Rich Communication Suite – RCS gRPC
Version 1.0
17 July 2026

Security Classification: Non-Confidential

Copyright Notice

Disclaimer

Compliance Notice

The information contain herein is in full compliance with the GSMA Antitrust Compliance Policy.

This Permanent Reference Document is classified by GSMA as an Industry Specification, as such it has been developed and is maintained by GSMA in accordance with the provisions set out GSMA AA.35 - Procedures for Industry Specifications.

Table of Contents

1 Introduction

1.1 Overview

RCS gRPC is an optimised connectionless protocol for Rich Communication Suite (RCS). This protocol can be deployed as an alternative to the IMS-based service defined in [[GSMA PRD-RCC.07]]. RCS gRPC minimises the number overall transmissions and reduces the processing required by all RCS endpoints. These optimisations are realised by using a combination of Transport Layer Security (TLS) or Quick UDP Internet Connections (QUIC) and protocol buffers.

RCS gRPC is fully interoperable with RCS SIP/MSRP-based systems and supports key RCS features including messaging, capability discovery, spam reporting and end-to-end encryption (E2EE).

1.2 Scope

This document defines how RCS uses the gRPC protocol in addition to SIP/MSRP protocol for both UNI and NNI. Along with the necessary SIP/MSRP to RCS gRPC interworking, translations and procedures.

1.3 Definition of Terms

Term	Description
APN	Access Point Name
CPIM	Common Profile for Instant Messaging
E2EE	End-to-End encryption
gRPC	GRPC Remote Procedure Calling
gUNI	gRPC User Network Interface
gNNI	gRPC Network-to-Network Interface
Group Chat Session Identity	Public Service Identity assigned to a Group Chat. It is assigned by the Messaging Server and used by the client for addressing the Messaging Server. Note: Also known as conference URI.
HTTP	Hypertext Transfer Protocol
IMDN	Instant Message Disposition Notification
IMS	IP Multimedia Subsystem
IWF	Interworking Function
MLS	Messaging Layer Security
MSISDN	Mobile Station International Subscriber Directory Number
MSRP	Message Session Relay Protocol
NNI	Network-to-Network Interface
Push Notification	A notification to the RCS client, when not connected to the RCS Service Provider, that a new incoming communication event (e.g. new Chat Message, File Transfer, Audio Message, Message or File Transfer Status / State notifications) is available for the user. The Push Notification is sent by the RCS Service Provider to wake up the RCS

Term	Description
	client for establishing a connection to the RCS Service Provider for the new incoming communication event.
Push Notification Mechanism	A mechanism that enables the RCS Service Provider to enable non-persistent connections between the RCS Service Provider and the RCS client (in contrast to the mechanism that uses persistent connections) for communication event delivery. The mechanism is not visible to the user.
Push Service Provider	An entity which provides Push Notification Mechanism to the RCS client.
QUIC	Quick UDP Internet Connections
RFC	Request For Comments
SIP	Session Initiation Protocol
SPN	Service Provider Network
SRV	Service Record
tel URI	telephone Uniform Resource Identifier
TLS	Transport Layer Security
UE	User Equipment
UNI	User Network Interface
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier

1.4 References

Ref	Doc Number	Title
1	[RFC2119]	"Key words for use in RFCs to Indicate Requirement Levels", S. Bradner, March 1997 http://www.ietf.org/rfc/rfc2119.txt
2	[RFC8174]	Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words https://www.rfc-editor.org/info/rfc8174
3	[GSMA PRD-RCC.71]	GSMA PRD RCC.71 RCS Universal Profile Service Description Document, Version 4.1, 17 July 2026 http://www.gsma.com/
4	[GSMA PRD-RCC.07]	GSMA PRD RCC.07 version 18.0 - "Rich Communication Suite - Advanced Communications Services and Client Specification" 17 July 2026 http://www.gsma.com/
4	[GSMA PRD-RCC.14]	GSMA PRD RCC.14 HTTP-based Service Provider Device Configuration, Version 13.0, 19 February 2026 http://www.gsma.com/
5	[gRPC]	gRPC Remote Procedure Calling https://grpc.io/
6	[RFC9000]	QUIC: A UDP-Based Multiplexed and Secure Transport http://tools.ietf.org/html/rfc9000

Ref	Doc Number	Title
7	[RFC9505]	Network Time Protocol Version 4: Protocol and Algorithms Specification https://www.rfc-editor.org/rfc/rfc5905.html
8	[GSMA PRD-RCC.16]	GSMA PRD RCC.16 Rich Communication Suite – End-to-End Encryption Specification, Version 4.0, 17 July 2026 http://www.gsma.com/
9	[RFC9420]	The Messaging Layer Security (MLS) Protocol https://www.rfc-editor.org/rfc/rfc9420.html
10	[GSMA PRD-IR.67]	GSMA PRD IR.67 - "DNS and ENUM Guidelines for Service Providers and GRX and IPX Providers" Version 15.0, 29 June 2018 http://www.gsma.com/
11	[RFC3261]	SIP (Session Initiation Protocol) IETF RFC http://tools.ietf.org/html/rfc3261
12	[GSMA PRD-RCC.11]	GSMA PRD RCC.11 RCS Endorsement of OMA CPM 2.2 Conversation Functions, Version 13.0, 28 Feb 2025 http://www.gsma.com/

1.5 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 RFC 2119 [1] and clarified by RFC 8174 [2] when, and only when, they appear in all capitals, as shown here.

2 Architecture

RCS gRPC defines a new interface to the SPN for UNI and NNI that includes several gRPC [gRPC] services. SPNs expose RCS gRPC services for RCS clients (called gUNI) and other SPNs (called gNNI). Those RCS gRPC services are defined separately for gUNI and gNNI. The list of services is outlined in Table 1.

Service Name	gUNI	gNNI
AuthTokenService	Yes	No
CapabilityService	Yes	Yes
MessagingService	Yes	Yes
GroupService	Yes	Yes

Table 1: RCS gRPC Services

RCS clients connect directly to the SPN to execute methods defined in the gUNI gRPC services. On the other hand, SPNs can bi-directionally over gRPC connect to each other to execute methods defined in the gNNI gRPC services.

SPNs which define an RCS gRPC interface will be responsible for the interworking with SIP/MSRP RCS clients and other SPNs as defined in section 6.

Push notification mechanism, as defined in [[GSMA PRD-RCC.07]] section 2.14, shall be supported for gUNI.

The architecture of RCS gRPC is illustrated in Figure 1.

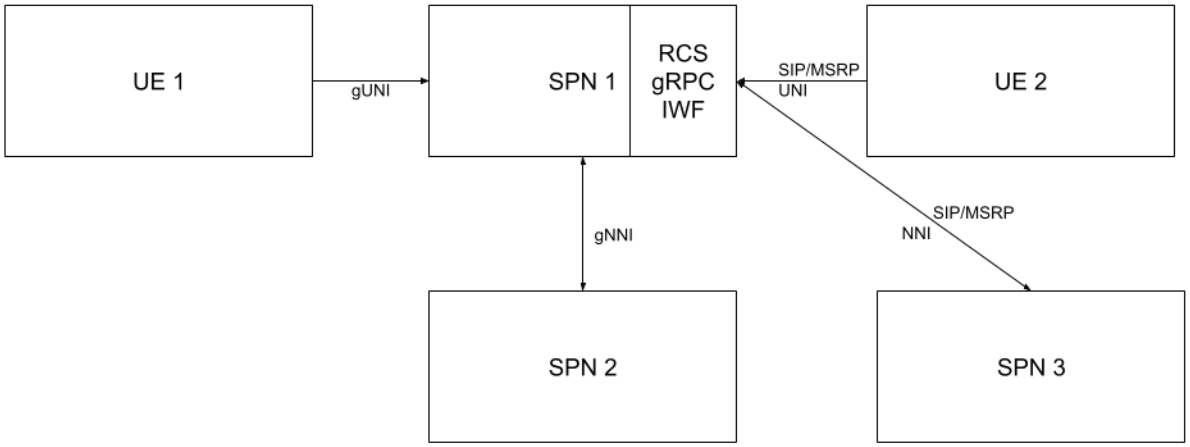

Figure 1: Architecture of RCS gRPC

2.1 Addressing and identities

RCS clients and SPNs that wish to implement RCS gRPC shall support connectivity to gRPC services via TLS. RCS clients and SPNs may connect over QUIC transport (as defined in [RFC9000]) if both endpoints support it.

2.1.1 Phone Numbers

Given the MSISDN, the RCS client or SPN shall construct the RcsId of the phone number by:

1. Setting the type of the RcsId for phone numbers to be `ID_TYPE_PHONE_NUMBER`.
2. Setting the rcs_id field as the E.164 international format of the MSISDN. If the MSISDN is not in E.164 international format, it shall follow procedures in 2.5.2 of [GSMA PRD-RCC.07] to convert it to an E.164 international format.

2.1.2 Group Chat

Given the Conversation-ID of the Group Chat, the RCS client or SPN shall construct the RcsId of the Group Chat by:

1. Setting the type of the RcsId for Group Chat to be `ID_TYPE_GROUP`.
2. Setting the rcs_id field as the Conversation-ID.

2.1.3 Chatbots

Chatbots shall use the SIP URI representation as defined in section 2.5.4.1.2 of [GSMA PRD-RCC.07]. Given the SIP URI of the Chatbot, the RCS client or SPN shall construct the RcsId of the Chatbot by:

1. Setting the type of the RcsId for Chatbot to be `ID_TYPE_RCS_BOT`.
2. Setting the rcs_id field as the SIP URI of the Chatbot.

2.2 RCS gRPC Access Technology

The IMS APN is not available to gRPC access technology, therefore the exact mechanism for enablement is deployment specific.

3 Wireformat

3.1 Protocol buffer definitions

3.1.1 Common

```
// Uniquely identifies an endpoint (user, group, or bot).
message RcsId {
  // The RCS id of the endpoint. This field is required and cannot be
  //empty.
  string rcs_id = 1 [(google.api.field_behavior) = REQUIRED];
  // The type of the endpoint.
  IdType type = 2 [(google.api.field_behavior) = REQUIRED];
}

// The header sent along with all requests
message RequestHeader {
  // REQUIRED:
  // UUID that can be used to trace this request for debugging purposes.
  // Generated by request creator per request.
  string request_id = 1 [
    (google.api.field_info).format = UUID,
    (google.api.field_behavior) = REQUIRED
  ];

  // OPTIONAL :
  // gNNI only: User who's generating the request.
  RcsId requester_id = 2 [(google.api.field_behavior) = OPTIONAL];

  // OPTIONAL:
  // gUNI only: Auth token included for authenticated client to server gUNI
  // requests. auth_token_payload identifies the client and is returned
  // from the ACS.
  AuthToken auth_token = 3 [(google.api.field_behavior) = OPTIONAL];

  // OPTIONAL:
  // SIP headers to be included in the request, if any.
  repeated SipHeader sip_headers = 4 [(google.api.field_behavior) =
  OPTIONAL];

  // OPTIONAL:
  // Device info of the client making the request.
  DeviceInfo device_info = 5 [(google.api.field_behavior) = OPTIONAL];
}

// Header returned along with all responses.
message ResponseHeader {
  // The server timestamp of the response.
  google.protobuf.Timestamp response_time = 1 [(google.api.field_behavior)
  = REQUIRED];

  // OPTIONAL:
  // SIP headers to be included in the response, used for inter-working
```

```
  // where needed.
  repeated SipHeader sip_headers = 2 [(google.api.field_behavior) =
  OPTIONAL];
}

// SIP header to be included in the request or response messages.
// The name and value of the header are set in this message.
message SipHeader {
  string name = 1 [(google.api.field_behavior) = REQUIRED];
  string value = 2 [(google.api.field_behavior) = REQUIRED];
}

// Metadata that can accompany messages (equivalent to CPIM headers).
message Metadata {
  // Custom headers in the metadata.
  // As Defined in
  /// https://datatracker.ietf.org/doc/html/rfc3862#section-3.4
  message Header {
    // Namespace URN for the header.
    // Value for default namespace should be
    // "urn:ietf:params:cpim-headers:"
    // See https://datatracker.ietf.org/doc/html/rfc3862#section-4
    //
    // Combining namespace + name is not allowed because they are
    // presented uniquely in the protocol buffer:
    //
    // Valid Examples:
    // {
    //   namespace: "urn:ietf:params:cpim-headers:"
    //   name: "Subject"
    //   value: "Some test subject
    // }
    // {
    //   namespace: "urn:ietf:params:custom-headers"
    //   name: "X-Custom-Header"
    //   value: "some-value"
    // }
    //
    // Namespace URN
    string namespace = 1 [(google.api.field_behavior) = REQUIRED];
    // Header name.
    string name = 2 [(google.api.field_behavior) = REQUIRED];
    // Header value.
    string value = 3 [(google.api.field_behavior) = OPTIONAL];
  }

  // Custom headers in the metadata equivalent to CPIM headers.
  repeated Header headers = 2 [(google.api.field_behavior) = REQUIRED];
}

message DeviceInfo {
  // The full OS of the device, e.g., 9.3.1 for iOS and
```

```
  // google/angler/angler:6.0.1/MHC19I/2590160:user/release-keys for
Android.
  string os = 1 [(google.api.field_behavior) = REQUIRED];

  // The user agent of the client.
  string user_agent = 2 [(google.api.field_behavior) = REQUIRED];
}

// Different types of identities supported by the backend.
enum IdType {
  // Unknown id type.
  ID_TYPE_UNKNOWN = 0;
  // Phone number id type.
  ID_TYPE_PHONE_NUMBER = 1;
  // Group id type.
  ID_TYPE_GROUP = 2;
  // RBM bot id type.
  ID_TYPE_RCS_BOT = 3;
}

// AuthTokens are returned to clients as part of the
// RefreshAuthTokenResponse
// They must be included in the header of all authenticated requests.
message AuthToken {
  // payload is the encrypted payload of the token.
  bytes payload = 1 [(google.api.field_behavior) = REQUIRED];
}

// Server-generated Token that can be used for routing on RCS backends.
// Its value should be opaque to clients, but clients are expected to
// persist the values they receive from the server and pass it to back to
// the server in their interactions.
message MlsOpaqueToken {
  // The value of the token.
  bytes payload = 1;
}

message ClientMlsControlMessage {
  // Client-generated id that identifies this message.
  string message_id = 1 [(google.api.field_behavior) = REQUIRED];

  // Epoch authenticator for the current epoch.
  //
  // This field is required to be set for all ClientMlsControlMessages
  // other than those that contain a WelcomeCommitBundle in the first epoch
  // for a new era.
  //
  // The EpochAuthenticator (described in
  // https://www.rfc-editor.org/rfc/rfc9420.html#name-epoch-
  // authenticators).
  bytes epoch_authenticator = 2 [(google.api.field_behavior) = REQUIRED];

  // The ClientMlsRcsMessage as per [GSMA PRD-RCC.16] being wrapped.
  ClientMlsRcsMessage mls_rcs_message = 3 [(google.api.field_behavior) =
```

```
  REQUIRED];

  // Latest MlsOpaqueToken that the client has received from the server.
  // Optional in case of requests to create a new MLS group, or in
  // scenarios
  // where the client has lost its MLS state.
  MlsOpaqueToken mls_opaque_token = 4 [(google.api.field_behavior) =
  OPTIONAL];
}

// Wrapper around ServerMlsRcsMessage that may include additional metadata.
message ServerMlsControlMessage {
  // The ServerMlsRcsMessage [GSMA PRD-RCC.16] being wrapped.
  ServerMlsRcsMessage mls_rcs_message = 1 [(google.api.field_behavior) =
  REQUIRED];

  // The EpochId for the epoch associated with the contained
  // ServerMlsRcsMessage.
  EpochId epoch_id = 2 [(google.api.field_behavior) = REQUIRED];

  // Server-generated MlsOpaqueToken that clients should persist and
  //include in
  // future ClientMlsControlMessages.
  // Required to be set in all ServerMlsControlMessages that contain
  // welcomes.
  MlsOpaqueToken mls_opaque_token = 3 [(google.api.field_behavior) =
  OPTIONAL];

}


// A request to apply a ClientMlsControlMessage to an existing MLS group.
message ApplyMlsControlMessageRequest {
  // Request header.
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // The message that should be applied to the MLS group.
  ClientMlsControlMessage client_mls_control_message = 2 [
    (google.api.field_behavior) = REQUIRED];

  // The destination ID that should be used to derive the intended MLS
  // group.
  // If the destination is a phone number, then the MLS group ID should be
  // derived from the normalized pair of phone numbers. If the destination
  // is a group ID, then the MLS group ID will be equal to the group ID.
  RcsId destination_id = 3 [(google.api.field_behavior) = REQUIRED];
}

// A response to a request to apply a ClientMlsControlMessage to an
// existing MLS group.
message ApplyMlsControlMessageResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];
}

// A request to create a new MLS group or advance the era of an existing
// MLS group.
message CreateMlsGroupRequest {
  // Request header.
```

```
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // The message that should be used to create the MLS group. This must be
  // a ClientMlsControlMessage with a ClientMlsRcsMessage containing a
  // WelcomeCommitBundle.
  ClientMlsControlMessage client_mls_control_message = 2 [
  (google.api.field_behavior) = REQUIRED];

  // The destination ID that should be used to derive the intended MLS
  // group.
  // If the destination is a phone number, then the MLS group ID should be
  // derived from the normalized pair of phone numbers. If the destination
  // is a group ID, then the MLS group ID will be equal to the group ID.
  RcsId destination_id = 3 [(google.api.field_behavior) = REQUIRED];
}

// A response to a request to create a new MLS group.
message CreateMlsGroupResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];
}

enum GetMlsGroupInfoRequestMode {
  GET_MLS_GROUP_INFO_REQUEST_MODE_UNKNOWN = 0;
  GET_MLS_GROUP_INFO_REQUEST_MODE_NORMAL = 1;
  GET_MLS_GROUP_INFO_REQUEST_MODE_ENHANCED = 2;
}

// A request to get information about an MLS group.
message GetMlsGroupInfoRequest {
  // Request header.
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // The destination ID that should be used to derive the intended MLS
  // group.
  // If the destination is a phone number, then the MLS group ID should be
  // derived from the normalized pair of phone numbers. If the destination
  // is a group ID, then the MLS group ID will be equal to the group ID.
  RcsId destination_id = 2 [(google.api.field_behavior) = REQUIRED];

  // Latest MlsOpaqueToken that the client has received from the server.
  MlsOpaqueToken mls_opaque_token = 3 [(google.api.field_behavior) =
  OPTIONAL];

  // The type of GetMlsGroupInfoRequest to perform.
  GetMlsGroupInfoRequestMode mode = 4 [(google.api.field_behavior) =
  REQUIRED];

  // Identifying information for the Epoch that the client is currently in.
  // If set, then the server will return any committed control messages
  // that were accepted since this Epoch.
  EpochId current_client_epoch_id = 5 [(google.api.field_behavior) =
  OPTIONAL];
}

// A response to a request to get information about an MLS group.
message GetMlsGroupInfoResponse {
  // Generic response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // The latest MlsGroupInfo for the MLS Conversation, the contents of
```

```
  // which can be used to initiate self-healing. Only set if the mode was
  // GET_MLS_GROUP_INFO_REQUEST_MODE_NORMAL.
  MlsGroupInfo mls_group_info = 2 [(google.api.field_behavior) = OPTIONAL];

  // The MlsEnhancedGroupInfo for the MLS Conversation. Only set if the
  // mode was GET_MLS_GROUP_INFO_REQUEST_MODE_ENHANCED.
  MlsEnhancedGroupInfo mls_enhanced_group_info = 3
      [(google.api.field_behavior) = OPTIONAL];
}


// gRPC error response sent as part of a status.proto as per [gRPC].
//
// This is sent as part of the details field of a google.rpc.Status proto
when
// an error occurs in the processing of a gRPC request.

message ErrorDetails {
  string error_code = 1 [(google.api.field_behavior) = REQUIRED];
  string error_reason = 2 [(google.api.field_behavior) = OPTIONAL];
}
```

3.1.2 Auth Base

```
// Request to refresh an existing authentication token for a user.
message RefreshAuthTokenRequest {
  // Request header.
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // Device hardware and OS information.
  DeviceInfo device_info = 2 [(google.api.field_behavior) = REQUIRED];

  // Auth token to be signed
  AuthTokenTBS auth_token_tbs = 3 [(google.api.field_behavior) = REQUIRED];

  // Signature of the auth token TBS
  bytes signature = 4 [(google.api.field_behavior) = REQUIRED];
}

message RefreshAuthTokenResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // Refreshed Auth token.
  AuthToken auth_token = 2 [(google.api.field_behavior) = REQUIRED];

  // Timestamp indicating the time when this auth token expires
  google.protobuf.Timestamp expiry_time = 3 [(google.api.field_behavior) =
  REQUIRED];
}

message AuthTokenTBS {
  // Current token to be signed by the RCS client.
  AuthToken auth_token = 1 [(google.api.field_behavior) = REQUIRED];
```

```
  // Timestamp of the request. Must be network time.
  google.protobuf.Timestamp timestamp = 2[(google.api.field_behavior) =
  REQUIRED];
}
```

3.1.3 Capabilities Base

```
// Capability message containing key-value pair where the key is the
// feature tag and the value is the optional IARI defined as per section
// 2.6.1 of [GSMA PRD-RCC.07].
// Valid examples:
// {
//   key: "+g.3gpp.iari-ref"
//   value: "urn%3Aurn-7%3A3gpp-application.ims.iari.rcse.im"
// }
// {
//   key: "+g.3gpp.iari-ref"
//   value: "urn%3Aurn-7%3A3gpp-application.ims.iari.rcs.fthttp"
// }
// {
//   key: "+g.gsma.rcs.msgrevoke"
// }
message Capability {
  // The capability name
  string key = 1 [(google.api.field_behavior) = REQUIRED];
  // The capability value
  string value = 2 [(google.api.field_behavior) = OPTIONAL];
}

// A list of capabilities.
message Capabilities {
  // List of capabilities.
  repeated Capability capabilities = 1 [(google.api.field_behavior) =
REQUIRED];
}

// Result of a capabilities lookup.
enum LookupResult {
  // Unknown result.
  CAPABILITIES_RESULT_UNKNOWN = 0;
  // Maps to 200
  CAPABILITIES_RESULT_OK = 1;
  // Maps to 404
  CAPABILITIES_RESULT_NOT_FOUND = 2;
  // Maps to 480
  CAPABILITIES_RESULT_TEMPORARILY_UNAVAILABLE = 3;
}

// The capabilities of a single user.
message UserCapabilities {
  // The RCS id of the user.
  RcsId rcs_id = 1 [(google.api.field_behavior) = REQUIRED];
```

```
  // The capabilities of the user.
  Capabilities capabilities = 2 [(google.api.field_behavior) = REQUIRED];
}

// The wrapper for capabilities lookup result of a single user.
message UserCapabilitiesResult {
  // The UseCapabilities for the user.
  UserCapabilities user_capabilities = 1 [(google.api.field_behavior) =
  REQUIRED];

  // The result of the capabilities lookup.
  LookupResult lookup_result = 2 [(google.api.field_behavior) = REQUIRED];
}

// A request to fetch the capabilities of one user.
message GetCapabilitiesRequest {
  // Request header
  RequestHeader header = [(google.api.field_behavior) = REQUIRED]1;

  // The users to fetch capabilities for.
  repeated RcsId user_ids = 2 [(google.api.field_behavior) = REQUIRED];
}

// The response to a request to fetch the capabilities of a list of users.
message GetCapabilitiesResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // Contains details about the requested user's capabilities.
  repeated UserCapabilitiesResult user_capabilities_results = 2
  [(google.api.field_behavior) = REQUIRED];
}

// A request to set the capabilities of a user.
message SetCapabilitiesRequest {
  // Request header.
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // The capabilities of the user.
  UserCapabilities user_capabilities = 2 [(google.api.field_behavior) =
  REQUIRED];
}

// The response to a request to set the capabilities of a user.
message SetCapabilitiesResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];
}
```

3.1.4 Messaging Base

```
// BindEvent is the event container that is sent from the SPN to the RCS
// client when a Bind stream is established.
```

```
message BindEvent {
  oneof event {
    MessagingEvent messaging_event = 1;
    KeepAliveEvent keep_alive_event = 2;
  }
}

// KeepAliveEvent, wrapped by a BindEvent, is an empty event that is sent
// from the SPN to the RCS client during a Bind stream to keep the
// connection alive.
message KeepAliveEvent {}

// PullMessagesRequest is the request to get un-acked messages from
// the local SPN.
// The response size is limited. The RCS client should send another
// PullMessagesRequest again, after acking the messages, if the pulled_all
// bit is false in the response.
message PullMessagesRequest {
  // Request Header
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // start_timestamp_usec is the time in microseconds to start the pull
  google.protobuf.Timestamp start_timestamp_usec = 2
  [(google.api.field_behavior) = REQUIRED];
}

// PullMessagesResponse is the response to a request to get the un-acked
// messages from the local SPN.
message PullMessagesResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // The messages.
  repeated MessagingEvent messages = 2 [(google.api.field_behavior) =
  OPTIONAL];

  enum PullStatus {
    // Unknown pull status.
    PULL_STATUS_UNKNOWN = 0;
    // Not all messages have been pulled.
    PULL_STATUS_INCOMPLETE = 1;
    // All messages have been pulled.
    PULL_STATUS_COMPLETE = 2;
  }

  // Indicates whether all messages have been pulled.
  PullStatus pull_status = 3 [(google.api.field_behavior) = REQUIRED];

}

// A request to prewarm the connection for a given set of contacts.
message PrewarmRequest {
```

```
  // Request header.
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // The rcs id to prewarm the connection for, must be a user or a Group
  // Chat id
  RcsId rcs_id = 2 [(google.api.field_behavior) = REQUIRED];
}

// The response to a request to prewarm the connection for a given set of
// contacts.
message PrewarmResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];
}

// MessagingEvent is the event container that is sent from the local and
// remote SPN to the RCS client.
message MessagingEvent {
  // Client or server-generated ID. Must be globally unique.
  // Must be unique between sender and receiver.
  string message_id = 1 [(google.api.field_behavior) = REQUIRED];
  // Message payload.
  // One of the following must be set.
  oneof payload {
    // A chat message.
    ChatMessage chat_message = 101;
    // Read or Deliver Receipt for a message.
    MessageReceipt message_receipt = 102;
    // A file transfer message.
    FileTransfer file_transfer = 103;
    // A typing notification message.
    TypingNotification typing_notification = 104;
    // A group change event.
    GroupChangeEvent group_change_event = 105;
    // A ServerMlsControlMessage.
    ServerMlsControlMessage server_mls_rcs_message = 106
  }

  // Server Timestamp of when the message was received by the originating
  // SPN.
  google.protobuf.Timestamp event_time = 2 [(google.api.field_behavior) =
  REQUIRED];


  // The rcs id of the sender, must be ID_TYPE_PHONE_NUMBER
  // or ID_TYPE_RCS_BOT  RcsId sender_id = 3 [(google.api.field_behavior) =
REQUIRED];

  // The rcs id where the message should be routed to, user or group id.
  RcsId receiver_id = 4 [(google.api.field_behavior) = REQUIRED];

  // The rcs id of the Group Chat, if this message belongs to a Group Chat.
  // In MO when sending Messages to a group, the group id is not set here,
```

```
  // instead, it is set in the receiver field. For IMDNs sent in a group
  // chat, the group ID is set here, and the sender of the original message
  // is set in receiver id.
  RcsId group_id = 5 [(google.api.field_behavior) = OPTIONAL];

  // Client defined metadata passed alongside the message content.
  Metadata metadata = 6 [(google.api.field_behavior) = OPTIONAL];

  // Requested delivery priority.
  MessagePriority priority = 7 [(google.api.field_behavior) = REQUIRED];

  // Indicates requested delivery guarantees.
  DeliveryType delivery_type = 8 [(google.api.field_behavior) = REQUIRED];

  // Content disposition(s) of the message.
  // https://datatracker.ietf.org/doc/html/rfc5438#section-7.1.1.3
  repeated ContentDisposition content_dispositions = 9
  [(google.api.field_behavior) = REQUIRED];

  // Optional field used to indicate the original message being referred to
  // in case of message-resends or IMDNS sent in response to re-send
  // messages.
  //
  // In the case of a re-sent message, this field will contain the message
  // id of the original message that was re-sent.
  //
  // In the case of an IMDN sent in response to a re-sent message, this
  // field will contain the message id of the original message that was re-
  // sent, and the `message_id` field will contain the id of the IMDN.
  string original_message_id = 10 [(google.api.field_behavior) = OPTIONAL];
}

// A part of a multipart message exchanged between users. Can be sent in a
// 1:1 or group chat.
message MessagePart {
  // Content-type of the message part.
  string content_type = 1 [(google.api.field_behavior) = REQUIRED];

  // Message content.
  bytes content = 2 [(google.api.field_behavior) = OPTIONAL];
}

// A multipart chat message
message ChatMessage {
  // The parts of the multipart message.
  repeated MessagePart parts = 1 [(google.api.field_behavior) = REQUIRED];
}

// A message receipt indicating message delivery and read statuses
message MessageReceipt {
  // Describes the type of receipt.
  enum ReceiptType {
    // Unknown receipt type.
```

```
    RECEIPT_TYPE_UNKNOWN = 0;
    // Indicates the message was delivered to MT User.
    RECEIPT_TYPE_POSITIVE_DELIVERY = 1;
    // Indicates the message was read by MT User.
    RECEIPT_TYPE_READ = 2;
    // Indicates the message was undeliverable to MT User.
    RECEIPT_TYPE_NEGATIVE_DELIVERY = 3;
  }
  // The type of the receipt.
  ReceiptType receipt_type = 1 [(google.api.field_behavior) = REQUIRED];

  // Information about the original message.
  ReceiptInfo receipt_info = 2 [(google.api.field_behavior) = REQUIRED];
}

// Information about the original message that was read or delivered.
message ReceiptInfo {
  // The message id of the original message.
  string message_id = 1 [(google.api.field_behavior) = REQUIRED];

  // The timestamp when the original message (not IMDN) was sent. See
  // <datetime> element in RFC5438.
  google.protobuf.Timestamp event_time = 2 [(google.api.field_behavior) =
  REQUIRED];

  // The optional reason provided if the IMDN failed delivery due to an MLS
  // server or client error (E2EE only)
  oneof mls_delivery_failed_reason {
    DeliveryFailedMlsServerReason delivery_failed_mls_server_reason = 3;
    DeliveryFailedMlsClientReason delivery_failed_mls_client_reason = 4;
  }
}

// Details about the file of a file transfer.
message FileTransferInfo {
  // The name of the file.
  string file_name = 1 [(google.api.field_behavior) = REQUIRED];
  // The size of the file in bytes.
  int64 size_bytes = 2 [(google.api.field_behavior) = REQUIRED];
  // The content-type of the file.
  string content_type = 3 [(google.api.field_behavior) = REQUIRED];

  // The file URL.
  string url = 4 [(google.api.field_behavior) = REQUIRED];

  // Timestamp until the content is valid on the content server.
  google.protobuf.Timestamp valid_until_time = 5
  [(google.api.field_behavior) = REQUIRED];

  // The disposition of the file.
  FileDisposition disposition = 6 [(google.api.field_behavior) = OPTIONAL];

  // The playing length of the file in seconds.
```

```
  // This is only used for Audio Messages.
  google.protobuf.Duration playing_length = 7 [
    (google.api.field_behavior) = OPTIONAL
  ];

}

// A file transfer message.
message FileTransfer {
  // Information about the file being transferred.
  FileTransferInfo file_info = 1 [(google.api.field_behavior) = REQUIRED];

  // Information about the thumbnail file.
  FileTransferInfo thumbnail = 2 [(google.api.field_behavior) = OPTIONAL];


  // Field containing the encrypted MlsMessage used to transmit the
  // encryption keys used for encrypting the file and thumbnail content.
  // Optional.
  bytes mls_encrypted_payload = 3 [(google.api.field_behavior) = REQUIRED];
}

// A typing notification message.
message TypingNotification {
  // The different states of the typing notification.
  enum State {
    // Unknown state.
    STATE_UNKNOWN = 0;
    // User is typing.
    STATE_ACTIVE = 1;
    // User stopped typing.
    STATE_IDLE = 2;
  }

  // The state of the typing notification. Valid values are active (user is
  // typing/composing) or idle (user stopped typing).
  State state = 1 [(google.api.field_behavior) = REQUIRED];

  // Defines the active-state refresh interval.
  google.protobuf.Duration refresh_interval = 2
  [(google.api.field_behavior) = REQUIRED];
}

// The result of an attempt to revoke a single message.
message RevokeResult {
  // The message id that was attempted to be revoked.
  string message_id = 1 [(google.api.field_behavior) = REQUIRED];

  // The status of the attempt to revoke the message.
  RevokeStatus status = 2 [(google.api.field_behavior) = REQUIRED];
}

// A request wrapper to send a message to a user.
```

```
message SendMessageRequest {
  // Request header.
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];


  // The message to send.
  MessagingEvent message = 2 [(google.api.field_behavior) = REQUIRED];
}

// The response to a request to send a message to a user on the RCS
backend.
message SendMessageResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];
}

// A request to revoke an undelivered message.
message RevokeMessageRequest {
  // Request header.
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // Message to revoke (best effort).
  string message_id = 2 [(google.api.field_behavior) = REQUIRED];

  // The id that the message was originally sent to.
  RcsId destination_id = 3 [(google.api.field_behavior) = REQUIRED];
}

// The response to a request to revoke an undelivered message.
message RevokeMessageResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // Contains the status for every message in the request.
  RevokeResult result = 2 [(google.api.field_behavior) = REQUIRED];
}

// Describes the disposition of the file.
// https://datatracker.ietf.org/doc/html/rfc5547#section-7
enum FileDisposition {
  // Unknown file disposition.
  FILE_DISPOSITION_UNKNOWN = 0;
  // The file should be rendered in the UI, suitable for thumbnails.
  FILE_DISPOSITION_RENDER = 1;
  // The file should be downloaded.
  FILE_DISPOSITION_ATTACHMENT = 2;
}

// Describes the priority for delivering messages.
enum MessagePriority {
  // Unknown message priority.
  MESSAGE_PRIORITY_UNKNOWN = 0;
  // Message should be delivered immediately.
```

```
  MESSAGE_PRIORITY_HIGH = 1;
  // Message can be delivered at recipient's leisure.
  MESSAGE_PRIORITY_NORMAL = 2;
  // Message delivery is only be attempted if recipient is currently
connected.
  MESSAGE_PRIORITY_BEST_EFFORT = 3;
}

// Describes the requested delivery options from the server.
enum DeliveryType {
  // Unknown delivery type.
  DELIVERY_TYPE_UNKNOWN = 0;
  // Delivery to the recipient should be guaranteed.  Where the message
will
  // stored in the store and forward if undeliverable at transmission.
  DELIVERY_TYPE_GUARANTEED = 1;
  // Messages that should be delivered as best-effort.
  DELIVERY_TYPE_BEST_EFFORT = 3;
}

// Describes the content disposition of a message.
enum ContentDisposition {
  // Unknown content disposition.
  CONTENT_DISPOSITION_UNKNOWN = 0;
  // Positive content disposition.
  CONTENT_DISPOSITION_POSITIVE = 1;
  // Negative content disposition.
  CONTENT_DISPOSITION_NEGATIVE = 2;
  // Display content disposition.
  CONTENT_DISPOSITION_DISPLAY = 3;
  // Indicates that no content disposition is expected.
  CONTENT_DISPOSITION_NONE = 4;
}

// Describes the outcome of an attempt to revoke a single message.
enum RevokeStatus {
  // Unknown revoke status.
  REVOKE_STATUS_UNKNOWN = 0;
  // Message was successfully revoked.
  REVOKE_STATUS_SUCCEEDED = 1;
  // Message has already been delivered and cannot be revoked.
  REVOKE_STATUS_DELIVERED = 2;
  // There is no record of this message sent to the destination.
  REVOKE_STATUS_MESSAGE_NOT_FOUND = 3;
  // The destination is not valid.
  REVOKE_STATUS_DESTINATION_NOT_FOUND = 4;
  // The user is not authorized to revoke the message.
  REVOKE_STATUS_PERMISSION_DENIED = 5;
  // An internal error occurred while revoking the message.
  REVOKE_STATUS_INTERNAL_ERROR = 6;
}

// A request to ack messages that were received from the server in a
```

```
// BindEvent.
message AckMessagesRequest {
  // Request header.
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];
  // A list of message ids to ack.
  repeated string message_ids = 2 [(google.api.field_behavior) = REQUIRED];
}

// The response to a request to ack messages that were received from the
// server in a BindEvent.
message AckMessagesResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];
}


// BindRequest is the request to establish a server to client
// unidirectional stream for receiving messages and events from the
// server.
message BindRequest {
  // Request header.
  RequestHeader header = 1 [
    (google.api.field_behavior) = REQUIRED
  ];
}

// The reason provided if the IMDN failed delivery due to an MLS server
// error. This corresponds to the XML field <mls-server-failure-reason>,
// which is an extension within the <delivery-notification>. This is only
// expected to be present if the <status> is "failed".
enum DeliveryFailedMlsServerReason {
  SERVER_REASON_UNKNOWN = 0;
  SERVER_REASON_INCORRECT_ERA = 1;
  SERVER_REASON_INCORRECT_EPOCH = 2;
  SERVER_REASON_INCORRECT_EPOCH_AUTHENTICATOR = 3;
  SERVER_REASON_EXPIRED_CREDENTIAL = 4;
  SERVER_REASON_MISMATCHED_RCS_GROUP_STATE = 5;
  SERVER_REASON_UNPARSABLE_COMMIT = 6;
  SERVER_REASON_MISMATCHED_CONFIRMATION_TAG = 7;
  SERVER_REASON_PENDING_PROPOSAL = 8;
  SERVER_REASON_TRANSIENT_ERROR = 9;
  SERVER_REASON_ENCRYPTION_NOT_AVAILABLE = 10;
  SERVER_REASON_INVALID_COMMIT_SERVER = 11;
  SERVER_REASON_MLS_GROUP_NOT_FOUND = 12;
  SERVER_REASON_INVALID_INPUT = 13;
  SERVER_REASON_ERA_ADVANCEMENT_QUOTA_REACHED = 14;
  SERVER_REASON_MLS_GROUP_HAS_END_MLS = 15;
  SERVER_REASON_EPOCH_ADVANCEMENT_QUOTA_REACHED = 16;
}

// The reason provided if the IMDN failed delivery due to an MLS client
// error. This corresponds to the XML field <mls-client-failure-reason>,
```

```
// which is an extension within the <delivery-notification>. This is only
// expected to be present if the <status> is "failed".
enum DeliveryFailedMlsClientReason {
  CLIENT_REASON_UNKNOWN = 0;
  CLIENT_REASON_MESSAGE_FROM_NON_MEMBER = 1;
  CLIENT_REASON_INVALID_CREDENTIAL = 2;
  CLIENT_REASON_INVALID_COMMIT = 3;
  CLIENT_REASON_FAILED_TO_DECRYPT = 4;
  CLIENT_REASON_IGNORED_FTD = 5;
  CLIENT_REASON_COMMIT_IN_PRIVATEMESSAGE = 6;
  CLIENT_REASON_RESENT_MESSAGE_FOR_ME_FAILED_TO_DECRYPT = 7;
  CLIENT_REASON_COMMIT_PROCESSED_IN_ENHANCED_SELF_HEAL = 8;
  CLIENT_REASON_COMMIT_PROCESSED_IN_SELF_HEAL = 9;
  CLIENT_REASON_COMMIT_FAILED_THEN_ERA_ADVANCEMENT = 10;
  CLIENT_REASON_CONTROL_MESSAGE_FAILED_DUE_TO_DOWNGRADE = 11;
  CLIENT_REASON_PROPOSAL_PROCESSED_IN_ENHANCED_SELF_HEAL = 12;
  CLIENT_REASON_PROPOSAL_PROCESSED_IN_SELF_HEAL = 13;
  CLIENT_REASON_CONTROL_MESSAGE_FAILED = 14;
}
```

