android
partner engineering

IMS Single Registration in Android

Contents

Objective

- Support a single IMS registration for managing both MMTEL features provided by an ImsService and RCS features provided by an RCS application in AOSP.
- Consolidate existing non-standard interfaces into a common AOSP API with CTS/VTS testing and validation.

Background

There have traditionally been many models for applications supporting RCS to access the carrier's core network in the Android framework, but they usually fall into one of two categories:

1. the "dual registration" model, as shown below in Figure 1.1. In this scenario, all applications that support IMS features perform their own IMS registration and manage their own IMS stacks to implement RCS and MMTEL features. This "over-the-top" model for supporting RCS is the easiest to manage from the device side and allows any application to be downloaded from the application store and provides RCS features to the user.
2. the "single registration" model shown in Figure 1.2. In this model, the device provides the ability for an RCS application to access the IMS stack that exists in the telephony layer to support MMTEL features, such as voice/video calling and SMS over IMS. This "primary" IMS stack manages the IMS registration for all IMS implementations on the device as well as the SIP signalling traffic to and from the network. There are many variations of this second model, for example, a partner can choose to implement all RCS features in the telephony stack itself and provide access to these RCS features to a messaging application at a higher level or instead provide a lower level signalling transport interface and an RCS application itself implements the RCS features.

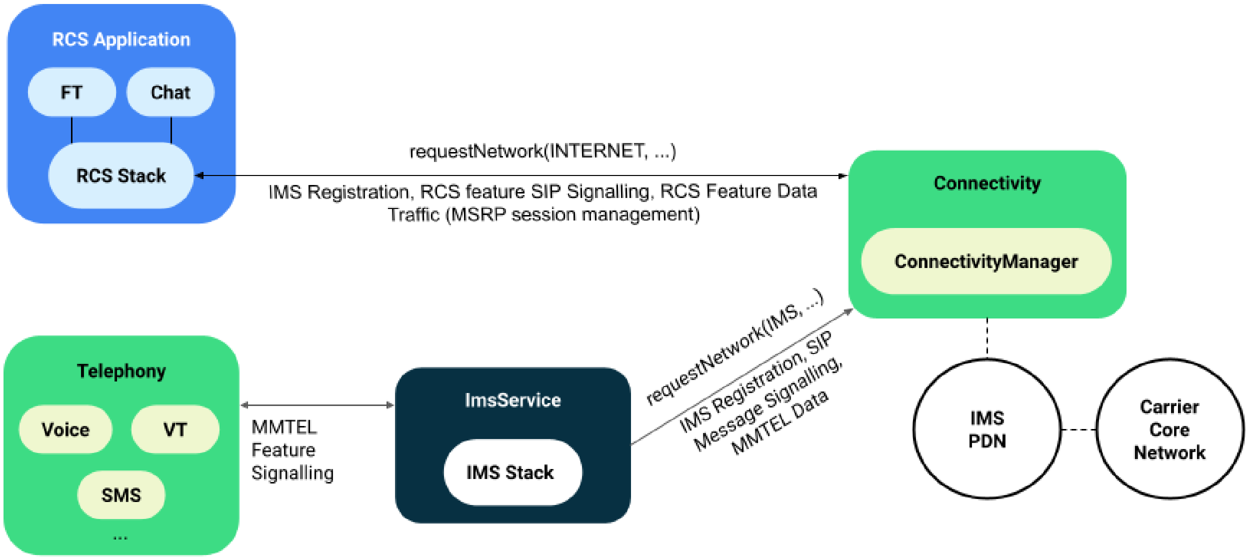

RCS Application
FT
Chat
RCS Stack
requestNetwork(INTERNET, ...)
IMS Registration, RCS feature SIP Signalling, RCS Feature Data Traffic (MSRP session management)
Connectivity
ConnectivityManager
Telephony
Voice
VT
SMS
...
MMTEL Feature Signalling
ImsService
IMS Stack
requestNetwork(IMS, ...)
IMS Registration, SIP Message Signalling, MMTEL Data
IMS PDN
Carrier Core Network

Figure 1.1: An RCS application in an over-the-top dual registration scenario. The RCS stack and device `ImsService` do not know about each other and manage their own IMS registration to the cellular carrier's network, SIP Message dialog handling, and data traffic.

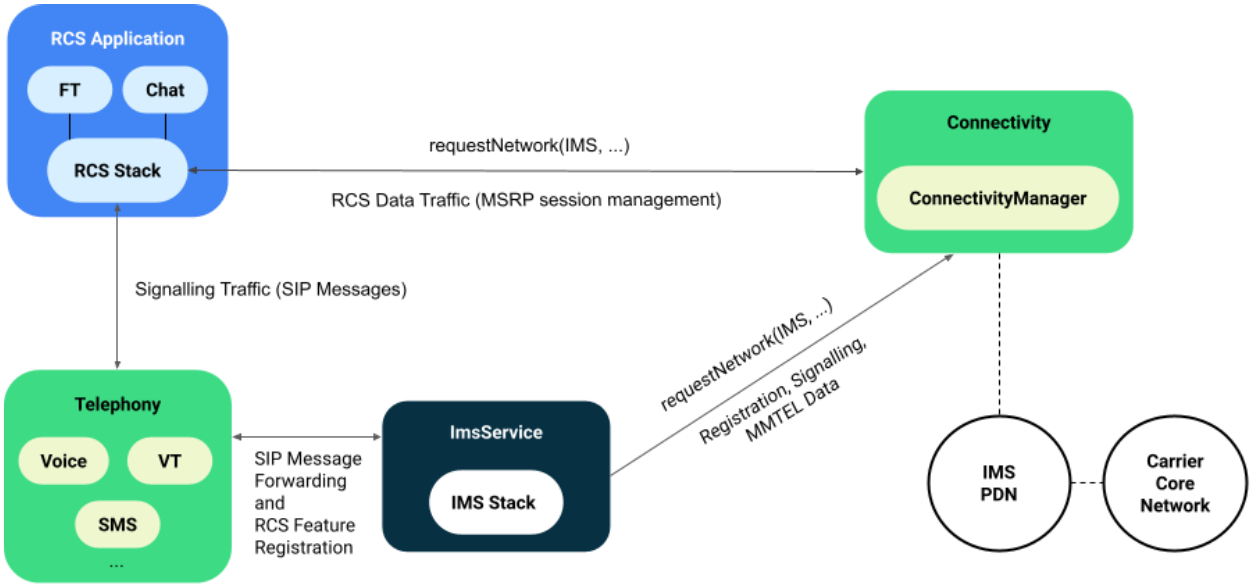


Figure 1.2: An RCS application configured in an example single registration scenario. The device `ImsService` manages a single IMS registration on the cellular carrier's network for all IMS enabled applications. Data traffic is still handled by the RCS application over an established dedicated bearer, however SIP messaging traffic is forwarded to the `ImsService` separately to

be sent over the same port bound to the carrier's core network.

North American cellular carriers have formed the Cross Carrier Messaging Initiative (CCMI) and have required that all devices on their network use the "single registration" model for implementing RCS moving forward. In order to support these requirements, Android 12 will be introducing new APIs to provide applications with the ability to integrate with the device `ImsService` to support RCS as part of the AOSP ecosystem.

Overview

There are four components involved in supporting IMS Single Registration: the RCS messaging application, AOSP telephony service, vendor or OEM provided IMS service, and vendor radio service. Figure 2.1 below illustrates the roles and responsibilities of all four components as well as the changes required to support single registration. In summary, single registration requires the following changes to be made:

1. Extend the IMS Service API by
    a. Defining a `SipTransport` API that allows the messaging application to forward SIP messages from its SIP stack to the vendor SIP stack and vice-versa to be sent and received over the carrier network.
    b. Extending the existing ImsConfig API to provide the ability for applications to provision the device for RCS.
    c. Provide the user capability exchange (UCE) service so that the IMS service can manage capability exchange on behalf of all applications.
2. Extend the vendor `IRadio` interface by defining a `QosSession` callback mechanism to allow the RCS messaging application to listen to dedicated bearer setup/teardown on the IMS PDN for active MSRP connections as well as any Quality of Service (QoS) information.
3. Provide a GbaService API to allow the vendor to provide GBA authentication if required by the carrier.
4. Refactor the RCS Messaging Application to
    a. Open up a SIP transport to the vendor IMS service for the features that it supports for SIP signalling through the existing IMS registration and authenticate when required.
    b. Listen to provisioning updates provided by the framework.
    c. Set up and tear down MSRP connections and use updates from the QoS callback mechanism to start data traffic when required.
    d. Provide capability updates to the framework for publishing UCE information as well as receive cached presence information from contacts.

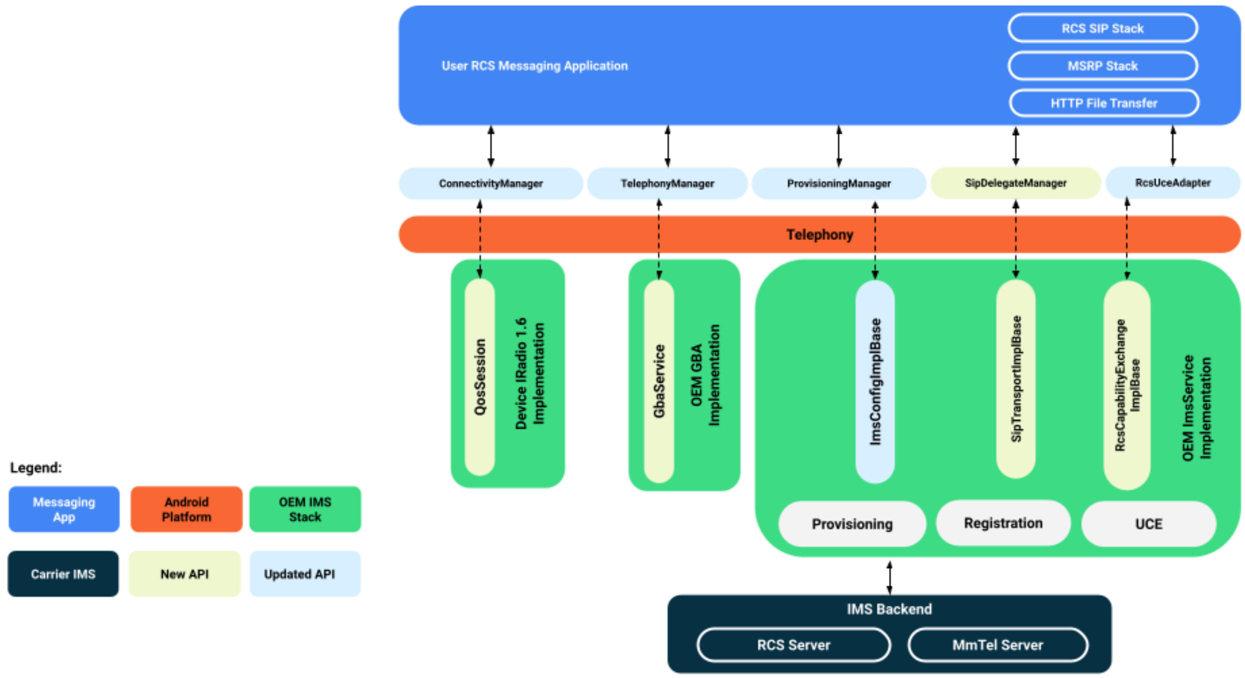


Figure 2.1: A summary of each of the components required to support single registration as well as a summary of the new/modified AOSP APIs as part of this feature.

This document covers the new API surfaces that will be available to both an RCS application wishing to use single registration as well as the partner changes required to support single registration on an Android device. If a new Android device supports IMS as part of the AOSP telephony stack, they will be required to support these AOSP APIs in order to meet the North American CCMI requirements.

For the purposes of readability and focused discussion, the API changes have been split into five surfaces, which are outlined above in Figure 2.1 and will all be required in order to support single registration. The following table outlines each API surface area as well as more details on what that surface area provides in the context of single registration:

API Surface Area	RCS application APIs	Vendor IMS APIs	Description
RCS Provisioning	ProvisioningManager	ImsConfigImplBase	Allows an OEM or carrier to provide an app to update the RCS provisioning status if the carrier uses a proprietary carrier entitlement mechanism. The ImsService must also support the standard AutoConfigurationServer (ACS) for provisioning for carriers that don't use a proprietary mechanism.
SIP Message Forwarding	SipDelegateManager	SipTransportImplBase	Allows an RCS application to first associate specific RCS feature tags with the device `ImsService` associated with the `RcsFeature` and then send/receive SIP messages and IMS registration updates associated with those RCS feature tags.
Dedicated Bearer Notifications	ConnectivityManager	DataCallResponse	Allows an app to listen to QoS notifications on a socket that's associated with a specific local port and connected to a specific remote port.
GBA Authentication	bootstrapAuthenticationRequest	GbaService	Allows an RCS app to authenticate with the network and access keys used for RCS features such as file transfer.
RCS User Capability Exchange	ImsRcsManager	RcsCapabilityExchangeImplBase	Provides AOSP the ability to send its MMTEL and RCS capabilities to the vendor ImsService so that they can be PUBLISHed under one entity to the network for RCS User Capability Exchange. Also allows other apps interested in the RCS capabilities of one or more contacts to query the network for the contact(s) RCS capabilities.

Design

The following section contains an in-depth design description as well as the API definitions, examples, and discussion for each API surface. Each section of the design contains the overview and implementation instructions for both the RCS application and the vendor ImsService.

Configuration

The following section outlines how to configure a device for IMS single registration support.

In order to enable IMS single registration APIs, the following device feature flag must be defined: FEATURE_TELEPHONY_IMS_SINGLE_REGISTRATION. If this feature is enabled, the device must support all of the new APIs defined in this document.

Carrier Configurations

- Once device support has been enabled, individual carrier support must be enabled via the carrier configuration : KEY_IMS_SINGLE_REGISTRATION_REQUIRED_BOOL.
- For User Capability exchange to be supported, all other configurations in CarrierConfigManager.Ims must be configured.
- ACS support must be enabled per carrier via the KEY_USE_ACS_FOR_RCS_BOOL key or else the carrier/OEM must provide a carrier entitlement application on a per carrier basis if the carrier requires provisioning.

Security

The purpose of this section is to outline a coherent strategy for secure access to the IMS PDN and the telephony framework. Unless otherwise noted in each of the API design sections below, all of the APIs in this document will be following the same security strategy in Android S. In future releases, this may be amended or modified to support more use cases, as these APIs become more mature.

In order for an application to access the IMS PDN and the SIP message transport APIs, the application must do the following:

1) The IMS application must have been granted the `android.permission.CONNECTIVITY_USE_RESTRICTED_NETWORKS` permission, which can only happen if the permission is granted using the OEM permission allowlist. This requires the application either be a privileged application or a system signed application, which also means that the application will need to be preinstalled and signed with the correct keys.
2) In order to access SIP Transport API, the application will need to be granted the `android.permission.PERFORM_IMS_SINGLE_REGISTRATION` permission and also be set as the user's default messaging application.
3) To be able to request GBA authentication, the application must have carrier permissions (see hasCarrierPrivileges), `android.permission.`PERFORM_IMS_SINGLE_REGISTRATION, or `android.permission.`MODIFY_PHONE_STATE permissions.
4) In order to access User Capability Exchange APIs, the IMS application must be either the default messaging application, default dialer application, or the default contacts application and request the `android.permission.`ACCESS_RCS_USER_CAPABILITY_EXCHANGE permission.

Note: The single registration APIs used to access the IMS PDN are not being made available to downloadable applications with "carrier privileges" at this time. It will only be made accessible to preinstalled applications that have been granted the `android.permission.CONNECTIVITY_USE_RESTRICTED_NETWORKS` permission.

Provisioning

To enable IMS Single Registration, operators update the RCS configuration supported by the network over the air. RCS configuration received over the air shall be broadly classified into the following categories as per GSMA RCC.07 Annex A:

1. RCS Volte Single Registration Enabled
2. Allowed RCS services
3. Allowed RCS services with mobile data off
4. Configuration parameters for each service (like file size, maximum message length etc)
5. User capability exchange (UCE) mechanism and related parameters for Presence and Subscribe

These parameters shall be consumed by:

- RCS application to trigger the single IMS registration over the IMS PDN and configure various RCS services
- AOSP for application validation, trigger User Capability Exchange, etc...
- IMS vendor stack to trigger initial IMS registration and support UCE signaling like PUBLISH/SUBSCRIBE/OPTIONS/NOTIFY etc

Also there are certain application specific parameters which may be required by the android framework and the ACS client like the RCS client version, vendor name, etc... which are required for fetching the configuration, constructing the user agent header, etc.

This document shall address these requirements for an end to end solution.

Proposed Design

Provisioning using entitlement server

- OEMs implement their own carrier settings application to fetch the carrier configuration from the entitlement server when this provisioning mechanism is used. Application shall use the AOSP APIs to pass the received configuration.
- The carrier settings application must parse the configuration XML and update the Android framework with the relevant RCS configuration (for example, `<RCSConfig>...</RCSConfig>`) section of the XML. For other non-RCS parameters in the XML, applications must use the pre existing mechanism to pass the configuration.
- The carrier configuration KEY_USE_ACS_FOR_RCS_BOOL must be set to false for this configuration.

Provisioning using AutoConfiguration Server

- The Vendor IMS stack shall implement the autoconfiguration client using the `ImsConfigImplBase` API when ACS is used. It shall use the AOSP APIs in `ProvisioningManager` to pass the received configuration into telephony.
- A carrier configuration KEY_USE_ACS_FOR_RCS_BOOL is introduced to indicate if Auto Configuration server is used.

The provisioning mechanism used by the provisioning application shall not impact the API interface used by the application to provide the configuration to the framework and the interface used by the RCS application to get the configuration.
`ProvisioningManager` shall be enhanced for storing and publishing the RCS configuration on a per subscription basis.

Design Description

RCS Messaging Application Responsibilities

`ProvisioningManager` defines a new Intent with action ACTION_RCS_SINGLE_REGISTRATION_CAPABILITY_UPDATE, which provides the single registration capability of the device and the carrier to the default messaging application. This intent only provides the capability of single registration and not the current provisioning status of the IMS Single Registration feature.
An Int extra EXTRA_SUBSCRIPTION_INDEX is included to specify the subscription index for which the intent is valid. An int extra EXTRA_STATUS shall also be included to provide the capability status. Possible values are :

- KEY_STATUS_CARRIER_NOT_CAPABLE
- KEY_STATUS_DEVICE_NOT_CAPABLE
- KEY_STATUS_CAPABLE

Telephony shall consider the carrier's IMS Single Registration capability and the IMS stack capability to support single IMS registration. If either of them do not support IMS Single Registration, the telephony framework shall trigger the Intent with state KEY_STATUS_CARRIER_NOT_CAPABLE or KEY_STATUS_DEVICE_NOT_CAPABLE as the case may be.
Otherwise, the intent is triggered with the state KEY_STATUS_CAPABLE. This explicit intent is sent to the registered application on the following events:

1. On boot, after the carrier configuration is loaded for the subscription
2. On active subscription change

3. On default messaging application change
4. If the carrier, or device capability is updated. For example if the carrier has a late single registration deployment, or the device adds capability after an IMS software upgrade etc.

When the application receives the intent with extra `KEY_STATUS_CAPABLE`. The application must send its RCS client parameters to telephony via the applicable `ProvisioningManager` API. When the UE and the network are capable and the default messaging application has sent the RCS client parameters to telephony, then the parameters will be provided to the provisioning client. The provisioning client shall then proceed with fetching the RCS provisioning XML.

Provisioning Client Responsibilities

Provisioning clients shall use the existing ***notifyRcsAutoConfigurationReceived***() API to provide the RCS configuration XML to `ProvisioningManager`. `ProvisioningManager` shall parse the XML and store some key parsed values like `rcsVolteSingleRegistration`, Services, DataOff, etc as defined in RCC.07 Annex A along with the complete XML. The keys are stored to allow the android framework to have an easier access to these parameters. If operators choose to define custom tags for these parameters, then mapping keys can be defined in `CarrierConfigManager` to parse the XML. RCS configuration shall be stored persistently across power cycles in the devices subscription database.
`ProvisioningManager` shall allow the RCS application to register a callback to receive the configuration XML and shall notify any changes to them using the registered callback.

When the default messaging application is changed by the user and it's no longer the preinstalled RCS application then the AOSP telephony framework shall detect this condition and purge the stored xml. It shall also inform the RCS application regarding the configuration reset and then unregister its provisioning callback.
When default messaging application is changed by the user and it's set back to the preinstalled RCS application then AOP telephony shall wait for the Provisioning client to update the XML using `ProvisioningManager#notifyRcsAutoConfigurationReceived`.
`ProvisioningManager` shall listen to the `PackageManager#MATCH_SYSTEM_ONLY` filter to determine if the application is a default system messaging application.

Figure 3.2.1: below illustrates the high level flow and the components involved.

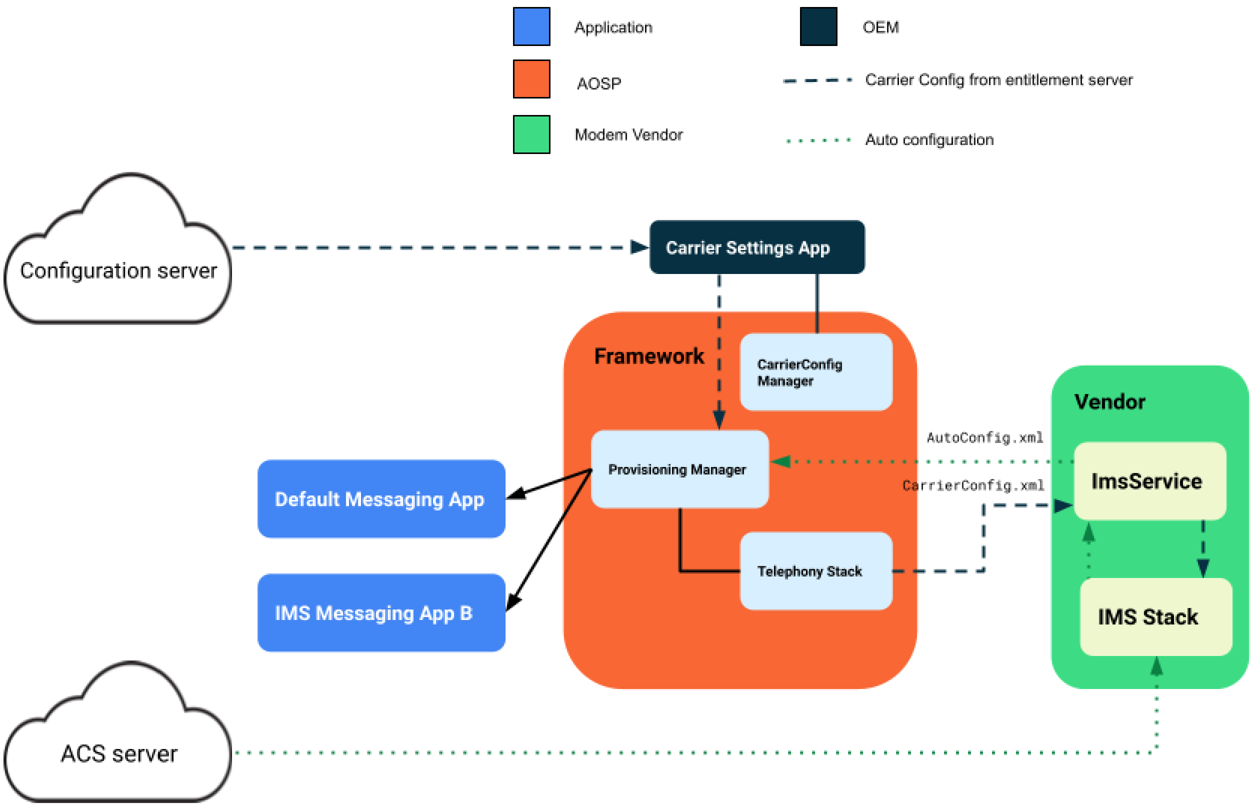


Figure 3.2.1: RCS Configuration update using either ACS or carrier configuration server.

API Definition

Provisioning manager interface shall be extended to expose an API to fetch the RCS configuration for each subscription.

Permissions

RCS Provisioning android API will only support the “system” application, which means that the application is one of the preinstalled, updatable applications that an OEM provides. This gives the application the ability to access privileged permissions that are not typically accessible to 3rd party applications that are downloaded onto the device. To receive provisioning updates, an RCS application must be the Default messaging application.

RCS Application API

The RCS Application shall create an instance of provisioning manager on bootup or whenever the application is ready to manage RCS traffic for the active subscription. RCS applications must register the RCS client parameters on boot or after its upgrade.These parameters shall be used by the ACS client during autoconfiguration HTTPS GET request.

Telephony shall send the `ACTION_RCS_SINGLE_REGISTRATION_CAPABILITY_UPDATE` Intent explicitly to the default messaging application (DMA). This will also start the application if it is not currently alive at the time. The DMA can cache the status of this intent for later use if it's not ready yet.

If the status indicates `KEY_STATUS_CAPABLE`, then the application must register for provisioning change events and process the provisioning XML sent by the carrier. It shall register a callback to receive the RCS configuration xml using the below defined new methods in the ProvisioningManager. The service listening to the provisioning updates is expected to remain alive to receive and process the callback events while the callback is registered.

If the status indicates device or the carrier are not capable of RCS volte single IMS registration, then the application can choose to unbind its provisioning monitoring service using the existing PackageManager#COMPONENT_ENABLED_STATE_DISABLED to save battery. If the status changes at later time and another Intent is received with the updated value, RCS application can again bind the provisioning monitoring service using PackageManager#COMPONENT_ENABLED_STATE_ENABLED.

Figure 3.2.2 depicts the updated class diagram

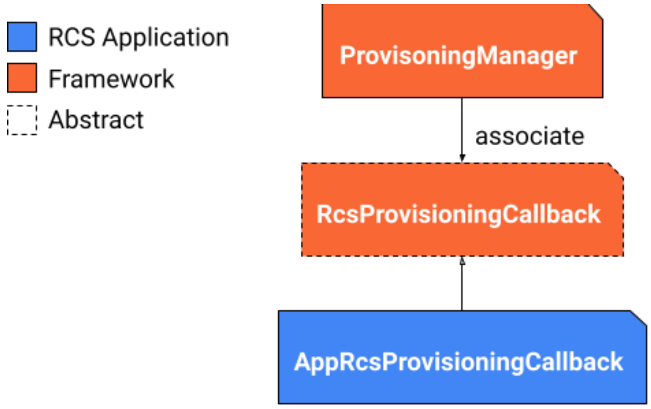


Figure 3.2.2: New Provisioning API class definitions

`ProvisioningManager` interface shall also be enhanced to provide APIs for the RCS application to trigger ACS reconfiguration due to a HTTP level authentication failure. When reconfiguration is triggered, `ProvisioningManager` shall purge the stored xml and inform the application regarding the configuration reset.

Please see the `ProvisioningManager` and `RcsClientConfiguration` documentation for more information on how the RCS provisioning API is used.

ImsService API Additions

Entitlement Server for Provisioning

When an entitlement server is used for RCS provisioning, then a carrier's entitlement application must use `ProvisioningManager` to pass the configuration XML. The platform will then use the `ImsConfigImplBase` interface ***notifyRcsAutoConfigurationReceived***() to provide the configuration xml to the vendor, which can then be used to update provisioning information in the IMS stack.
Ims stack is expected to store the xml in persistent memory and use the same XML across boot cycles until itl is updated by the entitlement application at a later time.

ACS for Provisioning

When ACS is configured to be used by a carrier for provisioning:

- IMS stack shall implement the ACS client and update the Android framework using the existing ProvisioningManager#***notifyRcsAutoConfigurationReceived***() API to provide the RCS configuration XML whenever a new XML is received.
- `ImsConfigImplBase#setRcsClientConfiguration` will be called when the RCS application sends the framework its RCS client configuration parameters. The ImsService or vendor IMS stack should detect the change in the application configuration and trigger ACS reconfiguration as required.
- Some operators require an additional authentication token to be used by the ACS client during the initial HTTP get request. The ImsService or the ACS client shall acquire the token as defined by the operator specifications.
- EAP-AKA, OTP SMS and other authentication means for ACS shall be supported by the ACS client.
- ImsService shall intercept the incoming port directed SMS with the one-time password or Network requested configuration request. ImsService uses it for initial authentication with the ACS server or triggers the reconfiguration procedure if required by the operator.

- If there is an error during ACS, `ImsConfigImplBase#notifyAutoConfigurationErrorReceived` must be called to notify the framework of the error.
- A new method shall be added to the ImsConfigImplBase class for the framework to pass the RCSApplication triggered reconfiguration request to the ACS client.
- `ImsService` shall detect conditions like factory reset, Maintenance release upgrade and trigger ACS reconfiguration as required.
- When default messaging application is changed by the user and it's set back to the preinstalled RCS application, then `ImsService` shall detect condition and trigger ACS reconfiguration as required.

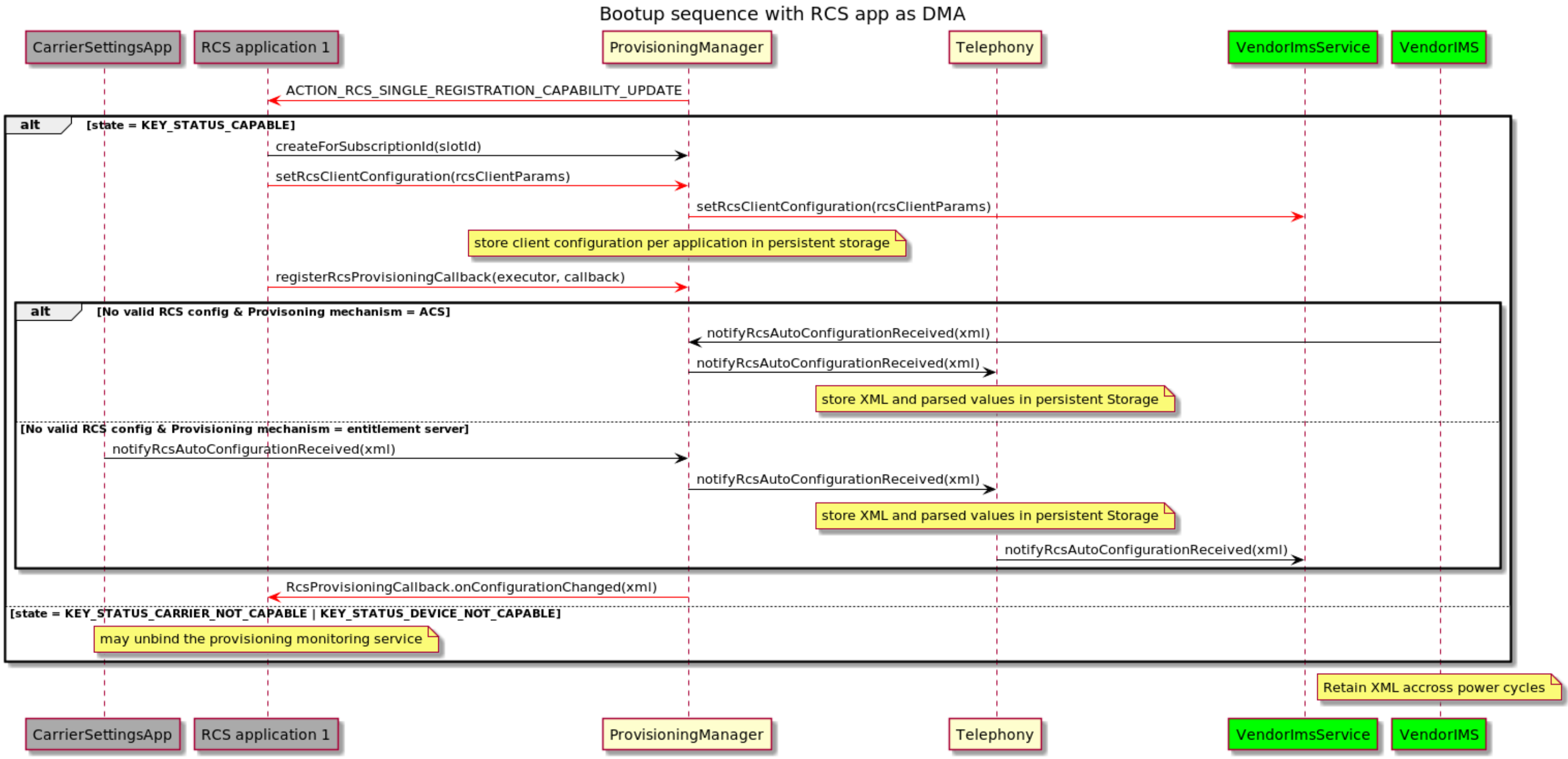
Sequence diagrams:

Figures below depict the execution flow for various sunny day use cases:

Scenario P.CS.1 - Device bootup sequence

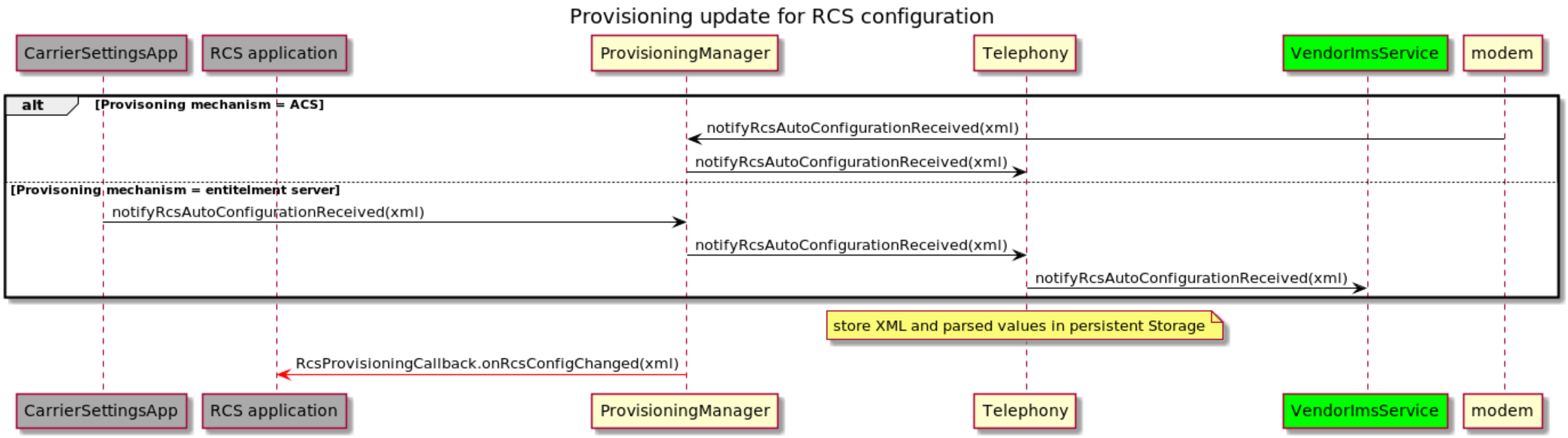
On boot, an RCS application shall provide the application configuration. It can use the intent ACTION_RCS_SINGLE_REGISTRATION_CAPABILITY_UPDATE to determine if single registration is possible and register a provisioning callback to retrieve the XML. When the operator has enabled `rcsVolteSingleRegistration`, it can create SIP delegate connections using the SIP transport API.

Bootup sequence with RCS application as the default messaging application. In the case that an RCS application is not set as the default messaging application by the user, then they will not receive the `ACTION_RCS_SINGLE_REGISTRATION_CAPABILITY_UPDATE` intent.


