol\_error\_convergence\_failure | The convergence call has returned an error. |
| mpdc\_protocol\_error\_convergence\_synchronized | The database is already synchronized. |
| mpdc\_protocol\_error\_decapsulation\_failure | The asymmetric cipher failed to decapsulate the shared secret. |
| mpdc\_protocol\_error\_decoding\_failure | The node or certificate decoding failed. |
| mpdc\_protocol\_error\_decryption\_failure | The decryption authentication has failed. |
| mpdc\_protocol\_error\_establish\_failure | The transmission failed at the KEX establish phase. |
| mpdc\_protocol\_error\_exchange\_failure | The transmission failed at the KEX exchange phase. |
| mpdc\_protocol\_error\_file\_not\_deleted | The application could not delete a local file. |
| mpdc\_protocol\_error\_file\_not\_found | The file could not be found. |
| mpdc\_protocol\_error\_file\_not\_written | The file could not be written to storage. |
| mpdc\_protocol\_error\_hash\_invalid | The public-key hash is invalid. |
| mpdc\_protocol\_error\_hosts\_exceeded | The server has run out of socket connections. |
| mpdc\_protocol\_error\_invalid\_request | The packet flag was unexpected. |
| mpdc\_protocol\_error\_certificate\_expired | The certificate has expired. |
| mpdc\_protocol\_error\_key\_expired | The MPDC public key has expired. |
| mpdc\_protocol\_error\_key\_unrecognized | The key identity is unrecognized. |
| mpdc\_protocol\_error\_listener\_fail | The listener function failed to initialize. |
| mpdc\_protocol\_error\_memory\_allocation | The server has run out of memory. |
| mpdc\_protocol\_error\_message\_time\_invalid | The network time is invalid or has substantial delay. |
| mpdc\_protocol\_error\_message\_verification\_failure | The expected data could not be verified. |
| mpdc\_protocol\_error\_no\_usable\_address | The server has no usable IP address assigned in the configuration. |
| mpdc\_protocol\_error\_node\_not\_available | The node is not available for a session. |
| mpdc\_protocol\_error\_node\_not\_found | The node could not be found in the database. |
| mpdc\_protocol\_error\_node\_was\_registered | The node was previously registered in the database. |
| mpdc\_protocol\_error\_operation\_cancelled | The operation was cancelled by the user. |
| mpdc\_protocol\_error\_packet\_header\_invalid | The packet header received was invalid. |
| mpdc\_protocol\_error\_packet\_unsequenced | The packet was received out of sequence. |
| mpdc\_protocol\_error\_receive\_failure | The receiver failed at the network layer. |
| mpdc\_protocol\_error\_root\_signature\_invalid | The root signature failed authentication. |
| mpdc\_protocol\_error\_serialization\_failure | The certificate could not be serialized. |
| mpdc\_protocol\_error\_signature\_failure | The signature scheme could not sign a message. |
| mpdc\_protocol\_error\_signing\_failure | The transmission failed to sign the data. |
| mpdc\_protocol\_error\_socket\_binding | The socket could not be bound to an IP address. |
| mpdc\_protocol\_error\_socket\_creation | The socket could not be created. |
| mpdc\_protocol\_error\_transmit\_failure | The transmitter failed at the network layer. |
| mpdc\_protocol\_error\_topology\_no\_agent | The topological database has no agent entries. |
| mpdc\_protocol\_error\_unknown\_protocol | The protocol string was not recognized. |
| mpdc\_protocol\_error\_verification\_failure | The transmission failed at the KEX verify phase. |

**Structs**

**10.3.8 mpdc\_certificate\_expiration**

|  |  |
| --- | --- |
| Name | Description |
| from | The starting time in seconds. |
| to | The expiration time in seconds. |

**10.3.9 mpdc\_child\_certificate**

|  |  |
| --- | --- |
| Name | Description |
| csig | The certificate’s signed hash. |
| verkey | The serialized public verification key. |
| issuer | The certificate issuer. |
| serial | The certificate serial number. |
| rootser | The root certificate's serial number. |
| expiration | The from and to certificate expiration times. |
| designation | The certificate type designation. |
| algorithm | The algorithm configuration identifier. |
| version | The certificate version. |

**10.3.10 mpdc\_idg\_hint**

|  |  |
| --- | --- |
| Name | Description |
| chash | The remote certificate's signed hash. |
| rootser | The remote certificate's root serial number. |

**10.3.11 mpdc\_idg\_certificate**

|  |  |
| --- | --- |
| Name | Description |
| csig | The certificate’s signed hash. |
| vkey | The serialized public verification key. |
| xcert | The serialized X509 certificate. |
| serial | The certificate serial number. |
| rootser | The root certificate's serial number. |
| hint | The certificate's topological hint. |
| issuer | The certificate issuer. |
| expiration | The from and to certificate expiration times. |
| designation | The certificate type designation. |
| algorithm | The algorithm configuration identifier. |
| version | The certificate version. |

