Avaya Application Enablement Services

Device, Media and Call Control API
Java Programmers Guide
R3.1.1
An Avaya MultiVantage® Communications Application

02-300359
Issue 2.2
May 2006

Notice

While reasonable efforts were made to ensure that the information in this document was complete and accurate at the time of printing, Avaya Inc. can assume no liability for any errors. Changes and corrections to the information in this document may be incorporated in future releases.

For full support information, please see the complete document, *Avaya Support Notices for Software Documentation*, document number 03-600758.

To locate this document on our Web site, simply go to http://www.avaya.com/support and search for the document number in the search box.

Documentation disclaimer

Avaya Inc. is not responsible for any modifications, additions, or deletions to the original published version of this documentation unless such modifications, additions, or deletions were performed by Avaya. Customer and/or End User agree to indemnify and hold harmless Avaya, Avaya's agents, servants and employees against all claims, lawsuits, demands and judgments arising out of, or in connection with, subsequent modifications, additions or deletions to this documentation to the extent made by the Customer or End User.

Link disclaimer

Avaya Inc. is not responsible for the contents or reliability of any linked Web sites referenced elsewhere within this documentation, and Avaya does not necessarily endorse the products, services, or information described or offered within them. We cannot guarantee that these links will work all of the time and we have no control over the availability of the linked pages.

Warranty

Avaya Inc. provides a limited warranty on this product. Refer to your sales agreement to establish the terms of the limited warranty. In addition, Avaya's standard warranty language, as well as information regarding support for this product, while under warranty, is available through the following Web site: http://www.avaya.com/support.

Copyright

Except where expressly stated otherwise, the Product is protected by copyright and other laws respecting proprietary rights. Unauthorized reproduction, transfer, and or use can be a criminal, as well as a civil, offense under the applicable law.

Avaya support

Avaya provides a telephone number for you to use to report problems or to ask questions about your product. The support telephone number is 1-800-242-2121 in the United States. For additional support telephone numbers, see the Avaya Web site: http://www.avaya.com/support.

Contents

About this document

This chapter describes the:

- Scope of this document
- Intended Audience
- Conventions used in this document
- Related documents
- Providing documentation feedback

Scope of this document

This document shows you how to use the Application Enablement (AE) Services Device, Media and Call Control API to develop, debug, and deploy applications that require third party device, media and call control. This document does not explain many of the concepts of the API. Please see the *Avaya MultiVantage™ Application Enablement Services Overview* document for this information.

- Chapter 1: API Services provides background information about the AE Services Device, Media and Call Control API and CSTA.
- Chapter 2: Getting started gets you ready to program to this API.
- Chapter 3: Writing a client application and Chapter 4: Compiling and Debugging guide you in developing and debugging applications.
- Appendix A: Communication Manager Features lists the Communication Manager features that your application can take advantage of.
- Appendix B: Migrating Communication Manager API 2.1 Applications to Application Enablement Services lists the changes required to migrate Communication Manager API 2.1 applications to AE Services Device, Media and Call Control API 3.0 and 3.1 applications.
- Appendix C: TSAPI Error Code Definitions appendix lists all of the values for the TSAPI error codes.
- The Glossary defines the terminology and acronyms used in this document.

Intended Audience

This document is written for applications developers. A developer must know:

- Java™
- basic Linux commands
- telephony concepts

You do not need to understand CSTA concepts or Avaya Communication Manager features and concepts, but they both might be helpful.

If you are new to CSTA, you may wish to start by reading *ECMA-269*, section 6.1, “CSTA Operational Model: Switching Sub-Domain Model”. Also become familiar with the table of contents so that you know the kinds of information available there. All of the descriptions of the services implemented by this API are also found in the Javadoc, *Avaya MultiVantage Application Enablement Services Device, Media and Call Control API Java Programmers Reference*, found online on the Avaya Developer Connection website (http://www.devconnectprogram.com) and on the Avaya Support Centre website (http://www.avaya.com/support) under “Communication Systems”.

For those new to Avaya Communication Manager, you may wish to take a course from Avaya University (http://www.avaya.com/learning) to learn more about Communication Manager and its features. It is recommended that you start with the *Avaya Communication Manager Overview* course (course ID AVA00383WEN). You may also wish to peruse Appendix A: Communication Manager Features in this guide to get some ideas of how applications can take advantage of Communication Manager’s abilities.

Conventions used in this document

The following fonts are used in this document:

To represent...	**This font is used...**
Code and Linux commands	`request = new GetDeviceId();`
Java class, method and field names	the `getDeviceID` method
Window names	The buttons are assigned on the **Station** form.

To represent...	This font is used...
Browser selections	Select **Member Login**
Hypertext links	Go to the http://www.avaya.com/support website. The term connector can be found in the glossary. For details, see Getting device identifiers on page 53.

Related documents

While planning, developing, deploying, or troubleshooting your application, you may need to reference other Avaya MultiVantage Application Enablement Services documents, Avaya Communication Manager documents, or CSTA documents listed below.

Application Enablement Services documents

For developers, the other important source of Java API information is the Javadoc:

- *Avaya MultiVantage® Application Enablement Services Device, Media and Call Control Java Programmers Reference*

Here you can find details about each package, interface, class, method, and field in the API. You can also find out what parts of the CSTA protocol have been implemented.