Bootup sequence with RCS app as DMA
CarrierSettingsApp
RCS application 1
ProvisioningManager
Telephony
VendorImsService
VendorIMS
ACTION_RCS_SINGLE_REGISTRATION_CAPABILITY_UPDATE
alt
[state = KEY_STATUS_CAPABLE]
createForSubscriptionId(slotId)
setRcsClientConfiguration(rcsClientParams)
setRcsClientConfiguration(rcsClientParams)
store client configuration per application in persistent storage
registerRcsProvisioningCallback(executor, callback)
alt
[No valid RCS config & Provisoning mechanism = ACS]
notifyRcsAutoConfigurationReceived(xml)
notifyRcsAutoConfigurationReceived(xml)
store XML and parsed values in persistent Storage
[No valid RCS config & Provisioning mechanism = entitlement server]
notifyRcsAutoConfigurationReceived(xml)
notifyRcsAutoConfigurationReceived(xml)
store XML and parsed values in persistent Storage
notifyRcsAutoConfigurationReceived(xml)
RcsProvisioningCallback.onConfigurationChanged(xml)
[state = KEY_STATUS_CARRIER_NOT_CAPABLE | KEY_STATUS_DEVICE_NOT_CAPABLE]
may unbind the provisioning monitoring service
Retain XML accross power cycles
CarrierSettingsApp
RCS application 1
ProvisioningManager
Telephony
VendorImsService
VendorIMS

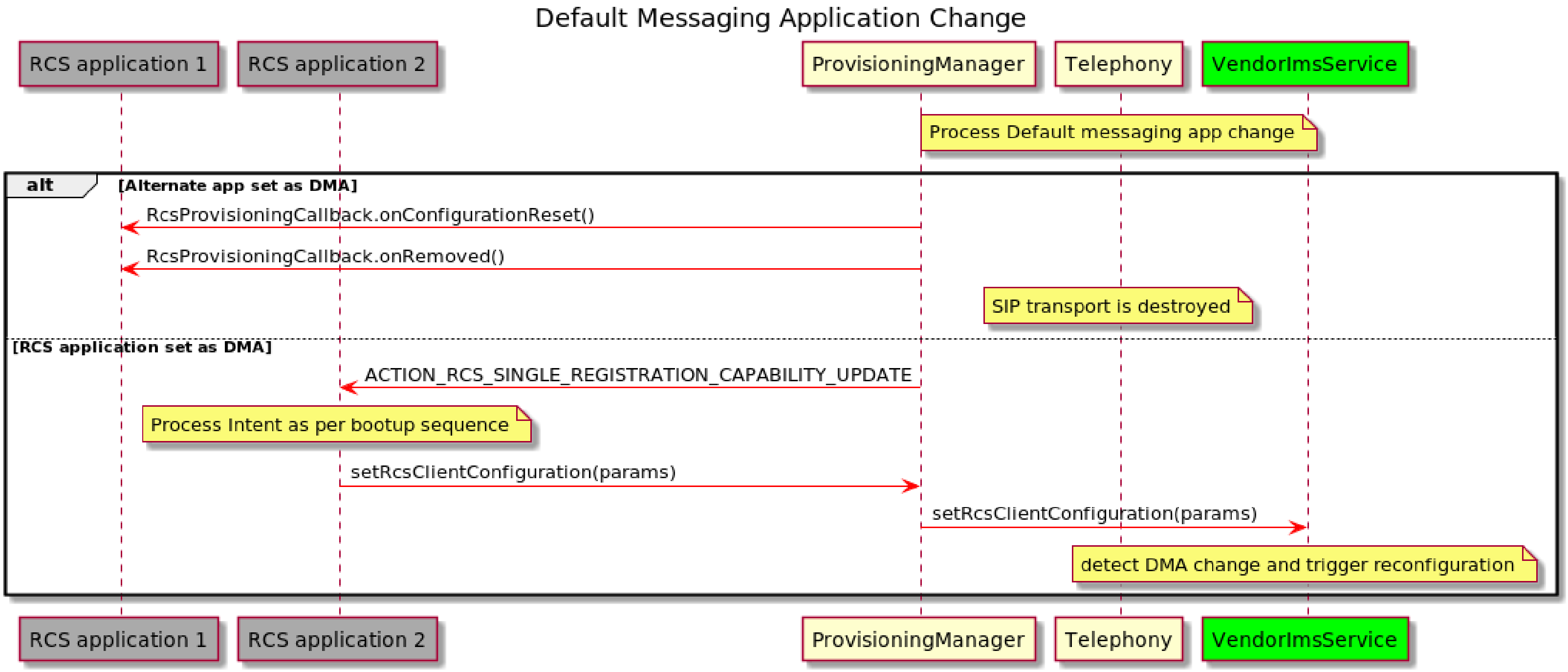
Scenario P.CS.2: RCS provisioning update

In case of a subsequent provisioning update, telephony will update the default messaging application with the latest XML on the registered callback.


Provisioning update for RCS configuration
CarrierSettingsApp
RCS application
ProvisioningManager
Telephony
VendorImsService
modem
alt
[Provisoning mechanism = ACS]
notifyRcsAutoConfigurationReceived(xml)
notifyRcsAutoConfigurationReceived(xml)
[Provisoning mechanism = entitelment server]
notifyRcsAutoConfigurationReceived(xml)
notifyRcsAutoConfigurationReceived(xml)
notifyRcsAutoConfigurationReceived(xml)
store XML and parsed values in persistent Storage
RcsProvisioningCallback.onRcsConfigChanged(xml)
CarrierSettingsApp
RCS application
ProvisioningManager
Telephony
VendorImsService
modem

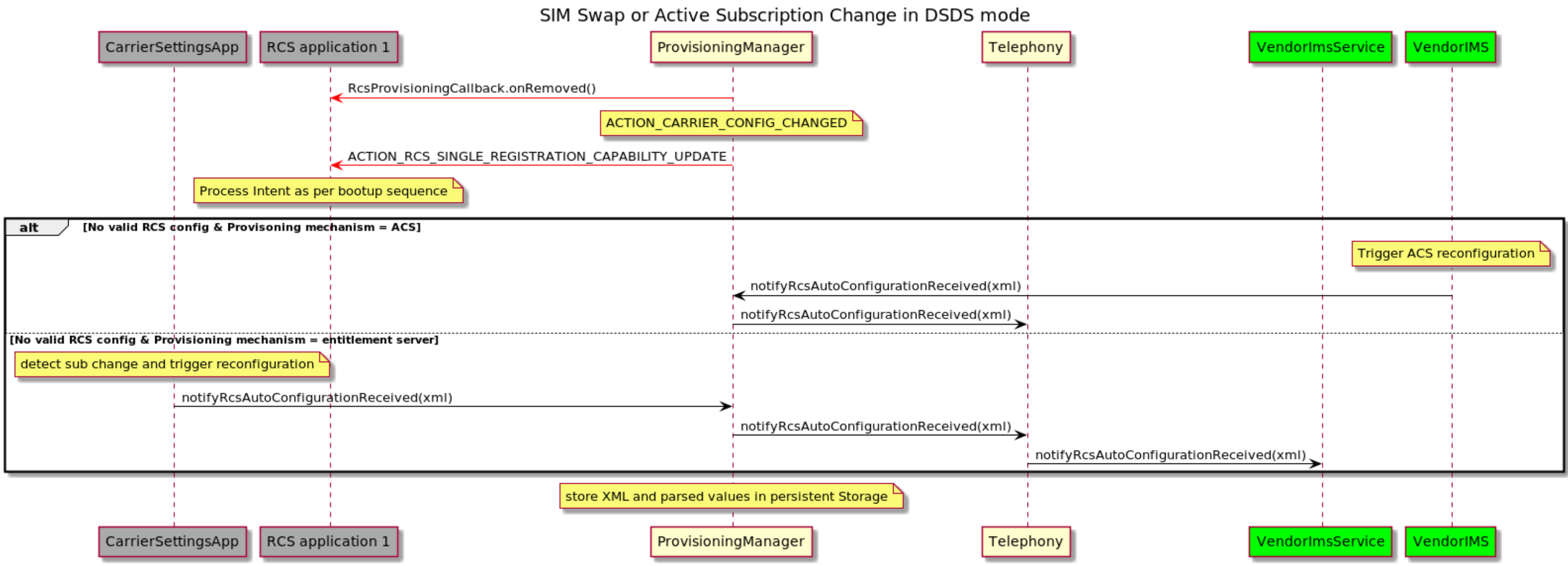
Execution flow for corner scenarios is discussed below.

Scenario P.EC.1: User changes the default messaging application

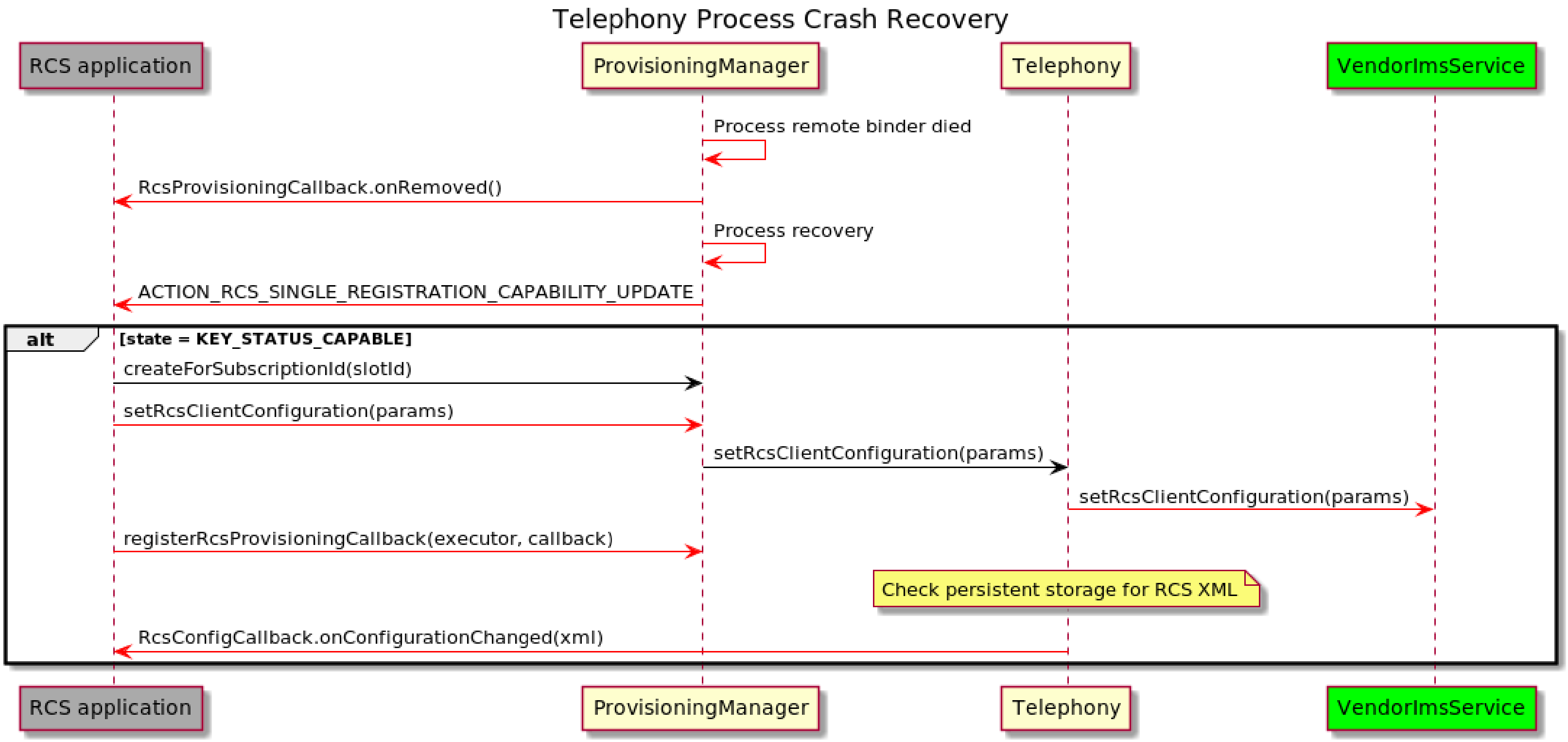

Default Messaging Application Change
RCS application 1
RCS application 2
ProvisioningManager
Telephony
VendorImsService
Process Default messaging app change
alt
[Alternate app set as DMA]
RcsProvisioningCallback.onConfigurationReset()
RcsProvisioningCallback.onRemoved()
SIP transport is destroyed
[RCS application set as DMA]
ACTION_RCS_SINGLE_REGISTRATION_CAPABILITY_UPDATE
Process Intent as per bootup sequence
setRcsClientConfiguration(params)
setRcsClientConfiguration(params)
detect DMA change and trigger reconfiguration
RCS application 1
RCS application 2
ProvisioningManager
Telephony
VendorImsService

Scenario P.EC.2: SIM change/Hot swap/Active subscription change

SIM change shall be detected by the provisioning client and the framework. If a valid XML is not available for the active subscription then the provisioning client shall trigger a reconfiguration.

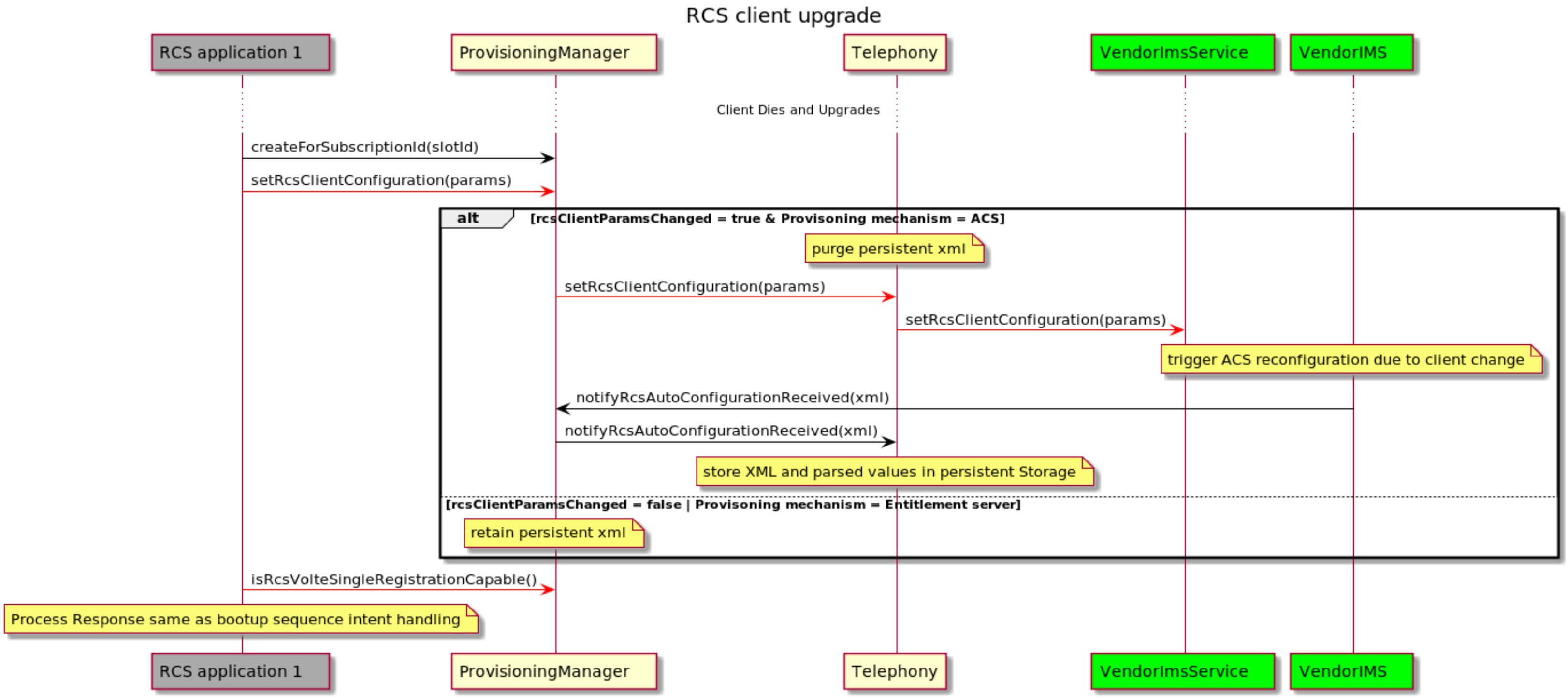

SIM Swap or Active Subscription Change in DSDS mode
CarrierSettingsApp
RCS application 1
ProvisioningManager
Telephony
VendorImsService
VendorIMS
RcsProvisioningCallback.onRemoved()
ACTION_CARRIER_CONFIG_CHANGED
ACTION_RCS_SINGLE_REGISTRATION_CAPABILITY_UPDATE
Process Intent as per bootup sequence
alt
[No valid RCS config & Provisoning mechanism = ACS]
Trigger ACS reconfiguration
notifyRcsAutoConfigurationReceived(xml)
notifyRcsAutoConfigurationReceived(xml)
[No valid RCS config & Provisioning mechanism = entitlement server]
detect sub change and trigger reconfiguration
notifyRcsAutoConfigurationReceived(xml)
notifyRcsAutoConfigurationReceived(xml)
notifyRcsAutoConfigurationReceived(xml)
store XML and parsed values in persistent Storage
CarrierSettingsApp
RCS application 1
ProvisioningManager
Telephony
VendorImsService
VendorIMS

Scenario P.EC.3: Telephony process crash recovery

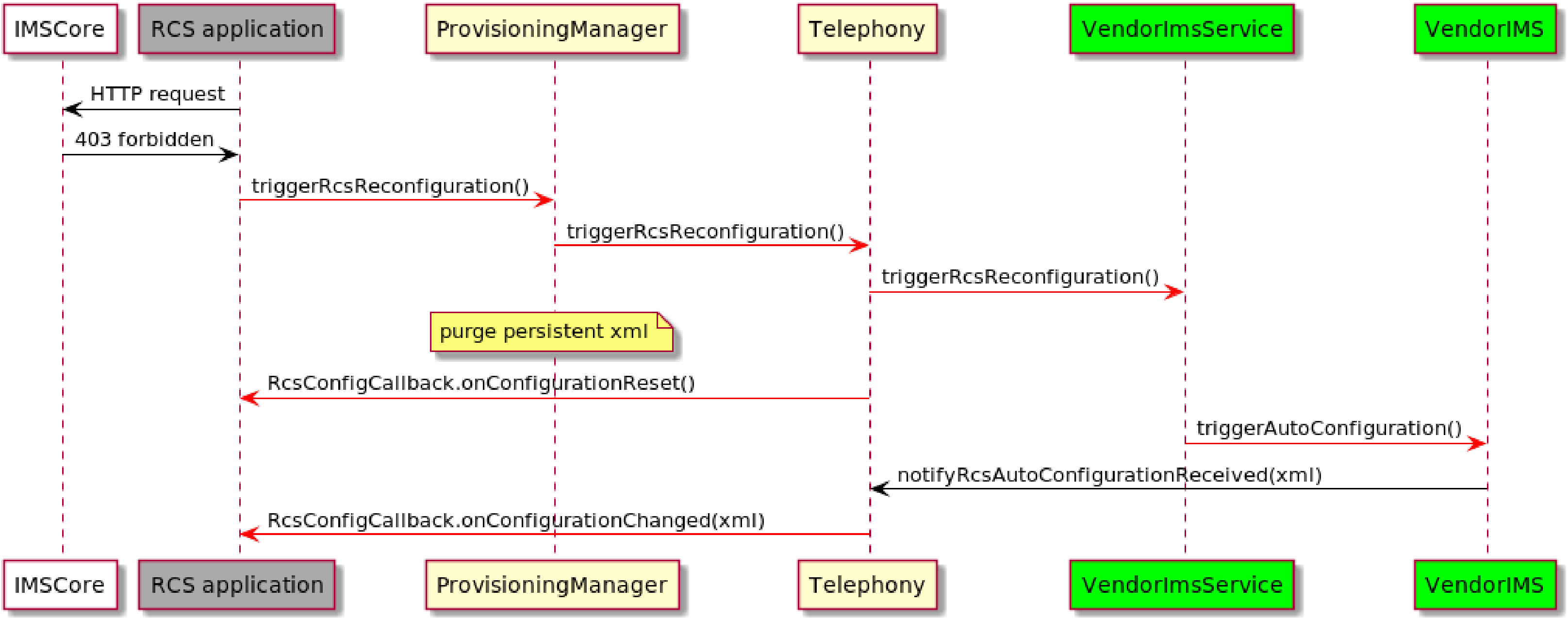

Telephony Process Crash Recovery
RCS application
ProvisioningManager
Telephony
VendorImsService
Process remote binder died
RcsProvisioningCallback.onRemoved()
Process recovery
ACTION_RCS_SINGLE_REGISTRATION_CAPABILITY_UPDATE
alt
[state = KEY_STATUS_CAPABLE]
createForSubscriptionId(slotId)
setRcsClientConfiguration(params)
setRcsClientConfiguration(params)
setRcsClientConfiguration(params)
registerRcsProvisioningCallback(executor, callback)
Check persistent storage for RCS XML
RcsConfigCallback.onConfigurationChanged(xml)
RCS application
ProvisioningManager
Telephony
VendorImsService

Scenario P.EC.4: RCS client upgrade or RCS application crash recovery

In cases when the RCS application crashes and recovers from it, or when its updated via playstore, the application must provide its latest RCS client parameters to telephony.
When the default messaging application specific parameters change, `ProvisioningManager` shall detect condition and purge the stored xml. If it has a registered application callback, it shall inform the application regarding the configuration reset.


RCS client upgrade
RCS application 1
ProvisioningManager
Telephony
VendorImsService
VendorIMS
Client Dies and Upgrades
createForSubscriptionId(slotId)
setRcsClientConfiguration(params)
alt
[rcsClientParamsChanged = true & Provisoning mechanism = ACS]
purge persistent xml
setRcsClientConfiguration(params)
setRcsClientConfiguration(params)
trigger ACS reconfiguration due to client change
notifyRcsAutoConfigurationReceived(xml)
notifyRcsAutoConfigurationReceived(xml)
store XML and parsed values in persistent Storage
[rcsClientParamsChanged = false | Provisoning mechanism = Entitlement server]
retain persistent xml
isRcsVolteSingleRegistrationCapable()
Process Response same as bootup sequence intent handling
RCS application 1
ProvisioningManager
Telephony
VendorImsService
VendorIMS

Scenario P.EC.5: Reconfiguration triggered by the RCS application due to 403 forbidden


ACS reconfiguration due to HTTP error
IMSCore
RCS application
ProvisioningManager
Telephony
VendorImsService
VendorIMS
HTTP request
403 forbidden
triggerRcsReconfiguration()
triggerRcsReconfiguration()
triggerRcsReconfiguration()
purge persistent xml
RcsConfigCallback.onConfigurationReset()
triggerAutoConfiguration()
notifyRcsAutoConfigurationReceived(xml)
RcsConfigCallback.onConfigurationChanged(xml)
IMSCore
RCS application
ProvisioningManager
Telephony
VendorImsService
VendorIMS

After triggering reconfiguration due to 403 forbidden error, the `ProvisioningManager` will clean up the stored XML and notify the application. When the new XML is available, application shall be notified via the previously registered callback.

Scenario P.EC.7: ACS triggered due to factory reset/ MR upgrade

These cases trigger an application processor restart. RCS application, telephony and IMSservice will all be restarted. ImsService will detect that software is upgraded or reset and it shall trigger reconfiguration with the ACS client. Similarly, carrier configuration OEM application will trigger reconfiguration with the entitlement server. Remaining execution sequence would be the same as the boot up flow depicted in scenario P.CS.1.

SIP Message Transport

In order to support "single registration" in AOSP, there must be one central authority managing the IMS registration and either implementing all IMS functionality for the device or providing the interfaces for another application to implement a subset of the IMS functionality. This API surface allows the device `ImsService` to open up a SIP Message transport layer between the device `ImsService` and another sufficiently privileged application (more information in the permissions section below) for one or more IMS feature tags. This will allow an RCS messaging application to maintain its own RCS stack and send and receive SIP messages to and from the carrier's network using the established IP address and port established during the registration process of the device `ImsService`. Figure 3.3.1 below shows the high level description of how this is designed. It also allows the RCS application the flexibility to switch between dual registration and single registration as needed, depending on the carrier/provisioning information.

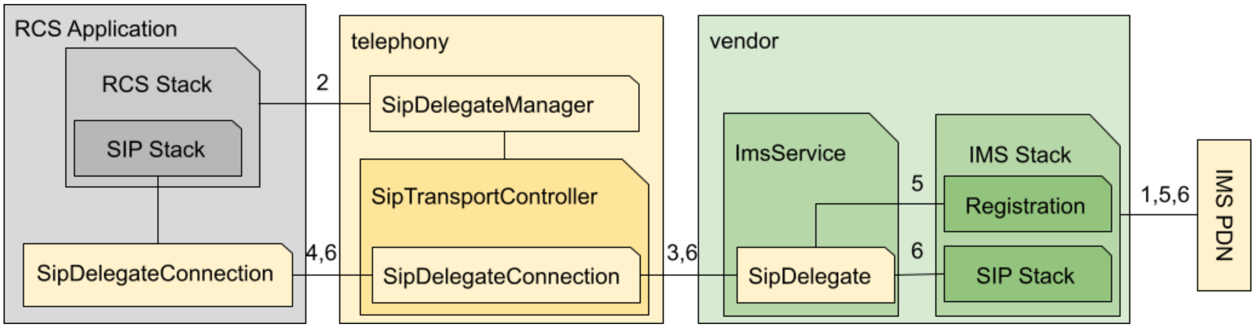


Figure 3.3.1: High level diagram illustrating the process of creating a SipDelegate in AOSP, which connects a remote RCS application to a device `ImsService` in order to forward SIP messages across the transport layer and through the socket set up for SIP message traffic. (1) Device IMS stack completes initial IMS registration. (2) RCS application boots and requests a `SipDelegate` for a set of RCS feature tags. (3) If allowed, the `SipDelegateService` creates the `SipDelegate` and then (4) returns the `SipDelegateConnection` to the RCS application, which connects the RCS application to the `SipDelegate` logically. (5) Telephony requests that the `ImsService` modifies the IMS registration to include the new feature tags on the network. The `SipDelegateConnection` is now free to forward SIP messages to the `SipDelegate` (6) and the `SipDelegate` can route the responses back to the RCS application. The `SipDelegate` can also route new out-of-dialog messages associated with the RCS feature tags to the RCS application.

We will first look at how routing will work in the device `ImsService` as well as how outgoing messages from a remote application will be validated to ensure the SIP messages are not malformed or invalid. We will then touch on how IMS registration should work in this design and then finally define the API surface for both the device ImsService and the RCS application that will be using this API. We will also be providing examples of expected behavior for all of these cases as required.

Routing

At a basic level, RCS applications must create and respond to SIP transactions, which are each a series of SIP requests and responses, in order to perform some type of procedure. But first, a step back to review some SIP specific terminology (RFC 3261):

- **Transaction** - a SIP transaction consists of a single request and any responses to that request, which include zero or more provisional responses and one or more final responses. In the case of a transaction where the request was an INVITE (known as an INVITE transaction), the transaction also includes the ACK only if the final response was not a 2xx response. If the response was a 2xx, the ACK is not considered part of the transaction.
    - For routing purposes, a transaction can be thought of as a group of SIP request and response messages, which all contain the same *via branch* parameter.
- **Dialog** -A dialog is identified at each user agent with a *dialog ID*, which consists of a *Call-ID* value, a local tag and a remote tag.
    - For routing purposes, a dialog contains multiple transactions, all of which contain the same *Call-ID*.
- **Session** - A session (audio, video,...) is started, which can contain multiple active dialogs between remote user agents and are part of the same call.

Although responses of outgoing SIP dialogs have the ability to be easily routed back to the application that started the dialog, it is a much more involved process for the ImsService to figure out which RCS application an incoming SIP message should be sent over.

> **Note**: RCC 59 requires that there should only be one RCS messaging client active at a time, so it is safe to assume there is no way to differentiate between clients on a more granular basis than per-feature for an incoming out-of-dialog request (such as an INVITE).

There have been multiple proposed designs based on the level of detail and control we wish the AOSP telephony platform to have when handling these messages. For example, the spectrum ranges from SIP transaction granularity, where the RCS application requests to start a SIP transaction or receive an incoming SIP transaction to a loose per-feature granularity, where a send/receive message transport level API is defined and the ImsService inspects the incoming/outgoing traffic to route the incoming messages properly. We are taking the **transport only** approach in this design, where incoming messages are routed to the correct application in the `ImsService` and outgoing messages are sent to the `ImsService` directly, with a small validation layer in between.

Incoming Message Routing

For incoming messages, there exists the problem of routing the initial SIP requests from the network, where there is no established transaction or dialog associated with them yet. In order to route these messages correctly, the ImsService can reference the *Contact* header, which should also specify feature tags (See [RFC 3840](), Section 9). This gives information about the message and which features it is for.

For incoming requests that are already in dialog or for responses to requests from the application, the *Call-ID* or the *via branch* parameters can be used to determine the correct routing.

Figure 3.3.2 below illustrates the proprietary routing layer used to route SIP messages between internal components (for example MMTEL) and remote applications.

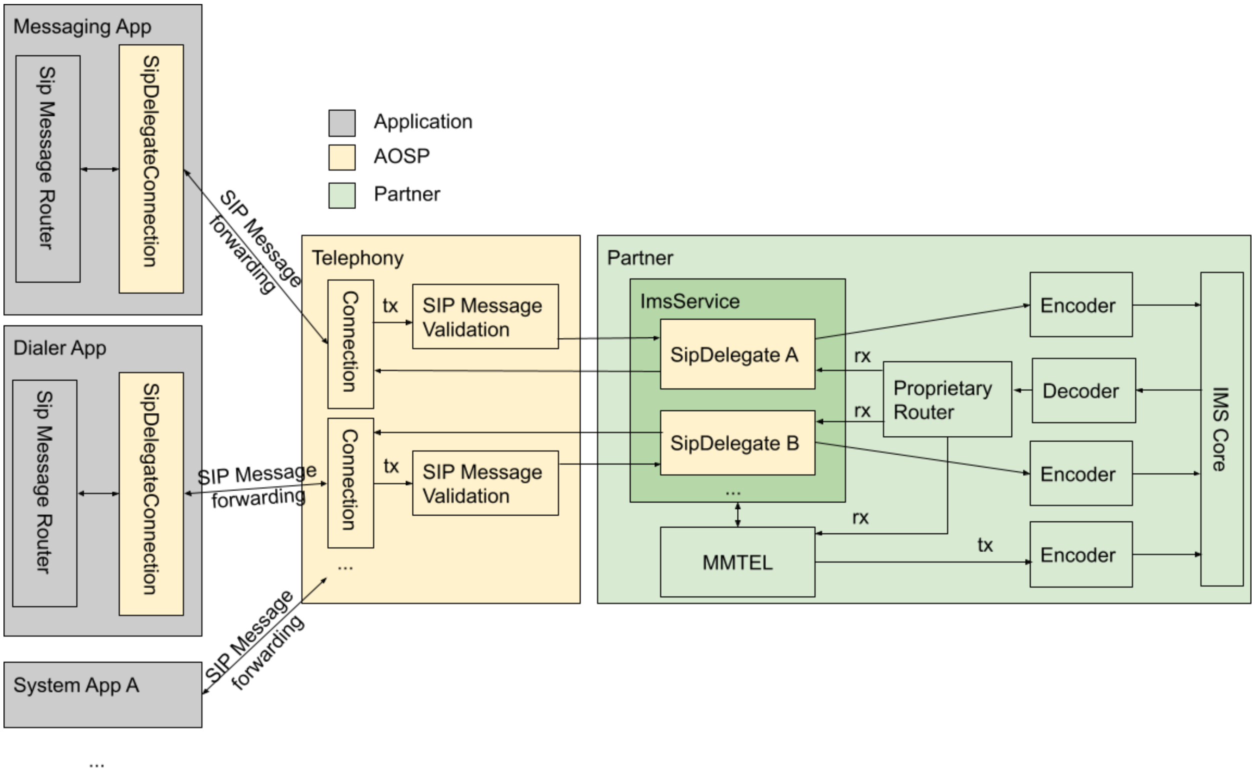


Figure 3.3.2: An additional verification layer helps verify outgoing SIP messages to ensure that they are following carrier and specification requirements.

Outgoing Message Validation

In order for the platform to be able to verify the SIP messages, the SIP messages should be provided to the platform using a hybrid structure:

```
class SipMessage {
  String startLine;
  String header;
  // May be encrypted/encoded
  byte[] content;
}
```

This allows both the framework and the `ImsService` implementation to verify the SIP message header in order to detect malformed or incorrect messages from being sent.

Since AOSP would allow the one or more applications to send SIP messages onto the carrier IMS core network, AOSP must ensure that the application is authorized to send the data that it is generating. This would only need to happen for outgoing SIP messages, since incoming SIP messages are being generated by the carrier IMS core network and have already been validated. AOSP will perform the following validation checks before allowing a SIP message to be sent to the `ImsService`:

1. Ensure the application can only send new out-of-dialog requests for the feature tags that are currently provisioned for by the carrier. Carrier provisioning XML, carrier provided AutoConfig XML, or the preprovisioned data in `CarrierConfigManager` could be used to get the allowed feature tag list and listen to changes to this list (see the provisioning section for more information on how this information is delivered).
2. Use the decoded header to perform data validation:
    a. Contact header - ensure that the feature tag added in the header is allowed for the delegate.
3. Sanity checking and sanitization of messages:
    a. Ensure the message does not contain any invalid characters that would cause UTF-8 encoding to fail
4. Filter out the following SIP message requests: REGISTER, OPTIONS, PUBLISH.
5. Filter out SIP SUBSCRIBE requests if they contain the “Event:presence” header.

This is an example of the first rules that will be implemented, more rules may be added in AOSP as needed. Once all the validation checks pass, AOSP SIP transport can now construct the SIP message using these headers and the other optional headers and body to send to the IMS stack. If any of the validations fail, the outgoing SIP message will be rejected by the SIP transport layer.

IMS Registration

Another point to consider with the SIP transport APIs is how IMS registration changes are handled. The API is designed to allow for multiple appropriately privileged applications to access the SIP Transport API and provide the implementation for a subset of the features defined in the RCS Universal Profile document. For example, the following configuration is possible today (although devices are only expected to support the default messaging application for RCS messaging through this API for Android S):

RCS Feature	Permission	Condition to Grant Access
RCS Messaging	Signed System/Carrier Application	Default Messaging Application
User Capability Exchange	System Signed Application	OEM installed UCE service
Enriched Calling	System/Carrier Signed Application	Default Dialer Application

This list could expand as more features are defined with application roles that implement these features.

We do not want one IMS registration happening for each of these applications during boot up or every time there is a PDN or provisioning change. Instead, the application should “associate” the features that they want to provide once after boot and the framework will manage the registration associated with those features based on provisioning or PDN changes. The application should not destroy the delegate unless they will no longer support single registration, as that will cause an unnecessary IMS registration change.

When a `SipDelegate` is destroyed, the IMS registration will need to be modified to include the new feature tags associated with that `SipDelegate`. In order to limit the number of back-to-back IMS registrations that may occur if there are multiple RCS applications registering `SipDelegates`, AOSP telephony will throttle the IMS registration change triggers to the `ImsService`. After the SipDelegate is destroyed, a configurable timer with a default of 1 seconds will begin, which will batch together any other `SipDelegate` requests beginning at the same time. Once the registration has been triggered, a configurable throttling timer with a default of 5 seconds will begin, which will limit the number of IMS registration triggers to a minimum time. This does not guarantee that the vendor ImsService will not modify the IMS registration due to independent factors, but it does throttle the potential IMS registration modifications due to RCS features changing.

> **Note**: Some carriers require that MMTEL and RCS features are registered at the same time and put strict requirements on when/how these tags should be registered as well as how fast the IMS registration should occur after boot. It is up to the partner `ImsService` implementation to ensure that the correct set of feature tags are registered once provisioning completes even if there are no `SipDelegates` connected yet. Once the device boots, AOSP will create the `SipDelegate` and trigger registration as soon as possible, but it is completely dependent on the time it takes to start the SMS application and receive the first request to create a `SipDelegate`. The vendor ImsService can either delay the registration of MMTEL and RCS features by some amount after boot and then register or preemptively register for MMTEL and provisioned RCS features and generate a new IMS registration later if the expected set of RCS features does not match the anticipated feature set.

IMS Registration over INTERNET PDN

There are some carriers that also require that the IMS stack register over the INTERNET PDN in some scenarios. This is still considered a single registration scenario and the device `ImsService` will still be in charge of setting up the IMS registration over the INTERNET PDN when required. The device `ImsService` will also still allow RCS applications to access the same single IMS registration via the SIP Transport API as well as handle RCS UCE. The RCS application will be able to identify when this transition has occurred using the `SipDelegateConfiguration` and use this information to perform PDN specific data setup procedures as required by carriers.

Permissions & Feature Tag Filtering

At this time, the Android API will only support single registration for a “system” application, which has been signed by the OEM and is one of the preinstalled, updatable applications that an OEM provides. This gives the application the ability to access privileged permissions that are not typically accessible to 3rd party applications that are downloaded onto the device. Please see the security section for more details on the general security model of an application trying to access these APIs.

In order to open a `SipDelegateConnection` to the device `ImsService`, an application must meet the following prerequisites:

1. Declare the `android.permission.CONNECTIVITY_USE_RESTRICTED_NETWORKS` permission in their manifest and grant the permission in the OEM maintained privileged permission whitelist, which allows the application to access the IMS PDN.
2. Fulfill the role required to access the feature tags that have been requested by the application. In Android S, the platform will enforce that all RCS feature tags are **only available to the Default Messaging Application**. See the below table for some examples of common RCS feature tags and their associated role restriction:

Feature	Associated Tags	Role Restriction
Standalone Messaging	+g.3gpp.icsi-ref="urn%3Aurn-7%3A3gppservice.ims.icsi.oma.cpm.msg,urn%3Aurn-7%3A3gppservice.ims.icsi.oma.cpm.largemsg,urn%3Aurn-7%3A3gppservice.ims.icsi.oma.cpm.deferred";+g.gsma.rcs.cpm.pager-large	Default Messaging Application
Chat/Group Chat	+g.3gpp.icsi-ref="urn%3Aurn-7%3A3gppservice.ims.icsi.oma.cpm.session"	Default Messaging Application
File Transfer	+g.3gpp.iari-ref="urn%3Aurn-7%3A3gppapplication.ims.iari.rcs.fthttp, urn%3Aurn-7%3A3gppapplication.ims.iari.rcs.ftsms"	Default Messaging Application
Geolocation PUSH via SMS	+g.3gpp.iari-ref="urn%3Aurn-7%3A3gppapplication.ims.iari.rcs.geosms"	Default Messaging Application
Chatbot	+g.3gpp.iari-ref="urn%3Aurn-7%3A3gppapplication.ims.iari.rcs.chatbot,urn%3Aurn-7%3A3gppapplication.ims.iari.rcs.chatbot.sa";+g.gsma.rcs.botversion="#=1,#=2"	Default Messaging Application
MMTEL	+g.3gpp.iari-ref="urn%3Aurn-7%3A3gpp-service.ims.icsi.mmtel”; video; +g.3gpp.smsip	Telephony Internal
Presence	+g.3gpp.iari-ref="urn:urn-7:3gpp-application.ims.iari.rcse.dp”	Telephony Internal
Carrier Specific	*	Default Messaging Application

Tags		
*	*	Default Messaging Application

Summary: Except for the cases mentioned above, all RCS feature tags are restricted to the Default Messaging Application. Any other application that requests a `SipDelegateConnection` to one or more feature tags will be denied and no `SipDelegate` will be created.