**10.3.12 mpdc\_connection\_state**

|  |  |
| --- | --- |
| Name | Description |
| target | The target socket structure. |
| rxcpr | The receive channel cipher state. |
| txcpr | The transmit channel cipher state. |
| rxseq | The receive channel's packet sequence number. |
| txseq | The transmit channel's packet sequence number. |
| instance | The connection’s instance count. |
| exflag | The network stage flag. |

**10.3.13 mpdc\_keep\_alive\_state**

|  |  |
| --- | --- |
| Name | Description |
| target | The target socket structure. |
| etime | The keep alive epoch time. |
| seqctr | The keep alive packet sequence counter. |
| recd | The keep alive response received status. |

**10.3.14 mpdc\_mfkey\_state**

|  |  |
| --- | --- |
| Name | Description |
| serial | The MFK serial number. |
| mfk | The master fragment key. |

**10.3.15 mpdc\_network\_packet**

|  |  |
| --- | --- |
| Name | Description |
| flag | The packet flag. |
| msglen | The packet's message length. |
| sequence | The packet sequence number. |
| utctime | The UTC time the packet was created in seconds. |
| pmessage | A pointer to the packet's message buffer. |

**10.3.16 mpdc\_root\_certificate**

|  |  |
| --- | --- |
| Name | Description |
| verkey | The serialized public key. |
| issuer | The certificate issuer text name. |
| serial | The certificate serial number. |
| expiration | The from and to certificate expiration times. |
| algorithm | The signature algorithm identifier. |
| version | The certificate version type. |

**10.3.17 mpdc\_serialized\_symmetric\_key**

|  |  |
| --- | --- |
| Name | Description |
| keyid | The key identity. |
| key | The symmetric key. |
| nonce | The symmetric nonce. |

**10.3.18 mpdc\_signature\_keypair**

|  |  |
| --- | --- |
| Name | Description |
| prikey | The secret signing key. |
| pubkey | The public signature verification key. |

**10.3.19 mpdc\_cipher\_keypair**

|  |  |
| --- | --- |
| Name | Description |
| prikey | The asymmetric cipher private key. |
| pubkey | The asymmetric cipher public key. |

**Functions**

**10.3.20 Function:** mpdc\_connection\_close

**Purpose:** Closes the network connection between hosts.

**Parameters:**

* rsock (Type: qsc\_socket\*): A pointer to the remote socket.
* err (Type: mpdc\_network\_errors): The error message.
* notify (Type: bool): Notify the remote host that the connection is closing.

**10.3.21 Function:** mpdc\_decrypt\_packet

**Purpose:** Decrypts a message and copies it to the message output.

**Parameters:**

* cns (Type: mpdc\_connection\_state\*): A pointer to the connection state structure.
* message (Type: uint8\_t\*): The message output array.
* msglen (Type: size\_t\*): A pointer receiving the message length.
* packetin (Type: const mpdc\_network\_packet\*): A pointer to the input packet structure.

**Returns:** mpdc\_network\_errors - The function error state.

**10.3.22 Function:** mpdc\_encrypt\_packet

**Purpose:** Encrypts a message and builds an output packet.

**Parameters:**

* cns (Type: mpdc\_connection\_state\*): A pointer to the connection state structure.
* packetout (Type: mpdc\_network\_packet\*): A pointer to the output packet structure.
* message (Type: const uint8\_t\*): The input message array.
* msglen (Type: size\_t): The length of the message array.

**Returns:** mpdc\_network\_errors - The function error state.

**10.3.23 Function:** mpdc\_connection\_state\_dispose

**Purpose:** Disposes of the tunnel state.

**Parameters:**

* cns (Type: mpdc\_connection\_state\*): The tunnel connection state.

**10.3.24 Function:** mpdc\_network\_error\_to\_string

**Purpose:** Returns a pointer to a string description of a network error code.

**Parameters:**

* error (Type: mpdc\_network\_errors): The network error type.

**Returns:** const char\* - A pointer to an error string or NULL.

**10.3.25 Function:** mpdc\_protocol\_error\_to\_string

**Purpose:** Returns a pointer to a string description of a protocol error code.

**Parameters:**

* error (Type: mpdc\_protocol\_errors): The protocol error type.

**Returns:** const char\* - A pointer to an error string or NULL.

**10.3.26 Function:** mpdc\_packet\_clear

**Purpose:** Clears a packet's state.

**Parameters:**

* packet (Type: mpdc\_network\_packet\*): A pointer to the packet structure.

**10.3.27 Function:** mpdc\_packet\_error\_message

**Purpose:** Populates a packet structure with an error message.

**Parameters:**

* packet (Type: mpdc\_network\_packet\*): A pointer to the packet structure.
* error (Type: mpdc\_protocol\_errors): The error type.

**10.3.28 Function:** mpdc\_packet\_header\_deserialize

**Purpose:** Deserializes a byte array to a packet header.

**Parameters:**

* header (Type: const uint8\_t\*): The header byte array to deserialize.
* packet (Type: mpdc\_network\_packet\*): A pointer to the packet structure.