Other Application Enablement Services documents include:

- Avaya MultiVantage® Application Enablement Services Overview (02-300360)
- *Avaya MultiVantage® Application Enablement Services Installation Guide Software Only Offer* (02-300355)
- *Avaya MultiVantage® Application Enablement Services Installation and Upgrade Guide for a Bundled Server* (02-300356)
- *Avaya MultiVantage® Application Enablement Services Administration and Maintenance Guide* (02-300357)
- Avaya Application Enablement Services *Device, Media and Call Control API XML Programmer Guide* (02-300358)
- *Avaya MultiVantage® Application Enablement Services Device, Media and Call Control XML Programmer Reference* (XMLdoc)
- Avaya MultiVantage® Application Enablement Services *Device, Media and Call Control API Web Services Programmer Guide (*(02-300362)

- *Avaya MultiVantage® Application Enablement Services Documentation Roadmap* (02-300361)
- *Avaya MultiVantage® Application Enablement Services OAM Help* (HTML)
- *Avaya MultiVantage® Application Enablement Services 3.1 TSAPI, JTAPI, and CVLAN Client Installation Guide* (02-300543)
- *Avaya MultiVantage® Application Enablement Services 3.1 TSAPI for Avaya Communication Manager Programmer'Reference* (02-300544)
- *Avaya MultiVantage® Application Enablement Services 3.1 TSAPI Programmer Reference* (02-300545)
- *Avaya MultiVantage® Application Enablement Services 3.1 CVLAN Programmer Reference* (02-300546)
- *Avaya MultiVantage® Application Enablement Services 3.1 Java Telephony API (JTAPI) Programmer Reference (JTAPI v1.2 Specification)* (02-300547)
- *Avaya MultiVantage® Application Enablement Services 3.1 Java Telephony API (JTAPI) for Avaya MultiVantage Programmer's Guide* (02-300548)
- *Avaya MultiVantage® Application Enablement Services 3.1 ASAI Technical Reference* (02-300549)
- *Avaya MultiVantage® Application Enablement Services 3.1 ASAI Protocol Reference* (02-300550)

You can find all these documents online on the Avaya Support Center Web Site (http://support.avaya.com).

Communication Manager documents

Since this API gives you programmable access to Avaya Communication Manager features, you may wish to reference documents about that system.

The following documents from the Communication Manager documentation set provide additional information about administering Communication Manager for Device, Media and Call Control API. They are on the Avaya Support Centre Web Site (http://www.avaya.com/support).

- *Administrator's Guide for Avaya Communication Manager* (Issue 11 for CM 3.1), *(*03-300509)
- *Administration for Network Connectivity for Avaya Communication Manager* (Issue 11 for CM 3.1), (555-233-504_10)

ECMA documents

The *Java Programmers Reference* (Javadoc) contains much of what you need to know about CSTA services. For CSTA details not found in the *Javadoc* or this document, please refer to the following documents. They are found in the Publications section of the ECMA web site (http://www.ecma-international.org/):

- *ECMA-269: Services for Computer Supported Telecommunications Applications (CSTA) Phase III*
- *ECMA-323: XML Protocol for Computer Supported Telecommunications Applications (CSTA) Phase III*
- *ECMA-354: Application Session Services*
- *ECMA Technical Report TR/72: Glossary of Definitions and Terminology for Computer Supported Telecommunications Applications (CSTA) Phase III*

Providing documentation feedback

Let us know what you like or do not like about this document. Although we cannot respond personally to all your feedback, we promise we read each response we receive.

Please email feedback to **`document@avaya.com`**

Thank you.

Chapter 1: API Services

This chapter provides an overview of what CSTA services the API supports and what extensions Avaya has implemented. This API supports the following telephony services:

- device control
- media control
- call control
- call recording, message playing and dubbing
- DTMF tone detection

These services are provided through a Java API interface. Some of the interfaces conform to the CSTA III standard (ECMA-269) and some are Avaya extensions to the CSTA standard.

CSTA specifies that for any given service some parameters are mandatory and some parameters are optional. To determine which of the optional parameters Avaya supports or which of the field values Avaya supports, refer to the requests and responses detailed in the programmer's reference (Javadoc).

Note:

The ECMA standards body requests that CSTA-compliant implementations reflect conformance to the standard through a *Protocol Implementation Conformance Statement* (PICS). The Application Enablement Services Device, Media and Call Control API PICS is reflected in the programmer's reference.

This chapter lists:

Supported CSTA services

In CSTA, each service is defined to be a request that either comes from the application to Communication Manager or from Communication Manager to the application. This API, however, is based on a client/server model where the application is the client and the AE Services server software and Communication Manager together act as the server. Thus, this API allows an application:

- to request services of Communication Manager
- to request notification of asynchronous events on Communication Manager

The following sets of CSTA services are supported in the Application Enablement Services Device, Media and Call Control API and described in the sections that follow:

Table 1: Supported CSTA services

Sets of supported CSTA services	CSTA specifications	CSTA XML protocol
Physical Device Services and Events	ECMA-269, section 21	ECMA-323, section 19
Voice Unit Services and Events	ECMA-269, section 26	ECMA-323, section 24
Call Control Services	ECMA-269, section 17	ECMA-323, section 15
Snapshot Services	ECMA-269, section 16	ECMA-323, section 14
Monitoring Services[1]	ECMA-269, section 15	ECMA-323, section 13

1. This API implements the intent of CSTA's Monitoring Services using Java Listeners

Physical Device Services and Events

CSTA's Physical Device Services provide physical device control. The control allows an application to manipulate and monitor the physical aspects of a device, which includes buttons, lamps, the display, and the ringer. The services simulate manual action on a device as well as provide the ability to request status of physical elements. The events provide notification of changes to the physical elements of the device. To learn how to use Physical Device Services and Events, see Monitoring and controlling physical elements on page 69.

