Rich Communication Suite – End-to-End Encryption Specification

Version 1.0

28 February 2025

Table of Contents

1 Introduction

1.1 Overview

End-to-end encryption (E2EE) refers to a generic private communication system in which only the communicating users can participate. As such, no one else, including the communication systems provider, telecom providers, internet providers, or malicious actors can access the cryptographic keys needed to communicate. Functionally, this assures that messages exchanged using E2EE cannot be read or secretly modified by anyone other than the intended senders and recipients. The precise definition of E2EE can be found in [IACR 2085]. Any implementation that deviates from the definition is not considered an E2EE system.

RCS will rely on Messaging Layer Security (MLS) Protocol, which is an IETF specification [RFC9420], for supporting end-to-end encryption. MLS is a formally verified standard that guarantees both forward secrecy and post-compromise security for messaging in 1-to-1 and group conversations. It is designed to scale efficiently with large group chats, and it supports post-quantum encryption.

E2EE is meant to run on top of RCS networks and clients. While cryptographic material will be exchanged independent of the RCS system, the encrypted messages and group chats are transmitted and stored via existing mechanisms as specified in [GSMA PRD-RCC.07].

1.2 Scope

This document defines how to incorporate MLS into RCS (Rich Communication Suite) and ensure that RCS users can securely exchange messages with one another in both 1-to-1 and group contexts. Note that E2EE only applies to P2P conversations and not Chatbot conversations.

1.3 Definition of Terms

Term	Description
Active Conversation	A Conversation with at least one User Message in the last 30 days.
Additional Authenticated Data	Information that is not encrypted but is bound to the ciphertext in a cryptographic way, so that a modification of the information renders the ciphertext invalid.
Application Message	Message defined by the [GSMA PRD-RCC.07] unrelated to encryption.
Certificate Chain	An ordered list of certificates, from the root certificate to the leaf certificate. Each certificate is signed by its parent.
Cipher Suite	As defined in [RFC9420]. A Cipher Suite is a combination of a protocol version and the set of cryptographic algorithms that should be used.
Client	A logical device owned by a Participant; an agent that uses this protocol to establish a shared cryptographic state with other Clients. A Client is defined by the cryptographic keys it holds.

Term	Description
Client Certificate	An X.509 representation of the Client. The X.509 certificate is verified by the KDS. Asserts the Client & Participant’s identity in a given time window.
Client Credential	As defined in [RFC9420]. Each member of a group presents a credential that provides one identity for the member and associates them with the member's signing key. The identities and signing key are verified by the KDS in use. In RCS, the Client Credential contains a Client Certificate.
Commit	As defined in [RFC9420]. A message that implements the changes to the group proposed in a set of Proposals.
Control Message	Message defined by the MLS specification (e.g. Commit, Welcome)
Cryptographic State	The set of keys and other MLS state required to encrypt, decrypt and sign messages as well as create and verify Commits.
End-to-End Encryption (E2EE)	A private communication system in which only communicating users can participate.
Epoch	As defined in [RFC9420]. A state of a group in which a specific set of authenticated Clients hold shared cryptographic state.
Epoch Authenticator	A short cryptographic representation of the state of a given epoch.
Era	An identifier representing a version of the MLS Group within an RCS conversation. Each new Era creates a new MLS Group with the same Group Identifier.
External Commit	As defined in [RFC9420]. A commit that is issued by a non-member of the cryptographic group.
External Proposal	As defined in [RFC9420]. A Proposal that is sent by a non-member of the group, particularly by the server hosting the group.
Foreign KDS	A KDS that a client is communicating with through their Home KDS.
Group Context	As defined in [RFC9420]. An object that summarises the shared, public state of the group. The GroupContext is typically distributed in a signed GroupInfo message, which is provided to new members to help them join a group.
Group Context Extensions	As defined in [RFC9420]. Additional application-level entries in GroupContext object.
Home KDS	A KDS that a client is directly communicating with.
Home KDS Interface	Interface between the client and their Home KDS.
Inter-KDS Interface	Interface between two KDSes that facilitates federation.
Key Delivery Service (KDS)	A server, provided by the application vendor, responsible for associating, holding, and distributing a user’s KeyPackage.

Term	Description
KeyPackage	As defined in [RFC9420]. A signed object describing a Client's identity and capabilities, including a hybrid public key encryption (HPKE) [RFC9180] public key that can be used to encrypt to that Client. Other clients can use a Client's KeyPackage to introduce the Client to a new group.
Last Resort KeyPackage	As defined in [RFC9420]. A reusable (by multiple members) KeyPackage.
Leaf Node	As defined in [RFC9420]. Leaf Node of the MLS Ratchet Tree that describes all the details of an individual Client's appearance in the MLS Group, signed by that Client.
MLS Control Message	A CPIM message containing either a Commit or a list of Proposals.
MLS Group	Represents a logical collection of Clients that share a common secret value at any given time. Its state is represented as a linear sequence of epochs in which each epoch depends on its predecessor.
MLS Message	A public or private message carrying MLS primitives (PublicMessage or PrivateMessage).
MLS (Ratchet) Tree	Represents a current state of an encryption in a given conversation and is used to distribute encryption keys to group members.
Participant	An entity identified by an RCS primary identifier that logically represents a single end user.
Plaintext Message	A message that is transmitted unencrypted.
Private-IM	An RCS Message sent to a single Participant of the Group. In 1-to-1 messaging, every message is considered a Private-IM.
PrivateMessage	As defined in [RFC9420]. An MLS protocol message that is signed by its sender, authenticated as coming from a member of the group in a particular epoch, and encrypted so that it is confidential to the members of the group in that epoch.
Proposal	As defined in [RFC9420]. A message that proposes a change to the group, e.g., adding or removing a member.
PublicMessage	As defined in [RFC9420]. An MLS protocol message that is signed by its sender and authenticated as coming from a member of the group in a particular epoch but not encrypted.
RCS Conversation	The RCS representation of 1-to-1 or group conversation, which includes all the Participants in that conversation.
Signed Encryption Identity Proof	A signature returned by ACS that proves ownership of MSISDN as well as the Participant Key and binds them together.
UpdatePath	As defined in [RFC9420]. An MLS procedure to update nodes of the ratchet tree with new secrets.
User Message	A Message containing user content, such as text, files, audio, as opposed to Control Messages.

1.4 Abbreviations

Term	Description
AAD	Additional Authenticated Data
ABNF	Augmented Backus-Naur Form
ACS	Auto-Configuration Server = Configuration Server as defined in [GSMA PRD-RCC.14]
AES-CTR	Advanced Encryption Standard using Counter Mode
CA	Certificate Authority
CLR	Certificate Revocation List
CPIM	Common Presence and Instant Messaging
CSPRNG	Cryptographically Secure Pseudorandom Number Generator
DBA	Doing Business As
E2EE	End-to-end encryption
ECDSA	Elliptical Curve Digital Signature Algorithm
FTD	Fail to decrypt
GUID	Globally Unique Identifier
HKDF	Hash Key Derivation Function
HKI	Home KDS Interface
HMAC	Hash-Based Message Authentication Code
HPKE	Hybrid Public Key Encryption
IKI	Inter-KDS Interface
IMDN	Instant Message Disposition Notification
KDS	Key Delivery Service
MIME	Multipurpose Internet Mail Extension
MLS	Messaging Layer Security
mTLS	Mutual Transport Layer Security
MSRP	Message Session Relay Protocol
NNI	Network Interface
NS	Name Space
NTP	Network Time Protocol
OID	Object Identifier
P2P	Person to Person (communication)
PRD	Permanent Reference Document
RPC	Remote Procedure Call
RCS	Rich Communication Suite
RCS SPN	RCS Service Provider Network
SIP	Session Initiation Protocol
UNI	User Network Interface
URI	Uniform Resource Identifier

Term	Description
URN	Uniform Resource Names
XML	Extensible Markup Language

1.5 Document Cross-References

Ref	Document Number	Title
1	[gRPC]	gRPC Remote Procedure Calling https://grpc.io/
2	[GSMA PRD-RCC.07]	GSMA PRD RCC.07 Rich Communication Suite - Advanced Communications Services and Client Specification, Version 15.0, 28 February 2025 http://www.gsma.com/
3	[GSMA PRD-RCC.11]	GSMA PRD RCC.07 Rich Communication Suite Endorsement of OMA CPM 2.2 Conversation Functions, Version 13.0, 28 February 2025 http://www.gsma.com/
4	[GSMA PRD-RCC.14]	GSMA PRD RCC.14 HTTP-Based Service Provider Device Configuration, Version 11.0, 28 February 2025 http://www.gsma.com/
5	[GSMA PRD-RCC.71]	GSMA PRD RCC.71 RCS Universal Profile Service Description Document, Version 3.0, 28 February 2025 http://www.gsma.com/
6	[IACR 2085]	Definition of End-to-end Encryption https://eprint.iacr.org/2024/2085
7	[ITU-T X.680]	X.680 : Information technology – ASN.1 Specification of basic notations https://www.itu.int/rec/T-REC-X.680
8	[ITU-T X.690]	X.690 : Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER) https://www.itu.int/rec/T-REC-X.690
9	[NIST SP800-38A]	Recommendation for Block Cipher Modes of Operation https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38a.pdf
10	[PADME]	Reducing Metadata Leakage from Encrypted Files and Communication with PURBs https://www.petsymposium.org/2019/files/papers/issue4/popets-2019-0056.pdf
11	[RFC2119]	“Key words for use in RFCs to Indicate Requirement Levels”, S. Bradner, March 1997 http://www.ietf.org/rfc/rfc2119.txt
12	[RFC3862]	Common Presence and Instant Messaging (CPIM): Message Format https://www.rfc-editor.org/rfc/rfc3862.html
13	[RFC3966]	The tel URI for Telephone Numbers https://www.rfc-editor.org/rfc/rfc3966

Ref	Document Number	Title
14	[RFC4648]	The Base16, Base32, and Base64 Data Encodings https://www.rfc-editor.org/rfc/rfc4648.html
15	[RFC5280]	Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile https://datatracker.ietf.org/doc/html/rfc5280
16	[RFC5438]	Instant Message Disposition Notification (IMDN) https://www.rfc-editor.org/rfc/rfc5438.html
17	[RFC5869]	HMAC-based Extract-and-Expand Key Derivation Function (HKDF) https://www.rfc-editor.org/rfc/rfc5869.html
18	[RFC8120]	Mutual Authentication Protocol for HTTP https://www.rfc-editor.org/rfc/rfc8120.html
19	[RFC8174]	Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words https://www.rfc-editor.org/info/rfc8174
20	[RFC8446]	The Transport Layer Security (TLS) Protocol Version 1.3 https://www.rfc-editor.org/rfc/rfc8446.html
21	[RFC9180]	Hybrid Public Key Encryption https://www.rfc-editor.org/rfc/rfc9180.html
22	[RFC9420]	The Messaging Layer Security (MLS) Protocol https://www.rfc-editor.org/rfc/rfc9420.html
23	[RFC9505]	Network Time Protocol Version 4: Protocol and Algorithms Specification https://www.rfc-editor.org/rfc/rfc5905.html
24	[RFC9562]	Universally Unique IDentifiers (UUIDs) https://www.rfc-editor.org/rfc/rfc9562.html

1.6 Conventions

The key words “must”, “must not”, “required”, “shall”, “shall not”, “should”, “should not”, “recommended”, “may”, and “optional” in this document are to be interpreted as described in [RFC2119] and clarified by [RFC8174].

Throughout this document, Client (with a capital C) refers to the MLS term defined in section 1.3. Specifically, it is the logical representation of a device inside of the MLS Group. When the term client (with a small c) is used, it refers to the software that is following the procedures defined in this document, as it would be used in [GSMA PRD-RCC.07].

2 Architecture

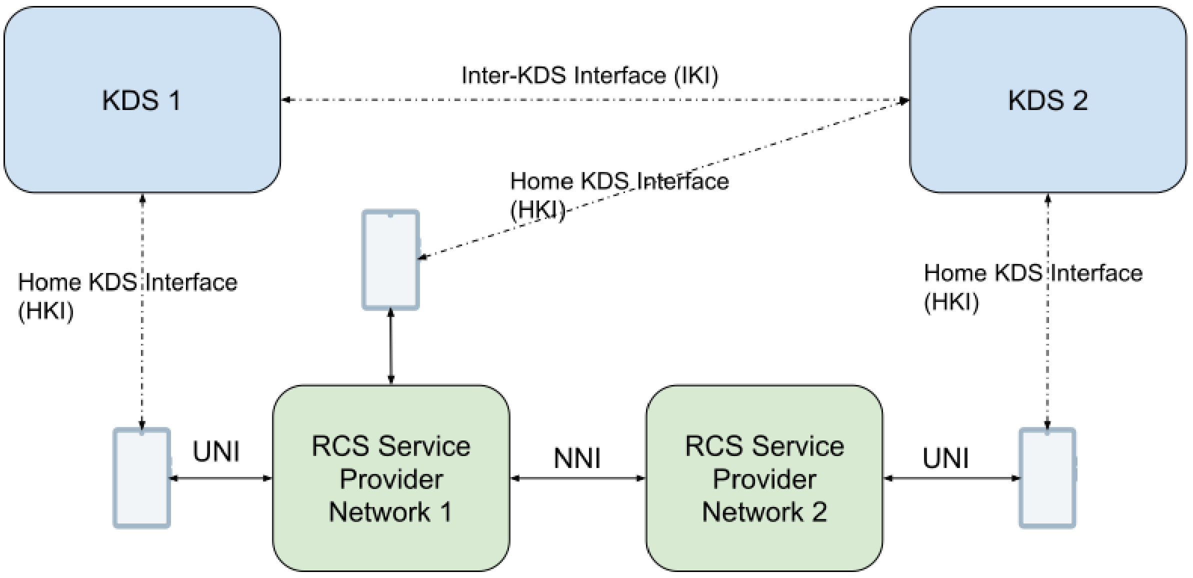

Figure 1: Overall Architecture

The E2EE system includes clients and Key Delivery Services (KDSes), which is the store for all KeyPackages uploaded by clients. The E2EE system lives alongside the RCS Service Provider Network (RCS SPN). KDSes do not directly communicate with the RCS SPN and the RCS SPN does not directly communicate with the KDSes. However, primitives generated from one entity shall be checked by the other.

E2EE is a client feature. Private or symmetric keys used for encryption/decryption are not stored on any server. Instead, those keys live only on the client. All RCS SPN operations in this document are meant to assist in improving the reliability of E2EE. Even without any server assistance, clients will be able to perform all E2EE operations.

To facilitate multiple vendors implementing E2EE, KDSes must be able to federate. Clients rely on the KDS assigned by their own client provider (called the Home KDS) to fetch all KeyPackages regardless of where they are stored. The Home KDS shall fetch KeyPackages from other KDSes (called Foreign KDSes).

The KDS is responsible for:

- Verifying the user identity and the Participant Keys of the client.
- Storing, certifying and delivering KeyPackages.
- Fetching KeyPackages from Foreign KDSes on request from served clients.

Within the context of E2EE, the client is responsible for:

- Generating KeyPackages.
- Encrypting, decrypting, and authenticating operations.
- Modifying the cryptographic state of the MLS Group (which includes changes in MLS Group membership).
- Verifying Commits sent by other clients and routed via the RCS SPN.

Within the context of E2EE, the RCS SPN is responsible for:

- User identity verification.
- Routing messages between clients.
- Arbitration and basic validation of Commits and messages.
- Storing and providing (to clients) the MLS GroupInfo and Epoch Authenticators (present and past) for all conversations (1-to-1 and group conversations).

Encryption will continue to operate over NNI. The state for each RCS Conversation will be stored in a single RCS SPN (called Conversation Focus) but can potentially be transferred to another RCS SPN.

3 Identity Model

3.1 Client and Participants

Participants are individual users, like Alice or Bob. Participants have one or more devices represented as Clients, like WebApp or phone. RCS Conversations (including both 1-to-1 messaging and group conversations) are associated with MLS Groups. An RCS Conversation is composed of the Participants, while the MLS Group is composed of the Clients corresponding to the Participants of the RCS Conversation. The Participants of an MLS Group is the set of all Participants whose Clients are in the MLS Group. The clients and the Conversation Focus shall ensure that all of the Participants of the MLS Group match the Participants of the RCS Conversation at all times.

NOTE: for the initial release of this specification, a Participant has only one Client.

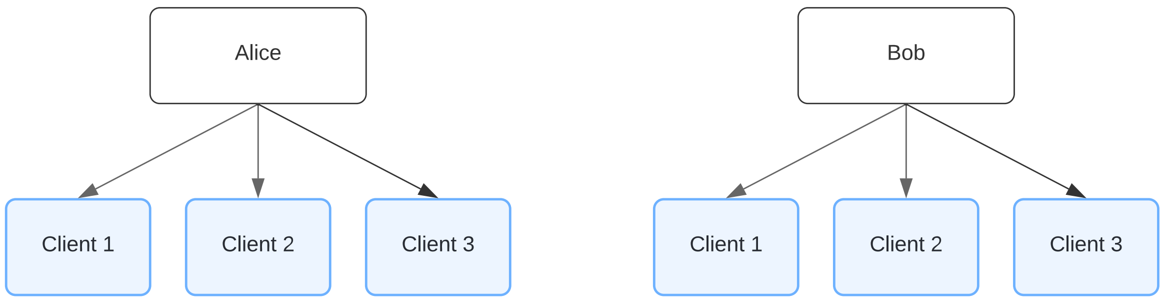

Figure 2: Relationship between Participants and Clients

The Participant is represented by the user identity and the Participant public key. The Client is represented by the Client Certificate (defined in Annex A.3)

An MLS Group is represented as a binary tree, in which Leaf Nodes contain Client Certificates to represent Clients (and transitively the Participants of the MLS Group).

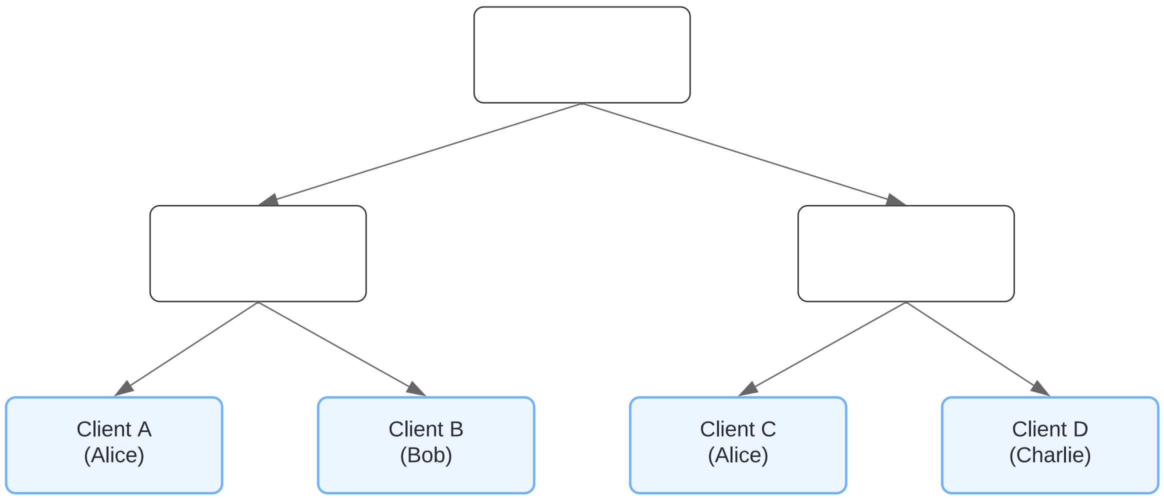

Figure 3: An MLS Group Ratchet Tree with Three Participants (Alice, Bob, Charlie); Alice Has Two Clients

3.2 Client Certificates

Client Certificates are issued by the KDS pursuant to section 4.2. Client Certificates are verified by the Conversation Focus (section A.4.3) and other clients at the time of any Commits, such as MLS Group membership changes. The Certificate Signing Model (section 3.3) describes how the certificates are authenticated. The Client Certificate shall follow procedures in Annex A.3.

3.3 Certificate Signing Model

The Certificate Chain is used to establish trust, prevent man-in-the-middle attacks, and prevent identity spoofing. The Root Certificates of application vendors must meet the requirements described in Annex A.1. Those vendors are responsible for issuing Client Certificates. For MLS in RCS to work, all interconnected Client vendors shall agree on the same set of trusted Root Certificates.

Additional intermediate certificates can be added between the Root and Leaf certificates. There has to be at least one, but could include multiple intermediates.

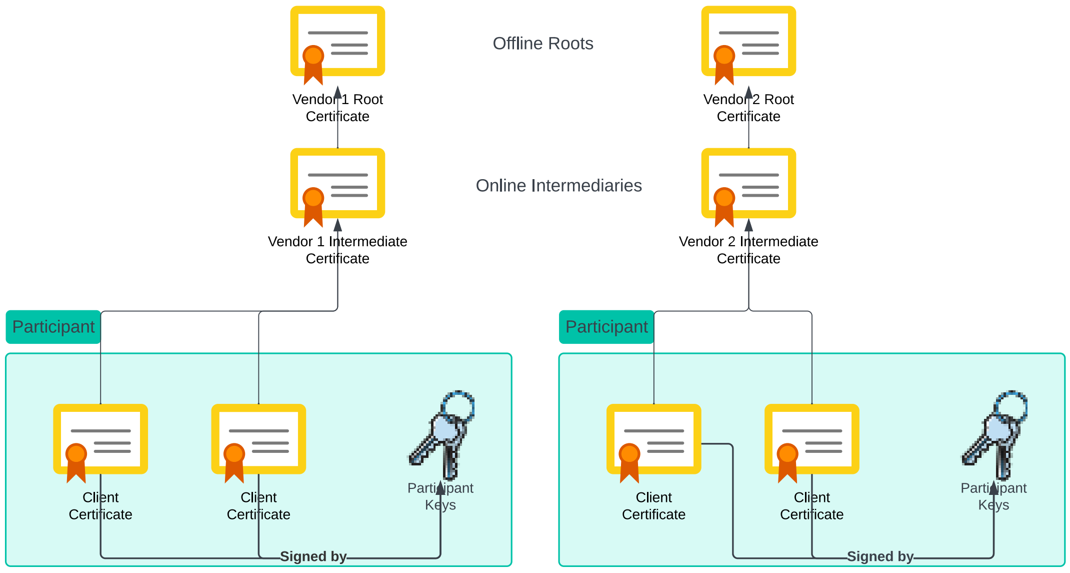

Figure 4: The Certificate Hierarchy

Each Client Certificate contains an assertion from the ACS. The ACS provides an extra guarantee of MSISDN ownership on top of the guarantees provided by the KDS. Assertions from both the ACS and KDS regarding phone number ownership must agree. Details are described in section 6.2.2 and Annex A.3.

