

joyn Terminal API Specification

Version 0.x

19 March 2013

This is a draft of a **Non-binding Permanent Reference Document** of the GSMA

Security Classification: Non-confidential

Access to and distribution of this document is restricted to the persons permitted by the security classification. This document is confidential to the Association and is subject to copyright protection. This document is to be used only for the purposes for which it has been supplied and information contained in it must not be disclosed or in any other way made available, in whole or in part, to persons other than those permitted under the security classification without the prior written approval of the Association.

Copyright Notice

Copyright © 2013 GSM Association

Disclaimer

The GSM Association (“Association”) makes no representation, warranty or undertaking (express or implied) with respect to and does not accept any responsibility for, and hereby disclaims liability for the accuracy or completeness or timeliness of the information contained in this document. The information contained in this document may be subject to change without prior notice.

Antitrust Notice

The information contain herein is in full compliance with the GSM Association’s antitrust compliance policy.

Table of Contents

[1 Introduction 4](#_Toc357525133)

[1.1 Overview 4](#_Toc357525134)

[1.2 Scope 4](#_Toc357525135)

[1.3 Definitions 4](#_Toc357525136)

[1.4 Abbreviations 4](#_Toc357525137)

[1.5 References 4](#_Toc357525138)

[1.6 Conventions 5](#_Toc357525139)

[2 API Architecture 5](#_Toc357525140)

[2.1 Architecture Overview 5](#_Toc357525141)

[**Fig.1 General Architecture Overview** 6](#_Toc357525142)

[2.1.1 API Descriptions 6](#_Toc357525143)

[2.1.2 Applications Types 7](#_Toc357525144)

[3 API concepts 8](#_Toc357525145)

[**3.1** **Servers and Listeners** 8](#_Toc357525146)

[3.1.1 Service 9](#_Toc357525147)

[3.1.2 Service Session 9](#_Toc357525148)

[**3.2** **Service Version/Available/Unavailable** 9](#_Toc357525149)

[4 Android API 9](#_Toc357525150)

[4.1 Components Interaction 9](#_Toc357525151)

[4.1.1 New service application 9](#_Toc357525152)

[4.1.2 Constraints 10](#_Toc357525153)

[4.2 Security 10](#_Toc357525154)

[4.3 UX API 11](#_Toc357525155)

[4.3.1 Package 11](#_Toc357525156)

[4.3.2 Methods and Callbacks 11](#_Toc357525157)

[4.3.3 Intents 11](#_Toc357525158)

[4.4 Services API 13](#_Toc357525159)

[4.4.1 Overview 13](#_Toc357525160)

[4.4.2 Access Control 13](#_Toc357525161)

[4.4.3 Common architecture 13](#_Toc357525162)

[4.4.4 Capability API 16](#_Toc357525163)

[4.4.5 IM/Chat API 21](#_Toc357525164)

[4.4.6 File Transfer API 35](#_Toc357525165)

[4.4.7 Image Share API 41](#_Toc357525166)

[4.4.8 Video Share API 47](#_Toc357525167)

[4.4.9 Contacts API 53](#_Toc357525168)

[4.4.10 API Versioning 55](#_Toc357525169)

[4.4.11 Utility API 56](#_Toc357525170)

[4.5 Privileged Client API 57](#_Toc357525171)

[4.5.1 Overview 57](#_Toc357525172)

[4.6 IMS APIs 57](#_Toc357525173)

[4.6.1 Overview 57](#_Toc357525174)

[Annex A Document Management 57](#_Toc357525175)

[A.1 Document History 57](#_Toc357525176)

[A.2 Other Information 57](#_Toc357525177)

# Introduction

## Overview

This document defines the architecture and a set of standardized Application Programming Interfaces (API) to develop Joyn UX, use Joyn services and develop IMS-based services.

## Scope

The scope of this document covers the APIs along with security limitations for the functionalities defined in [PRD RCC.50].

## Definitions

| Term | Description |
| --- | --- |
| 3rd Party Applications | Applications that are not part of the Joyn Client and developed by companies or individuals other than MNOs and OEMs. |
| Core Applications | Applications that are part of the Joyn Client. |
| Trusted Applications | Applications using the IMS API, developed by trusted parties (MNOs and OEMs). |
| IMS Stack | Component responsible for implementing IMS protocol suite and core services. |
| Joyn Client | Complete software package that passed Joyn accreditation. |
| Service API | APIs that expose Standard Services and can be used in multiple instances without any restrictions. |
| Privileged Client API | API shall expose key functionalities which are necessary for the proper working of the Joyn client. |
| IMS API | APIs that are exposed by the IMS Stack. |
| Standard Services | Services that are identified by feature tags, as defined by Joyn Specification. |

## Abbreviations

| Term | Description |
| --- | --- |
|  | <This is an optional section giving a list of the abbreviations necessary for the understanding of the GSMA document. This list shall contain all abbreviations and their corresponding full terms which are used within the document, in alphabetical order. If there are no entries needed in an abbreviations table, this section may be deleted> |
| <e.g. PRD> | Permanent Reference Document |

## References

| Ref | Doc Number | Title |
| --- | --- | --- |
|  | [PRD RCC.50] | GSMA RCS-e - Advanced Communications: Services and Client Specification version 1.2.2  <http://www.gsma.com/rcs/wp-content/uploads/2012/03/rcs-e_advanced_comms_specification_v1_2_2_approved.pdf> |
|  | RFC 2119 | “Key words for use in RFCs to Indicate Requirement Levels”, S. Bradner, March 1997. Available at <http://www.ietf.org/rfc/rfc2119.txt> |
|  | [RCS-e UI Connector] | GSMA RCS-e UI connector Guidelines (RCS UIC\_001) 1.0  <http://www.gsma.com/>  [Editor’s Note: To be updated with permanent Doc Number and Title] |

## Conventions

“The key words “must”, “must not”, “required”, “shall”, “shall not”, “should”, “should not”, “recommended”, “may”, and “optional” in this document are to be interpreted as described in RFC2119 [2].”

# API Architecture

## Architecture Overview

The Joyn Client architecture is composed of several subsystems, organised into functional layers as shown in the diagram below.

The fundamental enabling component is the **IMS Stack** which contains the protocol suite (SIP, MSRP, RTP/RTCP, HTTP etc.) and core services (IMS session Management, Media management, Registration etc). The functionality of this component is governed by the IMS specifications.

Above IMS there are the **RCS Enablers**, comprising the functionality to enable RCS-based Chat, Video and Image sharing, File Transfer and other RCS services. The functionality of this layer is governed by the GSMA RCS specifications.

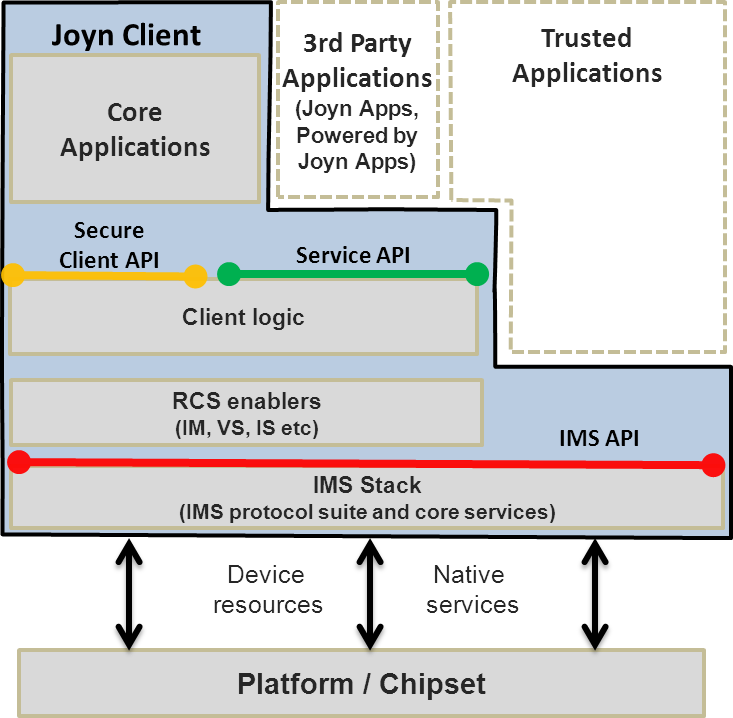
Access to these functional layers is mediated by the **Client logic**, which provides the client API and the functionality to mediate access to that API by multiple clients. Client applications and services access the underlying functionality exclusively through this interface. The client logic exposes two levels of client API:

* the **Service API**, which is the principal API providing access for client applications to the RCS services (OMA IM, GSMA Video Share, GSMA Image Share etc).
* the **Privileged Client API**, which provides access to particularly sensitive parts of the stack.

The Joyn **Core Applications** are the (typically embedded) applications that provide the end-user’s access to RCS services. The Core Applications also expose a **UX API** (not shown on the figure) whereby other applications can programmatically invoke operations that are interactively fulfilled by the Core Applications. Core Applications make use of both the Service API and the Privileged Client API.

The architecture is intended to enable **Third Party Applications** to make direct use also of the Service API, enabling programmatic access to the RCS services. The Service API is scoped so as to make access by Third Party Applications possible subject to those applications having the appropriate permission.

A special distinction is made for independent applications, labelled here as **Trusted Applications** that provide services directly on top of the IMS Stack (and which may themselves define their own Service APIs).



**Fig.1 General Architecture Overview**

### API Descriptions

Architecturally, the four API types can be classed as exposing Joyn functionality (UX API, Service API and Privileged Client API) and APIs exposing IMS stack functionality ( IMS API ).

#### UX APIs

UX APIs enable other installed applications to interface or link the native RCS service.

#### Service APIs

Client APIs shall expose the functionality of all Standard Services to be used in full by 3

Service APIs provide a functional interface to the RCS enablers, enabling the Core Applications and Third Party Applications to interoperate with other RCS devices whilst relying on the stack to ensure conformance to the RCS-e Specification.