3.1.5 Group Base

```
// A subject for a Group Chat.
message GroupSubject {
  // The content type of the icon content.
  // For unencrypted icons, the content type must be empty.
  // MLS-RCS encryption uses "message/mls-ft" according to
  // https://www.gsma.com/solutions-and-impact/technologies/networks/wp-
  // content/uploads/2025/03/RCC.16-v1.0.pdf

  string content_type = 1 [(google.api.field_behavior) = REQUIRED];

  // The subject of the Group Chat.
  // Empty payload means no Group Chat subject.

  bytes payload = 2 [(google.api.field_behavior) = OPTIONAL];}

// An image used as the icon for a Group Chat.
message GroupIcon {
  // The content type of the icon content.
  // For unencrypted icons, the content type must be empty.
  // MLS-RCS encryption uses "message/mls-ft" according to
  // https://www.gsma.com/solutions-and-impact/technologies/networks/wp-
  // content/uploads/2025/03/RCC.16-v1.0.pdf

  string content_type = 1 [(google.api.field_behavior) = REQUIRED];

  // The URI of the icon.
  // Empty string means no icon.
  //
  // The URI may be localized and hosted by the SPN as appropriate if not
  // publicly available.
  string icon_uri = 2 [(google.api.field_behavior) = OPTIONAL];}
```

```
// Contains user-defined custom information about a Group Chat.
message GroupMetadata {
  // The subject of the Group Chat. Optional.
  GroupSubject subject = 1 [(google.api.field_behavior) =
  OPTIONAL];

  // An image that represents the Group Chat. Optional.
  GroupIcon icon = 2 [(google.api.field_behavior) = OPTIONAL];
}

// Describes a Group Chat member.
message GroupMember {
  // The RCS id of the group member.
  RcsId rcs_id = 1 [(google.api.field_behavior) = REQUIRED];

  // The timestamp of when the user joined the group.
  google.protobuf.Timestamp joined_time = 2 [(google.api.field_behavior) =
  REQUIRED];
}

// Describes a Group Chat (members and properties).
message GroupInfo {
  // The RCS id of the Group Chat.
  RcsId group_id = 1 [(google.api.field_behavior) = REQUIRED];

  // The URI of the Group Chat.
  string conference_uri = 2 [(google.api.field_behavior) = REQUIRED];

  // The metadata of the Group Chat.
  GroupMetadata metadata = 3 [(google.api.field_behavior) = REQUIRED];

  // The members of the Group Chat.
  repeated GroupMember members = 4 [(google.api.field_behavior) =
  REQUIRED];

  // The MLS group info of the Group Chat.
  MlsGroupInfo mls_group_info = 5 [(google.api.field_behavior) = OPTIONAL];
}

// Sent to notify all Group Chat members when a Group Chat has been
// created.
//
// Note that newly added members for existing groups will also receive this
// notification instead of AddUsersToGroupNotification.
message CreateGroupNotification {
  // Timestamp of when the Group Chat was created.
  google.protobuf.Timestamp create_time = 1 [(google.api.field_behavior) =
OPTIONAL];

  // Contains all the Group Chat information.
  GroupInfo group_info = 2 [(google.api.field_behavior) = REQUIRED];
}
```

```
// Sent to existing Group Chat members when a user has been added to a
// Group Chat.
//
// Note that newly added members will receive CreateGroupNotification
// instead.
message AddUsersToGroupNotification {
  // Timestamp of when the users were added.
  google.protobuf.Timestamp addition_time = 1 [(google.api.field_behavior)
  = REQUIRED];

  // The users that have been added to the Group Chat.
  repeated RcsId added_users = 2 [(google.api.field_behavior) = REQUIRED];
}

// Sent to all Group Chat members (including the ones that were removed)
// when a user has been removed from the Group Chat.
message RemoveUsersFromGroupNotification {
  // Timestamp of when the users were removed.
  google.protobuf.Timestamp removal_time = 1 [(google.api.field_behavior) =
  REQUIRED];

  // The users that have been removed from the Group Chat and the reason
  // for removal.
  repeated RemoveGroupUserMetadata removed_users = 2
  [(google.api.field_behavior) = REQUIRED];
}

// Sent to all Group Chat members when the Group Chat metadata changes.
message ChangeGroupMetadataNotification {
  // Timestamp of when the metadata was changed.
  google.protobuf.Timestamp notification_time = 1
  [(google.api.field_behavior) = REQUIRED];

  // The updated metadata.
  GroupMetadata metadata = 2 [(google.api.field_behavior) = REQUIRED];
}

// The envelope transferred from the local SPN to clients
message GroupChangeEvent {
  // Message payload.
  // One of the following must be set.
  oneof payload {
    // Sent to notify all Group Chat members when a Group Chat has been
    // created.
    CreateGroupNotification create_group = 101;
    // Sent to existing group members when a user has been added to a
    // Group Chat.
    AddUsersToGroupNotification add_users = 102;
    // Sent to all group members (including the ones that were removed)
    // when a user has been removed from the Group Chat.
    RemoveUsersFromGroupNotification remove_users = 103;
    // Sent to all group members when the group Metadata changes.
    ChangeGroupMetadataNotification change_group_metadata = 104;
```

```
  }
  // ServerMlsControlMessage associated with the GroupChangeEvent.
  //Optional.
  ServerMlsControlRcsMessage server_mls_control_rcs_message = 1
  [(google.api.field_behavior) = OPTIONAL];}

// A request to add users to a Group Chat.
message AddGroupUsersRequest {
  // Request header.
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // The id of the Group Chat to add users to.
  RcsId group_id = 2 [(google.api.field_behavior) = REQUIRED];

  // The users to add to the Group Chat.
  repeated RcsId user_ids = 3 [(google.api.field_behavior) = REQUIRED];

  // A ClientMlsControlMessage containing a WelcomeCommitBundle.
  // Optional.  ClientMlsControlMessage client_mls_control_rcs_message = 4
  [(google.api.field_behavior) = OPTIONAL];
}

// The response to a request to add users to a Group Chat.
message AddGroupUsersResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // The users that were added to the Group Chat.
  repeated RcsId added_user_ids = 2 [(google.api.field_behavior) =
  REQUIRED];
}

// Reason for removing a user from a Group Chat.
enum RemoveGroupUserReason {
  // Unspecified reason.
  REMOVE_USER_REASON_UNSPECIFIED = 0;
  // The user was kicked from the group.
  REMOVE_USER_REASON_KICKED = 1;
  // The user left the group.
  REMOVE_USER_REASON_LEFT = 2;
  // The user was removed from the group by the server.
  REMOVE_USER_REASON_REMOVED_BY_SERVER = 3;
}

// A user and reason for removing them from a Group Chat.
message RemoveGroupUserMetadata {
  // The user to remove.
  RcsId user = 1 [(google.api.field_behavior) = REQUIRED];

  // The user and the reason for removal.
  RemoveGroupUserReason reason = 2 [(google.api.field_behavior) =
  REQUIRED];
}
```

```
// A request to remove users from a group.
message RemoveGroupUsersRequest {
  // Request header.
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // The id of the group to remove users from.
  RcsId group_id = 2 [(google.api.field_behavior) = REQUIRED];

  // The users to remove.
  repeated RemoveGroupUserMetadata user_ids = 3;

  // A ClientMlsControlMessage containing either a proposalList in the case
  // of self-leave or a CommitBundle in the case of removal.
Optional.  ClientMlsControlMessage client_mls_control_rcs_message = 4;
}

// The response to a request to remove users from a Group Chat.
message RemoveGroupUsersResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // The users that have been removed from the Group Chat.
  repeated RcsId removed_user_ids = 2 [(google.api.field_behavior) =
  REQUIRED];
}

// A request to update the properties in the Group Chat metadata.
message ChangeGroupMetadataRequest {
  // Request header.
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // The id of the Group Chat to update the metadata for.
  RcsId group_id = 2 [(google.api.field_behavior) = REQUIRED];

  // The updated metadata.
  GroupMetadata metadata_update = 3 [(google.api.field_behavior) =
REQUIRED];

  // A ClientMlsControlMessage containing a CommitBundle to all users.
  // Optional.  ClientMlsControlRcsMessage client_mls_control_rcs_message =
4
  [(google.api.field_behavior) = OPTIONAL];
}

// The response to a request to update the properties in the Group Chat
// metadata.
message ChangeGroupMetadataResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];
}

// A request to get the ids of all the Group Chats the user is a member of.
```

```
message GetGroupIdsRequest {
  // Request header.
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];
}

// The response to a request to get the ids of all the Group Chats the user
// is a member of.
message GetGroupIdsResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // All the Group Chats that the user is a member of.
  repeated RcsId group_ids = 2 [(google.api.field_behavior) = OPTIONAL];
}

// A request to get information about Group Chats.
message GetGroupInfosRequest {
  // Request header.
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // Ids of all the Group Chats we are requesting information for.
  repeated RcsId group_ids = 2 [(google.api.field_behavior) = OPTIONAL];
}

// The response to a request to get information about Group Chats.
message GetGroupInfosResponse {
  // Response header.
  ResponseHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // Will only contains an entry for Group Chats that the user is a member
  // of.
  repeated GroupInfo group_infos = 2 [(google.api.field_behavior) =
  REQUIRED];
}
// A request to create a Group Chat.
message CreateGroupRequest {
  // Request header.
  RequestHeader header = 1 [
  ];

  // Client generated Group Chat ID.
  RcsId group_id = 2 [
  ];

  // All users of the Group Chat, including the creator.
  repeated RcsId user_ids = 3 [
  ];

  // The metadata of the Group Chat.
  GroupMetadata metadata = 4 [(google.api.field_behavior) = OPTIONAL];
```

```
  // A MLS RCS Group MLS message containing either a CommitBundle or a
  // WelcomeCommitBundle
  ClientMlsControlRcsMessage client_mls_control_rcs_message = 5;
}

// The response to a request to create a Group Chat.
message CreateGroupResponse {
  // Response header.
  ResponseHeader header = 1 [
  ];

  // Contains all the Group Chat information.
  ];
}
```

3.1.6 Spam Reporting Base

```
// A request to report a spam message to the local SPN.
message SpamReportRequest {
  RequestHeader request_header = 1 [(google.api.field_behavior) =
  REQUIRED];

  // The identity of the user who is being reported as spam.
  RcsId reported_id = 2 [(google.api.field_behavior) = REQUIRED];

  // The message ids of the messages that are being reported as spam. Can
  // include between 0 and 10 message IDs.
  repeated string message_id = 3 [(google.api.field_behavior) = REQUIRED];

  // The content of the message reported as spam.
  string content = 4 [(google.api.field_behavior) = REQUIRED];

  // The type of spam being reported. Per spec, this can be set to “spam”,
  // “fraud”, “inappropriate-content”, or “other”. Default of "spam".
  string spam_type = 5 [(google.api.field_behavior) = REQUIRED];

  // A string that can be used by the client to provide additional
  // information about the spam report.
  string free_text = 6 [(google.api.field_behavior) = OPTIONAL];
}

// The response to a request to report a spam message to the local SPN.
message SpamReportResponse {
  ResponseHeader response_header = 1 [(google.api.field_behavior) =
  REQUIRED];
}
```

3.1.7 RCS Video Chat Base

```
// The request to send a video event to the SPN.
// The VideoChatEventNotification is defined in section 3.7.5.2.3.4 of
// [GSMA PRD-RCC.07]
message VideoChatEventNotificationRequest {
```

```
  // Request header.
  RequestHeader header = 1 [(google.api.field_behavior) = REQUIRED];

  // The video chat event to send to the server.
  VideoChatEvent video_chat_event = 2 [(google.api.field_behavior) =
REQUIRED];
}
```

3.2 gRPC gUNI service definitions

This section defines the gUNI gRPC definitions.

3.2.1 AuthTokenService

```
service AuthTokenService {
  // Refreshes an existing authentication token for a user.
  rpc RefreshAuthToken(RefreshAuthTokenRequest)
      returns (RefreshAuthTokenResponse) {
  }
}
```

3.2.2 CapabilitiesService

```
service CapabilitiesService {
  // Fetches information about other user's capabilities on the local SPN.
  //
  // The requester's capabilities are provided in the request and will be
  // validated before the requested user's capabilities are returned.
  //
  // The requester's capabilities may also be stored in the backend
  // capability cache for future use.
  rpc GetCapabilities(GetCapabilitiesRequest)
      returns (GetCapabilitiesResponse) {
  }

  // Sets the capabilities of a user on the local SPN.
  //
  // The requester's capabilities are provided in the request and will be
  // validated before the requested user's capabilities are set.
  //
  // The requester's capabilities may also be stored in the backend
  // capability cache for future use.
  //
  // Sets the capabilities of a list of users.
  rpc SetCapabilities(SetCapabilitiesRequest)
      returns (SetCapabilitiesResponse) {
  }
}
```

3.2.3 MessagingService

```
service MessagingService {

  // Send a message to a user on a local SPN.
```

```
  rpc SendMessage(SendMessageRequest) returns (SendMessageResponse) {
  }

  // Best-effort attempt to delete an undelivered message
  // on a local SPN.
  rpc RevokeMessage(RevokeMessageRequest) returns (RevokeMessageResponse) {
  }

  // Pull all un-acked messages from the local SPN.
  //
  // This method is paginated. When a response has pulled_all == false,
  // it should ack the messages and then call this method again.
  rpc PullMessages(PullMessagesRequest) returns (PullMessagesResponse) {
  }

  // Establishes a server to client stream for receiving messages and
  // events from the local SPN.
  rpc Bind(BindRequest) returns (stream BindEvent) {
  }


  // Prewarms the connection for a given set of contacts on the local SPN.
  rpc Prewarm(PrewarmRequest) returns (PrewarmResponse) {
  }

  // Acks the messages that were received from the server in a BindEvent.
  rpc AckMessages(AckMessagesRequest) returns (AckMessagesResponse) {
  }
}
```

3.2.4 GroupService

```
service GroupService {
  // Create a Group Chat on a local SPN only. All Group
  // members will receive a [CreateGroupNotification].
  rpc CreateGroup(CreateGroupRequest) returns (CreateGroupResponse) {
  }


  // Add users to a Group Chat on a local SPN. All existing Group Chat
  // members will receive a [AddGroupUsersNotification] and the new members
  // receive a [CreateGroupNotification].
  rpc AddGroupUsers(AddGroupUsersRequest) returns (AddGroupUsersResponse) {
  }

 // Removes users from a Group Chat on a local SPN. All Group Chat
 //members (including removed) will receive a
[RemoveUsersFromGroupNotification].
  rpc RemoveGroupUsers(RemoveGroupUsersRequest)
      returns (RemoveGroupUsersResponse) {
  }

  // Updates the properties in the Group Chat metadata on a local SPN.
  // All Group Chat members will receive a
```

```
  // ChangeGroupMetadataNotification].
  rpc ChangeGroupMetadata(ChangeGroupMetadataRequest)
      returns (ChangeGroupMetadataResponse) {
  }

  // Retrieves the ids of all the Group Chats the user represented by
  // RequestHeader.requester_id is a member of on a local SPN.
  rpc GetGroupIds(GetGroupIdsRequest) returns (GetGroupIdsResponse) {
  }

  // Retrieves information about Group Chats the user represented by
  // RequestHeader.requester_id is a member of on a local SPN.
  rpc GetGroupInfos(GetGroupInfosRequest) returns (GetGroupInfosResponse) {
  }
}
```

3.2.5 SpamReportingService

```
service SpamReportingService {
  // Report a spam message to the local SPN
  rpc SpamReport(SpamReportRequest) returns (SpamReportResponse) {
  }
}
```

3.2.6 MlsService

```
service MlsService {
  // Apply a ClientMlsControlMessage to an existing MLS group.
  rpc ApplyMlsControlMessage(ApplyMlsControlMessageRequest)
      returns (ApplyMlsControlMessageResponse) {
  }

  // Fetches information about an existing MLS group.
  rpc GetMlsGroupInfo(GetMlsGroupInfoRequest)
      returns (GetMlsGroupInfoResponse) {
  }

  // Creates a new MLS group.
  rpc CreateMlsGroup(CreateMlsGroupRequest) returns
  (CreateMlsGroupResponse) {
}
```

3.2.7 VideoChatService

```
// This API allows RCS clients E2EE to notify the SPN of events during the
// lifecycle of a Video Chat.
service VideoService {
  rpc SendVideoChatEventNotification(VideoChatEventNotificationRequest)
    returns (VideoChatEventNotificationResponse) {
    option security_level = PRIVACY_AND_INTEGRITY;
    option (google.api.http) = {
      post: "/v1/video:sendVideoChatEventNotification"
      body: "*"
    };
  }
}
```

3.3 gRPC gNNI service definitions

This section defines the gUNI gRPC definitions. All services on the gNNI interface do not require the AuthToken.

3.3.1 CapabilitiesService

```
service CapabilitiesService {
  // Fetches information about other user's capabilities on a remote SPN.
  //
  // The requester's capabilities are provided in the request and will be
  // validated before the requested user's capabilities are returned.
  //
  // The requester's capabilities may also be stored in the backend
  // capability cache for future use.
  rpc GetCapabilities(GetCapabilitiesRequest)
      returns (GetCapabilitiesResponse) {
  }

  // Sets the capabilities of a user on the remote SPN.
  //
  // The requester's capabilities are provided in the request and will be
  // validated before the requested user's capabilities are set.
  //
  // The requester's capabilities may also be stored in the backend
  // capability cache for future use.
  //
  // Sets the capabilities of a list of users.
  rpc SetCapabilities(SetCapabilitiesRequest)
      returns (SetCapabilitiesResponse) {
  }
}
```

3.3.2 MessagingService

```
service MessagingService {

  // Send a message to a user on a remote SPN.
  rpc SendMessage(SendMessageRequest) returns (SendMessageResponse) {
  }

  // Best-effort attempt to delete an undelivered message
  // on a local or remote SPN.
  rpc RevokeMessage(RevokeMessageRequest) returns (RevokeMessageResponse) {
  }

  // Prewarms the connection for a given set of contacts.
  rpc Prewarm(PrewarmRequest) returns (PrewarmResponse) {
  }

}
```

3.3.3 GroupService

```
service GroupService {

  // Add users to a Group Chat on a remote SPN.
  // All existing Group Chat members will
  // receive a [AddGroupUsersNotification] and the new members receive a
  // [CreateGroupNotification].
  rpc AddGroupUsers(AddGroupUsersRequest) returns (AddGroupUsersResponse) {
  }

  // Removes users from a Group Chat on a remote SPN.
  // All Group Chat members (including removed) will receive a
  // [RemoveGroupUsersNotification].
  rpc RemoveGroupUsers(RemoveGroupUsersRequest)
      returns (RemoveGroupUsersResponse) {
  }

  // Updates the properties in the Group Chat metadata
  // on a remote SPN.
  // All Group Chat members will receive a
  // [ChangeGroupMetadataNotification].
  rpc ChangeGroupMetadata(ChangeGroupMetadataRequest)
      returns (ChangeGroupMetadataResponse) {
  }

  // Retrieves the ids of all the Group Chats the user represented by
  // RequestHeader.requester_id is a member of on a remote SPN.
  rpc GetGroupIds(GetGroupIdsRequest) returns (GetGroupIdsResponse) {
  }

  // Retrieves information about Group Chats the user represented by
  // RequestHeader.requester_id is a member of on a remote SPN.
  rpc GetGroupInfos(GetGroupInfosRequest) returns (GetGroupInfosResponse) {
  }
}
```

3.3.4 MlsService

```
service MlsService {
  // Apply a ClientMlsControlMessage to an existing MLS group.
  rpc ApplyMlsControlMessage(ApplyMlsControlMessageRequest)
      returns (ApplyMlsControlMessageResponse) {
  }

  // Fetches information about an existing MLS group.
  rpc GetMlsGroupInfo(GetMlsGroupInfoRequest)
      returns (GetMlsGroupInfoResponse) {
  }

  // Creates a new MLS group.
  rpc CreateMlsGroup(CreateMlsGroupRequest
      returns (CreateMlsGroupResponse) {
  }
}
```

4 RCS gRPC gUNI Specific Procedures

4.1 Provisioning

4.1.1 Configuration and Token Provisioning

The parameter transport_protocol defined in Table 2 shall be used to indicate what transport protocol the RCS client can support.

Parameter	Description	Mandatory	Format
transport_protocol	RCS client shall send what it can support: **0**: Can only support SIP/MSRP **1**: Can support SIP/MSRP and RCS gRPC **2**: Can only support RCS gRPC	Y	Int (0,1,2)

Table 2: RCS gRPC Parameters

4.1.2 Configuration Procedures

4.1.2.1 Client Procedures

The RCS client shall indicate type of protocol it can support by including the `transport_protocol` defined in 4.1.1.

If the RCS client can only support SIP/MSRP, then it shall send 0 for the `transport_protocol` parameter.

If the RCS client can support both SIP/MSRP and RCS gRPC, it shall send 1 for the `transport_protocol` parameter.

If the RCS client can only support RCS gRPC, it shall send 2 for the `transport_protocol` parameter.

If the RCS client used value 1 for the `transport_protocol` parameter, it shall abide by whatever the SPN chooses to configure. If the SPN returns the configuration tree specified in A.1, then the RCS client shall configure RCS gRPC. If, however, the SPN chooses to return the configuration tree specified in defined in A.1.6.1 and A.1.6.2 of [[GSMA PRD-RCC.07]], then it shall configure SIP/MSRP. The RCS client shall honour any changes in the configuration parameters in subsequent configuration as per section 2.6.1 in [[GSMA PRD-RCC.14]].

4.1.2.2 SPN Procedures

If the value of `transport_protocol` is set to 0 or is absent from the initial service configuration request sent by the RCS client, then the SPN shall assume the RCS client can only support SIP/MSRP for the transport protocol as per [[GSMA PRD-RCC.14]].

If the value of `transport_protocol` is set to 2, the SPN shall assume the RCS client shall only support RCS gRPC for the transport protocol as defined by this document.

If the value of `transport_protocol` is set to 1, the SPN may choose what transport protocol to use, either SIP/MSRP or RCS gRPC. The SPN shall be able to change the transport protocol on subsequent configuration as per section 2.6.1 in [[GSMA PRD-RCC.14]].

If the transport protocol is RCS gRPC, the SPN shall return the parameters in the configuration document as defined in A.1. If, however, the transport protocol is SIP/MSRP, the SPN shall return the parameters defined in A.1.6.1 and A.1.6.2 in [[GSMA PRD-RCC.07]]. The RCS client shall configure whichever transport protocol is returned by the SPN.

4.2 General Procedures

For any gRPC invocation, RCS clients shall connect to the FQDN specified in the RCS GRPC ENDPOINT client configuration parameter defined in section A.1.1. The client shall construct the request protocol buffer and must include the request header defined in section 3.1.1. The request header must include the authentication token retrieved from section 4.1.1 or updated as per section 4.3.1.

The SPN shall check the validity of the auth token, and if it is invalid, return a `GRPC_STATUS_UNAUTHENTICATED` gRPC error back to the client.

If the SPN responds with a `GRPC_STATUS_UNAUTHENTICATED` gRPC error to any request, the client shall renew the AuthToken as per 4.3.1 and retry the request.

NOTE: The retry logic is left up to the RCS client.

4.2.1 gRPC Method Timeout

All gRPC clients (whether the RCS client or SPN calling over gNNI) shall set the deadline for all gRPC methods, as outlined in [gRPC], at 15*Timer_T1, which is defined in A.1.1.

When the gRPC methods exceed the deadline, the gRPC framework will throw a DEADLINE_EXCEEDED gRPC error code, which the RCS client shall handle as per procedures for the gRPC method.

4.2.2 Exponential Retry

Upon receiving any retriable failure, the RCS client shall employ an exponential backoff algorithm to retry the gRPC method. The initial retry interval shall be 15*Timer_T1.

4.3 Token management

After receiving the initial auth token in the configuration document in the parameter RCS GRPC TOKEN as per section 4.1.1, the RCS client shall renew the auth token within the interval specified in the RCS GRPC TOKEN REFRESH parameter as per section 4.3.1. Similarly, after receiving the auth token using the procedure in 4.3.1, RCS client shall renew the auth token within the interval specified in the RCS GRPC TOKEN REFRESH parameter using the same procedure defined in 4.3.1.

If the auth token expires before the RCS client is able to refresh it as per 4.3.1, the RCS client should still attempt the procedure in section 4.3.1. If the token refresh procedure in

section 4.3.1 fails, the client shall issue a configuration request as per section 2.6.1 in GSMA[GSMA PRD-RCC.14] to retrieve a new auth token.

4.3.1 Token Refresh

To refresh the auth token, the RCS client shall:

1. Create the AuthTokenTBS protocol buffer as per section 3.1.2. The timestamp included must be in network time as per [RFC9505].
2. Sign the AuthTokenTBS protocol buffer with the private key associated with the public key provided during the provisioning process with the parameter client_certificate_upload defined in section 2.12 of [GSMA PRD-RCC.14] .
3. Include the AuthTokenTBS and the signature in the RefreshAuthTokenRequest protocol buffer as per section 3.1.2.
4. Execute the RefreshAuthToken RPC defined in section 3.1.2.

When the SPN receives a RefreshAuthToken RPC, it shall:

5. Verify the signature of the AuthTokenTBS and ensure the AuthToken is still valid.
6. Create a new AuthToken.
7. Create the RefreshAuthTokenResponse protocol buffer as per section 3.1.2.
8. Respond to the RefreshAuthToken RPC with the RefreshAuthTokenResponse.

After the RCS client receives a new auth token as per the RefreshAuthToken gRPC call, the old token shall be considered invalid by the RCS client and SPN.

NOTE: The RCS client cannot use an auth token while calling RefreshAuthToken gRPC.

If the verification of the AuthTokenTBS signature or the AuthToken fails, the SPN shall fail the RefreshAuthToken RPC with a `GRPC_STATUS_UNAUTHENTICATED` gRPC error. The RCS client must issue a configuration request as per section 2.6.1 in [GSMA PRD-RCC.14] to retrieve a new auth token.

The flow is illustrated in Figure 2.

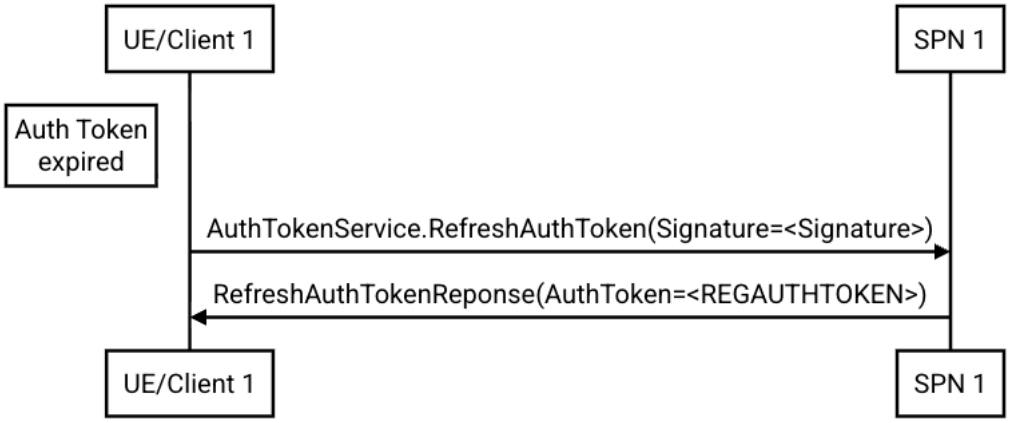

Figure 2: RefreshAuthToken Call

4.4 PullMessages

PullMessage is a mechanism for the RCS client to retrieve store and forward messages intended for the user. It allows for a batch operation where the SPN can deliver potentially all messages in one gRPC method, or if there are a large number of messages, paginate the responses. The RCS client would then use the AckMessage gRPC method defined in 4.6 to remove the messages from store and forward.

To check if there are pending messages, the RCS client shall:

1. Construct the PullMessageRequest protocol buffer as per section 3.1.4.
2. Call the PullMessage gRPC method defined in section 4.4.

When the SPN receives a PullMessage gRPC, it shall:

3. Check that the auth token is still valid as per section 4.2.
4. Check for undelivered MessagingEvent(s) for the caller in the store and forward.
5. Include some or all of the MessagingEvents inside the PullMessagesResponse as defined in section 3.1.4.
   a. If the SPN includes all the messages or there are no messages to be delivered, it shall set the PullStatus to `PULL_STATUS_COMPLETE`.
   b. If the SPN includes some but not all the messages, it shall set the PullStatus to `PULL_STATUS_INCOMPLETE`.
6. Respond to the PullMessages gRPC with the PullMessagesResponse.

When receiving a PullMessagesResponse, the RCS client shall:

7. Call AckMessage gRPC method for all MessagingEvents received as per 4.6.
8. If the PullStatus in the PullMessagesRespose is set to `PULL_STATUS_INCOMPLETE` then repeat procedures starting from step 1 until PullStatus is `PULL_STATUS_COMPLETE`.

If the RCS client receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the PullMessage gRPC method as per section 4.2.2.

The procedure is illustrated in Figure 3.

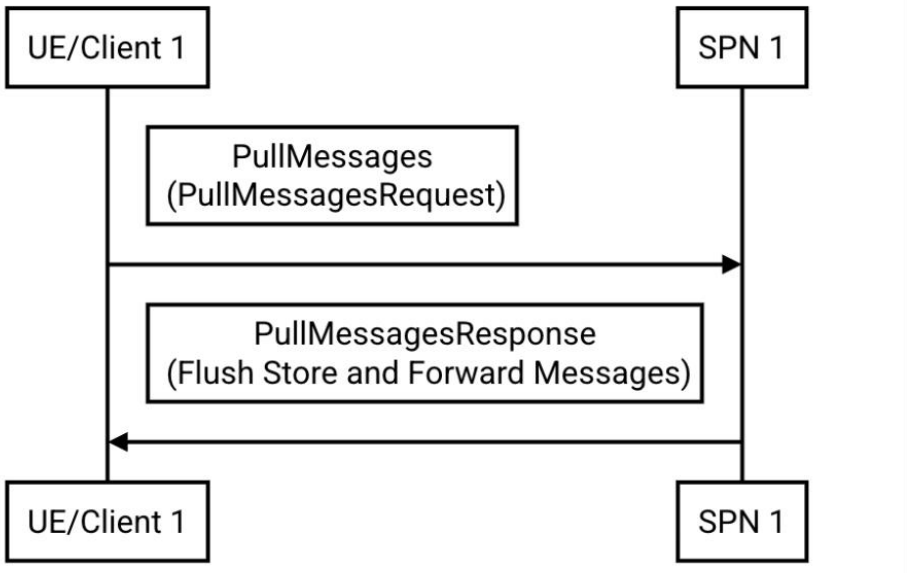

Figure 3: PullMessage

4.5 Bind

Bind is a uni-directional streaming gRPC that allows for an optimized delivery of inflight messages. The RCS client opens the bind channel, and the server can deliver messages (whether in store or forward, or incoming) via the opened channel.