This API supports the following Physical Device Services:

Table 2: Physical Device Services

Service	Description
Button Press	Simulates the depression of a specified button on a device
Get Button Information	Gets the button information for either a specified button or all buttons on a device, including the button identifier, button function, associated extension (if applicable), and associated lamp identifier (if applicable)
Get Display	Gets a snapshot of the contents of the physical device's display

1 of 2

Table 2: Physical Device Services (continued)

Service	Description
Get Hookswitch Status	Gets the hookswitch status of a specified device, either on hook or off hook
Get Lamp Mode	Gets the lamp mode status for either a specified button or all buttons on a device, including how the lamp is lit (flutter, off, steady, etc.), color and associated button
Get Message Waiting Indicator	Gets the message waiting status at a specified device, either on or off
Get Ringer Status	Gets the ringer status of the ringer associated with a device, including ring mode (ringing/not ringing) and the ring pattern such as normal ring or priority ring
Set Hookswitch Status	Sets the hookswitch status of a specified device to either onhook or offhook
	2 of 2

This API supports the following CSTA Physical Device events:

Table 3: Physical Device Events

Event	Description
Display Updated	Occurs if the contents of a device's display has changed
Hookswitch Status Changed	Occurs if Communication Manager has changed the device's hookswitch status
Lamp Mode Changed	Occurs if the lamp mode status of a particular lamp has changed
Ringer Status Changed	Occurs if the ringer attribute associated with a device has changed status

Voice Unit Services and Events

CSTA's Voice Unit Services allow an application to record voice stream data coming into a device and to play messages to the device's outgoing voice stream.

Voice Unit Services has been extended to also provide a dubbing service and more specific stop, suspend, and resume services. See Extended Voice Unit Services on page 23.

To learn how to use Voice Unit Services, see Recording and playing voice media on page 73.

This API supports the following Voice Unit Services:

Table 4: Voice Unit Services

Services	Description
Play Message	Plays a pre-recorded voice message on the outgoing RTP media stream of a particular device based on a specified criteria
Record Message	Starts recording the media stream for a specified device with the specified codec and criteria
Resume	Restarts the playing and recording of previously suspended messages at their current positions
Stop	Stops the playing and recording of messages
Suspend	Temporarily stops the playing and recording of messages and leaves their position pointers at their current locations
Delete Message	Deletes a specified file from the connector server

This API supports the following CSTA Voice Unit events:

Table 5: Voice Unit Events

Events	Description
Play	Indicates that a message is being played
Record	Indicates that a message is being recorded
Stop	Indicates that a play or record operation for a message on a device has been stopped or has completed
Suspend Play	Indicates that a message is suspended in play
Suspend Record	Indicates that a message is suspended during recording

Call Control Services

CSTA's Call Control Services provide single step conferencing capabilities to allow an application to add a device into an existing call.

The Call Control Services utilizes the TSAPI Service on the AE Services server. The use of the Call Control Services requires the setup of the connection and cti-link between the AE Services server and Communication Manager as well as a basic TSAPI license.

This API supports the following Call Control Service:

Table 6: Call Control Services

Services	Description
Single Step Conference Call	Adds a device to an existing call.

Note:

Call control events are not supported in this release.

Snapshot Services

CSTA's Snapshot Call allows an application to obtain3rd party information about the devices participating in a specified call. The information returned includes device identifiers, their connections in the call, and local connection states of the devices in the call as well as call related information.

The Snapshot Device service provides information about calls associated with a given device. The information provided identifies each call the device is participating in and the local connection state of the device in that call.

The use of the Snapshot Services requires the setup of the connection and cti-link between the AE Services server and Communication Manager as well as a basic TSAPI license

This API supports the following Snapshot Services:

Table 7: Snapshot Services

Services	Description
Snapshot Call	Provides information about the devices participating in a specified call.
Snapshot Device	Provides information on the status of calls at a specific device.

Monitoring Services

Monitoring services allows applications to request asynchronous notification of specific events. Avaya's implementation provides the same functionality as CSTA's Monitoring Services, but does so via Java listeners instead of CSTA service requests.

This implementation provides methods for adding and removing listeners for each set of services. If multiple listeners are added, all listeners will be notified of events.

To learn how to use these listeners, see Requesting notification of events on page 55.

This API supports these Monitoring Services:

Table 8: Monitoring Services

Services	Description
Add Call Information Listener	Adds a listener to monitor for changes in the status of the call information link.
Remove Call Information Listener	Removes the indicated listener so that it no longer receives notifications of call information link status changes.
Add Physical Device Listener	Adds a listener to monitor the specified physical device for events. Events report changes to the components of a physical element of the device (display, lamps and ringer).
Remove Physical Device Listener	Removes a previously specified physical device listener
Add Terminal Listener[1]	Adds a TerminalListener for receiving registration events for a specific device.
Remove Terminal Listener[2]	Removes the specified TerminalListener from receiving device registration events
Add Media Control Listener	Adds a MediaControlListener for receiving media control events for a specific device. Used by applications that wish to be notified of media events - usually these are applications that are controlling the media stream.
Remove Media Control Listener	Removes the specified MediaControlListener from receiving media control events for the specified device
Add Tone Collection Listener	Adds a listener to monitor the specified device for Tone collection events