Client Certificates are non-revocable. Their lifetime is defined in Annex A.3. Intermediate CAs are revocable using CRLs defined in the Offline Root.

4 Provisioning

The overall flow of provisioning and enrolling with Home KDS is outlined in the following figure.

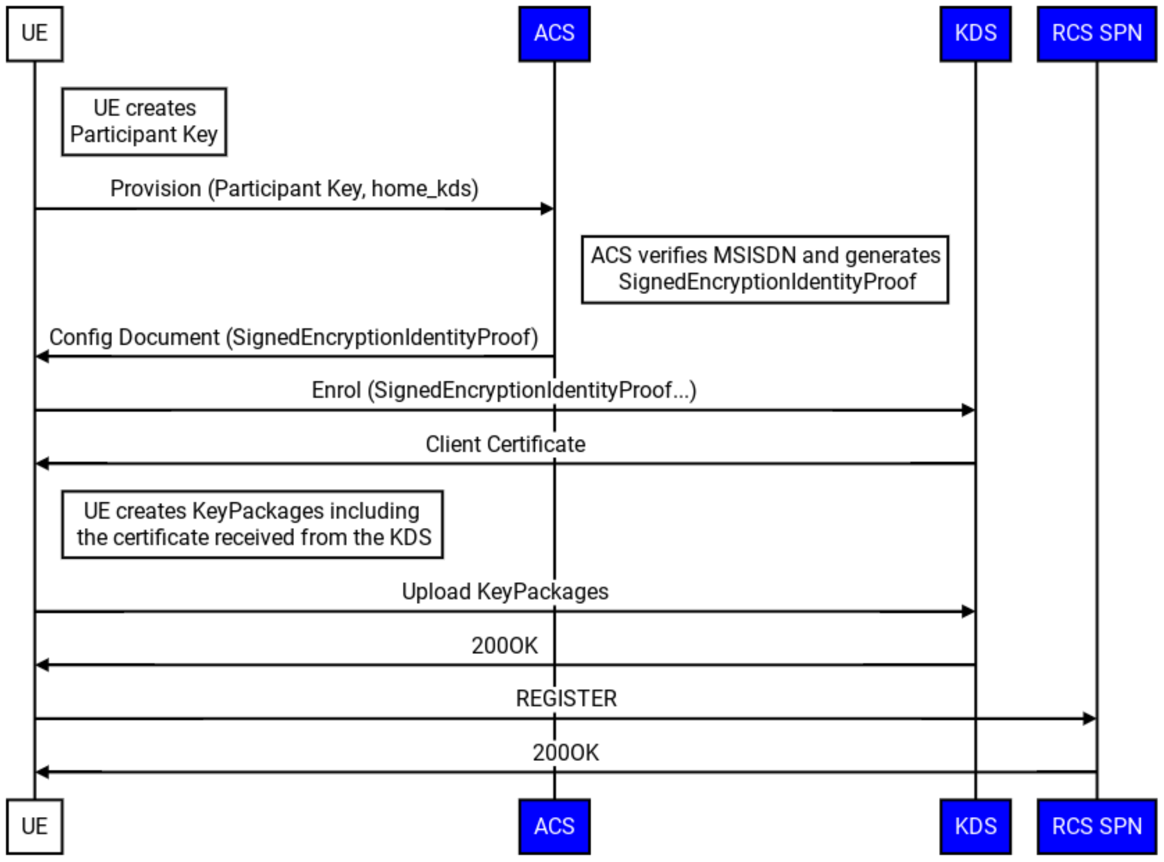
UE
ACS
KDS
RCS SPN
UE creates Participant Key
Provision (Participant Key, home_kds)
ACS verifies MSISDN and generates SignedEncryptionIdentityProof
Config Document (SignedEncryptionIdentityProof)
Enrol (SignedEncryptionIdentityProof...)
Client Certificate
UE creates KeyPackages including the certificate received from the KDS
Upload KeyPackages
200OK
REGISTER
200OK

Figure 5: Provisioning and Enrolment in Home KDS

4.1 ACS Signed Encryption Identity Proof

When provisioning or processing a configuration refresh of the client with ACS, as per [GSMA PRD-RCC.07], the client shall include the following:

- The Public Participant Key as defined in section 3.1
- The `home_kds`, an integer indicating the ID of the KDS vendor

Upon verifying the client, ACS shall sign the tuple of (MSISDN, Public Participant Key, `home_kds`), including an expiry as defined in section 7.12 and return it in the configuration document as per [GSMA PRD-RCC.07]. If ACS does not return the Signed Encryption Identity Proof, the client shall continue with enrolment with the Home KDS.

The ACS shall store the Public Participant Key and the `home_kds` values. Those values shall be used to construct the ACS Signed Encryption Identity Proof in future config refreshes.

4.2 Enrolment with KDS

After the provisioning with ACS, the client shall enrol with its Home KDS and upload KeyPackages as per section 5.1. If ACS returned the Signed Encryption Identity Proof as specified in section 4.1, the client shall include it in this enrolment request, to be included in the Client Certificate.

After completing the enrolment with Home KDS and uploading of KeyPackages, the client shall register to the RCS SPN as per section 2.4 in [GSMA PRD-RCC.07]. The client shall also re-add themselves via an External Commit to all MLS Groups they are aware of.

5 Key Delivery Service (KDS)

5.1 Uploading KeyPackages

After receiving the ACS Signed Encryption Identity Proof of (MSISDN, Public Participant Key, home_kds) during ACS provisioning as per section 4.1, the client shall generate HPKE key pairs (private and public keys) as per [RFC9420]. The client shall then upload the required fields to issue certificates as per section A.3 (including Cipher Suite and the ACS Signed Encryption Identity Proof) during enrolment with the Home KDS. The exact interface between the client and the KDS is vendor specific.

NOTE: The interface between client and Home KDS may be specified in future releases.

Upon client enrolment, the Home KDS shall verify the end user identity. Upon successful verification, the Home KDS shall use a globally unique Client ID and issue Client Certificates, adhering to the certificate format outlined in Annex A.3, for each Client Key. The Home KDS shall then return these Client Certificates to the client.

NOTE: The Home KDS can choose to do its own verification of the MSISDN or it can use the ACS Signed Encryption Identity Proof as a verification mechanism.

With the Client Certificate, the client shall create KeyPackages as per [RFC9420] and upload them to the Home KDS. The lifetime of the KeyPackage must match the lifetime of the Client Certificate issued by the Home KDS. The Home KDS shall store the KeyPackages to be fetched by other clients (directly via Home KDS or indirectly via Foreign KDS).

If the initial KeyPackage upload fails for any reason (including retries), the client shall not advertise MLS capabilities as described in section 5.2.

The KDSes and RCS SPN must maintain accurate clock by utilising Network Time Protocol (NTP). Clients should also maintain accurate clocks via NTP to validate certificate times.

5.2 MLS Capabilities

To check if MLS is enabled for a recipient client, the sender client shall request their RCS capabilities through the sender’s RCS SPN that will query the recipient’s RCS SPN. Clients that support MLS shall include Encryption version (indicating which version of this specification the client supports) and Encryption Home KDS capability (indicating the recipient’s Home KDS ID) as specified in [GSMA PRD-RCC.07].

5.3 Fetching Key Packages

To fetch KeyPackages, the client shall request them from their Home KDS. The client shall list, in order they prefer, the Cipher Suites for encryption, and a list of the recipients (with their `home_kds` capability value) to fetch KeyPackages for.

Upon receiving a KeyPackage request, the Home KDS shall first query the Cipher Suites for all Participants, and if necessary, federate with other Foreign KDSes. Based on the order listed in the request, the Home KDS chooses a common Cipher Suite. For example, if the client listed Cipher Suites a then b then c, some of the Participants were capable of a, and all were capable of b and c, then the Home KDS shall pick Cipher Suite b as the common Cipher Suite. If there are no common Cipher Suites, the Home KDS shall return no common Cipher Suite for all the Participants. The client shall create the group unencrypted.

If the KeyPackages are being fetched for adding users to a group, the client shall request KeyPackages for the Cipher Suite of the MLS Group. If there are Participants who do not support the Cipher Suite, the Home KDS shall return no common Cipher Suite. The client may request KeyPackages for all Participants in the group (existing and added) to achieve a common Cipher Suite. If a common Cipher Suite is found, the client shall advance the Era (as per section 8.3) with the common Cipher Suite.

The Home KDS shall then fetch the corresponding KeyPackages for the selected Cipher Suite from its local store or, if necessary, by federating to other Foreign KDSes as per section 5.4. The Home KDS shall validate the Client Credentials in the KeyPackages according to Annex A.4.2. The Home KDS shall then return the KeyPackages for all recipients.

The Home KDS shall return KeyPackages for all the Clients of the Participant. Clients that add a Participant to an RCS Conversation must include all Clients in the MLS Group for that Participant. For example, if a Participant has 3 Clients, the Home KDS shall return KeyPackages for all 3 Clients. The client shall then include all 3 KeyPackages in the MLS Group.

If unexpected errors arise during the KeyPackage fetching that are not recoverable by retrying, the client shall create the RCS Conversation unencrypted.

NOTE: The Home KDS should return results to indicate the specific response status for each Participant.

The exact interface between clients and their Home KDS is vendor specific.

5.4 KDS Federation

When the Home KDS receives a request for a Participant's KeyPackages that has a `home_kds` parameter other than its own, it shall request the KeyPackages from the Foreign KDS indicated by the `home_kds`.

First, the Home KDS shall request the Cipher Suites of the foreign Participants. Then, after calculating the Cipher Suite as per section 5.3, the Home KDS then shall request KeyPackages for that Cipher Suite from the Foreign KDS.

When a new (i.e. previously unknown) Participant enrols on a Home KDS, the Home KDS should inform all Foreign KDSes of the new enrolment using the `NotifyParticipantRegistration` RPC. Upon receiving this notification, a Foreign KDS should remove the Participant from its storage and should inform the previous device about the transfer of the Participant to the new device.

If the notification's enrolment timestamp for a Participant is before the Foreign KDS's enrolment time for the same Participant, the Foreign KDS may ignore the notification and inform the Home KDS that it ignored the notification via a `NEWER_ENROLMENT_EXISTS` error.

The Inter-KDS Interface (IKI) shall use gRPC [gRPC]. Besides the authentication supplied in gRPC, the federated KDSes shall authenticate with mTLS. Federated KDSes shall use native gRPC errors for indicating overall RPC error. In case there are partial errors, the Home KDS shall return those partial errors as specified in the interface definition.

In case the Foreign KDS cannot be reached, the Home KDS shall not return any KeyPackages for any Participant. The Home KDS shall return a "currently unavailable" error for all Participants that belong to the unreachable Foreign KDS and a success return status for other Participants. The client may retry the request or may create the RCS Conversation unencrypted.

The IKI is defined in Annex B.

5.4.1 Error Handling

When receiving a response from a Foreign KDS, each Participant must contain a ResponseStatus that indicates the success of each operation. They are:

- `UNKNOWN_STATUS`: this error shall not be used.
- `OK`: The query has succeeded and the value is returned.
- `NOT_FOUND`: The Participant queried was not found in the Foreign KDS. The Home KDS and client should consider the Participant unable to encrypt. The client may retry capabilities to update the `home_kds` of the Participant and retry the query.
- `MALFORMED_ID`: The E.164 of the phone number was not properly formatted. The number must be properly formatted before retyring the query.
- `UNSUPPORTED_CIPHER_SUITE`: The Participant does not have a KeyPackage in the requested Cipher Suite. The Home KDS must choose a different Cipher Suite or consider the Participant not capable of E2EE.
- `NEWER_ENROLMENT_EXISTS`: The Participant enrolled on the Foreign KDS with a newer timestamp than the one in the request.

5.5 Identity Verification

The clients shall allow users to verify the identity of the users they are communicating with. The identity verification happens between a pair of users. The identity verification code between the pair of users applies to all conversations they are part of.

To verify user identity for another user, the client shall:

- Calculate the identity verification code as per Annex C.5.
- Display the 80 digits to the user to allow the pair of users to compare them.

The clients may use a QR code to make the comparison easier. If done so, the client shall:

- Calculate the identity verification code as per Annex C.5.

- Convert the 80 digits to binary representation.
- Embed the binary representation in a QR code.
- Display the QR Code to the user to allow comparison.

6 MLS Conversation Representation

6.1 Conversation Management

6.1.1 MLS Group Life Cycle

Each MLS Group is identified by a unique identifier, the MLS Group Id. For 1-to-1 chats, the MLS Group Id will be a random GUID. For group chats, the MLS Group Id shall be the Conversation Id.

Under certain error conditions, the cryptographic state for an MLS Group may need to be recreated from scratch. This process preserves the MLS Group Id but changes the associated Era Identifier. This numeric identifier shall sequentially increase over the lifetime of the group whenever the Era is advanced.

Within an individual Era, Participants in an MLS Group may modify the group's state by sending Commit messages. These special MLS protocol messages contain an intent to update some aspect of the group. When clients receive a Commit message, they shall update their local MLS Group state to reflect the change.

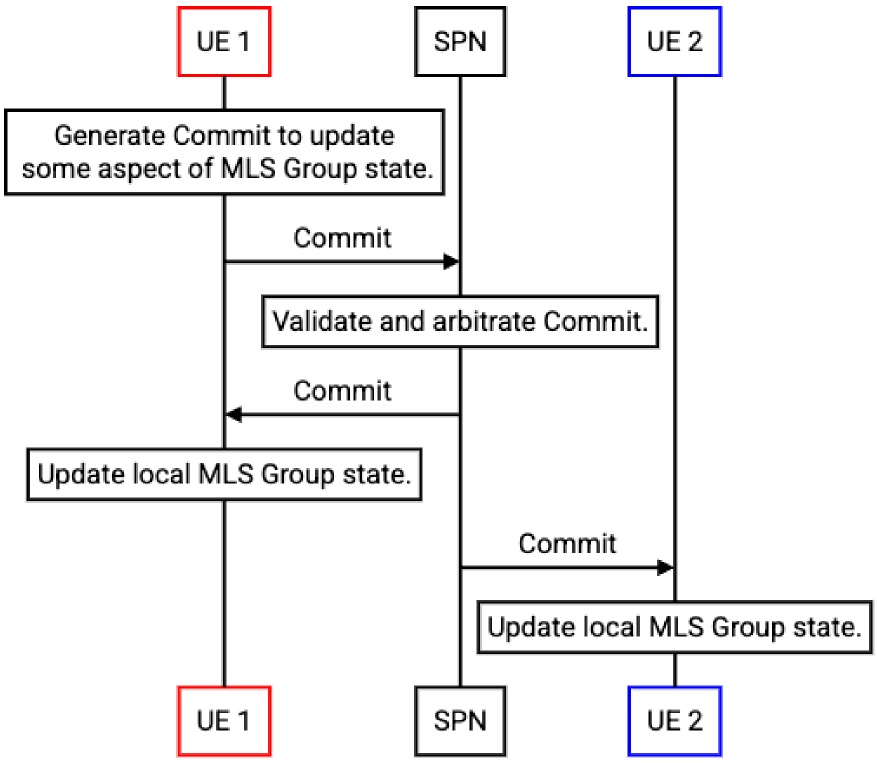
UE 1
SPN
UE 2
Generate Commit to update some aspect of MLS Group state.
Commit
Validate and arbitrate Commit.
Commit
Update local MLS Group state.
Commit
Update local MLS Group state.
UE 1
SPN
UE 2

Figure 6: Using Commit Messages to Update the Local MLS Group State

Clients may also send Proposal messages to update the MLS Group's state. Like Commits, Proposal messages also communicate an intended change to the MLS Group, but in order for that change to be applied, the Proposal must be used to build a new Commit message that contains it.

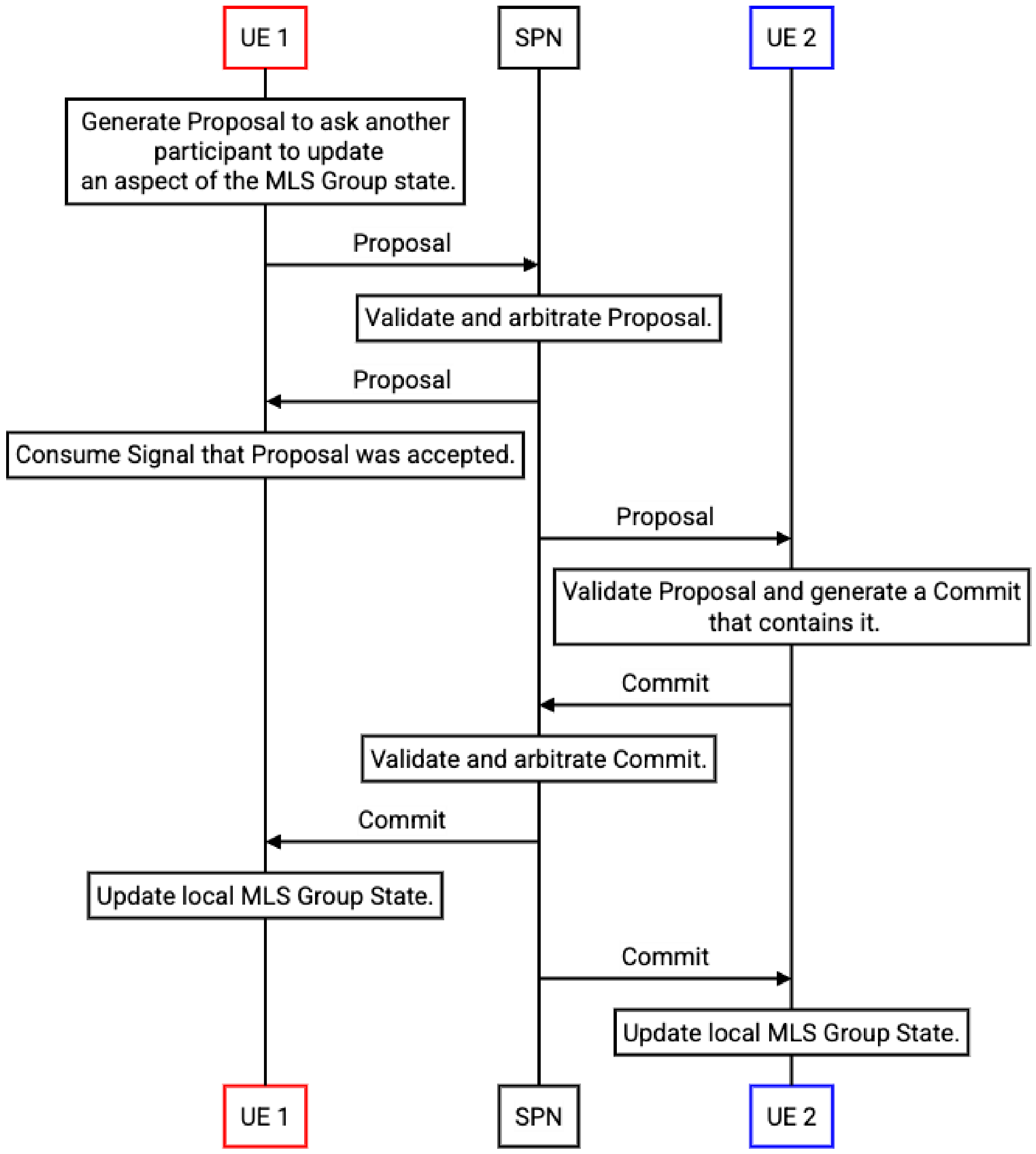
UE 1
SPN
UE 2
Generate Proposal to ask another participant to update an aspect of the MLS Group state.
Proposal
Validate and arbitrate Proposal.
Proposal
Consume Signal that Proposal was accepted.
Proposal
Validate Proposal and generate a Commit that contains it.
Commit
Validate and arbitrate Commit.
Commit
Update local MLS Group State.
Commit
Update local MLS Group State.
UE 1
SPN
UE 2

Figure 7: Using Proposal Messages to Update the Local MLS Group State

Commit messages contain an Epoch Identifier that indicates which state, or "Epoch", the MLS Group is transitioning from. Epochs shall advance sequentially, and must be applied atomically. Thus, if an MLS Group is currently in Epoch n, the only Commit that can be applied is one that advances the group to Epoch n+1, and such a Commit shall only be applied once. The very first Epoch in Era for an MLS Group is 0, and the initial Commit to create the RCS Conversation and add the members moves it to Epoch 1.

When receiving a new Commit, clients shall keep the secrets of the previous Epoch for decrypting messages for at least 3 days to ensure any messages that arrive for previous Epochs can be decrypted.