Upon start-up, or when it receives a push notification as per section 2.15 of [[GSMA PRD-RCC.07]], the RCS client shall:

1. Construct the BindRequest protocol buffer as per section Messaging Base.
2. Call the Bind gRPC stream defined in section 3.2.3.

When the SPN receives a bind gRPC, it shall:

3. Check that the auth token is still valid as per section 4.2.
4. Check for undelivered MessagingEvent(s) for the caller.
5. For each message, construct the BindEvent protocol buffer containing the undelivered MessagingEvent protocol buffer as per section 3.1.4.
6. Send the BindEvent(s) over the bind stream.

When the RCS client receives a BindEvent with a MessagingEvent inside, it shall:

7. Call AckMessage method for all messages as per section 4.6.

If the RCS client receives a DEADLINE_EXCEEDED or INTERNAL errors, it shall instead call PullMessage gRPC method as per section 4.2.

The procedure is illustrated in Figure 4.

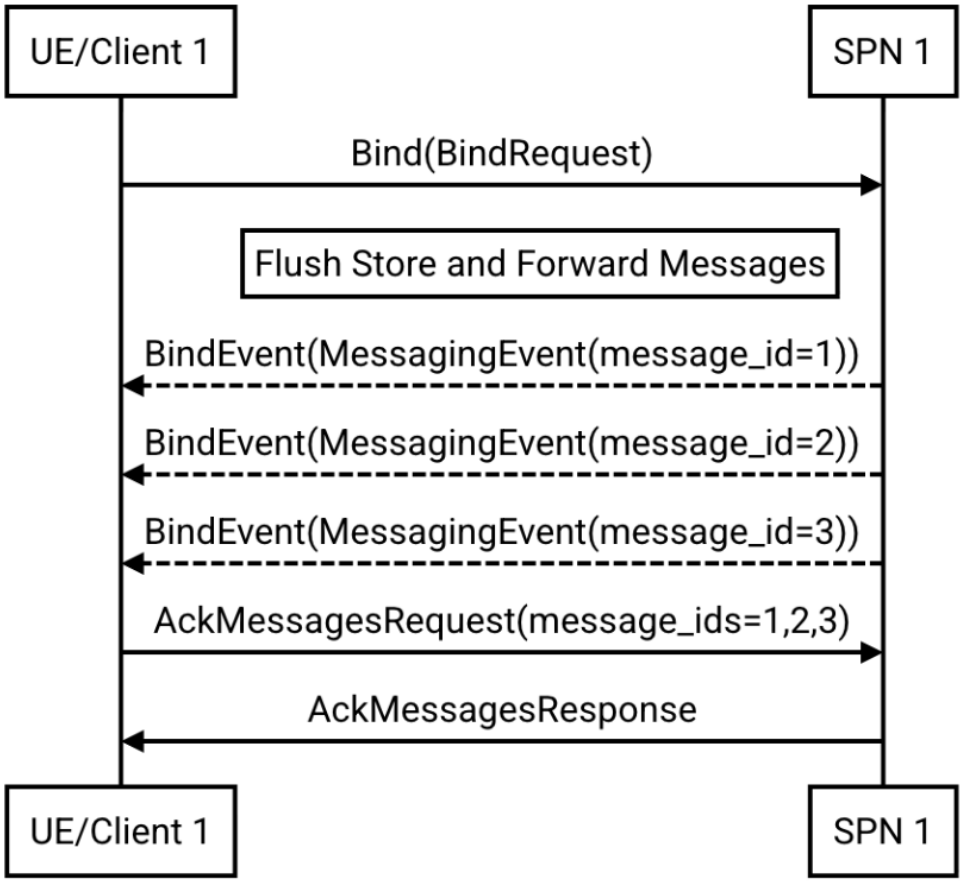

Figure 4: Bind

4.5.1 Receiving a message

When the SPN receives a MessagingEvent while the bind channel is open, the SPN shall:

1. Wrap the MessagingEvent in a BindEvent.
2. Send the BindEvent over the bind stream.

When the RCS client receives a BindEvent with a MessagingEvent inside, it shall:

3. Call AckMessage method for all messages as per section 4.6.

If the SPN discovers that the bind channel is closed while attempting to deliver the message, it shall issue a push notification to the recipient as per section 2.15 of [[GSMA PRD-RCC.07]] and store the message in store and forward.

The procedure is illustrated in Figure 5.

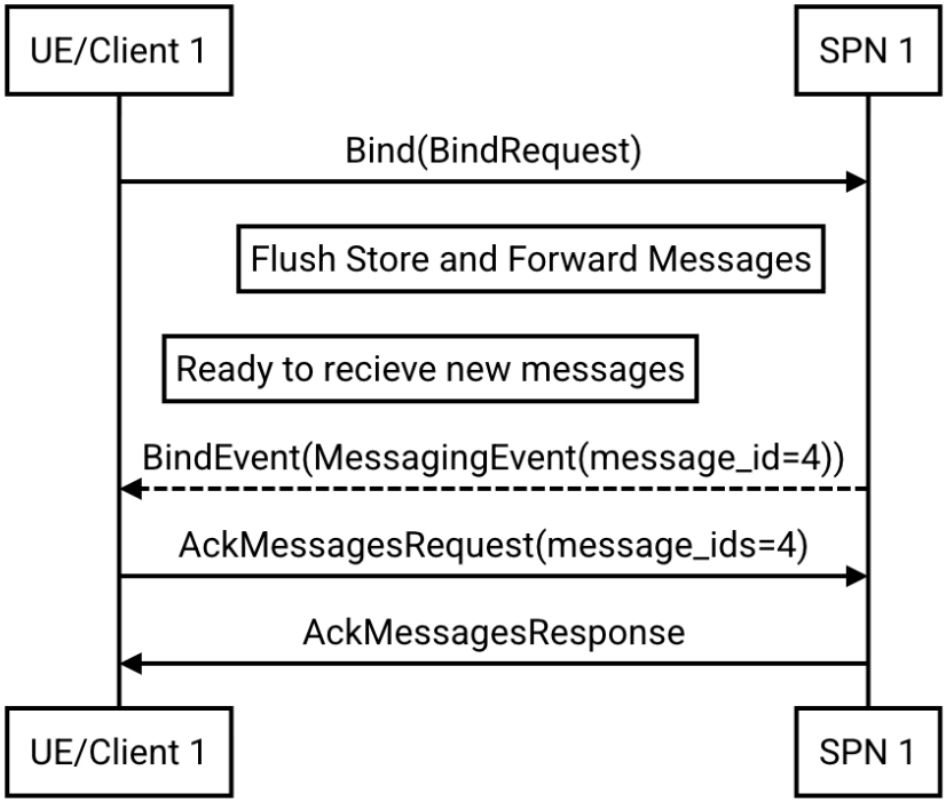

Figure 5: Receiving a Message after Bind

4.5.2 Keeping a connection alive

The SPN may choose to keep the bind channel alive either because it received a prewarm request as per section 5.2.2, or for any other optimization.

To keep the bind channel alive, the SPN may:

1. Construct the BindEvent protocol buffer containing the KeepAliveEvent buffer as per section 3.1.4.
2. Send the BindEvent over the bind stream.

Upon receiving the BindEvent the RCS client shall discard the event.

If the SPN discovers that the bind channel is closed while attempting to keep the bind channel alive, it shall issue a push notification to the recipient as per section 2.15 of [[GSMA PRD-RCC.07]] and store the message in store and forward.

The procedure is illustrated in Figure 6.

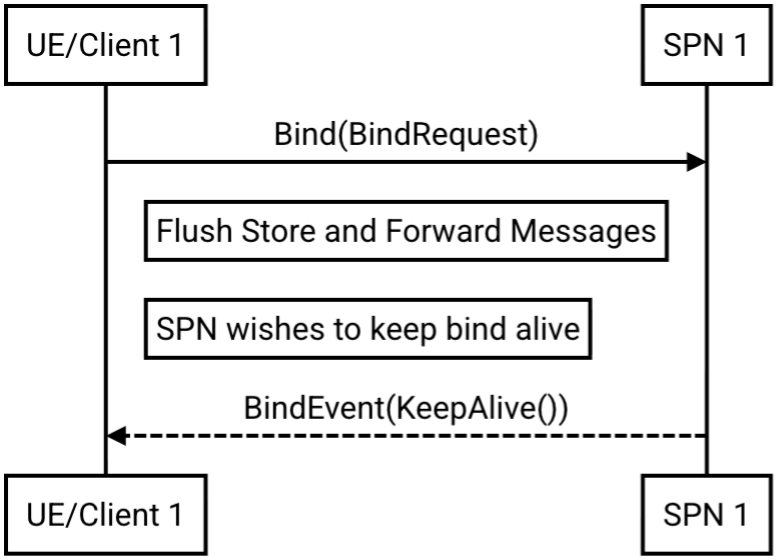

Figure 6: Bind with KeepAlive

4.6 AckMessage

To remove delivered messages from the store and forward or acknowledge receipt of inflight messages, the RCS client calls the AckMessage gRPC method.

For each MessagingEvent the RCS client receives, it shall:

1. Construct an AckMessagesRequest protocol buffer containing the list of messages_ids (as per section 3.1.4) to be acknowledged.
2. Call the AckMessages gRPC method defined in section 3.2.3.

Upon receiving the AckMessagesRequest, the SPN shall:

3. Check that the auth token is still valid as per section 4.2.
4. Remove all MessagingEvents identified by the message_ids in the store and forward.
5. Construct the AckMessageResponse protocol buffer as per section 3.1.4.
6. Respond to the AckMessages gRPC with the AckMessagesResponse.

If the RCS client receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the AckMessage gRPC method as per section 4.2.1.

This is illustrated in Figure 7.

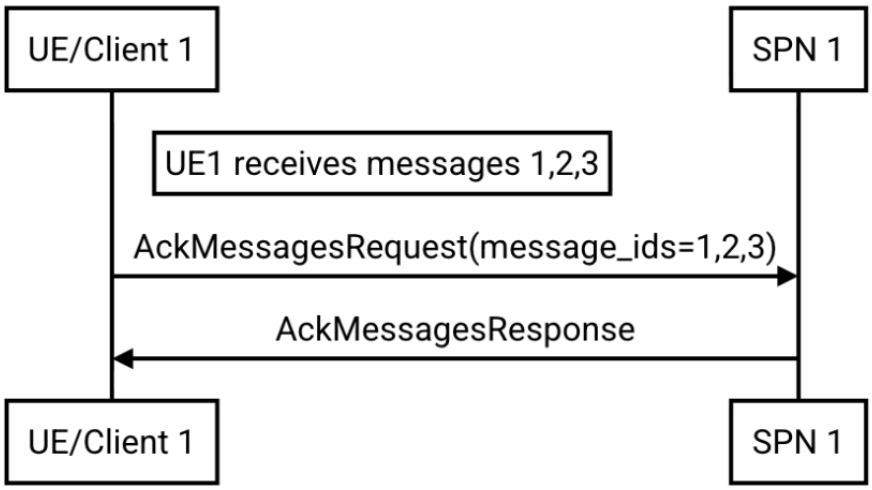

Figure 7: AckMessage

4.7 Spam Reporting

To report spam messages, the RCS client shall:

1. Construct the SpamReportRequest protocol buffer including the spam contents, message id(s), spam type and the reported id as per section 3.1.6.
2. Call the SpamReport gRPC method defined in section 3.2.5.

When the SPN receives a SpamReport gRPC, it shall:

3. Check that the auth token is still valid as per section 4.2.
4. Construct the SpamReportResponse protocol buffer as per section 3.1.6.
5. Respond to the SpamReport gRPC with the SpamReportResponse.

If the RCS client receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the SpamReport gRPC method as per section 4.2.1.

The flow is illustrated in Figure 8.

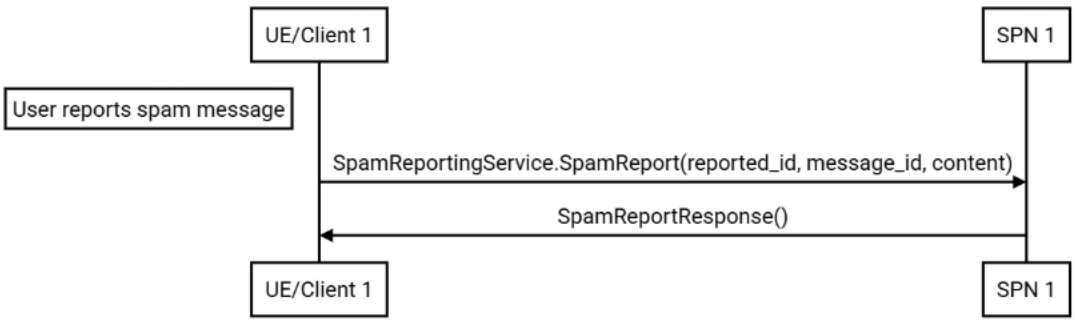

Figure 8: Spam reporting

4.8 Chatbot

All Chatbot procedures, other than Messaging as defined in section 5.2, and capabilities as defined in section 5.1 shall follow procedures defined in [[GSMA PRD-RCC.07]].

NOTE: The Chatbot platform to SPN interface is for further study.

4.9 RCS Video Chat Event Reporting

To send an RCS Video Chat event notification as per section 3.7.5.2.5 of [GSMA PRD-RCC.07], the RCS client shall:

1. Construct the VideoChatEventNotificationRequest protocol buffer as per section 3.1.7:
    a. The VideoChatEventNotificationRequest.video_chat_event shall be constructed as per section 3.7.5.2.5 of [GSMA PRD-RCC.07].
2. Execute the SendVideoChatEventNotification RPC defined in section 3.2.7.

When the SPN receives a SendVideoChatEventNotification RPC, it shall:

3. Check that the AuthToken is still valid as per section 4.2
4. Construct a VideoChatEventNotificationResponse protocol buffer as per section 3.1.7.
5. Respond to the sender's RCS client with the VideoChatEventNotificationResponse.

5 RCS gRPC Generic Procedures

5.1 Capability Discovery

Since push notification mechanism is mandatory for RCS gRPC, SPNs shall cache all capabilities.

5.1.1 Setting Capabilities

To set capabilities, the RCS client shall:

1. Create the SetCapabilitiesRequest protocol buffer with all client capabilities as per section 3.1.3.
2. Execute the SetCapabilities RPC defined in section 3.2.2.
3. Check that the AuthToken is still valid as per section 4.2.

When the SPN receives a SetCapabilities RPC, it shall:

4. Store the capabilities in the SPN cache.
5. Create the SetCapabilitiesResponse protocol buffer as per section 3.1.3.
6. Respond to the SetCapabilities RPC with the SetCapabilitiesResponse.

The flow is illustrated in Figure 9.

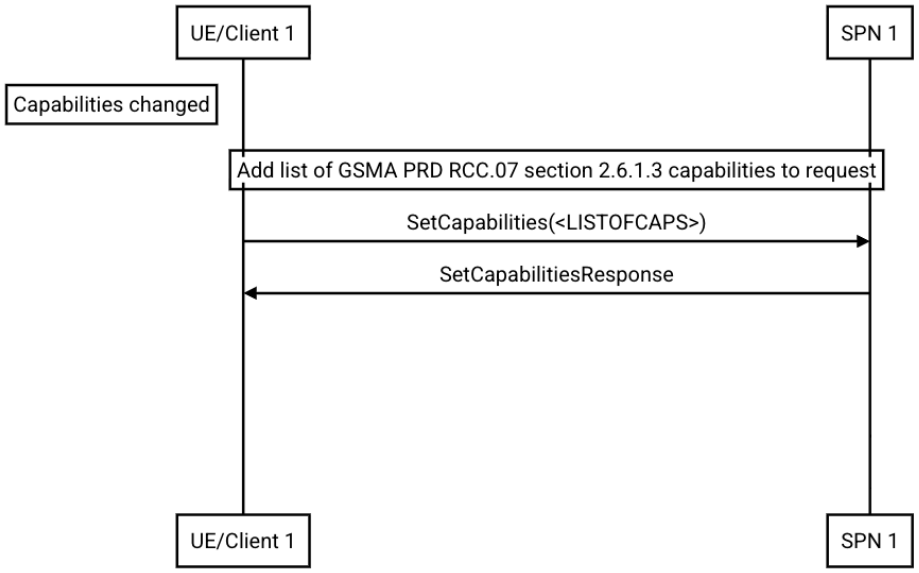

Figure 9: SetCapabilities Call

5.1.2 Getting Capabilities

To get capabilities for another user, the RCS client shall:

1. Create the GetCapabilitiesRequest protocol buffer as per section 3.1.3.
2. Execute the GetCapabilities RPC defined in section 3.2.2.

When the SPN receives a GetCapabilities RPC, it shall:

3. Check that the AuthToken is still valid as per section 4.2.
4. For each user, retrieve capabilities for each user on the network.
   a. If the user exists, set UserCapabilities and set `CAPABILITIES_RESULT_OK` for that user.
   b. If the user does not exist, include `CAPABILITIES_RESULT_NOT_FOUND` error for that user.

5. If the user is on another SPN:
   a. Remove the auth token from the GetCapabilitiesRequest.
   b. Issue a GetCapabilitiesRequest towards the SPN owning the user.
   c. Once the gNNI SPN responds, include the UserCapabilities from the GetCapabilitiesResponse.

6. Create the GetCapabilitiesResponse protocol buffer with the UserCapabilities as per section 3.1.3.
7. Respond to the GetCapabilities RPC with the GetCapabilitiesResponse.

When an SPN receives a GetCapabilities call over the gNNI, the SPN shall:

8. For each user, retrieve capabilities for each user on the network.
   a. If the user exists, set UserCapabilities and set `CAPABILITIES_RESULT_OK` for that user.
   b. If the user does not exist, include `CAPABILITIES_RESULT_NOT_FOUND` error for that user.

9. Create the GetCapabilitiesResponse protocol buffer with the UserCapabilities as per section 3.1.3.
10. Respond to the GetCapabilities RPC with the GetCapabilitiesResponse.

The flow is illustrated in Figure 10.

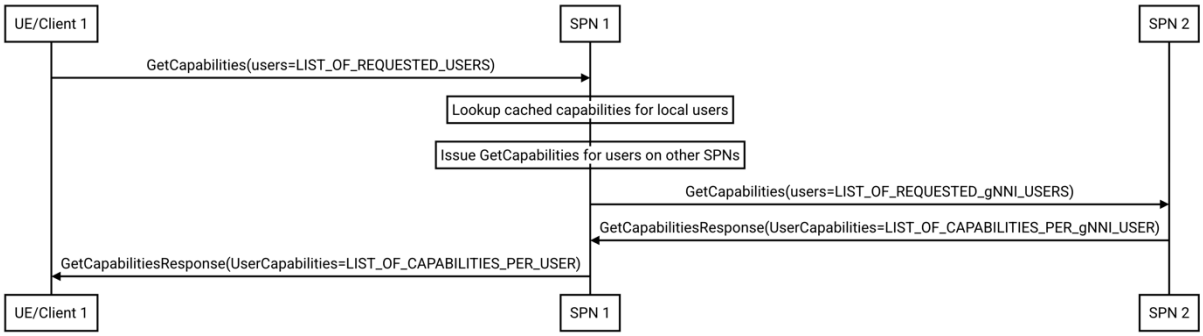

Figure 10: GetCapabilities Call

5.2 Messaging

5.2.1 Message sending

5.2.1.1 1-to-1 Conversation

When sending a 1-to-1 Chat message, the RCS client shall:

1. Construct the MessagingEvent protocol buffer representation of the message as per section 3.1.4.
2. Construct the SendMessageRequest protocol buffer as per section 3.1.4.
3. Call the SendMessage gRPC defined in section 3.2.3.

When receiving a SendMessage gRPC, the SPN shall:

4. Validate the auth token as per 4.2.
5. If the recipient is over NNI, the SPN shall:
   a. Construct the SendMessageRequest from the incoming gRPC.
   b. Ensure the MessagingEvent.sender_id field matches the sender specified in the auth token.
   c. Locate the gRPC endpoint for the recipient's SPN as per section 5.4.
   d. Call the recipient's SPN SendMessage gRPC method.
   e. Wait for the SendMessageResponse from the recipient's SPN.
      i. If a gRPC error is received from the recipient's SPN, propagate the gRPC error to the RCS client.
6. Construct a SendMessageResponse protocol buffer as per section 3.1.4.
7. Respond to the sender's RCS client with the SendMessageResponse.

When receiving a SendMessage gRPC, the recipient's SPN shall:

8. If the recipient is not on RCS, return a NOT_FOUND gRPC error. Otherwise,
9. If bind channel is open to the recipient, send the message to them as per section 4.5.1.
10. Otherwise, send a push notification to the recipient as per section 2.15 of [[GSMA PRD-RCC.07]] and store the message in store and forward.
11. Construct a SendMessageResponse protocol buffer as per section 3.1.4.
12. Respond to the SendMessage gRPC with the SendMessageResponse.

If the RCS client receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the SendMessage gRPC method as per section 4.2.1.

If the RCS client receives a NOT_FOUND gRPC error code, it shall fallback to xMS as per section 3.2.3.8 in [[GSMA PRD-RCC.07]].

The flow is illustrated in Figure 11.

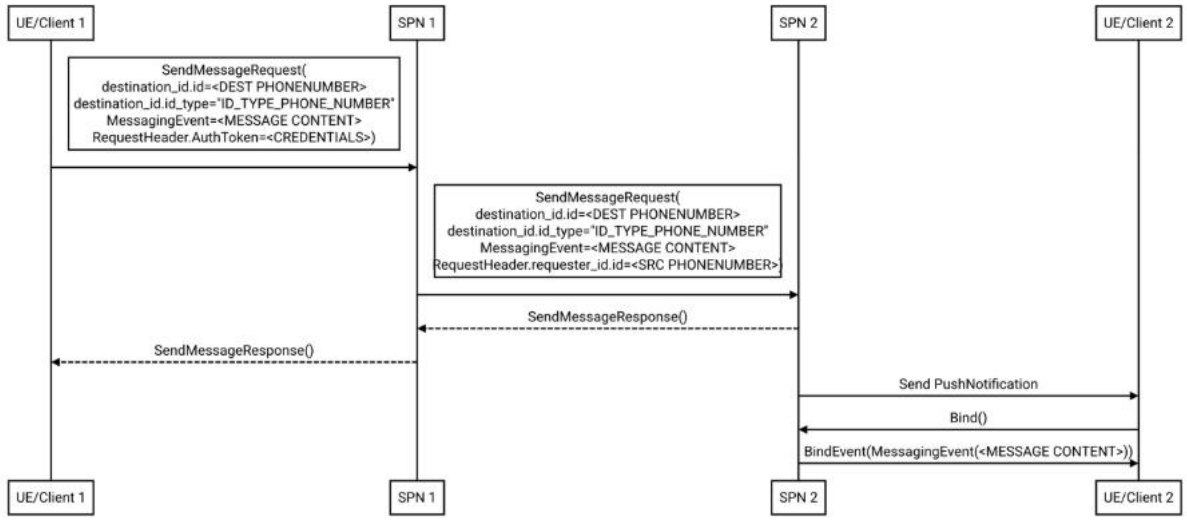

Figure 11: 1-to-1 Message Sending

5.2.1.2 Group Chat

When sending a Group Chat message, the RCS client shall:

1. Construct the MessagingEvent protocol buffer representation of the message as per section 3.1.4.
2. Construct the SendMessageRequest protocol buffer as per section 3.1.4.
3. Call the SendMessage gRPC defined in section 3.2.3.

When receiving a SendMessage gRPC, the Participating Function shall:

4. Validate the auth token as per 4.2.
5. Ensure the MessagingEvent.sender_id field matches the sender specified in the auth token.
6. Locate the gRPC endpoint for the Controlling Function for the Group Chat from local storage.
7. Call the Controlling Function's SendMessage gRPC method with the SendMessageRequest.
8. Wait for the SendMessageResponse from the Controlling Function. If the Controlling Function returns a gRPC error code, propagate the error code to the RCS client.
9. Construct the SendMessageResponse.
10. Respond to the sender's RCS client with the SendMessageResponse.

When receiving a SendMessage gRPC, the Controlling Function shall:

11. Check if the Group Chat exists in local storage, if not, return NOT_FOUND gRPC error code.
12. Check if the sender is a member of the Group Chat, if not, return PERMISSION_DENIED. Otherwise,
13. Lookup participants of Group in local storage, for each recipient, construct a SendMessageRequest with:

   a. Setting the receiver RcsId as per section 2.1.1.
   b. Setting the group_id RcsId as per section 2.1.2.
14. Locate the gRPC endpoint for the recipient's SPN as per section 5.4.
15. Call the SendMessage gRPC method on the recipient's SPN.
16. Wait for the SendMessageResponse from the recipient's SPN.
   a. If the recipient's SPN return a NOT_FOUND gRPC error, remove the participant from local storage, and send a RemoveUserFromGroupNotification as per section 5.3.4.
17. Construct the SendMessageResponse.
18. Respond to the Participating Function with the SendMessageResponse.

When receiving a SendMessage gRPC, the recipient's SPN shall:

19. If the recipient is not on RCS, return a NOT_FOUND gRPC error. Otherwise,
20. If bind channel is open to the recipient, send the message to them as per section 4.5.1.
21. Otherwise, send a push notification to the recipient as per section 15 of [[GSMA PRD-RCC.07]] and store the message in Store and Forward.
22. Construct a SendMessageResponse protocol buffer as per section 3.1.4.
23. Respond to the SendMessage gRPC from the Controlling Function with the SendMessageResponse.

If the RCS client receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the SendMessage gRPC method as per section 4.2.1.

If the RCS client receives a NOT_FOUND gRPC error code, it shall re-create the group as per section 5.3.1.

If the RCS client receives a PERMISSION_DENIED gRPC error code, it shall disallow the user from sending any further messages to the Group Chat.

The flow is illustrated in Figure 12.

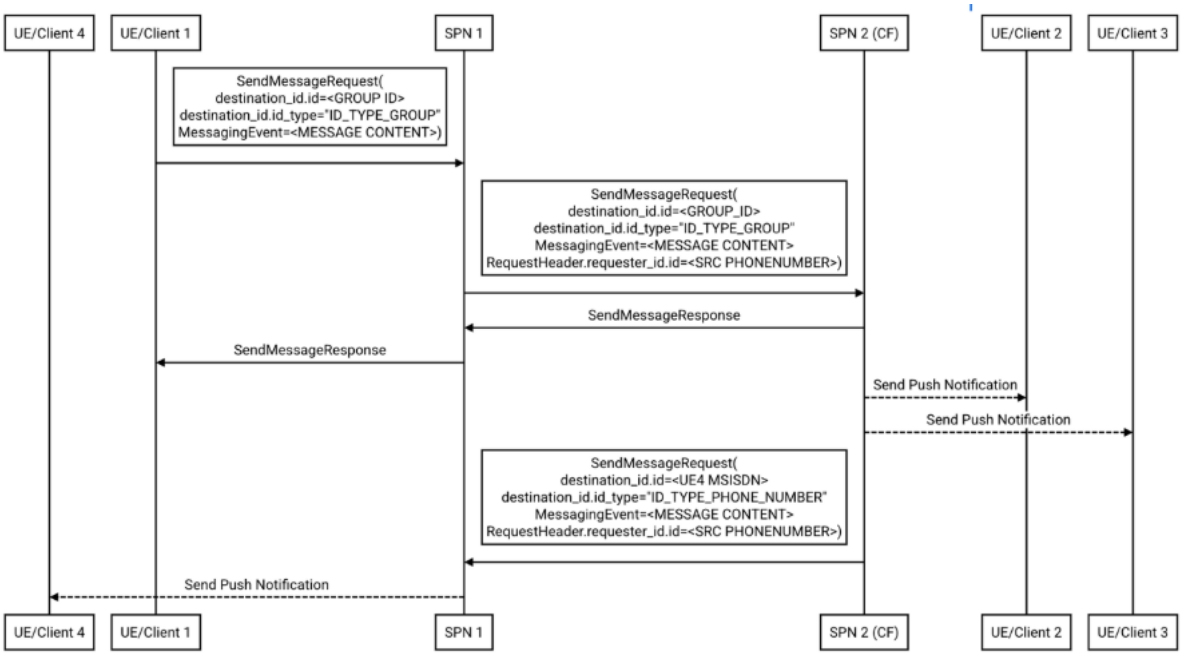

Figure 12: Group Chat Messaging

5.2.1.3 Chatbot

When sending a message to the Chatbot, the RCS client shall:

1. Construct the MessagingEvent protocol buffer representation of the message as per section 4.6.
2. Construct the SendMessageRequest protocol buffer as per section 3.1.4.
3. Call the SendMessage gRPC defined in section 3.2.3.

When receiving a SendMessage gRPC, the SPN shall:

4. Validate the auth token as per 4.2.
5. Forward the message to the Chatbot Platform.

If the RCS client receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the SendMessage gRPC method as per section 4.2.1.

The flow is illustrated in Figure 13.

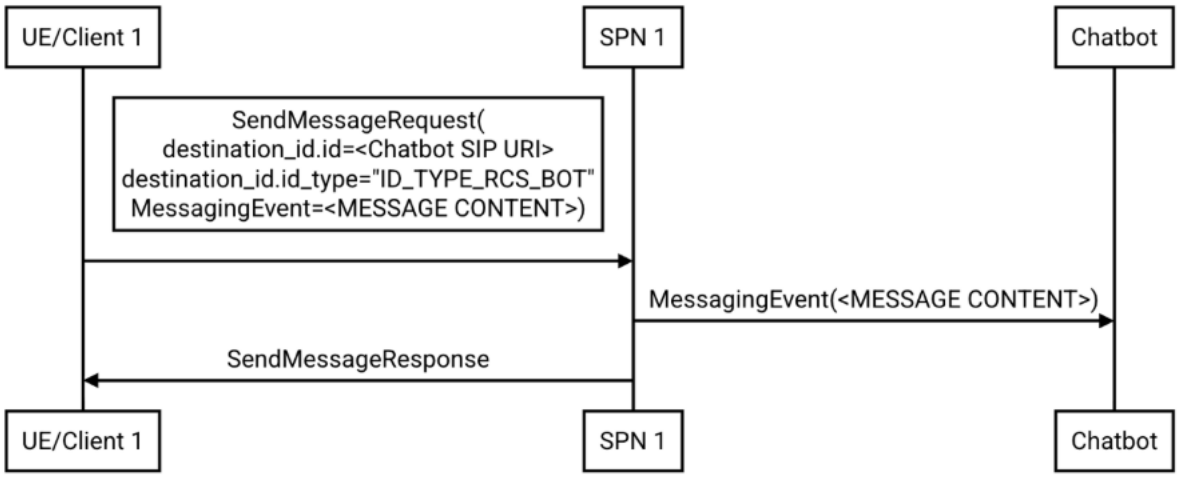

Figure 13: RCS Client to Chatbot

When sending a message to the RCS client, the Chatbot Platform shall:

1. Set the MessagingEvent.receiver_id as per section 2.1.1.
2. Set the MessagingEvent.sender_id as per section 2.1.3
3. Construct the MessagingEvent protocol buffer representation of the message as per section 4.6.
4. Construct the SendMessageRequest protocol buffer as per section 3.1.4.
5. Call the SendMessage gRPC defined in section 3.2.3.

When receiving a SendMessage gRPC, the SPN shall:

6. If the recipient is not on RCS, return a NOT_FOUND gRPC error. Otherwise,
7. If bind channel is open to the recipient, send the message to them as per section 4.5.1.
8. Otherwise, send a push notification to the recipient as per section 2.15 of [[GSMA PRD-RCC.07]] and store the message in Store and Forward.
9. Construct a SendMessageResponse protocol buffer as per section 3.1.4.
10. Respond to the SendMessage gRPC with the SendMessageResponse.

The flow is illustrated in Figure 14.

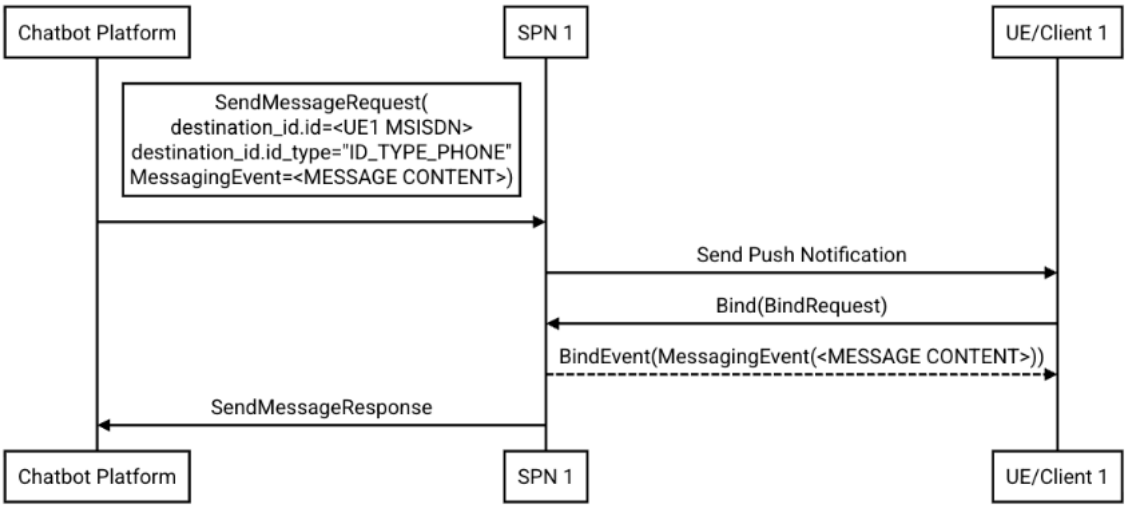

Figure 14: Chatbot to RCS client Message

5.2.2 Prewarm

RCS clients can use prewarm before the message is sent to wake up the recipient RCS client(s). This ensures that the recipient RCS client(s) have set up a bind channel to their SPN when the message is sent. This ensures faster delivery of messages.

In order to prewarm the recipient(s), the RCS client may:

1. Construct the PrewamRequest protocol buffer with the recipient rcs id(s) as per section 3.1.4.
2. Call the Prewarm gRPC defined in section 3.2.3.

Upon receiving the PrewarmRequest, the SPN shall:

3. Check that the auth token is still valid as per section 4.2.
4. If a recipient is over NNI, the SPN shall:
   a. Construct the PrewarmRequest from the incoming gRPC.
   b. Locate the gRPC endpoint for the recipient's SPN as per section 5.4.
   c. Call the recipient's SPN Prewarm gRPC method.
   d. Wait for the PrewarmResponse from the recipient's SPN. If it receives a gRPC error, propagate the error to the RCS client.
5. Construct the PrewarmResponse protocol buffer as per section 3.1.4.
6. Respond to the Prewarm gRPC with the PrewarmResponse.

When receiving a Prewarm gRPC, the recipient's SPN shall:

7. If the recipient is not on RCS, return a NOT_FOUND gRPC error. Otherwise,
8. If bind channel is open to the recipient, send a KeepAlive as per section 4.5.2.
9. Otherwise, send a push notification to the recipient as per section 2.15 of [[GSMA PRD-RCC.07]].
10. Construct a PrewarmResponse protocol buffer as per section 3.1.4.
11. Respond to the Prewarm gRPC with the PrewarmResponse.

If the RCS client receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the SendMessage gRPC method as per section 4.2.1.

If the RCS client receives a NOT_FOUND gRPC error code, it shall compose the message as xMS instead of RCS.

This is illustrated in Figure 15.