1 of 2

Table 8: Monitoring Services (continued)

Services	Description
Remove Tone Collection Listener	Removes a previously specified tone collection listener
Add Tone Detection Listener	Adds a listener to monitor the specified device for tone detection events.
Remove Tone Detection Listener	Removes a previously specified tone detection listener
Add Voice Unit Listener	Adds a listener to monitor the specified physical device for voice unit events. Voice unit events report changes to the voice stream data messages being recorded and played. In other words, the voice unit events reflect the state changes of a Voice Unit.
Remove Voice Unit Listener	Removes a previously specified voice unit listener
Add Registration Listener	Adds a listener to monitor for device registration.
Remove Registration Listener	Removes a previously specified registration listener.
	2 of 2

1. The add and remove listeners for Terminal Services were supported in the previous releases, but were deprecated in release 3.0 in favor of methods in the Avaya Registration Services. Services in deprecated status may still be used, but may be removed in future releases. Applications should move to the new services
2. See footnote 1

Avaya extensions

The API provides extensions to CSTA that are meant to enhance the capabilities of CSTA and provide higher-level services and useful events that make development of telephony applications easier. The extensions are summarized in this section. More complete descriptions of each extension can be found in the *Programmer's Reference* (Javadoc).

The Avaya extensions have been implemented per the CSTA guidelines described in *ECMA-269*, section 28, "Vendor Specific Extensions Services and Events".

The Avaya extensions are listed below and described in the following sections.

Table 9: Avaya extensions to CSTA services

Avaya extension	Extends which CSTA service set	Purpose
Asynchronous Services	None	Provides the ability to receive asynchronous responses to requests rather than blocking for them.
Call Information Services and Events	None	Provides the ability to obtain detailed call information and to determine the status of the call information link.
Device Services	None	Provides an identifier for a given extension on Communication Manager
Extended Voice Unit Services	Voice Unit Services	Provides dubbing of recorded messages and other extensions to playing and recording of files
Media Control Events	None	Provides the ability to be notified when the far-end RTP and RTCP parameters for a media stream change
Registration Services	None	Provides ability to gain exclusive or shared control over Communication Manager endpoints - also referred to as *device registration*
Service Provider	All	Provides a starting point to access all other CSTA and Avaya services
Tone Collection Services and Events	None	Detects DTMF tones and buffers them as requested before reporting them to the application
Tone Detection Events	Replaces Data Collection Services	Detects DTMF tones and reports each tone as it is detected
Terminal Services and Events[1]	None	Provides ability to gain exclusive or shared control over Communication Manager endpoints - also referred to as *device registration*

1. Terminal Services and Events were deprecated in release 3.1 in favor of methods in the Avaya Registration Services. Services in deprecated status are supported and may still be used, but may be removed in future releases. Applications using Terminal Services and Events are strongly encouraged to move to the new services.

Asynchronous Services

Avaya's Asynchronous Services provides the ability to receive asynchronous responses to requests rather than blocking for them.

To learn how to use the Asynchronous Services see Asynchronous Services on page 34.

Table 10: Asynchronous Services

Services	Descriptions
Send Request	Allows an application to send a request along with a callback object that will be notified asynchronously when the response is received

Note:

For this release, only the `RegisterTerminal` and `UnregisterTerminal` requests of the Registration Services, the `SingleStepConferenceCall` request of the Call Control Services and the `SnapshotCall` and `SnapshotDevice` requests of the Snapshot Services can be invoked using the Asynchronous Services `sendRequest` method. If any other request is passed in, a `ServiceNotSupportedException` will be thrown.

Call Information Services and Events

Avaya's Call Information Services allows applications to get detailed call information and to determine the status of the call information link. The call information link must be operational to get the call information. The call information link is one of the communication links between Communication Manager and the connector server.

This API supports the following Call Information Services:

Table 11: Call Information Services

Services	Description
Get Call Information	Used to get detailed call information.
Get Link Status	Used to get the status of the call information link to the connector server.

This API supports the following Call Information Events:

Table 12: Call Information Events

Events	Description
Link up	Occurs when a link has come up and is now active. Occurs the first time the link is brought up, as well as every time the link is brought up after being down.
Link Down	Occurs when a link has gone down and is now inactive. Occurs when it is determined that Communication Manager is not responding or Communication Manager and Device, Media and Call Control API are out of sync. Response will indicate which link is down and whether the connector server will attempt to reconnect automatically.

Device Services

All services that operate on a particular device use a device identifier to specify the device. Avaya's Device Services provide a device identifier for a given extension on Communication Manager. There are no events generated by this service.

To learn how to use the Device Services see Getting device identifiers on page 53.

This API supports the following Device Services:

Table 13: Device Services

Services	Description
Get Device ID	Gets the device identifier that represents the device described by its extension number and the Communication Manager upon which it resides
GetThird Party Device ID	Gets a third party device identifier for use with Call Control Services and Snapshot Services

Extended Voice Unit Services

Avaya's Extended Voice Unit Services are used in conjunction with CSTA's Voice Unit Services.

To learn how to use the Extended Voice Unit Services, see Recording and playing voice media on page 73

These Extended Voice Unit services are provided:

Table 14: Extended Voice Unit Services

Services	Descriptions
Start Dubbing	Starts replacing an existing recording session with the specified file
Stop Dubbing	Stops replacement of an existing recording session
Stop Playing	Stops only the player, not the recorder.
Stop Recording	Stops only the recorder, not the player.
Suspend Playing	Suspends only the player, not the recorder.
Suspend Recording	Suspends only the recorder, not the player.
Resume Playing	Resume only playing, not recording.
Resume Recording	Resumes only recording, not playing.