In the future, the specifications for single registration may change and multiple IMS applications may all be granted access to the device `ImsService` in order to provide implementations for subsets of the RCS specifications, as shown in Figure 3.3.2. If the mapping of associated feature tags to allowed application roles changes, we will expand the code to include a dynamic `feature_tag->allowed_roles` mapping.

API Definition

As stated above, this API acts as a transport only and passes through the telephony framework once a SIP delegate has been registered. The API is designed using a partially decoded SIP message in order for the telephony framework to validate that all outgoing messages are being sent for a feature tag that the RCS application is registered for. This first iteration of verification will be improved over time as needed.

Logically, the RCS application and the `ImsService’s` `SipDelegates` will be connected together with a small layer in between for validation (as mentioned above). See Figure 3.3.3.

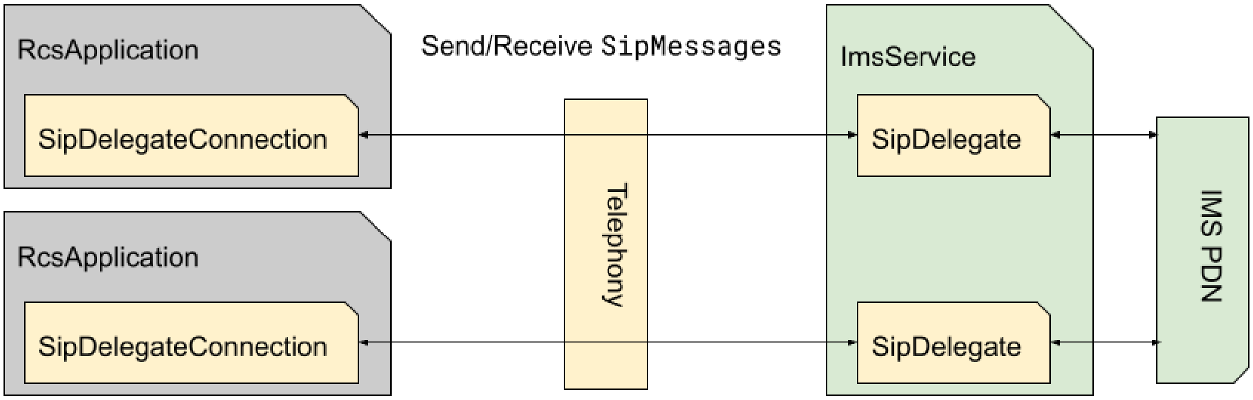


Figure 3.3.3: The logical connection between the `SipDelegateConnection` and the `SipDelegate`.

The SIP messages will be passed over `Binder` using a `Parcelable` class, which contains the partially encoded `SipMessage`. The header is sent decoded in order for the platform to easily access fields that may be necessary for outgoing message verification.

ImsService API Additions

When the framework requests that a `SipDelegate` is created, there will be one `SipDelegate` created per RCS application `SipDelegateConnection` (not per feature tag). This was done for the following reasons:

1. Greater application flexibility with respect to grouping SIP messages.
2. Decreases the management needed in the framework to maintain the state of each remote binder interface associated with a feature tag.
3. Allows vendors to encapsulate implementation specific logic into their `ImsService`.

Figure 3.3.4 below shows the new class structure in the `ImsService`.

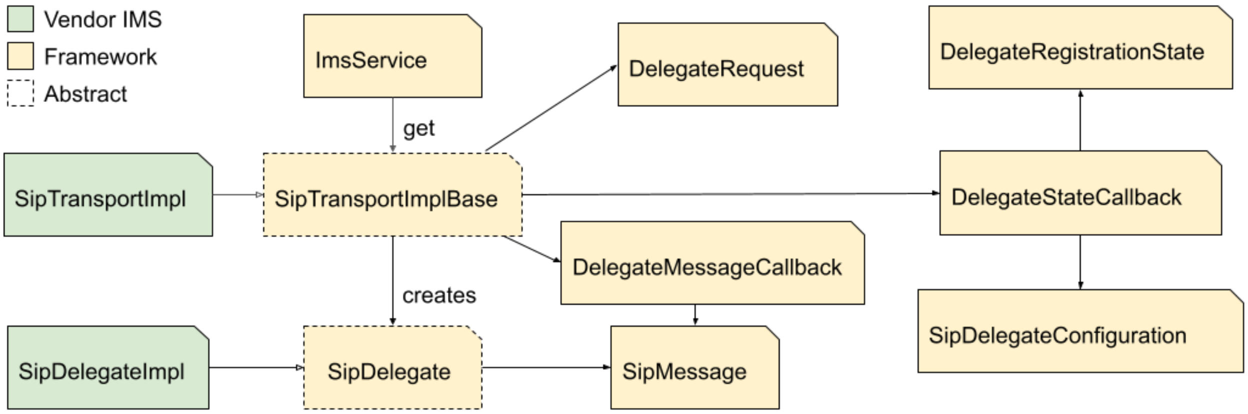


Figure 3.3.4: `SipDelegate` class relationship structure.

In order to support the creation of SipDelegate, the vendor ImsService must implement [SipTransportImplBase](), which is the base class for all SipDelegate management. First, the framework will request that the [`SipDelegate`]() be created in `SipTransportImplBase#createSipDelegate` using a [`DelegateRequest`](), which contains parameters about the request. For example, it currently contains which feature tags it should be created for. Once the `ImsService` has created the `SipDelegate`, it must call `DelegateStateCallback#onCreated`. The `DelegateRequest` will always be made for disjoint feature tag sets, meaning that there should never be a request for multiple `SipDelegates` with the same feature tags. If for some reason this occurs, the duplicate feature tag should be ignored and the tags should be allowed on a first-come first-serve basis.

The [`SipDelegate`]() is an interface used by the framework to send new events from the RCS application used to receive messages from the remote IMS application and acknowledge that a SIP message has been sent successfully. It also defines a `SipDelegate#notifyMessageReceiveError` callback, which is only used in the scenario when the application's `SipDelegateConnection` is unreachable due to the application crashing. In this condition, the framework will eventually call `SipTransportImplBase#destroySipDelegate` on this delegate. The API also provides a `SipDelegate#cleanupSession` method, which is used for routing purposes and allows the application to notify the `SipDelegate` when resources associated with routing that SIP dialog are no longer needed. This is because the `SipDelegate` is not keeping track of the state of the dialog and will not clean up routing resources associated with the SIP dialog automatically. The `SipDelegate#cleanupSession` will also be called by the RCS application after it receives an indication that a feature tag associated with that SIP dialog has been deregistered. This allows the `ImsService` to wait for the ongoing dialogs to be closed before modifying the IMS registration.

The `DelegateStateCallback` is a callback used to communicate state events to the remote IMS application. This includes notifications when the `SipDelegate` has been destroyed as well as when the network IMS registration state has changed for one or more of these features.

The DelegateRegistrationState class is used to communicate registration state changes of each feature tag to the IMS application. Registration changes are the primary way that the `SipDelegate` can communicate a change to the remote IMS application, since it will affect how the remote application responds. There are four states that a feature tag associated with a `SipDelegate` may be in: registered, registering, deregistering, and deregistered. Each feature tag must report one of these four states at all times. Intermediate registration states have been included in order to provide more information to the IMS application. The intermediate registering state exists to let the IMS application know that registration is in progress for UI reasons. The intermediate deregistering state has been included to allow the RCS application to respond to an in-progress deregistration and perform an action before the SipDelegate proceeds with the network registration. Currently, the only action that the RCS application must do is close existing SIP Dialogs for feature tags in the process of deregistration in order for the network IMS registration modification to proceed. For more information on when to report DEREGISTERING or DEREGISTERED for a feature tag can be found in the documentation for DelegateRegistrationState.

The DelegateMessageCallback is used to send incoming SIP messages to the remote application as well as acknowledge when a message has been successfully/unsuccessfully sent.

The SipDelegateConfiguration class is used to send IMS stack configuration information to the IMS application in order to communicate the state of the stack so that the IMS application can correctly generate SIP messages. In order for the `SipDelegate` to communicate the attributes that the remote application will need to construct SIP messages and operate their IMS stack, the `SipDelegate` will call `DelegateStateCallback#onConfigurationChanged`. The `SipDelegateConfiguration` will also contain a version associated with it. All outgoing `SipMessages` sent by the RCS application will also send the `SipDelegateConfiguration` version used to create it in order to avoid race conditions where the `ImsService` has created a new configuration, but the RCS application has constructed and sent a `SipMessage` using the now stale configuration. In this case, the `ImsService` should send the `MessageStateCallback#onMessageSendFailure` callback to the RCS application with error code

`SipDelegateManager#MESSAGE_FAILURE_REASON_STALE_IMS_CONFIGURATION` so that the RCS application can recreate and send the `SipMessage` again using the current `SipDelegateConfiguration`. The RCS messaging application should also use the SipDelegateConfiguration to determine the APN that the MSRP session will be created on (INTERNET vs IMS).

Finally, there has been a change to the ImsRegistrationImplBase class In order to reduce the number of registrations on the network. The framework will call `ImsRegistrationImplBase#updateSipDelegateRegistration` when one or more SIP delegates have changed their supported feature tags. The `ImsService` should not trigger a new IMS registration with the new tags beforehand, as multiple `SipDelegates` may be in the process of modifying their `SipDelegateConnections`. The API also provides the following method `ImsRegistrationImplBase#triggerNetworkReregistration`, which is used in the case that the remote RCS application has received a permanent failure response to a SIP message and the vendor IMS stack should perform a new IMS registration.

Figure 3.3.5 below shows the typical creation case for a new SIP delegate.

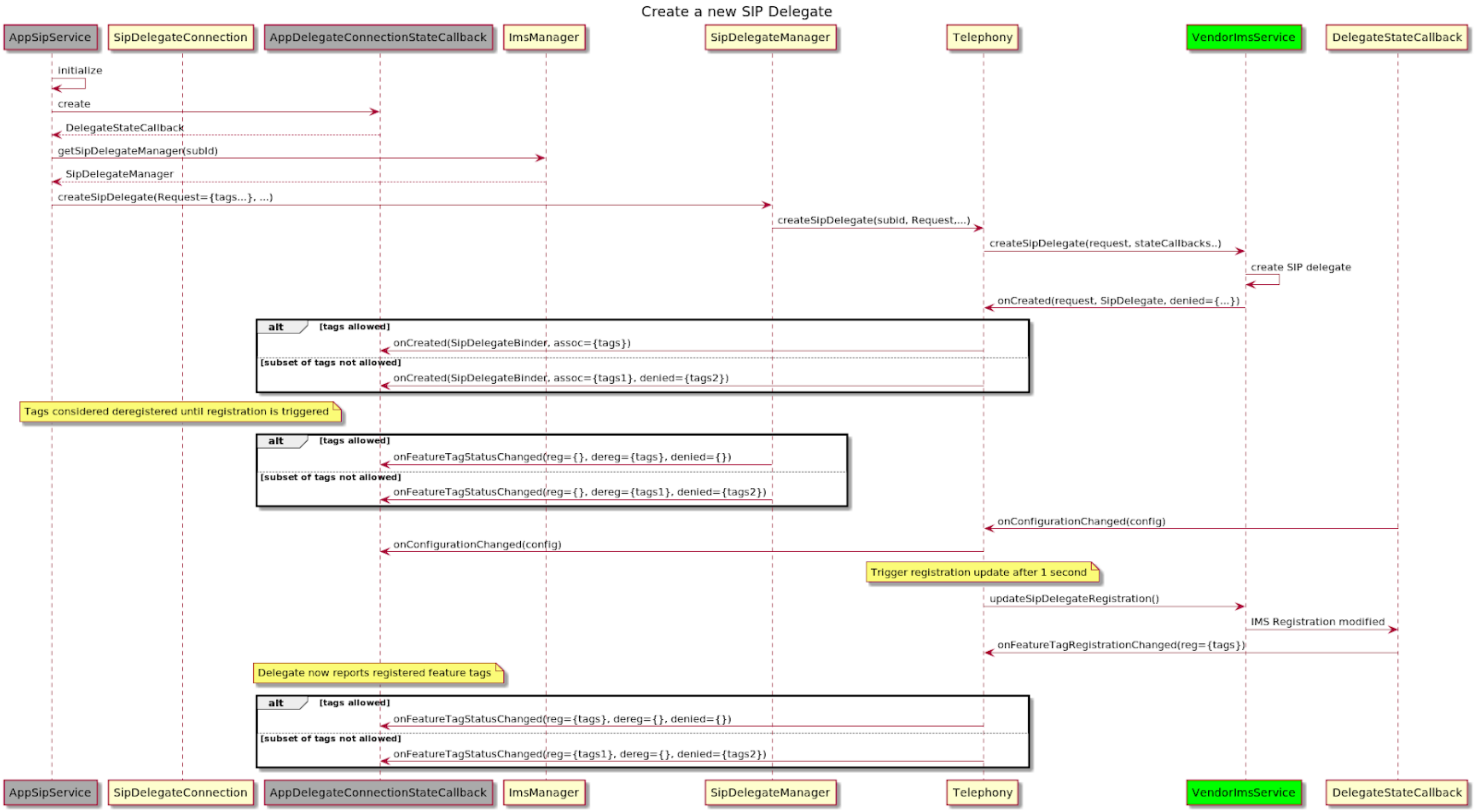

Create a new SIP Delegate
AppSipService
SipDelegateConnection
AppDelegateConnectionStateCallback
ImsManager
SipDelegateManager
Telephony
VendorImsService
DelegateStateCallback
initialize
create
DelegateStateCallback
getSipDelegateManager(subId)
SipDelegateManager
createSipDelegate(Request={tags...}, ...)
createSipDelegate(subid, Request,...)
createSipDelegate(request, stateCallbacks..)
create SIP delegate
onCreated(request, SipDelegate, denied={...})
alt
[tags allowed]
onCreated(SipDelegateBinder, assoc={tags})
[subset of tags not allowed]
onCreated(SipDelegateBinder, assoc={tags1}, denied={tags2})
Tags considered deregistered until registration is triggered
alt
[tags allowed]
onFeatureTagStatusChanged(reg={}, dereg={tags}, denied={})
[subset of tags not allowed]
onFeatureTagStatusChanged(reg={}, dereg={tags1}, denied={tags2})
onConfigurationChanged(config)
onConfigurationChanged(config)
Trigger registration update after 1 second
updateSipDelegateRegistration()
IMS Registration modified
onFeatureTagRegistrationChanged(reg={tags})
Delegate now reports registered feature tags
alt
[tags allowed]
onFeatureTagStatusChanged(reg={tags}, dereg={}, denied={})
[subset of tags not allowed]
onFeatureTagStatusChanged(reg={tags1}, dereg={}, denied={tags2})
AppSipService
SipDelegateConnection
AppDelegateConnectionStateCallback
ImsManager
SipDelegateManager
Telephony
VendorImsService
DelegateStateCallback

Figure 3.3.5: Typical SIP delegate creation.

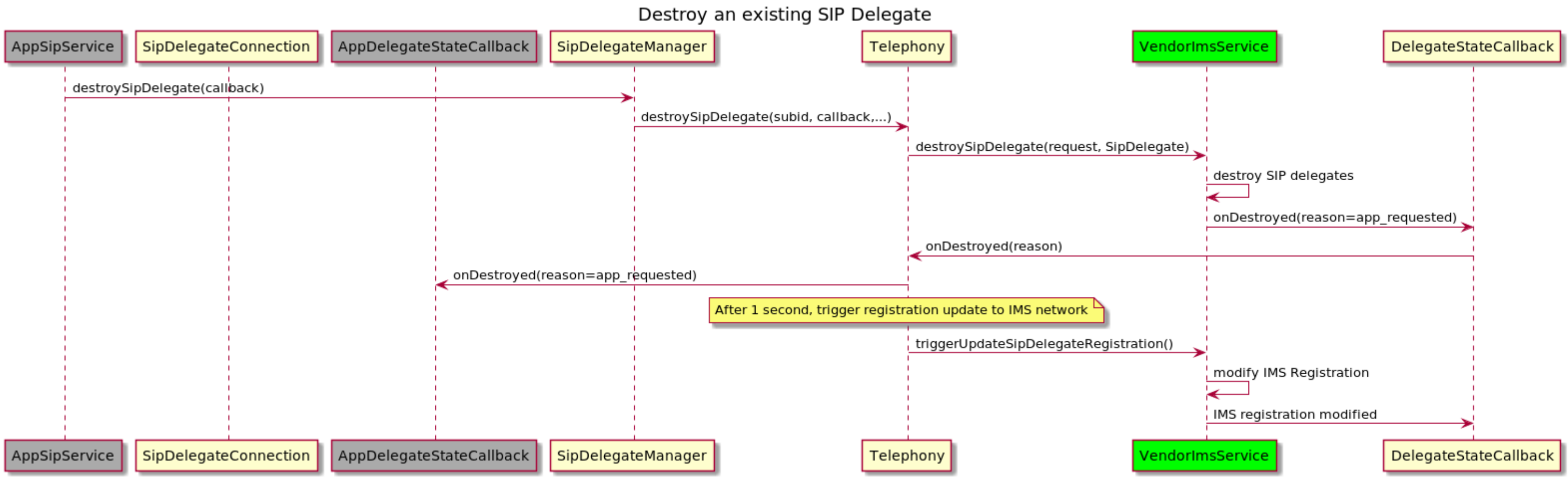
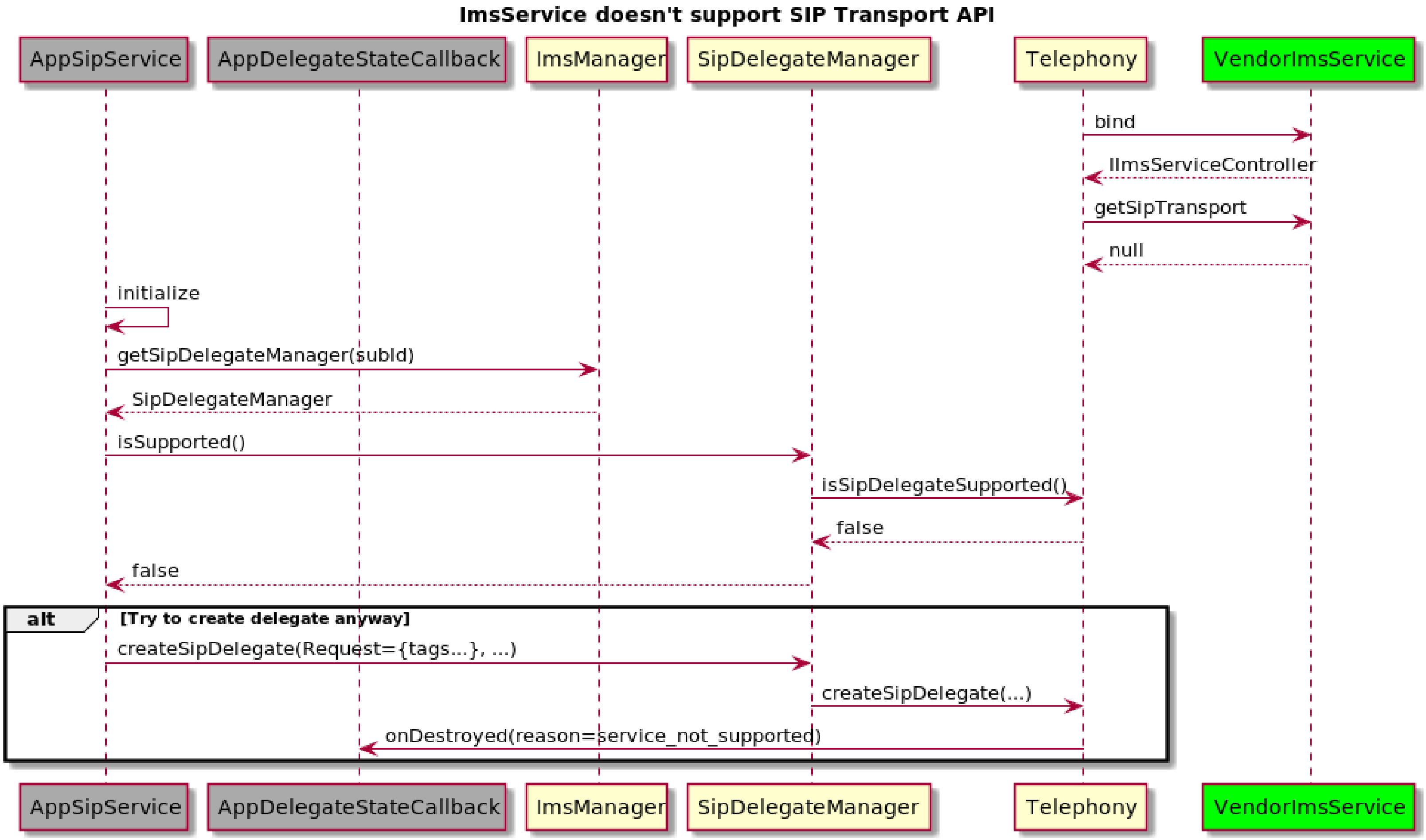

Destroy an existing SIP Delegate
AppSipService
SipDelegateConnection
AppDelegateStateCallback
SipDelegateManager
Telephony
VendorImsService
DelegateStateCallback
destroySipDelegate(callback)
destroySipDelegate(subid, callback,...)
destroySipDelegate(request, SipDelegate)
destroy SIP delegates
onDestroyed(reason=app_requested)
onDestroyed(reason)
onDestroyed(reason=app_requested)
After 1 second, trigger registration update to IMS network
triggerUpdateSipDelegateRegistration()
modify IMS Registration
IMS registration modified
AppSipService
SipDelegateConnection
AppDelegateStateCallback
SipDelegateManager
Telephony
VendorImsService
DelegateStateCallback

Figure 3.3.6: Destruction of a SIP delegate.

On older devices, the vendor ImsService may not be able to support the SIP Transport API at all. To determine if a device supports the SIP Transport API, an application can call `SipDelegateManager#isSupported`. If it is supported on the device, an application can then create a `SipDelegate`. If it is not supported and the device tries to create a `SipDelegate` anyway, `DelegateStateCallback#onDestroyed` will be called immediately. In the case that the vendor ImsService supports the SIP transport API, but the carrier configuration set by Android disables single registration for the carrier, then this will act in the same way. The application can then listen to the `CARRIER_CONFIG_CHANGED` intent to listen for `CarrierConfigManager#KEY_RCS_SINGLE_REGISTRATION_SUPPORTED_BOOL` changes. More information is available in the provisioning section. See Figure 3.3.7 below for more details.



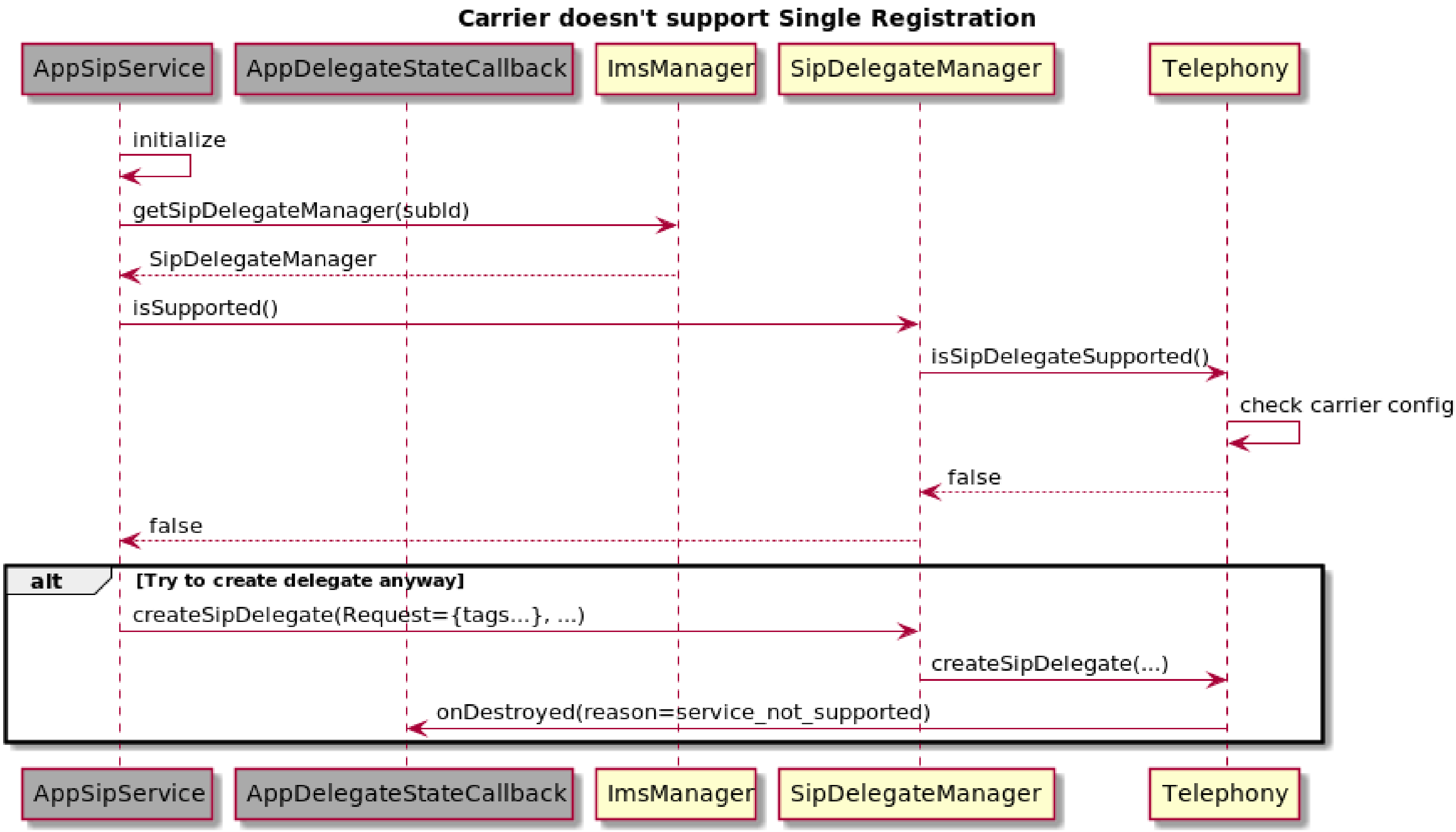


Figure 3.3.7: (top) A device that doesn't support the API. (bottom) A carrier is configured to not support single registration in the Android carrier configuration.

Once the `SipDelegate` has been created, messages can then be sent and received over this interface, see Figure 3.3.8 below. The case where a SIP dialog has ended is also shown below in Figure 3.3.9.

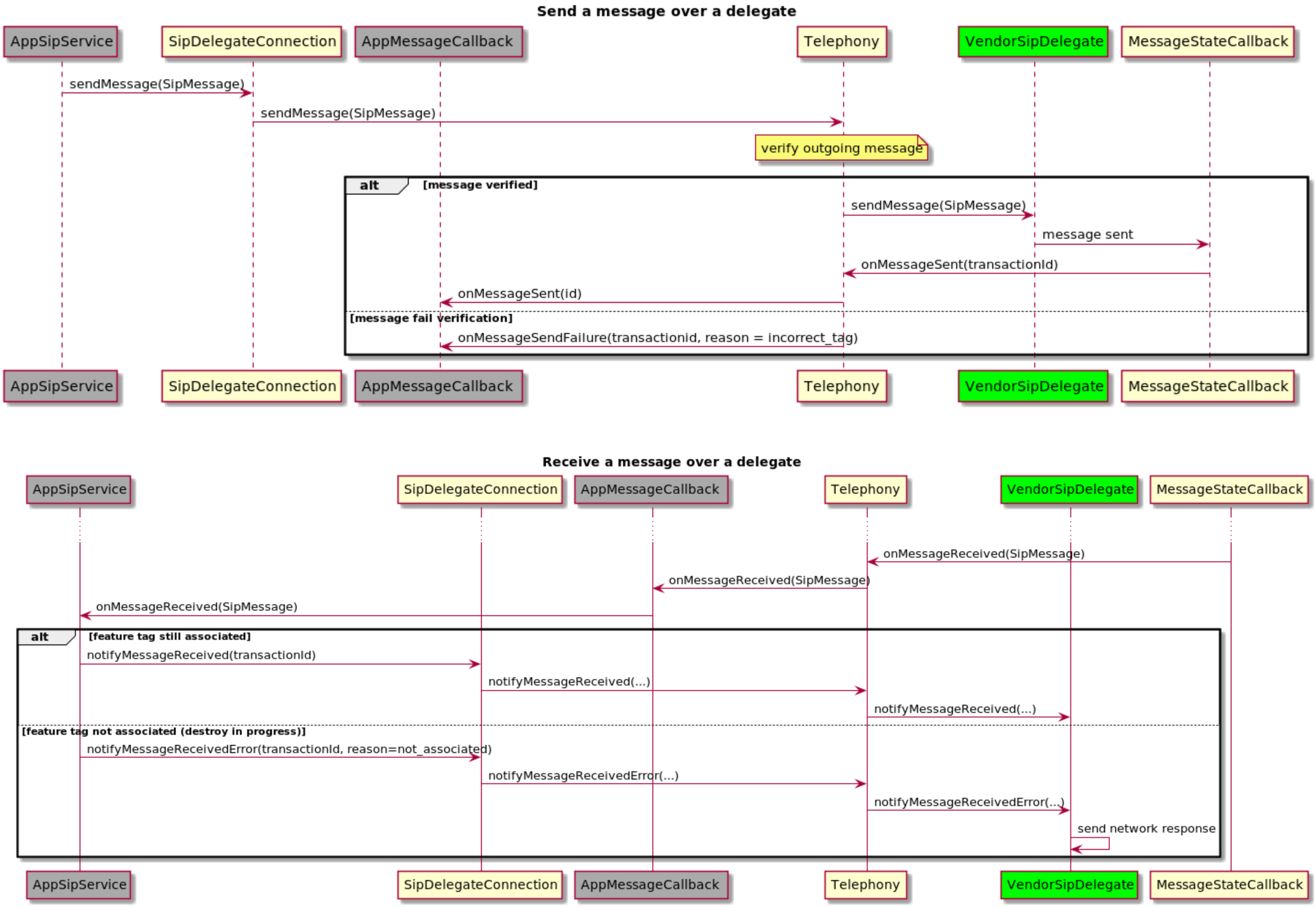

Send a message over a delegate
AppSipService
SipDelegateConnection
AppMessageCallback
Telephony
VendorSipDelegate
MessageStateCallback
sendMessage(SipMessage)
sendMessage(SipMessage)
verify outgoing message
alt
[message verified]
sendMessage(SipMessage)
message sent
onMessageSent(transactionId)
onMessageSent(id)
[message fail verification]
onMessageSendFailure(transactionid, reason = incorrect_tag)
Receive a message over a delegate
onMessageReceived(SipMessage)
onMessageReceived(SipMessage)
onMessageReceived(SipMessage)
alt
[feature tag still associated]
notifyMessageReceived(transactionId)
notifyMessageReceived(...)
notifyMessageReceived(...)
[feature tag not associated (destroy in progress)]
notifyMessageReceivedError(transactionId, reason=not_associated)
notifyMessageReceivedError(...)
notifyMessageReceivedError(...)
send network response

Figure 3.3.8: (top) Message send procedure, (bottom) message receive procedure.

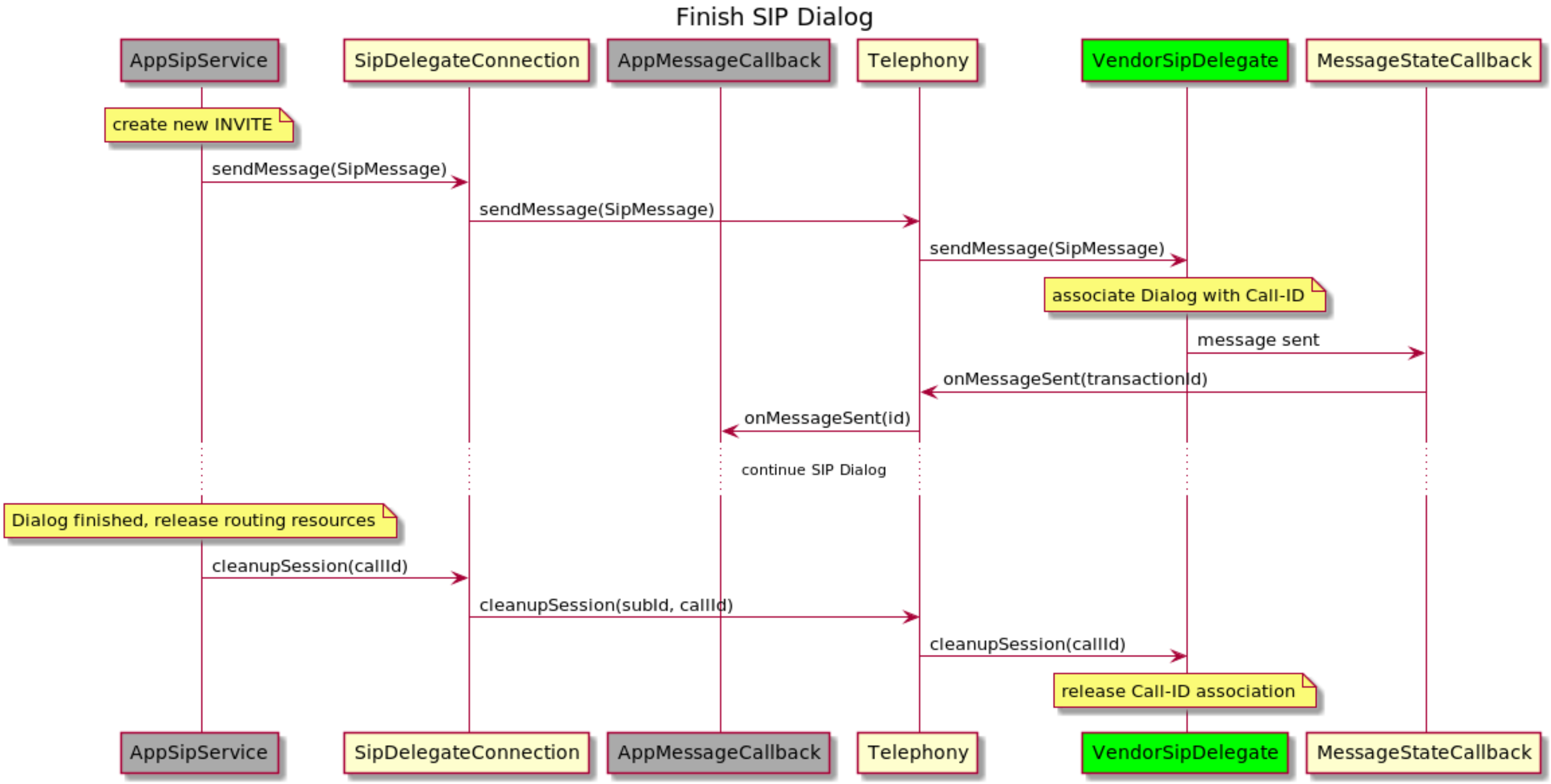


Figure 3.3.9: Closing an existing SIP dialog.

RCS Application APIs

The following APIs are implemented by telephony and used by a remote RCS application to create and destroy `SipDelegates` on the ImsService as needed.

Manager API definitions

The RCS application will interface with the telephony framework using the `ImsManager` API to create a `SipDelegateManager`. This will allow the RCS application to request a `SipDelegateConnection` to send and receive `SipMessage`s over the transport interface.

The `ImsManager` class will be extended to include a new method to retrieve a `SipDelegateManager` on a per-subscription basis. See the section on multi-SIM devices below for how DSDS will be supported.

The `SipDelegateManager` will be an interface defined by telephony and used by RCS applications to create and destroy `SipDelegateConnection`s, which can be created using the DelegateRequests and destroyed at a later time. It will also define the various status/failure messages used by the ImsService and RCS applications.

SipDelegateConnection Setup

When designing the SipDelegateConnection API, special consideration went into making sure that this API handles the following cases:

- Requesting a SipDelegateConnection for one or more features.
- Minimizing the number of IMS registration operations that occur on the network by decoupling registration from the creation and destruction of the SipDelegates. This also handles race conditions for receiving INVITEs during/after the deregistration and destruction procedure has started.

Figure 3.3.10 below shows the class level interfaces that the RCS application will use to set up this SIP delegate on the vendor ImsService.