Those functionlaties are:

* IM/Chat service
* Image Share
* Video Share

Note: For VSH and ISH functionality to be fully available, a call needs to be ongoing with a RCS contact posessing VSH and ISH capabilities.

* File Transfer
* Client Connector
* Capabilities

The Standard Services exposed in this layer also provide variant method calls that allow service-over-service implementation.

The APIs in this layer also expose common Joyn functionality for capability fetching and retrieving parts of the Joyn network configuration required for UI elements.

In case of Android OS, client connectors as specified in [RCS-e UI Connector], enables to interface the native RCS service functionality with 3rd party applications on the device.

#### Privileged Client APIs

The APIs on this level expose sensitive internal functionality of the Joyn stack and, accordingly, access to these APIs cannot be granted by a user to any application. The sensitive aspects include:

The Privileged APIs will be defined in the future version.

#### 2.1.1.4 IMS APIs

These APIs allow using a subset of IMS functionality to create new services that work along Standard Services.

The IMS APIs will be defined in the future version.

### Applications Types

Applications types can be divided into three broad categories:

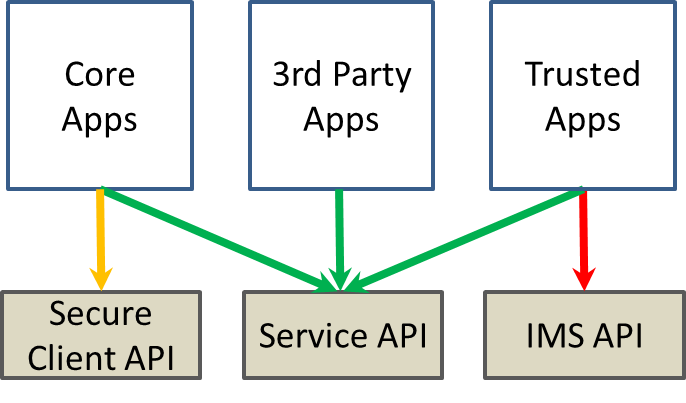


Fig.2 Mapping of target component types to APIs based on security

* Third Party Applications - All Third Party Apps can use only Service APIs. Some Third Party Application may provide service-over-service functionality. These applications use additional parameterization defined in the Service APIs, but have their own feature tags not defined by the Joyn Specification.
* Core Applications - Components that are included in the Joyn Client. These components must undergo GSMA accreditation as part of the Joyn Client. Core Applications are the only ones that access Privileged Client APIs. Core Applications may use Service APIs (such as instant messaging) and can have overlapping functionality with Third Party Applications.
* Trusted Applications - These applications can use both IMS APIs and Service APIs. They can also define new services that are not part of the Joyn Specification and need have their own feature tag.

# API concepts

**3.1 Servers and Listeners**

RCS APIs are provided with a client/server model. At any time, for a service, there may be zero or more clients. At any time, a client may be connected to zero or more services.

Prior to requesting a service, a client connects to that service.

Servers provide RCS services to the clients and notify the registered clients with the events through listeners. Clients request RCS services from the servers by invoking the appropriate API(s). Servers notify clients of RCS events by invoking the appropriate listener (callback functions). For RCS events that a client is required to monitor, the client must supply the listener to the server.

For each service, this document describes all the server APIs. This document also describes the set of events that are available for each service.

### Service

Prior to using a service, a RCS client invokes the appropriate API to create the service.. At this time, the client can also register for events by supplying the appropriate listener functions

After the service is created, the service communicates/notifies its clients about the service availability and/or service-specific functionality changes through the listener supplied by the clients.

At any time, a service may have zero or more sessions associated with it.

When a service is no longer needed, the client can destroy the service by invoking the appropriate API for that service. When a service is destroyed, all the sessions associated with that service are also terminated.

### Service Session

A service session is established based on external triggers, e.g. user attempting to establish a call or upon receipt of an event from the RCS service about a request from a remote user. When a service session is to be established, the appropriate API is invoked. At the time of establishment of a service session, the client registers for events by supplying appropriate listener functions. Each service session is associated with a RCS service.

After the service session is created, the RCS service communicates/notifies its clients about the session state through the listeners supplied by the clients.

At any time, the client can terminate a service session by invoking the appropriate API, for example, based on user action or based on events from the RCS server that indicate a change in the session state.

**3.2 Service Version/Available/Unavailable**

Each service is associated with a specific client. Services are designed to allow each service to have its own service version and its availability/unavailability attribute independently. Each service follows the same template API to provide versioning information and has the same type of listener functions through which the service informs its clients about specific service status.

# Android API

## Components Interaction

Each of the Terminal APIs for Android defines their interaction individually and how they can be used by an Android Application.

### New service application

When an Android application wants to define a new service, it needs to add its feature tag as meta-data value in its Android Application Manifest. The RCS-e Service Tag also needs to be accompanied by the feature tag. Refer to RCS-e Specification 1.2.2 for exact definitions of possible feature tags.

* The RCS-e meta-data name is defined by the string *“rcs service”*
* Example of usage:

<meta-data android:value=”[feature tag of the service]” android:name=”rcs service”></meta-data>

[Extensible as required]

### Constraints

1. Only a single Joyn Client can be active on a device. This is defined by *ID\_3\_1\_1* of RCS-e Implementation guidelines v3.2. This constraint limits the possibilities for deployment of additional Joyn Clients with the Terminal API, as they cannot replace the package of a previously installed client on the device. This constraint could be avoided if the Terminal APIs could be retrieved dynamically instead of static package reference.
2. When multiple applications are present, that support the same type of service notifications, multiple notification may be placed in the Notification Tray, if each application handles the broadcasted intent.
3. Trusted application can only run if any IMS Client is running. This means trusted applications can only be dynamically registered/de-registered. They are not allowed to be part of initial registration application set. Any exceptions need to be carefully considered**.**

## Security

Most of the Joyn APIs provide access to sensitive functionality, either because they enable access to privacy-sensitive information or because they can cause charges to be incurred for network and service usage. In addition, certain APIs expose the internal functionality of the stack, and abuse of those APIs could compromise the integrity of the stack or the Joyn services.

#### 4.2.1.1 Service API Access Control

The Service API comprises a number of specific services (IM, File Transfer, etc) and access to each of these is mediated by one or more permissions. The Service APIs are designed so that they may be used by Third Party Applications. In each case, the API exposes privacy-sensitive information or may trigger service charges, and the user must grant permission for any app to use that API. The permissions are defined for each service API in the sections that follow.

#### 4.2.1.2 Privileged Client API Access Control

The Privileged Client APIs are sensitive and their abuse could compromise the integrity of the stack or the Joyn services, Access is therefore restricted so that they may only be used by authorised applications such as the Core Applications that are part of the Joyn Client. Any component requiring access to the Privileged Client API must undergo review and approval in the same manner as the other elements of the Joyn Client. Procedures for obtaining access to these APIs are beyond the scope of this document.

#### 4.2.1.3 IMS API Access Control

Since the IMS API provides low-level access to the IMS stack, there is significant scope for abuse including generation of excessive network traffic, interception of requests intended for other applications, and other abuses resulting in denial of service.

Access to the IMS API is limited only to trusted components provided by MNO and/or OEMs. Procedures for obtaining access to these APIs are beyond the scope of this document.

## UX API

This API offers:

* Intents which permit to link joyn applications with other third party applications installed on the device. This overrides the document “GSMA RCS-e UI connector Guidelines (RCS UIC\_001)”.
* Methods to discover existing joyn services on the device and their activation states. This overrides the document “GSMA RCS IOT RCS-e Implementation Guidelines”.

### Package

Package name **org.gsma.joyn**

### Methods and Callbacks

**Class JoynUtils:**

**Method: returns the list of joyn clients installed on the device (except myself). An application is identified as a joyn client by including an intent filter with the ACTION\_CLIENT\_SETTINGS action in the Manifest.xml of the application.**

static List<ApplicationInfo> getJoynClients(Context context)

**Method: sends a broadcast intent and registers a listener that is triggered when all joyn clients have cleared their preferred status. Each joyn client must register a receiver that listens for the broadcast intent action ACTION\_CLIENT\_CLEAR\_PREFERRED. After a client receives the broadcast, it must clear its preferred status for the activity with the “org.gsma.joyn.client” action.**

static void clearPreferredIntent(Context context, BroadcastReceiver receiver);

**Method: sends a broadcast intent and registers a listener that is triggered when all joyn clients have been disabled. Each joyn client must register a receiver that listens for the broadcast intent action ACTION\_CLIENT\_DISABLE. After a client receives the broadcast, it must disable itself.**

static void disableClients(Context context, BroadcastReceiver receiver);

### Intents

**Class Intents.Chat:**

This class offers Intents to link applications to joyn applications for chat services.

**Intent: load the chat application to view a chat conversation. This Intent takes into parameter an URI on the chat conversation (i.e. content://chats/chat\_ID). If no parameter found the main entry of the chat application is displayed.**

static final String ACTION\_VIEW\_CHAT = "org.gsma.joyn.VIEW\_CHAT"

**Intent: load the chat application to start a new conversation with a given contact. This Intent takes into parameter a contact URI (i.e. content://contacts/people/contact\_ID). If no parameter the main entry of the chat application is displayed.**

static final String ACTION\_INITIATE\_CHAT = "org.gsma.joyn.INITIATE\_CHAT"

**Intent: load the group chat application. This Intent takes into parameter an URI on the group chat conversation (i.e. content://chats/chat\_ID). If no parameter found the main entry of the group chat application is displayed.**

static final String ACTION\_VIEW\_CHAT\_GROUP = "org.gsma.joyn.VIEW\_GROUP\_CHAT"

**Intent: load the group chat application to start a new conversation with a group of contacts. This Intent takes into parameter a list of contact URIs. If no parameter the main entry of the group chat application is displayed.**

static final String ACTION\_INITIATE\_CHAT\_GROUP = "org.gsma.joyn.INITIATE\_CHAT\_GROUP"

**Class Intents.Client:**

This class offers intents to link client applications.