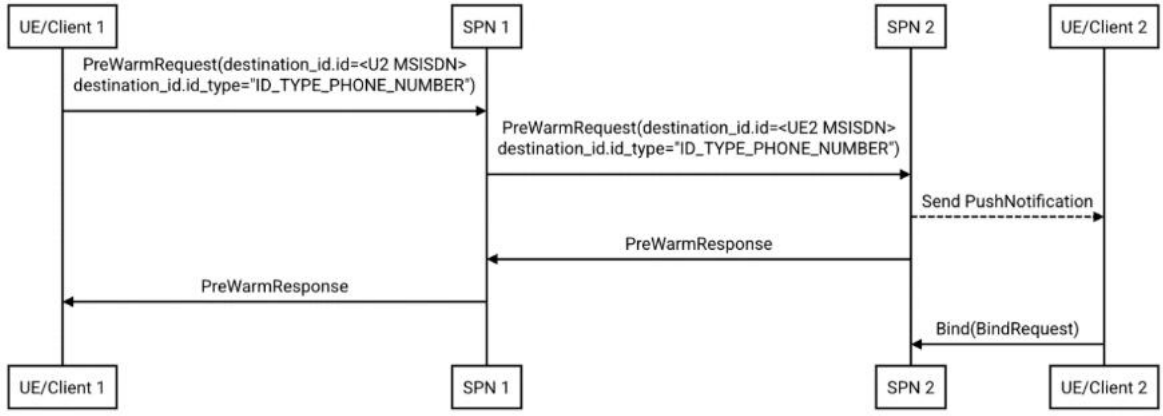

Figure 15: Prewarm

5.2.3 CPIM to Protocol Buffer

For every CPIM message constructed as per [[GSMA PRD-RCC.07]], other than the ones defined the following subsections, the RCS client shall:

1. For each part in a Multipart CPIM, construct a MessagePart as per section 3.1.4.
   a. The content_type shall be set to the content-type of the CPIM part.
   b. The contents shall be set to the CPIM part body.
2. Include the MessageParts in the ChatMessage.
3. Construct the MessagingEvent and include the ChatMessage.
4. Set the message_id to the value of the imdn.message-ID CPIM header.
5. Set the receiver_id to:
   a. For 1-to-1, the recipient as per section 2.1.1
   b. For Group Chat, ConversationID of the Group Chat as per section 2.1.2.
   c. For Chatbots, as per section 2.1.3.
6. Set the ContentDisposition to the values in imdn.Disposition-Notification.
7. Convert all CPIM headers (other than message_id and imdn.Disposition-Notification) to Metadata as per section 3.1.4 and include it in the MessagingEvent.
8. Set the MessagePriority to be `MESSAGE_PRIORITY_HIGH`.
9. Set the DeliveryType to be `DELIVERY_TYPE_GUARANTEED`.

5.2.3.1 IsComposing Notification

For IsComposing notifications, the RCS client shall construct a MessagingEvent with:

1. A TypingNotification with:
   a. The state to `STATE_ACTIVE` if the user is typing, `STATE_IDLE` if the user has stopped typing.
   b. The refresh_interval to be the value of the refresh.
2. The message_id to the value of the imdn.message-ID CPIM header.
3. The receiver_id to:
   a. For 1-to-1, the recipient as per section 2.1.1
   b. For Group Chat, the ConversationID of the Group Chat as per section 2.1.2.
   c. For Chatbots, as per section 2.1.3.
4. The ContentDisposition to `CONTENT_DISPOSITION_NONE`
5. All CPIM headers (other than message_id and imdn.Disposition-Notification) converted to Metadata as per section 3.1.4 and include it in the MessagingEvent.

6. The MessagePriority to be `MESSAGE_PRIORITY_BEST_EFFORT`
7. The DeliveryType to be `DELIVERY_TYPE_BEST_EFFORT`

5.2.3.2 Delivery/Read IMDNs

For IMDNs, the RCS client shall construct a MessagingEvent with:

1. A MessageReceipt with:
   a. The receipt type set to:
      i. RECEIPT_TYPE_POSITIVE_DELIVERY for delivery IMDNs.
      ii. RECEIPT_TYPE_READ for Display IMDNs.
      iii. RECEIPT_TYPE_NEGATIVE_DELIVERY for negative IMDNs.
   b. The ReceiptInfo with:
      i. The message_id with the message id of the original message.
      ii. The event_time set to the original message timestamp of when the message was sent.
      iii. If it is a Negative IMDN, one of the DeliveryFailedMlsClientReason for MLS failures.
2. The message_id to the value of the imdn.message-ID CPIM header.
3. The receiver_id to:
   a. For 1-to-1, the recipient as per section 2.1.1.
   b. For Group Chat, the sender of the original message:
      i. Also set the group_id to the ConversationID of the Group Chat as per section 2.1.2.
   c. For Chatbots, as per section 2.1.3.
4. The ContentDisposition to `CONTENT_DISPOSITION_NONE`.
5. All CPIM headers (other than message_id and imdn.Disposition-Notification) converted to Metadata as per section 3.1.4 and include it in the MessagingEvent.
6. The MessagePriority to be `MESSAGE_PRIORITY_NORMAL`.
7. The DeliveryType to be `DELIVERY_TYPE_GUARANTEED`.

5.2.3.3 File Transfer

For File Transfers, the RCS client shall construct a MessagingEvent with:

1. A FileTransfer with:
   a. A FileTransferInfo for the full-size file as per section 3.1.4.
   b. Optionally, a FileTransferInfo for the thumbnail if it is included as per section 3.1.4.
2. The message_id to the value of the imdn.message-ID CPIM header.
3. The receiver_id to:
   a. For 1-to-1, the recipient as per section 2.1.1.
   b. For Group Chat, the ConversationID of the Group Chat as per section 2.1.2.
   c. For Chatbots, as per section 2.1.3.
4. The ContentDisposition to the values in imdn.Disposition-Notification.
5. All CPIM headers (other than message_id and imdn.Disposition-Notification) converted to Metadata as per section 3.1.4 and include it in the MessagingEvent.
6. The MessagePriority to be `MESSAGE_PRIORITY_HIGH`.
7. The DeliveryType to be `DELIVERY_TYPE_GUARANTEED`.

5.2.4 Revoke

Message revocation happens when the timer for receiving a delivery IMDN expires. Since each message has a separate timer that expires, the revoke gRPC request includes only one message_id.

In order to revoke a message, the RCS client shall:

1. Construct the RevokeRequest protocol buffer with the message_id to be revoked as per 3.1.4.
2. Call the RevokeMessage gRPC defined in section 3.2.3.

Upon receiving the RevokeMessage, the SPN shall:

3. Check that the auth token is still valid as per section 4.2.
4. If a destination_id is over NNI, the SPN shall:
   a. Construct the RevokeMessage from the incoming gRPC request.
   b. Set the request_header.requester_id to the sender as per section 2.1.1.
   c. Locate the gRPC endpoint for the recipient's SPN as per section 5.4.
   d. Call the recipient's SPN RevokeMessage gRPC method. If it receives a gRPC error, propagate the error to the RCS client.
   e. Wait for the RevokeMessageResponse from the recipient's SPN.
5. Construct the RevokeMessageResponse protocol buffer as per section 3.1.4.
6. Respond to the RevokeMessage gRPC with the RevokeMessageResponse.

When receiving a RevokeMessage gRPC, the recipient's SPN shall:

7. If the destination_id is not on RCS, return a NOT_FOUND gRPC error. Otherwise,
8. Attempt to revoke the message from store and forward.
9. Construct a RevokeMessageResponse protocol buffer as per section 3.1.4 with the RevokeResult as:
   a. `REVOKE_STATUS_SUCCEEDED` If the revoke was successful.
   b. `REVOKE_STATUS_DELIVERED` If the message was already delivered to the recipient.
   c. `REVOKE_STATUS_MESSAGE_NOT_FOUND` If the message was not found (e.g., SPN never received the message with that message_id).
   d. `REVOKE_STATUS_PERMISSION_DENIED` If the sender was not allowed to revoke the message (e.g., they were not the sender of the original message).
10. Respond to the RevokeMessage gRPC with the RevokeMessageResponse.

If the RCS client receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the SendMessage gRPC method as per section 4.2.1.

If the RCS client receives a NOT_FOUND gRPC error code, it shall compose the message as xMS instead of RCS.

This flow is illustrated in Figure 16.

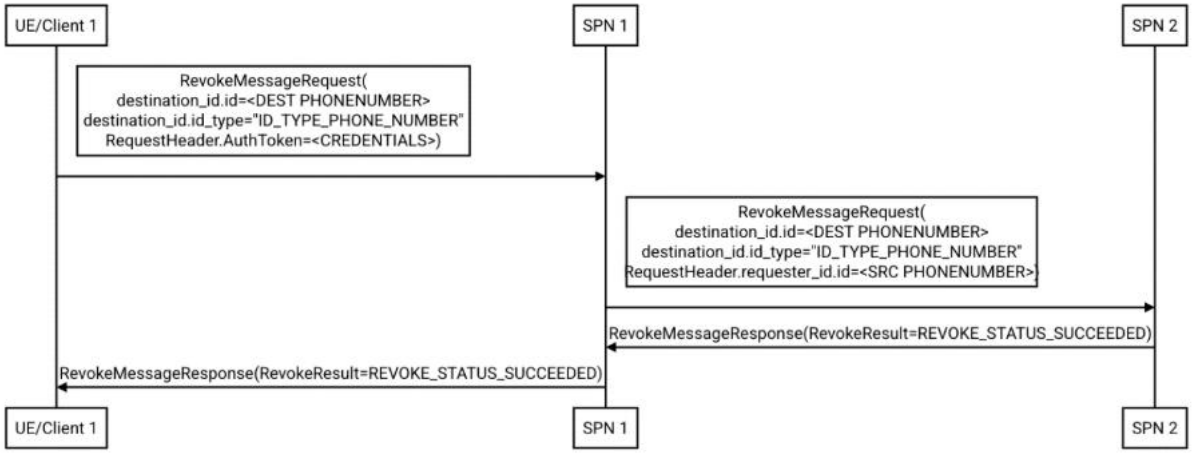

Figure 16: RevokeMessage

5.2.5 E2EE establishment

5.2.5.1 E2EE Messages

To send End-to-End-Encrypted messages, the RCS client shall:

1. Construct a CPIM for the end-to-end encrypted message as per section 7 of [GSMA PRD-RCC.16].
2. Convert the CPIM message to Protocol buffer as per section 5.2.3.

    a) If the Original-Message-ID CPIM header is specified, include it in the MessagingEvent.original_message_id field.

3. Follow procedures in section 5.2.1 to send the Message.

The Participating Function shall follow procedures in section 5.2.1 to propagate the message, and the Conversation Focus shall follow procedures in [GSMA PRD-RCC.16] to validate the message, and section 5.2.1 to propagate and respond to the gRPC.

If the Conversation Focus or any recipient client needs to send a Negative IMDN, they shall follow the process in section 5.2.3.2 to construct the IMDN, and section 5.2.1 to send it.

5.2.5.2 Create MLS Group

5.2.5.2.1 Create MLS Group for Existing Conversation

To create a new Group Chat with E2EE, the RCS client shall follow procedures in 5.3.1.

To create a new MLS Group for 1-to-1 conversation, add encryption to an existing Group Chat, or create a new Era for an encrypted 1-to-1 or Group Chat, the RCS client shall:

1. Construct the CreateMlsGroupRequest as per section 3.1.1 with:

    a) The destination_id being the RcsId of the recipient for a 1-to-1 conversation, or the Conversation-ID for a Group Chat.
    b) The ClientMlsControlMessage as per section 3.1.1 and [GSMA PRD-RCC.16].

2. Call the CreateMlsGroup gRPC method.

Upon receiving the CreateMlsGroup gRPC call, the Participating Function shall:

3. Validate the auth token as per 4.2.
4. Construct a CreateMlsGroupRequest by copying the incoming CreateMlsGroupRequest with:

    a) Set the requester_id to be the sender as per section 2.1.1.

5. Lookup the gRPC endpoint for the Conversation Focus that is stored locally.
6. Call the Conversation Focus CreateMlsGroup gRPC method.
7. Wait for the CreateMlsGroupResponse from the ConversationFocus or if receiving a gRPC error propagate the gRPC error back to the RCS client.
8. Construct a CreateMlsGroupResponse for the sender.
9. Respond to the CreateMlsGroup gRPC with the CreateMlsGroupResponse.

When receiving a CreateMlsGroup call, the Conversation Focus shall:

10. If a conversation is a Group Chat:

    a) Check if the Group Chat exists in local storage, if not, return NOT_FOUND gRPC error code.
    b) Check if the sender is a member of the Group Chat, if not, return PERMISSION_DENIED. Otherwise,

11. Validate the ClientMlsControlMessage and the Commit as per section 6.2.2 of [GSMA PRD-RCC.16].

    a) If the Commit validation fails, construct and send a negative MessageReceipt as per section 3.1.4 with the proper reason as per [GSMA PRD-RCC.16].

12. Otherwise, apply the Commit and generate a ServerMlsControlMessage based on the incoming ClientMlsControlMessage.
13. Construct a MessagingEvent as per section 3.1.4 and send the MessagingEvent as per section 5.2.1.
14. Construct a CreateMlsGroupResponse and respond to the CreateMlsGroup gRPC with the CreateMlsGroupResponse.

When receiving a MessagingEvent with a ServerMlsControlMessage, the RCS client shall:

15. Validate the Commit as part of section 6.2.6 of [GSMA PRD-RCC.16].
16. If the Commit validation fails, construct and send a negative MessageReceipt as per section 3 with the proper reason as per [GSMA PRD-RCC.16].
17. Otherwise, apply the Commit locally as per [GSMA PRD-RCC.16].

If the RCS client receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the SendMessage gRPC method as per section 4.2.1.

If the RCS client receives a NOT_FOUND gRPC error code for a Group Chat, it shall re-create the group as per section 5.3.1.

If the RCS client receives a PERMISSION_DENIED gRPC error code for a Group Chat, it shall disallow the user from sending any further messages to the Group Chat.

5.2.5.3 Apply MLS Control Message

To apply an MLS Control Message as per section 9.5.1 of [GSMA PRD-RCC.16], the RCS client shall:

1. Construct an ApplyMlsControlMessageRequest with:

    a) A ClientMlsControlMessage as per section 3.1.1 and as per [GSMA PRD-RCC.16].
    b) destination_id as the RcsId of the MSISDN of the recipient in a 1-to-1 conversation, or RcsId of the Conversation-ID for a Group Chat.

2. Call ApplyMlsControlMessage gRPC method.

Upon receiving an ApplyMlsControlMessage gRPC, the Participating Function shall:

3. Validate the auth token as per section 4.2.
4. Construct an ApplyMlsControlMessageRequest from the incoming an ApplyMlsControlMessageRequest with the requester_id set to the sender as per section 2.1.1.
5. Lookup the gRPC endpoint for the Conversation Focus that is stored locally.
6. Call the ApplyMlsControlMessage gRPC method on the Conversation Focus.
7. Wait for the ApplyMlsControlMessageResponse from the ConversationFocus. If receiving a gRPC error, propagate the error back to the RCS client.
8. Construct a ApplyMlsControlMessageResponse for the sender.
9. Respond to the ApplyMlsControlMessage gRPC with the ApplyMlsControlMessageResponse.

Upon receiving an ApplyMlsControlMessage gRPC, the Conversation Focus shall:

10. If a conversation is a Group Chat:

    a. Check if the Group Chat exists in local storage, if not, return NOT_FOUND gRPC error code.
    b) Check if the sender is a member of the Group Chat, if not, return PERMISSION_DENIED. Otherwise,

11. Validate the ClientMlsControlMessage and the Commit as per section 6.2.2 of [GSMA PRD-RCC.16].
12. If the Commit validation fails, construct and send a negative MessageReceipt as per section 3.1.4 with the proper reason as per [GSMA PRD-RCC.16].
13. Otherwise, apply the Commit and generate a ServerMlsControlMessage based on the incoming ClientMlsControlMessage.
14. Construct a MessagingEvent as per section 3.1.4 and send the MessagingEvent as per section 5.2.1.
15. Construct an ApplyMlsControlMessageResonse and respond to the ApplyMlsControlMessge gRPC with the ApplyMlsControlMessageResonse.

When receiving a MessagingEvent with a ServerMlsControlMessage, the RCS client shall:

16. Validate the Commit as part of section 6.2.6 of [GSMA PRD-RCC.16].
17. If the Commit validation fails, construct and send a Negative MessageReceipt as per section 3 with the proper reason as per [GSMA PRD-RCC.16].
18. Otherwise, apply the Commit locally as per [GSMA PRD-RCC.16].

If the RCS client receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the SendMessage gRPC method as per section 4.2.1.

If the RCS client receives a NOT_FOUND gRPC error code for a Group Chat, it shall re-create the group as per section 5.3.1.

If the RCS client receives a PERMISSION_DENIED gRPC error code for a Group Chat, it shall disallow the user from sending any further messages to the Group Chat.

5.2.5.4 Get MLS Group Info

To apply get MLS Group Info, the RCS client shall:

1. Construct a GetMlsGroupInfoRequest as per section 3.1.1 with:

    a) destination_id as the recipient as per section 2.1.1 in a 1-to-1 conversation, or the Conversation-ID for a Group Chat as per 2.1.2.
    b) MlsOpaqueToken set to the MlsOpaqueToken received from the SPN.
    c) Optionally, last known good epoch identifier that the client has successfully applied.

2. Call GetMlsGroupInfo gRPC method.

Upon receiving a GetMlsGroupInfo gRPC, the Participating Function shall:

3. If a conversation is a Group Chat:

    a) Check if the Group Chat exists in local storage, if not, return NOT_FOUND gRPC error code.
    b) Check if the sender is a member of the Group Chat, if not, return PERMISSION_DENIED. Otherwise,

4. Validate the auth token as per section 4.2.
5. Construct a GetMlsGroupInfoRequest from the incoming GetMlsGroupInfoRequest with the requester_id set to the RcsId for the sender.
6. Lookup the gRPC endpoint for the Conversation Focus that is stored locally.
7. Call the GetMlsGroupInfo gRPC method on the Conversation Focus.
8. Wait for the GetMlsGroupInfoResponse from the ConversationFocus. If receiving a gRPC error, propagate the error back to the RCS client.
9. Construct a GetMlsGroupInfoResponse for the sender.
10. Respond to the GetMlsGroupInfo gRPC with the GetMlsGroupInfoResponse.

Upon receiving a GetMlsGroupInfo gRPC, the Conversation Focus shall:

11. If a conversation is a Group Chat:

    a) Check if the Group Chat exists in local storage, if not, return NOT_FOUND gRPC error code.
    b) Check if the sender is a member of the Group Chat, if not, return PERMISSION_DENIED. Otherwise,

12. Construct a GetMlsGroupInfoResponse as per section 3.1.1 with:

    a) EpochIdentifier for the latest Era and Epoch of the MLS Group
    b) MlsGroupInfo that is stored
    c) If the RCS client sent up an EpochIdentifier:

        i. A list of EpochControlMessages as per section 3 that contains list of all ServerMlsRcsMessage since the EpochIdentifier and the message_id that is associated with the ServerMlsRcsMessage
        ii. Optionally an EpochIdentifier to indicate if there are more ServerMlsRcsMessages to be retrieved

13. Respond to the GetMlsGroupInfo gRPC with the GetMlsGroupInfoResponse.

When receiving a GetMlsGroupInfoResponse, the RCS client shall:

14. If there are Commits in EpochControlMessages, validate the Commit as part of section 6.2.6 of [GSMA PRD-RCC.16].
    a) If the Commit validation fails, construct and send a negative MessageReceipt as per section 3 with the proper reason as per [GSMA PRD-RCC.16].
    b) Otherwise, apply the Commit locally as per [GSMA PRD-RCC.16].
15. If there are proposals, process the proposals for future Commits.
16. If there is a paginated_epoch_identifier, restart the procedure from the top of this section with the paginated_epoch_identifier as the current_client_epoch_identifier.
17. If at the end of the procedure there are uncommitted Proposals, construct a Commit to apply those Proposals as per [GSMA PRD-RCC.16].
18. If necessary, use the MlsGroupInfo to Self Heal as per section 10 of [GSMA PRD-RCC.16].

If the RCS client receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the SendMessage gRPC method as per section 4.2.1.

If the RCS client receives a NOT_FOUND gRPC error code for a Group Chat, it shall re-create the group as per section 5.3.1.

If the RCS client receives a PERMISSION_DENIED gRPC error code for a Group Chat, it shall disallow the user from sending any further messages to the Group Chat.

5.3 Group Chat

5.3.1 Group Chat Creation

To create a Group Chat, the RCS client shall:

1. Construct the CreateGroupRequest protocol buffer as per section 3.1.5:
    a) Setting the new participant lists in user_ids RcsId as per section 2.1.1.
    b) Setting the new unique group_id RcsId as per section 2.1.2.
    c) Construct the metadata GroupMetadata with:
        i. Optionally, setting the GroupMetadata.icon.content_type and GroupMetadata.icon.icon_uri specified in section 3.1.5.
        ii. Setting the subject GroupMetadata.subject.content_type and GroupMetadata.payload specified in section 3.1.5.
        iii. If the Group Chat is end-to-end encrypted, the RCS client shall encrypt the contents of GroupMetadata.icon and GroupMetadata.subject as per section 9.7.1 of [GSMA PRD-RCC.16].
    d) If the Group Chat is end-to-end encrypted, the RCS client shall construct the ClientMlsControlMessage with:
        i. Setting a unique ClientMlsControlMessage.message_id.
        ii. Construct and include the ClientMlsRcsMessage.mls_rcs_message as per section 7.9.1 of [GSMA PRD-RCC.16].

iii. Setting the epoch_authenticator to the current epoch as per [RFC9420].

2. Call the CreateGroup gRPC method defined in section 3.2.4.

Upon receiving a CreateGroup gRPC, the SPN shall:

3. Check that the auth token is still valid as per section 4.2.
4. Validate that the CreateGroupRequest.group_id is unique.
5. Capture and store the group create_time.
6. For this group_id, store the Group Chat participants and the Group Chat metadata.
7. Construct a GroupInfo protocol buffer specified in section 3.1.5:

   a) Generating and setting a unique GroupInfo.conference_uri.
   b) Copying and setting the validated GroupInfo.group_id.
   c) Copying and setting the GroupChatMetadata received in the CreateGroupRequest.

8. Construct a CreateGroupResponse and respond to the sender's RCS client with the CreateGroupResponse.
9. Lookup participants in local storage, for each participant, construct a CreateGroupNotification as per section 3.1.5:

   a) Setting the CreateGroupNotification.group_info protocol buffer.
   b) Setting the CreateGroupNotification.create_time.

10. Wrap the CreateGroupNotification in a GroupChangeEvent as per section 3.1.5.
11. For each GroupChangeEvent, wrap it into a MessagingEvent protocol buffer as per section 3.1.5.
12. For each MessagingEvent, wrap it into a SendMessageRequest protocol buffer as per section 3.1.5 setting the MessagingEvent.receiver_id RcsId as per section 2.1.1.

    a) Locate the gRPC endpoint for the recipient's SPN as per section 5.4.
    b) Call the SendMessage gRPC method on the participants' SPN.

If the RCS client receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the gRPC method as per section 4.2.1.

This is illustrated in Figure 17.

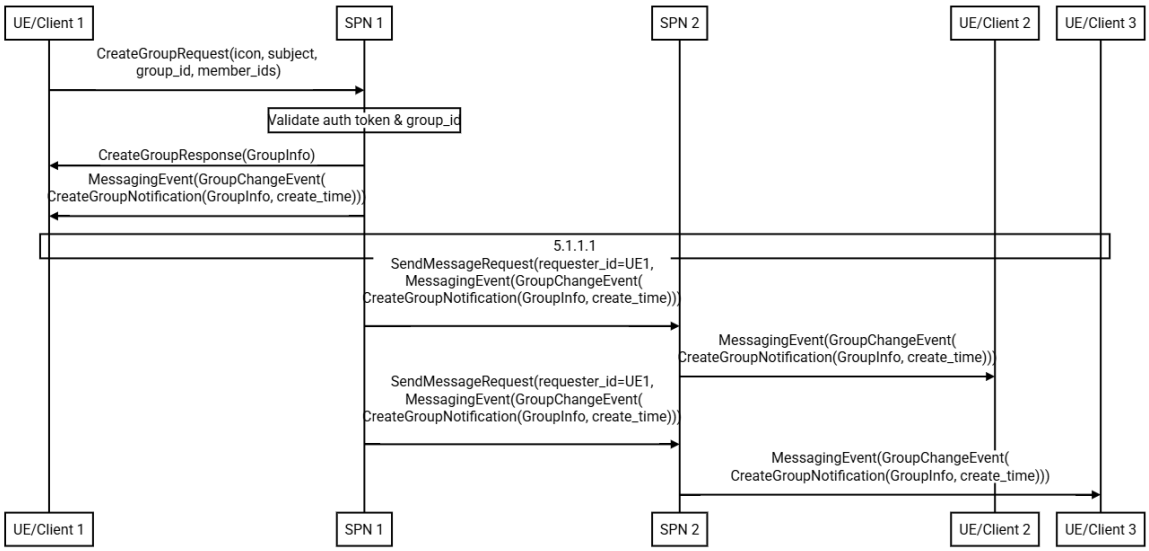

Figure 17: Group Chat creation

5.3.2 Group Chat Metadata update

To change any part of the metadata of an existing Group Chat, the RCS client shall:

1. Construct the ChangeGroupMetadataRequest protocol buffer as per section 3.1.5:
   a) Setting the ChangeGroupMetadataRequest.group_id RcsID of the group being updated as per section 2.1.2.
   b) Construct the ChangeGroupMetadataRequest.metadata_update with:
      i. Optionally, setting the GroupMetadata.icon.content_type and GroupMetadata.icon.icon_uri specified in section 3.1.5.
      ii. Setting the subject GroupMetadata.subject.content_type and GroupMetadata.payload specified in section 3.1.5.
      iii. If the Group Chat is end-to-end encrypted, the RCS client shall encrypt the contents of GroupMetadata.icon and GroupMetadata.subject as per section 9.7.1 of [GSMA PRD-RCC.16].
   c) If the Group Chat is end-to-end encrypted, the RCS client shall construct the ClientMlsControlMessage with:
      i. Setting a unique ClientMlsControlMessage.message_id.
      ii. Construct and include the ClientMlsRcsMessage.mls_rcs_message as per section 7.9.1 of [GSMA PRD-RCC.16].
      iii. Setting the epoch_authenticator to the current epoch as per [RFC9420].
2. Call the ChangeGroupMetadata gRPC method defined in section 3.2.4

Upon receiving a ChangeGroupMetadata gRPC, the Participating Function shall:

3. Check that the auth token is still valid as per section 4.2.
4. Locate the gRPC endpoint of the Controlling Function for the Group Chat as per section 5.4.

5. Construct the ChangeGroupMetadataRequest setting the RequestHeader.requester_id RcsId as per section 2.1.1.
6. Call the Controlling Function's ChangeGroupMetadata gRPC method with the ChangeGroupMetadataRequest.
7. Wait for the ChangeGroupMetadataResponse from the Controlling Function.

    a) If a gRPC error is received from the Controlling Function, propagate the gRPC error to the RCS client.

8. Construct the ChangeGroupMetadataResponse.
9. Respond to the sender's RCS client with the ChangeGroupMetadataResponse.

When receiving a ChangeGroupMetadata gRPC, the Controlling Function shall:

10. Check if the Group Chat exists in local storage, if not, return NOT_FOUND gRPC error code.
11. Check if the sender is a participant of the Group Chat, if not, return PERMISSION_DENIED. Otherwise,
12. For this group_id, update and store the Group Chat participants and the Group Chat metadata.
13. Construct the ChangeGroupMetaResponse and respond to the gRPC method.
14. Lookup participants in local storage.
15. For each participant, construct a ChangeGroupMetadataNotification as per section 3.1.5 setting the updated ChangeGroupMetadataNotification.metadata.
16. Wrap the ChangeGroupMetadataNotification in a GroupChangeEvent as per section 3.1.5.

    a) For each GroupChangeEvent, wrap it into a MessagingEvent protocol buffer as per section 3.1.5.
    b) For each MessagingEvent, wrap it into a SendMessageRequest protocol buffer as per section 3.1.5setting the MessagingEvent.receiver_id RcsId as per section 2.1.1.
    c) Locate the gRPC endpoint for the recipient's SPN as per section 5.4.
    d) Call the SendMessage gRPC method on the participants' SPN.

If the RCS client or participating function receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the gRPC method as per section 4.2.1.

If the RCS client receives a NOT_FOUND gRPC error code, it shall re-create the group as per section 5.3.1.

If the RCS client receives a PERMISSION_DENIED gRPC error code, it shall disallow the user from sending any further messages to the Group Chat.

This is illustrated in Figure 18.

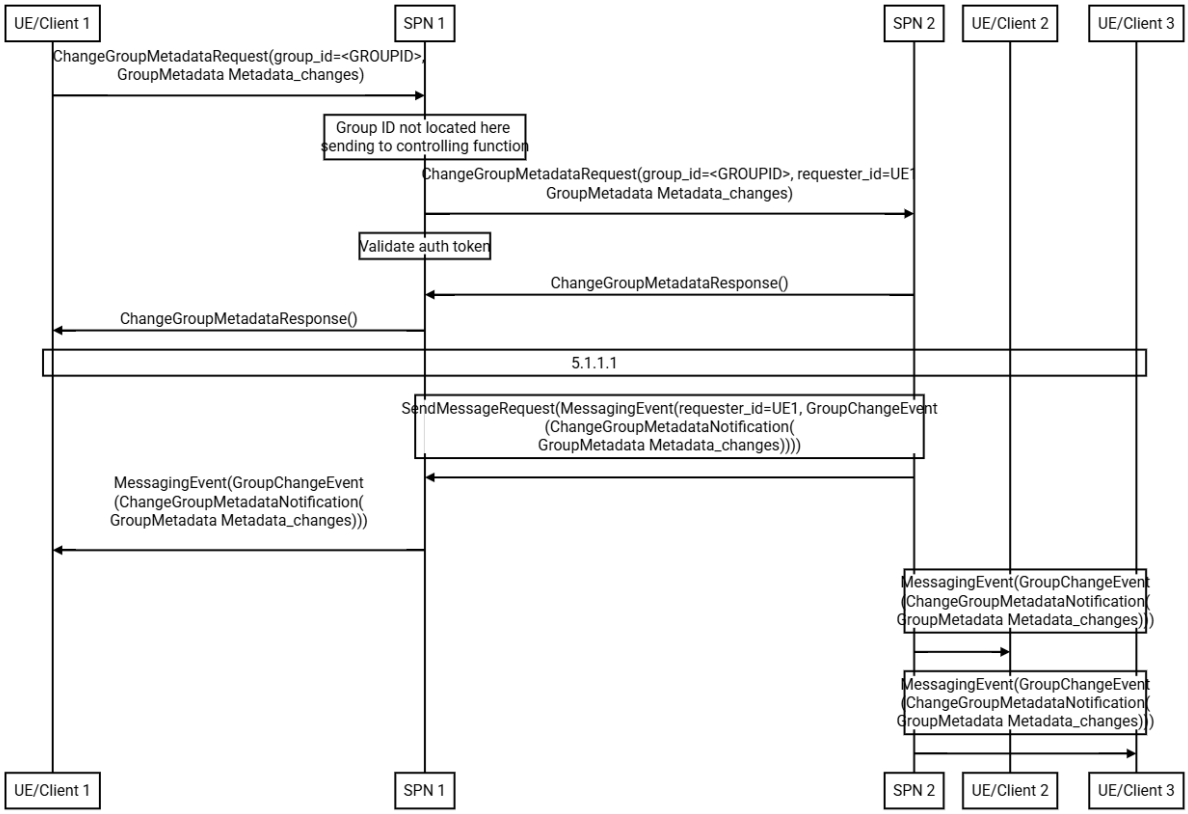

Figure 18: Group Chat metadata updated where the Controlling Function is not local

5.3.3 Adding Participants

To add new participants to an existing Group Chat, the RCS client shall:

1. Construct the AddGroupUsersRequest protocol buffer as per section 3.1.5:
    a) Setting the AddGroupUsersRequest.group_id RcsId of the group being updated as per section 2.1.2.
    b) For each participant being added, setting the AddGroupUsersRequest.user_ids RcsId as per section 2.1.1.
    c) If the Group Chat is end-to-end encrypted, the RCS client shall construct the ClientMlsControlMessage with:
        i. Setting a unique ClientMlsControlMessage.message_id.
        ii. Construct and include the ClientMlsRcsMessage.mls_rcs_message as per section 7.9.1 of [GSMA PRD-RCC.16].
        iii. Setting the epoch_authenticator to the current epoch as per [RFC9420].
2. Call the AddGroupUsers gRPC method defined in section 3.2.4.

Upon receiving an AddGroupUsers gRPC, the Participating Function shall:

3. Check that the auth token is still valid as per section 4.2.
4. Locate the gRPC endpoint of the Controlling Function for the Group Chat as per section 5.4.

5. Construct the AddGroupUsersRequest setting the RequestHeader.requester_id RcsId as per section 2.1.1.
6. Call the Controlling Function's AddGroupUsers gRPC with the AddGroupUsersRequest
7. Wait for the AddGroupUsersResponse from the Controlling Function

    a) If a gRPC error is received from the Controlling Function, propagate the gRPC error to the RCS client.

8. Construct the AddGroupUsersResponse.
9. Respond to the sender's RCS client with the AddGroupUsersResponse.

When receiving an AddGroupUsers gRPC, the Controlling Function shall:

10. Check if the Group Chat exists in local storage, if not, return NOT_FOUND gRPC error code.
11. Check if the sender is a participant of the Group Chat, if not, return PERMISSION_DENIED. Otherwise,
12. For this group_id, update and store the Group Chat participants and the Group Chat metadata.
13. Construct an AddGroupUsersResponse and set the AddGroupUsersResponse.added_users list and respond to the gRPC method.
14. Lookup participants in local storage.
15. For each new Group Chat participant, construct a CreateGroupNotification protocol buffer as per section 3.1.5 and set the updated CreateGroupNotification.group_info and CreateGroupNotification.create_time.
16. Wrap the CreateGroupNotification in a GroupChangeEvent as per section 3.1.5.
17. For each existing Group chat participant, construct an AddUsersToGroupNotification protocol buffer as per section 3.1.5 with setting the AddUsersToGroupNotification .added_users Rcs ID as per section 2.1.2.

    a) Wrap the AddUsersToGroupNotification in a GroupChangeEvent as per section 3.1.5.
    b) For each GroupChangeEvent, wrap it into a MessagingEvent protocol buffer as per section 3.1.5.
    c) For each MessagingEvent, wrap it into a SendMessageRequest protocol buffer as per section 3.1.5 setting the MessagingEvent.receiver_id RcsId as per section 2.1.1.
    d) Locate the gRPC endpoint for the recipient's SPN as per section 5.4.

Call the SendMessage gRPC method on the participant's SPN.

If the RCS client or participating function receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the gRPC method as per section 4.2.1.

If the RCS client receives a NOT_FOUND gRPC error code, it shall re-create the group as per section 5.3.1.

If the RCS client receives a PERMISSION_DENIED gRPC error code, it shall disallow the user from sending any further messages to the Group Chat.

This is illustrated in Figure 19.