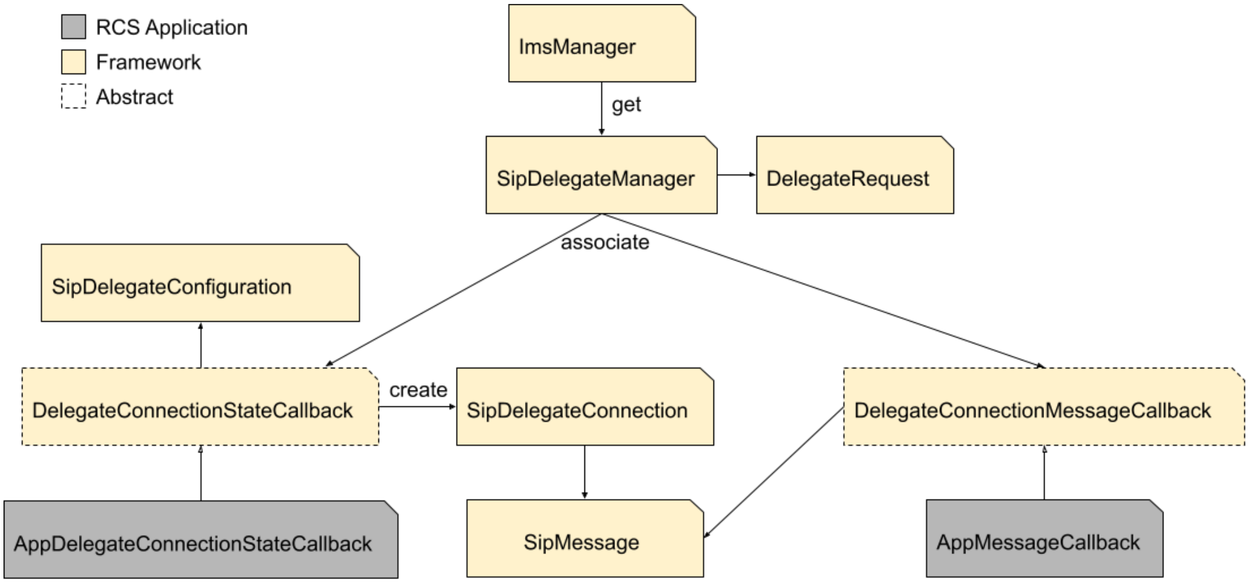


Figure 3.3.10: New RCS Application API class definitions

The SipDelegateManager allows an RCS application to create and modify SipDelegateConnections using a DelegateRequest, which contains the parameters that the SipDelegate will need to set up a connection. This decouples the expected parameters of the SipDelegate from the status of the SipDelegate (which feature tags are enabled and registered).

To create a SipDelegateConnection, a DelegateConnectionStateCallback must be passed in. When the SipDelegateConnection has been created and linked to a remote SipDelegate, the SipDelegateConnction will be passed to the IMS application using the DelegateConnectionStateCallback#onCreated API. After, DelegateConnectionStateCallback#onFeatureTagStatusChanged is used to notify the listener of the feature tags that are associated with the SipDelegateConnection as well as which ones were denied and the reason for the denial. DelegateConnectionStateCallback also provides the current registration state of each of the feature tags. This allows the RCS application to detect if there has been a provisioning change, which has caused an existing feature tag to become unavailable for the creation of new outgoing SIP dialogs.

After a request to create a SipDelegate has been sent, the setup occurs on the ImsService and the resulting SipDelegateConnection is passed back asynchronously to the requesting application. This is because the setup of this connection can take an arbitrary amount of time depending on the state of telephony and the ImsService. The initial associated feature tags will be set on the SipDelegateConnection and any subsequent changes will be notified to the application via the onFeatureTagStatusChanged method. Finally, if the SipDelegateManager#destroySipDelegate method is called, the delegate will still be active for a short period of time until the onDestroyed method has been called. At that point, the application can safely destroy the state related to these objects.

- Note: after destroySipDelegate is called, there will be a short window of time where incoming messages can still be routed here until the new registration to the network is complete. If this occurs, the message should still be handled by the SipDelegateConnection. The transport will still be available until onDestroyed is called on the SipDelegateConnection.

The RCS application will pass in a DelegateConnectionMessageCallback class as part of the SipDelegateManager#createSipDelegate method, which will handle incoming messages as well as the response from the framework for any outgoing messages sent. Any failures received at this layer are verification errors in the framework or network connectivity errors, it does not contain any errors related to SIP messaging, as that will come in the DelegateConnectionMessageCallback#onMessageReceived method.

The connection to the remote SipDelegate, which is used to get the properties of this delegate as well as send messages or notify the ImsService that messages have been received or that there was an error receiving the message. The RCS application should not rely on the ImsService to send any messages on the behalf of this RCS application. The only exception here is the case where a new incoming message was received after the application called the `destroySipDelegate` method. In this case, the application should notify the `ImsService` of the error using `SipDelegateConnection#notifyMessagedReceiveError(transactionId, MESSAGE_FAILURE_REASON_TAG_NOT_ASSOCIATED_WITH_APP)`. This will notify the `ImsService` that the tag is in the process of destroying the connection and it should handle the response.

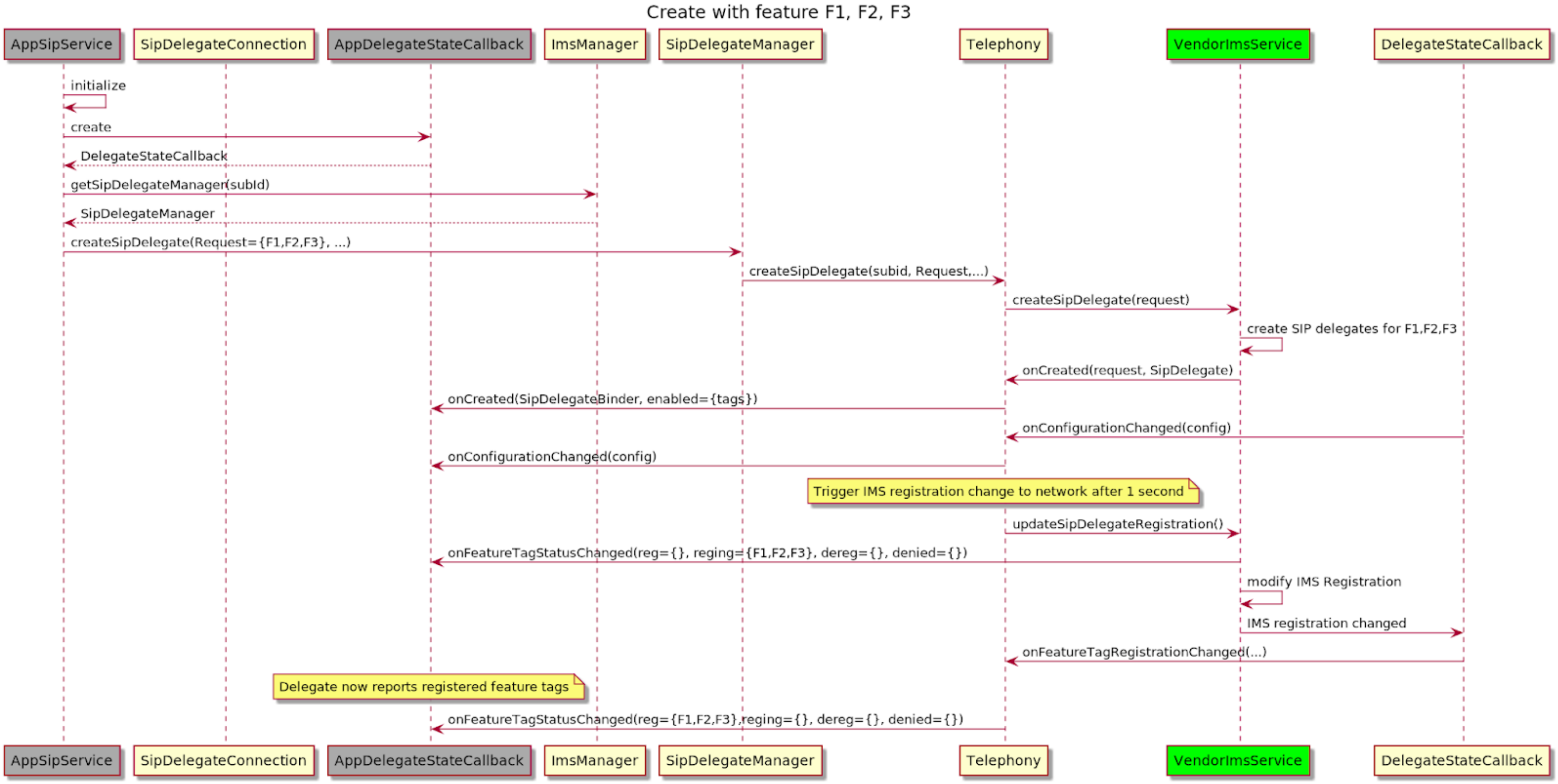
Once the `SipDelegateConnection` is set up, `SipMessages` can then be sent and received on this transport. Any time a feature tag associated with an ongoing SIP dialog moves to the deregistering state, the SipDelegateConnection must close the SIP Dialog in order for the IMS stack to modify the current IMS registration (as per some carrier's requirements).

Sequence Diagrams

Common Scenarios

Scenario ST.CS.1

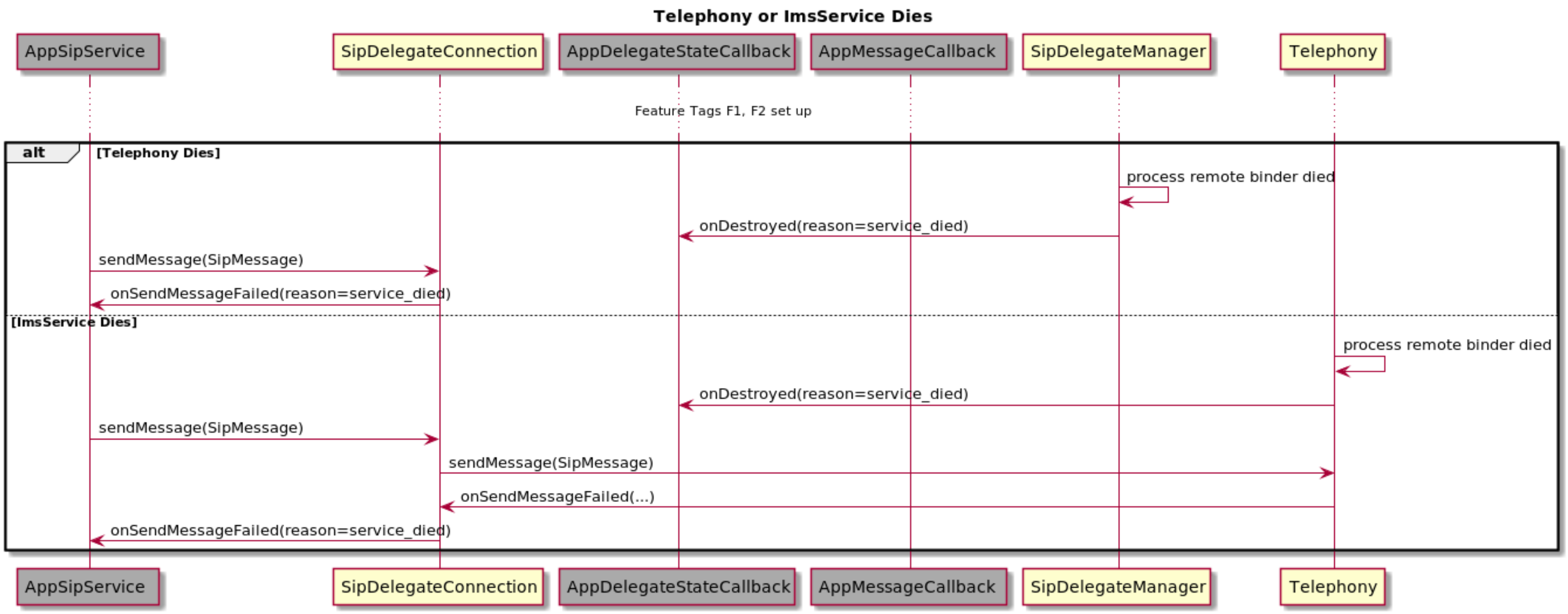
An RCS application that is sufficiently privileged creates a `SipDelegate` for features {F1, F2, F3}, which are all provisioned:

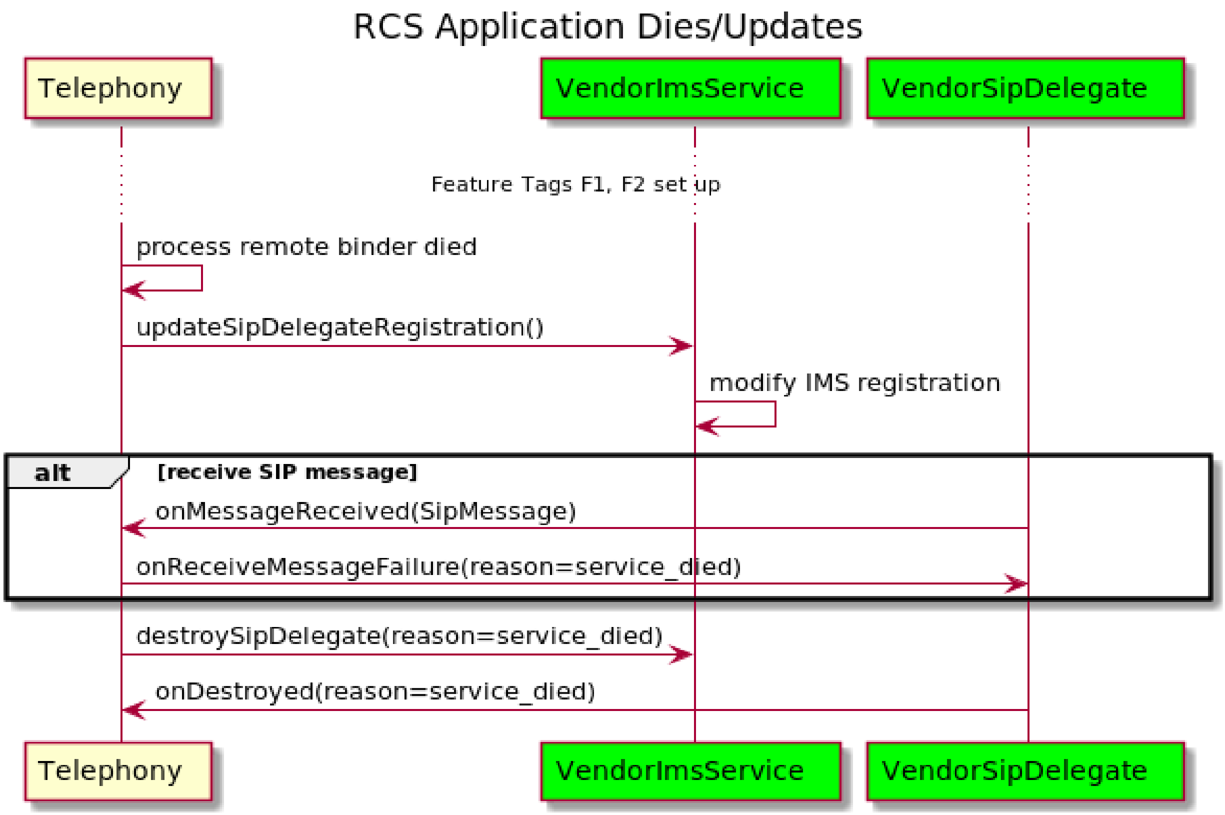

Create with feature F1, F2, F3
AppSipService
SipDelegateConnection
AppDelegateStateCallback
ImsManager
SipDelegateManager
Telephony
VendorImsService
DelegateStateCallback
initialize
create
DelegateStateCallback
getSipDelegateManager(subId)
SipDelegateManager
createSipDelegate(Request={F1,F2,F3}, ...)
createSipDelegate(subid, Request,...)
createSipDelegate(request)
create SIP delegates for F1,F2,F3
onCreated(request, SipDelegate)
onCreated(SipDelegateBinder, enabled={tags})
onConfigurationChanged(config)
onConfigurationChanged(config)
Trigger IMS registration change to network after 1 second
updateSipDelegateRegistration()
onFeatureTagStatusChanged(reg={}, reging={F1,F2,F3}, dereg={}, denied={})
modify IMS Registration
IMS registration changed
onFeatureTagRegistrationChanged(...)
Delegate now reports registered feature tags
onFeatureTagStatusChanged(reg={F1,F2,F3},reging={}, dereg={}, denied={})
AppSipService
SipDelegateConnection
AppDelegateStateCallback
ImsManager
SipDelegateManager
Telephony
VendorImsService
DelegateStateCallback

Edge Case Scenarios

Scenario ST.EC.1

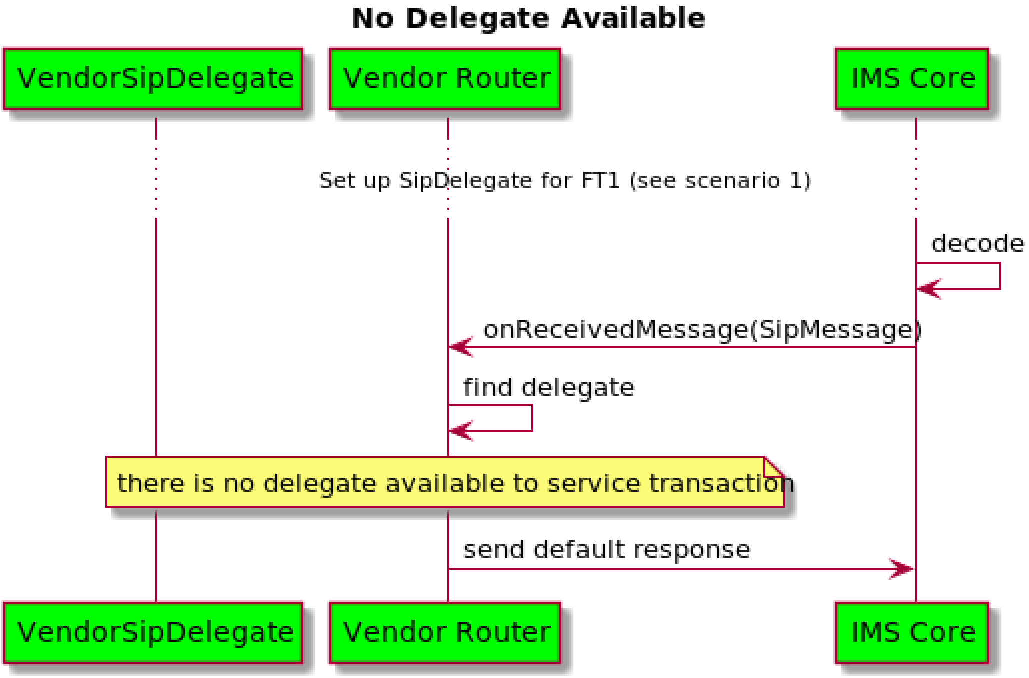
Situations where the remote application, the telephony process, or the ImsService dies unexpectedly.




RCS Application Dies/Updates
Telephony
VendorImsService
VendorSipDelegate
Feature Tags F1, F2 set up
process remote binder died
updateSipDelegateRegistration()
modify IMS registration
alt
[receive SIP message]
onMessageReceived(SipMessage)
onReceiveMessageFailure(reason=service_died)
destroySipDelegate(reason=service_died)
onDestroyed(reason=service_died)
Telephony
VendorImsService
VendorSipDelegate

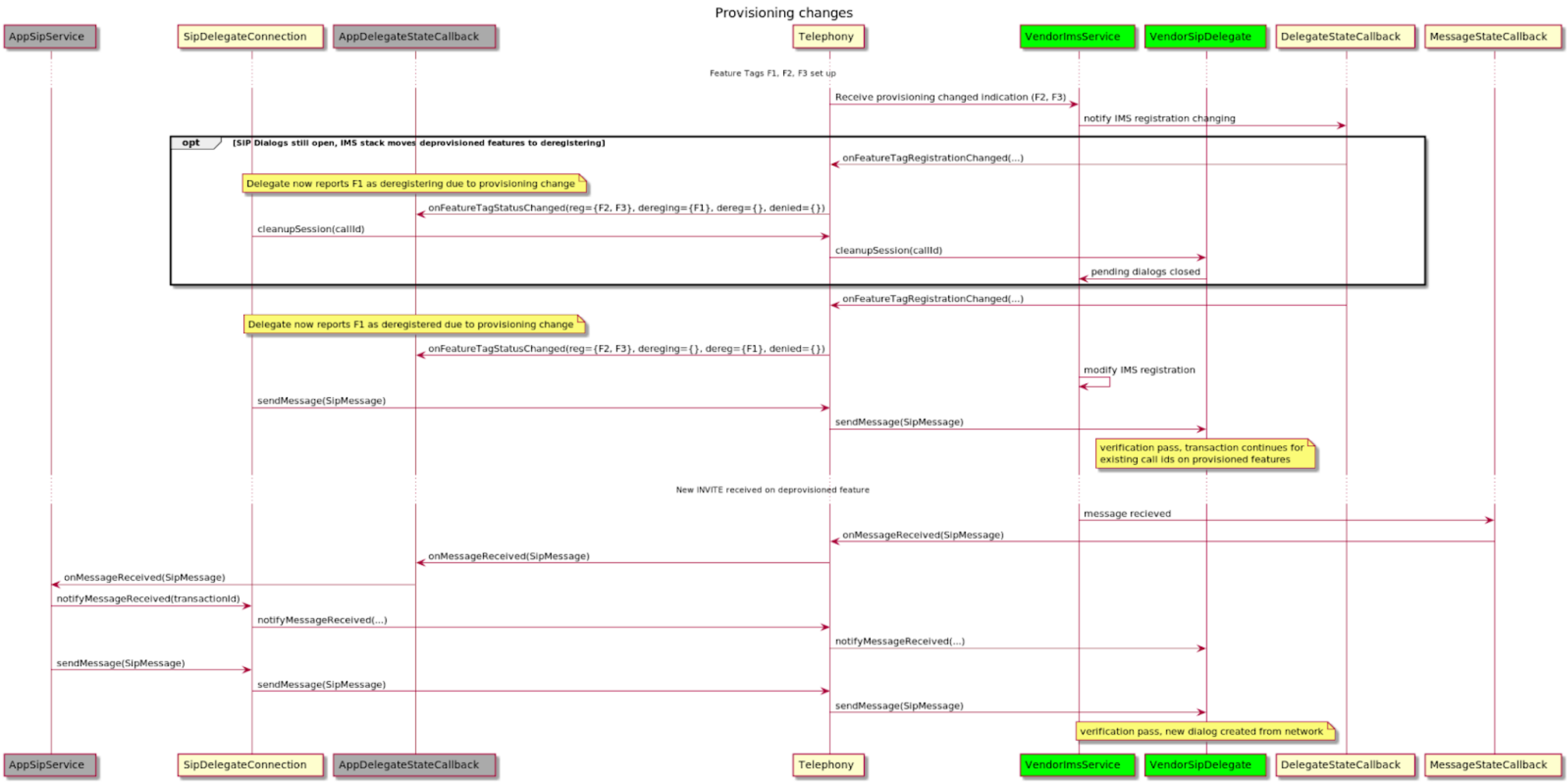
Scenario ST.EC.2

Scenarios where there is no `SipDelegate` available to handle the response to a message. This can happen if the remote application has just died, for example. The `ImsService` should be able to handle sending a default error response to the network.

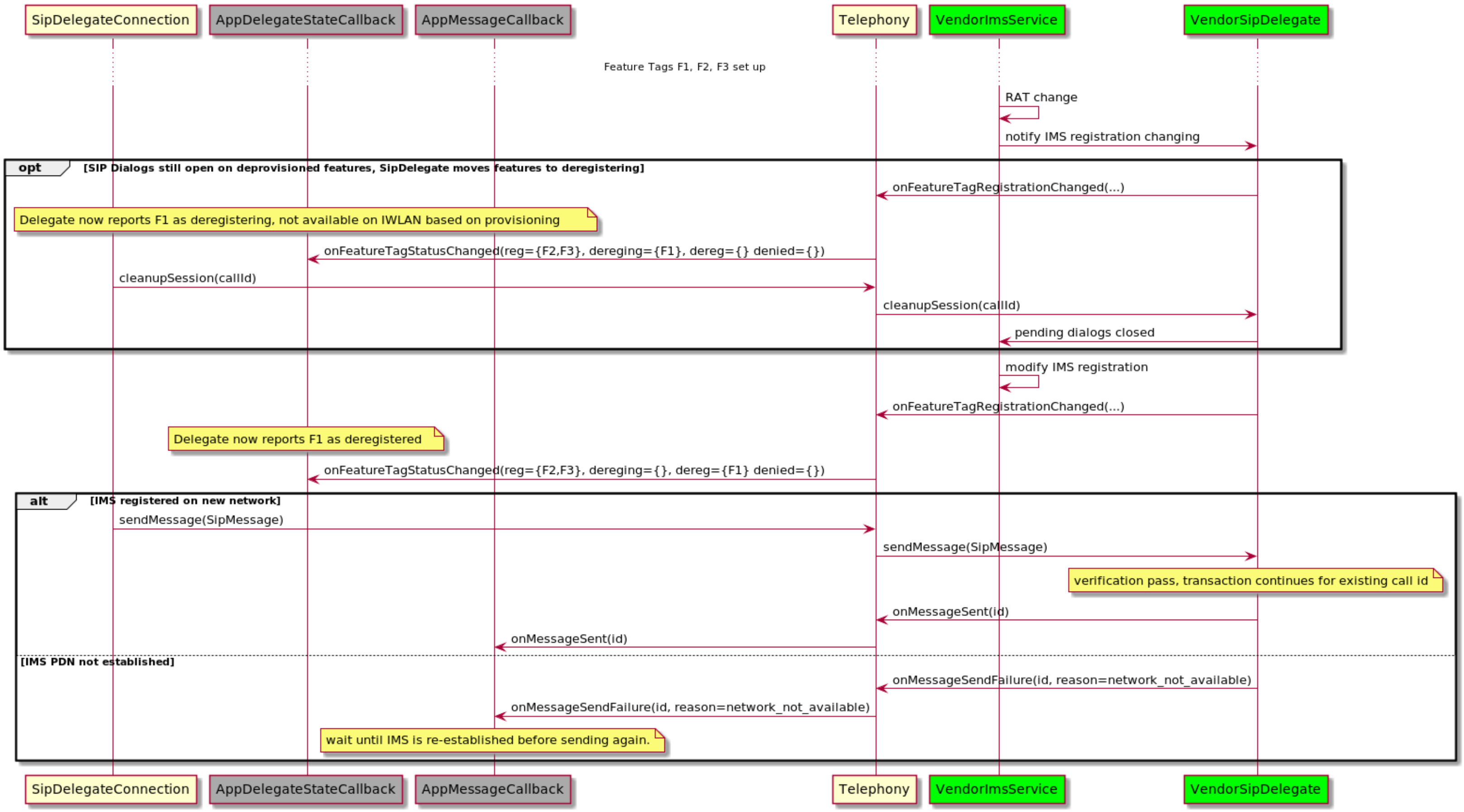


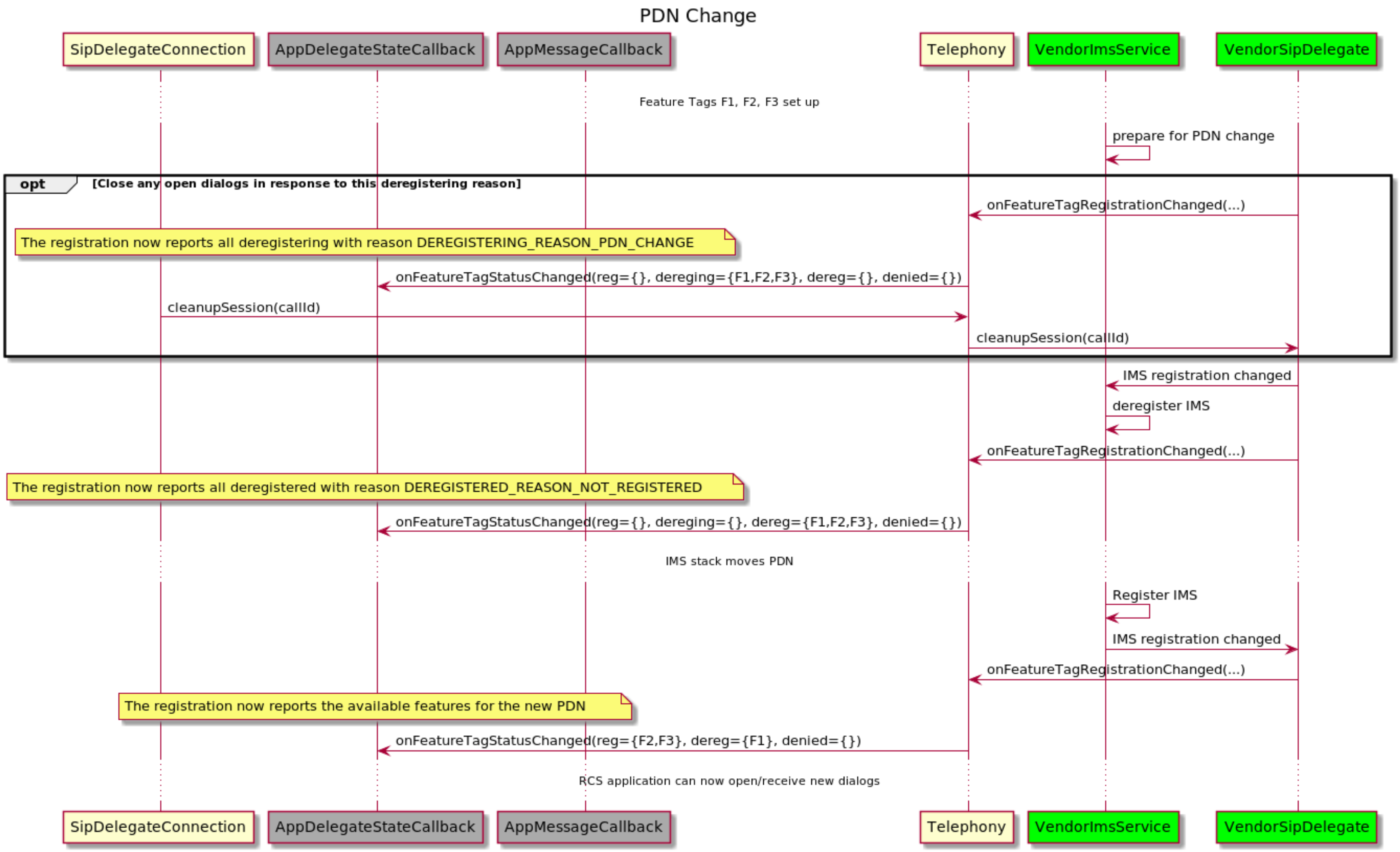
Scenario ST.EC.3

When provisioning has changed and caused a feature tag to be reported as deregistering, it is the RCS application's responsibility to call `SipDelegateConnection#cleanupSession` so the IMS stack can continue with modifying the IMS registration to reflect the new provisioning configuration. The diagrams below help illustrate how this should be handled. Provisioning changes can also occur when handing over between transports or when IMS moves across PDNs, so the same process must be applied. In the case that the RCS application still receives an incoming out-of-dialog message for a deprovisioned feature tag, it should still be able to respond accordingly.


Provisioning changes
AppSipService
SipDelegateConnection
AppDelegateStateCallback
Telephony
VendorImsService
VendorSipDelegate
DelegateStateCallback
MessageStateCallback
Feature Tags F1, F2, F3 set up
Receive provisioning changed indication (F2, F3)
notify IMS registration changing
opt
[SIP Dialogs still open, IMS stack moves deprovisioned features to deregistering]
onFeatureTagRegistrationChanged(...)
Delegate now reports F1 as deregistering due to provisioning change
onFeatureTagStatusChanged(reg={F2, F3}, dereging={F1}, dereg={}, denied={})
cleanupSession(callId)
cleanupSession(callId)
pending dialogs closed
onFeatureTagRegistrationChanged(...)
Delegate now reports F1 as deregistered due to provisioning change
onFeatureTagStatusChanged(reg={F2, F3}, dereging={}, dereg={F1}, denied={})
modify IMS registration
sendMessage(SipMessage)
sendMessage(SipMessage)
verification pass, transaction continues for existing call ids on provisioned features
New INVITE received on deprovisioned feature
message recieved
onMessageReceived(SipMessage)
onMessageReceived(SipMessage)
onMessageReceived(SipMessage)
notifyMessageReceived(transactionId)
notifyMessageReceived(...)
notifyMessageReceived(...)
sendMessage(SipMessage)
sendMessage(SipMessage)
sendMessage(SipMessage)
verification pass, new dialog created from network
AppSipService
SipDelegateConnection
AppDelegateStateCallback
Telephony
VendorImsService
VendorSipDelegate
DelegateStateCallback
MessageStateCallback

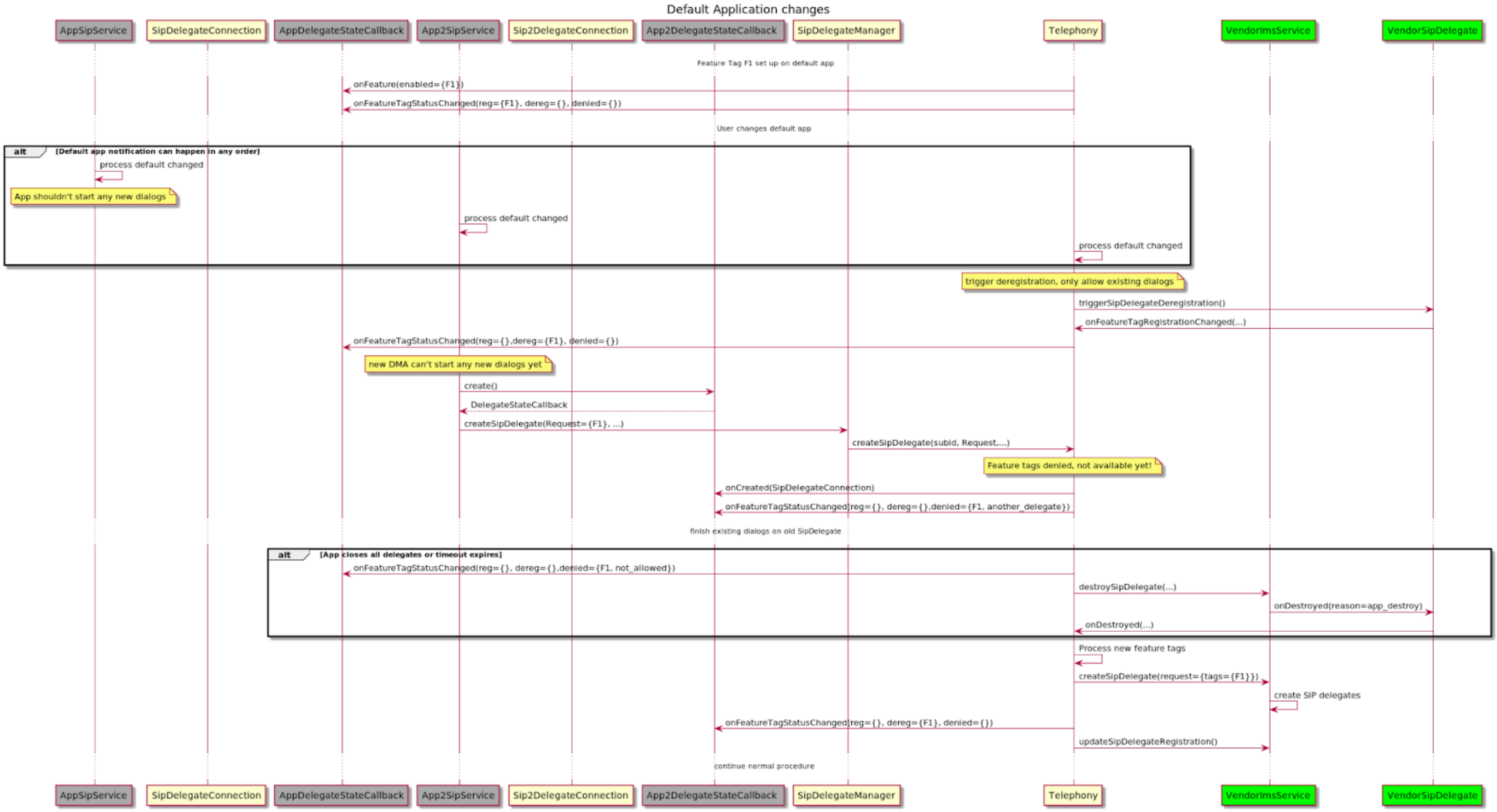
Handover LTE -> IWLAN


SipDelegateConnection
AppDelegateStateCallback
AppMessageCallback
Telephony
VendorImsService
VendorSipDelegate
Feature Tags F1, F2, F3 set up
RAT change
notify IMS registration changing
opt
[SIP Dialogs still open on deprovisioned features, SipDelegate moves features to deregistering]
onFeatureTagRegistrationChanged(...)
Delegate now reports F1 as deregistering, not available on IWLAN based on provisioning
onFeatureTagStatusChanged(reg={F2,F3}, dereging={F1}, dereg={} denied={})
cleanupSession(callId)
cleanupSession(callId)
pending dialogs closed
modify IMS registration
onFeatureTagRegistrationChanged(...)
Delegate now reports F1 as deregistered
onFeatureTagStatusChanged(reg={F2,F3}, dereging={}, dereg={F1} denied={})
alt
[IMS registered on new network]
sendMessage(SipMessage)
sendMessage(SipMessage)
verification pass, transaction continues for existing call id
onMessageSent(id)
onMessageSent(id)
[IMS PDN not established]
onMessageSendFailure(id, reason=network_not_available)
onMessageSendFailure(id, reason=network_not_available)
wait until IMS is re-established before sending again.
SipDelegateConnection
AppDelegateStateCallback
AppMessageCallback
Telephony
VendorImsService
VendorSipDelegate