**Intent: load the settings activity to enable or disable the client.**

static final String ACTION\_CLIENT\_SETTINGS = “org.gsma.joyn.client.settings”;

**Intent: action for broadcast intent that clears preferred client status. When a joyn client receives an intent with this action, it must clear its preferred status.**

static final String ACTION\_CLIENT\_CLEAR\_PREFERRED = “org.gsma.joyn.client.CLEAR\_PREFERRED”;

**Intent: action for a broadcast intent that disables joyn clients. When a joyn client receives an intent with this action, it must disable itself.**

static final String ACTION\_CLIENT\_DISABLE = “org.gsma.joyn.client.DISABLE”;

**Class Intents.FileTransfer:**

This class offers Intents to link applications to joyn applications for file transfer services.

**Intent: load the file transfer application to view a file transfer. This Intent takes into parameter an URI on the file transfer (i.e. content://filetransfers/ft\_ID). If no parameter found the main entry of the file transfer application is displayed.**

static final String ACTION\_VIEW\_FT = "org.gsma.joyn.VIEW\_FT"

**Intent: load the file transfer application to start a new file transfer to a given contact. This Intent takes into parameter a contact URI (i.e. content://contacts/people/contact\_ID). If no parameter the main entry of the file transfer application is displayed.**

static final String ACTION\_INITIATE\_FT = "org.gsma.joyn.INITIATE\_FT"

Note: for Intents using a contact URI as a parameter, if the contact has several phone numbers which are joyn compliant, then the application receiving the Intent should request to the user to select which phone number should be used by the service.

Note: sharing during a call (image & video) are part of the native dialer application and may be only visible when a call is established, in this case there is no public Intent to initiate a sharing.

## Services API

### Overview

This section contains all the Service APIs. Each of the presented APIs may have a Core Application using it, but a separate 3rd Party Application can also use it. Each API exposes all its functionality on a high level and does put constraints on the invoking application as to the preconditions and order of method calls. All Service APIs are stateless, meaning that any part of the API can be used without first satisfying any preconditions.

### Access Control

Each of the services requires one or more permissions to be held by the calling application; the permissions associated by each service are defined in the sections that follow.

The permissions are organised on a service-by-service basis and at a sufficiently fine-grained level – eg the ability to read contact details from the addressbook - that the user can make a meaningful choice when confronted with a request at the install prompt. The user is not asked to give blanket approval to a very broad permission such as the ability to read any user data.

### Common architecture

The joyn terminal API contains the following service API:

* Capability service API,
* Chat API,
* File Transfer API,
* Video Share service API,
* Image Share service API.

Each service API is based on a Client/Server model using the AIDL Android interface to communicate between the application using the service and the RCS service or stack implementing the service. So many applications can connect in parallel to the core RCS service.

#### Package

Package name **org.gsma.joyn**

#### Methods and Callbacks

**Class JoynService:**

Each service API should extends the abstract class JoynService.

**Constructor: instanciate a service API. This method takes in parameter a service event listener which permits to monitor the connection to the RCS service. The parameter context is an Android context which permits to initiate the binding with the corresponding service.**

JoynService(Context ctx, JoynServiceListener listener)

**Method: connects to the API. This method permits to bind to the service.**

void connect()

**Method: disconnects from the API. This method permits to unbind from the service.**

void disconnect()

**Method: returns “true” if connected to the service, else returns “false”.**

boolean isServiceConnected()

**Method: returns “true” if service registered to the RCS service platform, else returns “false”.**

boolean isServiceRegistered()

**Class JoynServiceListener:**

**Method: callback called when service is connected. This method is called when the service is well connected to the RCS service (binding procedure successfull): this means the methods of the API may be used.**

void onServiceConnected()

**Method: callback called when service has been disconnected. This method is called when the service is disconnected from the RCS service (e.g. service deactivated). The reason code may have the following values: CONNECTION\_LOST, SERVICE\_DISABLED, INTERNAL\_ERROR.**

void onServiceDisconnected(int reason)

**Method: callback called when service is registered to the RCS platform. This method is called when the terminal is registered to the RCS/IMS service platform.**

void onServiceRegistered()

**Method: callback called when service is unregistered from RCS platform. This method is called when the terminal is not more registered to the RCS/IMS service platform.**

void onServiceUnregistered()

**Class JoynServiceConfiguration:**

This class represents the particular configuration of joyn Service.

**Method: returns True if the joyn service is activated, else returns False. The service may be activated or deactivated by the end user via the joyn settings application.**

boolean isServiceActivated()

**Method: returns the display name associated to the joyn user account. The display name may be updated by the end user via the joyn settings application.**

String getUserDisplayName()

#### Exceptions

**Class JoynServiceException:**

This class is used to propagate generic service API exception.

**Class JoynServiceNotBoundException:**

This class is thrown when a method of the service API is called and the service API is not bound to the RCS service (e.g not yet connected).

**Class JoynServiceNotRegisteredException:**

This class is thrown when a method of the service API using the RCS service platform is called and the terminal is not registered to the RCS service platform (e.g. not yet registered).

#### Permissions

Access to the Services API is requires the org.gsma.joyn.READ\_RCS\_STATE permission. This is a new permission, analogous to READ\_PHONE\_STATE, covering general access to the RCS stack state.

This permission is additionally required to access any of the specific services, since use of those services implicitly reveals information about the current network and stack state

### Capability API

This API allows for querying the capabilities of a user or users and checking for changes in their capabilities:

* Read the supported capabilities locally by the user on its device.
* Retrieve all capabilities of a user.
* Checking a specific capability of a user.
* Refresh capabilities for all contacts.
* Registering for changes to a user/users ‘s capabilities
* Unregistering for changes to a user/users ‘s capabilities
* Define scheme for registering new service capabilities based on manifest defined feature tags.

This API may be accessible by any application (third party, MNO, OEM). The RCS extensions are controlled internally by the RCS service.

#### Capability Discovery API calling flow

The Capabilities Discovery (CD) service provides the API through which the user can get the capabilities of other contacts and also "announce" its own capabilities.

The figures in this section contain basic call flows of the CD service API.

Figure x.x-1 is an example that shows the retrieval of the capabilities of a list of remote contacts.



**Figure x.x-1: Get the capabilities of a list of remote contacts**

1. The RCS client instantiates a service instance of the Capability Discovery Service. At this time, it also specifies the list of listener functions.
2. The RCS client establishes a connection with the Capability Discovery Service. The Capability Discovery Service associates the listener with this RCS client.
3. The RCS client constructs a list of contacts for which it wants to get the latest capabilities. It invokes the API to get the capabilities of these contacts by providing the contact list as argument. The Capability Discovery Service returns the requested information from the local database.
4. Additionally, the Capability Discovery Service initiates procedures with the remote parties to retrieve the latest capabilities.
5. When the updated capability information is available for a contact, the listener function(s) are invoked to inform all the RCS clients that have installed a listener. This step is repeated for each contact for which updated capability information becomes available.
6. Finally, the RCS client, having retrieved the contact information, disconnects from the capability discovery service. At this time, the Capability Service discards all listeners associated with this client.

#### Package

Package name org.gsma.joyn.capability

#### Methods and Callbacks

**Class CapabilityService:**

This class offers the main entry point to the Capability service which permits to read capabilities of remote contacts, to initiate capability discovery and to receive capabilities updates. Several applications may connect/disconnect to the API.

A set of capabilities is associated to each MSISDN of a contact.

**Method: connects to the API.**

void connect()

**Method: disconnects from the API.**

void disconnect()

**Method: returns the capabilities supported by the local end user. The supported capabilities are fixed by the MNO and read during the provisioning.**

Capabilities getMyCapabilities()

**Method: returns the capabilities of a given contact from the local database. This method doesn’t request any network update to the remote contact. The parameter contact supports the following formats: MSISDN in national or international format, SIP address, SIP-URI or Tel-URI.**

Capabilities getContactCapabilities(String contact)

**Method: requests capabilities to a remote contact. This method initiates in background a new capability request to the remote contact by sending a SIP OPTIONS. The result of the capability request is sent asynchronously via callback method of the capabilities listener. A capability resfresh is only sent if the timestamp associated to the capability has expired (the expiration value is fixed via MNO provisioning). The parameter contact supports the following formats: MSISDN in national or international format, SIP address, SIP-URI or Tel-URI. If the format of the contact is not supported an exception is thrown. The result of the capability refresh request is provided to all the clients that have registered the listener for this event.**

void requestContactCapabilities(String contact)

**Method: requests capabilities for a group of remote contacts. This method initiates in background new capability requests to the remote contact by sending a SIP OPTIONS. The result of the capability request is sent asynchronously via callback method of the capabilities listener. A capability resfresh is only sent if the timestamp associated to the capability has expired (the expiration value is fixed via MNO provisioning). The parameter contact supports the following formats: MSISDN in national or international format, SIP address, SIP-URI or Tel-URI. If the format of the contact is not supported an exception is thrown. The result of the capability refresh request is provided to all the clients that have registered the listener for this event.**

void requestContactCapabilities(Set<String> contacts)

**Method: requests capabilities for all contacts existing in the local address book. This method initiates in background new capability requests for each contact of the address book by sending SIP OPTIONS. The result of a capability request is sent asynchronously via callback method of the capabilities listener. A capability resfresh is only sent if the timestamp associated to the capability has expired (the expiration value is fixed via MNO provisioning). The result of the capability refresh request is provided to all the clients that have registered the listener for this event.**

void requestAllContactsCapabilities()

**Method: registers a listener for receiving capabilities on any contact.**

void addCapabilitiesListener(ICapabilitiesListener listener);

**Method: unregisters a capabilities listener.**

void removeCapabilitiesListener(ICapabilitiesListener listener);

**Method: registers a capabilities listener for receiving capabilities on a list of contacts.**

void addCapabilitiesListener(Set<String> contacts, ICapabilitiesListener listener);

**Method: unregisters a capabilities listener on a list of contacts.**

void removeCapabilitiesListener(Set<String> contacts, ICapabilitiesListener listener);

**Class ICapabilitiesListener:**

This class offers callback methods for the listener of capabilities.