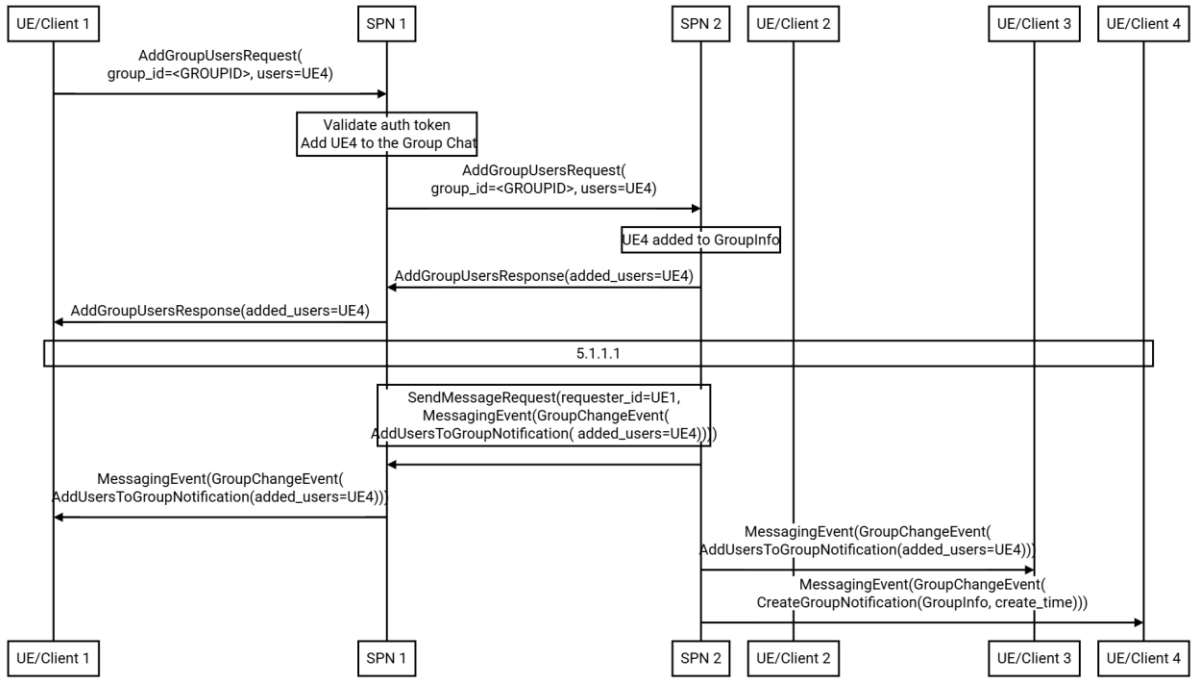

Figure 19: Adding a participant on a remote SPN

5.3.4 Removing Participants

To remove participants to an existing Group Chat, the RCS client shall:

1. Construct the RemoveGroupUsersRequest protocol buffer as per section 3.1.5:
    a) Set the RemoveGroupUsersRequest.group_id Rcs Id of the group being updated as per section 2.1.2.
    b) For all participants being removed, construct and set a list of RemoveGroupUsersRequest.removed_users, by:
        i. Setting RemoveGroupUserMetadata.user RcsId as per section 2.1.1.
        ii. Setting RemoveGroupUserMetadata.reason RemoveGroupUserReason as per section 3.1.5.
    c) If the Group Chat is end-to-end encrypted, the RCS client shall construct the ClientMlsControlMessage with:
        i. Setting a unique ClientMlsControlMessage.message_id.
        ii. Construct and include the ClientMlsRcsMessage.mls_rcs_message as per section 7.9.1 of [GSMA PRD-RCC.16].
        iii. Setting the epoch_authenticator to the current epoch as per [RFC9420].
2. Call the RemoveGroupUsers gRPC method defined in section 3.2.4.

Upon receiving a RemoveGroupUsers gRPC, the Participating Function shall:

3. Check that the auth token is still valid as per section 4.2.

4. Locate the gRPC endpoint of the Controlling Function for the Group Chat as per section 5.4.
5. Construct the RemoveGroupUsersRequest from the incoming gRPC with setting the RequestHeader.requester_id RcsId as per section 2.1.1.
6. Call the Controlling Function's RemoveGroupUsers gRPC with the RemoveGroupUsersRequest
7. Wait for the RemoveGroupUsersResponse from the Controlling Function.

    a) If a gRPC error is received from the Controlling Function, propagate the gRPC error to the RCS client.

8. Construct the RemoveGroupUsersResponse.
9. Respond to the sender's RCS client with the RemoveGroupUsersResponse.

When receiving a RemoveGroupUsers gRPC, the Controlling Function shall:

10. Check if the Group Chat exists in local storage, if not, return NOT_FOUND gRPC error code.
11. Check if the sender is a participant of the Group Chat, if not, return PERMISSION_DENIED. Otherwise,
12. For this group_id, update and store the Group Chat participants and the Group Chat metadata.
13. Construct a RemoveGroupUsersResponse and set the RemoveGroupUsersResponse.removed_user_ids RcsId as per section 2.1.1 and respond to the gRPC method.
14. For this group_id, update and store the Group Chat participants and the Group Chat metadata.
15. For each Group Chat participant, construct a RemoveUsersFromGroupNotification protocol buffer as per section 3.1.5 and set the RemoveUsersFromGroupNotification.removed_users RcsId as per section 3.1.5.
16. Wrap the RemoveUsersFromGroupNotification in a GroupChangeEvent as per section 3.1.5.

    a) For each GroupChangeEvent, wrap it into a MessagingEvent protocol buffer as per section 3.1.5.
    b) For each MessagingEvent, wrap it into a SendMessageRequest protocol buffer as per section 3.1.5 setting the MessagingEvent.receiver_id RcsId as per section 2.1.1.
    c) Locate the gRPC endpoint for the recipient's SPN as per section 5.4.
    d) Call the SendMessage gRPC method on the participant's SPN.

If the RCS client or participating function receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the gRPC method as per section 4.2.1.

If the RCS client receives a NOT_FOUND gRPC error code, it shall re-create the group as per section 5.3.1.

If the RCS client receives a PERMISSION_DENIED gRPC error code, it shall disallow the user from sending any further messages to the Group Chat.

This is illustrated in Figure 20.

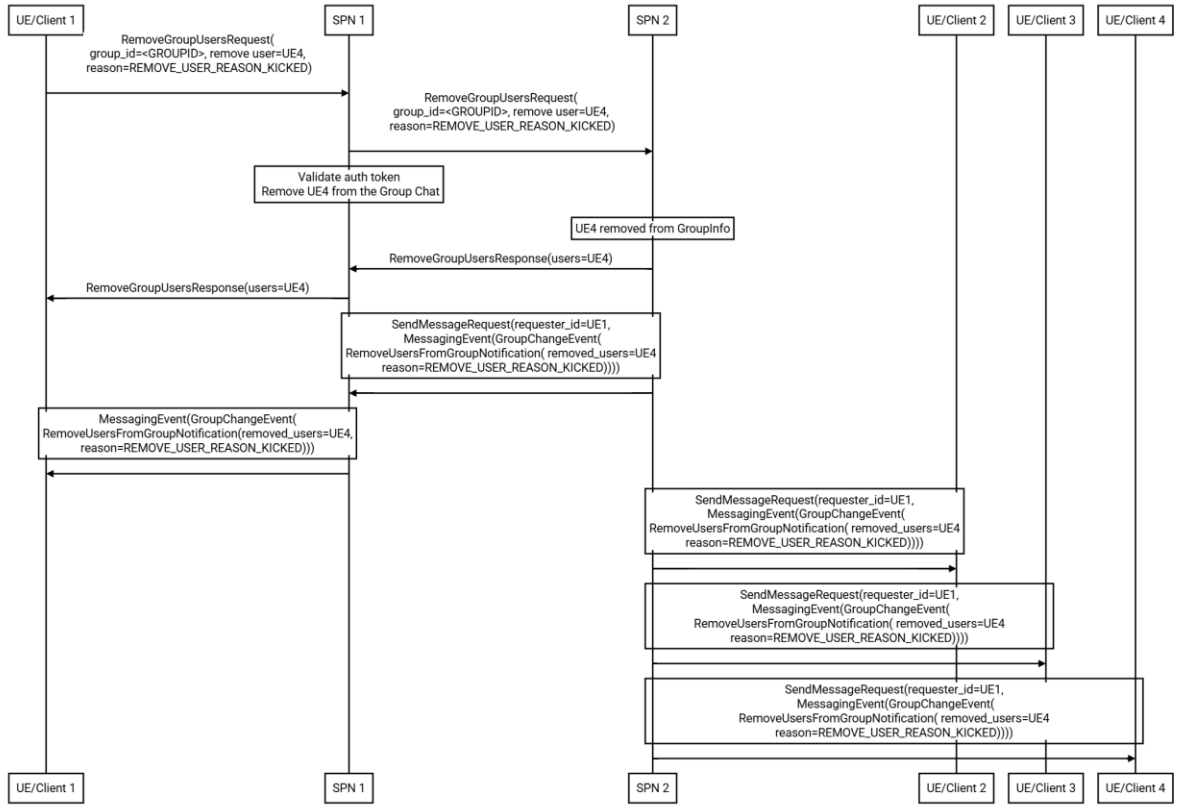

Figure 20: Removing a participant on a remote SPN

5.3.5 Leaving a Group Chat

Follow procedure in section 5.3.4 where the sender is the participant being removed.

5.3.6 Fetch all user's Groups Chats

To collect all existing Group Chat ids the user is currently a participant in, the RCS client shall:

1. Construct the GetGroupIdsRequest protocol buffer as per section 3.1.5.
2. Call the GetGroupIds gRPC method defined in section 3.2.4.

When receiving a GetGroupIds gRPC, the SPN shall:

3. Check that the auth token is still valid as per section 4.2.
4. Check if the sender is a participant of any local or locally stored remote Group Chat IDs.
5. If there is no local storage of the remote Group Chats:

    a) Locate the gRPC endpoint of the Controlling Function for the Group Chat as per section 5.4.
    b) Construct the GetGroupIdsRequest from the incoming gRPC with setting the RequestHeader.requester_id RcsId as per section 2.1.1.
    c) Call all the Controlling Functions GetGroupIds gRPC with the GetGroupIdsRequest.

d) Wait for the GetGroupIdsResponse(s) from all the Controlling Functions.

i. If a gRPC error is received from the Controlling Function, propagate the gRPC error to the RCS client.

6. Merge any of the local group_id(s) with the remote group_id(s) with all the GetGroupIdsResponse.group_ids received.
7. Construct the GetGroupIdsResponse and set the merged GetGroupIdsResponse.group_ids.
8. Respond to the sender's RCS client with the GetGroupIds.

When receiving a GetGroupIds gRPC, the Controlling Function shall:

9. Check if the sender is a member of the Group Chat,
10. Construct a GetGroupIdsResponse and set the GetGroupIdsResponse.group_ids list and respond to the gRPC method.

If the RCS client or SPN receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the gRPC method as per section 4.2.1.

This is illustrated in Figure 21.

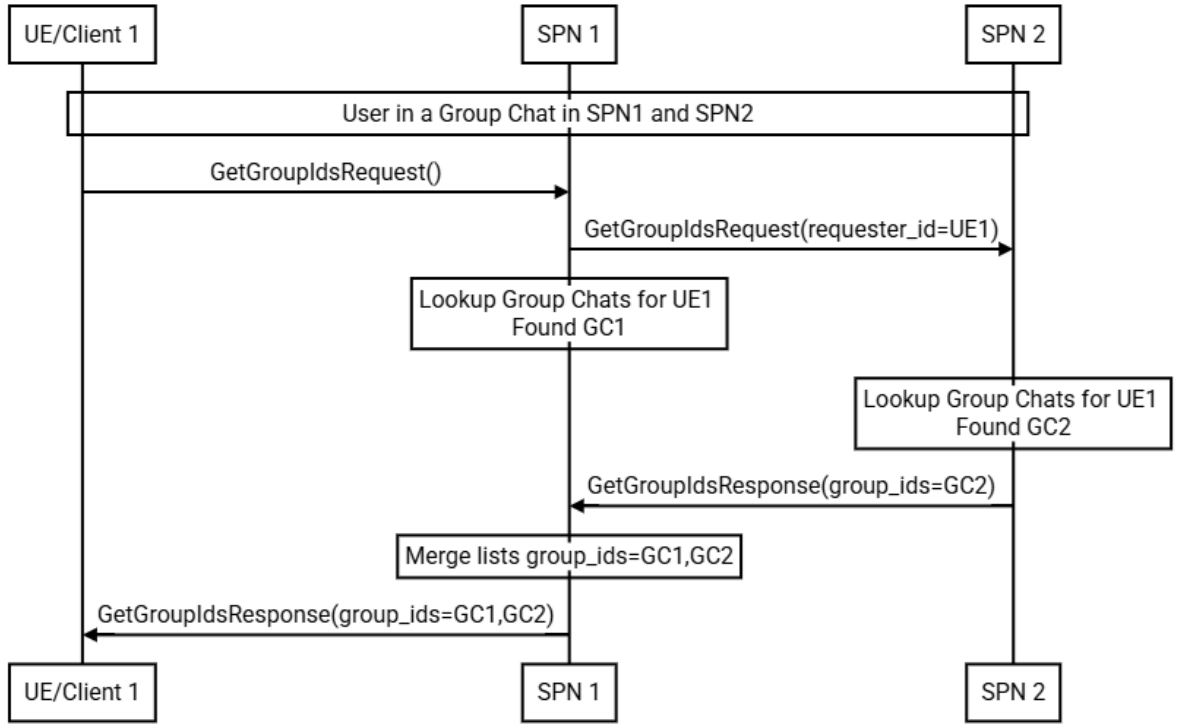

Figure 21: Requesting Group Ids from multiple SPNs

5.3.7 Fetch Group Chat Metadata

To fetch the Group Chat metadata of an existing Group Chat of which the user is a participant, the RCS client shall:

1. Construct the GetGroupInfosRequest protocol buffer as per section 3.1.5:

   a) Set the GetGroupInfosRequest.group_ids Rcs Id of the group being updated as per section 2.1.2.

2. Call the GetGroupInfos gRPC method defined in section 3.2.4.

Upon receiving a GetGroupInfos gRPC, the Participating Function shall:

3. Check that the auth token is still valid as per section 4.2.
4. Locate the gRPC endpoint of the Controlling Function for the Group Chat as per section 5.4.
5. Construct the GetGroupInfosRequest from the incoming gRPC with setting the RequestHeader.requester_id RcsId as per section 2.1.1.
6. Call all the Controlling Functions GetGroupInfos gRPC with the GetGroupInfosRequest.
7. Wait for the GetGroupInfosResponse(s) from all the Controlling Functions.

   a) If a gRPC error is received from the Controlling Function, propagate the gRPC error to the RCS client.

8. Check that the user is a participant in the local Group Chat(s) being looked up, if yes add the associated GroupInfo(s) to the group_infos list.
9. Merge the local GroupInfo group_infos with all the group_infos from all the GetGroupInfosResponse(s) received.
10. Construct the GetGroupInfosResponse and set the merged group_info.
11. Respond to the sender's RCS client with the GetGroupInfosResponse.

When receiving a GetGroupInfos gRPC, the Controlling Function shall:

12. Check if the Group Chat exists in local storage, if not, return NOT_FOUND gRPC error code.
13. Check if the sender is a participant of the Group Chat, if not, return PERMISSION_DENIED. Otherwise,
14. If the user is a participant in the local Group Chat(s) being looked up, if yes add the associated GroupInfo(s) to the GroupInfo.group_infos list.
15. Construct a GetGroupInfosResponse and set the GetGroupInfosResponse.group_infos list and respond to the gRPC method.

If the RCS client or participating function receives a DEADLINE_EXCEEDED or INTERNAL gRPC error code, it shall retry the gRPC method as per section 4.2.1.

If the RCS client receives a NOT_FOUND gRPC error code, it shall re-create the group as per section 5.3.1.

If the RCS client receives a PERMISSION_DENIED gRPC error code, it shall disallow the user from sending any further messages to the Group Chat.

This is illustrated in Figure 22.

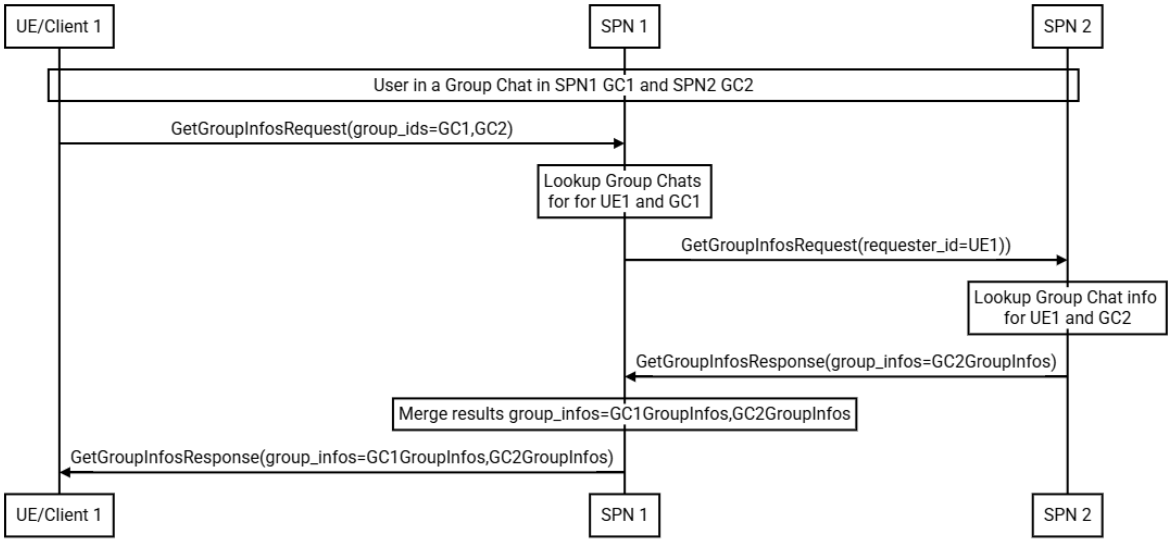

Figure 22: Requesting GroupInfo from multiple SPNs

5.4 Routing over gNNI

To locate the gRPC endpoint of a MSISDN across the gNNI, the SPN shall:

1. Follow section 4.5.2.1 and 4.5.2.2 of [GSMA PRD-IR.67] to do an ENUM query and retrieve the NAPTR records.
2. Replace the _sips service prefix in the NAPTR record with _grpc. The protocol prefix shall remain the same as in the NAPTR record.
3. Follow section 4.5.2.3 and 4.5.2.4 of [GSMA PRD-IR.67] with the modified SRV query from step 2 above.

5.4.1 Adding SRV Records for gNNI Endpoints

In order for gNNI endpoints to be discoverable, the SPN shall add one or more SRV record with the service prefix _grpc. For example:

```
_grpc._tcp.example.com. SRV 0 1 443 grpc1.example.com.
_grpc._tcp.example.com. SRV 0 2 443 grpc2.example.com.
_grpc._udp.example.com. SRV 0 1 443 grpc1.example.com.
_grpc._udp.example.com. SRV 0 2 443 grpc2.example.com.
_sips._tcp.example.com. SRV 0 1 5061 sipserv3.example.com.
_sips._tcp.example.com. SRV 0 2 5061 sipserv4.example.com.
```

SPN may choose the protocol that they support (TCP, UDP or both). However, all protocols must be secure: TLS for TCP and QUIC for UDP.

6 Interworking SIP+MSRP and gRPC

The RCS gRPC IWF is placed between the Messaging Server and the endpoints that implement a different transport protocol. To realise this interworking requirement, the RCS gRPC IWF implements the following mandatory functions:

1. Translation of the RCS gRPC protocol to SIP/MSRP protocol.
2. Translation of the SIP/MSRP protocol to RCS gRPC protocol.

3. Storage of the RCS capabilities on the local capabilities cache.
4. Storage of Group Chat Session Identities [GSMA PRD-RCC.07] (conference URIs) hosted by other interconnected SPNs.

This is illustrated in Figure 23 and Figure 24.

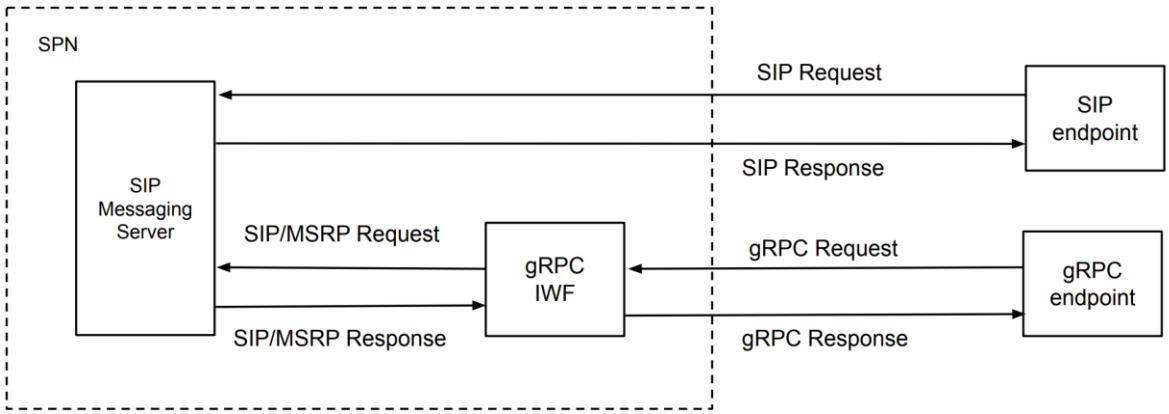

Figure 23: RCS gRPC IWF for SIP-based Service Provider Network

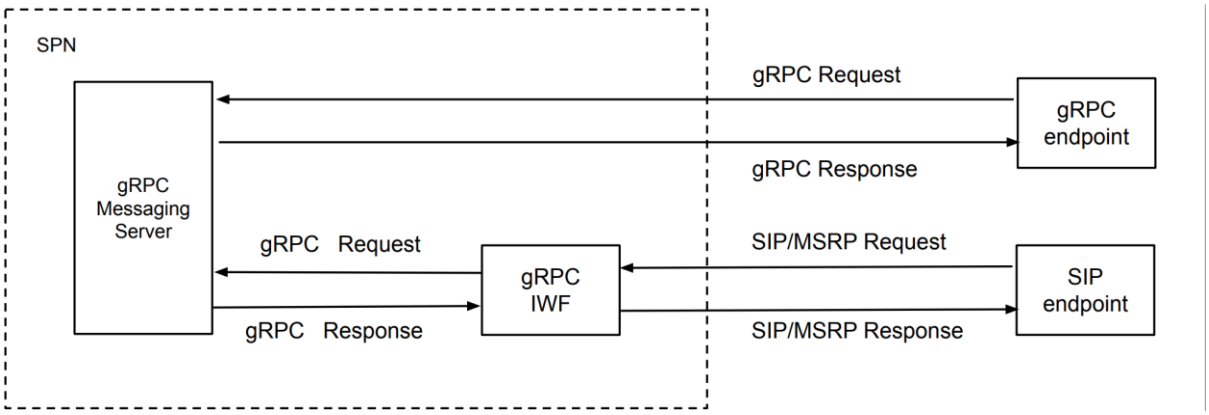

Figure 24: RCS gRPC IWF for gRPC-based Service Provider Network

6.1 SIP Headers and gRPC

6.1.1 gRPC to SIP

When constructing a SIP request, the RCS gRPC IWF may copy some or all SIP headers constructed from RequestHeader.sip_headers into SIP headers.

6.1.2 SIP to gRPC

When constructing a gRPC request, the RCS gRPC IWF may copy some or all SIP headers from SIP into RequestHeader.sip_headers.

6.2 SIP/MSRP - gRPC Error Handling

6.2.1 SIP to gRPC

When the RCS gRPC IWF sends a SIP request to an NNI/UNI, the RCS gRPC IWF may encounter SIP errors as per [RFC3261].

Upon receiving a SIP error from the NNI or UNI the RCS gRPC IWF shall:

1. Construct a ErrorDetails protocol buffer as per section 3.1.1:
   a. Map the SIP response code to the gPRC status code as per Table 3.
      i. If there is a Warning header:
         1. Set ErrorDetails.error_code based on the Warning header code defined in [RFC3261].
         2. Set the ErrorDetails.reason with the Warning header reason defined in [RFC3261].
2. Include the ErrorDetails in the error response as per the error handling section of [gRPC].

SIP response code	gRPC status code
2xx	OK
400	INVALID_ARGUMENT
403	PERMISSION_DENIED
404	NOT_FOUND
408	DEADLINE_EXCEEDED
409	INVALID_ARGUMENT
415	INVALID_ARGUMENT
429	RESOURCE_EXHAUSTED
480	UNAVAILABLE
487	CANCELLED
491	ALREADY_EXISTS
500	INTERNAL
501	UNIMPLEMENTED
503	INTERNAL
603	CANCELLED
other	INTERNAL

Table 3: SIP to gRPC error code mapping

6.2.2 gRPC to SIP

When receiving a gRPC error the RCS gRPC IWF shall:

1. Construct a SIP error as per [RFC3261]:
   a. Map the SIP response code as per Table 4.
      i. If there is an ErrorDetails protocol buffer, construct a Warning header:
         1. Set the code from ErrorDetails.error_code.
         2. Set the Warning reason from the ErrorDetails.reason.

gRPC status code	SIP response code
OK	200
CANCELLED	487
UNKNOWN	500
INVALID_ARGUMENT	400

gRPC status code	SIP response code
DEADLINE_EXCEEDED	408
NOT_FOUND	404
ALREADY_EXISTS	491
PERMISSION_DENIED	403
RESOURCE_EXHAUSTED	429
FAILED_PRECONDITION	400
ABORTED	500
OUT_OF_RANGE	400
UNIMPLEMENTED	501
INTERNAL	500
UNAVAILABLE	480
DATA_LOSS	500
UNAUTHENTICATED	401

Table 4: gRPC to SIP error code mapping

6.2.3 MSRP to gRPC

Upon receiving an MSRP error following an MSRP message to the NNI or UNI the RCS gRPC IWF shall:

Map the MSRP response code to the gPRC status code as per Table 5.

MSRP response code	gRPC status code
200	OK
400	INVALID_ARGUMENT
403	PERMISSION_DENIED
408	DEADLINE_EXCEEDED
413	INTERNAL
415	INTERNAL
423	OUT_OF_RANGE
481	INTERNAL
500	INTERNAL
501	UNIMPLEMENTED
506	ALREADY_EXISTS

Table 5: MSRP status to gRPC status mapping

6.2.4 gRPC to MSRP

When receiving a gRPC error the RCS gRPC IWF shall:

1. Construct a MSRP error as per [GSMA PRD-RCC.11]:
   a. Map the MSRP response code as per Table 6.

gRPC status code	MSRP response code
OK	200
CANCELLED	Not exposed
UNKNOWN	500
INVALID_ARGUMENT	400
DEADLINE_EXCEEDED	408
NOT_FOUND	400
ALREADY_EXISTS	506
PERMISSION_DENIED	403
RESOURCE_EXHAUSTED	500
FAILED_PRECONDITION	400
ABORTED	500
OUT_OF_RANGE	423
UNIMPLEMENTED	501
INTERNAL	500
UNAVAILABLE	500
DATA_LOSS	500
UNAUTHENTICATED	500

Table 6: gRPC status to MSRP status mapping

6.3 Capability Discovery

6.3.1 Status mapping

6.3.1.1 Capability Discovery: SIP to gRPC

SIP Response code	LookupResult
200	CAPABILITIES_RESULT_OK
404	CAPABILITIES_RESULT_NOT_FOUND
480	CAPABILITIES_RESULT_TEMPORARILY_UNAVAILABLE
other	CAPABILITIES_RESULT_UNKNOWN

Table 7: SIP OPTIONS response code to gRPC LookupResult

6.3.1.2 Capability Discovery: gRPC to SIP

LookupResult	SIP Response code
CAPABILITIES_RESULT_OK	200
CAPABILITIES_RESULT_NOT_FOUND	404
CAPABILITIES_RESULT_TEMPORARILY_UNAVAILABLE	480
CAPABILITIES_RESULT_UNKNOWN	503

Table 8: gRPC LookupResult to SIP OPTIONS response code

6.3.2 Capability Discovery: gRPC to SIP

When receiving a GetCapabilities gRPC request, the RCS gRPC IWF shall:

1. Initiate a separate Capability Discovery for each user in GetCapabilitiesRequest .user_ids as per [GSMA PRD-RCC.07].
    a. Construct and send SIP OPTIONS:
        i. The From header shall be constructed from:
            1. gNNI: the RequestHeader.requester_id.
            i. gUNI: the identity obtained from RequestHeader. auth_token.
        iii. The To header shall be constructed from GetCapabilitiesRequest.user_ids.
        iv. The Request URI header shall be constructed from GetCapabilitiesRequest.user_ids.
        v. Copy relevant SIP headers from SendMessageRequest.request_header.sip_headers as defined in section 3.1.1.
        vi. The Accept-Contact header shall be constructed from the sender's capabilities stored in the local capabilities cache.
2. Construct and return the GetCapabilitiesResponse:
    a. For each SIP OPTIONS response, add a UserCapabilitiesResult to the GetCapabilitiesResponse . user_capabilities_results. For each UserCapabilitiesResult:
        i. Copy the corresponding GetCapabilitiesRequest .user_ids into UserCapabilitiesResult. user_capabilities.rcs_id.
        ii. Copy the capabilities from the SIP Contact header into UserCapabilitiesResult. user_capabilities.capabilities.
        iii. Set the UserCapabilitiesResult.lookup_result based on the SIP OPTIONS response code as per Table 7.

This is illustrated in Figure 25.

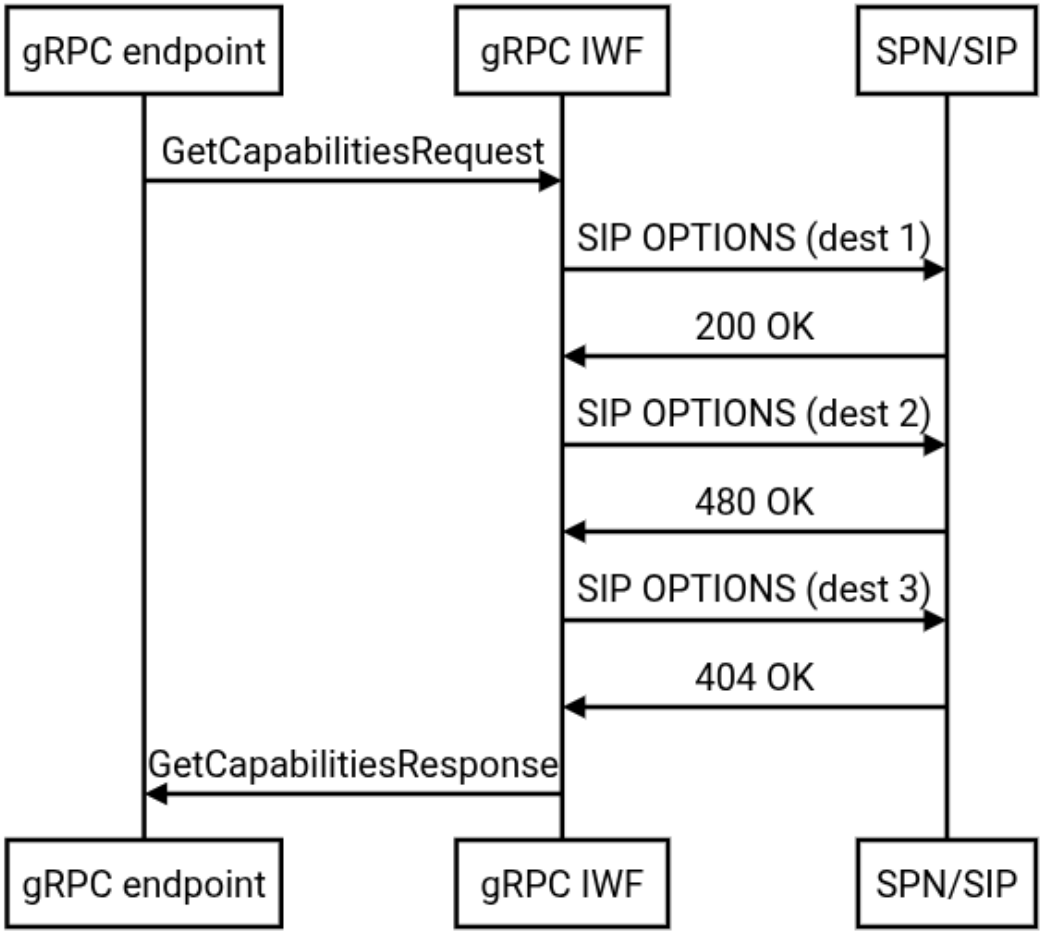

Figure 25: gRPC to SIP Capability Discovery

6.3.3 Capability Discovery: SIP to gRPC

When receiving a SIP OPTIONS, the RCS gRPC IWF shall store in the local capabilities cache the sender's capabilities from the Accept-Contact header in the local capabilities cache and send GetCapabilities gRPC request:

1. A single GetCapabilitiesRequest .user_ids shall be created from the SIP Request URI.
2. The RequestHeader.requester_id shall be constructed from the SIP From header.
3. Copy the relevant SIP OPTIONS headers to SendMessageRequest.request_header.sip_headers as defined in section 3.1.3.
4. Store Contact header capabilities in the local capabilities cache.
5. The SIP OPTIONS response shall be constructed and sent:

    a. The capabilities in the Contact header shall be constructed from UserCapabilitiesResult. user_capabilities.capabilities.
    b. The SIP response code shall be determined from the gRPC response as per Table 6.
    c. Copy the relevant SIP headers from SendMessageRequest.request_header.sip_headers as defined in section 3.1.3.

This is illustrated in Figure 26.

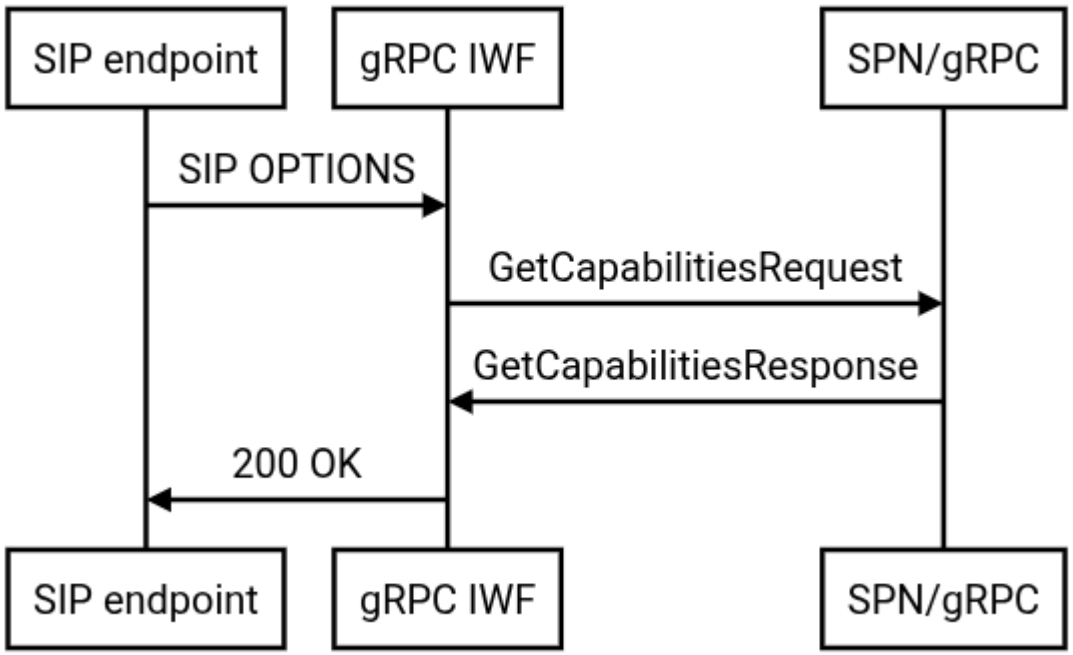

Figure 26: SIP to gRPC Capability Discovery

6.4 Message sending

6.4.1 Message sending: gRPC to SIP

6.4.1.1 1-to-1 Conversation

When receiving an RCS 1-to-1 gRPC message, the RCS gRPC IWF shall:

1. If a SIP session is not active between the RCS gRPC IWF and Service Provider Network, initiate a SIP session as per section 3.2.3 of [GSMA PRD-RCC.07] and connect over MSRP:

    a. The From header shall be constructed from the MessagingEvent.sender_id.
    a) The Request-URI and To headers shall be constructed from the MessagingEvent.receiver_id.
    b. The Request URI shall be a tel URI constructed from the MessagingEvent receiver_id.
    c. Copy relevant SIP headers from SendMessageRequest.request_header.sip_headers as defined in section 3.1.1.