**10.3.29 Function:** mpdc\_packet\_header\_serialize

**Purpose:** Serializes a packet header to a byte array.

**Parameters:**

* packet (Type: const mpdc\_network\_packet\*): A pointer to the packet structure to serialize.
* header (Type: uint8\_t\*): The header byte array.

**10.3.30 Function:** mpdc\_packet\_set\_utc\_time

**Purpose:** Sets the local UTC seconds time in the packet header.

**Parameters:**

* packet (Type: mpdc\_network\_packet\*): A pointer to a network packet.

**10.3.31 Function:** mpdc\_packet\_time\_valid

**Purpose:** Checks the local UTC seconds time against the packet sent time for validity within the packet time threshold.

**Parameters:**

* packet (Type: const mpdc\_network\_packet\*): A pointer to a network packet.

**Returns:** bool - Returns true if the packet was received within the valid-time threshold.

**10.3.32 Function:** mpdc\_packet\_to\_stream

**Purpose:** Serializes a packet to a byte array.

**Parameters:**

* packet (Type: const mpdc\_network\_packet\*): A pointer to the packet.
* pstream (Type: uint8\_t\*): A pointer to the packet structure.

**Returns:** size\_t - The size of the byte stream.

**10.3.33 Function:** mpdc\_stream\_to\_packet

**Purpose:** Deserializes a byte array to a packet.

**Parameters:**

* pstream (Type: const uint8\_t\*): The header byte array to deserialize.
* packet (Type: mpdc\_network\_packet\*): A pointer to the packet structure.

**10.4 Network.h**

**10.4.1 mpdc\_network\_register\_update\_request\_state**

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| address | const char\* | The server address |
| lcert | const mpdc\_child\_certificate\* | A pointer to the local certificate |
| list | mpdc\_topology\_list\_state\* | A pointer to the topology list |
| rcert | mpdc\_child\_certificate\* | A pointer to the remote certificate |
| root | const mpdc\_root\_certificate\* | A pointer to the root certificate |
| sigkey | const uint8\_t\* | A pointer to the secret signing key |

**10.4.2 mpdc\_network\_register\_update\_response\_state**

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| csock | const qsc\_socket\* | A pointer to the connected socket |
| lcert | const mpdc\_child\_certificate\* | A pointer to the local certificate |
| list | const mpdc\_topology\_list\_state\* | A pointer to the topology list |
| rcert | mpdc\_child\_certificate\* | A pointer to the output remote certificate |
| root | const mpdc\_root\_certificate\* | A pointer to the root certificate |
| sigkey | const uint8\_t\* | A pointer to the secret signing key |

**10.4.3 mpdc\_network\_mfk\_request\_state**

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| lcert | const mpdc\_child\_certificate\* | A pointer to the local certificate |
| mfk | uint8\_t\* | A pointer to the master fragment key |
| rcert | const mpdc\_child\_certificate\* | A pointer to the remote certificate |
| rnode | const mpdc\_topology\_node\_state\* | A pointer to the remote node structure |
| root | const mpdc\_root\_certificate\* | A pointer to the root certificate |
| sigkey | const uint8\_t\* | A pointer to the secret signing key |

**10.4.4 mpdc\_network\_mfk\_response\_state**

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| csock | const qsc\_socket\* | A pointer to the connected socket |
| ckp | mpdc\_cipher\_keypair | The asymmetric encryption key-pair |
| lcert | const mpdc\_child\_certificate\* | A pointer to the local certificate |
| mfk | uint8\_t\* | A pointer to the master fragment key |
| rcert | mpdc\_child\_certificate\* | A pointer to the remote certificate |
| root | const mpdc\_root\_certificate\* | A pointer to the root certificate |
| sigkey | const uint8\_t\* | A pointer to the secret signing key |

**10.4.5 mpdc\_network\_remote\_signing\_request\_state**

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| address | const char\* | The RDS server address |
| rcert | mpdc\_child\_certificate\* | A pointer to the remote certificate |
| root | const mpdc\_root\_certificate\* | A pointer to the root certificate |
| sigkey | const uint8\_t\* | A pointer to the secret signing key |

**10.4.6 mpdc\_network\_remote\_signing\_response\_state**

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| csock | qsc\_socket\* | A pointer to the connected socket |
| dcert | mpdc\_child\_certificate\* | A pointer to the DLA certificate |
| rcert | mpdc\_child\_certificate\* | A pointer to the remote certificate |
| root | const mpdc\_root\_certificate\* | A pointer to the root certificate |
| sigkey | const uint8\_t\* | A pointer to the secret signing key |

**10.4.7 mpdc\_network\_resign\_request\_state**

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| address | const char\* | The server address |
| lnode | const mpdc\_topology\_node\_state\* | A pointer to the local node structure |
| sigkey | const uint8\_t\* | A pointer to the secret signing key |