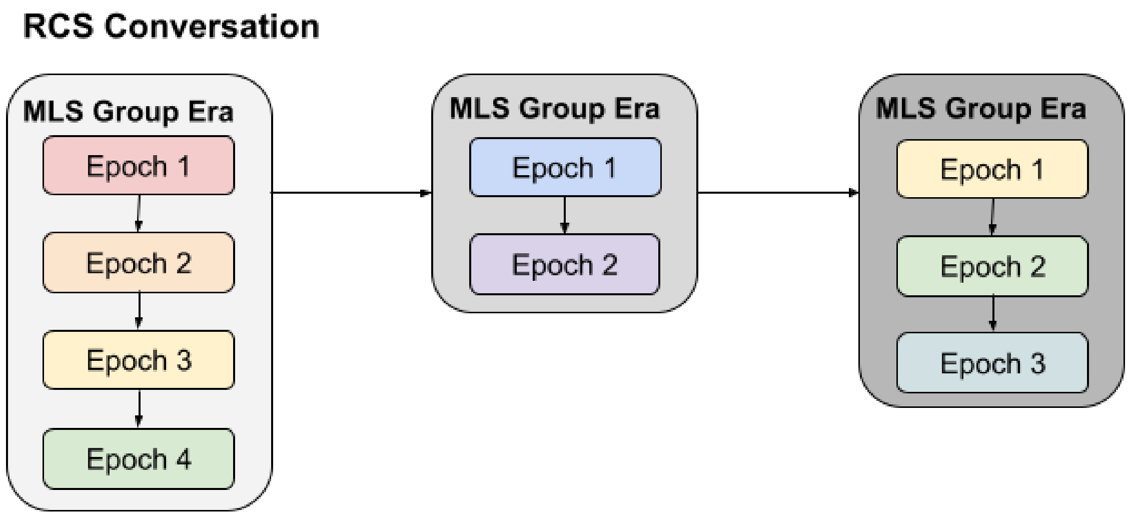

Figure 8: Progression of Epochs Within MLS Group Eras

To ensure that all Clients in an MLS Group have the same view of the Group, Messaging Servers shall arbitrate all incoming Proposal and Commit messages.

6.1.2 Conversation Focus

RCS Conversations using MLS shall be managed by one Messaging Server within the lifetime of a single Era. Between Eras the ownership may change.

The Messaging Server that processes the MLS Group Creation request for an Era shall be the Conversation Focus for the group in that Era.

The Conversation Focus must handle all state management for the Conversation. This includes:

- Arbitrating and validating Commit and Proposal messages.
- Managing and delivering MLS Group state.
- Ensuring that MLS Group state and RCS Conversation state remain in sync only when the `end_mls` tag is not present in the MLS Group.
- Storing and forwarding MLS Control and Application Messages to their intended destinations.
- Deleting the MLS Group after a configurable time, recommended to be 30 days of inactivity.

Messaging Servers that receive requests destined to an MLS Group that is managed by a different Messaging Server shall forward the requests to that server. Similarly, if a Messaging Server receives requests that were sent from an MLS Group managed by a different server, it shall forward the requests to its own users.

6.1.3 MLS-Opaque-Token Definition

The `MLS-Opaque-Token` SIP header allows Messaging Servers to embed MLS information (especially to identify the Conversation Focus) in the requests that they send to clients or to other Messaging Servers.

When an RCS Conversation is first created, Messaging Servers may communicate an `MLS-Opaque-Token` to both the conversation creator and all participants. If the `MLS-Opaque-Token` is present, clients must then include this MLS-Opaque-Token in any subsequent SIP requests that they generate for that conversation.

If clients receive a new `MLS-Opaque-Token` in SIP requests sent within the same RCS Conversation, then any new requests they send must include the updated MLS-Opaque-Token.

When the participants of an RCS Conversation span multiple RCS SPNs, the Participating Functions must use the `MLS-Opaque-Token` if supplied by the Conversation Focus in all SIP requests directed to that Conversation Focus.

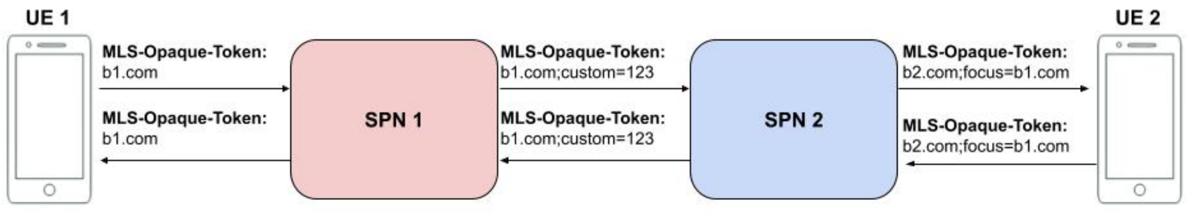
UE 1
MLS-Opaque-Token: b1.com
MLS-Opaque-Token: b1.com
SPN 1
MLS-Opaque-Token: b1.com;custom=123
MLS-Opaque-Token: b1.com;custom=123
SPN 2
MLS-Opaque-Token: b2.com;focus=b1.com
MLS-Opaque-Token: b2.com;focus=b1.com
UE 2

Figure 9: Use of MLS-Opaque-Token in RCS Conversations with Multiple RCS SPNs

6.1.4 Resolving RCS and MLS Identifiers

All RCS Conversations using MLS will continue to be addressed using their primary RCS identifiers. Specifically, requests sent in RCS Conversations using MLS will continue to be addressed to the same Request-URIs as specified in [GSMA PRD-RCC.07].

6.2 MLS Group Commit and Proposal Management

6.2.1 Commit and Proposal Arbitration

To ensure that all Participants have the same view of the MLS Group, MLS Conversation Focuses for a given MLS Group must validate, arbitrate, and order Commit and Proposal messages.

For a given MLS Group in a given Era, the Conversation Focus shall:

1. Verify that Commits adhere to the rules of Section 6.2.2.
2. Verify that Proposals adhere to the rules of Section 6.2.3.
3. Verify that newly created MLS Groups start at Epoch 0.
4. Accept exactly one Commit in each Epoch.
5. Accept only Proposals from a single user in each Epoch.

6. If any Proposals have been accepted in an Epoch, only accept Commit messages that contains all of the Proposals for that Epoch.
7. Only accept Commits and Proposals that advance the Epoch by exactly one.
8. Deliver accepted Commits and Proposals, as described in Section 6.2.4.
9. Explicitly inform the Client when Commits or Proposals are rejected to allow for optional resubmission as described in Section 6.2.5.
10. Store the MLS GroupInfo derived from all accepted Commits as described in Section 6.3.

6.2.2 Conversation Focus Commit Validation

To ensure that all Commits are valid, and to ensure that MLS Group state stays in sync with RCS Conversation state, the Conversation Focus shall verify, upon receiving a Commit message:

1. The Credential is Valid using the procedures defined in [RFC9420].
2. The Commit is Valid using the procedures defined in [RFC9420].
3. That if the MLS Context Extension in the stored MLS GroupInfo includes an `end_mls` tag as defined in section 11.2, the Commit is an initial Commit or a Commit that contains a removal of the `end_mls` GroupContext Extension as defined in section 7.11.2.2.
4. The Commit message is parsable according to the rules of [RFC9420].
5. The GroupInfo is parsable according to the rules of [RFC9420].
6. That if a new Credential is being introduced to the MLS Group, the certificates in the Credential are not expired.
7. That if the Era is being advanced for a Group whose Conversation Focus matches that of the previous Era, the membership of the MLS Group in the new Era matches the membership of the existing RCS Conversation.
8. That if a Participant is being added to the RCS Conversation, and the `end_mls` tag is not present in the MLS Group, at least one Client of the Participant is added to the corresponding MLS Group.
9. That if a Client is being added to the MLS Group, the identity of the Client's corresponding Participant must also be present in the associated RCS Conversation, or they must be added in the request that contained the Commit.
10. That if the final Client of a Participant is being removed from the MLS Group, the Participant must not be present in the associated RCS Conversation, or the Participant must be removed in the request that contained the Commit.
11. That if a Participant is being removed from an RCS Conversation, and the `end_mls` tag is not present in the MLS Group, all of the Clients of that Participant are also removed from the MLS Group in the Commit or Proposals contained within that request.
12. The Signatures in the Commit and GroupInfo are correct according to [RFC9420].
13. The Commit has the proper structure, given the group's Ratchet Tree and the position of the committer in the Tree.
14. The new LeafNode in the Commit is valid, and the contained Certificate is not expired.
15. The GroupContext includes all of the required GroupContext Extensions as described in Section 7.11.1.1.
16. The Commit's PublicMessage and the GroupInfo have the same `confirmation_tag`.
17. All Certificates in the MLS Group are valid for at least 30 days.

18. Each Participant in the MLS Group has exactly one Participant Key. That is, all Clients for the same Participant must share the same Participant Key.
19. That if the Commit is an External one, the new Client must belong to a Participant that is already in the MLS Group.
20. All Client Certificates in the MLS Group are valid according to the validation rules in Annex A.4.3.

6.2.3 Conversation Focus Proposal Validation

To ensure that all Proposals are valid, and that Clients will be able to generate a Commit that contains them, the Conversation Focus shall verify that:

1. The Proposal is parsable and valid according to the rules of [RFC9420].
2. The Proposal is either a Remove or Update Proposal.

For Remove Proposals, the Conversation Focus shall verify that:

1. The Proposals in the Epoch include all of the Clients for the Participant being removed.
2. All (and only) the Clients of the Participant sending the Remove Proposal are being removed. No Clients of any other Participant are to be removed.

For Update Proposals, the Conversation Focus shall verify:

1. The Leaf Node in the Proposal according to the rules of [RFC9420].
2. That the new Leaf Node’s Credential matches the old Leaf’s Credential (Certificate, Subject, Public Key).
3. That the `signature_key` matches the Credential Key contained within the Leaf Node.

6.2.4 Commit/Proposal Message Delivery

If a Conversation Focus accepts a Commit or Proposal message within an RCS Conversation, then it shall deliver the contents of the message to each Participant within the RCS Conversation, including the sender.

6.2.5 Signal for Rejected Commits/Proposals/Messages

If the Conversation Focus for an RCS Conversation rejects a Commit or Proposal, then it must explicitly inform the sender that the Commit, Proposal, or Message was rejected. In addition, if the Commit or Proposal was included as part of an operation that affected the RCS Conversation state (e.g. adding a user to an RCS group or kicking a user from an RCS group), then that operation must not be applied.

To achieve this, the Conversation Focus shall send a Negative-Delivery IMDN notification to the sender of the rejected Commit/Proposal message. This notification must include the <mls-server-failure-reason> extension within the <delivery-notification> to communicate the reason for rejection.

The `<mls-server-failure-reason>` (as defined in section 7.7.2.1) shall contain:

- The latest Era and Epoch for the associated MLS Group.

- The strategy that the Client should apply in order to re-attempt sending the Commit. This includes:
  - `<incorrect-era>`: The Client attempted to send a Commit in an Era that did not match the latest one on the Conversation Focus. The Client must wait to receive the latest MLS Group state before re-attempting.
  - `<incorrect-epoch>`: The Client attempted to send a Commit in the latest Era, but the Epoch did not match the latest one on the Conversation Focus. The Client must wait to receive the latest MLS Group state before re-attempting.
  - `<incorrect-epoch-authenticator>`: The Commit or Message contained an invalid `epoch_authenticator` which did not match the `epoch_authenticator` persisted by the Conversation Focus. The Client should attempt self-healing as per section 10.1, and should otherwise advance the Era as per section 8.3.
  - `<expired-credential>`: The Client attempted to send a Commit that included a new LeafNode with a Certificate that was expired, or the client attempted to send a Commit/Proposal/Message when an Expired Credential existed in the MLS Group. The Client must request a fresh KeyPackage of the Clients with expired Credential and Commit the new leaf update before re-attempting, as per section 9.5.4.
  - `<mismatched-rcs-group-state>`: The Client attempted to send a Commit for which the change to the MLS Group state was not reflected in the associated change to the RCS Conversation state. The Client may recreate and send the Commit after correcting this discrepancy.
  - `<unparsable-commit>`: The Client attempted to send a Commit that was unparsable. The Client shall not resend this Commit but may recover by recreating it and resending.
  - `<mismatched-confirmation-tag>`: The Client attempted to send a Commit whose `confirmation_tag` did not match the `confirmation_tag` of the persisted MLS GroupInfo. The Client should attempt self-healing as per Section 10.1 and should otherwise advance the Era as per section 8.3.
  - `<pending-proposal>`: The Client attempted to send a Commit that did not include the pending Proposal. The Client shall include the Proposal in the Commit and retry the operation.
  - `<transient-error>`: The Client attempted to send a Commit whose processing failed transiently on the RCS SPN. They may retry the creation and sending of the Commit.
  - `<encryption-not-available>`: The recipient is not capable of encryption and thus cannot receive encrypted messages. The sender can either move the conversation to unencrypted, or remove the recipient.
  - `<invalid-commit>`: The Commit failed to be validated. The client shall fix validation errors and try again.

6.2.6 Client Commit and Proposal Validation

When Clients receive a message containing a Commit or Proposal, they shall:

1. Validate the Commit as per section 6.2.2 or the Proposal as per section 6.2.3.

2. Validate that any Commits (other than External Commits) were sent by a Client of the MLS Group.
3. Validate that any Proposals were sent by a member of the MLS Group.
4. Follow the Credential Validation procedures defined in [RFC9420] and Annex A.4.1.
5. Follow the Commit Validation procedures defined in [RFC9420].

If the Client receives a Commit or Proposal message that fails validation, the Client shall attempt to initiate a Self-Heal as per section 10.1. If the validation failure persists, they must generate a Negative-Delivery IMDN notification and send it to the originator of the rejected message. Then the Client should recreate the MLS Group and and advance the Era as per section 8.3.

The negative delivery IMDN should encode a <mls-client-failure-reason> within the <delivery-notification>.

This `<mls-client-failure-reason>` (as per section 7.7.2.2) shall contain:

- The latest Era and Epoch of the MLS Group from the perspective of the Client.
- A signature that asserts that the sender of the failed IMDN is actually a member of the MLS Group.
- An indicator of why the message was rejected. This will include:
    - `<message-from-non-member>`: The message was sent from a user that wasn't a member of the underlying MLS Group.
    - `<invalid-credential>`: The Commit contained an invalid Credential. The client shall fetch a new KeyPackage for the Participant(s) and replace their Leaf Node(s) as per section 9.5.4.
    - `<invalid-commit>`: The Commit failed to be validated. The client shall fix validation errors and try again.
    - `<failed-to-decrypt>`: The recipient failed to decrypt the message. The sender shall advance to the latest epoch and resend the message.
    - `<commit-in-privatemessage>`: The sender included a commit in a Private Message instead of a Public Message. The sender must resend the Commit in a Public Message and send it again.

6.3 MLS GroupInfo Management

On behalf of each RCS Conversation, the Conversation Focus shall store the latest MLS GroupInfo associated with the conversation.

For the latest Era for a given RCS Conversation, the Conversation Focus shall store the `epoch_authenticator` for each Epoch that the MLS Group has advanced to.

Whenever a Conversation Focus accepts a Commit, it shall store the MLS GroupInfo and `epoch_authenticator` included with the Commit.

The persisted GroupInfo shall be exposed to members of the RCS Conversation for retrieval as per 6.3.1.

The persisted `epoch_authenticator` shall be used by the Conversation Focus to ensure that Clients sending PrivateMessages exchanged in the Conversation have persisted a valid GroupInfo locally.

6.3.1 Retrieving MLS GroupInfo

Conversation Focus shall allow Clients to retrieve the MLS GroupInfo associated with their RCS Conversations on demand.

To request the MLS GroupInfo associated with an RCS Conversation, Clients shall send a SIP INFO for the “MLS-Group-Info-Pull” Info-Package in the INVITE dialog associated with the RCS Conversation that they wish to request the state for. If they do not have an active INVITE session for the RCS Conversation when they wish to retrieve this state, they shall first create one.

Upon receiving this INFO request, the Conversation Focus shall:

- Generate a 469 (Bad Info Package) response if the RCS Conversation is not associated with an MLS Group.
- Generate a 200 OK response if the RCS Conversation is associated with an MLS Group.

The Conversation Focus shall embed the latest MLS GroupInfo for the RCS Conversation in the body of any 200 OK responses to MLS-Group-Info-Pull INFO requests.

6.3.1.1 Communicating Support for MLS GroupInfo

Whenever Messaging Servers send an INVITE request or response for an RCS Conversation using MLS, they must indicate that the associated INVITE dialog supports the MLS GroupInfo pull exchange.

To do so, they shall ensure that these requests and responses contain a Recv-Info header containing the “`MLS-Group-Info-Pull`” value.

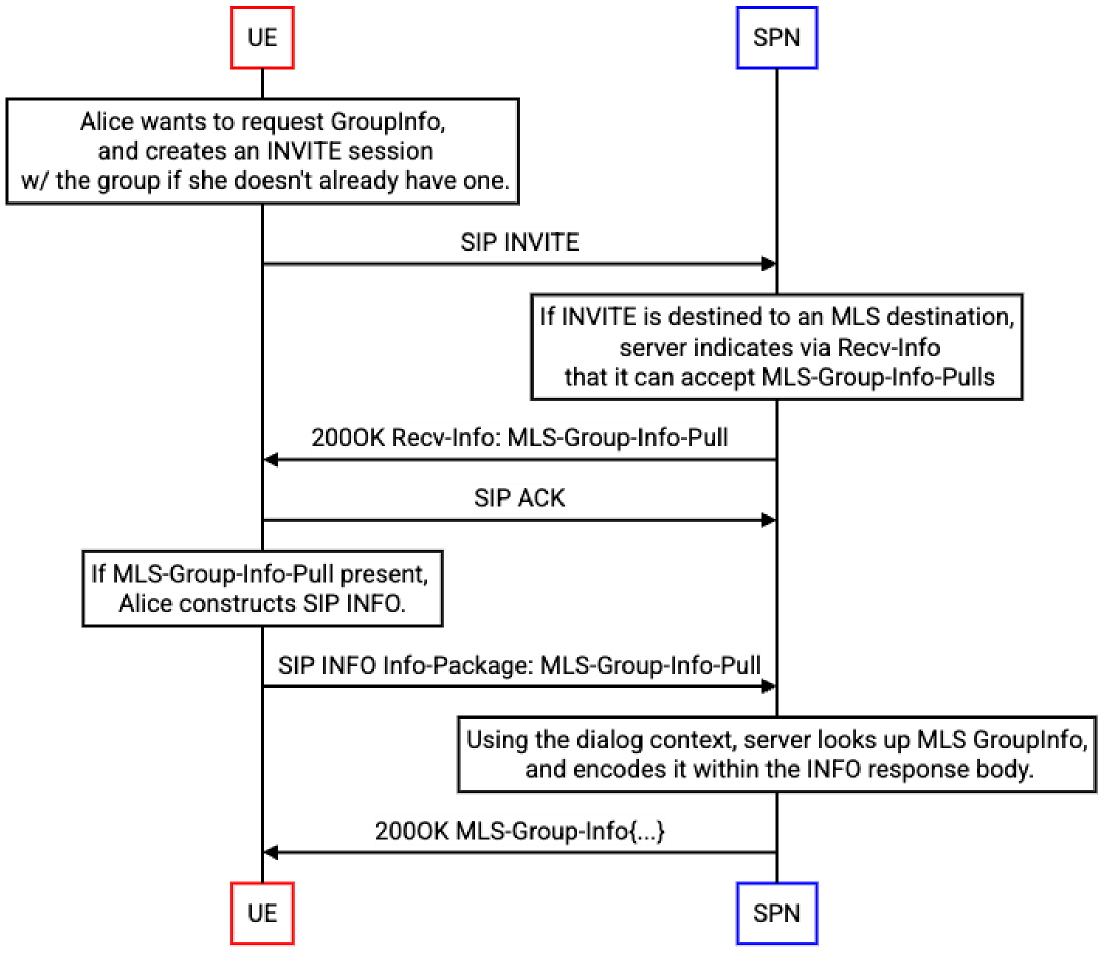
UE
SPN
Alice wants to request GroupInfo, and creates an INVITE session w/ the group if she doesn't already have one.
SIP INVITE
If INVITE is destined to an MLS destination, server indicates via Recv-Info that it can accept MLS-Group-Info-Pulls
200OK Recv-Info: MLS-Group-Info-Pull
SIP ACK
If MLS-Group-Info-Pull present, Alice constructs SIP INFO.
SIP INFO Info-Package: MLS-Group-Info-Pull
Using the dialog context, server looks up MLS GroupInfo, and encodes it within the INFO response body.
200OK MLS-Group-Info{...}
UE
SPN

Figure 10: Recv-Info Header Signals MLS-Group-Info-Pull Support

7 Wireformat

7.1 MLS Content Types

Two new content types are defined for MLS in the following table:

Content-Type	Specification	Included Message
message/mls	[RFC9420]	MlsMessage
message/mls-rcs-client	[RCC.16]	Control Messages defined in section 7.9.1
message/mls-rcs-server	[RCC.16]	Control Messages defined in section 7.9.1

Table 1: MLS Content Types

7.2 MLS CPIM Namespace

A new CPIM namespace is defined for new MLS-related CPIM headers.

As per CPIM [RFC3862], this specification defines a new namespace for the CPIM extension header fields defined in the following sections.

The namespace is:

<http://www.gsma.com/rcs/mls>

As per CPIM [RFC3862] requirements, the new header fields defined in the following sections are prepended, in CPIM messages, by a prefix assigned to the URN through the NS header field of the CPIM message.

The remainder of this specification always assumes an NS header field like this one:

NS: mls <http://www.gsma.com/rcs/mls/>

As specified in [RFC5438], clients are free to use any namespace prefix, while servers and intermediaries must accept any legal namespace prefix specification.

7.2.1 Epoch-Authenticator CPIM header

The header is defined as an extension to the [RFC3862] field definitions. The limits for the occurrence of the field are defined in the following table:

Field	Min Number	Max Number
Epoch-Authenticator	0	1

Table 2: Epoch-Authenticator CPIM Header

The field itself is defined in ABNF as follows:

```
Base-64-char = ALPHA / DIGIT / "+" / "/" / "="
epoch-authenticator = "Epoch-Authenticator:" epoch-authenticator-
value CRLF
epoch-authenticator-value = 1*base-64-char
```

An example CPIM header is `mls.Epoch-Authenticator: MTIzNDU=`.

7.2.2 MLS- Derived-Content-Signature CPIM Header

The header is defined as an extension to the [RFC3862] field definitions. The limits for the occurrence of the field are defined in the following table:

Field	Min Number	Max Number
MLS-Derived-Content-Signature	0	1

Table 3: MLS- Derived-Content-Signature CPIM Header

The field itself is defined in ABNF as follows:

```
Base-64-char = ALPHA / DIGIT / "+" / "/" / "="
mls-derived-content-signature = "MLS-Derived-Content-Signature:" mls-
derived-content-signature-value CRLF
```

```
    mls-derived-content-signature-value = 1*base-64-char
```

An example CPIM header is `mls.mls-derived-content-signature: MTIzNDU=`.