2. Convert MessagingEvent to a CPIM message as per section 6.5.2.
3. Construct a MSRP message with CPIM payload as per [GSMA PRD-RCC.07].
4. Send the MSRP message as per [GSMA PRD-RCC.07].
5. Wait for the MSRP response:

    a. If the gRPC returns an OK result, send 200OK as per [GSMA PRD-RCC.07].

b. If the gRPC returns an error result send mapped error status as per section 6.2.4 and as per the error handling section of [gRPC].

This is illustrated in Figure 27.

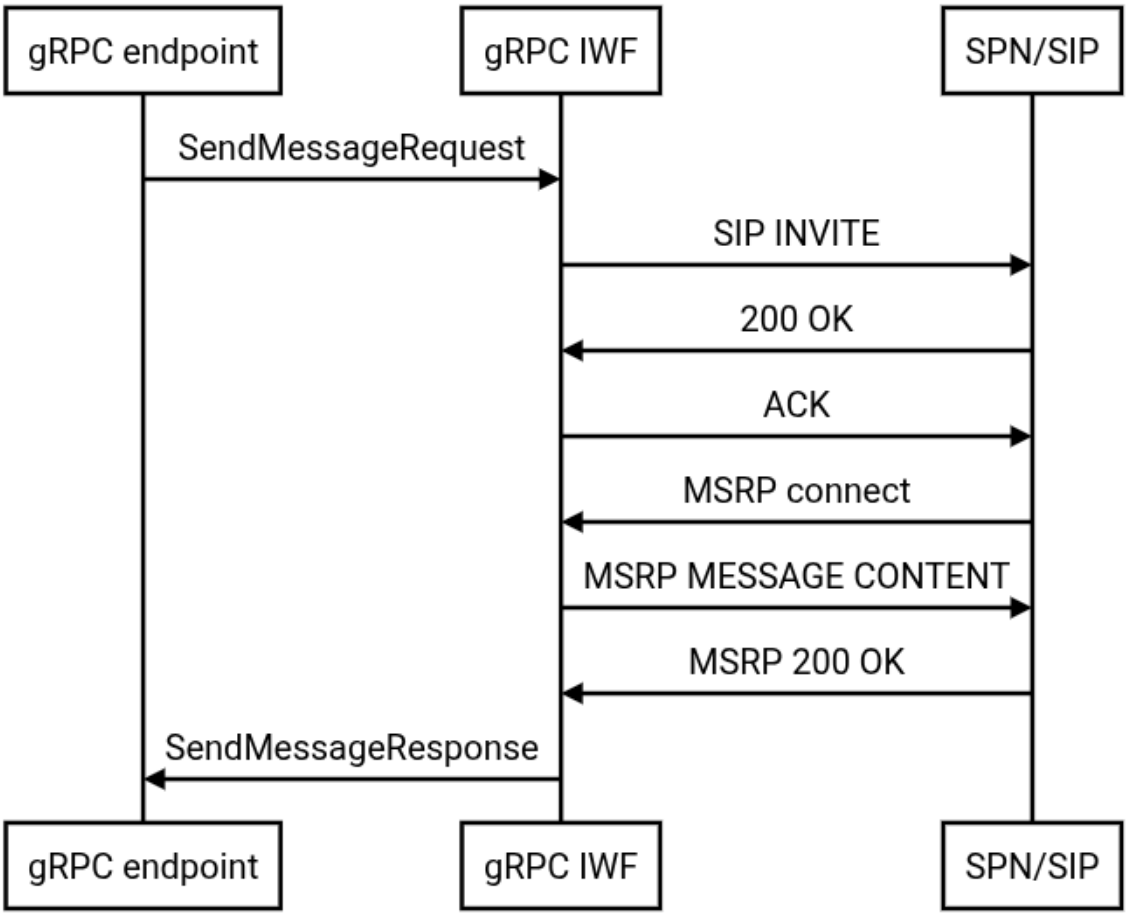

Figure 27: RCS gRPC to SIP

6.4.1.2 Group Chat

When receiving an RCS Group Chat gRPC message, the RCS gRPC IWF shall:

1. If a SIP session is not active between the RCS gRPC IWF and Service Provider Network, initiate a SIP session as per section 3.2.4 of [GSMA PRD-RCC.07] and connect over MSRP:

   a. The From header shall be constructed from the MessagingEvent.sender_id.
   a) The To header shall be constructed from the MessagingEvent.receiver_id.
   b) The Request URI shall be the conference sip URI. The RCS gRPC IWF must store the Storage of Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI) generated by the SIP Service Provider Network for the group.
   b. Copy relevant SIP headers from SendMessageRequest.request_header.sip_headers as defined in section 3.1.1.

2. Convert MessagingEvent to a CPIM message as per section 6.5.2.

3. Construct the MSRP message with CPIM payload as per [GSMA PRD-RCC.07].
4. Send the MSRP message as per [GSMA PRD-RCC.07].

This is illustrated in Figure 27.

6.4.1.3 Chatbot platform

When receiving an RCS gRPC message, the RCS gRPC IWF shall:

1. If a SIP session is not active between the RCS gRPC IWF and Service Provider Network, initiate a SIP session as per section 3.2.3 of [GSMA PRD-RCC.07] and connect over MSRP:

   a) The From header shall be constructed from the MessagingEvent.sender_id.
   b) The Request-URI and To headers shall be constructed from the MessagingEvent.receiver_id.
   c) Copy relevant SIP headers from SendMessageRequest.request_header.sip_headers as defined in section 3.1.1.

2. Convert MessagingEvent to a CPIM message as per section 6.5.2.
3. Construct the MSRP message with CPIM payload as per [GSMA PRD-RCC.07].
4. Send the MSRP message as per [GSMA PRD-RCC.07].

This is illustrated in Figure 27.

6.4.2 Message sending: SIP to gRPC

6.4.2.1 1-to-1 Conversation

When receiving an RCS 1-to-1 message over MSRP within a SIP session as per section 3.2.3 of [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. Send a SendMessage gRPC:

   a. Convert the CPIM message to a MessagingEvent as per section 5.2.3.
   b. The MessagingEvent sender_id shall be constructed from the SIP From header.
   c. The MessagingEvent receiver_id shall be constructed from the SIP To header.
   d. Copy relevant SIP headers to SendMessageRequest.request_header.sip_headers as defined in section 3.1.1.

2. Wait for the SendMessageResponse:

   a. If the gRPC returns an OK result, send 200OK as per [GSMA PRD-RCC.07].
   b. If the gRPC returns an error result, send mapped error status as per section 6.2.4 and as per [GSMA PRD-RCC.07].

This is illustrated in Figure 28.

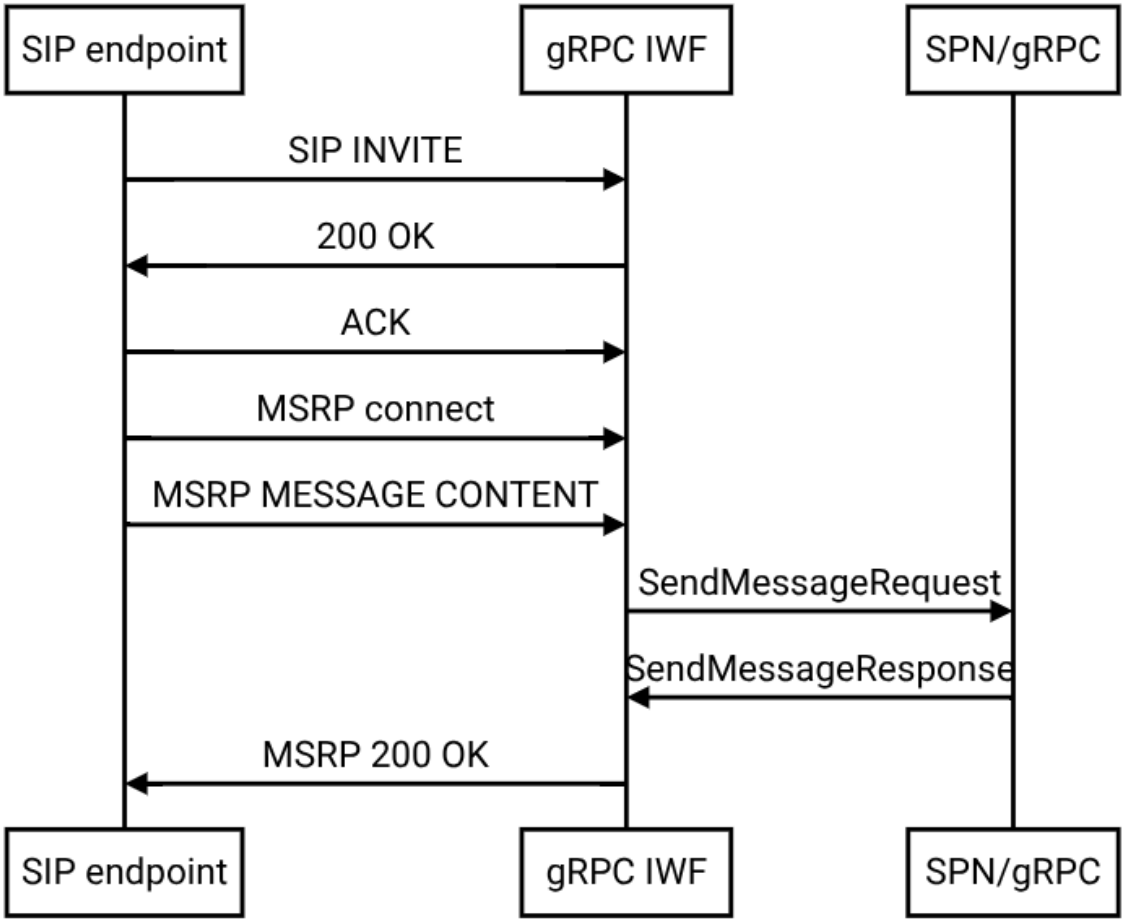

Figure 28: RCS SIP to gRPC

6.4.2.2 Group Chat

When receiving an RCS Group Chat message over MSRP within a SIP session as per section 3.2.4 of [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. Send a SendMessage gRPC:

    a. Convert the CPIM message to a MessagingEvent as per section 5.2.3.
    b. The MessagingEvent sender_id shall be constructed from the SIP From header.
    c. The MessagingEvent receiver_id shall be constructed from the SIP Conversation-ID header.
    d. Copy relevant SIP headers to SendMessageRequest.request_header.sip_headers as defined in section 3.1.1.

2. Wait for the SendMessageResponse response:

    a. If the gRPC returns an OK result, send 200OK as per [GSMA PRD-RCC.07].
    b. If the gRPC returns an error result, send mapped error status as per section 6.4.2 and as per [GSMA PRD-RCC.07].

This is illustrated in Figure 28.

6.4.2.3 Chatbot Platform

When receiving an RCS message over MSRP within a SIP session as per section 3.2.3 of [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. Send a SendMessage gRPC:
    a. Convert the CPIM message to a MessagingEvent as per section 5.2.3.
    b. The MessagingEvent sender_id shall be constructed from the SIP From header.
    c. The MessagingEvent receiver_id shall be constructed from the SIP To header.
    d. Copy relevant SIP headers to SendMessageRequest.request_header.sip_headers as defined in section 3.1.1.
2. Wait for the SendMessageResponse:
    a. If the gRPC returns an OK result, send 200OK as per [GSMA PRD-RCC.07].
    b. If the gRPC returns an error result, send mapped error status as per section 6.4.2 and as per [GSMA PRD-RCC.07].

This is illustrated in Figure 28.

6.5 CPIM body and Protocol Buffer

6.5.1 CPIM to Protocol Buffer

See procedure in section 5.2.3.

6.5.2 Protocol Buffer to CPIM

For each MessagingEvent Protocol Buffer message containing a ChatMessage, the RCS gRPC IWF shall construct a CPIM message:

1. For each MessagePart in ChatMessage.parts:
    a. Construct a MIME body:
        i. The Content-Type MIME header shall be set to MessagePart.content_type.
        ii. The MIME contents shall be set to MessagePart.contents.
2. Set the CPIM MIME payload:
    a. If there is a single MessagePart, the CPIM MIME is set to the single MIME payload constructed above.
    b. If there are multiple MessageParts, the CPIM MIME is set to a Multipart MIME consisting of the MIME constructed above.
3. Set the imdn.message-ID CPIM header from MessagingEvent.message_id.
4. Set the imdn.Disposition-Notification CPIM header from MessagingEvent.content_dispositions.
5. Add all CPIM headers (other than imdn.message-ID and imdn.Disposition-Notification) from MessagingEvent.Metadata.

6.5.2.1 IsComposing Notification

6.5.2.1.1 Protocol Buffer to CPIM body

When receiving a TypingNotification MessagingEvent defined in section 3.1.4 in an RCS Chat the RCS gRPC IWF shall:

1. Construct the `isComposing` XML CPIM body as per [GSMA PRD-RCC.11] with:
    a. Set the `state` element in the `isComposing` XML CPIM body with the state value in the TypingNotification.
    b. Set the `refresh` element in the `isComposing` XML CPIM body with the refresh_interval value in the TypingNotification.

6.5.2.1.2 CPIM body to Protocol Buffer

When receiving an RCS *application/im-iscomposing+xml* Content-Type over MSRP within the SIP session, the RCS gRPC IWF shall:

1. Parse the `state` element in the `isComposing` XML CPIM body as per [GSMA PRD-RCC.11].
2. Parse the `refresh` element in the `isComposing` XML CPIM body as per [GSMA PRD-RCC.11].
3. Construct a TypingNotification message as per section 3.1.4 and set the state and refresh values accordingly.

6.5.2.2 Delivery Notification

6.5.2.2.1 Protocol Buffer to CPIM body and headers

When receiving a MessageReceipt MessagingEvent defined in section 3.1.4 in an RCS Chat the RCS gRPC IWF shall construct in IMDN XML CPIM [GSMA PRD-RCC.07] with:

1. If MessageReceipt.receipt_type is RECEIPT_TYPE_READ, create a `display-notification` XML CPIM body, otherwise create a `delivery-notification` XML CPIM body.
2. Set the `message-id` node from MessageReceipt.receipt_info.message_id.
3. Set the `datetime` node from MessageReceipt.receipt_info.event_time.
4. In the `status` node, add a sub node of type:
    a) An empty `delivered` if MessageReceipt.receipt_type is RECEIPT_TYPE_POSITIVE_DELIVERY.
    b) An empty `displayed` if MessageReceipt.receipt_type is RECEIPT_TYPE_READ.
    a. A `failed` if MessageReceipt.receipt_type is RECEIPT_TYPE_NEGATIVE_DELIVERY.
        i. If the receipt_info contains a DeliveryFailedMlsServerReason, add an `mls-server-failure-reason` as a sub-node of the `failed` node as per section 7.7.2 of [GSMA PRD-RCC.16] with the failure reason.

ii. If the receipt_info contains a DeliveryFailedMlsClientReason, add an `mls-client-failure-reason` as a sub-node of the `failed` node as per section 7.7.2 of [GSMA PRD-RCC.16] with the failure reason.

6.5.2.2.2 CPIM body and headers to Protocol Buffer

When receiving an RCS *message/imdn+xml* Content-Type over MSRP within the SIP session, the RCS gRPC IWF shall construct a MessageReceipt:

1. Construct a MessageReceipt message:
   a. Set MessageReceipt.receipt_type to:
      i. RECEIPT_TYPE_POSITIVE_DELIVERY if there exists a `delivery-notification` node, and the `status` node contains a `delivered` node.
      ii. RECEIPT_TYPE_READ if there exists a `display-notification` node, and the `status` node contains a `displayed` node.
      iii. RECEIPT_TYPE_NEGATIVE_DELIVERY if there exists a `delivery-notification` node, and the `status` node contains a `failed` node which contains either an `mls-server-failure-reason` or an `mls-client-failure-reason`.
   b. Set DeliveryFailedMlsServerReason from the `mls-server-failure-reason` node if it exists.
   c. Set DeliveryFailedMlsClientReason from the `mls-client-failure-reason` node if it exists.
2. Set MessageReceipt.receipt_info.message_id from the `message-id` node.
3. Set MessageReceipt.receipt_info.event_time from the `datetime` node.

6.5.2.3 File Transfer

6.5.2.3.1 Protocol Buffer to CPIM body

When receiving a FileTransfer MessagingEvent defined in section 3.1.4 in an RCS Chat, the RCS gRPC IWF shall:

1. Construct the `file` XML CPIM body as per section 3.2.5.3 of [GSMA PRD-RCC.07] with:
   a. If present in the FileTransfer message, construct the thumbnail `file-info` node:
      i. Set the `file-size` element from FileTransfer.thumbnail.size_bytes.
      ii. Set the `content-type` element from FileTransfer.thumbnail.content_typeSet the `data` element attribute `until` from FileTransfer.thumbnail. valid_until_time.
      iii. Set the `data` element attribute `url` from FileTransfer.thumbnail.url.
   b. Construct the file `file-info` node:
      i. Set the file-disposition attribute from the FileTransfer.file.disposition.

ii. Set the `file-size` element from FileTransfer. file_info.size_bytes.
iii. Set the `content-type` element from FileTransfer.file.content_type.
iv. Set the `file-name` element from FileTransfer. file_info.file_name.
v. Set the `data` element attribute `until` from FileTransfer. file_info. valid_until_time.
vi. Set the `data` element attribute `url` from FileTransfer. file_info.url.
vii. For Audio Messaging only: Set the `am:playing-length` element from FileTransfer. file_info.playing_length.

6.5.2.3.2 CPIM body to Protocol Buffer

When receiving an RCS *application/vnd.gsma.rcs-ft-http+xml* Content-Type over MSRP within the SIP session, the RCS gRPC IWF shall:

1. Construct a FileTransfer message as per section 3.1.4 from the `file` XML CPIM body:

   a) If the thumbnail `file-info` node is present:

   i. Set FileTransfer.thumbnail.size_bytes from the `file-size` element
   ii. Set FileTransfer.thumbnail.content_type from the `content-type` element
   iii. Set FileTransfer.thumbnail. valid_until_time from the `data` element attribute `until`
   iv. Set FileTransfer.thumbnail.url from the `data` element attribute `url`

   b) From the file `file-info` node

   i. Set FileTransfer.file.disposition from the file-disposition attribute
   ii. Set FileTransfer. file_info.size_bytes from the `file-size` element
   iii. Set FileTransfer. file_info.content_type from the `content-type` element
   iv. Set FileTransfer. file_info.file_name from the `file-name` element
   v. Set FileTransfer.file_info. valid_until_time from the `data` element attribute `until`
   vi. Set FileTransfer.file_info.url from the `data` element attribute `url`
   vii. Audio Messaging only: Set FileTransfer. file_info.playing_length from the `am:playing-length` element

6.5.3 Group Chat

6.5.3.1 Group Chat Creation

6.5.3.1.1 Group Chat Creation: Request

6.5.3.1.1.1 Group Chat Creation: gRPC to SIP

Upon receiving a CreateGroupRequest, to translate into a SIP INVITE that creates a new Group Chat as per [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. Construct the From and P-Preferred-Identity headers from the CreateGroupRequest.header.requester_id.
2. Construct the Requester-URI and To headers with the conference factory URI.

3. Construct the Conversation-ID and Contribution-ID using the CreateGroupRequest.group_id.
4. Copy relevant SIP headers from CreateGroupRequest.header.sip_headers as defined in section 3.1.1.
5. Construct the Multi-part MIME body as per [GSMA PRD-RCC.07]:

   a) Construct the `recipient-lists` as per [GSMA PRD-RCC.11] by: enumerating each CreateGroupRequest.user_ids as an entry resource-lists and constructing a tel uri.

      i. For each tel uri creating a new `entry` element in the `list` node of the `resource-list`. Setting the `uri` attribute with the tel uri.

   b) If client_mls_control_rcs_message is not present: Construct the `groupstate` as per section 6.5.3.2.8.3.2.

   c) If client_mls_control_rcs_message is present: follow the end-to-end encrypted Group Chats: follow the procedure as per section 8.1.2 of [GSMA PRD-RCC.16].

6. Wait for the SIP response:

   a) If status is 200 OK:

      i. Store the Storage of Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI) locally.
      i. Construct and return CreateGroupResponse.

   b) For all other statuses:

      i. Map to gRPC error as per section 6.2.1.
      ii. Return the gRPC error.

This is illustrated in Figure 29.

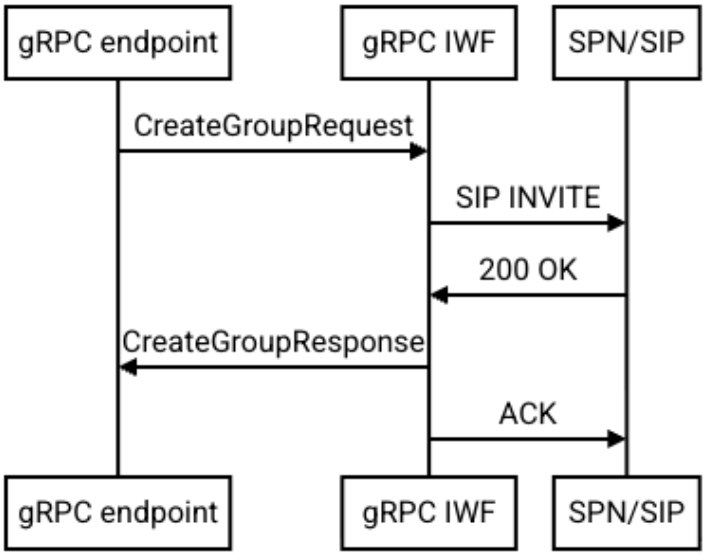

Figure 29: CreateGroup gRPC to SIP INVITE translation

6.5.3.1.1.2 Group Chat Creation: SIP to gRPC

Upon receiving a SIP INVITE that creates a new Group Chat as per [GSMA PRD-RCC.07], to translate to RCS gRPC the RCS gRPC IWF shall:

1. Send a CreateGroup gRPC:
    a) Set the CreateGroupRequest.header.requester_id using the SIP From header.
    b) Set the CreateGroupRequest.group_id using the Conversation-ID header.
    c) Set the CreateGroupRequest.member_ids from the entries in the recipient-list-history.
    d) Copy relevant SIP INVITE headers to the CreateGroupRequest.header.sip_headers as defined in section 3.1.13.1.4.
    e) Construct the CreateGroupRequest.member_ids.rcs_id by adding each tel uri in the `entry` element in the `list` node of the `resource-list`.
    f) Construct the CreateGroupRequest.metadata as per section 6.5.3.2.8.3.1.
2. Store the Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI) locally.
3. Wait for the gRPC response:
    a) If status is OK:
        i. Construct and return SIP 200 OK based on SIP INVITE as per [GSMA PRD-RCC.07].
    b) For all other statuses:
        i. Construct the SIP error as per section 6.2.2.
        ii. Return SIP error response based on SIP INVITE as per [GSMA PRD-RCC.07].

This is illustrated in Figure 30.

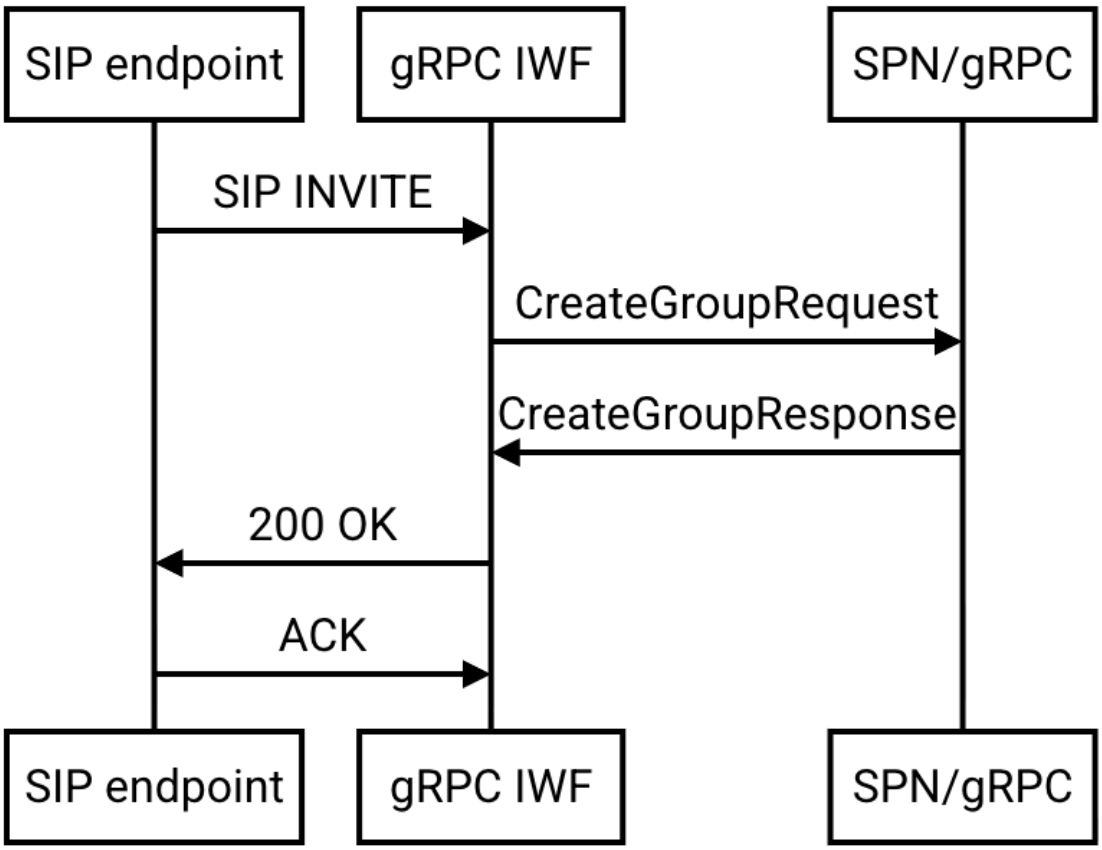

Figure 30: SIP INVITE to CreateGroup gRPC translation

6.5.3.1.2 Group Chat Creation: Notification

6.5.3.1.2.1 Group Chat Creation: gRPC to SIP

Upon receiving a SendMessage RPC with a SendMessageRequest.message.group_change_event.create_group, to translate the Group Chat members as per [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. Construct and send a SIP INVITE as per section 3.2.4.10 of [GSMA PRD-RCC.07]:

    a) Store the Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI) locally.
    b) Construct the From and Contact headers using the SendMessageRequest.message.group_change_event.create_group.group_info.conference_uri.
    c) Construct the Request-URI and To headers using the SendMessageRequest.message.receiver_id.
    d) Construct the Conversation-ID and Contribution-ID using the SendMessageRequest.message.group_change_event.create_group.group_info.group_id.
    e) Copy relevant SIP headers from SendMessageRequest.header.sip_headers as defined in section 3.1.1.

f) Construct the `<conference-info>` element:

i. Construct the `<users>` element with a `<user>` sub-element for each participant being added from SendMessageRequest.message.group_create_event.create_group.group_info.members.
ii. Each `<user>` element shall set an `entity` attribute using SendMessageRequest.message.group_create_event.create_group.group_info.members and a `<user>` element that has a `<status>` element as “connected“.
iii. For the Group Chat subject, construct the `<conference-description>` subelement as per section 6.5.3.2.8.1.2.
iv. If present, for the Group Chat icon, construct the `<conference-description>` subelement as per section 6.5.3.2.8.1.2.

2. Wait for the SIP response:

a) If status is 200 OK:

i. Construct and return a SendMessageResponse.

b) For all other statuses:

i. Construct the gRPC error as per section 6.2.1.
ii. Return the gRPC error.

This is illustrated in Figure 31.

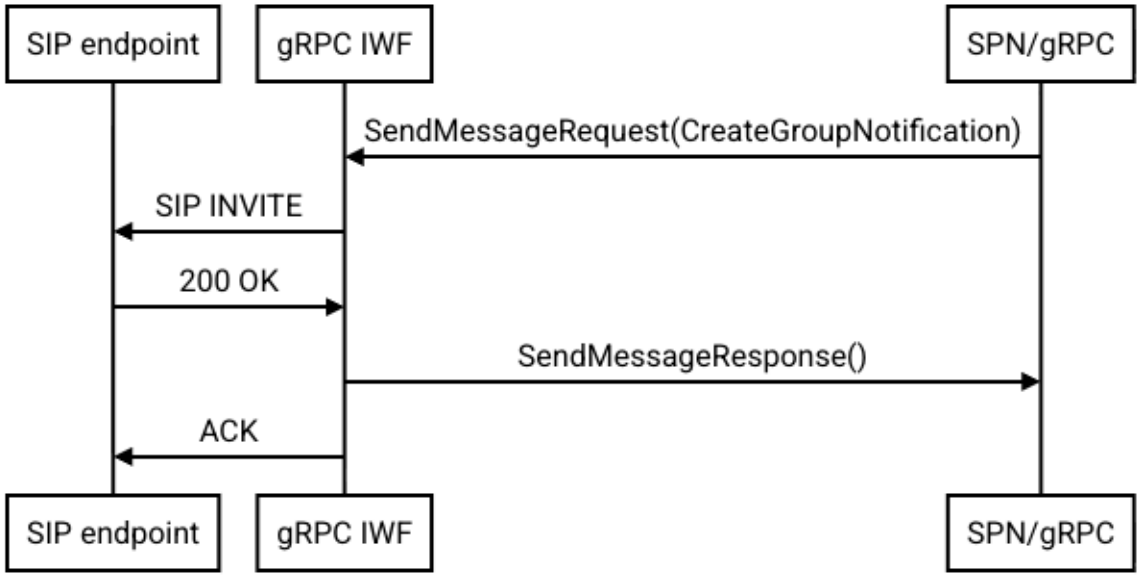

Figure 31: CreateGroup gRPC notification to SIP translation

6.5.3.1.2.2 Group Chat Creation: SIP to gRPC

Upon receiving a SIP INVITE for a new participant of a Group Chat as per [GSMA PRD-RCC.07], to translate to RCS gRPC the RCS gRPC IWF shall:

1. Construct a CreateGroupNotification:

    a) Set the group_info.group_id using the Conversation-ID header.
    b) Set the group_info.conference_uri from the From header.
    c) Construct the group_info.members using each `<user>` element with an `entity` attribute to set the RcsId and only choose the `<status>` element as “connected“ as per [GSMA PRD-RCC.11] for participants that are joining the Group Session.
    d) Construct the group_info.metadata as per section 6.5.3.2.8.1.1.

2. Construct a MessagingEvent:

    a) Set the sender_id using SIP Referred-By header.
    b) Set the receiver_id using the SIP To header.
    c) Set the group_id using the Conversation-ID header.
    d) Set the payload as a GroupChangeEvent and include the CreateGroupNotification.

3. Construct a SendMessageRequest:

    a) Set the SendMessageRequest.header.requester_id and SendMessageRequest.message.sender_id using the SIP Referred-By header.
    b) Copy relevant SIP headers to the SendMessageRequest.header.sip_headers as defined in section 3.1.1.

        i. Construct and return SIP 200 OK based on SIP INVITE as per [GSMA PRD-RCC.07].
        ii. Send a SIP SUBSCRIBE as per section 3.2.4.8 of [GSMA PRD-RCC.07]:

            1. Construct the From and P-Preferred-Identity headers from the requester_id.
            2. Construct the Request-URI and To headers from the Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI).

3. Construct the Event header with the conference value.

    c) For all other statuses:

        i. Construct the SIP error as per section 6.2.2.
        ii. Return SIP error response based on SIP INVITE as per [GSMA PRD-RCC.07].

This is illustrated in Figure 32.

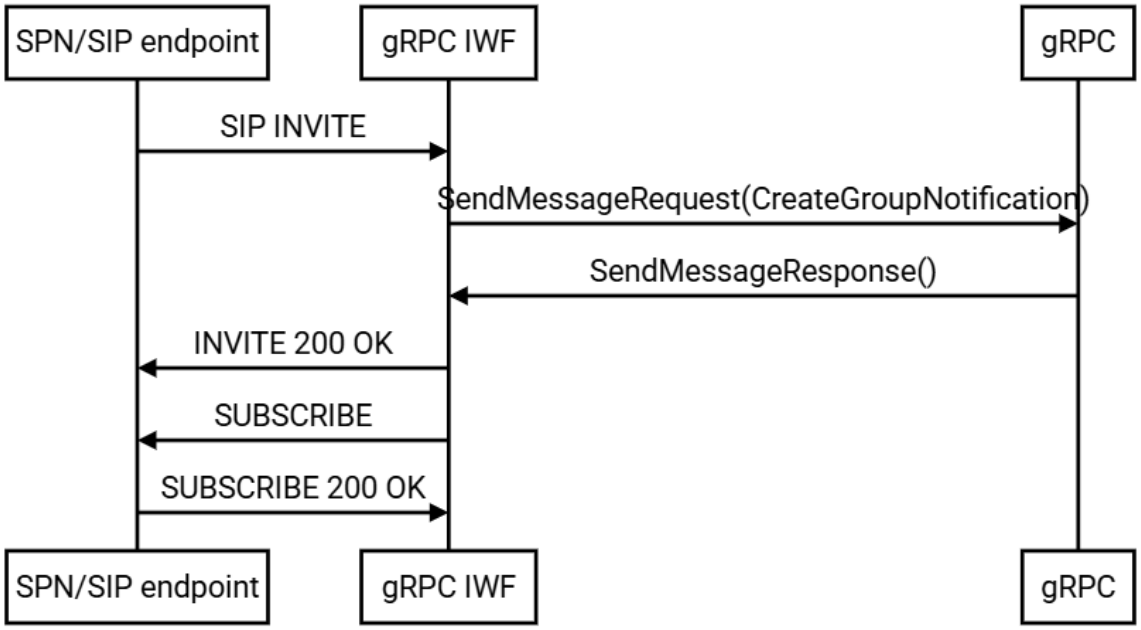

Figure 32: SIP INVITE to CreateGroupNotification translation

6.5.3.2 Group Chat Updates

6.5.3.2.1 General handling of SIP SUBSCRIBE

Upon receiving a SIP SUBSCRIBE with the event conference header as per [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. Send a GetGroupInfosRequest:
    a) Construct the group_ids from the Request URI.
    b) Copy relevant SIP SUBSCRIBE headers to the SendMessageRequest.header.sip_headers as defined in section 3.1.1.
2. Wait for the gRPC response:
    a) If status is OK:
        i. Respond with SIP 200 OK.
    b) For all other statuses:
        i. Construct SIP error as per section 6.2.4.
        ii. Return SIP error response as per [GSMA PRD-RCC.07].
3. Construct and send a SIP NOTIFY as per section 3.2.4.8 of [GSMA PRD-RCC.07]:
    a) Construct the From header using the locally stored Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI).
    b) Construct the Request-URI and To header using the SIP SUBSCRIBE's From header.

c) Copy relevant SIP headers from GetGroupInfosResponse.header.sip_headers as defined in section 3.1.1.
d) Construct the <conference-info> element:

i. Construct the <users> element with a <user> sub-element for each participant being added from GetGroupInfosResponse.group_infos.members.
ii. Each <user> element shall set an entity attribute using GetGroupInfosResponse.group_info.members and a <user> element that has a <status> element as “connected“.
iii. Construct the <conference-description> as per 6.5.3.2.8.1.2.