**10.4.8 mpdc\_network\_resign\_response\_state**

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| list | const mpdc\_topology\_list\_state\* | A pointer to the topology list |
| rcert | mpdc\_child\_certificate\* | A pointer to the remote certificate |
| rnode | mpdc\_topology\_node\_state\* | A pointer to the remote node structure |
| sigkey | const uint8\_t\* | A pointer to the secret signing key |

**10.4.9 mpdc\_network\_revoke\_request\_state**

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| designation | mpdc\_network\_designations | The node type designation |
| list | const mpdc\_topology\_list\_state\* | A pointer to the node database |
| rnode | const mpdc\_topology\_node\_state\* | A pointer to the remote node structure |
| sigkey | const uint8\_t\* | A pointer to the secret signing key |

**10.4.10 mpdc\_network\_revoke\_response\_state**

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| list | const mpdc\_topology\_list\_state\* | A pointer to the node database |
| rnode | mpdc\_topology\_node\_state\* | A pointer to the remote node structure |
| dcert | const mpdc\_child\_certificate\* | A pointer to the DLA certificate |

**10.4.11 mpdc\_network\_topological\_query\_request\_state**

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| dcert | const mpdc\_child\_certificate\* | A pointer to the DLA certificate |
| dnode | mpdc\_topology\_node\_state\* | A pointer to the DLA node structure |
| issuer | const char\* | A pointer to the query issuer string |
| rnode | mpdc\_topology\_node\_state\* | A pointer to the return remote node structure |
| serial | const uint8\_t\* | A pointer to the local serial number |
| sigkey | const uint8\_t\* | A pointer to the secret signing key |

**10.4.12 mpdc\_network\_topological\_query\_response\_state**

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| csock | const qsc\_socket\* | The connected socket |
| ccert | const mpdc\_child\_certificate\* | A pointer to the remote client’s certificate |
| rnode | const mpdc\_topology\_node\_state\* | A pointer to the remote node structure |
| sigkey | const uint8\_t\* | A pointer to the secret signing key |

**10.4.13 Function:** mpdc\_network\_announce\_broadcast

**Purpose:** Announces a certificate using the DLA and broadcasts it to the network.

**Parameters:**

* state (Type: mpdc\_network\_announce\_request\_state\*): The announce state structure.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.14 Function:** mpdc\_network\_announce\_response

**Purpose:** Processes an announce response message.

**Parameters:**

* state (Type: mpdc\_network\_announce\_response\_state\*): The announce response state structure.
* packetin (Type: const mpdc\_network\_packet\*): The input packet containing the announce request.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.15 Function:** mpdc\_network\_application\_to\_port

**Purpose:** Retrieves the network designation from a port number.

**Parameters:**

* tnode (Type: mpdc\_network\_designations): The target network designation type.

**Returns:** uint16\_t - The port number, or zero if the node type is invalid.

**10.4.16 Function:** mpdc\_network\_broadcast\_message

**Purpose:** Broadcasts a message to a node type on the network.

**Parameters:**

* list (Type: const mpdc\_topology\_list\_state\*): A pointer to the topology list.
* message (Type: const uint8\_t\*): The message to send.
* msglen (Type: size\_t): The length of the message.
* tnode (Type: mpdc\_network\_designations): The target node-type designation.

**Returns:** void

**10.4.17 Function:** mpdc\_network\_certificate\_verify

**Purpose:** Verifies a certificate’s format and root signature.

**Parameters:**

* ccert (Type: const mpdc\_child\_certificate\*): The child certificate.
* root (Type: const mpdc\_root\_certificate\*): The root certificate.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.18 Function:** mpdc\_network\_connect\_to\_address

**Purpose:** Connects a socket to a remote address and port.

**Parameters:**

* csock (Type: qsc\_socket\*): A pointer to the socket.
* address (Type: const char\*): The remote host’s address.
* port (Type: uint16\_t): The application port number.

**Returns:** qsc\_socket\_exceptions - The socket error.

**10.4.19 Function:** mpdc\_network\_connect\_to\_device

**Purpose:** Connects a socket to a remote address based on designation.

**Parameters:**

* csock (Type: qsc\_socket\*): A pointer to the socket.
* address (Type: const char\*): The remote host’s address.
* designation (Type: mpdc\_network\_designations): The remote host’s designation.