PDN Change
SipDelegateConnection
AppDelegateStateCallback
AppMessageCallback
Telephony
VendorImsService
VendorSipDelegate
Feature Tags F1, F2, F3 set up
prepare for PDN change
opt
[Close any open dialogs in response to this deregistering reason]
onFeatureTagRegistrationChanged(...)
The registration now reports all deregistering with reason DEREGISTERING_REASON_PDN_CHANGE
onFeatureTagStatusChanged(reg={}, dereging={F1,F2,F3}, dereg={}, denied={})
cleanupSession(callId)
cleanupSession(callId)
IMS registration changed
deregister IMS
onFeatureTagRegistrationChanged(...)
The registration now reports all deregistered with reason DEREGISTERED_REASON_NOT_REGISTERED
onFeatureTagStatusChanged(reg={}, dereging={}, dereg={F1,F2,F3}, denied={})
IMS stack moves PDN
Register IMS
IMS registration changed
onFeatureTagRegistrationChanged(...)
The registration now reports the available features for the new PDN
onFeatureTagStatusChanged(reg={F2,F3}, dereg={F1}, denied={})
RCS application can now open/receive new dialogs
SipDelegateConnection
AppDelegateStateCallback
AppMessageCallback
Telephony
VendorImsService
VendorSipDelegate

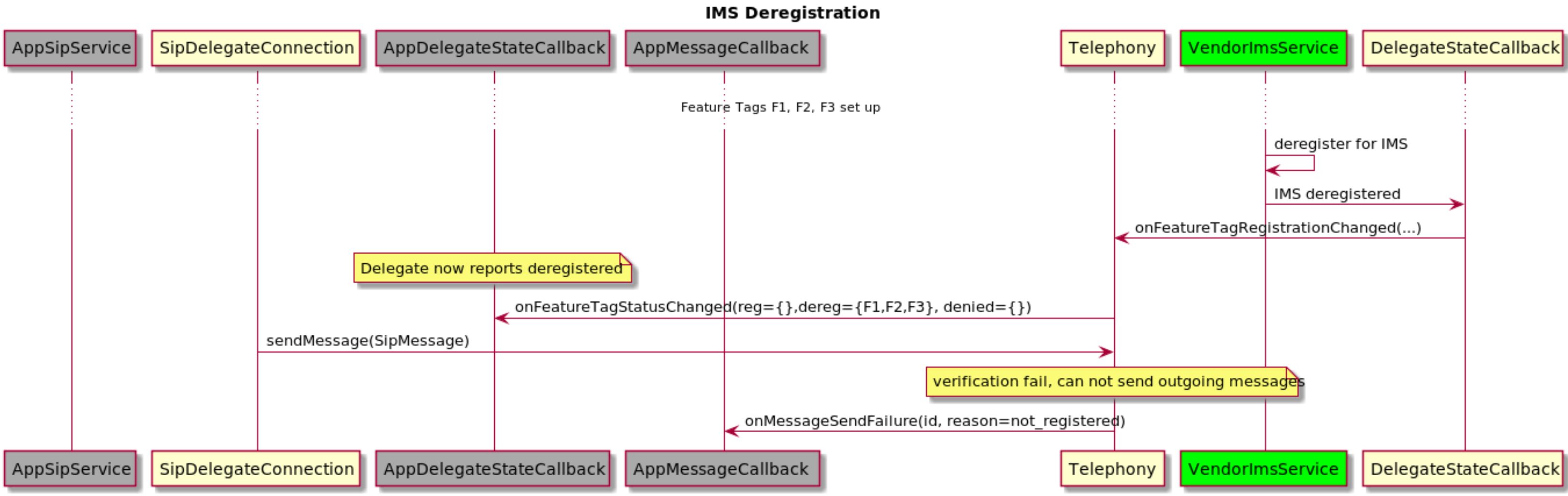
Scenario ST.EC.4

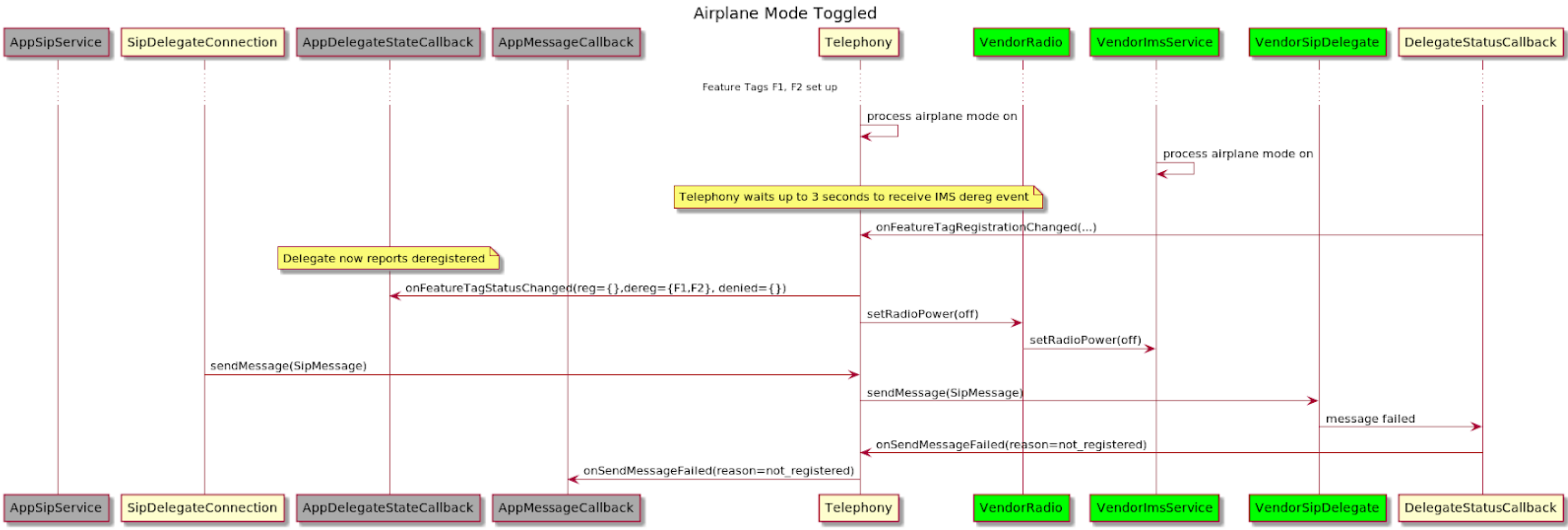
The user changes the default SMS application, requiring the framework to now deny feature tags that are no longer available to that app. First the framework deregisters the feature tags and allows the application to finish existing dialogs. Then after a timeout period, the framework tears down the SipDelegate and allows the new default SMS application to register for its own SipDelegateConnection.


Default Application changes
AppSipService
SipDelegateConnection
AppDelegateStateCallback
App2SipService
Sip2DelegateConnection
App2DelegateStateCallback
SipDelegateManager
Telephony
VendorImsService
VendorSipDelegate
Feature Tag F1 set up on default app
onFeature(enabled={F1})
onFeatureTagStatusChanged(reg={F1}, dereg={}, denied={})
User changes default app
alt
[Default app notification can happen in any order]
process default changed
App shouldn't start any new dialogs
process default changed
process default changed
trigger deregistration, only allow existing dialogs
triggerSipDelegateDeregistration()
onFeatureTagRegistrationChanged(...)
onFeatureTagStatusChanged(reg={},dereg={F1}, denied={})
new DMA can't start any new dialogs yet
create()
DelegateStateCallback
createSipDelegate(Request={F1}, ...)
createSipDelegate(subId, Request,...)
Feature tags denied, not available yet!
onCreated(SipDelegateConnection)
onFeatureTagStatusChanged(reg={}, dereg={},denied={F1, another_delegate})
finish existing dialogs on old SipDelegate
alt
[App closes all delegates or timeout expires]
onFeatureTagStatusChanged(reg={}, dereg={},denied={F1, not_allowed})
destroySipDelegate(...)
onDestroyed(reason=app_destroy)
onDestroyed(...)
Process new feature tags
createSipDelegate(request={tags={F1}})
create SIP delegates
onFeatureTagStatusChanged(reg={}, dereg={F1}, denied={})
updateSipDelegateRegistration()
continue normal procedure
AppSipService
SipDelegateConnection
AppDelegateStateCallback
App2SipService
Sip2DelegateConnection
App2DelegateStateCallback
SipDelegateManager
Telephony
VendorImsService
VendorSipDelegate

Scenario ST.EC.5

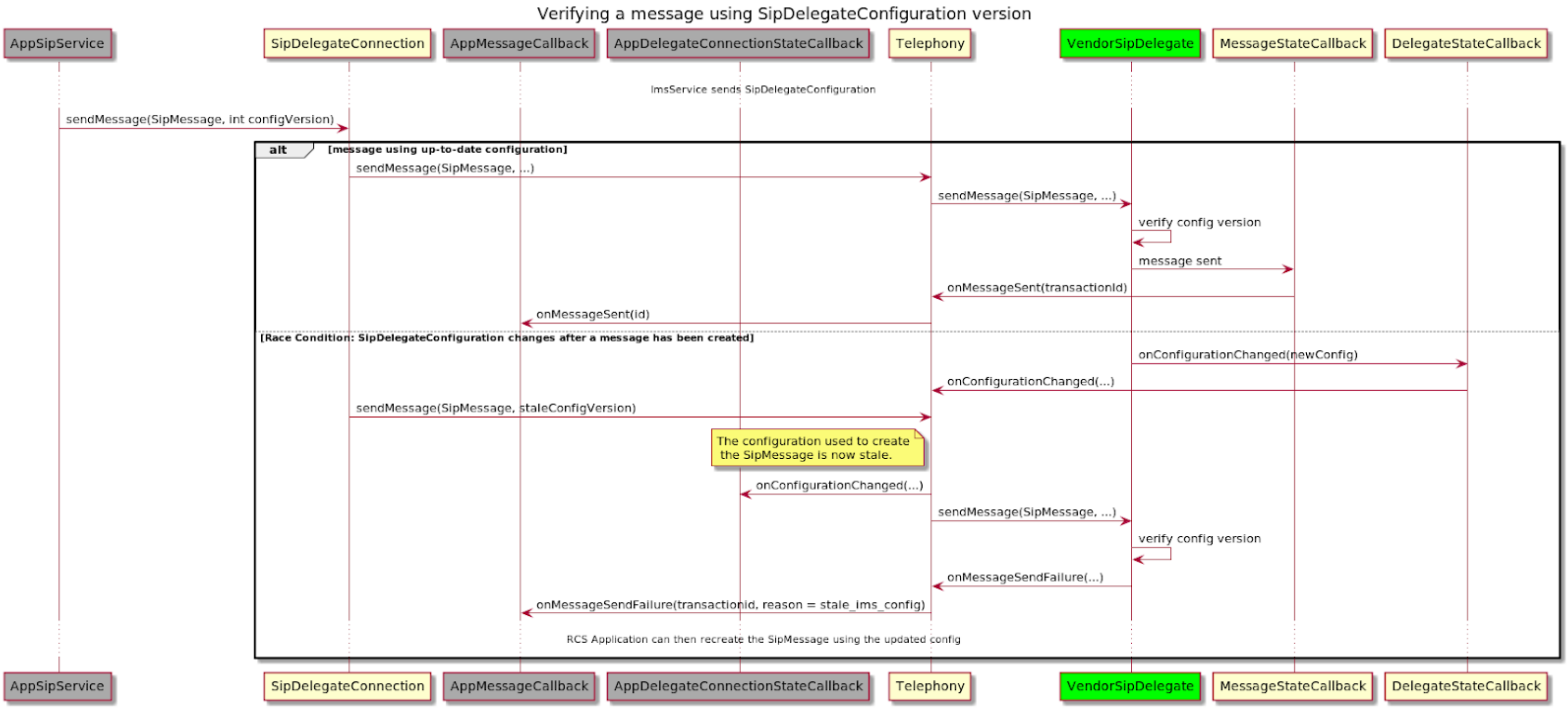
When the `ImsService` deregisters IMS, this should be communicated to the remote application and new outgoing SIP messages will be denied.





Scenario ST.EC.6

There are race conditions associated with the `SipDelegateConfiguration` changing after a `SipMessage` is on the way of being sent to the vendor `ImsService`. In order to reduce the amount of verification the vendor `ImsService` needs to do for each message to ensure it is using the most up-to-date configuration, each `SipMessage` generated by the RCS application will also be required to provide the `SipDelegateConfiguration` version used to generate it. This will allow the vendor `ImsService` to verify the message quickly and reject it if the configuration used to create it is out-of-date. The RCS application may then regenerate the `SipMessage` with the new configuration.


Verifying a message using SipDelegateConfiguration version
AppSipService
SipDelegateConnection
AppMessageCallback
AppDelegateConnectionStateCallback
Telephony
VendorSipDelegate
MessageStateCallback
DelegateStateCallback
ImsService sends SipDelegateConfiguration
sendMessage(SipMessage, int configVersion)
alt
[message using up-to-date configuration]
sendMessage(SipMessage, ...)
sendMessage(SipMessage, ...)
verify config version
message sent
onMessageSent(transactionId)
onMessageSent(id)
[Race Condition: SipDelegateConfiguration changes after a message has been created]
onConfigurationChanged(newConfig)
onConfigurationChanged(...)
sendMessage(SipMessage, staleConfigVersion)
The configuration used to create
the SipMessage is now stale.
onConfigurationChanged(...)
sendMessage(SipMessage, ...)
verify config version
onMessageSendFailure(...)
onMessageSendFailure(transactionId, reason = stale_ims_config)
RCS Application can then recreate the SipMessage using the updated config
AppSipService
SipDelegateConnection
AppMessageCallback
AppDelegateConnectionStateCallback
Telephony
VendorSipDelegate
MessageStateCallback
DelegateStateCallback

Dedicated Bearer Access

Objectives

- Create a generalized QoS app level API that provides consumers QoS attributes from various different sources such as an LTE, NR or WiFi Network.
- Create a QoS app level APIs that supports RECEIVING unsolicited requests, while still providing enough flexibility in the definition that it can later support making Qos requests. (see Appendix.)
- Provide QoS signals from network initiated Dedicated Bearers or Qos flows needed for the RCS Single Reg project.
- Create IRadio HAL APIs to expose the QOS information from vendors to support the above platform APIs.

Background

This proposal is tied to the overall RCS Single Reg project, and more specifically, a need to support MSRP (an instant messaging protocol) over Carrier Networks which includes connecting sockets over a Dedicated Bearer. Dedicated Bearers ensure Qos levels for various media types such as Video Streaming, Voice, etc. over an LTE or 5G network on the IMS PDN/DNN. A Dedicated Bearer / QOS flow becomes available as a part of an SIP session which is negotiated through the exchange of SIP \ SDP messages outside of the context of this API.

In this proposal, we propose a Qos API within the connectivity stack with a flexible enough shape to handle a wide range of Qos use cases including the one needed for RCS Single Reg. Since our current requirement has no need to request a qos in a generic way, **this API strictly provides the ability to receive Qos signals and NOT request a Qos level.**

This proposal is split up into two broad parts. Part 1 addresses the general Qos API, and Part 2 shows how we would apply this API to the specific RCS use case.

All APIs will start out as System APIs in V1, with the potential to be promoted to a public API if the need arises in either Android S or beyond.

Definitions

In order to reduce confusion, here are a few terms defined ahead of time:

- **3-tuple-** Transport Protocol \ Local IP \ Local Port
- **5-tuple-** Transport Protocol \ Local IP \ Local Port \ Remote IP \ Remote Port
- **Packet filter-** A broad term that covers both 3-Tuple and 5-Tuple
- **Socket-** Covers both a TCP and UDP socket unless otherwise specified. When in a Java code sample though, Socket is referring to a Java Socket object which is TCP only.
- **Local address-** Refers to both the Local IP and Local Port as a tuple.
- **Remote address-** Refers to both the Remote IP and Remote Port as a tuple.
- **DB-** Stands for Dedicated Bearer

Part 1 - General Qos API

API Concepts

Qos Sessions and Services

Qos Sessions

A Qos Session represents a Quality of Service level that is available to either an already established connection or a possible future connection that the UE could establish. We purposely call it a Session because it has a definitive start and end to it.

Each Qos Session has a type id associated with it which represents a type of Qos Session based off on the specifics of a Network.

The Qos Session also has a unique id in order for the API consumer to distinguish Qos Session from one another. The unique id is a long and is composed of two halves:

The first 32 bits are reserved for the Qos Session Type.
The last 32 bits are left to the Qos Service to set. Part II contains the details on how the session id is set in practice.

Qos Session Attributes

There are different types of Qos Sessions. Some attributes may be common between types of sessions and others will be unique to that type of Qos Session.

A common set of Qos attributes will exist within an interface called QosSessionAttributes. An example attribute could be the max bit rate on the downlink. A Qos attribute considered common enough to exist on this interface needs to exist in at least two session types to be useful. We currently propose to roll out with two common attributes: max uplink and max downlink. Since the common section is an interface, adding on additional attributes in later Android versions is trivial.

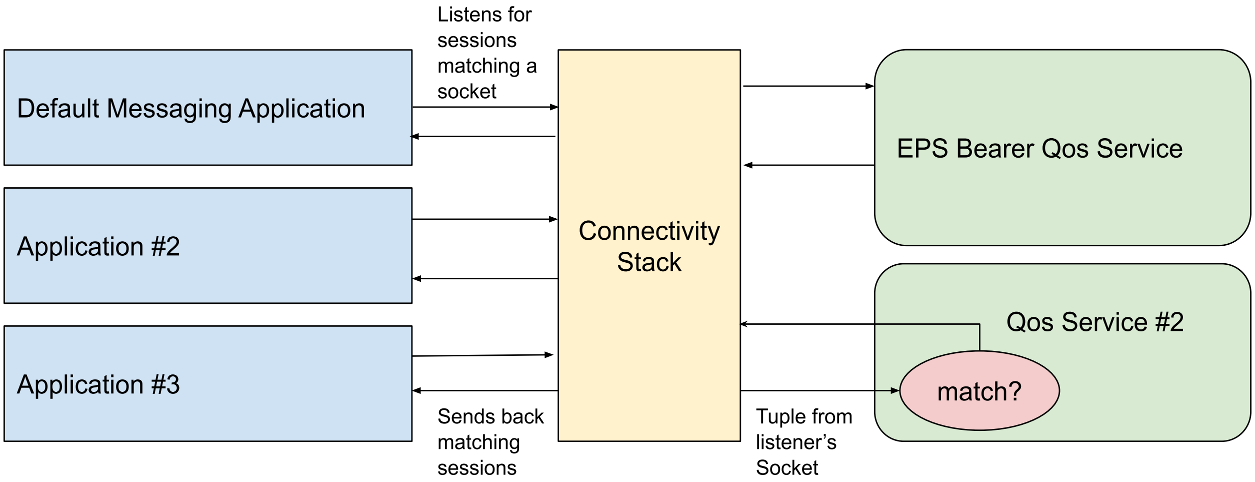
There are also non-common attributes that we need to expose. Non-common attributes will be exposed on a class that implements QosSessionAttributes. An example of such an attribute would be the qci (Qos Class Identifier) value that exists on an LTE Bearer Qos Session. That attribute is within the class EpsBearerQosSessionAttributes which implements QosSessionAttributes. The qci attribute is NOT within QosSessionAttributes though since it's specific to the eps bearer case.

Qos Service

A Qos Service generates Qos Sessions of a particular type. They are a non-API visible concept and exist within the system server under any AOSP component that the Connectivity Service has access to (eg. Telephony, Wifi, etc.)

Qos Services both generate Qos Sessions and evaluate which Qos Sessions are visible to which Qos Filters. Having the matching logic within the Qos Service and not a generic component gives Qos Services the flexibility around possible edge cases that would be difficult to code in a generic component.

Below is a conceptual diagram of how the components are associated with one another and not meant to serve as an architectural diagram:


Default Messaging Application
Application #2
Application #3
Listens for sessions matching a socket
Sends back matching sessions
Connectivity Stack
EPS Bearer Qos Service
Qos Service #2
match?
Tuple from listener's Socket

Receiving Qos Sessions

A Qos Session Callback listens for changes on Qos Sessions that match a given filter. The two parameters are the key inputs required to receive Qos Sessions. Both concepts are addressed in the sections below.

```
ConnectivityManager.registerQosCallback(QosFilter, executor, QosCallback);
```

The register method will not throw exceptions directly. Exceptions are communicated through the callback onError. For more information, see the “Exception Handling” section below.

To release the callback, thereby stopping it, the consumer simply calls the unregister method with the same callback instance used in the filter. Non-registered or previously unregistered callbacks are treated as a no op. The same callback may not be used if it is already registered with ConnectivityManager. **BUT**, it may be re-used if it was unregistered.

```
ConnectivityManager.unregisterQosCallback(callback);
```

Filtering Qos Sessions

There are several ways that a developer may want to filter down which Qos Sessions they're interested in. For example, a developer may be interested in all Qos Sessions that satisfy localPort = 2250, and another may want Qos Sessions that run over UDP to a specific IP address.

We do not know all of the combinations of filters that developers may want. In V1, we are providing just one type of filter that is referred to as a “Qos Socket Filter.” Long term though, we expect several different types of filters that can support a variety of parameters. For example, a filter could be created that listens for an exact 5-tuple match, another example is a 3-Tuple filter that matches on a local port range.

To keep the API shape flexible, each type of filter would come with its own factory method that returns a type of QosFilter.

Most types of Qos Filters will require a type of Network. If the consumer does NOT have access to the specified Network, a SecurityException will be passed to the API consumer (see Exception Handling section below for more information on how security exceptions are handled.)

Qos Socket Filter

A Qos Socket Filter takes a socket and a network as inputs. The filter matches Qos Sessions against the parameters used in its bind() method, making this a 3-Tuple filter with a couple of special rules:

1. The socket MUST be bound to on registration, otherwise an exception is thrown.
2. The callback is torn down if the socket becomes unbound while trying to deliver a callback.

Here are a couple of ways to create and then bind the socket using the Network object:

```
Socket bindSocketOne(Network network) {
        Socket socket = network.getSocketFactory().createSocket();
        //Create the socket and then bind using Socket#bind
        socket.bind("1.2.3.4", 99);
        return socket;
}
```

```
Socket bindSocketExampleTwo(Network network) {
        //Create and bind the socket in one shot
        return network.getSocketFactory().createSocket("1.2.3.4", 99);
}
```

To handle the cases where the network may not use the local port in the dedicated bearer establishment and only includes the remote port information, it's best if the applications use a connected socket for listening to the MSRP dedicated bearer events.

Exception Handling

All exceptions are asynchronously passed through onError(), including exceptions that occur upon registration of the callback. To access the underlying exception, the consumer checks for the cause set on QosCallbackException.

```
void onError(QosCallbackException exception) {
        Exception underlyingException = exception.getCause();
}
```

Exceptional Cases

Binder dies (or other unexpected runtime exception)- onError() is invoked with the underlying runtime exception as the cause within QosCallbackException.

The socket is unbound upon registration- onError() is invoked with a QosFilter.SocketNotBoundException as the cause.

The socket on a socket filter becomes unbound- onError() is invoked with a QosFilter.SocketNotBoundException as the cause.

The Application does not have access to the Network- onError() is invoked with a SecurityException as the cause.

Security

There is no overarching permission required to register a callback. The only requirement for V1 is that the application has access to the Network used within the Qos Filter. In the event that the consumer does not have the correct permission, onError() will be invoked with an underlying SecurityException as the cause.

Happy Path

The code sample below shows how an Application Developer would consume this API:

```
//A socket that is already bound to the source address and port.//
Socket mSocket;

void main(Network imsNetwork, Executor executor) {
        /* Set the appropriate network, callback, and socket. */
        QosFilter socketFilter = QosFilter.fromSocket(mSocket, imsNetwork);

        /* Requests a new qos callback to be attached with an associated callback. */
        ConnectivityManager.registerQosCallback(socketFilter, executor, mCallback);
}


/*The callback is associated with the callback request made with ConnectivityManager and receives the Qos
events emitted from matching Qos sessions.*/
QosCallback mCallback = new QosCallback() {
        /* Invoked when an active Qos session matches the socket. */
        @Override
        void onQosSessionAvailable(@NonNull QosSession session, @NonNull QosSessionAttributes
sessionAttributes) {
      /* A qos session is available. */
        }
}
```

When onQosSessionAvailable is called, the QosSessionAttributes returned will either be an instance of EpsBearerQosSessionAttributes or NrQosSessionAttributes, depending on the underlying radio access technology.

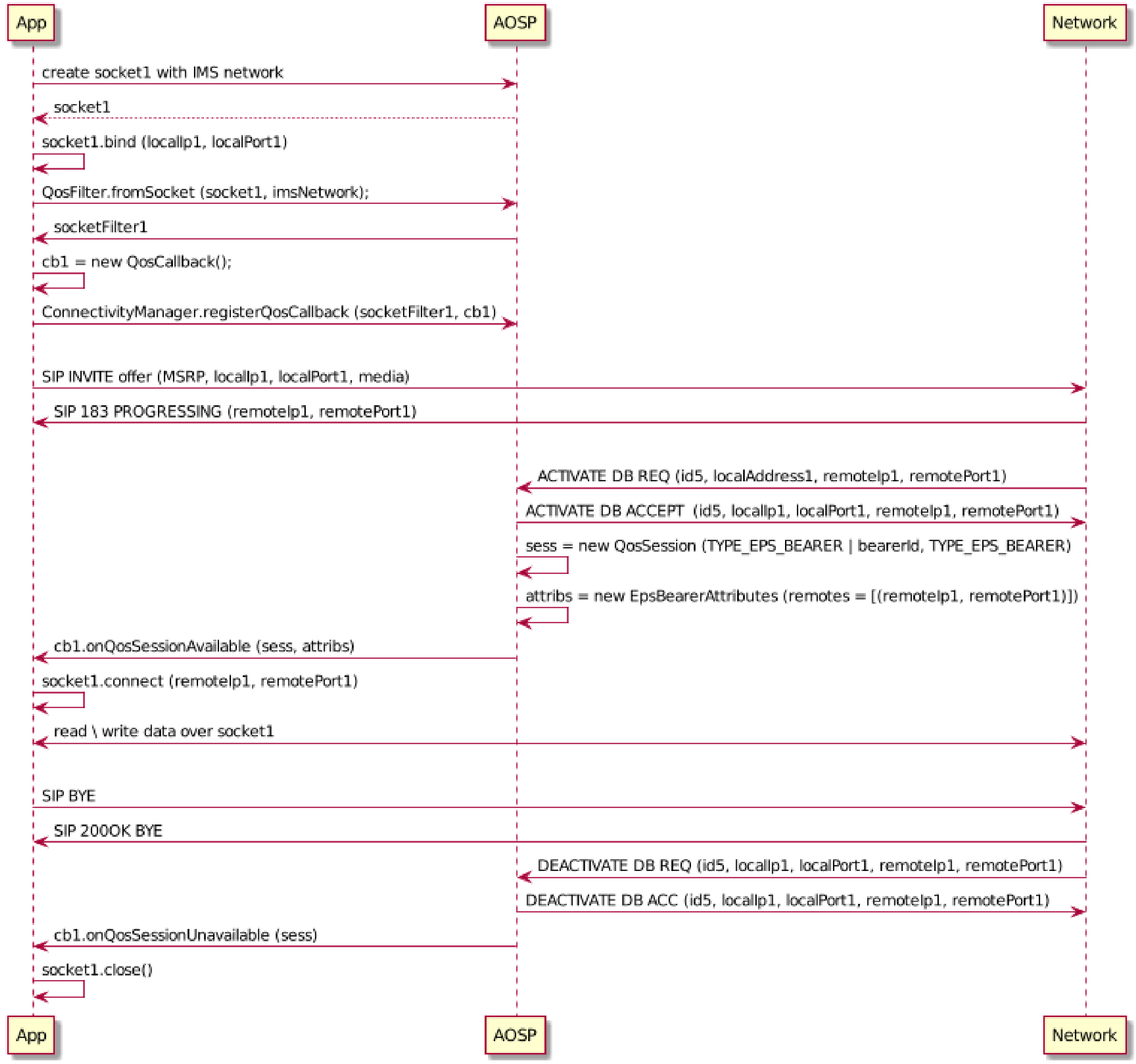
Security

The only security requirement to receive callbacks from EPS Bearer Qos Sessions is that the API consumer have access to the IMS Network. The IMS network is restricted. Restricted networks require the application to have the CONNECTIVITY_USE_RESTRICTED_NETWORKS permission which is only available to privileged applications.

Common Eps Bearer Workflow

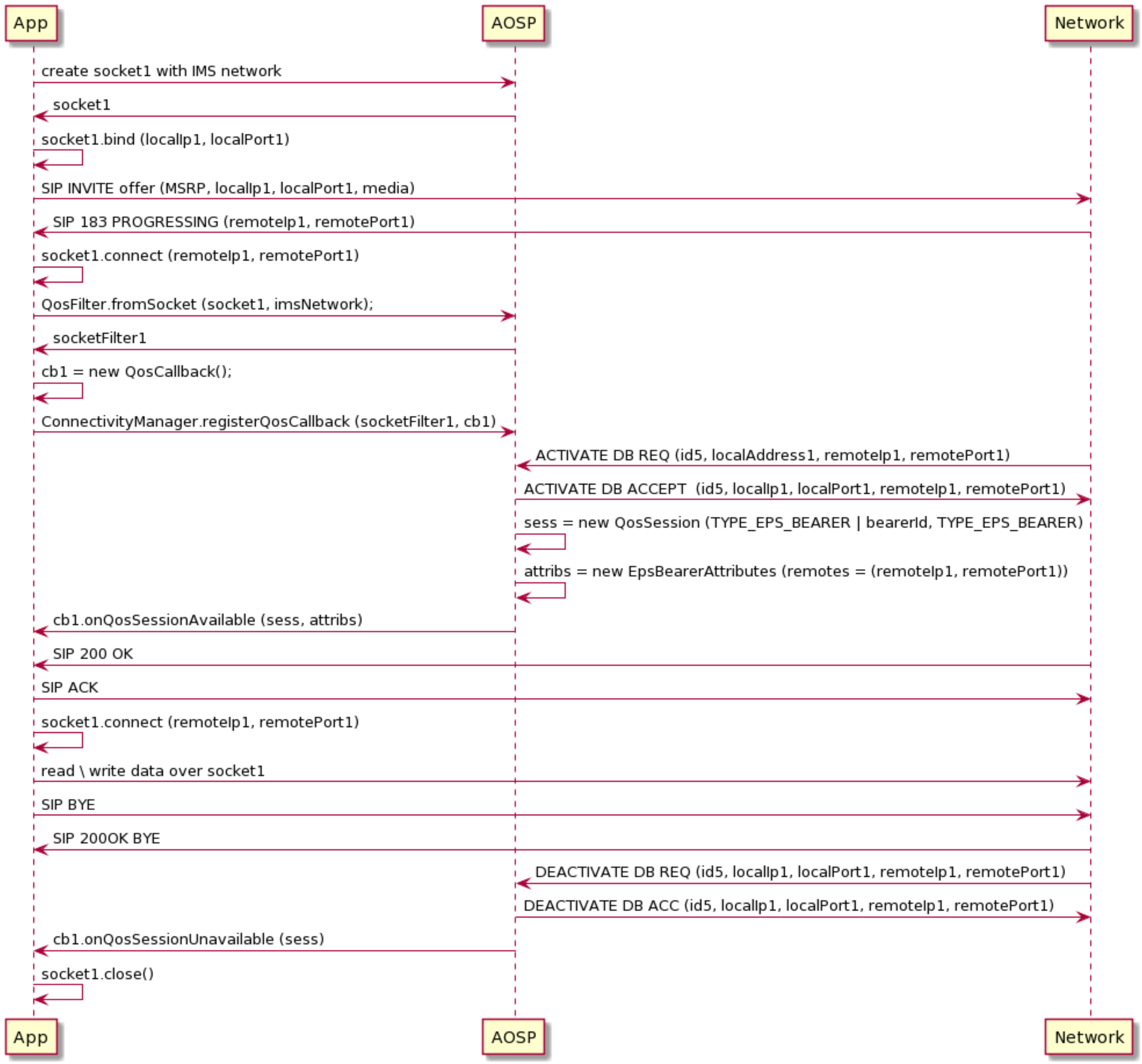
The workflow below shows how an App would perform a MSRP SDP negotiation alongside the Qos Callback API.

Following sequence diagram shows application registering for call back for a bound socket i.e socket is not yet connected. In this case the telephony will do filter matching based on the local address and port number and it will successfully match when the network provides local address and port number with the dedicated bearer.


App
AOSP
Network
create socket1 with IMS network
socket1
socket1.bind (localIp1, localPort1)
QosFilter.fromSocket (socket1, imsNetwork);
socketFilter1
cb1 = new QosCallback();
ConnectivityManager.registerQosCallback (socketFilter1, cb1)
SIP INVITE offer (MSRP, localIp1, localPort1, media)
SIP 183 PROGRESSING (remoteIp1, remotePort1)
ACTIVATE DB REQ (id5, localAddress1, remoteIp1, remotePort1)
ACTIVATE DB ACCEPT (id5, localIp1, localPort1, remoteIp1, remotePort1)
sess = new QosSession (TYPE_EPS_BEARER | bearerId, TYPE_EPS_BEARER)
attribs = new EpsBearerAttributes (remotes = [(remoteIp1, remotePort1)])
cb1.onQosSessionAvailable (sess, attribs)
socket1.connect (remoteIp1, remotePort1)
read \ write data over socket1
SIP BYE
SIP 200OK BYE
DEACTIVATE DB REQ (id5, localIp1, localPort1, remoteIp1, remotePort1)
DEACTIVATE DB ACC (id5, localIp1, localPort1, remoteIp1, remotePort1)
cb1.onQosSessionUnavailable (sess)
socket1.close()
App
AOSP
Network

Note: The workflow on the application side is not meant to be followed precisely but is rather an example of what one could do.

Following sequence diagram shows that the application is registering for a call back after the socket is connected. In this case the telephony will do filter matching based on the remote address and port number and it will successfully match when the network provides local address and port number with the dedicated bearer.


App
AOSP
Network
create socket1 with IMS network
socket1
socket1.bind (localIp1, localPort1)
SIP INVITE offer (MSRP, localIp1, localPort1, media)
SIP 183 PROGRESSING (remoteIp1, remotePort1)
socket1.connect (remoteIp1, remotePort1)
QosFilter.fromSocket (socket1, imsNetwork);
socketFilter1
cb1 = new QosCallback();
ConnectivityManager.registerQosCallback (socketFilter1, cb1)
ACTIVATE DB REQ (id5, localAddress1, remoteIp1, remotePort1)
ACTIVATE DB ACCEPT (id5, localIp1, localPort1, remoteIp1, remotePort1)
sess = new QosSession (TYPE_EPS_BEARER | bearerId, TYPE_EPS_BEARER)
attribs = new EpsBearerAttributes (remotes = (remoteIp1, remotePort1))
cb1.onQosSessionAvailable (sess, attribs)
SIP 200 OK
SIP ACK
socket1.connect (remoteIp1, remotePort1)
read \ write data over socket1
SIP BYE
SIP 200OK BYE
DEACTIVATE DB REQ (id5, localIp1, localPort1, remoteIp1, remotePort1)
DEACTIVATE DB ACC (id5, localIp1, localPort1, remoteIp1, remotePort1)
cb1.onQosSessionUnavailable (sess)
socket1.close()
App
AOSP
Network

Note: The workflow on the application side is not meant to be followed precisely but is rather an example of what one could do.

Abnormal Cases

Multiple File Transfers in MSRP

Multiple file transfers in a single MSRP was flagged as a potential issue. After review, there appears to be no conflict with the current design.

Each file transfer within an MSRP includes a SIP INVITE that is sent ahead of time. Fortunately, the same port is used on each file transfer and the Application uses the same socket. From the perspective of the Dedicated Bearer session, there is no impact.

Refer to: https://tools.ietf.org/html/rfc5547#section-8.2.3

Application Died

If an application registers Qos Callbacks and then dies for whatever reason, the Connectivity Service will notify the Qos Services to remove the corresponding callbacks.

Phone Process Dies

If the phone process dies, the active EPS Bearer Sessions will be removed. The callback will **NOT** receive onQosSessionLost() for each formerly available session, but instead, onError() will be called.