4. Wait for the SIP response:

a) If status is 200 OK:

i. Construct and return a SendMessageResponse.

b) For all other statuses:

i. Construct the gRPC error as per section 6.2.1.
ii. Return the gRPC error.

6.5.3.2.2 Reception of a conference event package: SIP to gRPC

Upon receiving a SIP NOTIFY containing an *application/conference-info+xml*, the RCS gRPC IWF shall:

1. Parse the XML to identify changes to the Group Chat and:

a) For each <user> sub-element containing the <status> connected subelement is present, the RCS gRPC IWF shall follow the procedures in section 6.5.3.2.5.2.2.
b) For each <user> sub-element containing the <status> disconnected is present, the RCS gRPC IWF shall follow the procedures in section 6.5.3.2.6.2.2.
c) For a <conference-description> containing a <subject> and/or a <icon> node, the RCS gRPC IWF shall follow the procedures in section 6.5.3.2.4.2.2.

6.5.3.2.3 Group Chat session idle timeout

Upon receiving a SIP BYE as a result of the conference focus of the Messaging Server ending the Group Chat session as per section 3.2.4.9.1 [GSMA PRD-RCC.07], the RCS gRPC IWF shall return 200OK.

6.5.3.2.4 Group Chat Metadata update

6.5.3.2.4.1 Group Chat Metadata update: Request

6.5.3.2.4.1.1 Group Chat Metadata update: gRPC to SIP

Upon receiving a ChangeGroupMetadataRequest, to manage the Group Chat meta-information as per [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. If a SIP session is not established between the RCS gRPC IWF and the controlling function, send a SIP INVITE as per section 3.2.4.10 of [GSMA PRD-RCC.07]:

   a) Construct the From and P-Preferred-Identity headers from the ChangeGroupMetadataRequest.header.requester_id.
   b) Construct the Conversation-ID and Contribution-ID using the ChangeGroupMetadataRequest.group_id.
   c) Look up previously stored Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI) and insert in the Request-URI and To SIP header.
   d) Copy relevant SIP headers from ChangeGroupMetadataRequest.header.sip_headers as defined in section 3.1.1.
   e) Wait for the SIP 200 OK.

2. Send the MSRP payload with a `<cpm-group-management>` as per section 6.5.3.2.8.2.2.
3. Wait for the MSRP response:

   a) If status is 200 OK:

      i. Construct and return a ChangeGroupMetadataResponse.

   b) For all other statuses:

      i. Construct the gRPC error as per section 6.2.3.
      ii. Return the gRPC error.

This is illustrated in Figure 33.

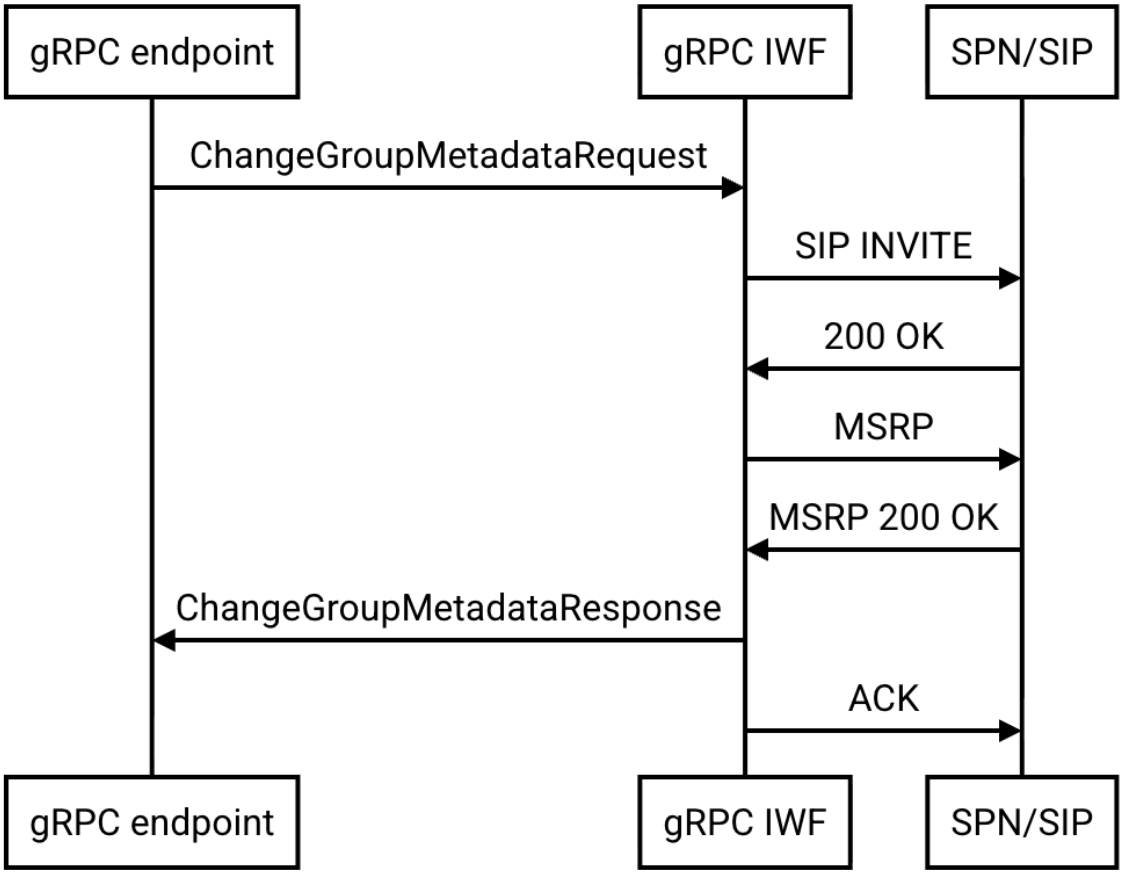

Figure 33: ChangeGroupMetadata gRPC to SIP/MSRP translation

6.5.3.2.4.1.2 Group Chat Metadata update: SIP to gRPC

Upon receiving a request that changes Group Chat meta-information as per [GSMA PRD-RCC.07], to translate to RCS gRPC the RCS gRPC IWF shall:

1. Send a ChangeGroupMetadataRequest:
    a) Set the ChangeGroupMetadataRequest.header.requester_id using the SIP From header.
    b) Set the ChangeGroupMetadataRequest.metadata_update as per section 6.5.3.2.8.2.1.
    c) Copy relevant SIP headers to the ChangeGroupMetadataRequest.header.sip_headers as defined in section 3.1.1.
2. Wait for the gRPC response:
    a) If status is OK:
        i. Respond with MSRP 200 OK.
    b) For all other statuses:
        i. Construct the MSRP error as per section 6.2.4.
        ii. Return MSRP error response as per [GSMA PRD-RCC.07].

This is illustrated in Figure 34.

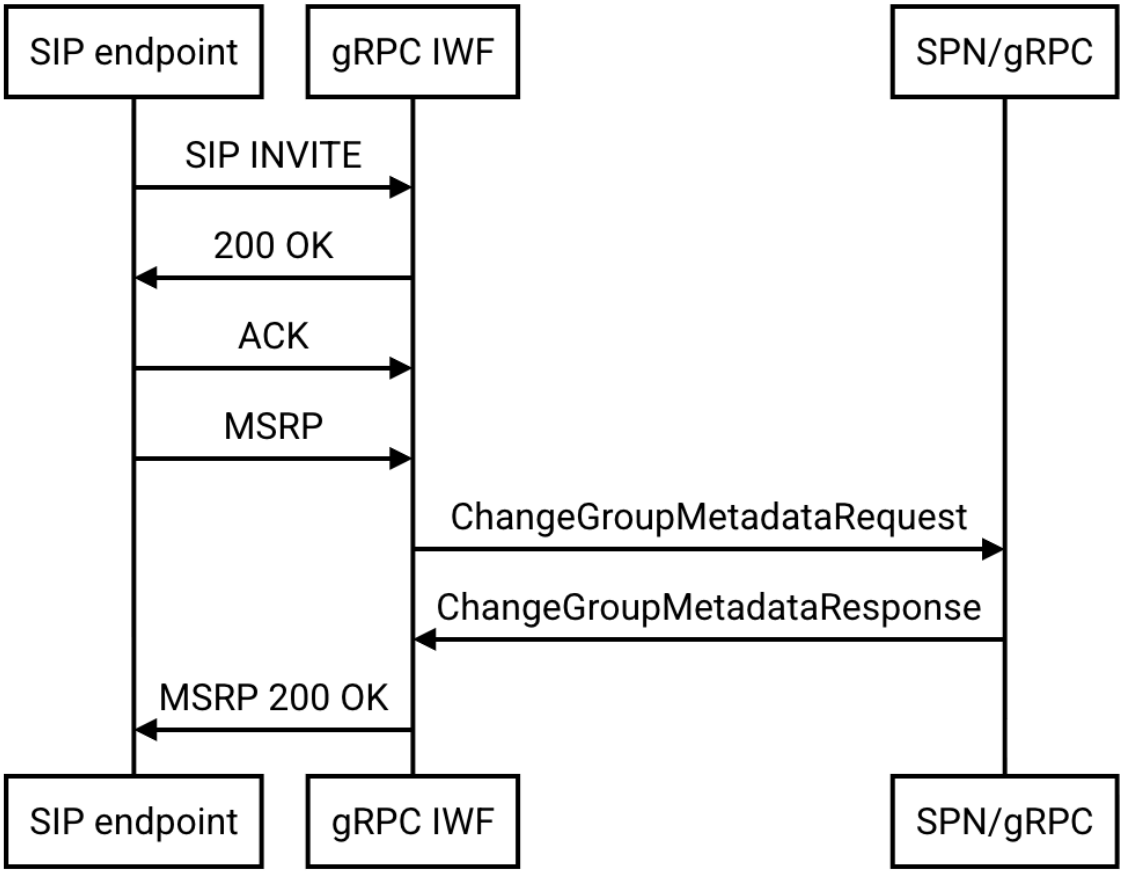

Figure 34: SIP/MSRP to ChangeGroupMetadata gRPC translation

6.5.3.2.4.2 Group Chat Metadata update: Notification

6.5.3.2.4.2.1 Group Chat Metadata update: gRPC to SIP

Upon receiving a SendMessage RPC with SendMessageRequest.message.group_change_event. change_group_metadata, to change the Group Chat metadata as per [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. If a SIP session is not established between the RCS gRPC IWF and the controlling function, send a SIP INVITE as per section 3.2.4.10 of [GSMA PRD-RCC.07]:

    a) Construct the From and P-Preferred-Identity headers from the SendMessageRequest.header.requester_id.
    b) Construct the Request-URI and To header from the locally stored Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI).
    c) Construct the Conversation-ID and Contribution-ID using the SendMessageRequest.message.group_id.
    d) Copy relevant SIP headers from SendMessageRequest.header.sip_headers as defined in section 3.1.1.
    e) Wait for the SIP 200 OK.

2. Wait for a SIP SUBSCRIBE and handle as per section 6.5.3.2.1.

If a SIP session is established between the RCS gRPC IWF and the controlling function, the RCS gRPC IWF shall:

3. Construct the `<conference-info>` element as per section 6.5.3.2.8.1.2.
4. Send the `<conference-info>` as per section 3.2.4.8 of [GSMA PRD-RCC.07] and [GSMA PRD-RCC.11][GSMA PRD-RCC.11].

    a) For the SIP NOTIFY mechanism defined in section 3.2.4.8 of [GSMA PRD-RCC.07], copy relevant SIP headers from SendMessageRequest.header.sip_headers as defined in section 3.1.1.

5. Wait for the response:

    a) If status is 200 OK:

        i. Construct and return a SendMessageResponse.

    b) For all other statuses:

        i. Construct the gRPC error as per section 6.2.1 and 6.2.3.
        ii. Return the gRPC error.

This is illustrated in Figure 35.

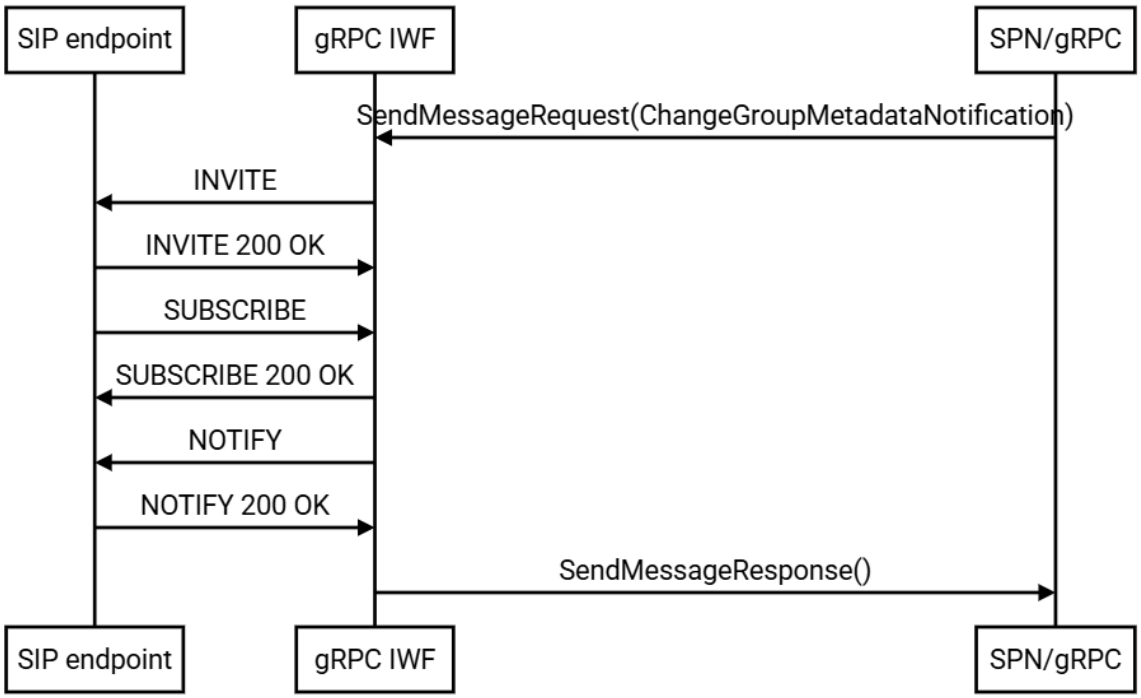

Figure 35: ChangeGroupMetadataNotification gRPC to SIP translation

6.5.3.2.4.2.2 Group Chat Metadata update: SIP to gRPC

Upon receiving a NOTIFY request that modifies the Group Chat metadata as per [GSMA PRD-RCC.07], to translate to RCS gRPC the RCS gRPC IWF shall:

1. Send a SendMessageRequest:

   a) Construct a ChangeGroupMetadataNotification:

      i. Set the group_id with the Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI).
      i. Construct the GroupMetadata as per section 6.5.3.2.8.1.1.

   b) Construct a MessagingEvent:

      i. Set the sender_id using the From header.
      ii. Set the receiver_id using the To header.
      iii. Include the ChangeGroupMetadataNotification as group_change_event. change_group_metadata.

   c) Construct the SendMessageRequest

      i. Set the header.requester_id using the Referred-By.
      ii. Copy relevant SIP headers to the SendMessageRequest.header.sip_headers as defined in section 3.1.1.
      iii. Include the MessagingEvent.

2. Wait for the gRPC response:

   a. If status is OK:

      i. Construct and return SIP 200 OK based on NOTIFY as per [GSMA PRD-RCC.07].

   b) For all other statuses:

      i. Construct the SIP error as per section 6.2.2.
      ii. Return SIP error response based on NOTIFY as per [GSMA PRD-RCC.07].

This is illustrated in Figure 36.

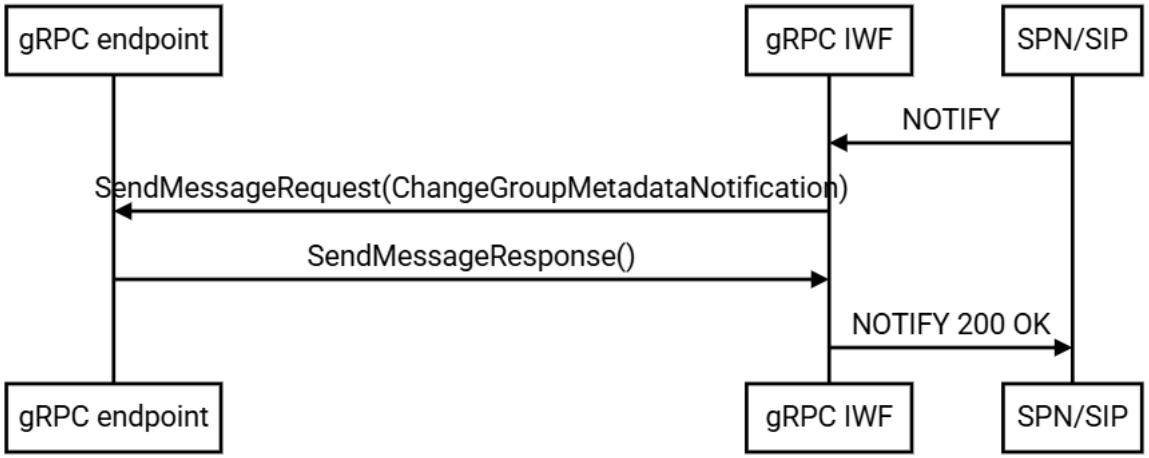

Figure 36: SIP to ChangeGroupMetadataNotification translation

6.5.3.2.5 Adding a participant

6.5.3.2.5.1 Adding a participant: Request

6.5.3.2.5.1.1 Adding a participant: gRPC to SIP

Upon receiving an AddGroupUsersRequest request, to translate to SIP as per [GSMA PRD-RCC.07] the RCS gRPC IWF shall:

1. If a SIP session is not established between the RCS gRPC IWF and the controlling function, send a SIP INVITE as per section 3.2.4.10 of [GSMA PRD-RCC.07]:

    a) Construct the From and P-Preferred-Identity headers from the AddGroupUsersRequest.header.requester_id.
    b) Construct the Request-URI and To header using the locally stored Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI).
    c) Construct the Conversation-ID and Contribution-ID using the AddGroupUsersRequest.group_id.
    d) Copy relevant SIP headers from AddGroupUsersRequest.header.sip_headers as defined in section 3.1.1.
    e) Wait for the SIP response.

2. Construct and send a SIP REFER for each participant being added to the Group Chat as per [GSMA PRD-RCC.07].

    a) Construct the From and P-Preferred-Identity headers from the AddGroupUsersRequest.header.requester_id.
    b) Construct the Request-URI and To headers using the locally stored Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI).
    c) Construct the Conversation-ID and Contribution-ID using the AddGroupUsersRequest.group_id.
    d) Construct the Refer-To header using the AddGroupUsersRequest.user_ids
    e) Copy relevant SIP headers from AddGroupUsersRequest.header.sip_headers as defined in section 3.1.1.

3. Wait for the SIP response:

    a) If the status is 200 OK:

        i. Construct and return a AddGroupUsersResponse.

    b) For all other statuses:

        i. Construct the gRPC error as per section 6.2.1.
        ii. Return the gRPC error.

This is illustrated in Figure 37.

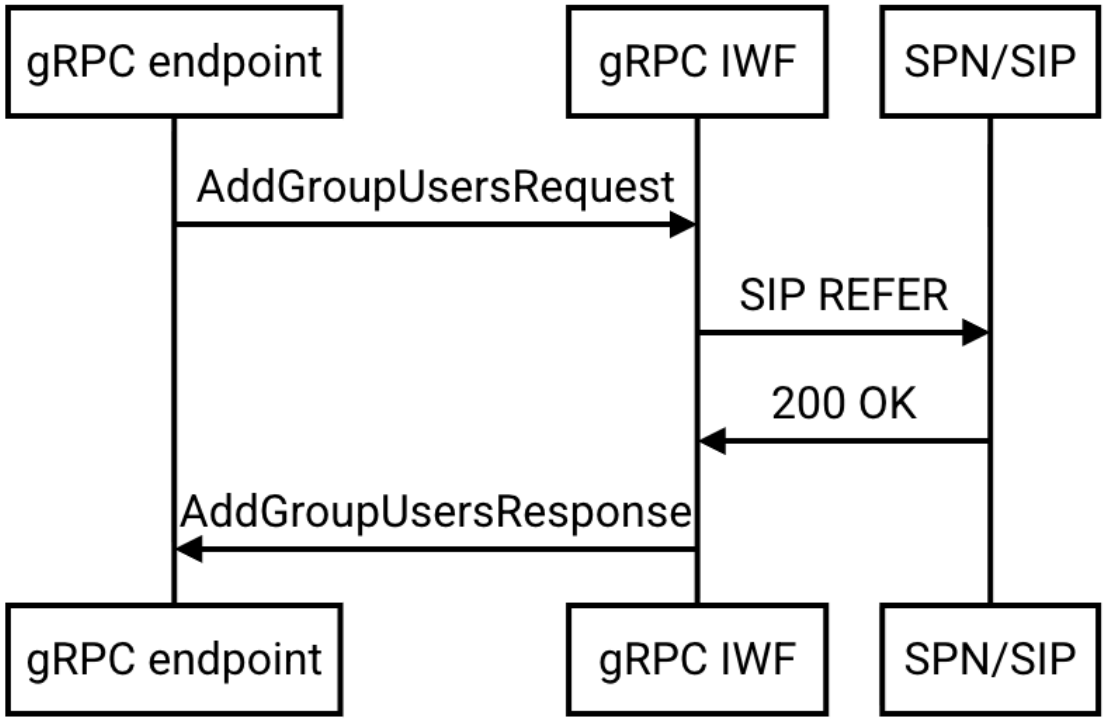

Figure 37: AddGroupUsersRequest gRPC to SIP translation

6.5.3.2.5.1.2 Adding a participant: SIP to gRPC

Upon receiving a SIP REFER that adds a participant to a Group Chat as per [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. Send a AddGroupUsers gRPC:

    a) Set the AddGroupUsersRequest.header.requester_id using the SIP From header.
    b) Set the AddGroupUsersRequest.user_ids using the Refer-To header.
    c) Set the AddGroupUsersRequest.group_id using the Conversation-ID header.
    d) Copy relevant SIP REFER headers to the AddGroupUsersRequest.header.sip_headers as defined in section 3.1.1.

2. Wait for the gRPC response:

    a) If status is OK:

        i. Construct and return SIP 200 OK based on REFER as [GSMA PRD-RCC.07].

    b) For all other statuses:

        i. Construct the SIP error as per section 6.2.2.
        ii. Return SIP error response based on REFER as per [GSMA PRD-RCC.07].

This is illustrated in Figure 38.

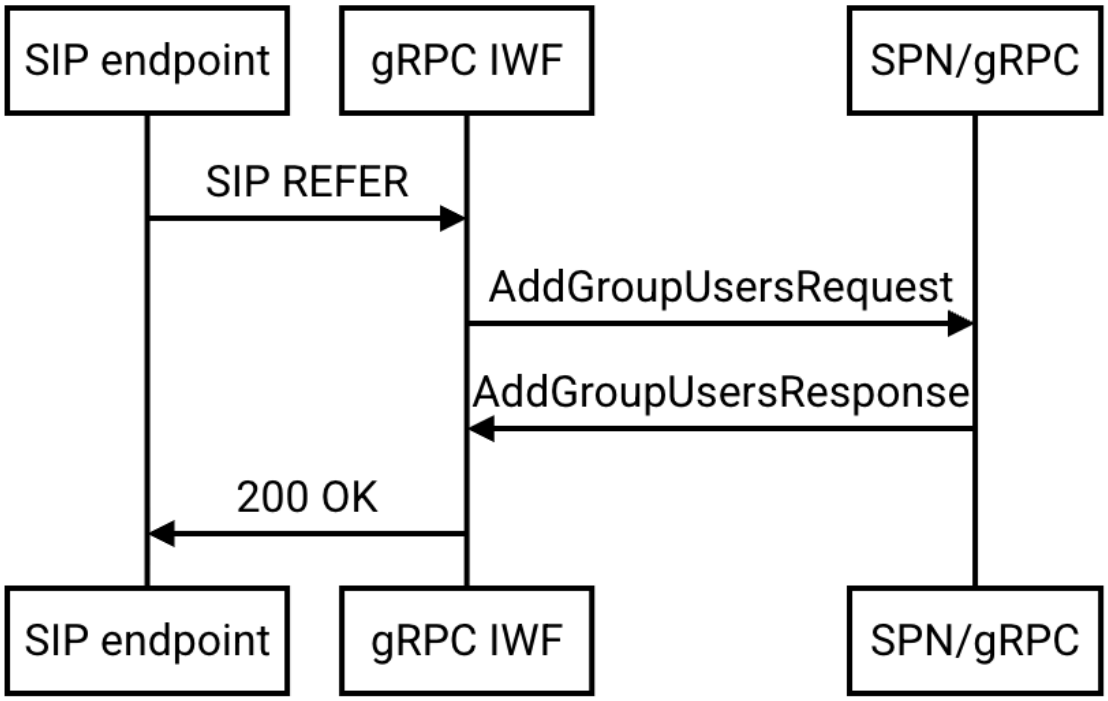

Figure 38: SIP REFER to AddGroupUsersRequest gRPC

6.5.3.2.5.2 Adding a participant: Notification

6.5.3.2.5.2.1 Adding a participant: gRPC to SIP

Upon receiving a SendMessage RPC with SendMessageRequest.message.group_change_event.add_users, to add participants to the Group Chat as per [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. If a SIP session is not established between the RCS gRPC IWF and the controlling function, send a SIP INVITE as per section 3.2.4.10 of [GSMA PRD-RCC.07]:

    a) Construct the From and P-Preferred-Identity headers from the SendMessageRequest.header.requester_id.
    b) Construct the Request-URI and To headers from the locally stored Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI).
    c) Construct the Conversation-ID and Contribution-ID using the SendMessageRequest.message.group_id.
    d) Copy relevant SIP headers from SendMessageRequest.header.sip_headers as defined in section 3.1.1.
    e) Wait for the SIP 200 OK.
    f) Wait for a SIP SUBSCRIBE and handle as per section 6.5.3.2.1.

If a SIP session is established between the IWF and the controlling function, the RCS gRPC IWF shall:

2. Construct the `<conference-info>` element

a) Construct the `<users>` element with a `<user>` sub-element for each participant being added from SendMessageRequest.message.group_change_event.add_users.added_users.
b) Each `<user>` element shall set the `entity` attribute using SendMessageRequest.message.group_change_event.add_users.added_users and a `<user>` element that has a `<status>` element as “connected“.

3. Send the `<conference-info>` as per section 3.2.4.8 of [GSMA PRD-RCC.07] and [GSMA PRD-RCC.11].

   a) For the SIP NOTIFY mechanism defined in section 3.2.4.8 of [GSMA PRD-RCC.07], copy relevant SIP headers from SendMessageRequest.header.sip_headers as defined in section 3.1.1.

4. Wait for the response:

   a) If status is 200 OK:

      i. Construct and return a SendMessageResponse.

   b) For all other statuses:

      i. Construct the gRPC error as per section 6.2.1 and 6.2.3.
      ii. Return the gRPC error.

This is illustrated in Figure 39.

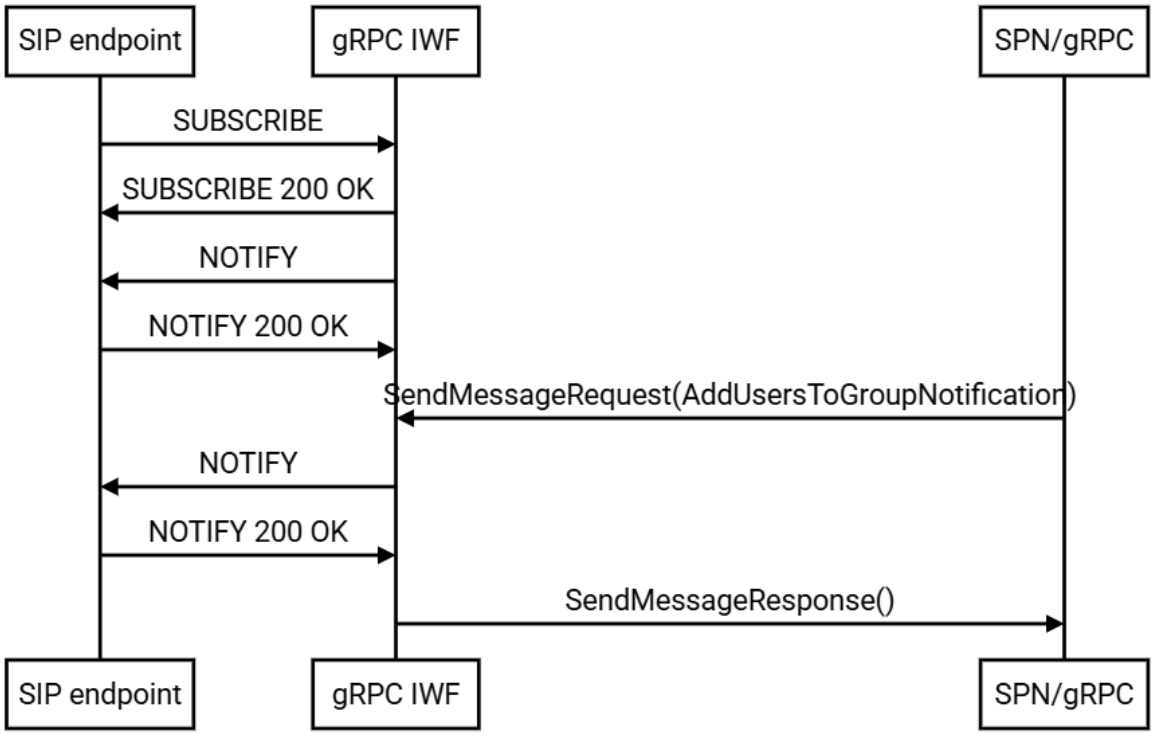

Figure 39: AddUsersToGroupNotification gRPC to SIP translation

6.5.3.2.5.2.2 Adding a participant: SIP to gRPC

Upon receiving a NOTIFY request that adds participants as per [GSMA PRD-RCC.07], to translate to RCS gRPC the RCS gRPC IWF shall:

1. Send a SendMessageRequest:
    a. Construct a AddUsersToGroupNotification:
        i. Construct added_users using each `<user>` element with an `entity` attribute to set the RcsId and only include the ones that have a `<status>` element as “connected“ as per [GSMA PRD-RCC.11] for participants that have joined the Group Chat.
    b) Construct a MessagingEvent:
        i. Set the sender_id using the Referred-By header.
        ii. Set the receiver_id using the To header.
        iii. Include the AddUsersToGroupNotification as group_change_event.add_users.
    c) Construct the SendMessageRequest:
        i. Set the header.requester_id using the Referred-By.
        ii. Copy relevant SIP headers to the SendMessageRequest.header.sip_headers as defined in section 3.1.1.
        iii. Include the MessagingEvent.
2. Wait for the gRPC response
    a) If status is OK:
        i. Construct and return SIP 200 OK based on NOTIFY as per [GSMA PRD-RCC.07].
    b) For all other statuses:
        i. Construct the SIP error as per section 6.2.2.
        ii. Return SIP error response based on NOTIFY as per [GSMA PRD-RCC.07].

This is illustrated in Figure 40.

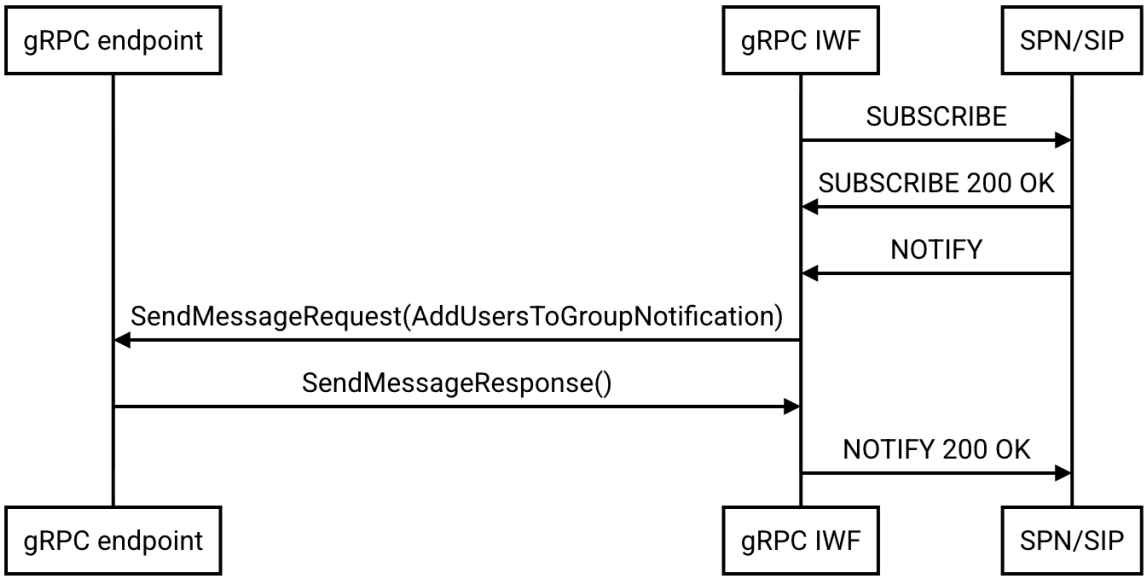

Figure 40: SIP to AddUsersToGroupNotification translation

6.5.3.2.6 Removing a participant

6.5.3.2.6.1 Removing a participant: Request

6.5.3.2.6.1.1 Removing a participant: gRPC to SIP

Upon receiving a RemoveGroupUsersRequest request, to translate to SIP as [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. If a SIP session is not established between the RCS gRPC IWF and the controlling function, send a SIP INVITE as per section 3.2.4.10 of [GSMA PRD-RCC.07]:

    a) Construct the From and P-Preferred-Identity headers from the RemoveGroupUsersRequest.header.requester_id.
    b) Construct the Request-URI and To headers from the locally stored Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI).
    c) Construct the Conversation-ID and Contribution-ID using the AddGroupUsersRequest.group_id.
    d) Wait for the SIP 200 OK.

Upon receiving the SIP 200 OK and if the requester is not the participant being removed from the Group Chat, the RCS gRPC IWF shall:

2. Send a SIP REFER for the participant being added to the Group Chat.
3. Construct the From and P-Preferred-Identity headers from the RemoveGroupUsersRequest.header.requester_id:

    a. Construct the Conversation-ID and Contribution-ID using the RemoveGroupUsersRequest.group_id.
    b) Construct the Refer-To header as per [GSMA PRD-RCC.07] using the participant from AddGroupUsersRequest.user_ids.

c) Copy relevant SIP headers from RemoveGroupUsersRequest.header.sip_headers as defined in section 3.1.1.