**Returns:** qsc\_socket\_exceptions - The socket error.

**10.4.20 Function:** mpdc\_network\_converge\_request

**Purpose:** Sends a convergence request from the DLA and broadcasts it to the network.

**Parameters:**

* state (Type: const mpdc\_network\_converge\_request\_state\*): The converge request state structure.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.21 Function:** mpdc\_network\_converge\_response

**Purpose:** Responds to a DLA network converge request.

**Parameters:**

* state (Type: const mpdc\_network\_converge\_response\_state\*): The converge response state structure.
* packetin (Type: const mpdc\_network\_packet\*): The input packet containing the verify response.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.22 Function:** mpdc\_network\_converge\_update\_verify

**Purpose:** Processes a converge response update message.

**Parameters:**

* state (Type: mpdc\_network\_converge\_update\_verify\_state\*): The converge update verify state structure.
* packetin (Type: const mpdc\_network\_packet\*): The input packet containing the verify response.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.23 Function:** mpdc\_network\_fkey\_request

**Purpose:** Requests and executes a key exchange for a fragmentation key.

**Parameters:**

* state (Type: mpdc\_network\_fkey\_request\_state\*): The fkey request state structure.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.24 Function:** mpdc\_network\_fkey\_response

**Purpose:** Responds to a key exchange request for a fragmentation key.

**Parameters:**

* state (Type: mpdc\_network\_fkey\_response\_state\*): The fkey response state structure.
* packetin (Type: const mpdc\_network\_packet\*): The input packet containing the request.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.25 Function:** mpdc\_network\_fragment\_collection\_request

**Purpose:** Requests a fragment collection from a MAS.

**Parameters:**

* state (Type: mpdc\_network\_fragment\_collection\_request\_state\*): The fragment collection request state.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.26 Function:** mpdc\_network\_fragment\_collection\_response

**Purpose:** Sends a collection response from the MAS to a client.

**Parameters:**

* state (Type: mpdc\_network\_fragment\_collection\_response\_state\*): The fkey response state structure.
* packetin (Type: const mpdc\_network\_packet\*): The input packet containing the request.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.27 Function:** mpdc\_network\_fragment\_query\_response

**Purpose:** Sends a fragment query response from an agent to a MAS.

**Parameters:**

* state (Type: mpdc\_network\_fragment\_query\_response\_state\*): The fragment query response state structure.
* packetin (Type: const mpdc\_network\_packet\*): The input packet containing the request.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.28 Function:** mpdc\_network\_get\_local\_address

**Purpose:** Retrieves the local IP address.

**Parameters:**

* address (Type: char[MPDC\_CERTIFICATE\_ADDRESS\_SIZE]): Output array to store the local address.

**Returns:** bool - Returns true if the address is successfully retrieved.

**10.4.29 Function:** mpdc\_network\_incremental\_update\_request

**Purpose:** Sends an incremental update request.

**Parameters:**

* state (Type: const mpdc\_network\_incremental\_update\_request\_state\*): The incremental update request function state.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.30 Function:** mpdc\_network\_incremental\_update\_response

**Purpose:** Sends a copy of a certificate to a remote host in response to an incremental update.

**Parameters:**

* state (Type: const mpdc\_network\_incremental\_update\_response\_state\*): The update response function state.
* packetin (Type: const mpdc\_network\_packet\*): The input packet containing the request.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.31 Function:** mpdc\_network\_mfk\_exchange\_request

**Purpose:** Requests and executes a key exchange request for a master fragmentation key.

**Parameters:**

* state (Type: mpdc\_network\_mfk\_request\_state\*): The MFK request state structure.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.32 Function:** mpdc\_network\_mfk\_exchange\_response

**Purpose:** Responds to a key exchange request for a master fragmentation key.

**Parameters:**

* state (Type: mpdc\_network\_mfk\_response\_state\*): The MFK response state structure.
* packetin (Type: const mpdc\_network\_packet\*): The input packet containing the request.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.33 Function:** mpdc\_network\_port\_to\_application

**Purpose:** Gets the network designation based on a port number.

**Parameters:**

* port (Type: uint16\_t): The network application port.

**Returns:** mpdc\_network\_designations - The network designation type.

**10.4.34 Function:** mpdc\_network\_register\_request

**Purpose:** Sends an Agent join request to the DLA.

**Parameters:**

* state (Type: mpdc\_network\_register\_request\_state\*): The join request function state.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.35 Function:** mpdc\_network\_register\_response

**Purpose:** Sends a join response to the agent.

**Parameters:**

* state (Type: mpdc\_network\_register\_response\_state\*): The join response function state.
* packetin (Type: const mpdc\_network\_packet\*): The input packet containing the request.

**Returns:** mpdc\_protocol\_errors - A protocol error flag.

**10.4.36 Function:** mpdc\_network\_register\_update\_request

**Purpose:** Sends a MAS or Client join update request to the DLA.

**Parameters:**

* state (Type: mpdc\_network\_register\_update\_request\_state\*): The join update request function state.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.37 Function:** mpdc\_network\_register\_update\_response

**Purpose:** Sends a join update response to the server or client.

**Parameters:**

* state (Type: mpdc\_network\_register\_update\_response\_state\*): The join response function state.
* packetin (Type: const mpdc\_network\_packet\*): The input packet containing the request.

**Returns:** mpdc\_protocol\_errors - A protocol error flag.

**10.4.38 Function:** mpdc\_network\_remote\_signing\_request

**Purpose:** Sends a certificate signing request from the DLA to the RDS.

**Parameters:**

* state (Type: mpdc\_network\_remote\_signing\_request\_state\*): The remote signing request state.