**Method: callback called when new capabilities are received for a given contact.** The first argument contact contains the canonical representation of the identity of the contact whose capabilities are indicated by the second argument capabilities.

void onCapabilitiesReceived(String contact, Capabilities capabilities)

**Class Capabilities:**

This class encapsulates the different capabilities which may be supported by the local user or a remote contact.

**Method: returns true if file transfer is supported, else returns false**

boolean isFileTransferSupported()

**Method: returns true if IM/Chat is supported, else returns false**

boolean isImSessionSupported()

**Method: returns true if image sharing is supported, else returns false**

boolean isImageSharingSupported()

**Method: returns true if video sharing is supported, else returns false**

boolean isVideoSharingSupported()

**Method: returns true if the specified feature tag is supported, else returns false. The parameter tag represents the feature tag to be tested.**

boolean isExtensionSupported(String tag)

**Method: returns the list of supported RCS extensions**

Set<String> getSupportedExtensions()

#### Content Providers

A content provider is used to store locally the capabilities of each remote contact. In this case the capabilities may be read even if there is no connection to the RCS platform. There is one entry per remote MSISDN.

The content provider has the following columns:

|  |  |  |
| --- | --- | --- |
| **Data** | **Data type** | **Comment** |
| CONTACT\_NUMBER | TEXT | Contains the MSISDN of the contact associated to the capabilities |
| CAPABILITY\_IMAGE\_SHARING | Integer | Image sharing capability. Values: 1 (true), 0 (false) |
| CAPABILITY\_VIDEO\_SHARING | Integer | Video sharing capability. Values: 1 (true), 0 (false) |
| CAPABILITY\_IM\_SESSION | Integer | IM/Chat capability. Values: 1 (true), 0 (false) |
| CAPABILITY\_FILE\_TRANSFER | Integer | File transfer capability. Values: 1 (true), 0 (false) |
| CAPABILITY\_EXTENSIONS | TEXT | Supported RCS extensions. List of features tags semicolon separated (e.g. <TAG1>;<TAG2>;…;TAGn) |

#### RCS extensions

A MNO/OEM application can create a new RCS/IMS service by defining a new RCS capability (or RCS extension. This new service is identified by a feature tag which is the unique key to identify the service in the RCS API and to trigger the service internally in the device and to route the service in the network side. Then the new feature tag will be also used by the IMS API in order to implement the service.

To create a new capability, the MNO/OEM application should declare the new supported feature tag in its Android Manifest file. Then, when the MNO/OEM application is deployed on the device, the RCS service will detect automatically the new installed application and will take into account the new feature tag in the next capability refreshes via SIP OPTIONS.

When the MNO/OEM application is removed the RCS service will remove the associated capability from the next capability refreshes via SIP OPTIONS.

The role of the RCS service is to manage the extensions and to decide to take into account the new feature tag or not. This may be done by analyzing the certificate of the application supporting the feature tag or by checking the provisioning.

##### MNO/OEM extension

A MNO/OEM extension starts with the prefix « +g.3gpp.iari-ref=”urn%3Aurn-7%3A3gpp-application.ims.iari.rcse.xxx.yyy », where « xxx » is the MNO/OEM id and « yyy » is an id associated to the application implementing the RCS extension.

See the following API syntax to be added in the Android Manifest file:

<intent-filter>

<action android:name="**org.gsma.joyn.capability.EXTENSION**"/>

<data android:mimeType="**+g.3gpp.iari-ref/urn%3Aurn-7%3A3gpp-application.ims.iari.rcse.xxx.yyy**"/>

</intent-filter>

Samples:

**Tic Tac Toe game:**

<application android:label=*"Tic Tac Toe"*>

<activity android:name=*".Main"*>

<intent-filter>

<action android:name=*"org.gsma.joyn.capability.EXTENSION"*/>

<data android:mimeType=*"****+g.3gpp.iari-ref/urn%3Aurn-7%3A3gpp-application.ims.iari.rcse.orange.tictactoe****"*/>

</intent-filter>

</activity>

**IP voice call application:**

<application android:label=*"VoLTE"*>

<activity android:name=*".Main"*>

<intent-filter>

<action android:name=*"org.gsma.joyn.capability.EXTENSION"*/>

<data android:mimeType=*"***+g.3gpp.icsi-ref/urn%3Aurn-7%3A3gpp-service.ims.icsi.mmtel**"/>

</intent-filter>

</activity>

#### Permissions

Access to the Capabilities API is requires the following permissions:

org.gsma.joyn.RCS\_READ\_CAPABILITIES: this is a new permission that governs access to capability information.

android.permission.READ\_CONTACTS: this permission is required by any client using the capabilities service, since use of the API implicitly reveals information about past and current contacts for the device.

### IM/Chat API

This API exposed all functionality for the Instant messaging/chat Service. It allows:

* Sending messages to a contact.
* Starting group chats with a predefined participants list with an optional subject.
* Joining existing group chats.
* Rejoining existing group chats.
* Restarting a previous group chat.
* Extends a 1-1 chat to a group chat.
* Sending messages in a group chat.
* Leaving a group chat.
* Adding participants to a group chat.
* Retrieving information about a group chat (status, participants and their status)
* Receiving notifications about incoming messages, “is-composing” events, group chat invitations and group chat events.
* Accept/reject an incoming chat invitation.
* Displaying chat history (messages and group chats).
* Erasing chat history by user, by group chat, or by single messages.
* Marking messages as displayed.
* Receiving message delivery reports.
* Read configuration elements affecting IM.

Note: a chat (single/group) is identified by a unique conversation ID which corresponds to the “Contribution-ID” header in the signalling flow. This permits to have a permanent chat or group chat like user experience.

#### IM/Chat API calling flow

The figures in this section contain basic call flows of the IM/Chat service API.

##### Session establishment

Figure 4.5.5.1.1-1 is an example that shows the flow for a RCS client establishing a IM/Chat session with a remote contact.



**Figure 4.5.5.1.1-1: Establish a one-to-one chat session with remote party**

1. The RCS client instantiates a service instance of the IM/Chat Service, establishes a connection with the IM/Chat Service and associates the listener with this RCS client.
2. By selecting a user, opening the IM window and typing a message, the user initiates a IM/Chat session request to the remote party
3. RCS service sends the session request to remote party
4. The session request is delivered to the remote party and a confirmation of delivery of the message is received by the RCS service. This delivery confirmation is provided to the RCS client.
5. By opening the chat window and viewing the incoming message, the remote party accepts the chat session and sends a confirmation of message display
6. RCS service provides the message displayed notification to the RCS client
7. Using procedures described in Figure 4.5.5.1-y, the two users exchange IM / Chat messages

##### Incoming session request

Figure 4.5.5.1.2-1 is an example that shows the flow for a RCS client receiving a IM/Chat session request from a remote contact.



**Figure 4.5.5.1.2-1: Establish a one-to-one chat session with remote party**

1. The RCS client instantiates a service instance of the IM/Chat Service, establishes a connection with the IM/Chat Service and associates the listener with this RCS client.
2. IM/Chat Service receives a request for a session from a remote party
3. IM/Chat Service notifies the RCS client through the invocation of the appropriate listener function
4. RCS client notifies the user by displaying the message received in the incoming session invitation to the user. The user accepts the session request,

OPEN ISSUE: How does the RCS client get the message in the received SIP INVITE request?

1. RCS client retrieves the chat object associated with this session.
2. RCS client installs the listener function for this chat session.
3. RCS client requests the IM/Chat service to send to the remote party, an indication of display of the message to the user.
4. Using procedures described in Figure 4.5.5.1-y, the two users exchange IM / Chat messages

##### Message exchange after chat session establishment

Figure 4.5.5.1.3-1 is an example that shows the flow for a RCS client receiving a IM/Chat session request from a remote contact.

 **Figure 4.5.5.1.2-1: Establish a one-to-one chat session with remote party**

1. A party and B party have already established a IM/Chat session
2. Party A starts to type a new chat / IM message
3. Client A provides an indication of composing event to the IM/Chat service
4. IM/Chat service of party A provides indication of composing toward party B.
5. IM/Chat service of party B provides the indication to the IM client of party B
6. Party A completes typing the message and presses enter
7. Client A provides the message to the IM/Chat service A
8. IM/Chat service A sends the message to IM/Chat service B
9. IM/Chat service B provides an indication to the client B
10. IM/Chat service B provides delivery indication to IM/Chat service A
11. IM/Chat service A provides the delivery indication to IM client A
12. IM Client B displays the message to user
13. IM client B provides the displayed indication to the IM/Chat service B
14. IM/Chat service B provides the displayed indication to IM/Chat service A
15. IM/Chat service A provides the indicaation to the client A

#### Package

Package name **org.gsma.joyn.chat**

#### Methods and Callbacks

**Class ChatService:**

This class offers the main entry point to initiate chat conversations with contacts: 1-1 and group chat conversation. Several applications may connect/disconnect to the API.

**Method: connects to the API.**

void connect()

**Method: disconnects from the API.**

void disconnect()

**Method: creates a single chat with a given contact and returns a Chat instance. The parameter contact supports the following formats: MSISDN in national or international format, SIP address, SIP-URI or Tel-URI.**

Chat initiateSingleChat(String contact, String firstMessage, IChatListener listener)

**Method: initiates a group chat with a group of contact and returns a GroupChat instance. The subject is optional and may be null.**

GroupChat initiateGroupChat(Set<String> contacts, String subject, IGroupChatListener listener)

**Method: rejoins an existing group chat from its unique chat ID.**

GroupChat rejoinGroupChat(String chatId)

**Method: restarts a previous group chat from its unique chat ID.**

GroupChat restartGroupChat(String chatId)

**Method: deletes a particular chat conversation (single/group).**

void deleteChat(String chatId)

**Method: adds a listener on new chat invitation event.**

void addEventListener(INewChatListener listener)

**Method: removes a listener on new chat invitation event.**

void removeEventListener(INewChatListener listener)

**Method: returns the list of single chats in progress.**

Set<Chat> getChats()

**Method: returns a chat in progress from its unique ID.**

Chat getChat(String chatId)

**Method: returns a single chat from its invitation Intent.**

Chat getChatFor(Intent chatIntent)

**Method: returns the list of group chats in progress.**

Set<GroupChat> getGroupChats()

**Method: returns a group chat in progress from its unique ID.**

GroupChat getGroupChat(String chatId)

**Method: returns a group chat from its invitation Intent.**

GroupChat getGroupChatFor(Intent chatIntent)

**Method: returns the configuration for chat service.**

static ChatServiceConfiguration getConfiguration()

**Class INewChatListener:**

This class offers callback methods on new chat invitation event.