Media Control Events

Avaya's Media Control events provide a way for an application that is using client media under exclusive control mode to respond to changes in the far-end RTP/RTCP parameters of a media stream.

To learn how to use the media control events, see Determining when far-end RTP media parameters change on page 81.

This API supports the following Media Control events:

Table 15: Media Control Events

Events	Descriptions
Media Start	Indicates when the far-end RTP parameters have changed and an RTP session has been established. Also provides the media encryption keys if media encryption is enabled for the device.
Media Stop	Indicates when the far-end RTP parameters have changed to null and the RTP session has been disconnected

Registration Services

Avaya'sRegistration Services, introduced in release 3.1, provide the ability to gain exclusive or shared control over a device and to specify the desired media mode for that device.

Exclusive control and shared control are described in the "Device and media control" section of the "Capabilities of the API" chapter of the *Avaya MultiVantage™ Application Enablement Services Overview* . For a list of device types that can be controlled with this API and for further distinction between shared and exclusive control, see the "Controllable telephone types" section of the "Capabilities of the API" chapter of the *Avaya MultiVantage™ Application Enablement Services Overview* .

The desired media parameters are also specified at registration time. The options for the Media parameters are described in the "Media control modes" section of the "Capabilities of the API" chapter of the *Avaya MultiVantage™ Application Enablement Services Overview .*

Registration Services is synchronous. The responses to the requests are true responses after the request has succeeded or failed. Your application will block while waiting for the reponse, which may take some time. To avoid blocking, many applications will want to use the Asynchronous Service to send Registration and Unregistration requests.

To learn how to use Registration Services, see Registering devices on page 57.

The Registration Services are:

Table 16: Registration Services

Services	Descriptions
Register Terminal	Registers a specific device with Communication Manager in order to gain exclusive or shared control over the phone or extension and specifies the desired media mode. Applications can specify whether media encryption is desired while registering.
Unregister Terminal	Unregisters the specified device from Communication Manager in order to give up control of the device

This API supports the following event:

Table 17: Registration Events

Events	Descriptons
Terminal Unregistered	Occurs when the device is unregistered by Communication Manager. This event will not be sent if the application requests unregistration.

Note:

If a device is registered in client media mode, then the Media Control events described in the section Media Control Events on page 23 may also occur.

Note:

Previously with Terminal Services, the actual response from the `RegisterDevice` and `UnregisterDevice` requests came as an event. With Registration Services the response your application receives is a true indication of success or failure.

Service Provider

Avaya's Service Provider is the starting point for obtaining all other services in this API.

To learn how to use the Service Provider, see Getting access to desired services on page 47.

Table 18: Service Provider Requests

Services	Descriptions
Get Service Provider	Returns an instance of a Service Provider object.
Get Service	Returns an instance of the specified service
Reconnect	Reestablishes the previous session
Disconnect	Disconnects from the server and optionally terminates the active session
Get Session ID	Retrieves the sessionID for this session
Get Negotiated API Version	Retrieves the API version for the server
Add Service Provider Listener	Adds a listener to monitor the specified device for service provider events.
Remove Service Provider Listener	Removes a previously specified service provider listener

These events may be generated by the service provider:

Table 19: Service Provider Events

Services	Descriptions
Server Connection Down	Occurs if the socket to the AE Services server has gone down for some reason
Server Session Not Active	Occurs if a message was received by the AE Services server but the session has timed out and been placed in the inactive state
Server Session Terminated	Occurs if a message was received by the AE Services server but the session cleanup timer expired after the session entered the inactive state

Terminal Services and Events

Note:

Terminal Services and Events were supported in previous releases, but are deprecated in release 3.1 in favor of the Avaya Registration Services. Services in deprecated status may still be used, but may be removed in future releases. Applications using Terminal Services and Events are strongly encouraged to move to the new services.

Tone Collection Services and Events

Avaya's Tone Collection Services collects DTMF tones coming into a device, stores them in a buffer, and reports the tones based on application-specified retrieval criteria. The retrieval criteria can be one or more of the following:

- The specified number of tones has been detected
- The specified tone has been detected
- The specified amount of time has passed

If multiple criteria are specified, then the first condition that occurs terminates the retrieval and reports the string of DTMF tones collected. Both in-band and out-of-band tone collection are supported. Out-of-band tone collection is recommended.

When tones are retrieved and reported to the application, they are removed from the buffer. If the buffer fills up, the oldest tones are overwritten with the new detected tones.

To learn how to use Tone Collection Services, see Detecting and collecting DTMF tones on page 78.

This API supports the following Tone Collection Services:

Table 20: Tone Collection Services

Services	Description
Start Tone Collection	Used to start collecting DTMF tones sent to a device and specifies the termination criteria.
Tone Collection Criteria	Used to specify the retrieval criteria.
Stop Tone Collection	Used to stop collecting DTMF tones sent to a device and report the tones that have been buffered. This flushes the buffer.
Flush Buffer	Reports the DTMF tones received since the last time the buffer was flushed and flushes the buffer.

Tone Collection Services generates these events:

Table 21: Tone Collection Events

Events	Description
Tones Retrieved	Occurs when DTMF tones are retrieved from the buffer. This event reports the retrieved tones to the application.