7.2.3 Original-Message-Id CPIM Header

The header is defined as an extension to the [RFC3862] field definitions. The limits for the occurrence of the field are defined in the following table:

Field	Min Number	Max Number
Original-Message-Id	0	1

Table 4: Original-Message-Id CPIM Header

The field itself is defined in ABNF as follows:

```
    Base-64-char = ALPHA / DIGIT / “+“ / “/” / “=”
    Original-message-id = “Original-Message-Id” : original-message-id
    CRLF
    Original-message-id = Token
```

```
An example CPIM header is mls.Original-Message-Id: 34jk324j
```

7.2.4 `MLS-Opaque-Token` SIP Header

The `MLS-Opaque-Token` SIP header is defined in ABNF as follows:

```
mls-opaque-token = “MLS-Opaque-Token:” mls-opaque-token-value CRLF
mls-opaque-token-value = 1*(alphanum / "." / "!" / "%" / "*" / "_" / "+" /
"`" / "'" / "~" )
```

7.3 Binary Encoding Format

All binary formats defined by this specification use the TLS representation language format introduced in [RFC8446].

They also use the variable length encoding scheme and optional fields introduced by [RFC9420].

7.4 KeyPackage Definition

KeyPackages uploaded by clients shall:

- Be represented as message/mls Messages containing a KeyPackage prescribed by [RFC9420].
    - Include a Certificate in the format described by Annex A.3.
    - Include a Certificate whose expiry is valid according to Annex A.3.

7.5 Encrypted Message Format

To construct an Encrypted Message, the client shall:

- Construct a SecretPayload Message (as per section 7.5.1).
- Encrypt the SecretPayload Message and embed that in a PrivateMessage (as per section 7.5.2).
- Include the AuthenticatedData in the PrivateMessage (as per section 7.5.3).
- Create the Epoch-Authenticator CPIM header (as per section 7.5.5).
- Assemble the PrivateMessage in a CPIM container (with the Epoch-Authenticator CPIM header) and embed that into an MSRP Message (as per section 7.5.6).

The end result shall look like Figure 11.

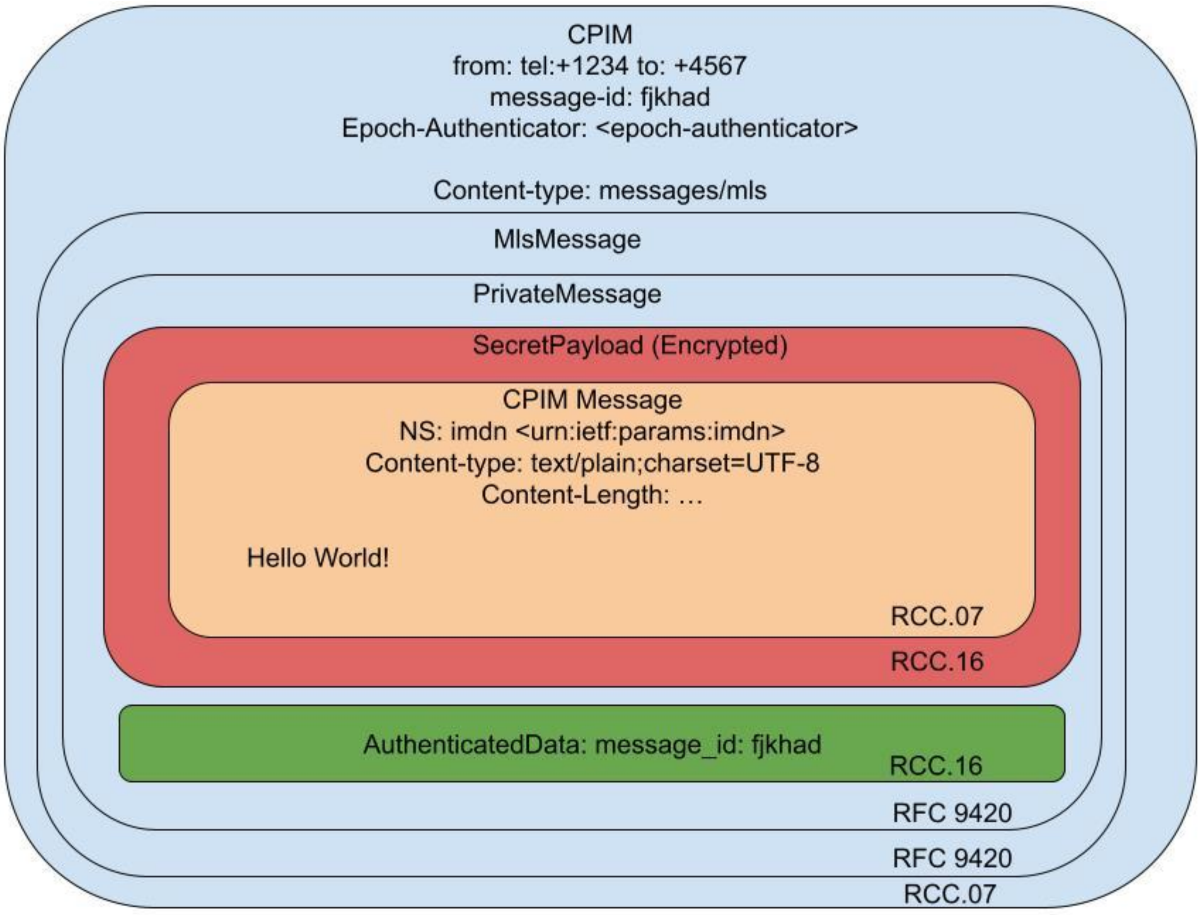
CPIM
from: tel:+1234 to: +4567
message-id: fjkhad
Epoch-Authenticator: <epoch-authenticator>
Content-type: messages/mls
MlsMessage
PrivateMessage
SecretPayload (Encrypted)
CPIM Message
NS: imdn <urn:ietf:params:imdn>
Content-type: text/plain;charset=UTF-8
Content-Length: …
Hello World!
RCC.07
RCC.16
AuthenticatedData: message_id: fjkhad
RCC.16
RFC 9420
RFC 9420
RCC.07

Figure 11: Wireformat for EncryptedMessage

7.5.1 SecretPayload Definition

SecretPayload objects are constructed from the contents of a CPIM Message that a client wishes to encrypt.

To Construct a SecretPayload (format defined in section 7.5.1.1), the client shall:

- Construct a CPIM message as per [GSMA PRD-RCC.07] with:
  - All CPIM namespaces not equal to <urn:ietf:params:imdn>
  - CPIM headers (which are not in the list of Unencrypted CPIM headers from section 7.5.7)
  - MIME Body

- Include the CPIM message in the payload field of the SecretPayload.
- Include the version in the SecretPayloadVersion.
- Include the SecretPayloadType:
    - `hpke_1_to_1_message` for Re-Sent Messages as per section 10.3.
    - `application` for any other message.

7.5.1.1 SecretPayload Binary Format

```
enum {
  reserved(0),
  v1(1),
  (65535)
} SecretPayloadVersion;
enum {
  reserved(0),
  // normal application message
  application(1),
  // resent message after receiving FTD
  hpke_1_to_1_message(2),
  (65535)
} SecretPayloadType;

struct {
  SecretPayloadVersion version = v1;
  SecretPayloadType type;
  // Contents of a CPIM message.
  opaque payload<V>;
} SecretPayload;
```

7.5.2 Application Message Definition

Once a SecretPayload is constructed, to create a PrivateMessage with a ContentType of “application” as per [RFC9420], clients shall:

- Encrypt the SecretPayload as per [RFC9420] to generate a ciphertext.
- Create an `authenticated_data` containing a binary encoded AuthenticatedData whose format is defined in section 7.5.3.2.
- Create an MLS PrivateMessage with those fields according to the rules of [RFC9420].
- Embed the MLS PrivateMessage in an MlsMessage as per [RFC9420].

7.5.3 AuthenticatedData Definition

The `authenticated_data` field within PrivateMessages can be used to ensure that a selected portion of the public content in the CpimMessage is not modifiable by anyone other than the sender.

All public information that needs to be authenticated in this manner shall be encoded within the AuthenticatedData struct.

To construct an AuthenticatedData (as defined in section 7.5.3.2), the client shall:

- Include the value of the CPIM IMDN Message-Id header of the message being sent in the `message_id` field.
- Include the version of the AuthenticatedData being encoded in the AuthenticatedDataVersion field.

7.5.3.1 AuthenticatedData Validation

When Clients receive Encrypted Messages, they shall validate the `authenticated_data` field according to the rules of [RFC9420].

In addition, they shall also ensure that the Message-Id encoded within the AuthenticatedData parsed from the verified `authenticated_data` matches the Message-Id of the CPIM message that the encrypted message was delivered in.

7.5.3.2 AuthenticatedData Binary Format

```
enum {
  reserved(0),
  v1(1),
  (65535)
} AuthenticatedDataVersion;

struct {
  AuthenticatedDataVersion version = v1;
  // Message-Id value. Encoded in UTF-8.
  opaque message_id<V>;

  // Original-Message-Id value. Encoded in UTF-8.
  opaque original_message_id<V>;
} AuthenticatedData;
```

7.5.4 Re-Sent Message Binary Format

```
Struct {
  opaque original_message<V>;
  // Padding for message of length (Original   Message)
  opaque padding<V>;
} ResentMessage;
```

The type field in the SecretPayload struct is set to `hpke_1_to_1_message`.

7.5.5 Epoch-Authenticator CPIM Header

The Epoch-Authenticator CPIM header is defined in the `mls.gsma.com` namespace.

This header shall be in all CPIM messages that contain either of the following:

- An Encrypted Message payload (including partially encrypted messages like File Transfers)
- An MLS Control Message payload

This header shall contain a Base64 encoded `epoch_authenticator` that has been derived from the epoch and MLS Group in whose context the message has been sent as per [RFC9420].

7.5.6 MSRP Message Format

Once the MlsMessage that contains the PrivateMessage (as per section 7.5.2) and the Epoch-Authenticator CPIM Header (as per section 7.5.5) are constructed, the client can construct an MSRP Message. The client shall:

- Construct a CPIM Message as per [GSMA PRD-RCC.07] with a MIME Content-Type of `message/mls`:
  - The body of the CPIM Message will be a binary encoded MlsMessage.
  - Include only the headers explicitly specified in section 7.5.7.
  - Include only the <urn:ietf:params:imdn>, <http://www.gsma.com/rcs> and <http://www.gsma.com/rcs/mls> namespaces.
  - Include a Disposition-Notification header that contains at least the "negative-delivery" enum, in addition to the ones defined in [GSMA PRD-RCC.07].
  - Include the Epoch-Authenticator CPIM header (from section 7.5.5) and its value.
- Include the CPIM Message in an MSRP Body as per [GSMA PRD-RCC.07]

7.5.7 Unencrypted CPIM headers

All headers other than the following shall be encrypted. If a namespace is not specified, then it should be assumed that the Header is in the default namespace.

- Default namespace:
  - To
  - From
  - DateTime
  - Require
- IMDN <urn:ietf:params:imdn> namespace:
  - Message-ID
  - Disposition-Notification
  - Original-To
  - IMDN-Record-Route
  - IMDN-Route
- RCS <http://www.gsma.com/rcs> namespace:
  - advised-action
  - source
- MLS <http://www.gsma.com/rcs/mls> namespace:
  - Epoch-Authenticator

7.6 Signed Message

Messages whose contents are not encrypted may be signed. This signature can be used by recipients to assert the authenticity of the sender and to allow recipients to assert that the contents of the message have not been modified since the message was sent.

Only specific message types may be signed. These include:

- Delivery and Display IMDNs
- Client-generated Negative-Delivery IMDNs
- File Transfer Messages

The derivation of exactly what content is to be signed is specific to the type of message being signed.

7.6.1 Signature Generation

In order to generate a signature, clients shall:

- Generate a VerifiableDerivedContent (as defined in section 7.6.3) from the contents of the CPIM message that they wish to send.
- Create a FramedContent as per [RFC9420] with the `authenticated_data` set to the VerifiableDerivedContents.
- Generate an MLS Public Message which was built using the created FramedContent as per [RFC9420]. The Public Message shall have the proposal type `rcs_signature` as per section 7.11.7.1.
- Encode in Base64 the generated binary MLS Public Message.
- Add this Base64 String to the CPIM MLS-Derived-Content-Signature header.

7.6.2 Signature Validation

In order to validate the signature of a message that included an MLS-Derived-Content-Signature header, clients shall:

- Base64 Decode the MLS-Derived-Content-Signature CPIM header.
- Parse the sequence of bytes as an MLS Public Message.
- Verify that the MLS Public Message was sent by a valid member of the group as per [RFC9420].
- Generate a VerifiableDerivedContent from the contents of the CPIM message.
- Ensure that this VerifiableDerivedContent matches the VerifiableDerivedContent included in the FramedContent of the PublicMessage.
- Verify the signature in the FramedContentAuthData as per [RFC9420].

7.6.3 VerifiableDerivedContent Format

7.6.3.1 VerifiableDerivedContentVersion

The VerifiableDerivedContentVersion is an enum identifying the version of the struct. It is used across all VerifiableDerivedContent.

```
enum {
  reserved(0),
  v1(1),
  (65535)
} VerifiableDerivedContentVersion;
```

7.6.3.2 Delivery IMDN VerifiableDerivedContent Format

VerifiableDerivedContent is structured in the same way for Positive- and Negative-Delivery IMDNs.

The format of the VerifiableDerivedContent for Delivery IMDNs is the following:

```
enum {
  reserved(0),
  delivered(1),
  failed(2),
  forbidden(3),
  error(4),
  (65535)
} DeliveryNotificationStatus;

enum {
  unset(0),
  message_from_non_member (1),
  invalid_credential(2),
  invalid_commit(3),
  failure_to_decrypt(4),
  commit_in_privatemessage(5),
  (65535)
} MlsClientFailureReason;

struct {
  VerifiableDerivedContentVersion version = v1;

  // Value of the <imdn><delivery-notification><status>.
  DeliveryNotificationStatus delivery_notification_status;

  // Message-Id derived from <imdn><message-id> element.
  // Encoded in UTF-8.
  opaque message_id<V>;

  // Value of the <imdn><delivery-notification>
  // <mls-client-failure-reason><status> element.
  // May be empty for a message that was delivered.
  MlsClientFailureReason mls_client_failure_reason;
} VerifiableDeliveryImdn;
```

7.6.3.3 Display IMDN VerifiableDerivedContent Format

The format of the VerifiableDerivedContent for Display IMDNs is the following:

```
enum {
  reserved(0),
  displayed(1),
  forbidden(2),
  error(3),
  (65535)
} DisplayNotificationStatus;

struct {
  VerifiableDerivedContentVersion version = v1;

  // Value of the <imdn><delivery-notification><status>.
  DisplayNotificationStatus display_notification_status;

  // Message-Id derived from <imdn><message-id> element.
  // Encoded in UTF-8.
  opaque message_id<V>;
} VerifiableDisplayImdn;
```

7.6.3.4 File Transfer VerifiableDerivedContent Format

The format of the VerifiableDerivedContent for File Transfers is the following:

```
  struct {
    VerifiableDerivedContentVersion version = v1;

    // Message-Id of enclosing message. Encoded in UTF-8.
    opaque message_id<V>;
  } VerifiableFileTransfer;
```

7.7 IMDN Definition

7.7.1 Positive-Delivery IMDN Definition

Positive-Delivery IMDNs are to be sent as Signed Messages which contain a message/imdn+xml Body with a <delivery-notification><status> element set to “delivered”.

When the Positive-Delivery is for a Re-Sent Message, the client shall include the Re-Sent Message ID (instead of the original Message ID) in the <imdn><message-id> element. The client shall include the original Message ID in the Original-Message-Id header as defined in section 7.2.3.

7.7.2 Negative-Delivery IMDN Definition

7.7.2.1 Server-Generated Negative-Delivery IMDN

Server-generated Negative-Delivery IMDNs shall be used to communicate errors as per section 6. Server shall not send Negative-Delivery IMDNs in any other contexts.

Server-generated Negative-Delivery IMDNs shall be sent as regular CPIM messages which contain a message/imdn+xml Body with a `<delivery-notification><status>` element set to “failed”.

The `<delivery-notification>` element shall also contain a new `<mls-server-failure-reason>` element.

The XML format of the server-generated Negative-Delivery IMDN shall follow the XML schema defined in section 7.7.2.3.

7.7.2.2 Client-Generated Negative-Delivery IMDN

Client-generated Negative-Delivery IMDNs shall be used to communicate that:

- The decryption of an Encrypted Message failed.
- The verification of a Signed Message failed.
- The validation of a Commit or Proposal failed.
- The client ignored an FTD as per section 10.3.

Clients shall not send Negative-Delivery IMDNs in any other contexts.

Client-generated Negative-Delivery IMDNs shall be sent as Signed Messages which contain a message/imdn+xml Body with a `<delivery-notification><status>` element set to “failed”.

The `<delivery-notification>` element shall also contain a new `<mls-client-failure-reason>` element.

The XML format of the client-generated Negative-Delivery IMDN shall follow the XML schema defined in Section 7.7.2.3.

7.7.2.3 Delivery IMDN Schema

The deliveryNotification XML element as defined in [RFC5438] will be modified to conform to the following schema:

```
<define name="deliveryNotification">
  <element name="delivery-notification">
    <element name="status">
      <choice>
        <element name="delivered">
          <empty/>
        </element>
        <element name="failed">
          <choice>
            <element name="mls-server-failure-reason">
              <choice>
                <element name="incorrect-era">
                  <empty/>
                </element>
                <element name="incorrect-epoch">
                  <empty/>
                </element>
                <element name="incorrect-epoch-authenticator">
                  <empty/>
                </element>
                <element name="expired-credential">
                  <empty/>
                </element>
                <element name="mismatched-rcs-group-state">
                  <empty/>
                </element>
                <element name="unparsable-commit">
```

```
                  <empty/>
                </element>
                <element name="mismatched-confirmation-tag">
                  <empty/>
                </element>
                <element name="pending-proposal">
                  <empty/>
                </element>
                <element name="transient-error">
                  <empty/>
                </element>
                <element name="encryption-not-available">
                  <empty/>
                </element>
                <element name="invalid-commit">
                  <empty/>
                </element>
              </choice>
            </element>
            <element name="mls-client-failure-reason">
              <choice>
                <element name="message-from-non-member">
                  <empty/>
                </element>
                <element name="invalid-credential">
                  <empty/>
                </element>
                <element name="invalid-commit">
                  <empty/>
                </element>
                <element name="failed-to-decrypt">
                  <empty/>
                </element>
                <element name="commit-in-privatemessage">
                  <empty/>
                </element>
              </choice>
            </element>
          </choice>
        </element>
        <ref name="commonDispositionStatus"/>
      </choice>
      <ref name="deliveryExtension"/>
    </element>
  </element>
</define>
```

7.7.3 Display IMDN Definition

Display IMDNs are to be sent as Signed Messages which contain a message/imdn+xml Body with a `<display-notification><status>` element set to “displayed”.

When the Display IMDN is for a Re-Sent Message, the client shall include the Re-Sent Message ID (instead of the original Message ID) in the `<imdn><message-id>` element. The client shall include the original Message ID in the Original-Message-Id header as defined in section 7.2.3.

7.8 File Transfer Message Definition

7.8.1 FileInfo Message

The file encryption key and other information are defined in this FileInfo proto:

```
enum Algorithm {
  ALGORITHM_UNSPECIFIED = 0;
  AES256_CTR_HMAC_SHA256_256TAG = 1;
}

message FileEncryptionInfo {
  bytes key_material = 1;
  bytes initialization_vector = 2;
  bytes hmac_tag = 3;
  Algorithm algorithm = 4;
}

message FileMetadata {
  string file_name = 1;
  string content_type = 2;
  FileEncryptionInfo encryption_info = 3;
}

message FileInfo {
  FileMetadata file = 1;
  FileMetadata thumbnail = 2;
  FileMetadata subject = 3;
  FileMetadata icon = 4;
}
```

The Client shall:

1. Create a FileInfo proto and binary encode it.
2. Construct an MlsMessage as per section 7.5.2 containing the binary encoded FileInfo as the MIME body with a ContentType of “message/mls-rcs-file-info”.

7.8.2 File Transfer Message Body

Once the client has generated an MlsMessage as per section 7.8.1, they shall:

1. Base64 encode the EncryptedMessage.
2. Embed the Base64 encoded message into the mls-file XML element defined in Table 5.
3. Set the content-type in the file_info section of the XML to “message/mls-ft” for both the file and the thumbnail

```
<?xml version="1.0" encoding="UTF-8"?>
<file xmlns="urn:gsma:params:xml:ns:rcs:rcs:fthttp"
        xmlns:x="urn:gsma:params:xml:ns:rcs:rcs:up:fthttpext">
        <file-info type="thumbnail">
            <file-size>[encrypted thumbnail size in bytes]</file-size>
            <content-type>message/mls-ft</content-type>
            <data url = "[HTTP URL for the thumbnail]" until = "[validity of the thumbnail]"/>
        </file-info>
        <file-info type="file" file-disposition="[file-disposition]">
            <file-size>[encrypted file size in bytes]</file-size>
            <file-name>encrypted_file</file-name>
            <content-type>message/mls-ft</content-type>
            <data url = "[HTTP URL for the file]" until = "[validity of the file]"/>
            <x:branded-url>[alternative branded HTTP URL of the file]</x:branded-url>
        </file-info>
        <mls-file>
            [base64 encoding of the MlsMessage]
        </mls-file>
</file>
```

Table 5: File Transfer Message XML Schema

7.9 Commit and Proposal Messages

Commit and Proposal Messages shall be represented as CPIM messages when they are exchanged to or from Clients.

These messages shall have a Content-Type of `message/mls-rcs-client` for messages originating from the client and `message/mls-rcs-server` for messages originating from the Conversation Focus.