**Method: callback called when a new chat invitation has been received**

void onNewSingleChat(String chatId, ChatMessage message)

**Method: callback called when a new group chat invitation has been received**

void onNewGroupChat(String chatId)

**Class ChatMessage:**

This class contains chat message information for single and group chat.

**Method: returns the contact who sends the message.**

String getContact()

**Method: returns the message ID.**

String getId()

**Method: returns the message content.**

String getMessage()

**Method: returns the receipt date.**

Date getReceiptDate()

**Method: is displayed delivery report requested.**

boolean isDisplayedReportRequested()

**Class Chat:**

This class maintains the information related to a single chat and offers methods to manage the chat conversation.

**Method: returns the chat ID.**

String getChatId()

**Method: returns the remote contact.**

String getRemoteContact()

**Method: sends a chat message. The method returns a unique message ID.**

String sendMessage(String message)

**Method: sends a displayed delivery report for a given message ID.**

void sendDisplayedDeliveryReport(String msgId)

**Method: sends an “is-composing” event. The status is set to true when typing a message, else it is set to false.**

void sendIsComposingEvent(boolean status)

**Method: extends to group by adding new participants to the current chat.**

void extendToGroup(Set<String> participants)

**Method: deletes particular chat message(s).**

void deleteMessages(Set<String> messageIds)

**Method: adds a listener on chat events.**

void addEventListener(IChatListener listener)

**Method: removes a listener on chat events.**

void removeEventListener(IChatListener listener)

**Class IChatListener:**

This class offers callback methods on 1-1 chat events.

**Method: callback called when a new message has been received**

void onNewMessage(ChatMessage message)

**Method: callback called when a message has been delivered to the remote.**

void onReportMessageDelivered(String msgId)

**Method: callback called when a message has been displayed by the remote.**

void onReportMessageDisplayed(String msgId)

**Method: callback called when a message has failed to be delivered to the remote.**

void onReportMessageFailed(String msgId)

**Method: callback called when a “is-composing” event has been received. If the remote is typing a message the status is set to true, else it is false.**

void onComposingEvent(boolean status)

**Method: callback called when a 1-1 conversation with a given contact has been extended to a group chat identified by its chat ID.**

void onChatExtendedToGroup(String contact, String groupChatId)

**Class GroupChat:**

This class maintains the information related to a group chat and offers methods to manage the group chat conversation.

**Method: returns the chat ID.**

String getChatId()

**Method: returns the subject of the group chat.**

String getSubject()

**Method: returns the list of connected participants. A participant is identified by its MSISDN in national or international format, SIP address, SIP-URI or Tel-URI.**

Set<String> getParticipants()

**Method: accepts chat invitation.**

void acceptInvitation()

**Method: rejects chat invitation.**

void rejectInvitation()

**Method: sends a message to the group. This method returns a unique message ID.**

String sendMessage(String text)

**Method: sends a “is-composing” event. The status is set to true when typing a message, else it is set to false.**

void sendIsComposingEvent(boolean status)

**Method: adds participants to a group chat.**

void addParticipants(Set<String> participants)

**Method: returns the max number of participants for a group chat from the group chat info subscription (this value overrides the provisioning parameter).**

int getMaxParticipants()

**Method: quits a group chat conversation. The conversation will continue between other participants if there are enough participants.**

void quitConversation()

**Method: deletes particular chat message(s).**

void deleteMessages(Set<String> messageIds)

**Method: adds a listener on chat events.**

void addEventListener(IGroupChatListener listener)

**Method: removes a listener on chat events.**

void removeEventListener(IGroupChatListener listener)

**Class IGroupChatListener:**

This class offers callback methods on group chat events.

**Method: callback called when the session is well established and messages may be exchanged with the group of participants.**

void onSessionStarted()

**Method: callback called when a new message has been received**

void onNewMessage(ChatMessage message)

**Method: callback called when a message has failed to be delivered to the remote.**

void onReportMessageFailed(String msgId)

**Method: callback called when a “is-composing” event has been received. If the remote is typing a message the status is set to true, else it is false.**

void onComposingEvent(String contact, boolean status)

**Method: callback called when the session is terminated. The reason may be: TERMINATED\_BY\_USER, TERMINATED\_BY\_REMOTE or any other reason.**

void onSessionTerminated(int reason)

**Method: callback called when the session has failed.**

void onSessionError(int reason)

**Method: callback called when a new participant has joined the group chat.**

void onParticipantJoined(String contact, String contactDisplayname)

**Method: callback called when a participant has left voluntary the group chat.**

void onParticipantLeft(String contact)

**Method: callback called when a participant is disconnected from the group chat.**

void onParticipantDisconnected(String contact)

**Class ChatServiceConfiguration:**

This class represents the particular configuration of IM Service.

**Method: returns the “imCapAlwaysOn” configuration. True if Store and Forward capability is supported, False if no Store & Forward capability.**

boolean isImCapAlwaysOn()

**Method: returns the “imWarnSF” configuration. True if user should be informed when sending message to offline user. False if user should not be informed when sending message to offline user. This should be used with imCapAlwaysOn.**

boolean isImWarnSf()

**Method: returns the time after inactive chat session could be closed.**

int getChatSessionTimeout()

**Method: returns maximum participants in group chat session.**

int getGroupChatMaxParticipantsNumber()

**Method: return maximum single chat message’s length can have**

long getSingleChatMessageMaxLength()

**Method: returns the maximum single group chat message’s length can have**

long getGroupChatMessageMaxLength()

**Method: returns the max number of simultaneous single chats.**

int getMaxSingleChats()

**Method: returns the max number of simultaneous group chats.**

int getMaxGroupChats()

**Method: returns the SMS fallback configuration. True if SMS fallback procedure activated, else returns False.**

boolean isSmsFallback()

**Method: returns True if auto accept mode activated for chat, else returns False.**

boolean isChatAutoAcceptMode()

**Method: returns True if auto accept mode activated for group chat, else returns False.**

boolean isGroupChatAutoAcceptMode()

**Method: activates or deactivates the displayed delivery report on received chat messages.**

void setDisplayedDeliveryReport(boolean state)

#### Intents

**Intent broadcasted when a new chat conversation has been received. This Intent contains the following extras:**

* **“contact”: MSISDN of the contact sending the invitation.**
* **“contactDisplayname”: display name of the contact sending the invitation (extracted from the SIP address).**
* **“chatId”: unique ID of the single chat conversation.**
* **“firstMessage”: first chat message in the new conversation (parcelable object).**

org.gsma.joyn.chat.CHAT\_INVITATION

**Intent broadcasted when a new group chat invitation has been received. This Intent contains the following extras:**

* **“contact”: MSISDN of the contact sending the invitation.**
* **“contactDisplayname”: display name of the contact sending the invitation (extracted from the SIP address).**
* **“chatId”: unique ID of the group chat conversation.**
* **“subject”: subject of the group chat (optional).**

org.gsma.joyn.chat.GROUP\_CHAT\_INVITATION

#### Content Providers

A content provider is used to store the chat history persistently. There is one entry per chat message.

The content provider has the following columns:

CHAT

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Data | Data Type | Description |
| 1. | CHAT\_ID | String | Id for chat room |
| 2. | IS\_GROUP\_CHAT | Integer | Type of chat room (SINGLE = 0 or GROUP = 1) |
| 3. | STATE | Integer | State of chat room (INACTIVE = 0, ACTIVE = 1, CLOSED\_BY\_USER = 2, FAILED = 3, PENDING = 4, DECLINED = 5) |
| 4. | SUBJECT | String | Subject of the group chat room |

Message

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Data | Data Type | Description |
| 1. | MESSAGE\_ID | Long | Id of the message |
| 2. | CHAT\_ID | String | Id of chat room |
| 3. | CONTACT\_NUMBER | String | Contact number of sender |
| 4. | BODY | String | Body of the message |
| 5. | INSERTED\_TIMESTAMP | Long | Time when message inserted |
| 6. | SENT\_TIMESTAMP | Long | Time when message sent. If 0 means not sent. |
| 7. | DELIVERED\_TIMESTAMP | Long | Time when message delivered. If 0 means not delivered. |
| 8. | DISPLAYED\_TIMESTAMP | Long | Time when message displayed. If 0 means not displayed. |
| 9. | MIME\_TYPE | String | MIME type of message |
| 10. | MESSAGE\_STATUS | Integer | Status of chat message (UNREAD = 0, UNREAD\_REPORT = 1, READ = 2, SENDING = 3, SENT = 4, FAILED = 5, TO\_SEND = 6, BLOCKED = 7)  Status of system message (INVITED = 0, ACCEPTED = 1, DECLINED= 2, TIMEOUT = 3, JOINED = 4, GONE = 5, DISCONNECT = 6, BUSY = 7, FAILED = 8) |
| 11. | DIRECTION | Integer | Status direction of message (INCOMING = 0, OUTGOING = 1, IRRELEVANT = 2) |
| 12. | MESSAGE\_TYPE | Integer | Type of message (MESSAGE= 0, SYSTEM = 1) |

Note :

Status values for chat session management in CHAT->STATE :

* INACTIVE : inactive chat session (not started).
* ACTIVE : active chat session (started).
* CLOSED\_BY\_USER : ended session closed by user.
* FAILED : chat session has failed.

Type values of a chat session in CHAT->IS\_GROUP\_CHAT :

* SINGLE : a 1-to-1 chat session.
* GROUP : a group chat session.