**Returns:** mpdc\_protocol\_errors - A protocol error flag.

**10.4.39 Function:** mpdc\_network\_remote\_signing\_response

**Purpose:** Sends a signed certificate response from the RDS to the DLA.

**Parameters:**

* state (Type: mpdc\_network\_remote\_signing\_response\_state\*): The remote signing response state.
* packetin (Type: const mpdc\_network\_packet\*): The input packet containing the request.

**Returns:** mpdc\_protocol\_errors - A protocol error flag.

**10.4.40 Function:** mpdc\_network\_resign\_request

**Purpose:** Sends a resign request to the DLA.

**Parameters:**

* state (Type: mpdc\_network\_resign\_request\_state\*): The resign request state structure.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.41 Function:** mpdc\_network\_resign\_response

**Purpose:** Sends a resign response to the agent or server.

**Parameters:**

* state (Type: mpdc\_network\_resign\_response\_state\*): The resign response state structure.
* packetin (Type: const mpdc\_network\_packet\*): The input packet containing the request.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.42 Function:** mpdc\_network\_revoke\_broadcast

**Purpose:** Sends a revocation request from the DLA.

**Parameters:**

* state (Type: mpdc\_network\_revoke\_request\_state\*): The revocation broadcast function state.

**Returns:** mpdc\_protocol\_errors - A protocol error flag.

**10.4.43 Function:** mpdc\_network\_revoke\_response

**Purpose:** Verifies a revocation request sent from the DLA.

**Parameters:**

* state (Type: mpdc\_network\_revoke\_response\_state\*): The revocation verify function state.
* packetin (Type: const mpdc\_network\_packet\*): The input packet containing the request.

**Returns:** mpdc\_protocol\_errors - A protocol error flag.

**10.4.44 Function:** mpdc\_network\_send\_error

**Purpose:** Sends an error message.

**Parameters:**

* csock (Type: const qsc\_socket\*): A pointer to the socket.
* error (Type: mpdc\_protocol\_errors): The error code.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.45 Function:** mpdc\_network\_socket\_dispose

**Purpose:** Shuts down and disposes of a socket instance.

**Parameters:**

* csock (Type: qsc\_socket\*): A pointer to the socket.

**Returns:** void

**10.4.46 Function:** mpdc\_network\_topological\_query\_request

**Purpose:** Queries a device for its topological information.

**Parameters:**

* state (Type: const mpdc\_network\_topological\_query\_request\_state\*): The topological query request state.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.47 Function:** mpdc\_network\_topological\_query\_response

**Purpose:** Responds to a topological query request.

**Parameters:**

* state (Type: const mpdc\_network\_topological\_query\_response\_state\*): The topological query response state.
* packetin (Type: const mpdc\_network\_packet\*): The packet containing the topological query request.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.48 Function:** mpdc\_network\_topological\_status\_request

**Purpose:** Sends a status request from the DLA to a client device.

**Parameters:**

* state (Type: const mpdc\_network\_topological\_status\_request\_state\*): The topological status request state.
* query (Type: const char\*): The device query string.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.49 Function:** mpdc\_network\_topological\_status\_response

**Purpose:** Processes the status response from the client device and sends a response.

**Parameters:**

* state (Type: const mpdc\_network\_topological\_status\_response\_state\*): The topological status response state.
* packetin (Type: const mpdc\_network\_packet\*): The packet containing the topological status request.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.4.50 Function:** mpdc\_network\_topological\_status\_verify

**Purpose:** Verifies the status response from the DLA.

**Parameters:**

* state (Type: const mpdc\_network\_topological\_status\_request\_state\*): The topological status verify state.
* packetin (Type: const mpdc\_network\_packet\*): The packet containing the topological status response.

**Returns:** mpdc\_protocol\_errors - The error code.

**10.5 Topology.h**

**10.5.1 Function:** mpdc\_topology\_address\_from\_issuer

**Purpose:** Retrieves an IP address based on an issuer string.

**Parameters:**

* address (Type: char\*): The output array for the node’s network address.
* issuer (Type: const char\*): The issuer string to look up.
* list (Type: const mpdc\_topology\_list\*): Pointer to the topology list.

**10.5.2 Function:** mpdc\_topology\_node\_add\_alias

**Purpose:** Adds an alias string to an issuer path.

**Parameters:**

* node (Type: mpdc\_topology\_node\*): The network node to update.
* alias (Type: const char\*): The alias to add.

**10.5.6 Function:** mpdc\_topology\_nodes\_are\_equal

**Purpose:** Compares two topological nodes for equality.

**Parameters:**

* a (Type: const mpdc\_topology\_node\*): First node for comparison.
* b (Type: const mpdc\_topology\_node\*): Second node for comparison.