7.9.1 message/mls-rcs-client and message/mls-rcs-server ContentTypes

The `message/mls-rcs-client` and `message/mls-rcs-server` content-types are used to compactly encode the contents of multiple message/mls contents. The `message/mls-rcs-client` shall contain the `ClientMlsRcsMessage` proto defined below. The `message/mls-rcs-server` shall contain the `ServerMlsRcsMessage` proto defined below. The client and Conversation Focus shall include the binary encoding of the proto in the CPIM body.

The following structres are represented in TLS format:

```
struct {
  // List of MlsMessages containing a Proposal.
  MlsMessage mls_messages<V>;
} ProposalList;

struct {
  // List of MlsMessaes containing a Commit.
  MlsMessage mls_messages<V>;
} CommitList;
```

The following structures are represented in Protocol Buffer format:

```
// MlsRcsMessage payload type that represents Commit messages and
// Welcome messages.
message WelcomeCommitBundle {
  // contains a CommitList TLS structure that is a List of commit
  // messages.
  bytes commit_list = 1;

  // Raw bytes that represent a welcome message. This is expected to
  // be a serialized MlsMessage that contains a Welcome message.
  bytes welcome = 2;

  // Raw bytes that represent the MLS GroupInfo. This is expected to
  // be wrapped by an MlsMessage.
  bytes group_info = 3;

  // Raw bytes that represent the EpochAuthenticator.
  bytes epoch_authenticator = 4;

  // Raw bytes that represent a RatchetTree.
  // This should only be included for the first commit in an Era.
  bytes ratchet_tree = 5;
}

// MlsRcsMessage payload type that represents a Commit message.
message CommitBundle {
  // contains a CommitList TLS structure that is a List of commit
  // messages.
  bytes commit_list = 1;

  // Optional MlsMessage containing a PrivateMessage that should
  // be delivered transactionally with the group state changes in
  // this bundle.
  bytes private_message = 2;

  // Raw bytes that represent the MLS GroupInfo. This is expected to
  // be wrapped by an MlsMessage.
  bytes group_info = 3;

  // Raw bytes that represent the EpochAuthenticator.
  bytes epoch_authenticator = 4;
}

// ServerMlsRcsMessage payload that is generated by the server
// after processing a CommitBundle or WelcomeCommitBundle.
message ServerCommitBundle {
  // contains a CommitList TLS structure that is a List of commit
  // messages.
  bytes commit_list = 1;

  // Raw bytes that represent a welcome message. This is expected to
  // be a serialized MlsMessage that contains a welcome message.
  bytes welcome = 2;

  // Raw bytes that represent a private message bundled with the
```

```
    // commit. This is expected to be included for Group Subject/Icon
    // changes.
    bytes private_message = 3;
  }

  // Information persisted by the server that can be used by clients to
  // initiate self-healing.
  message MlsGroupInfo {
    // The latest GroupInfo for the MLS Group.
    // This is expected to a serialized MlsMessage that contains a
    //  GroupInfo.
    bytes group_info = 1;

   // The latest RatchetTree for the MLS Group.
    bytes ratchet_tree = 2;

    // Proposals that have been accepted by the server, but not yet
    // committed.
    // This is a ProposalList TLS message.
    bytes pending_proposals = 3;
  }

  // MLS control messages sent by servers.
  message ServerMlsRcsMessage {
    oneof payload {
      // ProposalList TLS payload that was sent by clients.
      bytes proposal_list = 1;

      // Commit message payload that was sent by clients.
      ServerCommitBundle server_commit_bundle = 2;

      // Information stored about an MLS group that can be used by
      // clients to initiate self-healing.
      MlsGroupInfo mls_group_info = 3;
    }
  }

  // MLS control messages sent by clients.
  message ClientMlsRcsMessage {
    oneof payload {
      // ProposalList TLS payload for Proposals by the client.
      bytes proposal_list = 1;

      // Welcome and Commit message payload.
      WelcomeCommitBundle welcome_commit_bundle = 2;

      // Commit message payload.
      CommitBundle commit_bundle = 3;
    }
  }
```

7.9.2 Client-Generated Commits

When creating a new RCS Conversation, or a new Era, the Initial Commit Messages generated by clients shall include:

- A Commit representing a change to the MLS Group as per [RFC9420].
- A GroupInfo object which represents the state of the MLS Group after the contents of the associated Commit have been applied as per [RFC9420].
- An Epoch Authenticator as defined by [RFC9420] for the new Epoch.
- Welcome Message as defined by [RFC9420].
- A Ratchet Tree representing the current members of the MLS Group.

The above will be embedded in the `welcome_commit_bundle` field of the `ClientMlsRcsMessage` as defined in section 7.9.1. The `ClientMlsRcsMessage` shall be embedded in a CPIM message.

For a Commit that adds a new Client to the MLS Group, or performs a KeyPackage Update as per section 9.5.4, the client shall include:

- A Commit representing a change to the MLS Group as per [RFC9420].
- A GroupInfo object which represents the state of the MLS Group after the contents of the associated Commit have been applied as per [RFC9420].
- An Epoch Authenticator as defined by [RFC9420] for the new Epoch.
- Welcome Message as defined by [RFC9420].

The above will be embedded in the `welcome_commit_bundle` field of the `ClientMlsRcsMessage` as defined in section 7.9.1. The ractchet_tree field shall be empty. The `ClientMlsRcsMessage` shall be embedded in a CPIM message.

For any other type of Commit, the client shall include:

- A Commit representing a change to the MLS Group as per [RFC9420].
- A GroupInfo object which represents the state of the MLS Group after the contents of the associated Commit have been applied as per [RFC9420].
- An Epoch Authenticator as defined by [RFC9420] for the new Epoch.
- An Optional PrivateMessage including icon and subject keys when there is a change to either.

The above will be embedded in the `commit_bundle` field of the `ClientMlsRcsMessage` as defined in section 7.9.1. The ClientMlsRcsMessage shall be embedded in a CPIM message.

7.9.3 Server-Processed Commits

After the Conversation Focus accepts the Commit, the Conversation Focus shall remove the GroupInfo body, Ratchet Tree and Epoch Authenticator before delivering the message.

The Conversation Focus shall create a `ServerCommitBundle` including the Commit and optionally the Welcome Message and the private message for group subject and icon and embed it in the `server_commit` bundle field of the `ServerMlsRcsMessage`.

When receiving the very first `welcome_commit_bundle` in an RCS Conversation or new Era, the Conversation Focus shall store the `ratchet_tree`. For all Commits other than that the first commit, the Conversation Focus shall update the `ratchet_tree` stored with the changes in the Commit.

7.9.4 Proposal Lists

When sending a list of proposals, the Client shall:

- Embed each Proposal in an MlsMessage.
- Costruct a `ProposalList` Message containing all the MlsMessages.
- Construct a `ClientMlsRcsMessage` containing the ProposalList in the `proposal_list` field.

7.10 MLS GroupInfo Retrieval Format

7.10.1 SIP Info Response Body

When the Conversation Focus receives a SIP INFO request as per section 6.3.1, the Conversation Focus shall:

- Retrieve the MLS GroupInfo, Ratchet Tree and any pending Proposals stored for the MLS Group.
- Construct an `ServerMlsRcsMessage` with a ServerGroupInfo as per section 7.9.1.
- Construct a MIME body with the binary-encoded `ServerMlsRcsMessage` and the content-type being `message/mls-rcs-server`.
- Include the MIME body in the body of the SIP INFO 200OK.

7.11 MLS Extensions

7.11.1 Era

7.11.1.1 Era GroupContext Extension

The Era extension is a GroupContext Extension as defined in section 17.3 of [RFC9420]. Its attributes are as follows:

- Extension Value: 0xF001
- Extension Name: era
- extension_data: uint32 numeric value of the Era
- Applicable Messages: GroupInfo and Welcome Message

7.11.2 end_mls

7.11.2.1 end_mls Proposal

The `end_mls` extension is a Proposal extension as defined in section 17.4 of [RFC9420]. Its attributes are as follows:

- Extension Value: 0xF001
- Extension Name: `end_mls`
- External: No
- Path Required: No

7.11.2.2 end_mls GroupContext Extension

The `end_mls` extension is a Group Context Extension as defined in section 17.3 of [RFC9420]. Its attributes are as follows:

- Extension Value: 0xF002
- Extension Name: `end_mls`
- extension_data: "`end_mls`"
- Applicable Messages: GroupInfo and Welcome Message

7.11.3 icon_key

7.11.3.1 icon_key GroupContext Extension

The `icon_key` extension is a Group Context Extension as defined in section 17.3 of [RFC9420]. Its attributes are as follows:

- Extension Value: 0xF003
- Extension Name: icon_key
- extension_data: the symmetric key used to encrypt the icon.
- Applicable Messages: Welcome Message

7.11.4 icon_commitment

7.11.4.1 icon_commitment GroupContext Extension

The `icon_commitment` extension is a Group Context Extension as defined in section 17.2 of [RFC9420]. Its attributes are as follows:

- Extension Value: 0xF004
- Extension Name: icon_commitment
- extension_data: the icon commitment as defined in Annex C.1
- Applicable Messages: GroupInfo and Welcome Message

7.11.5 subject_key

7.11.5.1 subject_key GroupContext Extension

The `subject_key` extension is a Group Context Extension as defined in section 17.2 of [RFC9420]. Its attributes are as follows:

- Extension Value: 0xF005
- Extension Name: `subject_key`
- extension_data: the symmetric key used to encrypt the subject.

Applicable Messages: Welcome Message

7.11.6 subject_commitment

7.11.6.1 subject_commitment GroupContext Extension

The `subject_commitment` extension is a Group Context Extension as defined in section 17.2 of [RFC9420]. Its attributes are as follows:

- Extension Value: 0xF006
- Extension Name: `subject_commitment`
- extension_data: the subject commitment as defined in Annex C.1
- Applicable Messages: GroupInfo and Welcome Message

7.11.7 rcs_signature

7.11.7.1 rcs_signature Proposal

The rcs_signature extension is a Proposal extension as defined in section 17.4 of [RFC9420]. Its attributes are as follows:

- Extension Value: 0xF002
- Extension Name: `rcs_signature`
- External: No
- Path Required: No

7.11.8 self_remove

7.11.8.1 self_remove Proposal

The self_remove extension is a Proposal extension as defined in section 17.4 of [RFC9420]. Its attributes are as follows:

- Extension Value: 0xF003
- Extension Name: `self_remove`
- External: Yes
- Path Required: Yes

7.11.9 server_remove

7.11.9.1 server_remove Proposal

The server_remove extension is a Proposal extension as defined in section 17.4 of [RFC9420]. Its attributes are as follows:

- Extension Value: 0xF004
- Extension Name: `server_remove`
- External: Yes
- Path Required: Yes

7.12 ACS Signed Encryption Identity Proof

When ACS signs the Encryption Identity Proof as defined in section 4.1, ACS shall construct and sign `SignedEncryptionIdentityProofTBS` as per section 7.12.1 and include all the elements including the signature in `SignedEncryptionIdentityProof` as per section 7.12.1.

7.12.1 ACS Signed Encryption Identity Proof Format

```
struct {
  // MSISDN of the Participant verified by ACS
  opaque msisdn<V>;
  // Participant Key sent in the request, encoded in DER format
  opaque participant_key<V>;
  // Home KDS sent in the request
  uint32 home_kds<V>;
  // Expiry of this signed Tuple. Measured in seconds since the Unix
  // epoch (1970-01-01T00:00:00Z)
  uint64 expiry_seconds<V>;
} SignedEncryptionIdentityProofTBS;

struct {
  // Expiry of this signed Tuple. Measured in seconds since the Unix
  // epoch (1970-01-01T00:00:00Z) as per []
  uint64 expiry_seconds<V>;
  // Signature of the above request
  opaque signature<V>;
} SignedEncryptionIdentityProof;
```

8 Conversation Creation

8.1 Client Procedures

The Client shall use procedures defined in section 5.2 to check if the Participants are capable of E2EE. If any of the Participants are not capable of E2EE, then the client shall revert to creating an unencrypted conversation as specified in [GSMA PRD-RCC.07].

When all Participants are capable of E2EE, the client shall request and validate all KeyPackages for all Participants in the Conversation as per section 5.3. The Home KDS shall return KeyPackages for all Clients of each Participant. If the Home KDS does not return KeyPackages for one or more of the Participants requested, the client shall create an unencrypted conversation.

With all KeyPackages fetched, the client shall create an MLS Group as per [RFC9420]. The MLS Group shall contain all KeyPackages for all Clients of the Participants. The MLS Group must contain the Era as a GroupContext Extension. The Client shall generate an MLS GroupInfo, initial Commit, Ratchet Tree, Epoch Authenticator and Welcome Message.

Upon creating the MLS Group Info, initial Commit, Ratchet Tree, Epoch Authenticator and Welcome Message, the client shall create a SIP INVITE as per section 8.1.1 or 8.1.2 and send the INVITE to the Messaging Server.

After receiving the 200 OK from the Messaging Server that includes the `MLS-Opaque-Token`, the client shall store the `MLS-Opaque-Token` and use it for all subsequent requests.

When receiving an INVITE from the Messaging Server with an initial Commit and Welcome Message, the client shall validate the Welcome Message and Commit as per section 6.2.6, store the MLS Group locally, and use the `MLS-Opaque-Token` specified in the INVITE for all future requests to the Messaging Server for that RCS Conversation.

8.1.1 1-to-1 INVITE

The SIP INVITE created by the client shall (in addition to the requirement set in [GSMA PRD-RCC.07]):

- Include a `multipart/mixed` body.
- Include the SDP as the first part of the INVITE body as per [GSMA PRD-RCC.07].
- Include the binary-encoded MLS GroupInfo, initial Commit, and Welcome Message generated as per section 7.9.2 as the second part of the INVITE body.

8.1.2 Group INVITE

The SIP INVITE created by the client shall (in addition to the requirement set in [GSMA PRD-RCC.07]):

- Include a `multipart/mixed` body.
- Include the SDP as the first part of the INVITE body as per [GSMA PRD-RCC.07].
- Include the resource-lists for the participants of the group as the second part of the INVITE as per [GSMA PRD-RCC.07].
- Include the binary-encoded MLS GroupInfo, initial Commit, and Welcome Messages generated as per section 7.9.2 as the third part of the INVITE body.

8.2 Messaging Server Procedures

8.2.1 Conversation Focus

Upon receiving a SIP INVITE from the client, the originating Messaging Server for a 1-to-1 chat or the Conference Focus for group conversations shall:

- Become the Conversation Focus (as per section 6.1.2) for the RCS Conversation.
- Validate the initial Commit to create the group as per section 6.2.2.
- Store the GroupInfo.
- Return the 200 OK with the `MLS-Opaque-Token`, MLS-Epoch, and MLS-Era headers.

The Conversation Focus shall then fan out the INVITE with the MLS Group, initial Commit, and Welcome Message to the other recipient(s) in the 1-to-1 or group chat (in the resource-list) as per [GSMA PRD-RCC.07]. The Conversation Focus shall also send the initial Commit back to the sender as an MSRP message.

8.2.2 Participant Function

Upon receiving a SIP INVITE from a Conversation Focus, the terminating Messaging Server for a 1-to-1 Chat or the Participation Function for group chat shall:

- Follow the procedures in section 6.1.3 for the `MLS-Opaque-Token`.
- Forward the INVITE as per [GSMA PRD-RCC.07].

8.3 Creating a New Era

Clients shall create a new Era in the following scenarios:

- When self-heal is not successful as per section 10.1.1.
- When resurrecting the MLS Group is not possible due to expired certificates as per section 11.1.
- When restarting an RCS Conversation after becoming inactive (over 30 days of no messages sent).
- When updating the RCS MLS version to a lower version than the current one.
- When downgrading the Cipher Suite of the MLS Group.

8.3.1 Client Procedures

When a client needs to create a new Era, it shall:

- Advance the Era in the conversation by 1.
- Follow the procedures in section 8.1 with the new Era value.

If the client receives a 409 Conflict as a response to the INVITE and/or a negative IMDN with error code `<incorrect-era>`, the client shall fetch the GroupInfo as per section 7.10, create the correct Era, and retry the operation.

8.3.2 New Conversation Focus

When the originating Messaging Server for a 1-to-1 Chat or the new Conversation Focus receives the INVITE with the new Era, it shall assume the Conversation Focus role and follow the procedures in section 8.2.1.

If the Conversation Focus is moving, the new Conversation Focus must wait for a response from the old Conversation Focus before responding to the client.

8.3.3 Old Conversation Focus

If the new Conversation Focus is different from the Conversation Focus for the previous Era, the old Conversation Focus shall:

- Validate that the Era is advanced exactly by 1.
  - If the new Era is not modifying the state exactly by 1, the server shall return a 409 Conflict as a response to the INVITE and send a negative IMDN as per section 6.2.5.
- Delete the MLS GroupInfo stored locally.
- Follow the procedures in section 8.2.2.

8.3.4 Participating Functions

Upon receiving a SIP INVITE from a Conversation Focus, the terminating Messaging Server for a 1-to-1 Chat or the Participation Function for a group chat shall follow procedures as per section 8.2.2.

9 Conversation Operations

9.1 Messaging

9.1.1 Encrypted Messages

The client shall not send encrypted messages if the `end_mls` GroupContext Extension is present in the GroupInfo.

For all content types to be encrypted, such as regular messages, replies, edits, or any other kind of content, the client shall:

- Create the Secret Message format as per section 7.5.1.
- Create a PrivateMessage containing the Secret Message as per [RFC9420].
  - Using the PADME algorithm in [PADME], add padding in the PrivateMessageContent as per [RFC9420].
- Wrap the PrivateMessage in the format indicated in section 7.5. Messages will then be sent over MSRP.

When receiving a message, the Participating Function shall:

- Forward the message to the Conversation Focus.

When receiving an encrypted message, the Conversation Focus shall:

- Verify the message against the Epoch Authenticator stored for the Epoch the message is intended for.
  - If the Epoch Authenticator fails, the Conversation Focus shall reject the message as per section 6.2.5.

- Verify that the sender of the message in the Private Message matches the RCS sender identity (in the Invite from/to header and/or CPIM identity).
  - If they do not match, the Conversation Focus shall reject the message as per section 6.2.5.
- Verify that the `end_mls` is not present in the GroupInfo.
  - If the `end_mls` is present, the server shall reject the message as per section 6.2.5.
- Fan out the message as per [GSMA PRD-RCC.07].

When receiving an encrypted message, the client shall:

- Retrieve the PrivateMessage from the CPIM container.
- Decrypt the Secret Message at the Epoch of the message as per [RFC9420] and remove padding.
- Ensure the sender of the message in the Private Message matches the RCS sender identity (in the Invite from/to header and/or CPIM identity).
  - If the values do not match, the client shall use the MLS identity.

9.1.2 File Transfer

9.1.2.1 File Encryption

To encrypt the file the client shall:

- Follow the file encryption algorithm defined in Annex C.2.
  - The client shall generate fresh key material and use AES-256-CTR for encryption per [NIST SP800-38A], and HMAC-SHA256 to calculate the tag per [RFC5869].
- Follow the same algorithm to encrypt the thumbnail but with a freshly generated key.
- Upload the encrypted files to the HTTP Content Server as per [GSMA PRD-RCC.07].
  - The filename uploaded to the HTTP Content Server shall be set to “encrypted_file”-. The MIME type shall be message/mls-ft.
- Encapsulate the encryption keys, initialization vectors, tags, original file name and types into the new FileInfo proto as defined in section 7.8.1.
- Create a Secret Message containing the serialized FileInfo protocol buffer and encrypt it using MLS Message Encryption per section 9.1.
- The encrypted payload is added to the XML file as per section 7.8.2.
- Wrap the file transfer XML in a CPIM container and send as per [GSMA PRD-RCC.07]

9.1.2.2 File Decryption

The recipient client downloads the encryped file from the HTTP Content Server and shall:

- Decrypt the MLS message part in the XML payload.
- Decode the FileInfo defined in section 7.8.1 to get the encryption keys and tags.
- Compute the tag as defined in Annex C.3 and validate it against the tag in the FileEncryptionInfo struct.
  - If they match, the client shall decrypt the encrypted file using the algorithm defined in Annex C.3.
- Replace the file name and type with the original filename and type from the FileInfo proto.
- Delete the encrypted file.
- Zero-out the file encryption key.

The client shall follow the same steps to decrypt the thumbnail.

9.1.3 Delivery Report

Delivery receipts are not encrypted. However, they must be signed. The client shall:

- Create the AAD as per section 7.6.3.2.
- Store the AAD as per section 7.6.1.
- Send the delivery report as per [GSMA PRD-RCC.07],

9.1.4 Display Report

Display receipts are not encrypted. However, they must be signed. The client shall:

- Create the AAD as per section 7.6.3.3.
- Store the AAD as per section 7.6.1.
- Send the display report as per [GSMA PRD-RCC.07].

9.1.5 User Alias

If present, the client shall encrypt the user alias. When the alias is encrypted, the client shall:

- Create a PrivateMessage containing the alias as per [RFC9420].
  - Using the PADME algorithm in PADME, add padding in the PrivateMessageContent as per [RFC9420].
- Base-64 URL encode the PrivateMessage containing the alias.
- Include the Base-64 encoded message as an extra SIP URI parameter “encrypted-alias=<base-64 encoding>.

An example of the SIP URI with an encrypted alias would be:

From: sip:+1234578901@operator.com;user=phone;encrypted-alias=aGVsbG8gd29ybGQ

When receiving a SIP URI with an encrypted alias header, the client shall:

- Base-64 URL decode the encrypted-alias parameter to retrieve the PrivateMessage.

- Decrypt the raw alias from the PrivateMessage as per [RFC9420].