PDN no longer accessible

If the PDN is lost, the UE can no longer receive bearer deactivation signals from the network. onError() will be invoked with NetworkReleasedException as the underlying cause. onQosSessionLost() will be called for each session.

Handover

On handover from LTE to IWLAN, any sockets connected over a Dedicated Bearer will lose its Qos. AOSP will send onQosSessionLost() for each session since they are no longer available to the socket.

UE misses DEACTIVATE EPS BEARER CONTEXT REQUEST

The scenario starts with the following steps taking place:

1. A new SDP session was initiated
2. The SDP session was closed
3. The UE missed the DEACTIVATE EPS BEARER signal from the Network.

If this occurs, we run the risk that the UE may think Qos is ready to go prematurely. In reality though, this is HIGHLY unlikely because the following conditions would need to apply:

1. The BYE SIP message was received by the Network
2. The Network retried sending the DEACTIVATE signal 4 times but the UE never received it.
3. The PDN stayed connected.
4. The same remote ip and port was sent back from the Network for the next SDP session.

There was a possible solution proposed to solve this issue, but the solution added enough complexity to both the Application and the Framework that the team deemed it unnecessary.

To avoid this scenario all together, the Application **should rotate local ports between SDP sessions.**

IRADIO HAL changes

To expose the QoS of LTE dedicated bearers and NR PDU sessions to applications via the connectivity manager APIs defined above, telephony would need the following HAL APIs supported by vendors. This is included in the IRADIO 1.6 hal.

1. Vendor shall provide the global QoS indications per PDN in the `SetupDataCallResult`
2. Telephony will track the QoS callback requests from CS and perform the filter matching.

LTE to IWLAN handover

As IWLAN does not support dedicated bearers / QoS, it's better to drop the dedicated bearers even if the network doesn't deactivate explicitly. The vendor shall delete the dedicated bearers and notify telephony so that the call back can be given to the applications which have already registered.

PDN drop by vendor / network

If the vendor or network drops the PDN for any reason then the vendor shall clean up all the dedicated bearers associated with it and notify telephony with the empty list.

Vendor crash

When vendor RIL / modem crashes, the dedicated bearers shall be cleaned up locally. Usually telephony would send RADIO_POWER on=true when vendor services are up after a crash and its expected that vendor services will do fresh registration.

Phone process crash / Airplane mode toggle

Upon phone process crash or airplane mode toggle, telephony would send RADIO_POWER on=false followed by on=true. The vendor shall clean up all the dedicated bearers and notify telephony with the empty list.

RCS User Capability Exchange

Overview

The ImsService API is a trusted extension of the Android telephony framework and encapsulates the entirety of a vendor or carrier’s IMS implementation for both MMTEL and RCS services that the framework supports. This allows the framework to place video/voice calls, send SMS, provide supplementary services over Ut, and perform other IMS specific features, such as RCS User Capability Exchange (UCE). Using the ImsService API also allows vendor specific implementation details to be encapsulated and allows the Android telephony framework to provide implementation agnostic IMS features.

The vendor IMS service will use new extensions to the ImsService API to provide the framework the ability to implement UCE. This new UCE API will extend the existing RcsFeature API, which has only been used to provide over-the-top RCS features up to this point. Figure 3.5.1 below shows the new architecture of the UCE API and its integration into the existing RcsFeature API. This mirrors the existing design pattern used in the MmTelFeature class to provide calling, SMS, and Ut specific features.

From an application perspective, RCS applications may then use the new RcsUceAdapter API extensions to communicate with the android telephony framework and request the RCS capabilities of one or more contacts. If the SIP transport API is being used due to IMS Single Registration being enabled, the single registration capable messaging application will also be able to notify the framework of changes to its RCS capabilities so that these capabilities can be published to the network or be reflected during SIP OPTIONS exchange.

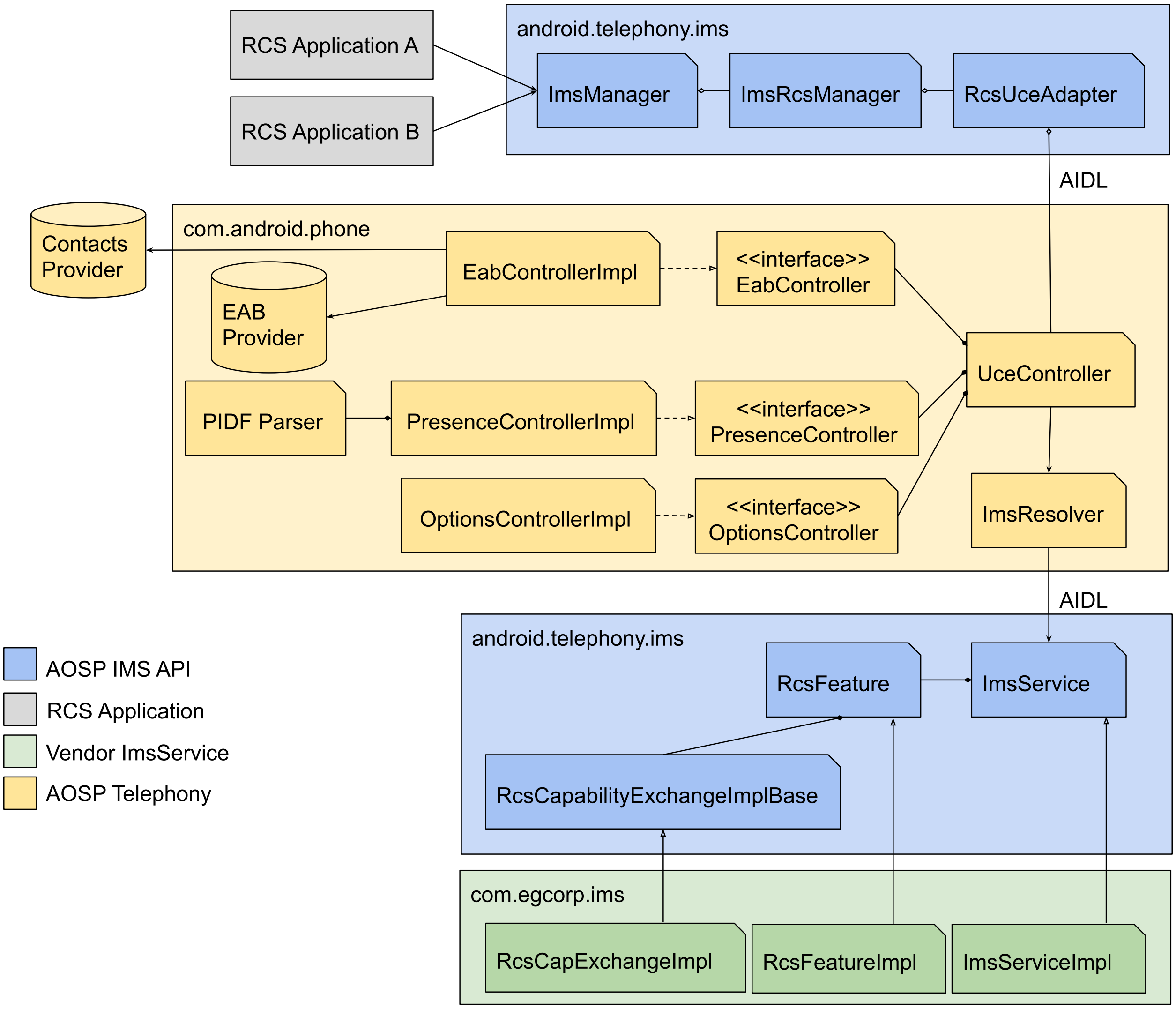


Figure 3.5.1: High level summary of the changes required to support UCE in AOSP. The RCS applications will use the ImsManager API to request the cached capabilities of contacts or request availability queries from the network. AOSP Telephony will control UCE and send requests to the vendor ImsService when necessary to publish capabilities and subscribe to the capabilities of one or more contacts.

Design

The RCS capabilities of one or more contacts are communicated to the device in different ways, depending on the mechanism that the cellular carrier uses: the OMA presence SIMPLE specification (usually simplified as “presence”) or via SIP OPTIONS. These mechanisms are User Capability Exchange (as defined in RCC.07), however there is a brief summary of the operation of these mechanisms below.

SIP OPTIONS

This mechanism is based on the exchange of a SIP OPTIONS request (RFC3261), which is a peer-to-peer capability exchange between two clients. In this approach, there are no intermediate servers caching the capabilities of the contacts, unlike the presence method described below. In this method the following procedure is performed for RCS client A to get the RCS capabilities of client B:

1. A SIP OPTIONS request is created by client A, which contains the full set of RCS capabilities supported by client A in the *contact header* field of the message. This is received by client B, who then updates the capabilities of client A’s URI in their contacts address book (if it exists).
2. Client B responds with a 200 OK message containing its full set of RCS capabilities in the *contact header* field. Client A then uses that response message to update its cached capabilities of the contact URI associated with client B in the contacts address book.
    a. The network will instead send a 480 TEMPORARILY UNAVAILABLE or 408 REQUEST TIMEOUT if client B is a known IMS user, but is currently not registered for IMS. Client A can still use RCS services for Client B that are still allowed when Client B is offline.
    b. The network will instead send a 404 NOT FOUND or 604 DOES NOT EXIST ANYWHERE if client B is not an IMS user or there is no known subscriber matching the URI on the network. The contact should not be considered an RCS user.
    c. Any other final response from the network should mean that Client A should keep Client B’s capability cache in the state it was in before.
    d. According to RCC.07, SIP OPTIONS shall NOT contain an Session Description Protocol body.

NOTE: According to RCC.07 there is no standardized way for SIP OPTIONS to return the aggregated result of multiple clients with the same user identity. Depending on the carrier, it may send a response with only the partial capabilities of that user.

The `ImsService` will return the feature tags associated with the 200 OK response for each contact requested.

Presence

As an alternative to a SIP OPTIONS message exchange, a carrier may instead choose to implement a “presence” server, which relies on two mechanisms to provide RCS capability information about client B to client A: A SIP PUBLISH procedure for all RCS clients to publish their RCS capabilities to the presence server for caching and a SIP SUBSCRIBE/NOTIFY mechanism for other clients such as Client A to use to asynchronously request the cached RCS capabilities of their contacts, such as the capabilities of Client B. The procedure for SIP PUBLISH as well as SIP SUBSCRIBE/NOTIFY is documented in RCC.07 and differs slightly depending on the number of contacts being requested, however the device’s capabilities as well as the capabilities of other remote contacts are always communicated using the Presence Information Data Format (PIDF), which is an XML-encoded format. Please see RFC3863 section 4 for more details on each of the elements as well as which elements are mandatory vs optional. RCC.07 further defines how this is used in the context of RCS presence based on presence extensions defined in the OMA DDS Presence Data Extensions document. As mentioned in RCC.07 and RFC3863, carrier extensions to the PIDF are also supported through these standards.

The `ImsService` will pass the entire XML PIDF document to AOSP as the presence documents are being received in subsequent NOTIFYs. Telephony can then parse the XML file and pull out the relevant information to be stored in the EAB database as well as pass the information to applications consuming this information.

Presence - Querying Multiple Contacts

When requesting for the capabilities of multiple contacts, instead of creating an individual SUBSCRIBE request for each contact and expecting a NOTIFY response, the subscriber will use an RLMI (Resource List Meta-Information) document (See RFC4662 for more information). The RLMI document in the SUBSCRIBE request allows a subscriber to subscribe to many resources and receive NOTIFY responses as resources in the resource list change. In the context of presence, the RLMI document in the NOTIFY response will contain multiple PIDF documents for the resources that the subscriber has listed in the initial SUBSCRIBE request resource list. Each individual NOTIFY response may only contain a subset of the presence information subscribed for as the carrier’s resource list server is searching other entities for presence information about one or more of the subscribed resources. Typically, the SUBSCRIBE request will also have a relatively short `expires` header field defined (30-60 seconds) in this context in order to give time for the carrier’s server to find presence information about the subscribed resources as well as not constantly take up resources if the status of one of the resources changes.

It will be the `ImsService’s` responsibility to handle the SUBSCRIBE and subsequent NOTIFY messages from the network as well as parse and interpret the RLMI and PLMI documents that are returned in this case. The AOSP framework will only require the following information in order to properly manage a pending presence request to the `ImsService` for one or more contacts:

- The `RcsContactPresenceCapability` objects as they are parsed from the incoming NOTIFY messages,
- SIP error and reason codes in the event that a SUBSCRIBE request generates an error response,
- The *reason*, and *retry-after* parameters from the final NOTIFY response determined when the *Subscription-State* is *terminated*,
- The *reason* code specified by a *resource* with a *state* of *terminated* and will not be receiving further RLMI information for this subscription.

The presence information should be streamed from the ImsService to the framework as it is being received; the `ImsService` should not wait for the subscription to be terminated.

Capability Discovery

If the carrier has provisioned the device for periodic capability refresh of the user’s contacts, the telephony enhanced address book (EAB) service will periodically schedule a refresh of the RCS capabilities of each contact based on the phone number saved. This requires that the user’s contacts’ phone numbers either need to be periodically sent to the carrier’s presence server in order to retrieve the RCS capabilities of each phone number or a SIP OPTIONS exchange needs to take place for each contact. Applications integrated with RCS will then be able to use the contact capability query API defined below in `RcsUceAdapter` to retrieve this information in order to determine if the contact supports RCS capabilities related to RCS messaging and video calling.

Below is a table defining the carrier configurable optional parameters associated with capability discovery, defined in RCC.07 A.1.9, although there may be more carrier specific parameters not listed here:

Parameter Name	Description
DISABLE-INITIAL-ADDRESS-BOOK-SCAN	When set to **0** (Default value), the EAB service shall perform a capability check for all contacts in the address book when it is

	first started. When set to **1**, the device shall skip the scan and only perform capability exchange requests based on other triggers defined below.
CAPABILITY-INFO-EXPIRY	The amount of time in seconds that the last RCS capabilities fetched from the network are valid for. The default value is 259200 seconds (30 days).
SERVICE-AVAILABILITY-INFO-EXPIRY	The amount of time in seconds that the RCS availability information associated with a contact is valid for. The default value is 60 seconds.
CAPABILITY-DISCOVERY-MECHANISM	If set to **0**, the preferred mechanism for capability discovery is OPTIONS. If set to **1**, the preferred mechanism for capability discovery is presence. If set to **2** or not provided (default), the capability discovery mechanism is disabled.
CAPABILITY-DISCOVERY-ALLOWED-PREFIXES	If absent, all phone numbers shall be considered for capability discovery. If not empty, this configuration provides a list of prefixes or regular expressions that phone numbers must match in order to be considered for capability discovery.
NON-RCS-CAPABILITY-INFO-EXPIRY	If a contact is labeled as non-RCS, this parameter controls how long that label is valid before the RCS capabilities of the contact should be queried again. The default is 2592000 seconds (30 days).

The parameters defined above will affect when the device EAB service queries the RCS capabilities of the user’s contacts. If DISABLE_INITIAL_ADDRESS_BOOK_SCAN is set to 0, the user’s contacts will be scanned and sent to the network using either Presence or SIP OPTIONS, depending on the carrier’s method of capability discovery, defined in CAPABILITY-DISCOVERY-ALLOWED-PREFIXES (as long as the user has opted-in for capability discovery, more information defined below in “Privacy Considerations”).

The following triggers will cause the EAB service to scan the user's contacts to perform capability discovery for one more entries whose cached capabilities do not exist or have expired.

- First time activation of the service after factory reset or sim swap,
- CAPABILITY-INFO-EXPIRY or NON-RCS-CAPABILITY-INFO-EXPIRY has elapsed for one or more contacts,
- New contacts added or modified due to contacts sync,
- New contacts added or modified manually.

Note: If using presence, this will trigger either an individual SUBSCRIBE or RLS list SUBSCRIBE, depending on whether or not the capabilities are being fetched for an individual number or a list of numbers.

If DISABLE_INITIAL_ADDRESS_BOOK_SCAN is set to 1, the framework will not periodically refresh the capabilities of the user's contacts and will only perform a capability request if requested by an external RCS application. Once the capabilities are requested, the cache of the capabilities for the requested contacts will stay valid for the amount of time specified in CAPABILITY-INFO-EXPIRY or NON-RCS-CAPABILITY-INFO-EXPIRY if the RCS application requests the capabilities of that contact again.

An RCS application may also request the "availability" information for a contact, which will bypass the cached capabilities of the contact and perform a capability request. The response will be updated in the capability cache and will also be cached separately in the shorter availability cache, which expires based on the SERVICE-AVAILABILITY-INFO-EXPIRY parameter. This allows applications to get the near real-time RCS status and capabilities of a contact to show real time video calling status to the user or to determine if a contact is online/capable for a service such as RCS 1-to-1 or group messaging.

Telephony will use a private Provider to store the cached contacts in a database since these contacts may need to be cached for days or weeks, depending on the carrier's configuration. Telephony will provide an interface for applications to use to request the capability or availability information of one or more contact URIs. If there is non-stale cached information about these contacts in the telephony enhanced address book (EAB) provider, then this information will be returned immediately. If the request is for a URI that has cached information that is either stale or doesn't exist, telephony will then request that the `ImsService` performs a capability request for the contact URIs that the application is requesting capabilities for. Figure 3.5.2 illustrates how telephony will cache the capabilities of the contacts. Figure 3.5.3 shows how AOSP will cache real-time capability and availability requests, which expire based on the configuration options used in the above table.

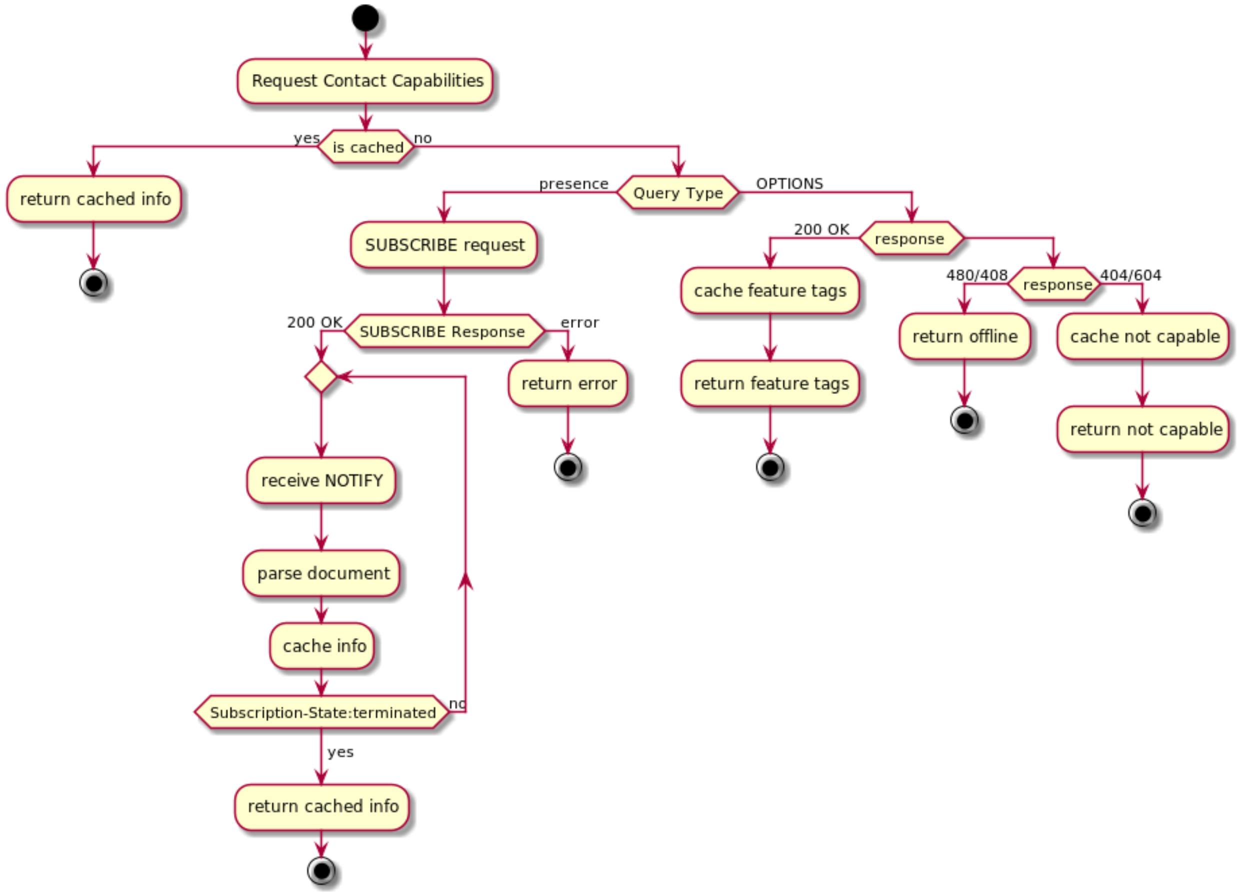
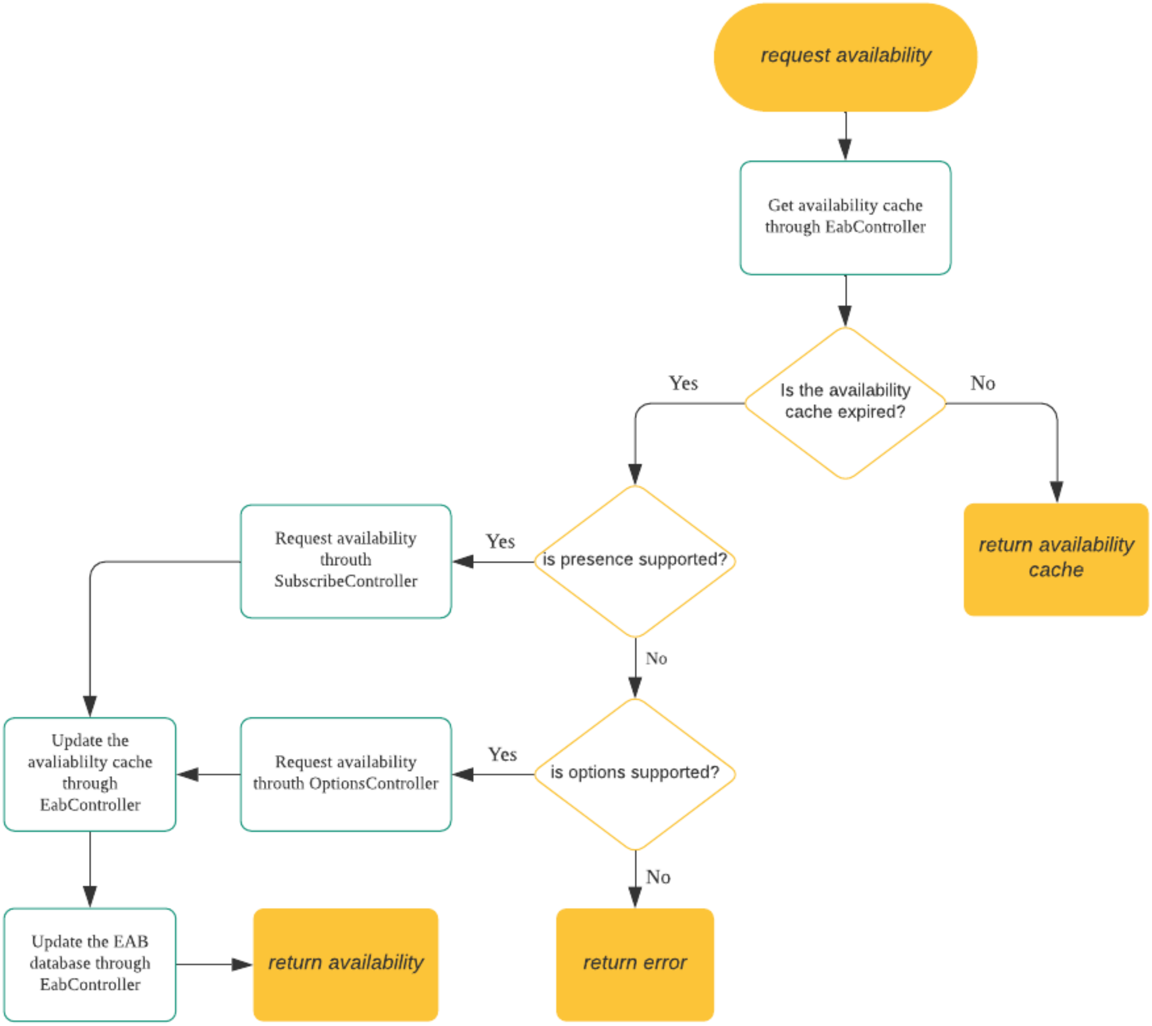


FIgure 3.5.2: Flow diagram showing the caching behavior of AOSP for the RCS capabilities of the user’s contacts.


request availability
Get availability cache through EabController
Is the availability cache expired?
Yes
No
return availability cache
is presence supported?
Yes
Request availability throuth SubscribeController
No
is options supported?
Yes
Request availability throuth OptionsController
Update the avaliablilty cache through EabController
Update the EAB database through EabController
return availability
No
return error

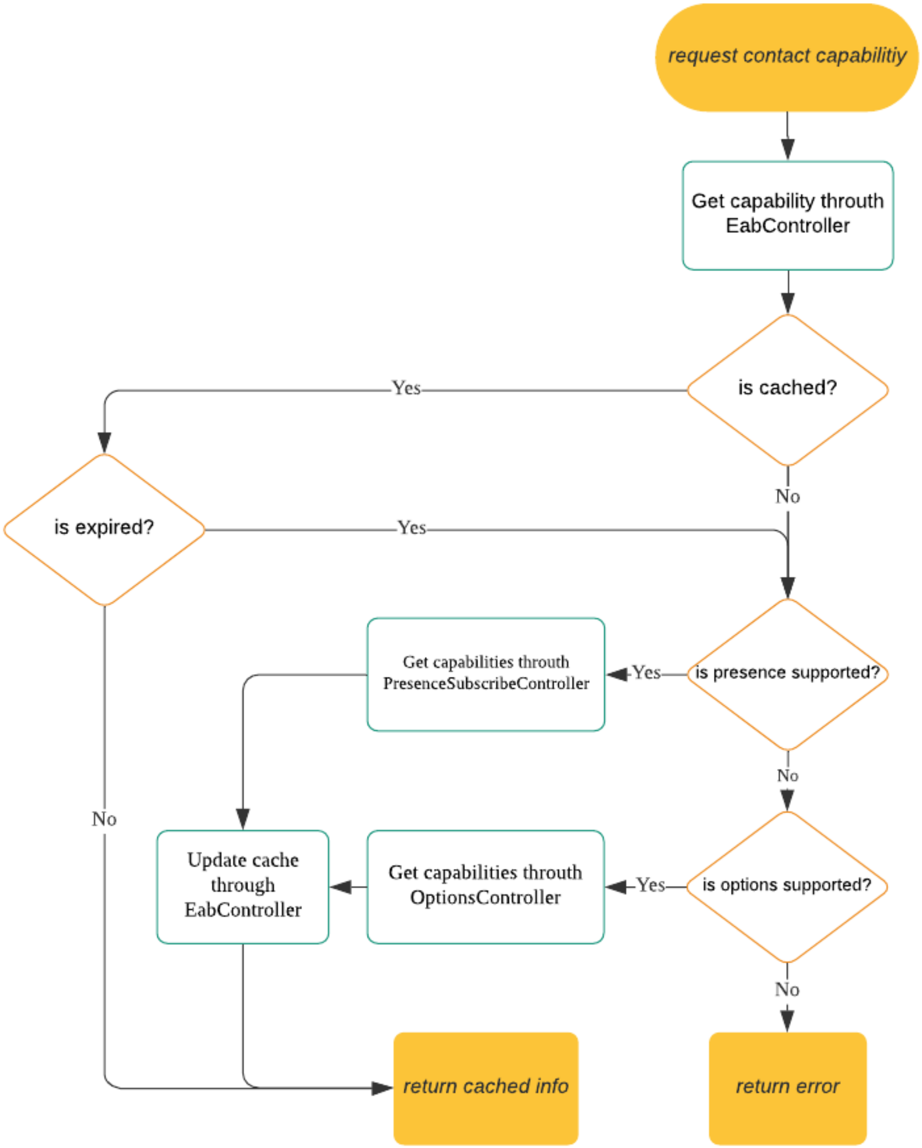

request contact capabilitiy
Get capability throuth EabController
is cached?
Yes
No
is expired?
Yes
No
is presence supported?
Yes
Get capabilities throuth PresenceSubscribeController
No
is options supported?
Yes
Get capabilities throuth OptionsController
Update cache through EabController
No
return cached info
return error

FIgure 3.5.3: Flow diagram showing the caching behavior of AOSP for real-time availability (top) and capability requests (bottom) of a contact.

For carriers that require that the database of cached contact capabilities to be purged when a subscription is removed, the database will be purged of cached contact information related to that subscription only. Other cached information for other subscriptions will not be purged. For carriers that do not require the cached contact capabilities to be purged, the information will stay in the database, however it will not be updated or used for capability requests from RCS applications. If a user swaps the SIM card back to a subscription where the database of cached information has not been purged, the EAB service will only perform a capability fetch for contacts whose cached contact information has exceeded the expiration time set in CAPABILITY-INFO-EXPIRY.

Privacy Considerations

The RCS universal profile specficiation RCC.71 as well as RCC.07 specify that by default, the device should perform a full address book scan of the user's contacts to get the RCS capabilities of each one if DISABLE_INITIAL_ADDRESS_BOOK_SCAN is set to 0. Starting in Android R, this is no longer enough justification to upload the phone numbers associated with all the contacts in the user's address book. The user must also separately consent to this periodic upload of contact information on the device. In Android S, this will continue to be enforced and the RCS messaging application will also need to get user consent before the contact presence polling service can be started. The messaging application will be able to use the `ImsRcsManager#ACTION_SHOW_CAPABILITY_DISCOVERY_OPT_IN` Intent to show the settings screen, where the user can choose to enable capability discovery before being able to use RCS services. This opt-in will not affect availability requests from RCS applications, however, as these requests are on-demand and required for RCS functionality to work. In order for on-demand requests from the RCS application to be completed by the framework UCE service, the RCS application must have been granted CONTACTS permission.

Publishing Presence Information from RCS Applications

Along with requesting the RCS capabilities of another contact, the device must also provide its RCS capabilities so that other devices can perform the same look up to look up the user's RCS capabilities. When using a presence server, the user's device sends a SIP PUBLISH request to the carrier's network periodically to refresh the network's cached information on the user's RCS capabilities. This periodic interval is governed by the SOURCE-THROTTLE-PUBLISH configuration parameter. See figure 3.5.8 below for the typical PUBLISH sequence.

If using the OPTIONS mechanism, the device caches the RCS capabilities of the device locally and responds to the request with a 200 OK containing the cached capabilities. See figure 3.5.10 for the typical OPTIONS exchange sequence.

RCS applications will not directly provide the features that they wish telephony to PUBLISH to the network or respond to via SIP OPTIONS. These features will be determined by the telephony framework based off of the Feature Tags that the RCS application is currently associated with. The related SIP PUBLISH will follow after the first IMS registration as well as after any IMS registration modification that occurs. Since RCS applications do not modify the PUBLISHed tags directly, they will need to tear down and then recreate a new SipDelegate if they no longer support a specific feature tag. This will generate a new PUBLISH on the network following the modification of the IMS registration. Figure 3.5.4 below outlines the expected operation of UCE based on changes to the IMS registration due to the state of the SipDelegates.

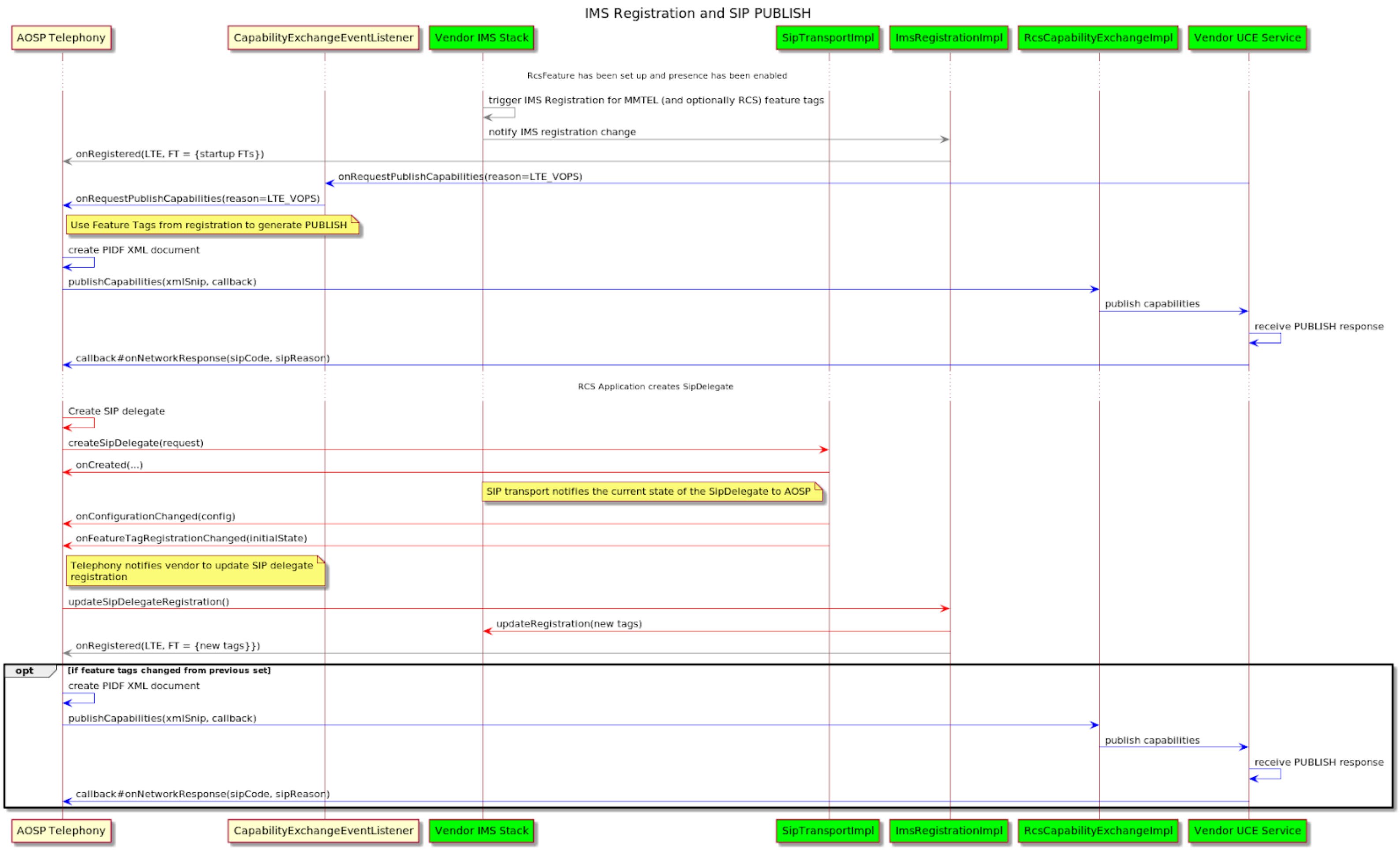

IMS Registration and SIP PUBLISH
AOSP Telephony
CapabilityExchangeEventListener
Vendor IMS Stack
SipTransportImpl
ImsRegistrationImpl
RcsCapabilityExchangeImpl
Vendor UCE Service
RcsFeature has been set up and presence has been enabled
trigger IMS Registration for MMTEL (and optionally RCS) feature tags
notify IMS registration change
onRegistered(LTE, FT = {startup FTs})
onRequestPublishCapabilities(reason=LTE_VOPS)
onRequestPublishCapabilities(reason=LTE_VOPS)
Use Feature Tags from registration to generate PUBLISH
create PIDF XML document
publishCapabilities(xmlSnip, callback)
publish capabilities
receive PUBLISH response
callback#onNetworkResponse(sipCode, sipReason)
RCS Application creates SipDelegate
Create SIP delegate
createSipDelegate(request)
onCreated(...)
SIP transport notifies the current state of the SipDelegate to AOSP
onConfigurationChanged(config)
onFeatureTagRegistrationChanged(initialState)
Telephony notifies vendor to update SIP delegate registration
updateSipDelegateRegistration()
updateRegistration(new tags)
onRegistered(LTE, FT = {new tags}})
opt
[if feature tags changed from previous set]
create PIDF XML document
publishCapabilities(xmlSnip, callback)
publish capabilities
receive PUBLISH response
callback#onNetworkResponse(sipCode, sipReason)
AOSP Telephony
CapabilityExchangeEventListener
Vendor IMS Stack
SipTransportImpl
ImsRegistrationImpl
RcsCapabilityExchangeImpl
Vendor UCE Service

Figure 3.5.4: The expected operation of a SIP PUBLISH in response to a SIP delegate being created. To help distinguish these paths, existing operational paths are in gray, new UCE related sequences are in blue, and new SIP transport related operations are in red.