Tone Detection Events

Avaya's Tone Detection Events notifies an application whenever a DTMF tone has been detected coming into a device. Both in-band and out-of-band tone detection is supported. Out-of-band tone detection is recommended.

To learn how to use Tone Detection Events, see Detecting and collecting DTMF tones on page 78.

When the application requests monitoring for DTMF tones, the following event will be generated when a DTMF has been sent to the device:

Table 22: Tone Detection Events

Events	Description
Tone Detected	Occurs when a DTMF tone has been sent to the device

Differences between Avaya API and ECMA-269

The Avaya API differs from the ECMA specification in the following ways:

- Voice Unit Services perspective
- Negative acknowledgements
- Monitoring

Voice Unit Services perspective

The mechanism for call control in this API is to register an extension with Communication Manager using Registration Services and then to use Physical Device Services to manipulate that extension. Therefore this API follows a device-based call control model. There are a few subtle side effects of using the device-based control model that are worth noting.

- CSTA specifies that the Voice Unit Play Message service “plays a voice message on a particular connection”. While this is an ambiguous description, the apparent intent was to play a message *to* a particular device, which is a third party perspective. This API’s implementation of the Play Message service is just the opposite of this. This API’s Play Message service plays a message *from* the device, a first party perspective. It plays the message as if coming from the device and going to everyone else on the call.
- Similarly, CSTA specifies that the Voice Unit Record Message service “starts recording a new message from a specified connection.” The apparent intent was to record the data coming *from* the device. This API implementation records the data coming *to* the device. It records what the device hears instead of what someone says at the device.
- Since Avaya’s implementation of Voice Unit Services are relative to a device instead of a connection, only the device identifier portion of a connection identifier is used.

Negative acknowledgements

The CSTA standard uses negative acknowledgements to indicate that an error has occurred. That is, a request can have different responses, either a valid response, or an error code. This API maps each of the error codes specified by CSTA to a specific exception. Each service request returns a valid response or throws and exception if an error occurs. Each exception also contains the specific CSTA error code.

Each exception may also contain a number of chained exceptions. The full stack trace of the chained exceptions may be helpful to application developers in understanding the error and in debugging their own code. This stack trace may also be helpful in debugging problems in the server.

Monitoring

The CSTA standard has a mechanism called Monitoring Services to allow clients to register interest in specific events. This mechanism can be mapped to a language such as Java using a single method for receiving all events, but the burden on the application developer to determine the specific event type would be too high. Therefore, this API implements standard Java listeners instead.

This API has a listener interface for each class of events. For example, there is a Physical Device listener interface for Physical Device events. The application programmer implements the listener interface and adds a listener to a specific device. Behind the scenes, the API uses Monitoring Services to register interest for the specific events. When an event of interest occurs on the server, Monitoring Services notifies the specific listener, and the corresponding method on the application's listener is invoked.

Chapter 2: Getting started

This section describes what you need to do and what you need to know before you begin programming to this API, including:

Setting up the development environment

Applications can be developed in any environment supporting Sun Microsystems™ Java 2 Platform, Standard Edition (J2SE™) 1.5 or higher.

Downloading the Java SDK

Your development machine needs to have the latest J2SE Java Development Kit (JDK) installed (currently 1.5.0_02). If it is not already installed, download it from Sun Microsystem's website:

1. Download the latest 1.5 J2SE JDK (currently 1.5.0_02) from http://java.sun.com.
2. Install the Java J2SE JDK according to the instructions provided by Sun Microsystems.
3. Run a sample Java program to verify the proper installation of the Java platform.

Note:

JDK 1.4.2 or earlier versions are not supported.

Downloading the Application Enablement Services Device and Media Control Java API SDK

To download the Application Enablement Services Device, Media and Call Control Java API SDK from the Avaya Developer's Connection website:

1. Go to www.devconnectprogram.com.
2. Select **Member Login**.
3. Log in with your email address and password.

   The **Avaya Communication Manager Developers** page appears.
4. Download the SDK (cmapijava-sdk-3_1_*x*.zip).

   The download location defaults to the desktop, but it does not matter where you download the files in your directory system. The SDK file is:

   cmapijava-sdk-3_1_*x*.zip

   where *x* is the load number.
5. Expand the SDK ZIP file using any application or tool that recognizes the ZIP file format. All of the SDK files are placed into a directory named cmapisdk.

Setting up your test environment

Before running an application you will need to have an AE Services server machine setup. For instructions see the appropriate *Avaya MultiVantage™ Application Enablement Services Installation Guide* for the offer you have purchased (bundled server or software only).

Understanding basic CSTA concepts

CSTA stands for Computer-Supported Telecommunications Applications. It is a standard produced by ECMA, an international standards body (http://www.ecma-international.org). CSTA provides a standard for Computer-Telephony Integration (CTI). When fully implemented, CSTA allows an application to:

- monitor calls on extension lines or trunks
- modify the behavior of calls
- make a call between two parties

The Avaya Application Enablement Services Device, Media and Call Control API implements a subset of CSTA. The API supports monitoring and making calls at the physical device level. Applications using this API have first party device control and media control.

The following sections describe what you need to know about the CSTA concepts of:

- Devices
- Physical elements and Logical elements
- Calls
- Service requests and Service responses
- Events
- Negative acknowledgements

Devices

In the context of this API, a device refers to a software instantiation of a phone or extension that is registered on Communication Manager. Such a device is also referred to as a softphone. A device has physical and logical elements.