Status values for a message in MESSAGE->MESSAGE\_STATUS:

In case of a chat message (MESSAGE->MESSAGE\_TYPE = MESSAGE):

* UNREAD : the message is delivered.
* UNREAD\_REPORT : the message is delivered and a displayed delivery report is requested.
* READ : message is read (i.e. DISPLAYED).
* SENDING : message is in progress of sending.
* SENT : message is sent.
* FAILED : message is failed to be sent.
* TO\_SEND : message is queued to be sent by stack.
* BLOCKED : message is a spam message.

In case of a system message (MESSAGE->MESSAGE\_TYPE = SYSTEM):

* INVITED : invitation pending.
* ACCEPTED : invitation has been accepted (i.e; JOINED).
* DECLINED : the invitation has been declined on remote participant.
* TIMEOUT : the invitation has been timeout .
* JOINED: participant has joined the group.
* GONE : participant has left the group (i.e. DEPARTED).
* DISCONNECT (i.e. BOOTED).
* BUSY.
* FAILED.

Type values of a message direction in MESSAGE->DIRECTION :

* INCOMING : a message from remote user (incoming message)
* OUTGOING : a message sent for a remote user / conference
* IRRELEVANT : a system messages (events)

Type values for a system message (conference subscription) MESSAGE->MESSAGE\_TYPE :

* MESSAGE: the message represents a chat message.
* SYSTEM : the message is considered a system (events) message.

#### Permissions

Access to the Chat API is requires the following permissions:

org.gsma.joyn.RCS\_USE\_CHAT: this is a new permission that governs access to the chat API, and is required both to receive and to send over an RCS chat session.

org.gsma.joyn.RCS\_READ\_CHAT: this is a new permission that that is required by a client in order to read the chat history from the content provider..

### File Transfer API

This API exposes all functionality related to transferring files via the File Transfer Service. It allows:

* Send a file transfer request
* Receive notifications about incoming file transfer and file transfer events.
* Retrieve the list of all file transfers and their statuses for a specific contact
* Clean all file transfer history or single file transfers (including the transferred files if possible)
* Monitor a file transfer’s progress.
* Cancel a file transfer in progress.
* Accept/reject an incoming file transfer request.
* Read configuration elements affecting file transfer

#### File Transfer API calling flow

The figures in this section contain basic call flows of the File Transfer service API.

##### Session establishment

Figure 4.5.6.1.1-1 is an example that shows the flow for a FT client establishing a FT session with a remote contact.



**Figure 4.5.6.1.1-1: Successful sending of a file to remote party**

1. The FT client instantiates a service instance of the File Transfer Service, establishes a connection with the FT Service and associates the listener with this RCS client.
2. By selecting a user and the file to be transferred, the user initiates a FT session request to the remote party
3. FT service sends the session request to remote party
4. The session request is delivered to the remote party and indication is received that the remote user has accepted the file transfer request.
5. The FT client is notified that the file transfer has started.
6. The FT service transfers the file to the remote party.
7. The FT client is notified that the file transfer is completed.

##### Incoming session request

Figure 4.5.6.1.2-1 is an example that shows the flow for a FT client receiving a File Transfer request from a remote contact.



**Figure 4.5.5.1.2-1: Successful reception of a file from remote party**

1. The FT client instantiates a service instance of the FT Service and establishes a connection with the FT Service.
2. The FT client associates the listener with this FT client.
3. The FT Service receives a request for a session from a remote party
4. The FT Service notifies the FT client through the invocation of the appropriate listener function
5. The FT client retrieves the file transfer session object from the FT service
6. The FT client retrieves the identity of the remote party that is sending the file.
7. The FT client retireves the name of the file that is being sent by the remote party.
8. The FT client retrieves the type of the file that is being sent by the remote party
9. The FT client retrieves the size of the file that is being sent by the remote party.
10. The FT client notifies the user by displaying the details of the file that is being sent to the user. The user accepts the request,
11. The FT client installs the listener function for this FT session.
12. RCS client notifes the FT service that the user has accepted to receive the file
13. The FT service informs the remote end that it is ready to receive the file
14. The FT client is notified that the file transfer has started.
15. The FT service interacts with the remote end to receive the file
16. The FT client is notified that the file transfer is complete.

#### Package

Package name **org.gsma.joyn.ft**

#### Methods and Callbacks

**Class FileTransferService:**

This class offers the main entry point to transfer files and to receive files. Several applications may connect/disconnect to the API.

**Method: connects to the API.**

void connect()

**Method: disconnects from the API.**

void disconnect()

**Method: returns the list of file transfers in progress.**

Set<FileTransfer> getFileTransfers()

**Method: returns a current file transfer from its unique ID.**

FileTransfer getFileTransfer(String transferId)

**Method: returns a current file transfer from its invitation Intent.**

FileTransfer getFileTransferFor(Intent ftIntent)

**Method: transfers a file to a contact. The parameter contact supports the following formats: MSISDN in national or international format, SIP address, SIP-URI or Tel-URI. The parameter file contains the complete filename including the path to be transfered.**

FileTransfer transferFile(String contact, String filename, IFileTransferListener listener)

**Method: returns the configuration for File Transfer service.**

static FileTransferServiceConfiguration getConfiguration()

**Method: adds a new file transfer invitation listener.**

void addNewFileTransferListener(INewFileTransferListener listener)

**Method: removes a new file transfer invitation listener.**

void removeNewFileTransferListener(INewFileTransferListener listener)

**Class FileTransfer:**

This class maintains the information related to a file transfer and offers methods to manage the transfer.

**Method: returns the file transfer ID of the file transfer.**

String getTransferId()

**Method: returns the remote contact.**

String getRemoteContact()

**Method: returns the complete filename including the path of the file to be transfered.**

String getFileName()

**Method: returns the size of the file to be transferred (in bytes).**

String getFileSize()

**Method: returns the MIME type of the file to be transferred.**

String getFileType()

**Method: returns the state of the file transfer. The status may have the following values: INVITED, INITIATED, STARTED, TRANSFERED, ABORTED, FAILED.**

int getState()

**Method: accepts file transfer invitation.**

void acceptInvitation()

**Method: rejects file transfer invitation.**

void rejectInvitation()

**Method: aborts the file transfer.**

void abortTransfer()

**Method: adds a listener on file transfer events.**

void addEventListener(IFileTransferListener listener)

**Method: removes a listener from file transfer.**

void removeEventListener(IFileTransferListener listener)

**Class IFileTransferListener:**

This class offers callback methods on file transfer events.

**Method: callback called when the file transfer is started.**

void onTransferStarted()

**Method: callback called when the file transfer has been aborted.**

void onTransferAborted()

**Method: callback called when the transfer has failed.**

void onTransferError(int reason)

**Method: callback called during the transfer progress.**

void onTransferProgress(long currentSize, long totalSize)

**Method: callback called when the file has been transfered. The parameter filename contains the complete filename including the path.**

void onFileTransfered(String filename)

**Class INewFileTransferListener:**

This class offers callback method to receive new file transfer invitation.

**Method: callback called when a new file transfer invitation has been received.**

void onNewFileTransfer(String transferId)

**Class FileTransferServiceConfiguration:**

This class represents the particular configuration of FT Service.

**Method: returns the file size warning of File Transfer configuration. It can return null if this value was not set by the autoconfiguration server (no need to warn).**

long getWarnSize()

**Method: returns the max file size of File Transfer configuration. It can return null if this value was not set by the autoconfiguration server.**

long getMaxSize()

**Method: returns the Auto Accept Mode of File Transfer configuration.**

boolean getAutoAcceptMode()

**Method: returns the max number of simultaneous file transfers.**

int getMaxFileTransfers()

#### Intents

**Intent broadcasted when a new file transfer invitation has been received. This Intent contains the following extras:**

* **“contact”: MSISDN of the contact sending the invitation.**
* **“contactDisplayname”: display name of the contact sending the invitation (extracted from the SIP address).**
* **“transferId”: unique ID of the file transfer.**
* **“filename”: name of the file.**
* **“filesize”: size of the file in bytes.**
* **”filetype”: MIME type of the file.**

org.gsma.joyn.ft.FILE\_TRANSFER\_INVITATION

#### Content Providers

A content provider is used to store the file transfer history persistently. There is one entry per file transfer.

The content provider has the following columns:

|  |  |  |
| --- | --- | --- |
| **Data** | **Data type** | **Comment** |
| FT\_ID | TEXT | Unique file transfer identifier |
| CONTACT\_NUMBER | TEXT | Contains the MSISDN of the remote contact |
| FILENAME | TEXT | Filename |
| TYPE | TEXT | MIME type of the file |
| DIRECTION | Integer | Incoming transfer or outgoing transfer |
| FILE\_SIZE | Long | File size in bytes |
| TRANSFERED\_SIZE | Long | Size transfered in bytes |
| TIMESTAMP | Long | Date of the transfer |
| STATE | Integer | See note below for the list of states |

Note :

Status values for file transfer session management :

* INVITED : incoming session.
* INITIATED : outgoing session.
* STARTED : invitation has been accepted and transfer is started.
* TRANSFERED : session has been terminated and transfer successful.
* ABORTED: session has been aborted.
* FAILED : session has failed.

#### Permissions

Access to the File Transfer API is requires the following permissions:

org.gsma.joyn.RCS\_FILETRANSFER\_RECEIVE: this is a new permission that is required by a client in order to handle the receipt of a file transferred from a remote party.

org.gsma.joyn.RCS\_FILETRANSFER\_SEND: this is a new permission that is required by a client in order to initiate the transfer of a file transferred to a remote party.

org.gsma.joyn.RCS\_FILETRANSFER\_READ: this is a new permission that is required by a client in order to read the file transfer history from the content provider.

### Image Share API

This API exposes all functionality related to transferring images during a CS call via the Image Share Service. It allows:

* Send an image share request
* Receive notifications about incoming image share invitation and sharing events.
* Monitors an image share’s progress.
* Cancel an image share in progress.
* Accept/reject an incoming image share request.
* Read configuration elements affecting image share.
  + - 1. **Image Share API calling flow**

The figures in this section contain basic call flows of the Image Share service API.