**Returns:** bool - Returns true if the nodes are identical.

**10.5.7 Function:** mpdc\_topology\_child\_add\_empty\_node

**Purpose:** Retrieves an empty node pointer from the topology list (not thread-safe).

**Parameters:**

* list (Type: mpdc\_topology\_list\*): Pointer to the topology list.

**Returns:** mpdc\_topology\_node\* - Pointer to the node or NULL.

**10.5.8 Function:** mpdc\_topology\_child\_add\_item

**Purpose:** Adds a node to the topology list.

**Parameters:**

* list (Type: mpdc\_topology\_list\*): Pointer to the topology list.
* node (Type: const mpdc\_topology\_node\*): Node to add.

**10.5.9 Function:** mpdc\_topology\_canonical\_to\_issuer\_name

**Purpose:** Converts a canonical name to an issuer name.

**Parameters:**

* issuer (Type: char\*): Output issuer name.
* isslen (Type: size\_t): Length of the issuer name.
* domain (Type: const char\*): The domain name.
* cname (Type: const char\*): Input device canonical name.

**Returns:** bool - Returns false if the conversion failed.

**10.5.10 Function:** mpdc\_topology\_issuer\_to\_canonical\_name

**Purpose:** Converts an issuer name to a canonical name.

**Parameters:**

* cname (Type: char\*): Output canonical name.
* namelen (Type: size\_t): Length of the canonical name string.
* issuer (Type: const char\*): Input issuer name.

**Returns:** bool - Returns false if the conversion failed.

**10.5.11 Function:** mpdc\_topology\_child\_register

**Purpose:** Registers a child node to a topology list.

**Parameters:**

* list (Type: mpdc\_topology\_list\*): Pointer to the topology list.
* ccert (Type: const mpdc\_child\_certificate\*): Node's child certificate.
* address (Type: const char\*): Node's network address.

**10.5.12 Function:** mpdc\_topology\_list\_clone

**Purpose:** Clones a topology list.

**Parameters:**

* tlist (Type: const mpdc\_topology\_list\*): Pointer to the topology list to clone.
* tcopy (Type: mpdc\_topology\_list\*): Pointer to the new list.

**10.5.13 Function:** mpdc\_topology\_list\_deserialize

**Purpose:** Deserializes a topology list.

**Parameters:**

* list (Type: mpdc\_topology\_list\*): Pointer to the topology list.
* input (Type: const uint8\_t\*): The serialized list.
* inplen (Type: size\_t): Size of the input array.

**10.5.14 Function:** mpdc\_topology\_list\_dispose

**Purpose:** Disposes of the topology list and releases memory.

**Parameters:**

* list (Type: mpdc\_topology\_list\*): Pointer to the topology list.

**10.5.15 Function:** mpdc\_topology\_list\_initialize

**Purpose:** Initializes the topology list.

**Parameters:**

* list (Type: mpdc\_topology\_list\*): Topology list state.

**10.5.16 Function:** mpdc\_topology\_list\_item

**Purpose:** Retrieves a node from an index in the topology list.

**Parameters:**

* list (Type: mpdc\_topology\_list\*): Topology list state.
* node (Type: mpdc\_topology\_node\*): Pointer to the node structure.
* index (Type: size\_t): Node index.

**Returns:** bool - Returns false if the item was not found.

**10.5.17 Function:** mpdc\_topology\_list\_remove\_duplicates

**Purpose:** Removes duplicate nodes from the topology list.

**Parameters:**

* list (Type: mpdc\_topology\_list\*): Topology list state.

**Returns:** size\_t - Number of items in the list.

**10.5.18 Function:** mpdc\_topology\_list\_server\_count

**Purpose:** Counts nodes of a specified type in the database.

**Parameters:**

* list (Type: const mpdc\_topology\_list\*): Topology list state structure.
* ntype (Type: mpdc\_network\_designations): Type of node to count.

**Returns:** size\_t - Number of nodes found.

**10.5.19 Function:** mpdc\_topology\_list\_serialize

**Purpose:** Serializes a topology list.

**Parameters:**

* output (Type: uint8\_t\*): Output array for serialized topology.
* list (Type: const mpdc\_topology\_list\*): Topology list state structure.

**Returns:** size\_t - Size of the serialized topology.

**10.5.20 Function:** mpdc\_topology\_list\_size

**Purpose:** Returns the byte size of a serialized topology list.

**Parameters:**

* list (Type: const mpdc\_topology\_list\*): Topology list state structure.

**10.5.21 Function:** mpdc\_topology\_list\_to\_string

**Purpose:** Converts the topology list to a printable string.

**Parameters:**

* list (Type: const mpdc\_topology\_list\*): Topology list state structure.
* output (Type: char\*): Output array for the string.
* outlen (Type: size\_t): Length of the output array.

**Returns:** size\_t - Byte size of the serialized topology.

**10.5.22 Function:** mpdc\_topology\_list\_update\_pack

**Purpose:** Packs a node update set into an array.

**Parameters:**

* output (Type: uint8\_t\*): Output array for serialized topology.
* list (Type: const mpdc\_topology\_list\*): Topology list state structure.
* ntype (Type: mpdc\_network\_designations): Type of node entry to pack.