In the case that the default messaging application has changed, the following procedure will be followed:

1. Default Messaging Application change notification is sent to framework telephony.
2. The old Default Messaging Application then destroys its SipDelegateConnection, which removes those RCS feature tags from being tracked by the ImsService.
3. After the old Default Messaging Application destroys its SipDelegateConnection, a timer will be started to trigger a new IMS registration and PUBLISH. During this time, the new Default Messaging Application should create its own SipDelegateConnection.
4. Once the IMS registration and PUBLISH timer expires, the new IMS registration and PUBLISH will be triggered. A new IMS registration and PUBLISH will only be sent to the network if the feature tags of the new Default Messaging Application differ from the old set. This will reduce the number of back-to-back PUBLISHes on the network.

See figure 3.5.5 below for more information about when a SIP PUBLISH is generated after a DMA change.

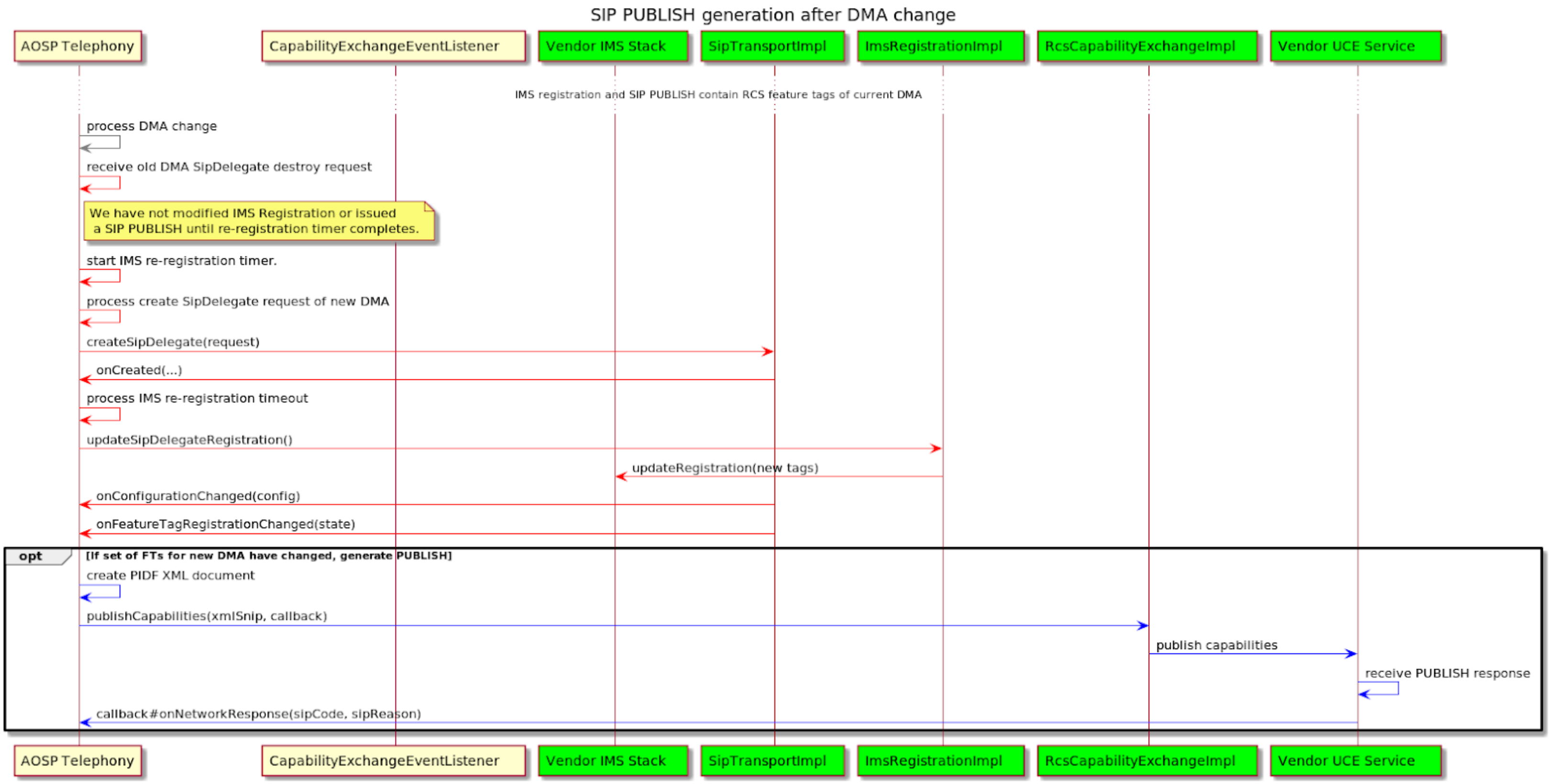

SIP PUBLISH generation after DMA change
AOSP Telephony
CapabilityExchangeEventListener
Vendor IMS Stack
SipTransportImpl
ImsRegistrationImpl
RcsCapabilityExchangeImpl
Vendor UCE Service
IMS registration and SIP PUBLISH contain RCS feature tags of current DMA
process DMA change
receive old DMA SipDelegate destroy request
We have not modified IMS Registration or issued
a SIP PUBLISH until re-registration timer completes.
start IMS re-registration timer.
process create SipDelegate request of new DMA
createSipDelegate(request)
onCreated(...)
process IMS re-registration timeout
updateSipDelegateRegistration()
updateRegistration(new tags)
onConfigurationChanged(config)
onFeatureTagRegistrationChanged(state)
opt
[If set of FTs for new DMA have changed, generate PUBLISH]
create PIDF XML document
publishCapabilities(xmlSnip, callback)
publish capabilities
receive PUBLISH response
callback#onNetworkResponse(sipCode, sipReason)
AOSP Telephony
CapabilityExchangeEventListener
Vendor IMS Stack
SipTransportImpl
ImsRegistrationImpl
RcsCapabilityExchangeImpl
Vendor UCE Service

Figure 3.5.6: Sequence of events that occur after the Default Messaging Application (DMA) changes. The events in red are related to SIP Transport signalling and the events in blue are related to UCE PUBLISH.

APIs

The ImsService API will be extended to allow the system IMS stack to support RCS capabilities from contacts as well as publish UCE state updates to the IMS stack. Applications such as dialer, contacts, and messaging apps will be able to access new APIs through ImsManager, which will allow them to query the RCS capabilities of one or more contact URIs. In single registration mode, the framework will also support allowing RCS features managed by RCS applications using SipDelegateConnections to publish UCE updates for those associated feature tags only.

ImsService API Changes

The ImsService API will be extended to support both presence and OPTIONS forms of UCE. See Figure 3.5.6 below for a summary of the new API surface.

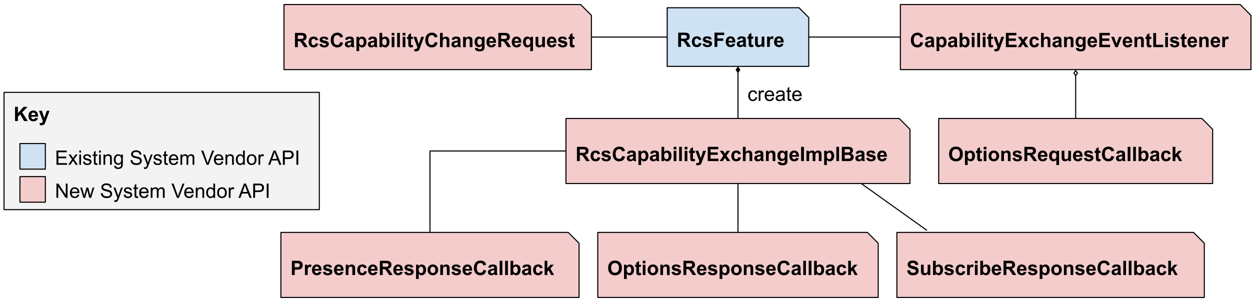

RcsCapabilityChangeRequest
RcsFeature
CapabilityExchangeEventListener
create
Key
Existing System Vendor API
New System Vendor API
RcsCapabilityExchangeImplBase
OptionsRequestCallback
PresenceResponseCallback
OptionsResponseCallback
SubscribeResponseCallback

Figure 3.5.6: The modified system API surface used for UCE as part of the ImsService API.

Signalling UCE capability status to the framework

The first component is the additional APIs added to the RcsFeature class, which has the responsibility of creating an implementation of RcsCapabilityExchangeImplBase that is implemented by the vendor as well as signalling to the RcsFeature which types of capability exchange the carrier supports. The RcsFeature can also respond to the framework and notify the framework of the capability status for UCE. The framework will only start using the RcsCapabilityExchangeImplBase implementation once the RcsFeature signals that the capability status is enabled. This is similar to how MmTelFeature notifies the framework of the capability status for voice, video, Ut, and SMS over IMS. The sequence diagram in figure 3.5.7 below illustrates this process.

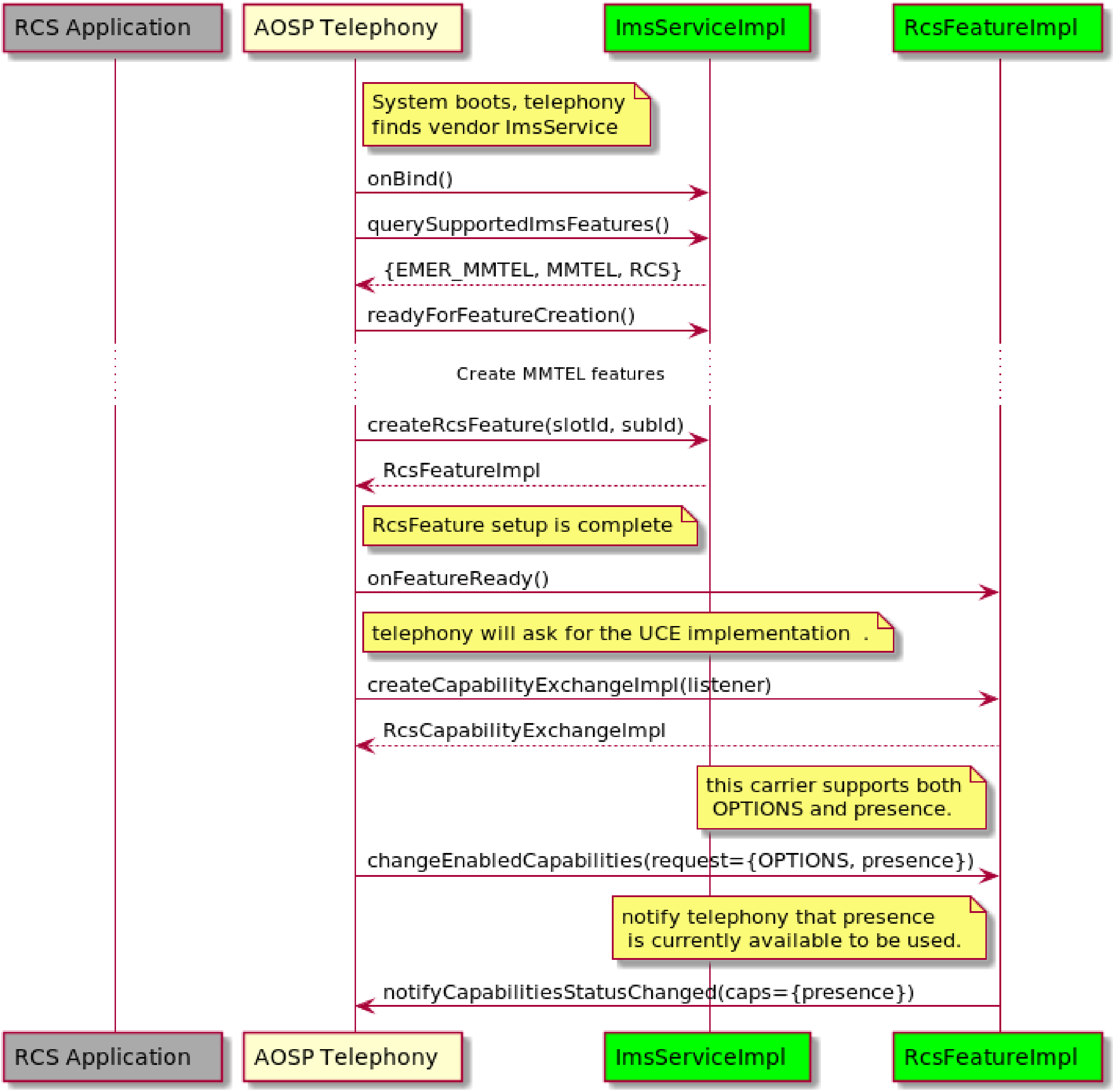

RCS Application
AOSP Telephony
ImsServiceImpl
RcsFeatureImpl
System boots, telephony finds vendor ImsService
onBind()
querySupportedImsFeatures()
{EMER_MMTEL, MMTEL, RCS}
readyForFeatureCreation()
Create MMTEL features
createRcsFeature(slotId, subId)
RcsFeatureImpl
RcsFeature setup is complete
onFeatureReady()
telephony will ask for the UCE implementation .
createCapabilityExchangeImpl(listener)
RcsCapabilityExchangeImpl
this carrier supports both OPTIONS and presence.
changeEnabledCapabilities(request={OPTIONS, presence})
notify telephony that presence is currently available to be used.
notifyCapabilitiesStatusChanged(caps={presence})
RCS Application
AOSP Telephony
ImsServiceImpl
RcsFeatureImpl

Figure 3.5.7: The typical setup flow of the `RcsFeature` associated with the `ImsService` and creation of the vendor `RcsCapabilityExchangeImplBase` implementation.

Integrating User Capability Exchange into the Platform

The second component is the UCE implementation of `RcsCapabilityExchangeImplBase` provided by the vendor `ImsService` as well as the `CapabilityExchangeEventListener`, which allows the vendor to notify telephony when there is a new event. Figures 3.5.8 - 3.5.11

below show the expected operation of these APIs.

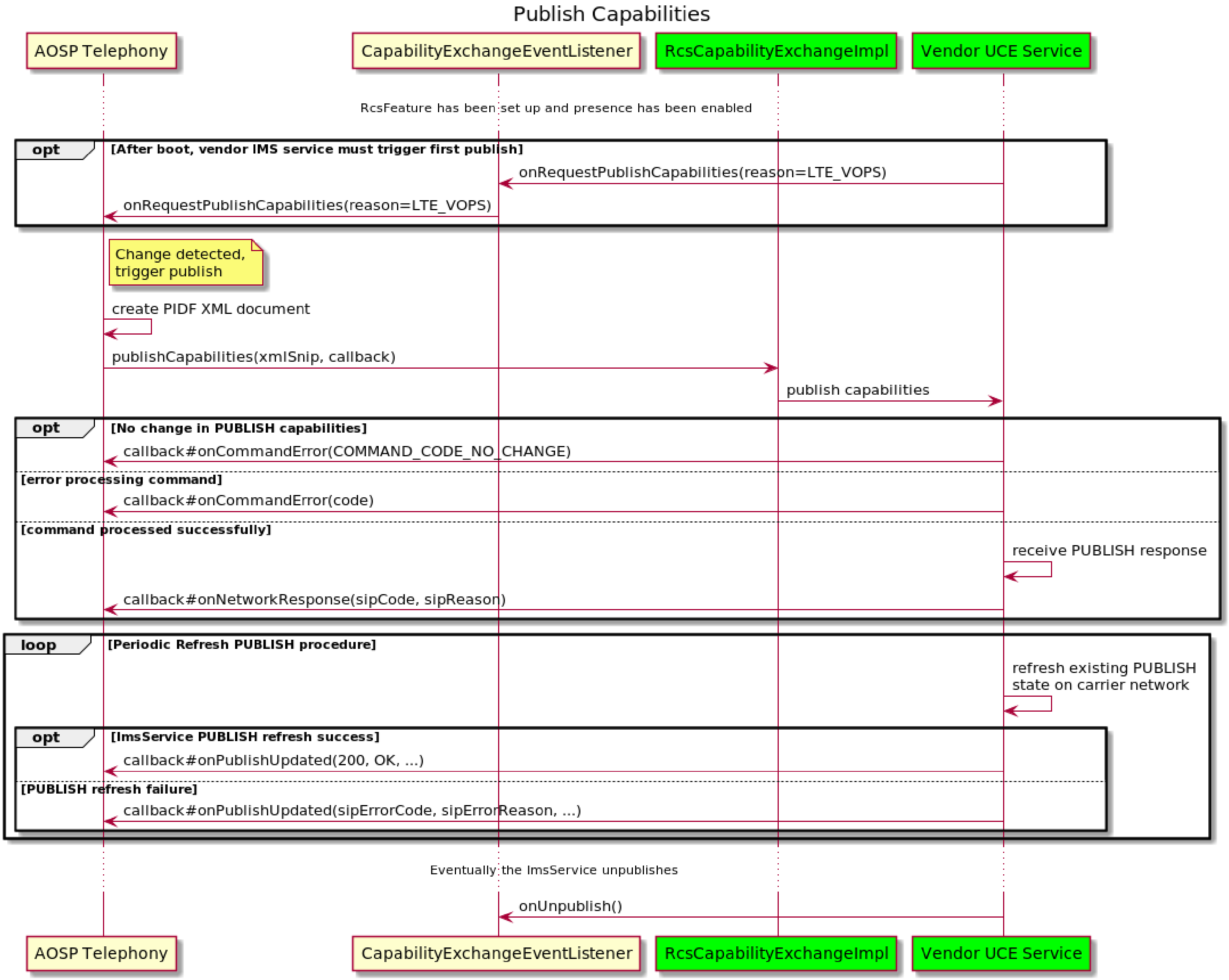

Publish Capabilities
AOSP Telephony
CapabilityExchangeEventListener
RcsCapabilityExchangeImpl
Vendor UCE Service
RcsFeature has been set up and presence has been enabled
opt
[After boot, vendor IMS service must trigger first publish]
onRequestPublishCapabilities(reason=LTE_VOPS)
onRequestPublishCapabilities(reason=LTE_VOPS)
Change detected,
trigger publish
create PIDF XML document
publishCapabilities(xmlSnip, callback)
publish capabilities
opt
[No change in PUBLISH capabilities]
callback#onCommandError(COMMAND_CODE_NO_CHANGE)
[error processing command]
callback#onCommandError(code)
[command processed successfully]
receive PUBLISH response
callback#onNetworkResponse(sipCode, sipReason)
loop
[Periodic Refresh PUBLISH procedure]
refresh existing PUBLISH
state on carrier network
opt
[ImsService PUBLISH refresh success]
callback#onPublishUpdated(200, OK, ...)
[PUBLISH refresh failure]
callback#onPublishUpdated(sipErrorCode, sipErrorReason, ...)
Eventually the ImsService unpublishes
onUnpublish()
AOSP Telephony
CapabilityExchangeEventListener
RcsCapabilityExchangeImpl
Vendor UCE Service

Figure 3.5.8: Expected operation of the `ImsService` during PUBLISH operation for presence.

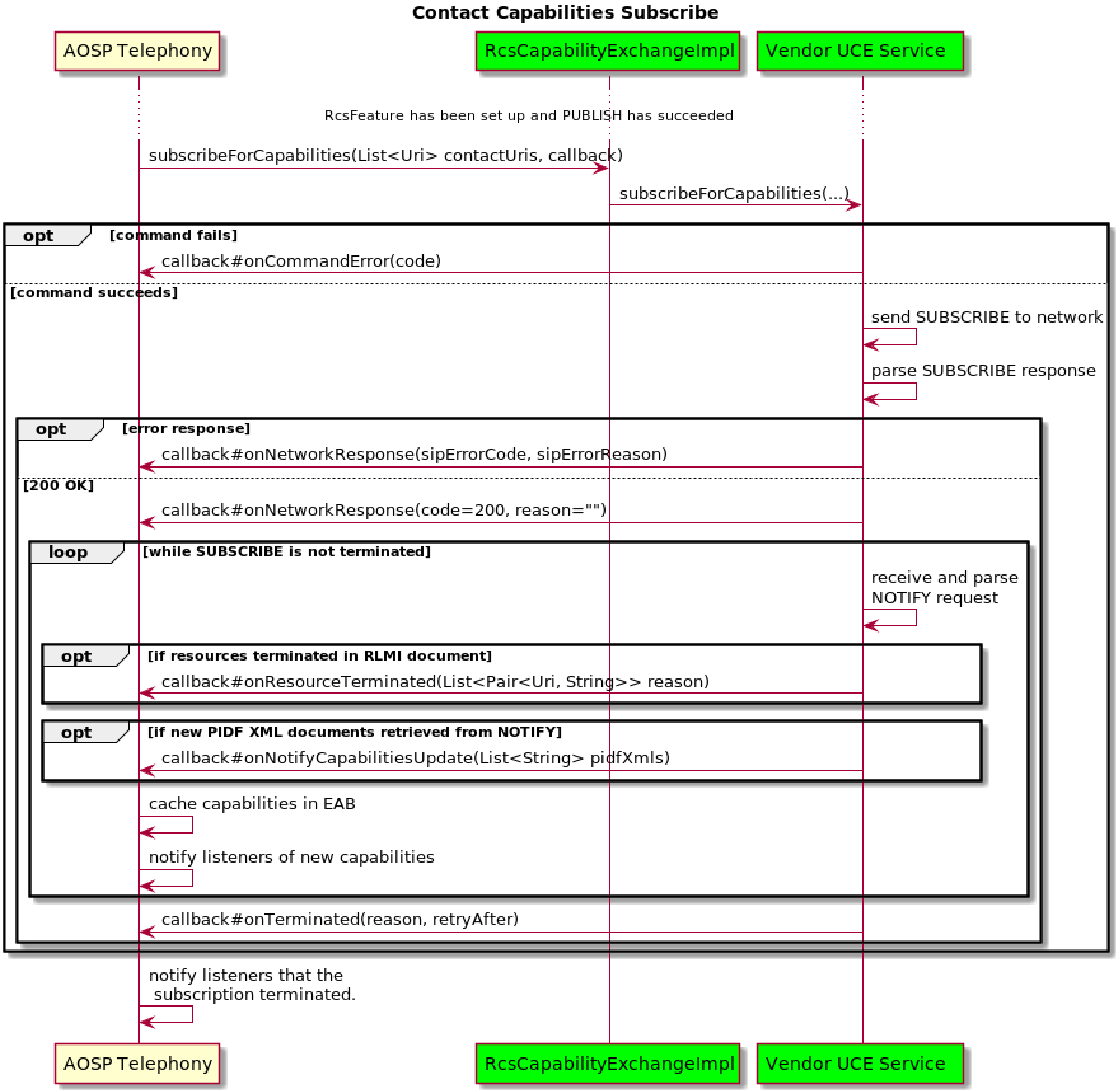

Contact Capabilities Subscribe
AOSP Telephony
RcsCapabilityExchangeImpl
Vendor UCE Service
RcsFeature has been set up and PUBLISH has succeeded
subscribeForCapabilities(List<Uri> contactUris, callback)
subscribeForCapabilities(...)
opt
[command fails]
callback#onCommandError(code)
[command succeeds]
send SUBSCRIBE to network
parse SUBSCRIBE response
opt
[error response]
callback#onNetworkResponse(sipErrorCode, sipErrorReason)
[200 OK]
callback#onNetworkResponse(code=200, reason="")
loop
[while SUBSCRIBE is not terminated]
receive and parse NOTIFY request
opt
[if resources terminated in RLMI document]
callback#onResourceTerminated(List<Pair<Uri, String>> reason)
opt
[if new PIDF XML documents retrieved from NOTIFY]
callback#onNotifyCapabilitiesUpdate(List<String> pidfXmls)
cache capabilities in EAB
notify listeners of new capabilities
callback#onTerminated(reason, retryAfter)
notify listeners that the subscription terminated.
AOSP Telephony
RcsCapabilityExchangeImpl
Vendor UCE Service

Figure 3.5.9: Expected operation of the `ImsService` during SUBSCRIBE operation for presence.

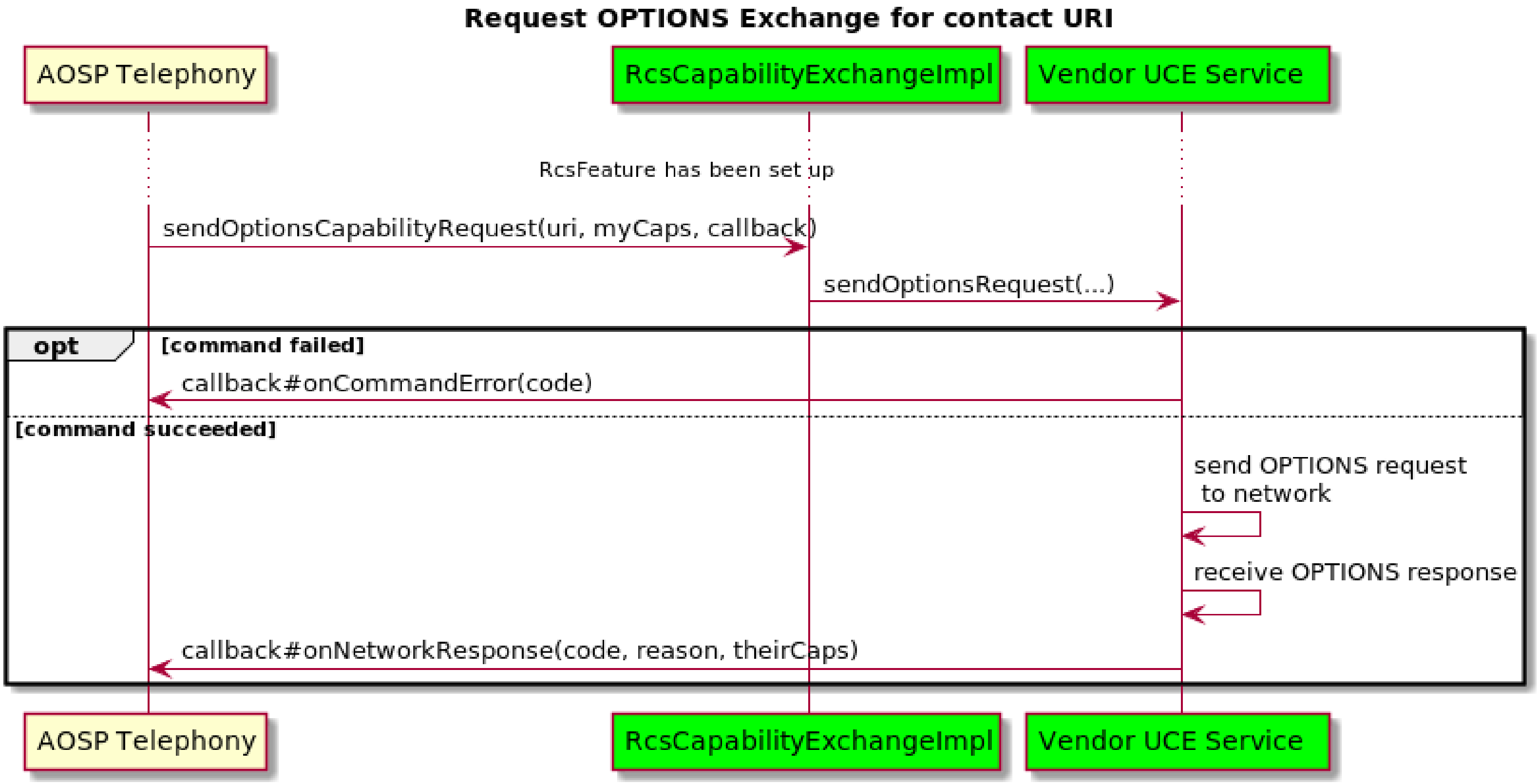

Request OPTIONS Exchange for contact URI
AOSP Telephony
RcsCapabilityExchangeImpl
Vendor UCE Service
RcsFeature has been set up
sendOptionsCapabilityRequest(uri, myCaps, callback)
sendOptionsRequest(...)
opt
[command failed]
callback#onCommandError(code)
[command succeeded]
send OPTIONS request
to network
receive OPTIONS response
callback#onNetworkResponse(code, reason, theirCaps)
AOSP Telephony
RcsCapabilityExchangeImpl
Vendor UCE Service

Figure 3.5.10: Expected operation of the `ImsService` when sending out a SIP OPTIONS request.

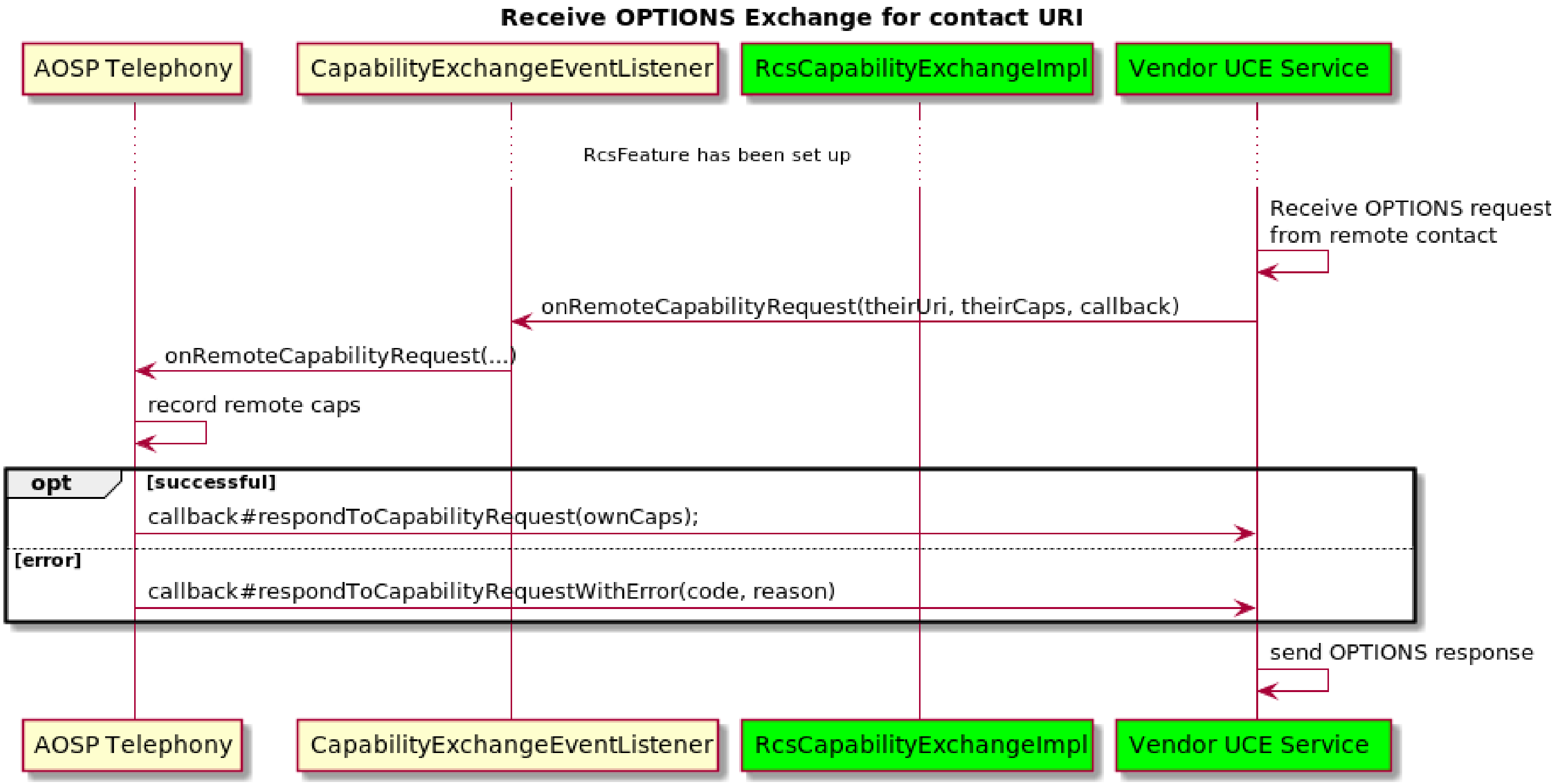

Receive OPTIONS Exchange for contact URI
AOSP Telephony
CapabilityExchangeEventListener
RcsCapabilityExchangeImpl
Vendor UCE Service
RcsFeature has been set up
Receive OPTIONS request from remote contact
onRemoteCapabilityRequest(theirUri, theirCaps, callback)
onRemoteCapabilityRequest(...)
record remote caps
opt
[successful]
callback#respondToCapabilityRequest(ownCaps);
[error]
callback#respondToCapabilityRequestWithError(code, reason)
send OPTIONS response
AOSP Telephony
CapabilityExchangeEventListener
RcsCapabilityExchangeImpl
Vendor UCE Service

Figure 3.5.11: Expected operation of the `ImsService` during when receiving a SIP OPTIONS request.

Application API Changes

On the RCS application side, the `RcsUceAdpater` will allow RCS applications to request the capabilities of one or more contacts from the EAB provider or bypass the EAB cache and request the network capabilities of one contact from the network for an availability query.

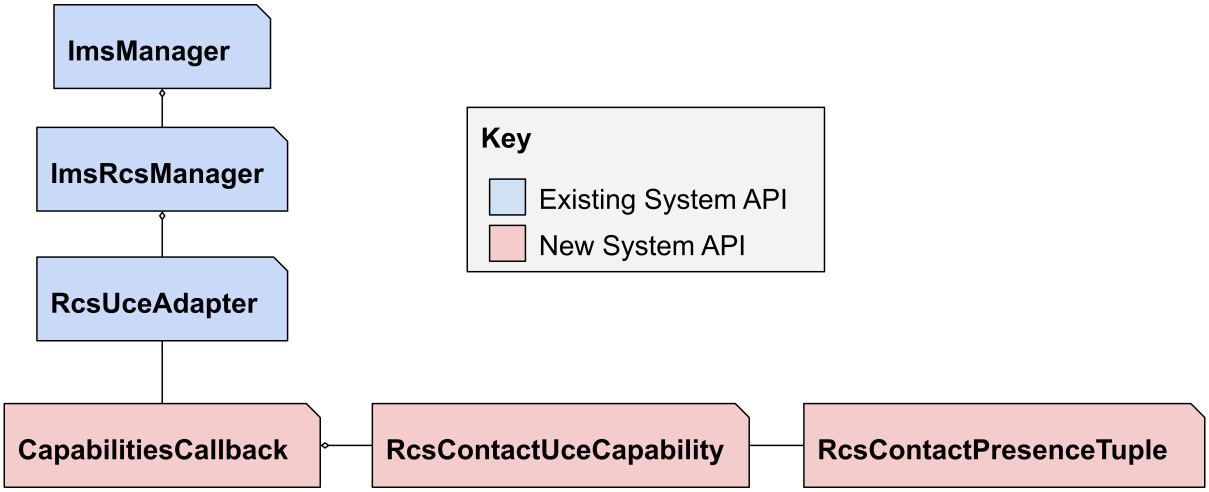


Figure 3.5.12: Application API changes to support UCE capability queries and capability updates for interested applications.

RCS applications will be able to use `ImsRcsManager` to query the state of the capabilities of the vendor `ImsService` that is associated with the cellular subscription that the instance was created for. It also allows the RCS application to get the associated `RcsUceAdapter`, which is used for UCE related features. The capability/availability callbacks in the `ImsRcsManager` is **not** the same as the capability/availability related to UCE and is a more general concept that is used throughout all features in the `ImsService`. `ImsRcsManager#isCapable` is used to query AOSP for whether or not carrier configurations and telephony configurations have enabled a specific feature. At this time, the only defined features for the `ImsService` are:

- UCE via presence exchange and
- UCE via OPTIONS exchange.

The `ImsRcsManager#isCapable` API does not determine if UCE via presence or OPTIONS is available at the current time, only that the carrier supports this feature and telephony has enabled the feature for the vendor ImsService.

The `ImsRcsManager#isAvailable` callback can be used to determine if the feature is capable **and** the `ImsService` has successfully set up the feature. When this method returns `true` for a specific feature, it is available to be used at the current time. Similarly, the `ImsRcsManager#AvailabilityCallback` allows an application to get callbacks when a specific feature has moved to available/unavailable and can use this information to start or stop services.

Figure 3.9.13 below shows how these APIs may be used by the RCS application.