* + - * 1. **Session establishment**

Figure 4.5.7.1.1-1 is an example that shows the flow for a image share client establishing a image share session with a remote contact.



**Figure 4.5.7.1.1-1: Successful sharing an image with remote party**

1. The IS client instantiates a service instance of the Image Share Service, establishes a connection with the IS Service and associates the listener with this IS client.
2. By selecting a user and the image file to be transferred, the user initiates a IS session request to the remote party
3. IS service sends the session request to remote party
4. The session request is delivered to the remote party and indication is received that the remote user has accepted the image sharing request.
5. The IS client is notified that the image sharing has started.
6. The IS service transfers the image to the remote party.
7. The IS client is notified that the image sharing is completed.
   * + - 1. **Incoming session request**

Figure 4.5.7.1.2-1 is an example that shows the flow for a IS client receiving a Image Share request from a remote contact.



**Figure 4.5.7.1.2-1: Successful sharing of an image from remote party**

1. The IS client instantiates a service instance of the IS Service and establishes a connection with the IS Service.
2. The IS client associates the listener with this IS client.
3. IS Service receives a request for a session from a remote party
4. IS Service notifies the IS client through the invocation of the appropriate listener function
5. The IS client retrieves the image share session object from the FT service
6. The IS client retrieves the identity of the remote party that is sharing the image.
7. The IS client retireves the name of the image file that is being shared by the remote party.
8. The IS client retrieves the type of the image file that is being shared by the remote party
9. The IS client retrieves the size of the image file that is being shared by the remote party.
10. The IS client notifies the user by displaying the details of the image that is being shared to the user. The user accepts the request,
11. The IS client installs the listener function for this IS session.
12. The IS client notifes the IS service that the user has accepted to receive the image sharing request
13. The IS service informs the remote end that it is ready to receive the image to be shared
14. The IS client is notified that the image share has started.
15. The IS service interacts with the remote end to receive the image being shared
16. The IS client is notified that the image sharing is complete.

#### Package

Package name org.gsma.joyn.ish

#### Methods and Callbacks

**Class ImageShareService:**

This class offers the main entry point to share images during a CS call, when the call hangs up the sharing is automatically stopped. Several applications may connect/disconnect to the API.

**Method: connects to the API.**

void connect()

**Method: disconnects from the API.**

void disconnect()

**Method: returns the list of image sharing in progress.**

Set<ImageShare> getImageShares()

**Method: returns a current image sharing from its unique ID.**

ImageShare getImageShare(String sharingId)

**Method: returns a current image sharing from its invitation Intent.**

ImageShare getImageShareFor(Intent ishIntent)

**Method: shares an image with a contact. The parameter contact supports the following formats: MSISDN in national or international format, SIP address, SIP-URI or Tel-URI. The parameter file contains the complete filename including the path of the image to be shared. An exception if thrown if there is no ongoing CS call.**

ImageShare shareImage(String contact, String filename, IImageShareListener listener)

**Method: returns the configuration for image share service.**

static ImageShareServiceConfiguration getConfiguration()

**Method: adds a new image share invitation listener.**

void addNewImageShareListener(INewImageShareListener listener)

**Method: removes a new image share invitation listener.**

void removeNewImageShareListener(INewImageShareListener listener)

**Class ImageSharing:**

This class maintains the information related to an image sharing and offers methods to manage the sharing.

**Method: returns the sharing ID of the image sharing.**

String getSharingId()

**Method: returns the remote contact.**

String getRemoteContact()

**Method: returns the complete filename including the path of the file to be shared.**

String getFileName()

**Method: returns the size of the file to be shared (in bytes).**

String getFileSize()

**Method: returns the MIME type of the file to be shared.**

String getFileType()

**Method: returns the state of the image share. The status may have the following values: INVITED, INITIATED, STARTED, TRANSFERED, ABORTED, FAILED.**

int getState()

**Method: accepts image share invitation.**

void acceptInvitation()

**Method: rejects image share invitation.**

void rejectInvitation()

**Method: aborts the sharing.**

void abortSharing()

**Method: adds a listener on image sharing events.**

void addEventListener(IImageShareListener listener)

**Method: removes a listener from image sharing.**

void removeEventListener(IImageShareListener listener)

**Class IImageShareListener:**

This class offers callback methods on image sharing events.

**Method: callback called when the sharing is started.**

void onSharingStarted()

**Method: callback called when the sharing has been aborted. The reason may be: TERMINATED\_BY\_USER, TERMINATED\_BY\_REMOTE or TERMINATED\_CALL or any other reason**

void onSharingAborted(int reason)

**Method: callback called when the sharing has failed.**

void onSharingError(int reason)

**Method: callback called during the sharing progress.**

void onSharingProgress(long currentSize, long totalSize)

**Method: callback called when the sharing has been terminated. The parameter filename contains the complete filename including the path.**

void onImageShared(String filename)

**Class INewImageShareListener:**

This class offers callback method to receive new image share invitation.

**Method: callback called when a new image share invitation has been received.**

void onNewImageShare(String sharingId)

**Class ImageShareServiceConfiguration:**

This class represents the particular configuration of Image Sharing Service.

**Method: returns the file size warning of Image Sharing configuration. It returns 0 if this value was not set by autoconfiguration server (no need to warn).**

long getWarnSize()

**Method: returns the max file size of Image Sharing configuration. It can return null if this value was not set by autoconfiguration server.**

long getMaxSize()

#### Intents

**Intent broadcasted when a new image sharing invitation has been received. This Intent contains the following extras:**

* **“contact”: MSISDN of the contact sending the invitation.**
* **“contactDisplayname”: display name of the contact sending the invitation (extracted from the SIP address).**
* **“sharingId”: unique ID of the image sharing.**
* **“filename”: name of the image file**
* **“filesize”: size of the image file in bytes.**
* **”filetype”: MIME\_type of the image.**

org.gsma.joyn.ish.IMAGE\_SHARING\_INVITATION

#### Content Providers

A content provider is used to store the image sharing history persistently. There is one entry per image sharing.

The content provider has the following columns:

|  |  |  |
| --- | --- | --- |
| **Data** | **Data type** | **Comment** |
| SHARING\_ID | TEXT | Unique sharing identifier |
| CONTACT\_NUMBER | TEXT | Contains the MSISDN of the remote contact |
| FILENAME | TEXT | Filename |
| TYPE | TEXT | MIME type of the file |
| DIRECTION | Integer | Incoming sharing or outgoing sharing |
| FILE\_SIZE | Long | File size in bytes |
| TRANSFERED\_SIZE | Long | Size transfered in bytes |
| TIMESTAMP | Long | Date of the sharing |
| STATE | Integer | See note below for the list of states |

Note :

Status values for file transfer session management :

* INVITED : incoming session.
* INITIATED : outgoing session.
* STARTED : invitation has been accepted and sharing is started.
* TRANSFERED : session has been terminated and sharing successful.
* ABORTED: session has been aborted.
* FAILED : session has failed.

#### Permissions

Access to the Image Share API is requires the following permissions:

org.gsma.joyn.RCS\_IMAGESHARE\_RECEIVE: this is a new permission that is required by a client in order to handle the receipt of a image shared by a remote party.

org.gsma.joyn.RCS\_IMAGESHARE\_SEND: this is a new permission that is required by a client in order to initiate the sharing of an image with a remote party.

org.gsma.joyn.RCS\_IMAGESHARE\_READ: this is a new permission that is required by a client in order to read the image share history from the content provider.

### Video Share API

This API exposes all functionality related to sharing live video stream during a CS call via the Video Share Service. It allows:

* Send a video share request.
* Receive notifications about incoming video share invitation and sharing events.
* Cancel an ongoing video share.
* Accept/reject an incoming video share request.
* Read configuration elements affecting video share.

#### Package

Package name **org.gsma.joyn.vsh**

#### Methods and Callbacks

**Class VideoShareService:**

This class offers the main entry point to share live video during a CS call, when the call hangs up the sharing is automatically stopped. Several applications may connect/disconnect to the API.

**Method: connects to the API.**

void connect()

**Method: disconnects from the API.**

void disconnect()

**Method: returns the list of video sharing in progress.**

Set<VideoShare> getVideoShares()

**Method: returns a current video sharing from its unique ID.**

VideoShare getVideoShare(String sharingId)

**Method: returns a current video share from its invitation Intent.**

VideoShare getVideoShareFor(Intent vshIntent)

**Method: shares a live video stream with a contact. The parameter contact supports the following formats: MSISDN in national or international format, SIP address, SIP-URI or Tel-URI. The parameter player contains a media player which streams over RTP the live video from the camera. The media player is just an interface which permits to have a player implementation independants from the joyn API. An exception if thrown if there is no ongoing CS call.**

VideoSharing shareVideo(String contact, IVideoPlayer player, IVideoShareListener listener)

**Method: returns the configuration for video share service.**

static VideoShareServiceConfiguration getConfiguration()

**Method: adds a new video share invitation listener.**

void addNewVideoShareListener(INewVideoShareListener listener)

**Method: removes a new video share invitation listener.**

void removeNewVideoShareListener(INewVideoShareListener listener)

**Class VideoShare:**

This class maintains the information related to a video sharing and offers methods to manage the sharing.

**Method: returns the sharing ID of the video sharing.**

String getSharingId()

**Method: returns the remote contact.**

String getRemoteContact()

**Method: returns the video encoding (e.g. H264).**

String getVideoEncoding()

**Method: returns the video format (e.g. QCIF).**

String getVideoFormat()

**Method: returns the state of the video share. The status may have the following values: INVITED, INITIATED, STARTED, TRANSFERED, ABORTED, FAILED.**

int getState()

**Method: accepts video share invitation with a given renderer.**

void acceptInvitation(IVideoRenderer renderer)

**Method: rejects video share invitation.**

void rejectInvitation()

**Method: aborts the sharing.**

void abortSharing()

**Method: adds a listener on video sharing events.**

void addEventListener(IVideoShareListener listener)

**Method: removes a listener from video sharing.**

void removeEventListener(IVideoShareListener listener)

**Class IVideoShareListener:**

This class offers callback methods on video sharing events.