**Returns:** size\_t - Size of the serialized topology.

**10.5.23 Function:** mpdc\_topology\_list\_update\_unpack

**Purpose:** Unpacks a node update set into the topology list.

**Parameters:**

* list (Type: mpdc\_topology\_list\*): Topology list state structure.
* input (Type: const uint8\_t\*): Serialized topology array.
* inplen (Type: size\_t): Length of the input array.

**10.5.24 Function:** mpdc\_topology\_ordered\_server\_list

**Purpose:** Returns a sorted list of nodes by serial number.

**Parameters:**

* olist (Type: mpdc\_topology\_list\*): Sorted output topology list.
* tlist (Type: const mpdc\_topology\_list\*): Unsorted input topology list.
* ntype (Type: mpdc\_network\_designations): Type of node entry to sort.

**Returns:** size\_t - Number of nodes in the list.

**10.5.25 Function:** mpdc\_topology\_node\_clear

**Purpose:** Erases a node structure.

**Parameters:**

* node (Type: mpdc\_topology\_node\*): Pointer to the topology node to erase.

**10.5.26 Function:** mpdc\_topology\_node\_copy

**Purpose:** Copies a source node to a destination node structure.

**Parameters:**

* source (Type: const mpdc\_topology\_node\*): Pointer to the source node.
* destination (Type: mpdc\_topology\_node\*): Pointer to the destination node.

**10.5.27 Function:** mpdc\_topology\_node\_deserialize

**Purpose:** Deserializes a serialized topological node.

**Parameters:**

* node (Type: mpdc\_topology\_node\*): Pointer to the topology node.
* input (Type: const uint8\_t\*): Serialized topology node array.

**10.5.28 Function:** mpdc\_topology\_node\_encode

**Purpose:** Encodes a topological node into a printable string.

**Parameters:**

* node (Type: mpdc\_topology\_node\*): Pointer to the topology node.
* output (Type: char\*): Serialized node string.

**Returns:** size\_t - Size of the serialized node.

**10.5.29 Function:** mpdc\_topology\_node\_exists

**Purpose:** Checks if a node exists in the topology list by serial number.

**Parameters:**

* list (Type: const mpdc\_topology\_list\*): Topology list state.
* serial (Type: const uint8\_t\*): Node's serial number.

**Returns:** bool - Returns true if the node exists.

**10.5.30 Function:** mpdc\_topology\_node\_find

**Purpose:** Finds a node in the list by serial number.

**Parameters:**

* list (Type: const mpdc\_topology\_list\*): Topology list state.
* node (Type: mpdc\_topology\_node\*): Pointer to the destination node.
* serial (Type: const uint8\_t\*): Certificate serial number.

**Returns:** bool - Returns false if the node was not found.

**10.6 Agent**

**9.6.1 Function:** mpdc\_agent\_pause\_server

**Purpose:** Pause the Agent server

**Returns:** void

**10.6.2 Function:** mpdc\_agent\_start\_server

**Purpose:** Start the Agent server

**Returns:** int - Returns zero on success

**10.6.3 Function:** mpdc\_agent\_stop\_server

**Purpose:** Stop the Agent server

**Returns:** void

**10.7 Client**

**10.7.1 Function:** mpdc\_client\_pause\_server

**Purpose:** Pause the Client server

**Returns:** void

**10.7.2 Function:** mpdc\_client\_start\_server

**Purpose:** Start the Client server

**Returns:** int - Returns zero on success

**10.7.3 Function:** mpdc\_client\_stop\_server

**Purpose:** Stop the Client server

**Returns:** void

**10.8 DLA**

**10.8.1 Function:** mpdc\_dla\_pause\_server

**Purpose:** Pause the DLA server

**Returns:** void

**10.8.2 Function:** mpdc\_dla\_start\_server

**Purpose:** Start the DLA server

**Returns:** int - Returns zero on success

**10.8.3 Function:** mpdc\_dla\_stop\_server

**Purpose:** Stop the DLA server

**Returns:** void

**10.9 MAS**

**10.9.1 Function:** mpdc\_mas\_pause\_server

**Purpose:** Pause the MAS server

**Returns:** void

**10.9.2 Function:** mpdc\_mas\_start\_server

**Purpose:** Start the MAS server

**Returns:** int - Returns zero on success

**10.9.3 Function:** mpdc\_mas\_stop\_server

**Purpose:** Stop the MAS server

**Returns:** void

**10.10 RDS**

**10.10.1 Function:** mpdc\_rds\_pause\_server

**Purpose:** Pause the RDS server

**Returns:** void

**10.10.2 Function:** mpdc\_rds\_start\_server

**Purpose:** Start the RDS server

**Returns:** int - Returns zero on success

**10.10.3 Function:** mpdc\_dla\_stop\_server

**Purpose:** Stop the DLA server

**Returns:** void