9.2 Adding Participants to a Group Chat

When adding a new Participant(s), the client shall:

- Query capabilities to ensure the Participant(s) have MLS capabilities as per section 5.2.
- Fetch KeyPackages for the Participant(s) as per section 5.3.
    - The Home KDS shall return KeyPackages for all Clients of each Participant.
    - If the Participant(s) do not support MLS in capabilities or do not have KeyPackages, the client shall move the RCS Conversation to Unencrypted state as per section 11.2.
- Using the KeyPackages, create a Commit to the existing MLS Group that adds all the Clients of the Participant(s) and an Epoch Authenticator for that Epoch as per [RFC9420].
- Create a Welcome Message for the new Client(s) as per section 7.9.2.
    - If the RCS Conversation includes a subject and/or icon, the client shall follow the procedures in section 9.7.1.3.
- Create the updated GroupInfo to be uploaded to the server as per section 7.9.2.
- Create a REFER as per [GSMA PRD-RCC.07] section 3.2.4.6.
    - If it is a single user REFER, create a body with type that includes the `ClientMlsRcsMessage` as per section 7.9.2.
    - If it is a REFER for multiple recipients, include a `multipart/mixed` body that includes:
        - The resource-lists as the first part with the new Participant(s).
        - The `ClientMlsRcsMessage` as per section 7.9.2.
- Send the REFER to the Messaging Server as per [GSMA PRD-RCC.07].
- Wait for the Commit to arrive back as an MSRP message to apply it locally.

When receiving a REFER, the Participating Function shall:

- Include the Conversation Focus's `MLS-Opaque-Token`.
- Forward the REFER to the Conversation Focus.

When receiving a REFER, the Conversation Focus shall:

- Verify the Commit as per 6.2.2.
    - If the Commit is for a previous Epoch, the server shall return a 409 Conflict as a response to the REFER. The client shall then sync their state (either by retrieving Store and Forward messages or self-healing) and trying again.
    - If the Commit fails verification for any other reason, the server shall return a 400 Bad Request.

- In both cases, the server shall send a negative IMDN to the sender indicating the failure as per section 7.7.2.1.

- Verify that the Participants(s) added in the Commit is the same as in the REFER.
- Store the new GroupInfo and Epoch Authenticator and update the ratchet tree.
- Send an INVITE (as per [GSMA PRD-RCC.07]) including the Welcome Message as per section 7.9.3.
    - If the recipient is offline when the INVITE arrives at the Terminating Function, the Terminating Function shall store the Welcome Message for deferred delivery when the recipient comes back online.
- Forward NOTIFYs to all members of the RCS Group as per [GSMA PRD-RCC.07].
- Forward the Commit as MSRP message to all members (including the sender).

When a Commit via MSRP, the client shall:

- Validate the Commit as per section 6.2.6.
- Apply the Commit locally.
- Apply the change to the RCS Conversation state.

9.3 Removing Participants from a Group Chat

When the client wishes to kick out Participant(s), the client shall:

- Create a Commit to remove all the Clients of the Participant(s) to be removed from the RCS Conversation as per [RFC9420].
- Create the updated GroupInfo to be uploaded to the server.
- Create a REFER as per [GSMA PRD-RCC.11] section 7.3.6
    - If it is a single user REFER, create a body with type that includes the epoch information and GroupInfo as per section 7.9.2.
    - If it is a REFER for multiple users, include a `multipart/mixed` body that includes:
        - The resource-lists as the first part with the kicked Participant(s) as per [GSMA PRD-RCC.11] section 7.3.6.
        - Commit, Epoch Authenticator, and GroupInfo as per section 7.9.2.
- Send the REFER to the Messaging Server as per [GSMA PRD-RCC.07].
- Wait for the Commit to arrive back as an MSRP message to apply it locally.

When receiving a REFER, the Participating Function shall:

- Include the Conversation Focus’s `MLS-Opaque-Token`.
- Forward the REFER to the Conversation Focus.

When receiving a REFER, the Conversation Focus shall:

- Verify the Commit as per 6.2.2.

- If the Commit is for a previous Epoch, the server shall return a 409 Conflict as a response to the REFER. The client shall then sync their state (either by retrieving Store and Forward messages or self-healing) and trying again.
- If the Commit fails verification for any other reason, the server shall return a 400 Bad Request.
- In both cases, the server shall send a negative IMDN to the sender as per section 7.7.2.1.

- Verify that the Participants(s) removed in the Commit is the same as in the REFER.
- Store the new GroupInfo and Epoch Authenticator and update the ratchet tree.
- Forward NOTIFYs to all members of the RCS Group as per [GSMA PRD-RCC.07].
  - The server shall include `SIP;cause=410;text="Kicked"` in the `<disconnection-info>` for the removed user.
- Forward the Commit as MSRP message to all members (including the sender).

When receiving a Commit via MSRP, the client shall:

- Validate the Commit as per section 6.2.6.
- Apply the Commit locally.
- Apply the change to the RCS Conversation state.

9.4 Self Leave

When a Participant wants to leave the RCS Conversation, the client shall:

- Create Proposal(s) to remove all the Clients of the Participant as per [RFC9420] using the self_remove Proposal extension as per 7.11.8.1.
- Create a SIP BYE as per [GSMA PRD-RCC.07] with a body that includes the Proposal(s) as per section 7.9.4.
- Send the SIP BYE to the Messaging Server as per [GSMA PRD-RCC.07].
- Wait for the 200OK to arrive before leaving the group.

When receiving a SIP BYE, the Participating Function shall:

- Include the Conversation Focus's `MLS-Opaque-Token`.
- Forward the SIP BYE to the Conversation Focus.

When receiving a SIP BYE, the Conversation Focus shall:

- If the leave is for the last Participant in the RCS Conversation:
  - Delete the MLS Group
  - Return a 200 OK to the Sender as per [GSMA PRD-RCC.07]
- Otherwise:
  - Verify the Proposal as per 6.2.3.

    - If the Proposal is for a previous Epoch, the server shall return a 409 Conflict as a response to the REFER. The client shall then sync their state (either by retrieving Store and Forward messages or self-healing) and trying again.
    - If the Proposal fails verification for any other reason, the server shall return a 400 Bad Request.
    - In both cases, the server shall send a negative IMDN to the sender as per section 7.7.2.1.

  - Verify that the Participant (and all their Clients) that is leaving is the same as the Proposal.
  - Forward NOTIFYs to all members of the RCS Group as per [GSMA PRD-RCC.07].

    - The server shall include `SIP;cause=200;text="Call completed"` in the `<disconnection-info>` for the departing user.

  - Forward the Proposal as an MSRP message to all members (not including the sender).

    - The server may choose to send the Proposal to a single member of the RCS Conversation (e.g. an online member) to avoid race condition on Commits.

  - Not accept any Proposals or Commits as per 6.2.1 until a Commit for the Proposal(s) is applied.

When a client receives a SIP NOTIFY to remove Participant(s) and a Proposal via MSRP, the client shall:

- Validate the Proposal as per section 6.2.6.
- Create a Commit with the Proposal(s) sent in the MSRP to remove the Clients and create the epoch information and send it to the Messaging Server

  - If a Commit arrives that includes the Proposal(s) before the client creates/sends the Commit, the client shall abandon Commit creation/sending.

- Wait for the Commit with the leave Proposal(s) to arrive as an MSRP message to apply it locally.

9.5 Commits

9.5.1 Commit Procedure

When a Participant wants to send a Commit that doesn’t involve changes to the RCS Conversation (e.g. update their Keys, add Clients to existing Participants, update GroupContext/GroupContext Extension) the client shall:

- Create a Commit as per [RFC9420].
- Create the GroupInfo to be uploaded to the server.
- Create an MSRP message with a body that includes the binary encoded Commit, Epoch Authenticator and GroupInfo per section 7.9.2.

- Send the MSRP to the Messaging Server as per [GSMA PRD-RCC.07].
- Wait for the Commit to arrive back as an MSRP message to apply it locally.

When receiving an MSRP message with a Commit, the Participating Function shall:

- Forward the MSRP to the Conversation Focus as per [GSMA PRD-RCC.07]

When receiving an MSRP message with a Commit, the Conversation Focus shall:

- Validate the Commit as per 6.2.2.
    - If the Commit fails validation, the server shall send a negative IMDN as per section 7.7.2.1.
- Store the new GroupInfo and epoch authenticator.
- Create the new `ServerMlsRcsMessage` proto as per section 7.9.3
- Create a new MSRP message containing the `ServerMlsRcsMessage.`
- Forward to all members of the RCS Group (including the sender) as per [GSMA PRD-RCC.07].

When a client receives an MSRP with a Commit, it shall:

- Validate the Commit as per section 6.2.6.
- Apply the Commit locally.

9.5.2 Key Updates

Clients must update their keys in the MLS Group for Active Conversations. If the user has joined with a last resort KeyPackage, the initial key update shall follow the intervals in the first row of the following table. All other key updates shall follow the intervals in the second row.

The key update must occur after either the maximum number of days or the maximum number of outgoing messages, whichever comes first. It shall not happen earlier than the minimum number of days.

Key update scenario	Minimum # days	Maximum # days	Maximum # outgoing messages
Initial join with a last resort KeyPackage	0	1	30
Regular key update	7	30	50

Table 6: Key Update Intervals

Clients receiving Key Updates outside of the intervals above shall accept the Key Updates.

To create a key update, the client shall create an empty commit with UpdatePath or any Commit with an UpdatePath as per [RFC9420] and send the commit as per 9.5.1.

9.5.3 Certificate Update

Client must update their certificate in the MLS Group at least every 30 days for Active Conversations. In order to achieve that, the client shall:

- Update the ACS Signed Encryption Identity Proof via a config refresh as per [GSMA PRD-RCC.07].
- Create new KeyPackages as per section 5.
- Create an empty Commit with UpdatePath containing the new leaf from the newly created KeyPackage.
- Send the Commit as per section 9.5.1.

If the client has a certificate and/or ACS Signed Encryption Identity Proof that expires in more than 60 days, it may use the certificate and/or the ACS-Signed Encryption Identity Proof instead of calling the server.

9.5.4 KeyPackage Update

If another Client in the MLS Group has an expired certificate, or has not updated their leaf Keys in over 30 days, the client shall:

- Fetch KeyPackage(s) as per section 5.3 for all Clients with expired certificates or have not updated keys.
- Create a Commit that removes the old Leaf Nodes of the Client(s) and adds new leaves from the KeyPackage as per [RFC9420].
- Create a Welcome Message for all Clients replaced.
- Send the Commit as per section 9.5.1.

If there are no valid KeyPackages for any of the Clients, the client shall move the RCS Conversation to unencrypted as per section 11.2.

9.6 Server-Initiated User Removal

The Messaging Server can only initiate a user removal from the RCS Conversation if the user has lost RCS due to inactivity or deactivation. The Messaging Server shall:

- Create NOTIFYs to all members of the RCS Group as per [GSMA PRD-RCC.07] with the Participant(s) leaving.
    - o The server shall include `SIP;cause=410;text="Removed by server"` in the `<disconnection-info>` for the removed user.
- Not accept any Proposals or Commits as per 6.2.1 until a Commit for the Participant kicked arrives.

When a client receives a NOTIFY to remove Participant(s):

- Follow section 9.5.1 to create a Commit using the server_remove Proposal as per section 7.11.9.1. This Commit should remove all Clients associated with the Participant(s).Then create the new GroupInfo and send it to the Messaging Server.

    - If a Commit arrives that includes the removal of the Participant(s) before the client creates/sends the Commit, the client shall abandon Commit creation/sending.

- Wait for the Commit with the leave Proposal to arrive as an MSRP message to apply it locally.

9.7 Group Metadata Management

9.7.1 Group Icon and Subject

9.7.1.1 Encrypting Group Icon

To encrypt the group icon, the client shall:

- Encrypt the group chat Icon as per Annex C.2.
- Upload the encrypted content to the RCS File Transfer Server, following procedures outlined in [GSMA PRD-RCC.07].
- Create the `icon_commitment` extension as per 7.11.4.1.

9.7.1.2 Encrypting Group Subject

To encrypt the group subject, the client shall:

- Encrypt the group chat subject as per Annex C.2.
- Base-64 encode the encrypted subject and include it in the MSRP.
- Create the `subject_commitment` extension as per 7.11.6.1.

9.7.1.3 Group Icon and Subject Extensions for New Joiners

When a new member joins the group or creates a new group, the client adding the new member shall send the secrets for decrypting the group icon and subject in the Welcome message. The client shall:

- Create the `icon_key` extension containing the symmetric key as per 7.11.3.1.
- Create the `subject_key` extension containing the symmetric key as per 7.11.5.1.
- Add the `icon_key`, `icon_commitments`, `subject_key` and `subject_commitment` extension to GroupContext.

9.7.1.4 Changing the Group Icon

When changing the group icon, and the new icon is encrypted, the client shall:

- Follow the procedures in section 9.7.1.1
- Create a GroupContext Extensions Proposal that contains the `icon_commitment` extension and a Commit as per Section 12.1.7 of [RFC9420].
- Create an MLS PrivateMessage to transport the symmetric key as per section 7.8.1 with file_name is “group_icon”.
- Replace the old symmetric key in the `icon_key` extension of the local GroupInfo object with the new symmetric key.

- Create a separate GroupInfo object without the `icon_key` extension in it.
- Create a multi-part message consisting of the Commit, GroupInfo, Epoch Authenticator, and the PrivateMessage with the wireformat as defined in section 7.9.2.
- Send the multi-part message and the sanitized GroupInfo object to the server.

When receiving an encrypted group icon change request, the Conversation Focus shall:

- Validate the Commit as per 6.2.2.
    - If the Commit fails validation, the server shall send a negative IMDN as per section 7.7.2.1.
- Store the new icon, GroupInfo, and epoch authenticator.
- Create the new `ServerMlsRcsMessage` proto as per section 7.9.3
- Create a new MSRP message containing the `ServerMlsRcsMessage.`
- Forward the MSRP to all members of the RCS Group (including the sender) as per [GSMA PRD-RCC.07].
- Send a NOTIFY with the new icon as per [GSMA PRD-RCC.07].

When receiving an encrypted new icon, the client shall:

- Verify the correctness of the Commit message as per section 6.2.6.
- Use the Commit to derive the next-epoch application encryption key as per section 9.1 of [RFC9420].
- Decrypt the Application message using the next-epoch application encryption key to obtain the symmetric key.
- Download the encrypted icon from the RCS File Transfer Server and use the symmetric key to decrypt it as per Annex C.3.
- Verify the correctness of the symmetric key and the decrypted group icon using the data in the `icon_commitments` extension.
- Replace the symmetric key in the `icon_key` extension of the GroupInfo with the new symmetric key.

9.7.1.5 Changing the Group Subject

When changing the group subject, and the new subject is encrypted, the client shall:

- Follow the procedures in section 9.7.1.2.
- Create a GroupContext Extensions Proposal that contains the `subject_commitments` extension and a Commit as per Section 12.1.7 of [RFC9420].
- Create an MLS PrivateMessage to transport the symmetric key as per section 7.8.1with file_name is “group_subject”.
- Replace the old symmetric key in the `subject_key` extension of the local GroupInfo object with the new symmetric key.
- Create a separate GroupInfo object without the `subject_key` extension in it.

- Create a multi-part message consisting of the Commit, GroupInfo, Epoch Authenticator, and the PrivateMessage with the wireformat as defined in section 7.9.2.
- Send the multi-part message and the sanitized GroupInfo object to the server.

When receiving an encrypted group subject change request, the Conversation Focus shall:

- Validate the Commit as per 6.2.2.
    - If the Commit fails validation, the server shall send a negative IMDN as per section 7.7.2.1.
- Store the new subject, GroupInfo, and epoch authenticator.
- Create the new `ServerMlsRcsMessage` proto as per section 7.9.3
- Create a new MSRP message containing the `ServerMlsRcsMessage.`
- Forward the MSRP to all members of the RCS Group (including the sender) as per [GSMA PRD-RCC.07].
- Send a NOTIFY with the new subject as per [GSMA PRD-RCC.07].

When receiving the new encrypted subject, the client shall:

- Verify the correctness of the Commit message as per section 6.2.6.
- Use the Commit to derive the next-epoch application encryption key as per section 9.1 of [RFC9420].
- Decrypt the Application message using the next-epoch application encryption key to obtain the symmetric key.
- Decrypt the subject as per Annex C.3.
- Verify the correctness of the symmetric key and the decrypted group subject using the data in the `icon_commitments` extension.
- Replace the symmetric key in the `subject_key` extension of the GroupInfo with the new symmetric key.

10 MLS Group Recovery

Devices may encounter errors or disruptions that prevent them from encrypting or decrypting MLS messages. Examples include messages getting malformed during transfer, on-device storage issues, or a user switching from one device to another without the transfer of the cryptographic state, and many more. When those errors occur, the client needs to heal its MLS state to enable uninterrupted participation in encryption.

10.1 Self-Healing Mechanism

Self-Heal is a process of repairing the local state of the MLS group. It works by removing the previous Client owned by the user and re-adding a new representation of the Client (owned by the same user).

The Self-Heal procedure shall be initiated upon:

- Receiving an Application Message that can’t be decrypted.

- Receiving an MLS Control Message that can't be processed (e.g. failure to apply or rejection due to validation, as described in section 6.2.6).
- Failing to send an Application Message due to epoch authenticator mismatch.

The self-healing process involves fetching the latest GroupInfo from the Messaging Server (as defined in section 6.3.1) and creating an External Commit (as per [RFC9420]) based on the latest Group Info from the backend to resync its own Leaf Node.

If the MLS Group contains expired certificates, the client shall Self-Heal first. The client shall fetch new KeyPackage(s) for the Clients with expired certificates and replace them in the MLS Group via an update mechanism in section KeyPackage Update. If there are no available KeyPackages, the client shall move the Conversation to unencrypted as per section 11.2.

The application shall have an upper limit on the number of retries for self-healing in a given group. The application may use 5 as the maximum number of retries, and may not retry for more than a day. After any of the limits is reached, the procedure is considered failed. Fallback procedures defined in [GSMA PRD-RCC.71] shall apply.

Upon a failure of the Self-Heal procedure, the client shall create a new Era for the group, as per procedures in section 8.3. The new Era can only include Participants whose Clients were in the latest known MLS Group to the client self-healing. If there are Participants in the RCS Conversation that are not in the latest known MLS Group, the client shall remove those Participants from the RCS Conversation.

10.1.1 Self-Heal Procedure

The client shall:

- Request the GroupInfo from the Messaging Server, following the procedure from section 6.3.1.
- Create an External Commit (as per [RFC9420]):
  - If the client is replacing an old leaf with the same Participant Key: the client shall use a resync External Commit.
  - If the client is replacing an old leaf with a different Participant Key: the client must use resync External Commit to replace all the Clients of the Participant with KeyPackages that are signed with the new Participant Key.
  - If the client is adding itself to the MLS Group (because it is a new Client of the Participant): the client shall use the add External Commit to add itself to the MLS Group.
- Follow the procedures to upload the Commit to the Messaging Server, as per section 9.5.1, including any necessary retries of this procedure.

During the Self-Heal procedure, the client may attempt to decrypt incoming messages and may attempt to process incoming Commits. The client shall not initiate another Self-Heal on the same MLS Group while one is ongoing.

The client shall not send outgoing messages to the RCS Conversation while Self-Heal is ongoing.

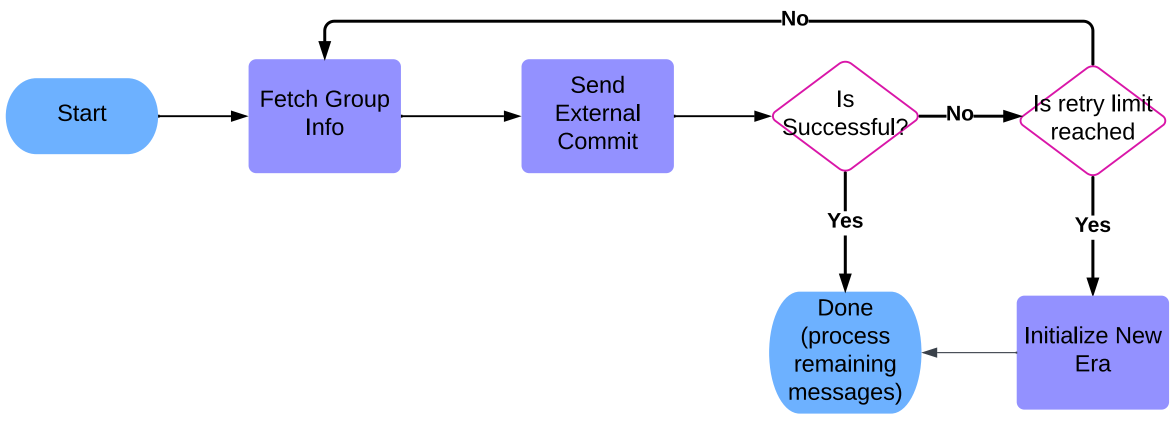

Figure 12: Illustration of the Self-Heal Procedure

10.2 Sending Fail to Decrypt (FTD)

After the Self-Heal Procedure is complete (as per section 0) the client shall:

- Send an FTD message for all Application Messages that it could not decrypt, one per message to the sender of the Original Message.
- Construct the FTD as per section 0.
- Send an FTD as an MSRP message as per [GSMA PRD-RCC.07]. The FTD shall be a Private-IM message. The FTD messages may be queued for automatic sending while the Self-Heal for the same MLS Group is ongoing.

10.3 Receiving an FTD message

In this section:

- Original Message refers to the message that failed to decrypt on the recipient.
- Re-Sent Message refers to the message that was re-encrypted and re-sent with a new Message-ID.

The FTD Message shall be ignored if:

- The Participant was not a member of the group at the time the Original Message was sent, or
- The Original Message was sent earlier than 30 days ago.
- The Participant Key of the Participant has changed (without the new key being part of the key roll as defined in section A.3.8.9) and the Original Message was sent over an hour ago.

An IMDN Negative-Delivery report shall be sent if an FTD was ignored. A Negative-Delivery shall follow [RFC5438] and shall include an mls:report extension as defined in section 7.7.2.2.

If the FTD was incorrectly signed, or the client cannot verify the signature of the FTD, the client shall self-heal itself.

Upon receipt of a valid FTD message, the client shall:

- Find the Original Message, if present. The client shall ignore any edits to the Original Message.
- Construct a ResentMessage struct containing the original as defined in section 7.5.4.
- Encrypt the struct using the one-to-one HPKE encryption algorithm defined in Annex C.4.1 to the FTD sender client.
- The HPKE ciphertext is then reencrytped to all other clients using MLS message per section 9.1.1.
- Set the original-message-id CPIM header.
- Perform a Key Update procedure as defined in section 9.5.2.