4. Wait for the SIP response:

   a. If status is 200 OK:

      i. Construct and return a RemoveGroupUsersResponse.

   b) For all other statuses:

      i. Construct the gRPC error as per section 6.2.1.
      ii. Return the gRPC error.

This is illustrated in Figure 41.

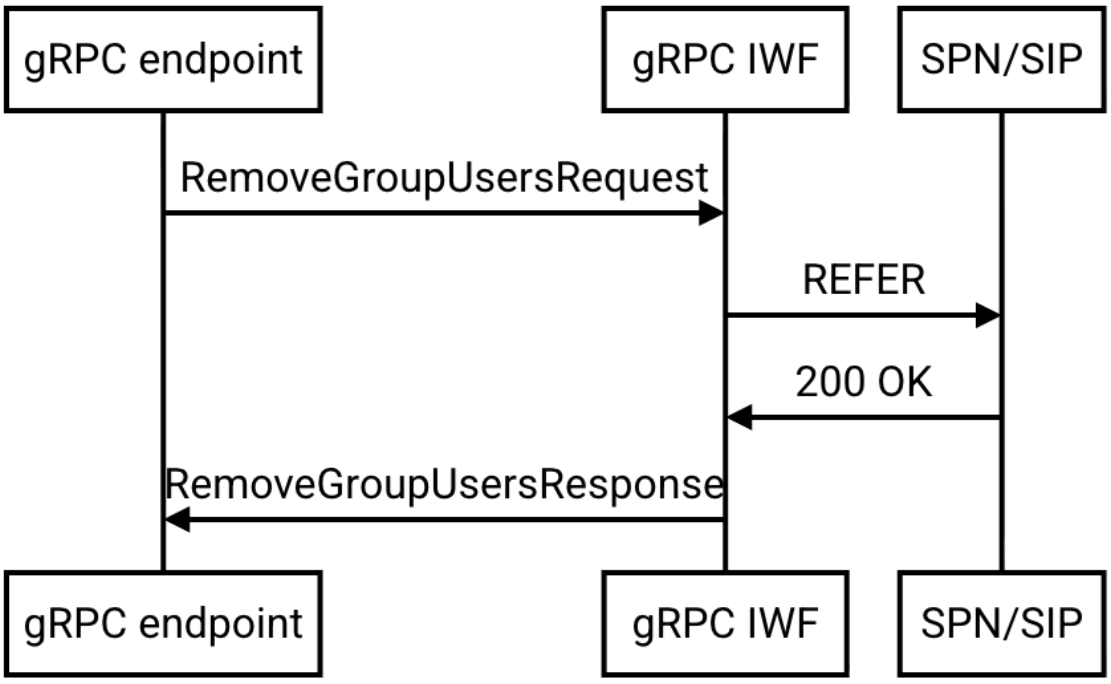

Figure 41: RemoveGroupUsers gRPC to SIP translation

6.5.3.2.6.1.2 Removing a participant: SIP to gRPC

Upon receiving a SIP REFER that removes a participant from a Group Chat as per [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. Send a RemoveGroupUsers gRPC:

   a) Set the RemoveGroupUsersRequest.header.requester_id using the SIP Referred-By header.
   b) Set the RemoveGroupUsersRequest.user_ids using the Refer-To header.
   c) Set the RemoveGroupUsersRequest.group_id using the Conversation-ID header.
   d) Copy relevant SIP REFER headers to the RemoveGroupUsersRequest. header.sip_headers as defined in section 3.1.1.

2. Wait for the gRPC response:

    a) If status is OK:

        i. Construct and return SIP 200 OK based on REFER as per [GSMA PRD-RCC.07].

    b) For all other statuses:

        i. Construct the SIP error as per section 6.2.2.
        ii. Return SIP error response based on REFER as per [GSMA PRD-RCC.07].

This is illustrated in Figure 42.

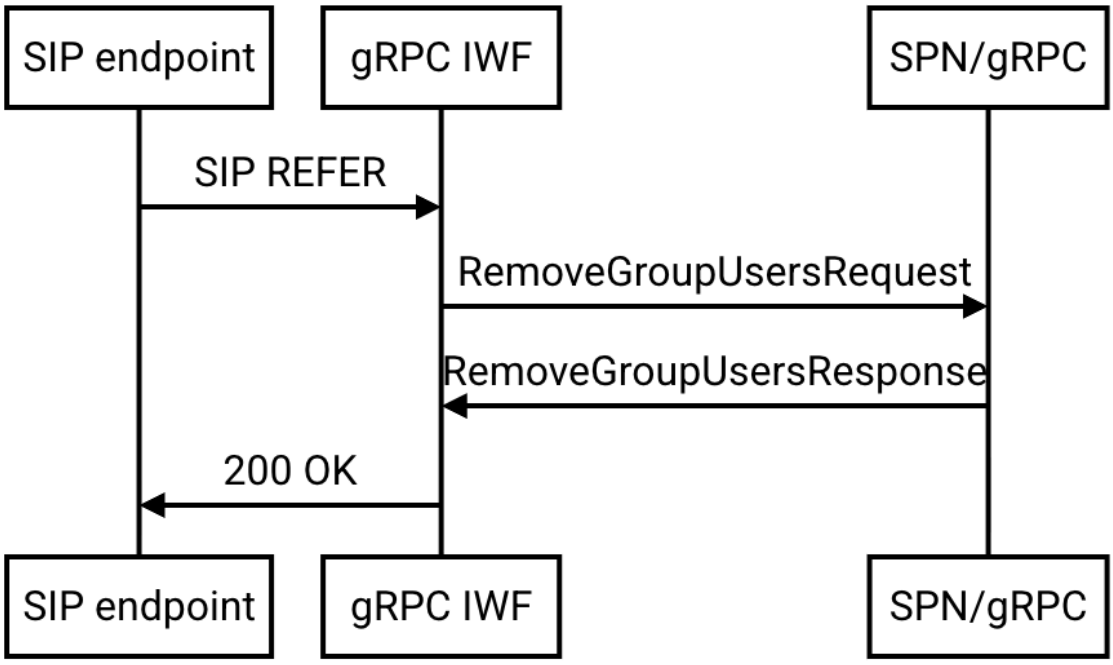

Figure 42: SIP to RemoveGroupUsers gRPC translation

6.5.3.2.6.2 Removing a participant: Notifications

6.5.3.2.6.2.1 Removing a participant: gRPC to SIP

Upon receiving a SendMessage RPC with SendMessageRequest.message.group_change_event.remove_users, to remove participants from the Group Chat as per [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. For each participant being removed:

    a) Send a SIP BYE with a Reason header field as per section 9.2.11 of [GSMA PRD-RCC.11] to each participant listed in remove_users as being removed:
    b) Wait for the SIP 200 OK.

2. If a SIP session is not established between the RCS gRPC IWF and the end-point, send a SIP INVITE as per section 3.2.4.10 of [GSMA PRD-RCC.07]:

a) Construct the From header using the locally stored Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI).
b) Construct the Referred-By header from SendMessageRequest.header.requester_id.
c) Construct the Request-URI and To headers from the SendMessageRequest.receiver_id.
d) Construct the Conversation-ID and Contribution-ID using the SendMessageRequest.message.group_id.
e) Copy relevant SIP headers from SendMessageRequest.header.sip_headers as defined in section 3.1.1.
f) Wait for the SIP 200 OK.

3. For all other remaining users, wait for a SIP SUBSCRIBE and handle as per section 6.5.3.2.1.

If a SIP session is established between the RCS gRPC IWF and the controlling function, the RCS gRPC IWF shall:

4. Construct the `<conference-info>` element:

   a) Construct the `<users>` element with a `<user>` sub-element for each removed participant being added from SendMessageRequest.message.group_change_event.remove_users.removed_users.
   b) Each `<user>` includes the `entity` attribute set to the SendMessageRequest.message.group_change_event.remove_users.removed_users.user and the `<status>` subelement is set to `disconnected`.

5. Send the `<conference-info>` as per section 3.2.4.8 of [GSMA PRD-RCC.07] and [GSMA PRD-RCC.11].

   a) For the SIP NOTIFY mechanism defined in section 3.2.4.8 of [GSMA PRD-RCC.07], copy relevant SIP headers from SendMessageRequest.header.sip_headers as defined in section 3.1.1.

6. Wait for the SIP response:

   a) If status is 200 OK:

      i. Construct and return a SendMessageResponse.

   b) For all other statuses:

      ii. Construct the gRPC error as per section 6.2.1 and 6.2.3.
      iii. Return the gRPC error.

This is illustrated in Figure 43.

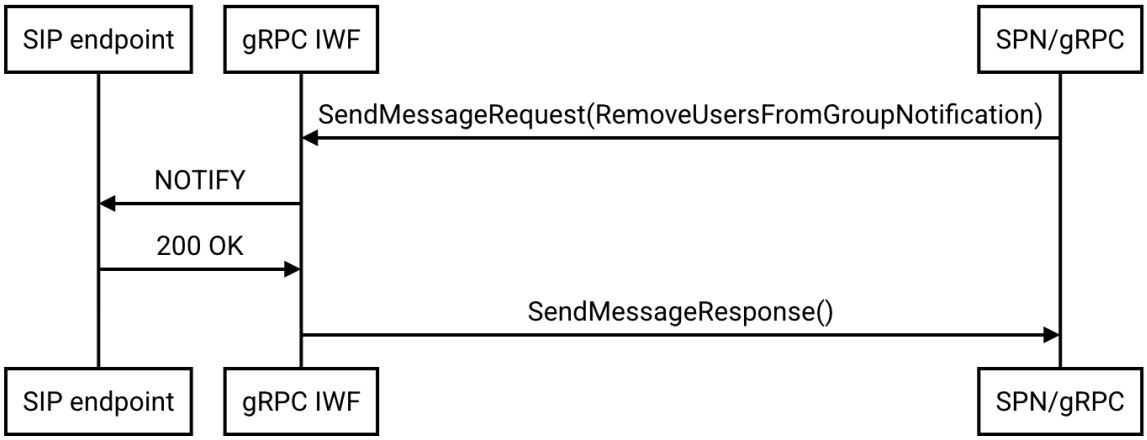

Figure 43: RemoveUsersFromGroupNotification gRPC to SIP translation

6.5.3.2.6.2.2 Removing a participant: SIP to gRPC

Upon receiving a NOTIFY request that removes participants as per [GSMA PRD-RCC.07], to translate to RCS gRPC the RCS gRPC IWF shall:

1. Send a SendMessageRequest:

    a) Construct a RemoveUsersFromGroupNotification:

    i. Construct remove_users using each `<user>` element with an `entity` attribute to set the RcsId and only select that has a `<status>` subattribute set as `disconnected` as per [GSMA PRD-RCC.11] for participants that have been removed from the Group Chat.

    b) Construct a MessagingEvent:

    i. Set the sender_id using the Referred-By header.
    ii. Set the receiver_id using the To header.
    iii. Include the RemoveUsersToGroupNotification as group_change_event.remove_users.

    c) Construct the SendMessageRequest:

    i. Set the header.requester_id using the From.
    ii. Copy relevant SIP headers to the SendMessageRequest.header.sip_headers as defined in section 3.1.1.
    iii. Include the MessagingEvent.

2. Wait for the gRPC response:

    a. If status is OK:

    i. Construct and return SIP 200 OK based on NOTIFY as per [GSMA PRD-RCC.07].

    b) For all other statuses:

i. Construct the SIP error as per section 6.2.2.
ii. Return SIP error response based on NOTIFY as per [GSMA PRD-RCC.07].

This is illustrated in Figure 44.

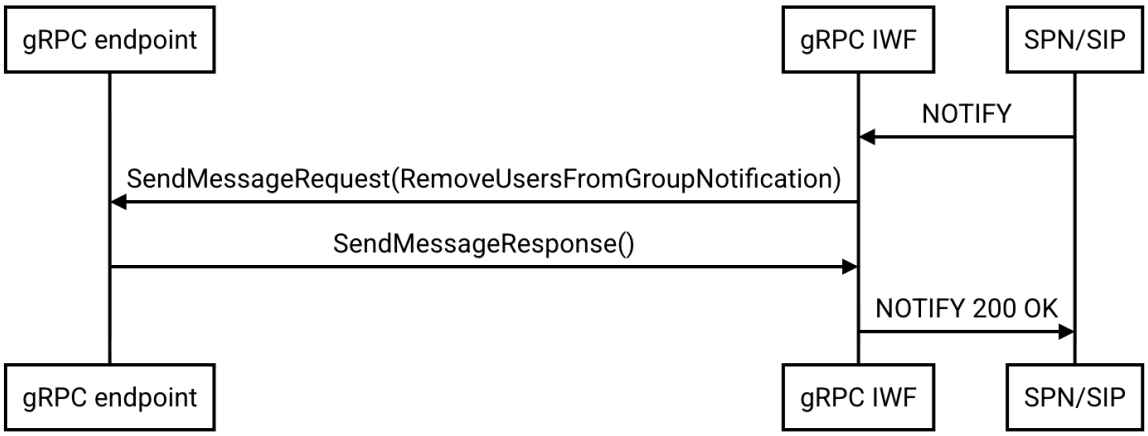

Figure 44: SIP to RemoveUsersFromGroupNotification gRPC translation

6.5.3.2.7 Leaving a Group Chat

6.5.3.2.7.1 Leaving a Group Chat: Request

6.5.3.2.7.1.1 Leaving a Group Chat: gRPC to SIP

Upon receiving a RemoveGroupUsersRequest request to leave a Group Chat, to translate to SIP as per [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. If a SIP session is not established between the RCS gRPC IWF and the controlling function, send a SIP INVITE as per section 3.2.4.10 of [GSMA PRD-RCC.07]:

    a) Construct the From and P-Preferred-Identity headers from the AddGroupUsersRequest.header.requester_id.
    b) Construct the Request-URI and To header from locally stored Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI).
    c) Construct the Conversation-ID and Contribution-ID using the AddGroupUsersRequest.group_id.
    d) Wait for the SIP 200 OK.

Upon receiving the SIP 200 OK and if the requester is the participant being removed from the Group Chat, the RCS gRPC IWF shall:

2. Construct and send a SIP BYE to the Service Provider Network as per [GSMA PRD-RCC.07].

    a) Construct the From and P-Preferred-Identity headers from the RemoveGroupUsersRequest. header.requester_id.
    b) Construct the Request-URI and To headers with the locally stored Group Chat Session Identity [GSMA PRD-RCC.07] (conference URI).

c) Construct the Conversation-ID and Contribution-ID using the RemoveGroupUsersRequest.group_id.
d) Copy relevant SIP headers from RemoveGroupUsersRequest.header.sip_headers as defined in section 3.1.1.

3. Wait for the SIP response.

a) If the status is 200 OK:

i. Construct and return a RemoveGroupUsersResponse.

b) For all other statuses

i. Construct the gRPC error as per section 6.2.1.
ii. Return the gRPC error.

This is illustrated in Figure 45.

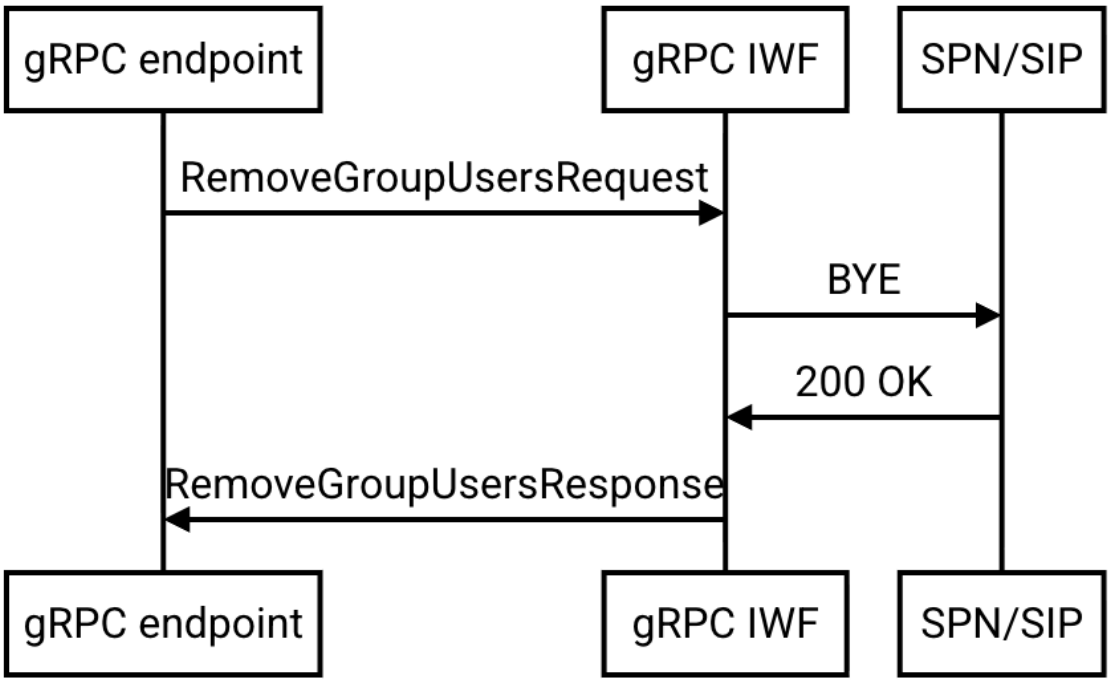

Figure 45: Leave - RemoveGroupUsers gRPC to SIP translation

6.5.3.2.7.1.2 Leaving a Group Chat: SIP to gRPC

Upon receiving a SIP BYE to leave a Group Chat as per [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. Send a RemoveGroupUsers gRPC:

a) Set the RemoveGroupUsersRequest.header.requester_id and RemoveGroupUsersRequest.user_ids using the SIP From header.
b) Set the RemoveGroupUsersRequest.group_id using the Conversation-ID header.

c) Copy relevant SIP BYE headers to the RemoveGroupUsersRequest.header.sip_headers as defined in section 3.1.1.

2. Wait for the gRPC response:

a) If status is OK:

i. Construct and return SIP 200 OK based on SIP BYE as per [GSMA PRD-RCC.07].

b) For all other statuses:

i. Construct the SIP error as per section 6.2.2.
ii. Return SIP error response based on SIP BYE as per [GSMA PRD-RCC.07].

This is illustrated in Figure 46.

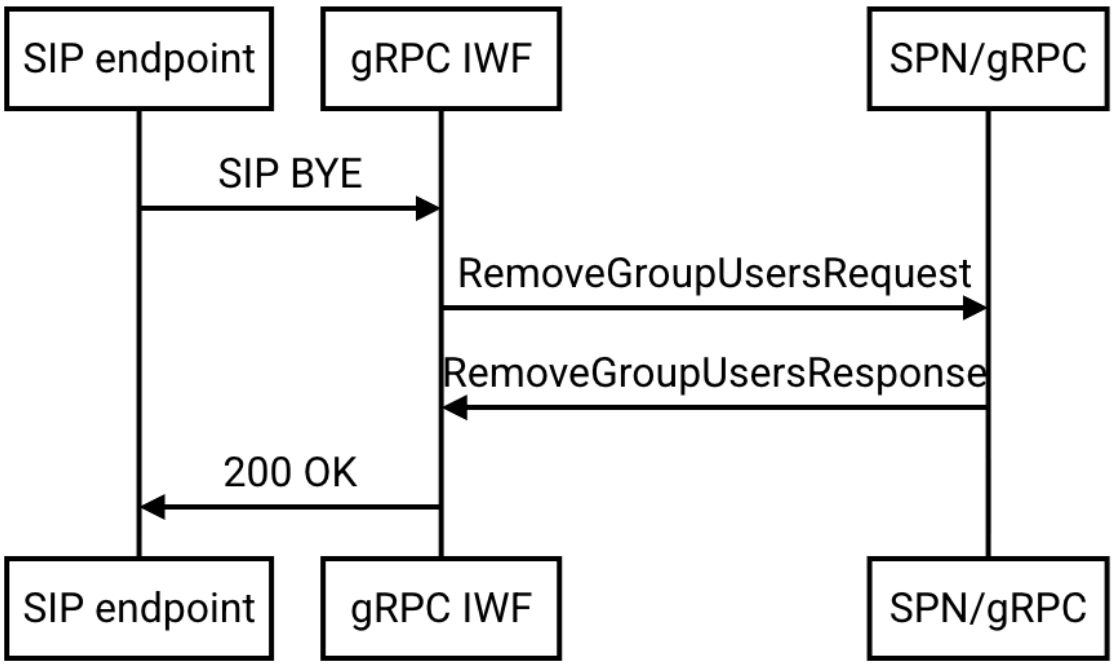

Figure 46: Leave - SIP to RemoveGroupUsers gRPC translation

6.5.3.2.7.2 Leaving a Group Chat: Notifications

6.5.3.2.7.2.1 Leaving a Group Chat: gRPC to SIP

See procedure in section 6.5.3.2.6.2.1.

6.5.3.2.7.2.2 Leaving a Group Chat: SIP to gRPC

See procedure in section 6.5.3.2.6.2.2.

6.5.3.2.8 Group Chat metadata translation

6.5.3.2.8.1 Conference event package

6.5.3.2.8.1.1 Conference event package XML to protocol buffer

When constructing a GroupMetadata from the `<conference-description>` conference event package, the RCS gRPC IWF shall:

1. If `<subject>` subelement is present as per [GSMA PRD-RCC.11], set the GroupMetadata.subject.payload with the subject value encoded as a utf-8 string and set GroupMetadata.subject.content_type to *text/plain; charset=utf-8.*
2. If `<icon-uri>` subelement is present as per [GSMA PRD-RCC.11], set GroupMetadata.icon.icon_uri with the value and set Groupmetadata.icon.content_type to *application/octet-stream.*
3. If `<encrypted-subject>` element is present as per [GSMA PRD-RCC.16], set the GroupMetadata.subject.payload set to a base64 decoding of subject value and set GroupMetadata.subject.content_type to *message/mls-ft.*
4. If `<encrypted-icon>` element is present as per [GSMA PRD-RCC.16], set GroupMetadata.icon.icon_uri with the value and set GroupMetadata.icon.content_type to *message/mls-ft.*

6.5.3.2.8.1.2 Protocol buffer to conference event package XML

When constructing a `<conference-description>` sub element of a conference event package, the RCS gRPC IWF shall:

1. If the content-type of the GroupMetadata.subject is *text/plain* add a `<subject>` subelement as per [GSMA PRD-RCC.11] set from the GroupMetadata.subject.payload parsed using the encoding specified in content-type.
   a. If the encoding is not specified, GroupMetadata.subject.payload is parsed as a utf-8 string.
   b. If GroupMetadata.subject.payload is empty, the `<subject>` shall be set as empty.
2. If the content-type of GroupMetadata.subject is *message/mls-ft* as per section 3.1 add a `<encrypted-subject-description-type>` sub-element with a `<encrypted-subject>` subelement as per section 7.13.2 of [GSMA PRD-RCC.16] set from a base64 encoded GroupMetadata.subject.payload.
3. If present, and if the content-type of GroupMetadata.icon *message/mls-ft* as per section 3.1, add a `<encrypted-icon-description-type>` subelement with a `<encrypted-icon>` subelement as per section 7.13.2 of [GSMA PRD-RCC.16] set from the GroupMetadata.icon.icon_uri.
4. If present, and for all other content-type values of GroupMetadata.icon add a `<icon>` subelement with a `<source>` subelement with a `<icon-uri>` subelement as per [GSMA PRD-RCC.11] set from GroupMetadata.icon.icon_uri.

6.5.3.2.8.2 CPM Group Session Data Management XML

6.5.3.2.8.2.1 CPM Group Session Data Management XML to protocol buffer

When constructing a GroupMetadata from a CPM Group Session Data Management XML `<cpm-group-management>`, the RCS gRPC IWF shall:

1. If `<subject>` subelement is present and the `<action>` element is “set” as per [GSMA PRD-RCC.11], set the GroupMetadata.subject.payload with the subject value encoded as a utf-8 string and set GroupMetadata.subject.content_type to *text/plain; charset=utf-8.*
2. If `<icon>` element is present and the `<action>` element is “set” as per [GSMA PRD-RCC.11], set GroupMetadata.icon.icon_uri with the value and the set GroupMetadata.icon.content_type to *application/octet-stream.*
3. If `<encrypted-subject>` element is present and the `<action>` element is “set” as per section 7.13.1 of [GSMA PRD-RCC.16], set the GroupMetadata.subject.payload set to a base64 decoding of subject value and set GroupMetadata.subject.content_type to *message/mls-ft.*
4. If `<encrypted-icon>` element is present and the `<action>` element is “set” as per section 7.13.1 of [GSMA PRD-RCC.16], set GroupMetadata.icon.icon_uri with the value and the set GroupMetadata.icon.content_type to *message/mls-ft.*

6.5.3.2.8.2.2 Protocol buffer to CPM Group Session Data Management XML

When constructing a `<cpm-group-management>` top element of a CPM Group Session Data Management XML, the RCS gRPC IWF shall:

1. Add a `<group-data>` subelement and a `<request>` subelement with a `target` attribute of the field being changed, add both an `<action>` subelement with a value of “set” and a `<data>` subelement as per [GSMA PRD-RCC.11]:

    a) If the content-type of the GroupMetadata.subject is *text/plain* add a `<subject>` subelement as per [GSMA PRD-RCC.11] set from the GroupMetadata.subject.payload parsed using the encoding specified in content-type.

        i. If the encoding is not specified, GroupMetadata.subject.payload is parsed as a utf-8 string.
        ii. If GroupMetadata.subject.payload is empty, the <subject> shall be set as empty.

    b) If the content-type of GroupMetadata.subject is *message/mls-ft* as per section 3.1 add a `<encrypted-subject>` subelement as per section 7.13.1 of [GSMA PRD-RCC.16] set from a base64 encoded GroupMetadata.subject.payload.
    c) If present, and if the content-type of GroupMetadata.icon *message/mls-ft* as per section 3.1, add a `<encrypted-icon>` subelement as per section 7.13.1 of [GSMA PRD-RCC.16] set from the GroupMetadata.icon.icon_uri.
    d) If present, and for all other content-type values of GroupMetadata.icon add a `<icon>` subelement with a `<icon-uri>` subelement as per [GSMA PRD-RCC.11] set from GroupMetadata.icon.icon_uri.

6.5.3.2.8.3 CPM Group State Object XML

6.5.3.2.8.3.1 CPM Group State Object XML to protocol buffer

When constructing a GroupMetadata from the `<groupstate>` top element of a CPM Group State Object XML, the RCS gRPC IWF shall:

1. If `<subject>` subelement is present as per [GSMA PRD-RCC.11], set the GroupMetadata.subject.payload with the subject value encoded as a uft-8 string and set GroupMetadata.subject.content_type to *text/plain; charset=utf-8.*

2. If `<icon-uri>` subelement is present as per [GSMA PRD-RCC.11], set GroupMetadata.icon.icon_uri with the value and the set GroupMetadata.icon.content_type to *application/octet-stream.*

6.5.3.2.8.3.2 Protocol buffer to Group State Object XML

When constructing a `<groupstate>` top element of a CPM Group State Object XML, the RCS gRPC IWF shall:

1. If the content-type of the GroupMetadata.subject is *text/plain* add a `<subject>` subelement as per [GSMA PRD-RCC.11] set from the GroupMetadata.subject.payload parsed using the encoding specified in content-type.
   a. If the encoding is not specified, GroupMetadata.subject.payload is parsed as a utf-8 string.
   b. If GroupMetadata.subject.payload is empty the <subject> shall be set as empty.
2. If present, and for all other content-type values of GroupMetadata.icon add a `<icon>` subelement with a `<icon-uri>` subelement as per [GSMA PRD-RCC.11] set from GroupMetadata.icon.icon_uri.

6.6 RCS Video Chat Event Reporting

6.6.1 RCS Video Chat Event Reporting: gRPC to SIP

Upon receiving a VideoChatEventNotificationRequest, the RCS gRPC IWF shall:

1. Construct and send a SIP MESSAGE with:

   a. The From header shall be constructed from the VideoChatEventNotificationRequest.header.requester_id.
   b. The the Request-URI and To headers shall be constructed from the MIVC REPORTING URI as per section 3.7.5.2.5 of [GSMA PRD-RCC.07].
   c. Copy relevant SIP headers from VideoChatEventNotificationRequest.request_header.sip_headers as defined in section 3.1.1.
   d. The body containing the VideoChatEventNotificationRequest.VideoChatEvent protocol buffer as per section 3.7.5.2.5 of [GSMA PRD-RCC.07].

2. Wait for the SIP response:
   a. If status is 200 OK:
      i. Construct and return VideoChatEventNotificationResponse

   b. For all other statuses:

ii. Map to gRPC error as per section 6.2.1.
iii. Return the gRPC error.

6.6.2 RCS Video Chat Event Reporting: SIP to gRPC

Upon receiving a SIP MESSAGE containing a VideoChatEvent protocol buffer as per section 3.7.5.2.5 of [GSMA PRD-RCC.07], the RCS gRPC IWF shall:

1. Send a SendVideoChatEventNotification gRPC:
   a. Set the VideoChatEventNotificationRequest.sender_id and the VideoChatEventNotificationRequest.header.requester_id using the SIP From header.
   b. Copy relevant SIP MESSAGE headers to the VideoChatEventNotificationRequest.header.sip_headers as defined in section 3.1.1.

2. Wait for the gRPC response:
   a. If status is OK:
      iv. Construct and return SIP 200 OK as per [[GSMA PRD-RCC.07]].

   b. For all other statuses:
      v. Construct the SIP error as per section 6.2.2.
      vi. Return SIP error response as per [[GSMA PRD-RCC.07]].

Annex A Managed Objects and Configuration Parameters

A.1 Management objects parameters overview

A.1.1 RCS gRPC Parameters

If the SPN is configuring the RCS client for RCS gRPC, it shall include the parameters in this section but not the parameters defined in A.1.6.1 and A.1.6.2 in [GSMA PRD-RCC.07]. If instead the SPN is configured for SIP, the configuration document shall include the parameters defined in A.1.6.1 and A.1.6.2 in [GSMA PRD-RCC.07] but not the parameters in this section.

Configuration parameter	Description	RCS usage
RCS GRPC ENDPOINT	Represents the endpoint for RCS clients to connect to for all RCS gRPCs.	Mandatory parameter for RCS gRPC
RCS GRPC TOKEN	Auth token used to authenticate the RCS client to the SPN. Shall be included in all RCS gRPC requests. The auth token is the initial token and it is identical in use to the auth token returned by the AuthTokenRefresh gRPC call defined in 4.3.1.	Mandatory parameter for RCS gRPC
RCS GRPC TOKEN REFRESH	Interval, in seconds, for which the RCS client must refresh the auth token	Mandatory parameter for RCS gRPC

Configuration parameter	Description	RCS usage
Timer_T1	Defines the timer T1 – the RTT estimate	Mandatory parameter

Table 9: RCS gRPC Configuration Parameters

A.2 Provisioning Document of the RCS Management tree

A.2.1 RCS gRPC sub tree

This RCS specification includes the following additions as a new configuration sub tree, called the RCS gRPC sub tree. Please note this sub tree is not included in any other specifications. So, no other nodes from those specifications need to be added:

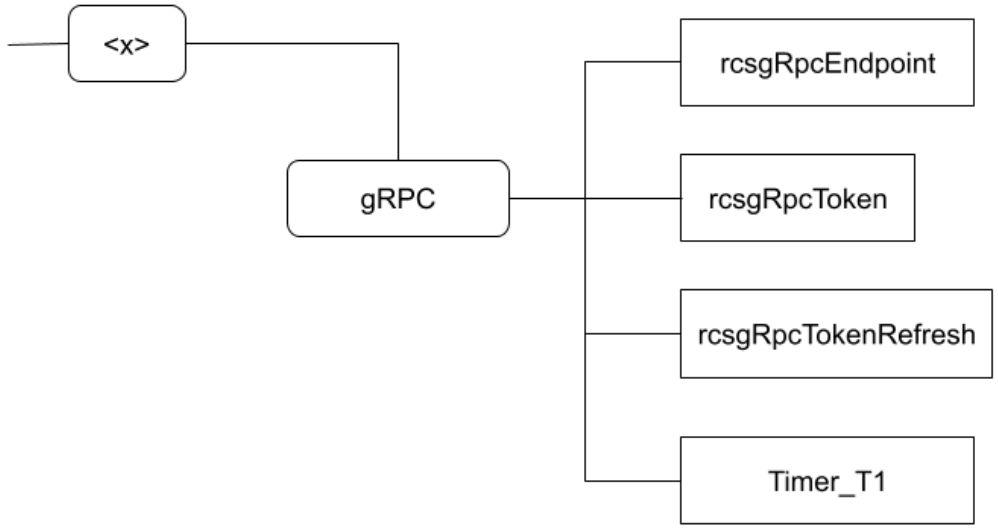

Figure 47: RCS additions, RCS gRPC sub tree

The associated HTTP configuration XML structure is presented in the Table 10 below:

```
<characteristic type="gRPC">
        <parm name="rcsgRpcEndpoint" value="X"/>
        <parm name="rcsgRpcToken" value="X"/>
        <parm name="rcsgRpcTokenRefresh" value="X"/>
        <parm name="Timer_T1" value="X"/>
</characteristic>
```

Table 10: RCS gRCP sub tree associated HTTP configuration XML structure

Node: /<X>/gRPC

Under this interior node the RCS parameters related to Service Provider specific extensions are placed.

Status	Occurrence	Format	Min. Access Types
Optional	ZeroOrOne	node	Get

Table 11: Service Provider Extensions MO sub tree addition node

Node: /<x>/gRPC/rcsgRpcEndpoint

Leaf node that indicates the FQDN of the RCS gRPC endpoint

Status	Occurrence	Format	Min. Access Types
Required	One	String	Get, Replace

Table 12: gRPC sub tree addition parameters (rcsgRpcEndpoint)

- Values: FQDN of the gRPC endpoint
- Post-reconfiguration actions: no additional action at the time of re-configuration.
- Associated HTTP XML parameter ID: “rcsgRpcEndpoint”

Node: /<x>/gRPC/rcsgRpcToken

Leaf node that indicates the auth token for RCS gRPC

Status	Occurrence	Format	Min. Access Types
Required	One	String	Get, Replace

Table 13: gRPC sub tree addition parameters (rcsgRpcTokenRefresh)

- Values: interval, in seconds, for when the client must refresh the auth token.
- Post-reconfiguration actions: no additional action at the time of re-configuration.
- Associated HTTP XML parameter ID: “rcsgRpcTokenRefresh”

Node: /<x>/gRPC/Timer_T1

Leaf node that indicates the timer T1 – the RTT estimate

Status	Occurrence	Format	Min. Access Types
Required	One	int	Get, Replace

Table 14: gRPC sub tree addition parameters (Timer_T1)

- Values: interval, in milliseconds, for Timer_T1.
- Post-reconfiguration actions: no additional action at the time of re-configuration.
- Associated HTTP XML parameter ID: “Timer_T1”

Annex B Document Management

B.1 Document History

Version	Date	Brief Description of Change	Approval Authority	Editor / Company
1.0	17 Jul 2026	initial version	ISAG	JP Cormier / Google

B.2 Other Information

Type	Description
Document Owner	RCS Group
Editor / Company	JP Cormier / Google

It is our intention to provide a quality product for your use. If you find any errors or omissions, please contact us with your comments. You may notify us at prd@gsma.com

Your comments or suggestions & questions are always welcome.