**Method: callback called when the sharing is started.**

void onSharingStarted()

**Method: callback called when the sharing has been aborted. The reason may be: TERMINATED\_BY\_USER, TERMINATED\_BY\_REMOTE or TERMINATED\_CALL or any other reason**

void onSharingAborted(int reason)

**Method: callback called when the sharing has failed.**

void onSharingError(int reason)

**Class IVideoPlayer:**

This class offers an interface to manage the video player instance independently of the joyn service. The video player is implemented in the application side.

**Method: opens the player and prepares resources (e.g. encoder, camera).**

void open(VideoCodec codec, String remoteHost, int remotePort)

**Method: closes the player and deallocates resources.**

void close()

**Method: starts the player.**

void start()

**Method: stops the player.**

void stop()

**Method: returns the local RTP port used to stream video.**

int getLocalRtpPort()

**Method: returns the list of codecs supported by the player.**

VideoCodec[] getSupportedCodecs()

**Class IVideoRenderer:**

This class offers an interface to manage the video renderer instance independently of the joyn service. The video renderer is implemented in the application side.

**Method: opens the renderer and prepares resources (e.g. decoder, display).**

void open(VideoCodec codec, String remoteHost, int remotePort)

**Method: closes the renderer and deallocates resources.**

void close()

**Method: starts the renderer.**

void start()

**Method: stops the renderer.**

void stop()

**Method: returns the local RTP port used to stream video.**

int getLocalRtpPort()

**Method: returns the list of codecs supported by the renderer.**

VideoCodec[] getSupportedCodecs()

**Class VideoCodec:**

This class maintains the information related to a video codec.

**Method: returns the encoding name (e.g. H264).**

String getEncoding()

**Method: returns the codec payload (e.g. 96).**

int getPayload()

**Method: returns the codec clock rate (e.g. 90000).**

int getClockRate()

**Method: returns the codec frame rate (e.g. 10).**

int getFrameRate()

**Method: returns the codec bit rate (e.g. 64000).**

int getBitRate()

**Method: returns the video width (e.g. 176).**

int getVideoWidth()

**Method: returns the video height (e.g. 144).**

int getVideoHeight()

**Method: returns a codec parameter from its key name (e.g. profile-level-id, packetization-mode).**

String getParameter(String key)

**Class VideoShareServiceConfiguration:**

This class represents the particular configuration of Video Sharing Service.

**Method: returns maximum authorized duration of the content that can be shared in a VSH session. It can return null if this value was not set by autoconfiguration server.**

int getMaxTime()

#### Intents

**Intent broadcasted when a new video sharing invitation has been received. This Intent contains the following extras:**

* **“contact”: MSISDN of the contact sending the invitation.**
* **“contactDisplayname”: display name of the contact sending the invitation (extracted from the SIP address).**
* **“sharingId”: unique ID of the sharing.**
* **”encoding”: video encoding (e.g. H264).**
* **“format”: video format (e.g. QCIF).**

org.gsma.joyn.vsh.VIDEO\_SHARING\_INVITATION

#### Content Providers

A content provider is used to store the video sharing history persistently. There is one entry per video sharing.

The content provider has the following columns:

|  |  |  |
| --- | --- | --- |
| **Data** | **Data type** | **Comment** |
| SHARING\_ID | TEXT | Unique sharing identifier |
| CONTACT\_NUMBER | TEXT | Contains the MSISDN of the remote contact |
| DIRECTION | Integer | Incoming sharing or outgoing sharing |
| TIMESTAMP | Long | Date of the sharing |
| STATE | integer | See note below for the list of states |

Note :

Status values for video sharing session management :

* INVITED : incoming session.
* INITIATED : outgoing session.
* STARTED : invitation has been accepted and sharing is started.
* ABORTED: session has been aborted.
* FAILED : session has failed.

#### Sequence diagrams

The following diagrams explain the interactions between the video player or video renderer instanciated by the application and the joyn API.

**Originating side:**



**Terminating side:**



#### Permissions

Access to the Video Share API is requires the following permissions:

org.gsma.joyn.RCS\_VIDEOSHARE\_RECEIVE: this is a new permission that is required by a client in order to handle the receipt of a video shared by a remote party.

org.gsma.joyn.RCS\_VIDEOSHARE\_SEND: this is a new permission that is required by a client in order to initiate the sharing of an video with a remote party.

org.gsma.joyn.RCS\_VIDEOSHARE\_READ: this is a new permission that is required by a client in order to read the video share history from the content provider.

### Contacts API

There is already an Android API to manage contacts of the local address book, see Android package **android.provider.ContactsContract .** This API offers additional methods to:

* To add RCS info in the local address book,
* To extract RCS info from the local address book.

#### Package

Package name **org.gsma.joyn.contacts**

#### Methods and Callbacks

**Class ContactsService:**

This class offers methods to extract RCS info associated to contacts of the local address book.

**Method: connects to the API.**

void connect()

**Method: disconnects from the API.**

void disconnect()

**Method: returns the list of joyn contacts.**

Set<JoynContact> getJoynContacts()

**Method: returns the list of contacts online (i.e. registered).**

Set<JoynContact> getJoynContactsOnline()

**Method: returns the list of contacts supporting a given feature tag (i.e. capability).**

Set<JoynContact> getJoynContactsSupporting(String tag)

**Class JoynContact:**

This class maintains the information related to a joyn contact.

**Method: returns the canonical contact ID (i.e. MSISDN).**

String getContactId()

**Method: returns the capabilities associated to the contact.**

org.gsma.joyn.capability.Capabilities getCapabilities()

**Method: is contact online (i.e. registered to the service platform).**

boolean isRegistered()

#### Content Providers

In addition to the methods, the RCS information are stored in the local address book thanks to the Contacts Contract interface of the Android SDK. This permits to have a native integration of joyn in the address book.

See the following MIME-type to be supported:

|  |  |
| --- | --- |
| **MIME type** | **Comment** |
| vnd.android.cursor.item/org.gsma.joyn.number | RCS phone number |
| vnd.android.cursor.item/org.gsma.joyn.registration-state | Registration state (online | offline) |
| vnd.android.cursor.item/org.gsma.joyn.image-share | Image share capability supported |
| vnd.android.cursor.item/org.gsma.joyn.video-share | Video share capability supported |
| vnd.android.cursor.item/org.gsma.joyn.im-session | IM/Chat capability supported |
| vnd.android.cursor.item/org.gsma.joyn.file-transfer | File transfer capability supported |
| vnd.android.cursor.item/org.gsma.joyn.extensions | RCS extensions supported |

Implementation notes :

* To store the MIME-type see the following tutorial <http://developer.android.com/reference/android/provider/ContactsContract.RawContacts.html>.
* A raw contact is created to store the RCS info associated to a contact. A RCS account is created to manage raw contacts.
* When a contact becomes enriched with RCS information, we associate a corresponding raw contact with MIME type vnd.android.cursor.item/vnd.joyn.
* The number associated to the contact is put into the field Data.DATA1.
* The supported MIME type is put into the field Data.MIMETYPE.
* The description associated to the supported MIME type is always put into the field Data.DATA2. This label is displayed at UI level (i.e. menu item of the local native address book).
* If a MIME type is not set for a contact, this means the associated capability is not supported.

#### Permissions

* Access to the Contacts API is requires the following permissions:
* android.permission.READ\_CONTACTS: this permission is required by any client using the capabilities service, since use of the API implicitly reveals information about past and current contacts for the device.
* Additionally, methods that reveal contact capabilities (getJoynContactsSupporting() and getCapabilities()) require:
* org.gsma.joyn.RCS\_READ\_CAPABILITIES: this is a new permission that governs access to capability information.

### API Versioning

This API maintains information about the current version of the RCS terminal API.

A build is identified by:

* GSMA version: hotfixes, Blackbird, .etc.
* Implementor name: entity name who has implemented the API.
* Release number of the API.
* Incremental number to identify the build into a release number.

A software release of the API is identified uniquely by its release number and the incremental number.

* + - 1. Package

Package name **org.gsma.joyn**

* + - 1. Methods and Callbacks

**Class Build:**

This class offers information related to the build version of the API.

**Constant: GSMA version number from class Build.GSMA\_CODES.**

public final static int GSMA\_VERSION

**Constant: API release implementer name.**

public final static String API\_CODENAME

**Constant: API version number from class Build.VERSION\_CODES.**

public final static int API\_VERSION.

**Constant: Internal number used by the underlying source control to represent this build.**

public final static int API\_INCREMENTAL

**Class Build.GSMA\_CODES:**

This class contains the list of GSMA versions.

**Constant: joyn hotfixes version**

public final static int *RCSE\_HOTFIXES\_1\_2*

**Constant: joyn Blackbird version**

public final static int *RCSE\_BLACKBIRD*

**Class Build.VERSION\_CODES:**

This class contains the list of API versions.

**Constant: The original first version of joyn API**

public final static int *BASE*

### Utility API

#### Canonical Identity

public string canonicalIdentity (string contact)

Called when the application needs to determine the canonical identity associated with a contact.

**Parameters**

*contact MSISDN that is to be converted to canonical form*

#### Intents

<text here>

#### Content Providers

<text here>

## Privileged Client API

### Overview

This section lists all APIs that are Joyn Client exclusive. Each of these APIs will only be used by Core Application. Each API exposes all its functionality on a high level and does put constraints on the invoking application as to the method preconditions.

This section will be defined in future version.

## IMS APIs

### Overview

This API exposes IMS core service functionalities. Stack APIs are divided into 2 main categories.

This section will be defined in future version.

1. Document Management
   1. Document History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version | Date | Brief Description of Change | Approval Authority | Editor / Company |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

* 1. Other Information

|  |  |
| --- | --- |
| Type | Description |
| Document Owner | <Working Group/Project> |
| Editor / Company |  |

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@gsm.org](mailto:prd@gsm.org)

Your comments or suggestions & questions are always welcome.