The sender shall stop a repeated chain of FTDs for the same Original Message after a maximum of 5 attempts. Fallback procedures defined in [GSMA PRD-RCC.71] shall apply.

10.4 Receiving a Re-Sent Message

The resent message is double encrypted and sent to all group members. When receiving the encrypted message, the client shall

- Decrypt the outer layer using the MLS message decryption per section 9.1.1.
- Process the decrypted payload:
    - Check the type field of the SecretPayload struct. If it is hpke_1_to_1_message type, it deserialize the decrypted payload to HPKEInnerEncapsulatedKeyAndCiphertext struct. Otherwise follow procedures in section 9.1.1
    - Check whether the receiver_leaf_index matches its leaf index in the MLS ratchet tree or not.
        - If it matches, then it performs the second decryption using the HPKE algorithm defined in Annex C.4.2.
        - Otherwise the message is ignored.
    - Use the original-message-id CPIM header as the message id.

The Re-Sent Message shall be ignored if the user never received the Original Message. A Negative-Delivery shall be sent.

If the user received the Original Message but has not successfully decrypted it, the client shall, upon decryption of the Re-Sent Message, display it to the user. The client may use the received timestamp of the Original Message to determine the correct placement of the message in the conversation.

The client shall send a delivery report for the new message as defined in section 7.7.1. The delivery report shall be sent for the Re-Sent Message ID.

A read report for the re-sent message shall be sent for the Re-Sent Message ID.

Any other features (such as incoming edit, delete, incoming reactions, or outgoing reactions) shall always refer to the RCS Message ID of the Original Message.

If an incoming reaction is received before the Re-Sent Message is received, the client shall apply the reaction to the future (not yet received) Re-Sent Message.

If an incoming edit of the Original Message is received before the Re-Sent Message is received, the client may ignore the Re-Sent Message.

If an incoming delete of the Original Message is received before the Re-Sent Message is received, the client should ignore/delete the Re-Sent Message.

10.5 Recovering Group Subject and Icon

When a group member receives an External Commit from a member and successfully applies it, they shall:

- Take the symmetric key from `icon_key` and `subject_key` in their local GroupInfo object.
- Create an MLS PrivateMessage that encrypts the symmetric keys using the application encryption key of the current epoch as described in section 6.3 of [RFC9420].
- Follow the procedures in section 9.1.1 to send the PrivateMessage.

When the member who sent the external commit receives an application message that contains the symmetric key, they shall:

- Decrypt the Application Message to get the symmetric key.
- Download the encrypted Group Chat Icon from the RCS File Transfer server, and use the symmetric key to decrypt it, as per Annex C.4.2C.3.
- Decrypt the group subject as per Annex C.3.
- Create an `icon_key` and `subject_key` extension as per sections 7.11.3.1 and 7.11.5.1.
- Add the `icon_key` and `subject_key` extensions to the locally maintained GroupInfo object.

11 Encryption Status Change

RCS Conversations can be in one of two states—encrypted or unencrypted—and those states may change during the life of the RCS Conversation. While the state is encrypted, the clients shall not send unencrypted messages in the RCS Conversation.

11.1 Unencrypted to Encrypted

The client shall, periodically request the capabilities of the Participants of an unencrypted Active RCS Conversation. The frequency of these requests is defined in section 11.1.1.

Once the client detects that all Participants are capable of MLS, the client shall attempt to resurrect the previous MLS Group (if present) for the RCS Conversation (section 11.1.2), provided that all certificates in that MLS Group are valid and non-expired (as per Annex A). If resurrection is not possible, the client shall create a new MLS Group Era (including first-time creation) as described in section 8.3.

11.1.1 Periodic Capability Refresh for Unencrypted Groups

The capabilities of Participants in Unencrypted Groups shall be refreshed only for Active Groups:

- The client shall refresh the Participants' capabilities at least once per month.
- It is recommended to refresh capabilities once per week.
- Capabilities may be refreshed when the conversation is opened.
- For scheduled refreshes, the client shall add a random backoff between attempts.

NOTE: Conversation moving from Inactive to Active is left for further specification.

11.1.2 Resurrecting former MLS Group

To resurrect the RCS Conversation as MLS, the client shall:

- Fetch the KeyPackages for the Participants that do not exist in the MLS Group or who have expired Certificates, and create Proposals to add those Participants' Clients to the MLS Group.
- Create proposals to remove the Clients of any Participants who are not part of the RCS Conversation from the MLS Group.
- Create and send a multi-part MSRP message with:
  - Welcome message to all added or re-added MLS members (if any), as described in section 9.2.
  - Single Commit with the Proposals (Add/Remove) and removal of the `end_mls` extension.

11.2 Encrypted to Unencrypted

The RCS Conversation shall migrate from encrypted to unencrypted in the following situations:

- An unsigned delivery report is received for an encrypted message.
- A Plaintext RCS Message is received.
- A new Participant is being added who does not support encryption.
- A capability check made for a Participant in the RCS Conversation did not return an MLS capability.

Before changing the encryption state of an RCS Conversation to unencrypted, the client may perform a capability check to determine if all users still support encryption and, if they do, the client may choose not to change the encryption status to unencrypted.

In order to move the encryption state of an RCS Conversation to unencrypted, the client sends a Commit to the current members of the MLS Group with an `end_mls` extension, as an MSRP message.

Annex A Certificate profiles

A.1 Root Certificate Profile

A.1.1 Version

Certificates shall be of type X.509 v3.

A.1.2 Serial Number

Certificate Authorities (CAs) shall generate non-sequential Certificate serial numbers greater than zero (0) and less than 2^159 containing at least 64 bits of output from a CSPRNG.

A.1.3 Signature Algorithm

All objects signed by a CA Private Key shall conform to these requirements on the use of the AlgorithmIdentifier or AlgorithmIdentifier-derived type in the context of signatures.

- The signatureAlgorithm field of a Certificate.
- The signature field of a TBSCertificate (for example, as used by a Certificate).

A.1.3.1 ECDSA

The CA shall use the appropriate signature algorithm and encoding based upon the signing key used.

If the signing key is P-384, the signature shall use ECDSA with SHA-384. When encoded, the AlgorithmIdentifier shall be byte-for-byte identical with the following hex-encoded bytes: `300a06082a8648ce3d040303`.

If the signing key is P-521, the signature shall use ECDSA with SHA-512. When encoded, the AlgorithmIdentifier shall be byte-for-byte identical with the following hex-encoded bytes: `300a06082a8648ce3d040304`.

A.1.4 Issuer

The encoded content of the Issuer Distinguished Name field of a Certificate shall be byte-for-byte identical with the encoded form of the Subject Distinguished Name field.

A.1.5 Validity

The maximum validity period (From RFC 5280, “the period of time from notBefore through notAfter, inclusive”) is 3652 days (approximately 10 years). The minimum validity period is 365 days (approximately 1 years). The notBefore date is the time of signing or a time no earlier than one day prior to the time of signing.

For the purpose of calculations, a day is measured as 86,400 seconds. Any amount of time greater than this, including fractional seconds and/or leap seconds, shall represent an additional day.

A.1.6 Subject

The Subject field shall contain a countryName, organizationName, and commonName, as specified with ObjectIdentifiers (OID [ITU-T X.680]), encoding requirements, and values as in table below. The Subject field may contain a stateOrProvinceName and localityName, as specified with OIDs, encoding requirements, and values as in table below. Other Attributes should not be included.

The subject shall encode the fields in the relative order as they appear in the table. Each Name must contain an RDNSequence. Each RelativeDistinguishedName must contain exactly one AttributeTypeAndValue. Each Name must not contain more than one instance of a given AttributeTypeAndValue across all RelativeDistinguishedNames.

Attribute	OID	Presence	Encoding Requirements	Max Length	Value
countryName	2.5.4.6	must	must use PrintableString	2	The two-letter ISO 3166-1 country code for the country in which the CA's place of business is located.
stateOrProvinceName	2.5.4.8	may	must use UTF8String or PrintableString	128	If present, the CA's state or province information.
localityName	2.5.4.7	may	must use UTF8String or PrintableString	128	If present, the CA's locality.
organizationName	2.5.4.10	must	must use UTF8String or PrintableString	64	The CA's name or DBA.
commonName	2.5.4.3	must	must use UTF8String or PrintableString	64	The contents should be an identifier for the certificate such that the certificate's Name is unique across all certificates issued by the issuing certificate.
Any other attribute		should not			

Table 7: Root Certificate Attributes

A.1.7 Subject Public Key

The following requirements apply to the subjectPublicKeyInfo field within a Certificate. No other encodings are permitted.

A.1.7.1 ECDSA

The CA shall indicate an ECDSA key using the id-ecPublicKey (OID: 1.2.840.10045.2.1) algorithm identifier. The parameters shall use the namedCurve encoding.

- For P-384 keys, the namedCurve shall be secp384r1 (OID: 1.3.132.0.34).
- For P-521 keys, the nameCurve shall be secp521r1 (OID: 1.3.132.0.35).

When encoded, the AlgorithmIdentifier for ECDSA keys shall be byte-for-byte identical with the following hex-encoded bytes:

- For P-384 keys, `301006072a8648ce3d020106052b81040022`.
- For P-521 keys, `301006072a8648ce3d020106052b81040023`.

A.1.8 Extensions

The following list of extensions is defined for root certificates. Any other extension not defined herein shall not be included.

A.1.8.1 Subject KeyIdentifier

This extension shall be present and shall not be marked critical. It should contain a value that is derived from the Public Key included in the Root Certificate.

A.1.8.2 Key Usage

This extension shall be present and shall be marked critical.

Bit positions shall be set for keyCertSign and cRLSign. Other bit positions shall not be set.

A.1.8.3 Certificate Policies

This extension should not be present and should not be marked critical. It shall include exactly one of the reserved policyIdentifiers documented herein. The certificatePolicies shall not include policyQualifiers.

A.1.8.4 Basic Constraints

This extension shall be present and shall be marked critical. The cA field shall be true. pathLenConstraint field should not be present.

A.1.8.5 Vendor ID

This extension shall be present and should not be marked critical.

This extension asserts the Vendor ID assigned by the GSMA to the vendor that owns this root.

```
id- gsmaRCSE2EE OBJECT IDENTIFIER  ::=
         { joint-iso-itu-t(2) international-organizations(23) gsma(146)
rcs(2) rcsE2EE (1)}
id-rcsVendorId OBJECT IDENTIFIER ::= { gsmaRCSE2EE 6 }
vendorId ::= INTEGER
```

A.2 Intermediate CA Certificate Profile

A.2.1 Version

Certificates shall be of type X.509 v3.

A.2.2 Serial Number

CAs shall generate non-sequential Certificate serial numbers greater than zero (0) and less than 2^159 containing at least 64 bits of output from a CSPRNG.

A.2.3 Signature Algorithm

All objects signed by a CA Private Key shall conform to these requirements on the use of the AlgorithmIdentifier or AlgorithmIdentifier-derived type in the context of signatures.

- The signatureAlgorithm field of a Certificate.
- The signature field of a TBSCertificate (for example, as used by a Certificate).

A.2.3.1 ECDSA

The CA shall use the appropriate signature algorithm and encoding based upon the signing key used.

If the signing key is P-384, the signature shall use ECDSA with SHA-384. When encoded, the AlgorithmIdentifier shall be byte-for-byte identical with the following hex-encoded bytes: `300a06082a8648ce3d040303`.

If the signing key is P-521, the signature shall use ECDSA with SHA-512. When encoded, the AlgorithmIdentifier shall be byte-for-byte identical with the following hex-encoded bytes: `300a06082a8648ce3d040304`.

A.2.4 Issuer

The encoded content of the Issuer Distinguished Name field of a Certificate shall be byte-for-byte identical with the encoded form of the Subject Distinguished Name field of the Issuing CA Certificate.

A.2.5 Validity

The maximum validity period (From RFC 5280, “the period of time from notBefore through notAfter, inclusive”) is 1827 days (approximately 5 years). The minimum validity period is 365 days (approximately 1 year). The notBefore date is the time of signing or a time no earlier than one day prior to the time of signing.

For the purpose of calculations, a day is measured as 86,400 seconds. Any amount of time greater than this, including fractional seconds and/or leap seconds, shall represent an additional day.

A.2.6 Subject

The Subject field shall contain a countryName, organizationName, and commonName, as specified with OIDs, encoding requirements, and values as in table below. The Subject field may contain a stateOrProvinceName and localityName, as specified with OIDs, encoding requirements, and values as in table below. Other Attributes should not be included.

The subject shall encode the fields in the relative order as they appear in the table. Each Name must contain an RDNSequence. Each RelativeDistinguishedName must contain exactly one AttributeTypeAndValue. Each Name must not contain more than one instance of a given AttributeTypeAndValue across all RelativeDistinguishedNames.

Attribute	OID	Presence	Encoding Requirements	Max Length	Value
countryName	2.5.4.6	must	must use PrintableString	2	The two-letter ISO 3166-1 country code for the country in which the CA's place of business is located.
stateOrProvinceName	2.5.4.8	may	must use UTF8String or PrintableString	128	If present, the CA's state or province information.
localityName	2.5.4.7	may	must use UTF8String or PrintableString	128	If present, the CA's locality.
organizationName	2.5.4.10	must	must use UTF8String or PrintableString	64	The CA's name or DBA.
commonName	2.5.4.3	must	must use UTF8String or PrintableString	64	The contents should be an identifier for the certificate such that the certificate's Name is unique across all certificates issued by the issuing certificate.
Any other attribute		should not			

Table 8: Intermediate Certificate Attributes

A.2.7 Subject Public Key

The following requirements apply to the subjectPublicKeyInfo field within a Certificate. No other encodings are permitted.

A.2.7.1 ECDSA

The CA shall indicate an ECDSA key using the id-ecPublicKey (OID: 1.2.840.10045.2.1) algorithm identifier. The parameters shall use the namedCurve encoding.

- For P-256 keys, the namedCurve shall be secp256r1 (OID: 1.2.840.10045.3.1.7).
- For P-384 keys, the namedCurve shall be secp384r1 (OID: 1.3.132.0.34).

When encoded, the AlgorithmIdentifier for ECDSA keys shall be byte-for-byte identical with the following hex-encoded bytes:

- For P-256 keys, `301306072a8648ce3d020106082a8648ce3d030107`.
- For P-384 keys, `301006072a8648ce3d020106052b81040022`.

A.2.8 Extensions

The following list of extensions are defined for intermediate CA certificates. Any other extension not defined herein should not be included.

A.2.8.1 Authority Key Identifier

This extension shall be present and shall not be marked critical. The keyIdentifier field shall be present and must be identical to the subjectKeyIdentifier field of the Issuing CA. authorityCertIssuer and authorityCertSerialNumber fields shall not be present.

A.2.8.2 Subject Key Identifier

This extension shall be present and shall not be marked critical. It should contain a value that is derived from the Public Key included in the intermediate CA Certificate.

A.2.8.3 Key Usage

This extension shall be present and shall be marked critical.

Bit positions shall be set for keyCertSign and cRLSign. Other bit positions shall not be set.

A.2.8.4 Certificate Policies

This extension may be present and should not be marked critical.

The CA may restrict the policies which this CA may issue. If the CA is policy-restricted, this extension shall include exactly one of the reserved policyIdentifiers documented herein and may contain one or more identifiers documented by the CA in its Certificate Policy (CP) and/or Certificate Practice Statement (CPS) and must not include the anyPolicy Policy Identifier.

The certificatePolicies shall not include policyQualifiers.

A.2.8.5 Basic Constraints

This extension shall be present and shall be marked critical. The cA field shall be true. pathLenConstraint field may be present.

A.2.8.6 Authority Information Access

This extension may be present. This extension shall not be marked critical.

When provided, every accessMethod shall have the UniformResourceIdentifier (URI) scheme HTTP. Other schemes or GeneralName types shall not be present.

The authorityInformationAccess extension may contain accessMethod values of type id-ad-ocsp that specifies the URI of the Issuing CA's OCSP responder.

The authorityInformationAccess extension may contain at least one accessMethod value of type id-ad-caIssuers that specifies the URI of the Issuing CA's Certificate.

Other accessMethod types shall not be present.

A.2.8.7 CRL Distribution Points

This extension shall be present and should not be marked critical. The CRL Distribution Points extension must contain at least one DistributionPoint; containing more than one is not recommended. The DistributionPointName must be a fullName with at least one GeneralName. All GeneralNames shall have the URI scheme HTTP. The reasons and cRLIssuer fields must not be present.

A.2.8.8 Extended Key Usage

This extension shall be present. The extension shall contain a single element, a KeyPurposeId with value

```
id-kp-rcsMlsClient OBJECT IDENTIFIER ::= { id-appleDraftRCSE2EE 3 }
```

A.3 Client Certificate Profile

A.3.1 Version

Certificates shall be of type X.509 v3.

A.3.2 Serial Number

CAs shall generate non-sequential Certificate serial numbers greater than zero (0) and less than 2^159 containing at least 64 bits of output from a CSPRNG.

A.3.3 Signature Algorithm

All objects signed by a CA Private Key shall conform to these requirements on the use of the AlgorithmIdentifier or AlgorithmIdentifier-derived type in the context of signatures.

- The signatureAlgorithm field of a Certificate.
- The signature field of a TBSCertificate (for example, as used by a Certificate).
- The participantSignatureAlgorithm field of a ParticipantInfo.

A.3.3.1 ECDSA

The CA shall use the appropriate signature algorithm and encoding based upon the signing key used.

If the signing key is P-256, the signature shall use ECDSA with SHA-256. When encoded, the AlgorithmIdentifier shall be byte-for-byte identical with the following hex-encoded bytes: `300a06082a8648ce3d040302`.

If the signing key is P-384, the signature shall use ECDSA with SHA-384. When encoded, the AlgorithmIdentifier shall be byte-for-byte identical with the following hex-encoded bytes: `300a06082a8648ce3d040303`.

A.3.4 Issuer

The encoded content of the Issuer Distinguished Name field of a Certificate shall be byte-for-byte identical with the encoded form of the Subject Distinguished Name field of the Issuing CA Certificate.

A.3.5 Validity

The maximum validity period (From RFC 5280, “the period of time from notBefore through notAfter, inclusive”) is 76 days. The minimum validity period is 45 days. The notBefore field must not be more than one day prior to the time of issuance and should be at least one hour prior to the time of issuance.

For the purpose of calculations, a day is measured as 86,400 seconds. Any amount of time greater than this, including fractional seconds and/or leap seconds, shall represent an additional day. For this reason, Subscriber Certificates should be issued with a notAfter that is not more than 6,566,340 seconds after the notBefore.

The CA or Rregistration Authority (RA) shall validate all identity attributes of the Subject and SubjectAlternativeName to be included in the Certificate. If the evidence has an explicit validity period, the CA shall verify that the time of the identity validation is within this validity period. In context this can include the notBefore and notAfter fields of a digital signature Certificate or the date of expiry of an identity document or the expiry of the ACS Signed Tuple (section 7.12). The CA or RA shall retain information sufficient to evidence the fulfillment of the identity validation process and the verified attributes.

A.3.6 Subject

The Subject Name should contain an id-clientIdentifier Attribute type with an RcsMlsClientIdentifer value containing a UUID (Universally Unique Identifier) conforming to RFC 9562 version 4 created with random bytes and using a UTF8String encoding. The Subject Name may contain a CommonName containing the RcsMlsClientIdentifer. The Subject Name must contain either a id-clientIdentifier type and a CommonName and must not contain both a id-clientIdentifier type and a CommonName. The Subject Name shall not contain any other fields.

```
id-clientIdentifier AttributeType ::= { id-appleDraftRCSE2EE 1 }
RcsMlsClientIdentifier ::= UTF8String (SIZE (36))
```

A.3.7 Subject Public Key

The following requirements apply to the subjectPublicKeyInfo field within a Certificate. No other encodings are permitted.

A.3.7.1 ECDSA

The CA shall indicate an ECDSA key using the id-ecPublicKey (OID: 1.2.840.10045.2.1) algorithm identifier. The parameters shall use the namedCurve encoding.

- For P-256 keys, the namedCurve shall be secp256r1 (OID: 1.2.840.10045.3.1.7).
- For P-384 keys, the namedCurve shall be secp384r1 (OID: 1.3.132.0.34).

When encoded, the AlgorithmIdentifier for ECDSA keys shall be byte-for-byte identical with the following hex-encoded bytes:

- For P-256 keys, `301306072a8648ce3d020106082a8648ce3d030107`.
- For P-384 keys, `301006072a8648ce3d020106052b81040022`.

A.3.8 Extensions

The following list of extensions are defined for leaf certificates. Any other extension not defined herein should not be included.

A.3.8.1 Authority Key Identifier

This extension shall be present and shall not be marked critical. The keyIdentifier field shall be present and must be identical to the subjectKeyIdentifier field of the Issuing CA. authorityCertIssuer and authorityCertSerialNumber fields shall not be present.

A.3.8.2 Subject Key Identifier

This extension should be present and shall not be marked critical. It should contain a value that is derived from the Public Key included in the Client Certificate.

A.3.8.3 Key Usage

This extension shall be present and should be marked critical.

Bit positions shall be set for digitalSignature. Other bit positions shall not be set.

A.3.8.4 Certificate Policies

This extension shall be present and should not be marked critical. It shall include exactly one of the reserved policyIdentifiers documented herein and may contain one or more identifiers documented by the CA in its CP and/or CPS. The certificatePolicies shall not include policyQualifiers.

```
id-RCSE2EEPolicyId OBJECT IDENTIFIER ::= { id-appleDraftRCSE2EE 2 }
```

A.3.8.5 Subject Alternative Name

This extension shall be present. This extension should not be marked critical.

The Subject Alternative Name shall contain at least one GeneralName of type UniformResourceIdentifier (URI). The URI shall be a Global Number tel URI per [RFC3966]. The URI shall not contain visual separators. The Global Number shall not contain any parameters, extensions, or isdn-subaddress.

Multiple GeneralNames are supported for re-numbering situations.

A.3.8.6 Basic Constraints

This extension may be present. The cA field shall not be true. pathLenConstraint field shall not be present.