Physical elements

The physical element of a device encompasses the set of attributes, features, and services that have any association with physical components of the device that make up the user interface of the device. Physical elements can be manipulated, such as pushing buttons or going offhook, or they can be observed, such as observing the ringer or whether a lamp is lit. The physical elements of a device include:

- Buttons
- Hookswitch
- Display
- Lamps
- Message waiting indicator
- Ringer

This API supports all of these physical elements.

Logical elements

A logical element is the part of a device that is used to manage and interact with calls at a device. This element represents the media stream channels and associated call handling

facilities that are used by the device when involved in a call. The logical elements that this API supports are:

- DTMF tones coming into the device
- Media stream coming into and out of the device

Calls

Calls are made from and received by a device using the device's physical and logical elements. CSTA performs most telephony services against a connection that identifies a particular call. In this API telephony services are requested for a device instead of a connection. For Voice Unit Services and Extended Voice Unit Services, on the device portion of the ID is used.

Request and response framework

Your application will need to be able to:

- make service requests
- process service responses
- catch exceptions
- listen for and respond to events

This section describes what requests, responses, exceptions and events are and how to use them.

Asynchronous Services

Asynchronous Services was added in the 3.1 release as a means to allow applications to receive responses to requests asynchronously. With Asynchronous Services, the call to send a request returns immediately after the request is sent. The response is then given to the application through a callback object when the response is received.

Asynchronous Services does not have any unique requests: it simply provides a means to asynchronously receive responses to a subset of the other Services' requests. The only requests currently supported through Asynchronous Services are those for Registration Services, Call Control Services and Snapshot Services. The same request object is given to the Asynchronous Services `sendRequest` method as would be given to the corresponding method in its own service. The difference is that the application also provides an object which implements a callback interface. This object is then invoked on a different thread when the response is received.

It is important to note that nothing changes with respect to the messages being exchanged between the application machine and the connector server when using Asynchronous Services

versus making calls into the other Services. The exact same messages are sent with the exact same timing in either case. The only difference is in the threading model in the Java application.

Service requests

In this API an application requests services of the AE Services server server. Examples of a request are “give me a device identifier”, “press a button”, “give me information about a lamp’s status”, or “notify me when a tone comes into the device”.

Each request is processed by the AE Services server. The server may complete a request synchronously in one logical step or it may take additional steps to pass the request to Communication Manager and handle the response(s) before responding asynchronously to the application with the request’s results in an event or exception.

To make a request, the application must set up the appropriate request Java bean before passing the bean to the service method. Each parameter in the request bean is specified with either a `set` method, or in the case of a list parameter, with an `add` method.

For example, to set the values of the list parameter called Codecs in the `MediaInfo` bean, there are these methods to choose from:

- `addCodecs (String vCodecs)`

  This method adds the codec string to the end of the array.

- `setCodecs (String[] codecsArray)`

  This method sets the array elements to values of the `codecsArray` elements.

  Tip:

  It is recommended that you *not* use the methods that have index parameters:

  - `addCodecs (int index, String vCodecs)`
  - `setCodecs (int index, String vCodecs)`

Service responses

Each service request is acknowledged with a service response (positive acknowledgement). Some service requests, particularly those that request information, are completed in a single logical step, thus the results of the request are reported in the service response (single-step requests follow CSTA’s atomic model). Other service requests, particularly those that require the connector server to pass on a request to Communication Manager, take multiple steps to complete. (These follow CSTA’s multi-step model.) Responses to multi-step requests merely indicate that the request was received and is being processed.

In some cases, response data values may indicate an error in the request or in the processing of a single-step request. However, most errors are indicated through exceptions (see Negative acknowledgements below) or negative events (see Events below).

Service responses are in the form of Java beans. To get the information out of a response bean, use the `get` methods. List parameters have multiple types of gets, such as these for getting one or more `ButtonItem`s from a `ButtonList`:

- `ButtonItem[] getButtonItem()`

  Returns the entire array of button items.

- `ButtonItem getButtonItem`(int index)

  Returns the ButtonItem in position `index`.

- int `getButtonItemCount`()

  Returns the number of ButtonItems in the array.

- `Enumeration enumerateButtonItem()`

  Returns an enumeration of `ButtonItems`.

Events

Events are asynchronous occurrences on a device that an application can choose to listen for and respond to. Examples of events are the `DisplayUpdatedEvent` in `PhysicalDeviceListener`, which indicates that the device's display has changed, or `TerminalUnregisteredEvent` in `RegistrationListener`, which indicates that the device has been unregistered by Communication Manager.

An application indicates its desire to listen for events by adding listener implementations. Listener implementations implement pre-defined listener interfaces or extend pre-defined listener adapters. Listeners provide a callback method implementation for one or more listener methods. When the AE Services server determines that a particular event has occurred on a device, it sends the event to the client proxy for each listener that was added, which in turn calls the corresponding listener callback method.

A callback method, and all the methods called from a callback method, execute in the context of a client event thread that is created by the client proxy. Therefore, your application will not be notified of subsequent events until the previous event's callback method is complete.

Information about the event is provided to the callback method via a Java bean parameter. Use the bean's `get` methods to extract the information. Do not use the bean's `set` methods.

Note:

Once you receive an event, release the event thread immediately and continue to do event processing on a different thread.

Negative acknowledgements

A negative acknowledgement indicates that the service request has failed. CSTA error categories are used to categorize errors into a hierarchy of error return values. Negative acknowledgements (errors) are implemented as Java *exceptions* in this API. Each exception includes an error value and a printable description.