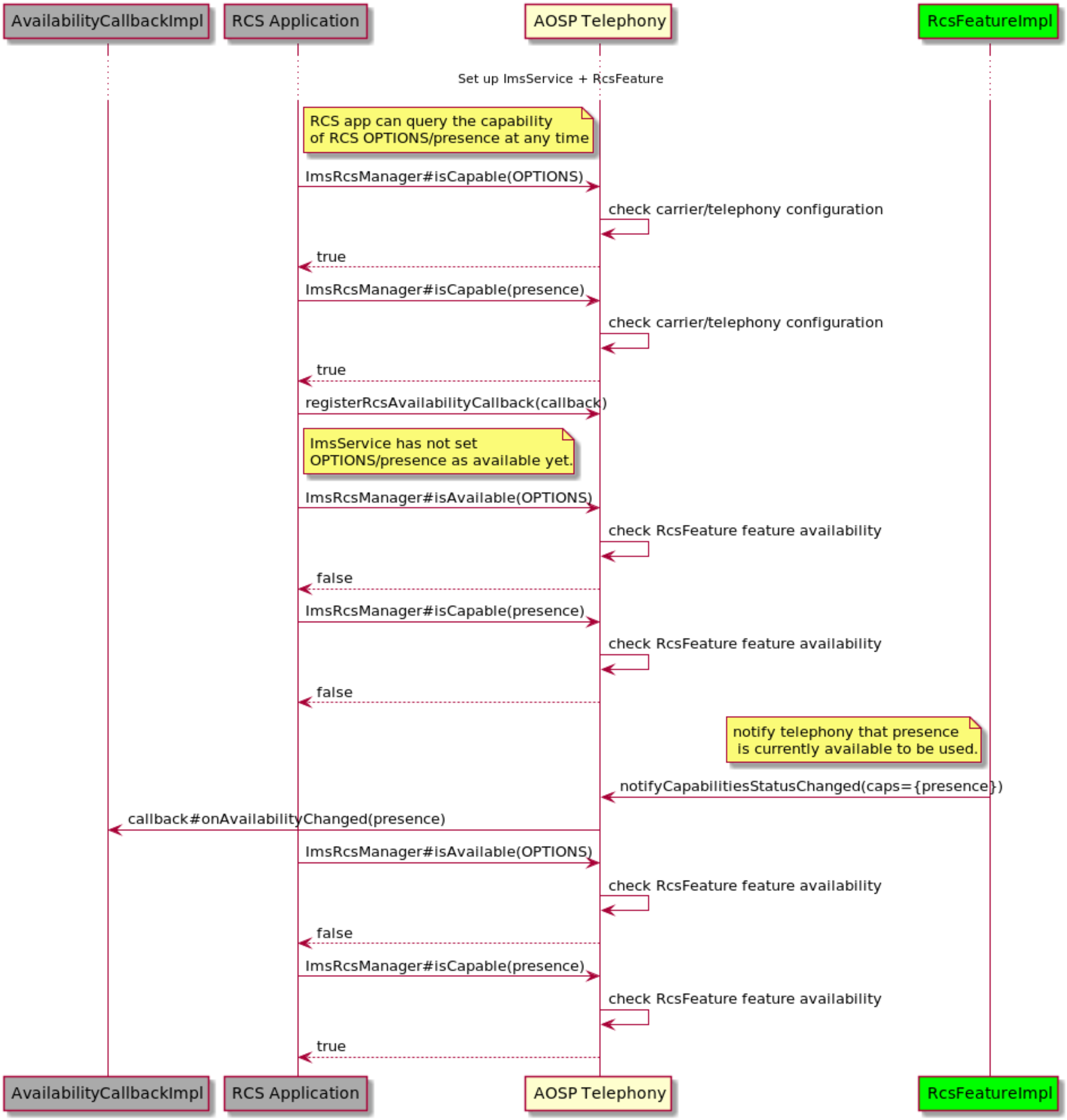

AvailabilityCallbackImpl
RCS Application
AOSP Telephony
RcsFeatureImpl
Set up ImsService + RcsFeature
RCS app can query the capability
of RCS OPTIONS/presence at any time
ImsRcsManager#isCapable(OPTIONS)
check carrier/telephony configuration
true
ImsRcsManager#isCapable(presence)
check carrier/telephony configuration
true
registerRcsAvailabilityCallback(callback)
ImsService has not set
OPTIONS/presence as available yet.
ImsRcsManager#isAvailable(OPTIONS)
check RcsFeature feature availability
false
ImsRcsManager#isCapable(presence)
check RcsFeature feature availability
false
notify telephony that presence
is currently available to be used.
notifyCapabilitiesStatusChanged(caps={presence})
callback#onAvailabilityChanged(presence)
ImsRcsManager#isAvailable(OPTIONS)
check RcsFeature feature availability
false
ImsRcsManager#isCapable(presence)
check RcsFeature feature availability
true
AvailabilityCallbackImpl
RCS Application
AOSP Telephony
RcsFeatureImpl

Figure 3.9.13: How the `ImsRcsManager` APIs can be used to query the capability and availability of certain RCS features implemented by the vendor `ImsService`.

Below are two examples of using the `RcsUceAdapter` for capability fetch as well as availability fetch. The first is the result the RCS application should expect when calling `RcsUceAdapter#requestCapabilities`, as shown in Figure 3.9.14 below. Note the list

SUBSCRIBE will be shortened to a single SUBSCRIBE/NOTIFY procedure if the capabilities of only a single contact needs to be retrieved. The second example is what the RCS application should expect when performing a single contact query for live network availability, as shown in Figure 3.9.15 below. Note that this query bypasses the normal capability cache and first checks the separate availability cache, which expires after SERVICE-AVAILABILITY-INFO-EXPIRY seconds and does not automatically refresh.

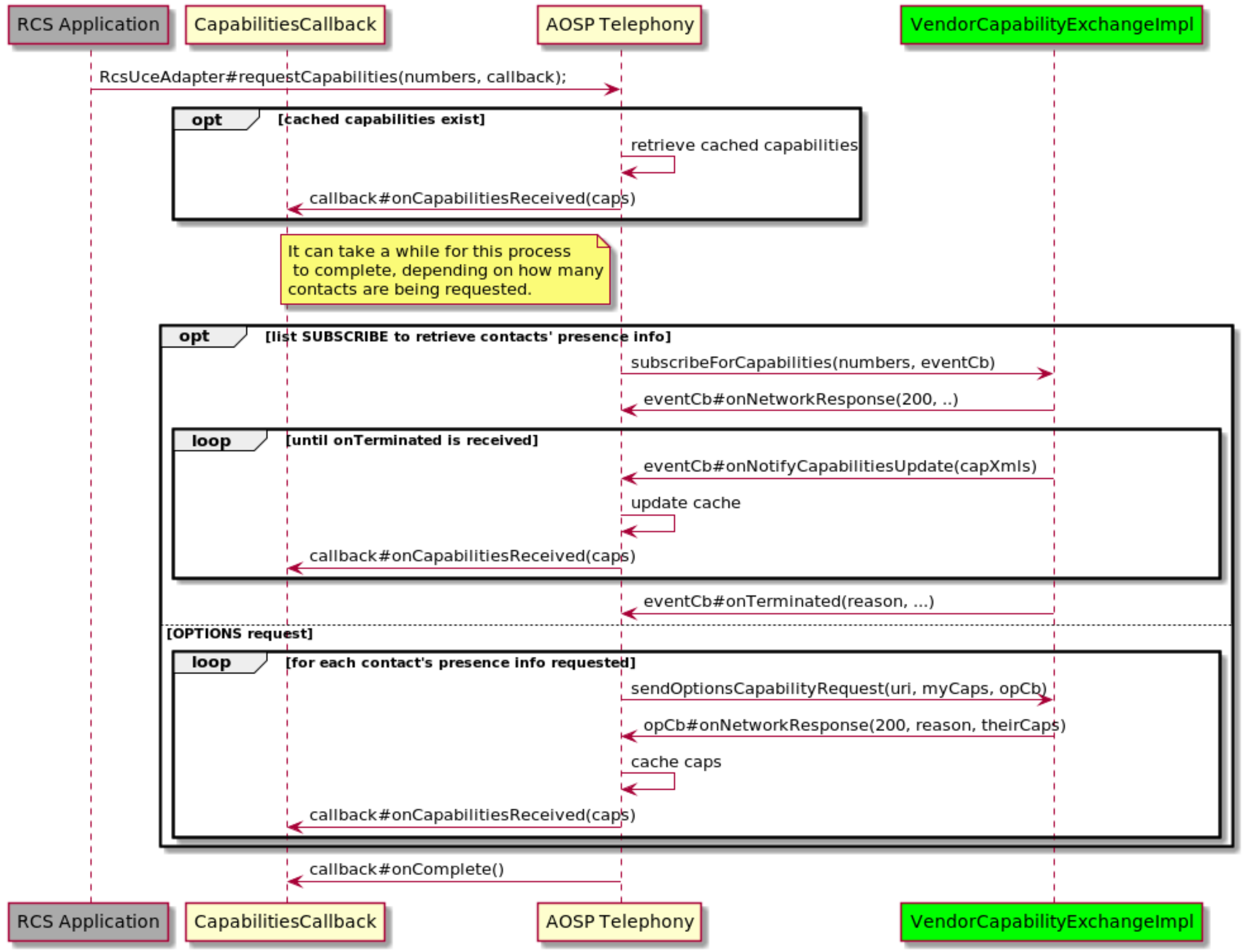

RCS Application
CapabilitiesCallback
AOSP Telephony
VendorCapabilityExchangeImpl
RcsUceAdapter#requestCapabilities(numbers, callback);
opt
[cached capabilities exist]
retrieve cached capabilities
callback#onCapabilitiesReceived(caps)
It can take a while for this process
to complete, depending on how many
contacts are being requested.
opt
[list SUBSCRIBE to retrieve contacts' presence info]
subscribeForCapabilities(numbers, eventCb)
eventCb#onNetworkResponse(200, ..)
loop
[until onTerminated is received]
eventCb#onNotifyCapabilitiesUpdate(capXmls)
update cache
callback#onCapabilitiesReceived(caps)
eventCb#onTerminated(reason, ...)
[OPTIONS request]
loop
[for each contact's presence info requested]
sendOptionsCapabilityRequest(uri, myCaps, opCb)
opCb#onNetworkResponse(200, reason, theirCaps)
cache caps
callback#onCapabilitiesReceived(caps)
callback#onComplete()
RCS Application
CapabilitiesCallback
AOSP Telephony
VendorCapabilityExchangeImpl

Figure 3.9.14: The expected result when an application calls `RcsUceAdapter#requestCapabilities` to fetch the cached capabilities of one or more contacts.

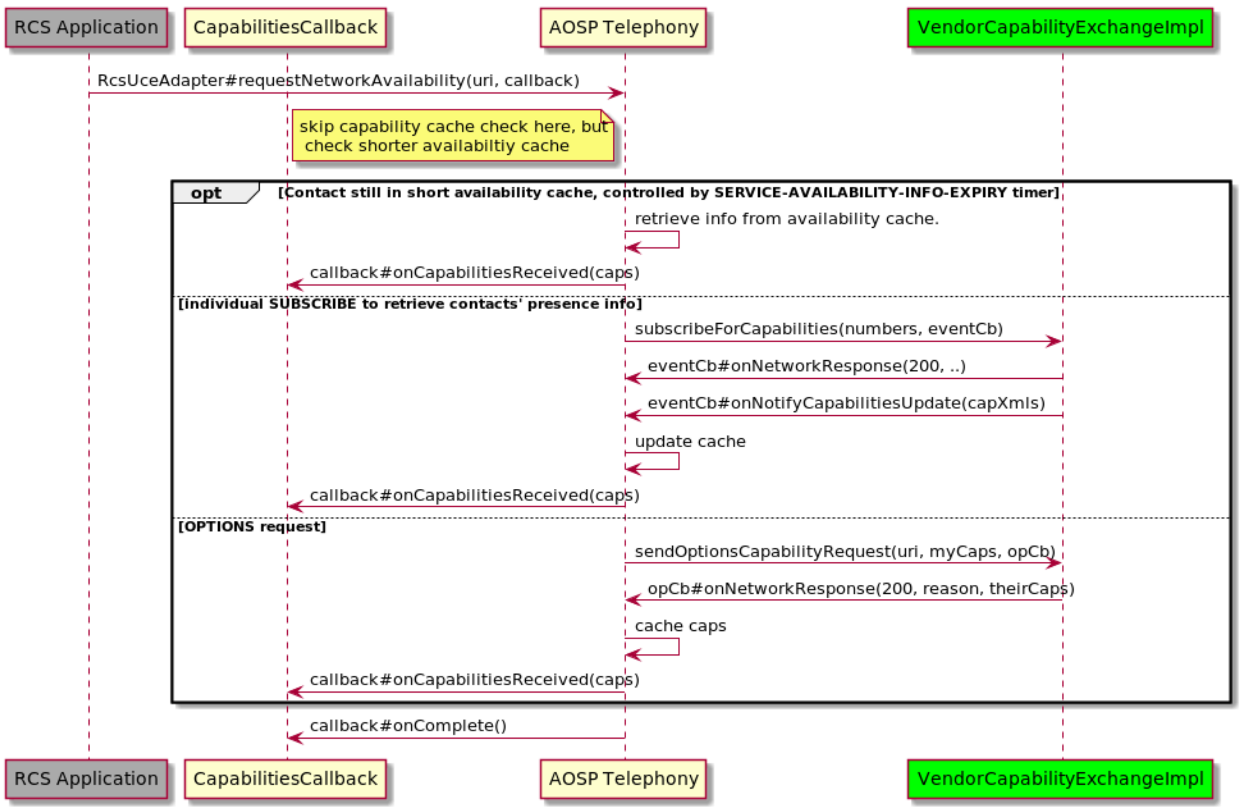

RCS Application
CapabilitiesCallback
AOSP Telephony
VendorCapabilityExchangeImpl
RcsUceAdapter#requestNetworkAvailability(uri, callback)
skip capability cache check here, but
check shorter availabiltiy cache
opt
[Contact still in short availability cache, controlled by SERVICE-AVAILABILITY-INFO-EXPIRY timer]
retrieve info from availability cache.
callback#onCapabilitiesReceived(caps)
[individual SUBSCRIBE to retrieve contacts' presence info]
subscribeForCapabilities(numbers, eventCb)
eventCb#onNetworkResponse(200, ..)
eventCb#onNotifyCapabilitiesUpdate(capXmls)
update cache
callback#onCapabilitiesReceived(caps)
[OPTIONS request]
sendOptionsCapabilityRequest(uri, myCaps, opCb)
opCb#onNetworkResponse(200, reason, theirCaps)
cache caps
callback#onCapabilitiesReceived(caps)
callback#onComplete()
RCS Application
CapabilitiesCallback
AOSP Telephony
VendorCapabilityExchangeImpl

Figure 3.9.15: The expected result when an application calls `RcsUceAdapter#requestNetworkAvailability` to fetch the live network capabilities of a single contact URI.

AOSP and Vendor Responsibilities

Since AOSP does not have an IMS stack that can be used to directly generate the SIP signaling required to implement the PUBLISH, SUBSCRIBE/NOTIFY and OPTIONS procedures required for capability exchange, it must rely on the hybrid approach described in detail above. In general, this means that the AOSP stack manages capability requests from applications, RCS capability state of the device, contact capability refreshing, and PUBLISH trigger/refresh events. The ImsService contains the actual SIP stack and manages the SIP level signalling required to complete the procedures requested by the AOSP telephony stack and report the result of the operation back to telephony.

The **AOSP stack** handles:

- Managing capability/availability requests from many applications and responding with

either the cached data in the EAB database or generating SUBSCRIBE or OPTIONS requests to the `ImsService` for the RCS capabilities of one or more contacts.
- Using the device's IMS/RCS state to trigger PUBLISH events when required as well as generate the PIDF presence document required for PUBLISH as well as handle error conditions generated in response that may require a retry.
- Handling error responses to SUBSCRIBEs and either retrying later or failing permanently, depending on the operator requirements.
- Receiving the NOTIFY response for one or more contacts interpreted by the `ImsService` and using that to update the RCS capabilities and availability of those contacts. In the case of list subscribes requested by AOSP, AOSP can not handle the raw RLMI data from each individual NOTIFY and will instead rely on the `ImsService` to interpret this information and send the appropriate PIDF document for each contact as it is received.
- Parsing PIDF XML documents from the ImsService and updating the EAB database.
- Managing capability and availability parameters received during provisioning for caching and configuration and pushing them to the ImsService.
- Triggering PUBLISH events as required by the operator after initial IMS registration has occurred.
- Generating the feature tags required to request or respond to an OPTIONS request.
- Handling SIP error codes that require a retry of a specific operation after a specific time period.
- Throttling multiple PUBLISH requests as per the SOURCE-THROTTLE-PUBLISH command.

The Vendor **ImsService** handles:

- SIP procedures related to generating the full SIP PUBLISH, SUBSCRIBE, and OPTIONS requests/responses as well as handling NOTIFY requests/responses.
- Handling list SUBSCRIBEs and parsing the RLMI documents contained in the subsequent NOTIFY requests from the network before sending the relevant updates to the framework for the in progress list SUBSCRIBE.
- Generating and handling SIP Entity-Tag related information as required.
- Notifying the AOSP UCE service of RAT changes separately from IMS registration indications. **Note:** The AOSP UCE service may not start PUBLISH procedure until the initial IMS Registration has completed after boot up.
- Cleaning up presence documents and notifying the framework that the presence document has been unpublished before deregistering for IMS.
- Persisting UCE configuration items set via `ImsConfigImplBase` across device reboots.
- Handling SIP error codes that require an immediate retry, such as ones that result from

IMS registration misconfiguration or error codes generated in response to a SIP message misconfiguration.

  - The only exception here is “403 Forbidden” and “489 Bad Event” responses to a PUBLISH Request. In these cases, the AOSP framework will move into a “disabled” state, where we will no longer generate PUBLISH requests based on state change or allow capability requests from applications to be sent to the vendor UCE stack. The AOSP framework will move out of this state when one of the two events occurs:
    - The vendor stack initiates a PUBLISH request, which the AOSP framework will respond to. If the network reports a 200 OK, then the AOSP framework will be enabled again.
    - For some carriers, the “489 Bad Event” header will also require AOSP to set a timer. The device will stay in the disabled state for that amount of time until the timer expires (across reboots) and we generate a PUBLISH request that results in a 200 OK response or the vendor stack initiates a PUBLISH request and the network responds with a 200 OK.

GBA Authentication

Generic bootstrapping Authentication is a shared secret based authentication specified in TS33.220. GBA authentication extends SIM based authentication to web applications (for example using HTTP/HTTPS requests) hence enhancing user security. AKA authentication is used to mutually authenticate the UE and the network and then derive the GBA authentication keys.
Primary use cases considered here are for RCS file transfer, XCAP etc.

Design

Applications shall use the `TelephonyManager` interface to access the new GBA authentication APIs. Service to run GBA authentication for the application could be implemented by the vendor or in AOSP or by a third party. To allow for a flexible architecture, a GBA resolver shall be implemented within the telephony process, which can dynamically link to the available GBA service. Figure 3.6.2: below illustrates the high level flow and the components involved.

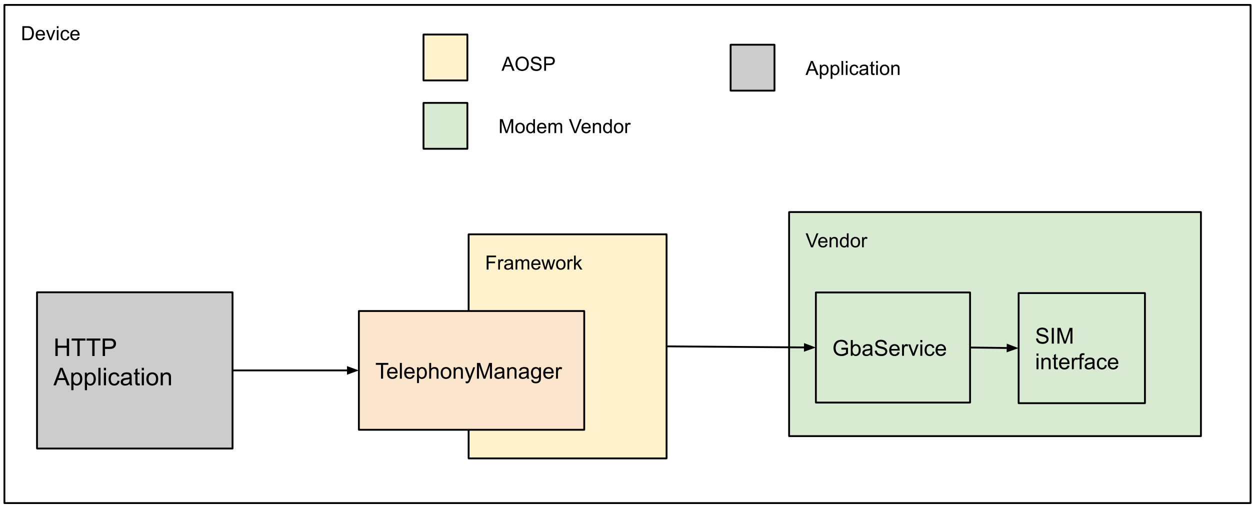
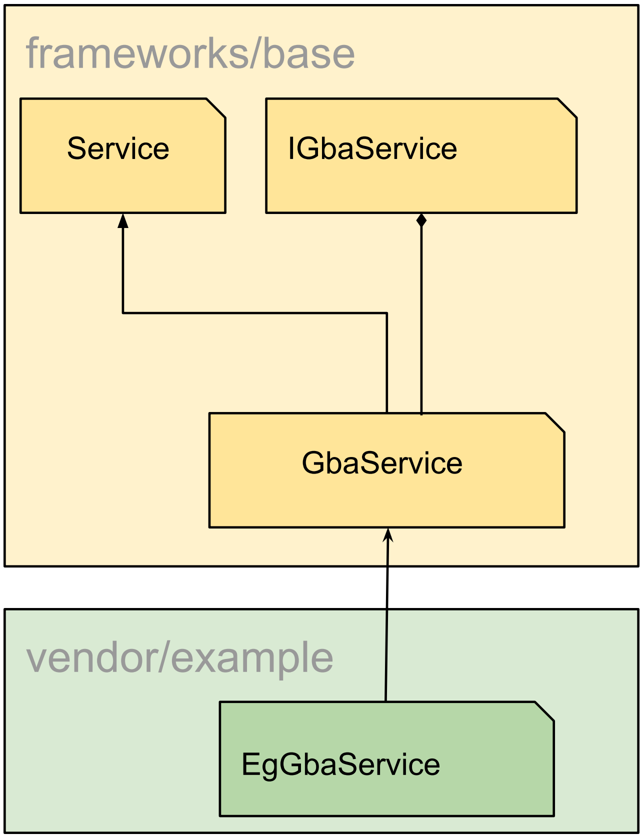


Figure 3.6.2: GBA High level design diagram

GbaService

The `GbaService` will register itself as a service in the AndroidManifest and will be bound to based on the Device configuration. IGbaService AIDL will be defined in the framework for exposing the GBA authentication APIs. GbaService will extend this interface.



Permissions

GBA APIs can be exposed to the OEM approved applications and the applications approved by the carrier. Users will not have the capability to allow an application to run GBA. Applications will need carrier permissions (see hasCarrierPrivileges), PERFORM_IMS_SINGLE_REGISTRATION, or `MODIFY_PHONE_STATE` permissions to use these APIs.

Application APIs

New APIs are defined in `TelephonyManager` for the application to get GBA keys. Application must register a callback to receive the keys for each request. If the bootstrapping procedure was already completed and a valid key is available corresponding to the NAF, the application callback shall be invoked immediately.

In order to ensure that the GBA keys are valid at the time of use, it is recommended for the application to always invoke the API and get the keys at the time of use, instead of caching the keys.
Each request from the application to get the GBA keys is treated independently and must have a unique callback. If multiple calls are made to request the keys before receiving the keys for the previous request, then the requests shall be processed in the order in which they are received and the callback corresponding to each request shall be invoked. If all requests point to the same NAF and do not force bootstrapping, then they may all get the same key. If bootstrapping occurs while processing the request, then the new key shall be returned.

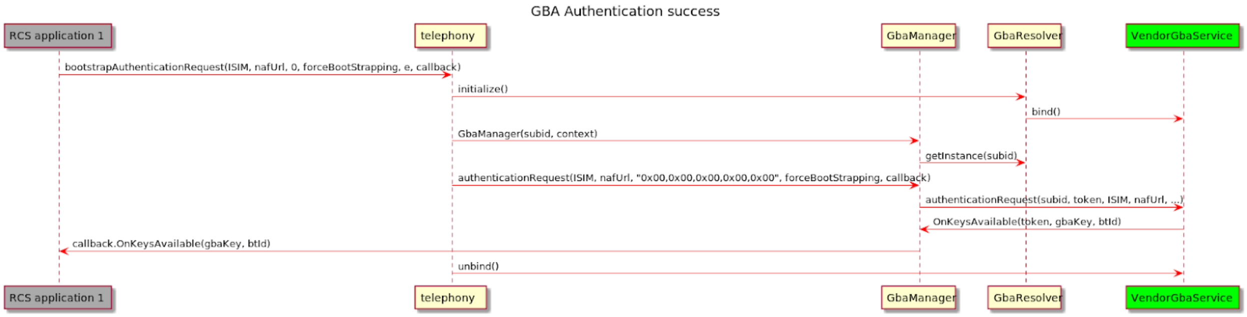
GbaService interface

The vendor must implement the GbaService, which shall maintain the central database to store bootstrapping keys and keys requested by the application and their validity. Gba keys are stored until their lifetime expires, or the keys are invalidated due to another bootstrapping session. When the GBA keys are requested, the service will check its internal cache to determine if the valid keys exist for the NAF. If they do, then it can return the keys immediately. If not then it can rerun bootstrapping and get new keys.
Gba Service shall be started and bound by telephony on demand when the application requests Gba authentication keys and unbound once the response to all ongoing requests is received. In case the Gba Service dies it must ensure that its internal database is updated with all the valid keys and their parameters like NAF url, BTID, valid until duration etc.

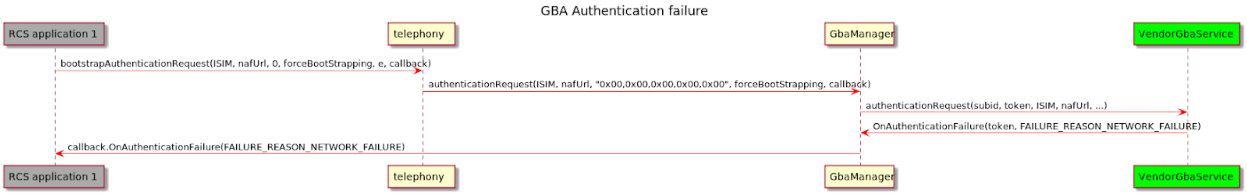
Sequence diagrams

Figures below depict the execution flow for various use cases:

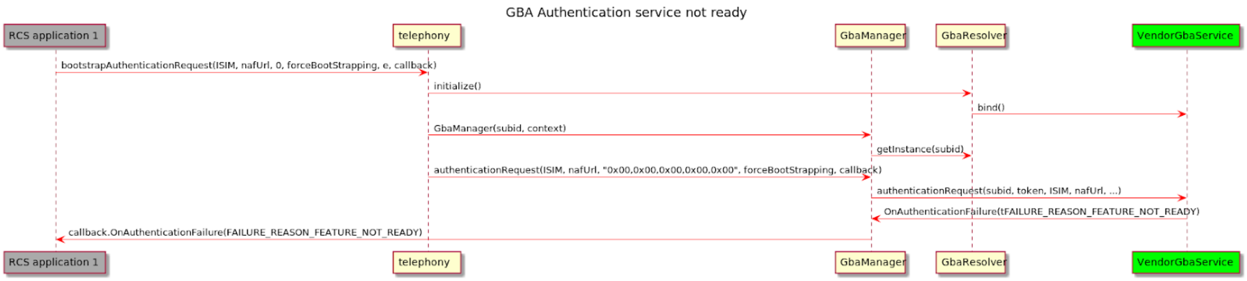
Scenario GBA.1: Service initialization and GBA authentication success


GBA Authentication success
RCS application 1
telephony
GbaManager
GbaResolver
VendorGbaService
bootstrapAuthenticationRequest(ISIM, nafUrl, 0, forceBootStrapping, e, callback)
initialize()
bind()
GbaManager(subid, context)
getInstance(subid)
authenticationRequest(ISIM, nafUrl, "0x00,0x00,0x00,0x00,0x00", forceBootStrapping, callback)
authenticationRequest(subid, token, ISIM, nafUrl, ...)
OnKeysAvailable(token, gbaKey, btId)
callback.OnKeysAvailable(gbaKey, btId)
unbind()
RCS application 1
telephony
GbaManager
GbaResolver
VendorGbaService

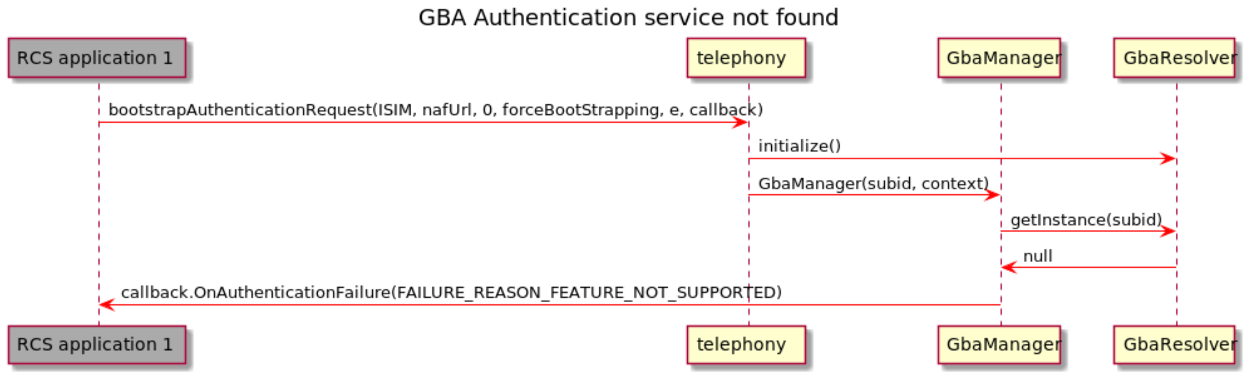
Scenario GBA.2: GBA authentication network failure


GBA Authentication failure
RCS application 1
telephony
GbaManager
VendorGbaService
bootstrapAuthenticationRequest(ISIM, nafUrl, 0, forceBootStrapping, e, callback)
authenticationRequest(ISIM, nafUrl, "0x00,0x00,0x00,0x00,0x00", forceBootStrapping, callback)
authenticationRequest(subid, token, ISIM, nafUrl, ...)
OnAuthenticationFailure(token, FAILURE_REASON_NETWORK_FAILURE)
callback.OnAuthenticationFailure(FAILURE_REASON_NETWORK_FAILURE)
RCS application 1
telephony
GbaManager
VendorGbaService

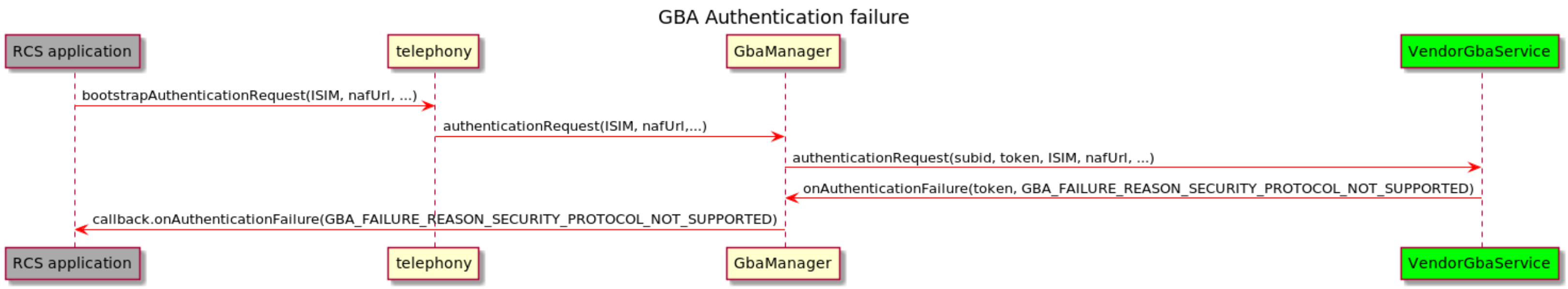
Scenario GBA.3: GBA authentication service not ready


GBA Authentication service not ready
RCS application 1
telephony
GbaManager
GbaResolver
VendorGbaService
bootstrapAuthenticationRequest(ISIM, nafUrl, 0, forceBootStrapping, e, callback)
initialize()
bind()
GbaManager(subid, context)
getInstance(subid)
authenticationRequest(ISIM, nafUrl, "0x00,0x00,0x00,0x00,0x00", forceBootStrapping, callback)
authenticationRequest(subid, token, ISIM, nafUrl, ...)
OnAuthenticationFailure(tFAILURE_REASON_FEATURE_NOT_READY)
callback.OnAuthenticationFailure(FAILURE_REASON_FEATURE_NOT_READY)
RCS application 1
telephony
GbaManager
GbaResolver
VendorGbaService

Scenario 4: GBA authentication service not supported on device


GBA Authentication service not found
RCS application 1
telephony
GbaManager
GbaResolver
bootstrapAuthenticationRequest(ISIM, nafUrl, 0, forceBootStrapping, e, callback)
initialize()
GbaManager(subid, context)
getInstance(subid)
null
callback.OnAuthenticationFailure(FAILURE_REASON_FEATURE_NOT_SUPPORTED)
RCS application 1
telephony
GbaManager
GbaResolver

Scenario 5: Security protocol not supported


GBA Authentication failure
RCS application
telephony
GbaManager
VendorGbaService
bootstrapAuthenticationRequest(ISIM, nafUrl, ...)
authenticationRequest(ISIM, nafUrl,...)
authenticationRequest(subid, token, ISIM, nafUrl, ...)
onAuthenticationFailure(token, GBA_FAILURE_REASON_SECURITY_PROTOCOL_NOT_SUPPORTED)
callback.onAuthenticationFailure(GBA_FAILURE_REASON_SECURITY_PROTOCOL_NOT_SUPPORTED)
RCS application
telephony
GbaManager
VendorGbaService

Multi-SIM Device Configurations

With a device in Dual-SIM, Dual-Standby (DSDS) configuration, telephony and the vendor IMS stack may support dual IMS stacks. When two SIMs are inserted on the device (using either pSIM or eSIM) then IMS may be registered simultaneously on both subscriptions to handle IMS traffic. All new AOSP single RCS registration APIs shall be capable of managing dual IMS registration for RCS and handle RCS traffic on both subscriptions simultaneously.

Specific impacts to each API category are discussed below.

Provisioning

- Telephony shall be able to store RCS client configuration per subscription.Telephony shall maintain the two provisioning xmls independently and will allow applications to register callbacks and retrieve the configurations for each.
- When provisioning updates are received for either subscription, telephony shall be able to update its internal storage and also notify applications which have registered a callback to receive updates on that subscription.

SIP Transport, Registration and UCE

- SIP transport APIs allow applications to create `SipDelegates` on either or both the subscriptions simultaneously
- If RCS is provisioned for both subscriptions and `SipDelegateConnections` are created by the application for both subscriptions, then IMS shall register on both subs for RCS.
- RCS applications may create an `ImsRcsManager` instance per subscription and trigger user capability exchange using the `ImsRcsManager` APIs.
- SIP transport APIs allow applications to send and receive data on either subscription irrespective of DDS configuration.

GBA Authentication

- Application shall be able to use GBA authentication API on both subscriptions.

Dedicated Bearer

- The dedicated bearer listener can be registered for either subscription irrespective of the

Default Data Subscription configuration and the subscription being the primary or secondary call subscription.

- Dedicated bearer indications can be sent to registered applications, irrespective of the Default Data Subscription configuration and the subscription being either the primary or secondary subscription.

RCS Application considerations

- Irrespective of the application support for DSDS, applications must be aware that an ongoing transaction may get interrupted due to voice call activity on the other subscription. In that case application retransmissions should kick in.
- RCS applications supporting dual RCS registration must be able to maintain the provisioning status and data for individual subscriptions.
- RCS applications supporting dual RCS registration must be capable of handling incoming messages on either subscription.
- Subscription selection for outgoing traffic shall be the RCS Application's responsibility.

In the event that an RCS application does not support simultaneous IMS Single Registration even though both SIMs are RCS provisioned, then it shall be the application's responsibility to choose the subscription for RCS registration.

Specifications and References

Reference	Title
RFC3261	"SIP: Session Initiation Protocol", June 2002, https://tools.ietf.org/html/rfc3261
RCC.59	RCC.59 - North America RCS Common Implementation Guidelines, Version 1.0, https://www.gsma.com/futurenetworks/wp-content/uploads/2015/05/RCC-59-v1-0.pdf
RFC 3840	"Indicating User Agent Capabilities in the Session Initiation Protocol (SIP)", August 2004, https://tools.ietf.org/html/rfc3840
RCC.07	"Rich Communication Suite - Advanced Communications Services and Client Specification", Version 11, 16 October 2019, https://www.gsma.com/futurenetworks/wp-content/uploads/2019/10/RCC.07-v11.0.pdf
RCC.14	"Service Provider Device Configuration", Version 7.0 16 October 2019, https://www.gsma.com/newsroom/wp-content/uploads//RCC.14-v7.0-1.pdf
TS33.220	"Generic Authentication Architecture (GAA); Generic Bootstrapping Architecture (GBA)", 3GPP TS 33.220, Version 15.4.0, Release 15, April 2019, https://www.etsi.org/deliver/etsi_ts/133200_133299/133220/15.04.00_60/ts_133220v150400p.pdf
RCC.71	"RCC.71 - RCS Universal Profile Service Definition Document", Version 2.4, 16 October 2019, https://www.gsma.com/futurenetworks/wp-content/uploads/2019/10/RCC.71-v2.4.pdf
OMA DDS	"Presence SIMPLE Data Specification", Version 2.3, 22 Dec 2015,

Presence Extensions	http://www.openmobilealliance.org/release/PDE/V1_4-20151222-A/OMA-DDS-Presence_Data_Ext-V2_3-20151222-A.pdf
RFC3863	“Presence Information Data Format (PIDF)”, August 2004, https://tools.ietf.org/html/rfc3863