A.3.8.7 Extended Key Usage

This extension shall be present. The extension shall contain a single element, a KeyPurposeId with value

```
id-kp-rcsMlsClient OBJECT IDENTIFIER ::= { id-appleDraftRCSE2EE 3 }
```

A.3.8.8 Authority Information Access

This extension should be present. This extension shall not be marked critical.

When provided, every accessMethod shall have the URI scheme HTTP. Other schemes or GeneralName types shall not be present.

The authorityInformationAccess extension should not contain accessMethod values of type id-ad-ocsp that specifies the URI of the Issuing CA's OCSP responder.

The authorityInformationAccess extension should contain at least one accessMethod value of type id-ad-caIssuers that specifies the URI of the Issuing CA's Certificate.

Other accessMethod types shall not be present.

A.3.8.9 Participant Information

This extension shall be present. This extension shall be marked critical.

The Participant Information extension binds this client certificate to a particular participant in the RCS/MLS ecosystem.

Extension ASN.1 definition:

```
id-participantInformation OBJECT IDENTIFIER ::=
                                    { appleDraftRCSE2EE 4 }
ParticipantInformation ::= SEQUENCE {
    vendorId                           INTEGER,
    participantSignatureValidity       Validity,
    participantSignatureAlgorithm      AlgorithmIdentifier,
    participantSignatureValue          BIT STRING,
    participantKey                     SubjectPublicKeyInfo OPTIONAL,
    participantKeyRolls   [0] IMPLICIT SEQUENCE SIZE (1..5) OF
                                       ParticipantKeyRoll OPTIONAL
}

Validity ::= SEQUENCE {
    notBefore      Time,
    notAfter       Time }

Time ::= CHOICE {
    utcTime        UTCTime,
    generalTime    GeneralizedTime }

ParticipantKeyRoll ::= SEQUENCE {
    participantRollSignatureAlgorithm      AlgorithmIdentifier,
    participantRollSignatureValue          BIT STRING,
    oldParticipantKey                      SubjectPublicKeyInfo
}
```

The vendorId shall match the vendorId asserted in the root certificate.

The participantSignatureValidity shall have a maximum validity period (from [RFC5280], “the period of time from notBefore through notAfter, inclusive”) of 76 days. The participantSignatureValidity shall have a minimum validity period of 45 days. The notBefore field must not be more than one day prior to the time of signature creation and should be at least one hour prior to the time of signature creation. The choice of Time format shall be conformant with section 4.1.2.5 of [RFC5280].

The particpantSignatureAlgorithm and participantRollSignatureAlgorithm shall be one of the allowed AlgorithmIdentifiers from Signature Algorithm.

The participantKey is the key that produced the participantSignatureValue. The participantKey shall be one of the allowed algorithms in Subject Public Key. If omitted, the participantKey is the same as the Subject Public Key. The issuer shall validate the participantKey for the subject and subjectAltName of the certificate for lifetime of issuance.

The participantSignatureValue contains a digital signature computed upon the following ASN.1 DER-encoded structure, matching fields from the tbsCertificate:

```
tbsParticipantInfo ::= SEQUENCE {
    subject                            Name,
    vendorId                           INTEGER,
    participantSignatureValidity       Validity,
    subjectPublicKeyInfo               SubjectPublicKeyInfo,
    subjectAltName                     SubjectAltName }
```

The tbsParticipantInfo contains the Subject, Subject Public Key, and Subject Alternative Name (byte-for-byte matching the fields in the tbsCertificate) and the vendorId and participantSignatureValidity (byte-for-byte matching the fields in the ParticipantInfo). The participant creates this signature upon user approval of a client that will use the subject public key, subject, and subject alternative name. The issuer shall validate the participantSignatureValue is valid before issuing the certificate.

The SEQUENCE OF ParticipantKeyRoll objects forms a chain of continuity across ParticipantKeys as they are changed by Participants. In the first ParticipantKeyRoll, the participantRollSignatureValue is a signature over the participantKey by the oldParticipantKey. For each subsequent ParticipantKeyRoll item the participantRollSignatureValue is a signature over the oldParticipantKey in the prior ParticipantKeyRoll with the oldParticipantKey contained in that ParticipantKeyRoll item.

A.3.8.10 ACS Participant Information

This extension shall be present.

The ACS Participant Information extension value is an OCTET STRING of the encoded SignedEncryptionIdentityProof (section 7.12).

Extension ASN.1 definiton:

```
id-acsParticipantInformation OBJECT IDENTIFIER ::=
                                    { appleDraftRCSE2EE 5 }
```

A.4 Certificate Validation Procedures

A.4.1 Client Validation

Client will validate the credentials on

1. Query receipt from Home KDS
2. All commits with special handling required for

    a) Update
    b) Re-sync

Note that except where specified, commits signed by expired (or otherwise invalid) credentials will be rejected; however, the flows below allow a participant with an invalid credential to update to a valid credential.

A.4.1.1 Default Validation Requirements

Clients verifying RCS E2EE credentials must in all flows:

1. Verify that the client credential properly chains by verifying signatures to a provided intermediate CA. (see [RFC5280]). Including the following detailed client certificate verification items:

    a) Ensure the extendedKeyUsage contains the id-kp-rcsMlsClient key purpose in the leaf certificate and no others.
    b) Ensure the key usage has the digitalSignature bit set.
    c) Verify the correctness of the ParticipantInfo Extension:

        i. Construct the tbsParticipantInfo from the certificate
        ii. Verify the participantSignatureValue with the participantKey (or if not present, the subject public key info of the certificate) against the constructed tbsPartipantInfo using the specified participantSignatureAlgorithm.
        iii. Verify all ParticipantKeyRoll items by verifying the participantRollSignatureValue with the oldParticipantKey against the participantKey (or prior oldParticipantKey) using the specified participantRollSignatureAlgorithm

    d) Certificate lifetime is 76 days or fewer.
    e) That the certificate and ParticipantInfo are not expired, except as in detailed in specific flows below.

2. Verify that the provided intermediate CA properly chains to a trusted root certificate (see [RFC5280]). Including the following detailed CA verification items:

    a) That the CA certificate is not revoked per the CRL DP.
    b) That the BasicConstraints extension has cA field equal to TRUE (where TRUE is DER-encoded, meaning 0xff)
    c) Ensure the key usage has the keyCertSign bit set.

   d) Ensure the extendedKeyUsage contains the id-kp-rcsMlsClient key purpose in the leaf certificate and no others.
   e) Certificate lifetime is 1827 days or fewer.
   f) That the certificate is not expired, except as in detailed in specific flows below.

3. Verify that the VendorID in the ParticipantInfo matches the VendorID in the root certificate.

A.4.1.2 Query Validation

In addition to the Default Validation, on query response from the Home KDS the client will verify:

1. that the SAN URI matches the expected MSISDN of the RCS participant queried.
2. that the client certificate is not expired and has at least 30 days left before expiration.

A.4.1.3 Add Proposal and Commit

Clients must not issue Add Proposals with less than 30 days remaining before the expiry of the credential in the updated Leaf Node.

A.4.1.4 Welcome Package

Upon receipt of the Welcome Package, a client should verify all of the certificates in the Leaf Nodes of the group.

A.4.1.5 Self Update Commit

In addition to the Default Validation Requirements, on a Commit with an Update Path that changes the committer’s LeafNode with a new Certificate the client will verify:

1. That the new LeafNode credential has at least one matching SAN URI with the existing certificate in the LeafNode.
2. That the new certificate an issuance date *after* the existing certificate.
3. Expiration of the existing LeafNode certificate is not checked in this flow (allowing a participant with an expired certificate to update to a new, non-expired certificate).

Clients must not issue self update commits with less than 30 days remaining before the expiry of the credential in the updated Leaf Node.

A.4.1.6 Resync Commit

In addition to the Default Validation Requirements, on a Resync Commit, the client will verify:

1. That the certificate for the participant the Remove Proposal has at least one matching SAN URI with the certificate in the Add Proposal.
2. That the external commit is signed by the certificate in the Add proposal.
3. That the client certificate in the Add Proposal has an issuance date *after* the certificate in the Removed node.
4. Expiration of the certificate in the Remove Proposal is not checked.

Clients must not issue Add Proposals with less than 30 days remaining before the expiry of the credential in the updated Leaf Node.

A.4.2 KDS Validation

KDSs will validate the credentials on

1. KeyPackage update
2. Query receipt from a peer KDS

A.4.2.1 Default Validation Requirements

KDSs verifying RCS E2EE credentials must in all flows:

1. Verify that the client credential properly chains by verifying signatures to a provided intermediate CA. (See RFC 5280.) Including the following detailed client certificate verification items:

    a) Verify that the client certificate has a remaining lifetime of at least 30 days.

2. Verify that the provided intermediate CA properly chains to a root certificate on the GSMA Trust List. (See RFC 5280.) Including the following detailed CA verification items:

    a) That the CA certificate is not revoked per the CRL DP.

3. That the certificate and ParticipantInfo are not expired.
4. Verify that the VendorID extension in the root certificate asserts the KDS vendor's ID for KeyPackage update or the same vendor ID as the replying KDS.

A.4.2.2 KDS Query Fulfillment Behavior

KDSs must not return KeyPackages to a query where the credential has less than 30 days before expiry. This requirement prevents situations where the KDS fulfills a query without the client being able to perform the Add proposal and have delivery of that proposal within the lifetime of the certificate.

A.4.3 RCS SPN Validation

RCS SPN shall validate the credentials on

1. All commits
2. Within all proposals

A.4.3.1 Default Validation Requirements

RCS SPN verifying RCS E2EE credentials shall in all flows:

1. Verify that the client credential properly chains by verifying signatures to a provided intermediate CA. (See RFC 5280.) Including the following detailed client certificate verification items:

    a) That the certificate is not expired and has 30 days before expiry, except as in detailed in specific flows below.

2. Verify that the provided intermediate CA properly chains to a trusted root certificate. (See RFC 5280.) Including the following detailed CA verification items:

    a) That the CA certificate is not revoked per the CRL DP.

   b) That the certificate is not expired.

3. Verify the ACSParticipantInfo extension:

   a) Verify that the participantKey matches the ParticipantKey in the ParticipantInfo extension
   b) Verify that the VendorID asserted the root certificate matches the VendorID asserted in the ParticipantInfo and ACSParticipantInfo
   c) Verify the MSISDNs asserted in the ACSParticipantInfo extension match the SAN URIs in the client certificate.
   d) Verify that the MSISDNs asserted in the ACSParticipantInfo match the expected MSISDN for the RCS channel.
   e) Verify the signature with one of the ACS certificates.
   f) Verify that the ACSParticipantInfo is not expired.

A.4.3.2 Self Update Commit

In addition to the Default Validation Requirements, on a Commit with an Update Path that changes the committer's LeafNode with a new Certificate the RCS SPN will verify:

1. That the new LeafNode credential has at least one matching ACSParticipantInfo MSISDN with existing certificate in the LeafNode.
2. That the new certificate has an issuance date *after* the existing certificate.
3. Expiration of the existing LeafNode certificate is not checked in this flow (allowing a participant with an expired certificate to update to a new, non-expired certificate).

A.4.3.3 Resync Commit

In addition to the Default Validation Requirements, on a Resync Commit, the client will verify:

1. That the certificate for the participant the Remove Proposal has at least one matching ACSParticipantInfo MSISDN with the certificate in the Add Proposal.
2. That the external commit is signed by the certificate in the Add proposal.
3. That the client certificate in the Add Proposal has an issuance date after the certificate in the Removed node.
4. Expiration of the certificate in the Remove Proposal is not checked.

Annex B Inter-KDS Interface

A The schema for the Inter-KDS Interface is written in gRPC.

Inter-kds.proto:

```
syntax = "proto3";
import "google/protobuf/timestamp.proto";

package kds_proto;

message RequestHeader {
  // UUID used to identify the request. Used for debugging and tracing
  // only.
  uint64 request_id = 1;
}

enum ResponseStatus {
  UNKNOWN_STATUS = 0;
  OK = 1;
  NOT_FOUND = 2;
  MALFORMED_ID = 3;
  UNSUPPORTED_CIPHER_SUITE = 4;
  NEWER_ENROLMENT_EXISTS = 5;
}

message Identifier {
  enum Type {
    UNKNOWN = 0;
    // Specified as E.164 format.
    PHONE_NUMBER = 1;
  }
  Type type = 1;
  string identifier = 2;
}

message GetSupportedCipherSuitesRequest {
  RequestHeader header = 1;
  repeated Identifier participant_id = 2;
}

message ParticipantCipherSuite {
  ResponseStatus status = 1;
  Identifier participant_id = 2;
  // All the Cipher Suites supported by the participant.
  // The Cipher Suites are defined in [RFC 9420]
  repeated uint32 cipher_suite = 3;
}

message GetSupportedCipherSuitesResponse {
  repeated ParticipantCipherSuite participant_cipher_suite = 1;
}

message GetKeyPackagesRequest {
  RequestHeader header = 1;
  // The highest common Cipher Suite supported by all the participants.
  uint32 cipher_suite = 2;
  repeated Identifier participant_id = 3;
}

message KeyPackage {
  // The client id that is contained in the key package. Must be a
  // globally unique identifier.
  string client_id = 1;
  // As defined in [RFC9420]. No encoding, padding or escaping
  // applied.
  bytes key_package = 2;
}
```

```
message ParticipantKeyPackage {
  ResponseStatus status = 1;
  Identifier participant_id = 2;
  repeated KeyPackage key_package = 3;
}

message GetKeyPackagesResponse {
  repeated ParticipantKeyPackage participant_key_package = 1;
}

message ParticipantRegistration {
  Identifier participant_id = 1;

  // Timestamp of when the user is registered. If the Participant on the
  // local KDS is newer than this timestamp, it may ignore this request.
  google.protobuf.Timestamp enrolment_time = 2;
}

// Sending a list of participants that have registered with the calling
// KDS. Can send a maximum of 50 participants in a single request.
message ParticipantRegistrationNotificationRequest {
  RequestHeader header = 1;
  repeated ParticipantRegistration participant_registration = 2;
}

message ParticipantNotificationStatus {
  ResponseStatus status = 1;
  Identifier participant_id = 2;
}

message ParticipantRegistrationNotificationResponse {
  repeated ParticipantNotificationStatus participant_notification_status =
1;
}

service InterKdsService {
  // Fetch the Cipher Suites supported by the Participants.
  rpc GetSupportedCipherSuites(GetSupportedCipherSuitesRequest)
      returns (GetSupportedCipherSuitesResponse);

  // Fetch KeyPackages for specified Participants and Cipher Suite
  rpc GetKeyPackages(GetKeyPackagesRequest) returns
(GetKeyPackagesResponse);

  // Updates this KDS instance that the Participant has registered with
  // another KDS instance.
  rpc
NotifyParticipantRegistration(ParticipantRegistrationNotificationRequest)
      returns (ParticipantRegistrationNotificationResponse);
}
```

Annex C Cryptographic Operations

In this section, all cryptographic primitives, including hash functions, KDF, and AEAD are defined by the Cipher Suite chosen by the group if not explicitly specified.

The symbol || denotes the append operation.

The symmetric key is an AEAD key of the AEAD algorithm defined by the group Cipher Suite.

UInt32 is a 32 bit unsigned integer that is encoded in Big Endian representation.

== denotes equality checking

= denotes an assignment

C.1 Creating a Commitment for a Value

Given a value V and a label L, a Commitment for the Value V is computed as follows:

```
Commitment = Hash(HashContent)
struct {
  opaque label<V>;
  opaque value<V>;
} HashContent
```

And the fields are set to:

```
label = L;
value = V;
```

C.2 Encrypting a File

HKDF references HKDF<SHA256>.
HMAC references HMAC<SHA256> with a 256-bit tag output.
M is a byte vector of length <2^31 bytes representing the file to be encrypted.
K is a 256 bit randomly chosen key used for one and only one file.
Info denotes the original filename and is a byte vector of length < (2^16) - 1 bytes.

```
Encrypt(Key, M, Info)

Let ZV be a vector of 0 bytes of length 4.
Let IV be a 96 bit random nonce
Let salt =
0x3243f6a8885a308d313198a2e03707344a4093822299f31d0082efa98ec4e6c8
 be the 256 bit hex representation of pi.

Let k_enc||k_hmac = HKDF (Key, salt, Info) where each of k_enc and k_hmac
are exactly 256 bits long.

Variable messageLength is a UInt32 value
Let messageLength = LengthInBytes(M)
```

```
Variable paddingLength is a UInt32 value
Let paddingLength = Padme(messageLength)- messageLength

Let pad be a vector of 0 bytes that that has length equal to paddingLength
Let paddedMessage = messageLength||paddingLength||M||pad
Let IV’ = IV||ZV
Let Ciphertext = AES_CTR_ENC (k_enc, IV’, paddedMessage)
Let Tag = HMAC(k_hmac, IV||Ciphertext)  (Tag is exactly 256 bits)

Output (IV, Ciphertext, Tag)
```

C.3 Decrypting a File

K is a 256 bit random key.
IV is a 96 bit nonce.
FileInfo, containing FileMetadata and FileEncryptionInfo, is received separately.
Ciphertext is the encrypted file which is a byte vector of size < (2^32)- 1.

Key = FileEncryptionInfo.key_material
IV = FileEncryptionInfo.iv
Tag = FileEncryptionInfo.hmac_tag
Info = FileMetadata.file_name

```
Decrypt(Key, Info, IV, Ciphertext, Tag).
Let ZV be a vector of 0 bytes of length 4.
Let salt =
0x3243f6a8885a308d313198a2e03707344a4093822299f31d0082efa98ec4e6c8 be the
256 bit representation hex of pi.

Verify LengthInBytes(C)>=8 else output error: ciphertext too small

Let k_dec||k_hmac = HKDF(Key, salt, Info) where each of k_dec and k_hmac
are exactly 256 bits long.
Let ComputedTag = HMAC(k_hmac, IV||Ciphertext) where ComputedTag is 256
bits long.
Verify that ComputedTag == Tag otherwise output error: Validation failure.

Let IV’ = IV||ZV
Let plaintext = AES_CTR_DEC(k_dec, IV’, Ciphertext)

Variable messageLength is a UInt32 value
Let messageLength = plaintext[0..3]
Verify messageLength<2^31 else output error: message too long

Variable paddingLength is a UInt32 value
Let paddingLength = plaintext[4..7]
Let paddedMessage = plainText[8..]
Verify LengthInBytes(Ciphertext) == (8 + messageLength + paddingLength)
else  output error: ciphertext decoding
Verify paddingLength == (Padme(messageLength) - messageLength) else output
error: ciphertext decoding
```

```
Let message = paddedMessage[0.. messageLength -1]
Let pad = paddedMessage[messageLength ..]
Verify pad byte vector is all 0s vector else output error: ciphertext
decoding

Output (message)
```

C.4 One to one HPKE Encryption for Re-Sent Messages

This section defines an encryption schema to a single client in the MLS group using the node HPKE Key.

It involves two layers of encryption: inner encryption using the recipient Leaf Node HPKE key followed by outer encryption using the group MLS message encryption.

```
Struct {
  opaque discriminant[26] = "MLSv1 RCSv1 1:1 Encryption" (UTF8 Encoded);
  opaque group_id[32];
  uint64 epoch;
  uint32 era;
  uint32 receiver_leaf_index;
  uint32 sender_leaf_index;
} OuterInfo;

Struct {
  opaque group_id[32];
  uint64 epoch;
  uint32 era;
  uint32 receiver_leaf_index;
  uint32 sender_leaf_index;
} HPKEAADStruct;
```

C.4.1 Encryption

The OuterInfo struct and recipient Leaf Node public HPKE key are used to set up the HPKE context and key per [RFC9180] and then the context is used to encrypt the serialized ResentMessage struct as defined in section 7.5.7.

```
HPKEEncapsulatedKey, ContextS = SetupBaseS(pkR, OuterInfo);
HPKECiphertext = ContextS.Seal(HPKEAADStruct, message)
```

The key and the ciphertext are then wrapped in HPKEInnerEncapsulatedKeyAndCiphertext struct.

```
Struct {
  uint32 receiver_leaf_index;
  opaque HPKEEncapsulatedKey<V>;
  opaque HPKECiphertext<V>;
  opaque original_message_hmac<V>;
} HPKEInnerEncapsulatedKeyAndCiphertext;
```

A SecurePayload struct is constructed as defined in section 7.5.4to include the serialized HPKEInnerEncapsulatedKeyAndCiphertext struct, and the type field of the SecurePayload is set to hpke_1_to_1_message. The SecurePayload will be encrypted using the normal MLS message encryption.

C.4.2 Decryption

After the client decrypts an incoming MLS message and the SecurePayload type is hpke_1_to_1_message, and the receiver leaf_leaf_index matches the client leaf index in the MLS ratchet tree, it constructs the HPKE context and decrypts the payload.

```
contextR = SetupBaseR(HPKEEncapsulatedKey, skR, info):
message = context.Open(HPKEAADStruct, HPKECiphertext);
```

C.5 Identity Verification Code

Given a pair of Participants with their MSISDNs and Participants Keys, calculate the following values:

```
struct {
    opaque msisdn<V>;
    opaque participant_identity_public_key<V>;
} User;

// Users are sorted by MSISDN ascending
struct {
   User first_user;
   User second_user;
} UserPairKeys;

string generate_code(UserPairKeys user_pair_keys) {
      if(user_pair_keys.second_user.msisdn >
      user_pair_keys.first_user.msisdn) { swap(&user_pair_keys.first_user,
      &user_pair_keys.second_user); }
      hash = SHA512(user_pair_keys)
      // returns ~265 bit representation
      return HASH_TO_DIGITS(/*digit_count=*/80, hash);
}
identity_verification_code = generate_code(user1, user2);
```

The `identity_verification_code` is the value shown to the user.

Annex D Document Mangement

D.1 Document History

Version	Date	Brief Description of Change	Approval Authority	Editor / Company
1.0	28 February 2025	Initial version	ISAG	Basel Al-Naffouri / Google

D.2 Other Information

Type	Description
Document Owner	RCS Group
Editor / Company	Basel Al-Naffouri / Google