Signaling Encryption

In AE Services R3.1 is the ability to encrypt the signaling channel between the AE Services server and Communication Manager was added. However, the option to encrypt this link is not under the direct control of your application. Instead, it is an option that is provisioned on Communication Manager, via the “set network-region” page. See the "Setting up a network region for Device and Media Control″ subsection of the "Administering Communication Manager for AE Services" chapter of the *Avaya MultiVantage™ Application Enablement Services Administration and Maintenance Guide*.

The encryption of the signaling link is automatically negotiated between the AE Services server and Communication Manager during the device registration, and the resultant encryption used is dependent on the options set for the network region. For AE Services R3.1, the signaling encryption will be one of the following types:

- Challenge
- Pin-Eke

When registering the device using Registration Services, the negotiated signaling encryption type is included as a parameter in the `RegisterTerminalResponse` message.

Media Encryption

In AE Services R3.1 is the ability to encrypt the RTP (media) stream between the application and the other endpoint of the call was added.

The option to encrypt the RTP stream is provisioned on the Communication Manager, this time via the “set codec” page. See the "Creating the Device and Media Control codec set″ subsection of the "Administering Communication Manager for AE Services" chapter of the *Avaya MultiVantage™ Application Enablement Services Administration and Maintenance Guide*. However, in this case, the application does have some (albeit limited) control over the type of encryption used. For AE Services R3.1, the media encryption will be one of the following types:

- Advanced Encryption Scheme (AES)
- no media encryption

When registering the device, the application may specify which media encryption schemes that it supports. For more details see Choosing the media encryption on page 65.

This list of application-supported media encryption schemes is matched to the list of encryption schemes provisioned on Communication Manager’s “set codec” page for the device’s codec set. Communication Manager will pick an encryption scheme that is common to both lists, if possible.

Note that the encryption scheme chosen by Communication Manager will be indicated in the `MediaStartEvent` message. See MediaStartEvent Handling on page 91.

Accessing the client API reference documentation

You will need to frequently reference the *Java Programmer's Reference* (Javadoc)provided with this API. It is available on the Avaya Support site (from the Avaya support website (www.avaya.com/support) as both a viewable HTML document and a downloadable zip file.

If you choose to download the zip file, then do the following to browse the HTML-based Javadoc:

1. Go to the cmapijava-sdk/docs/api directory.
2. Double click on index.html (or open this file with a browser).

The Javadoc includes descriptions of the classes in these three areas:

- the service layer (in the api directory)
- the Java beans representing CSTA XML data elements (in the castor directory)
- the Java beans representing Avaya XML data elements (in the castor_avaya directory)

A three-pane window should display entitled **Device, Media and Call Control API**.

- The right pane gives you an overview of the API and its packages.
- The upper left pane lists the service layer's packages.
- The lower left pane lists the service layer's interfaces and classes.

Note:

The Java bean packages are not listed with the service layer packages. Nor are the Java beans listed with the service layer interfaces and classes. You can reach the Java bean Javadoc from the service layer methods that use them.

Learning from sample code

Another key learning tool is the set of sample code files that are provided with this API. Five sample applications and a sample class are provided in this directory:

cmapijava-sdk/examples/src/sampleapps:

Table 23: Sample code

Application or Class	Registration Mode Used	File path names under cmapi/examples/src/sampleapps	Description
Simple Tutorial application	Exclusive Control Server media mode	tutorial/TutorialApp.java	This simple application allows you to record a message and then have it played back. The application also demonstrates the use of media encryption.
Simple Interactive Voice Response (IVR) application	Exclusive Control Server media mode	simpleIVR/ SimpleIVR.java	This application sets up a softphone that serves as a simple IVR system. It plays a greeting and gives instructions to a caller using Voice Unit Services and detects DTMF tones pressed by the caller using Tone Collection Services. Based on the caller's selections, it may record the caller's voice stream, play back the recording, conference in another extension, or transfer to another extension.
Missed-call Email application	Shared Control mode	email/EmailApp.java	This application monitors a station's incoming calls, using shared control of the station and Physical Device Services, and sends an email to the station owner for each missed call. The email includes the caller's name (if available) and phone number and the time of the call.

1 of 4

Table 23: Sample code (continued)

Application or Class	Registration Mode Used	File path names under cmapi/examples/src/sampleapps	Description
Click to Call application	Shared Control mode	click2call/Click2Call.java	This application registers an extension with Communication Manager and monitors that extension for incoming and dialed calls. This application also provides a GUI to perform directory lookup using LDAP. Incoming call data is collected from display updates on monitored extension and displayed in a GUI as caller id, caller number, call time and status (answered, missed, called back). Dialed calls are also logged. Calls of different status are displayed in different colors. The user can use the call log GUI to return a call or the directory lookup GUI to make a call. This application provides a good demonstration of the proper recovery techniques. Additionally, this application has a login screen to supply a username/password, which are authenticated on the AE Services server.

2 of 4

Table 23: Sample code (continued)

Application or Class	Registration Mode Used	File path names under cmapi/examples/src/sampleapps	Description
Simple Media Stack application	Exclusive Control Client media mode	clientmediastack/ TutorialMediaStack.java	This application registers an extension with Communication Manager and waits for that extension to be dialed. It also adds a sample Speaker and Microphone to the Audio RTP Channel. When a user dials the associated extension number, the application answers the incoming call. The Speaker and Microphone are enabled. The incoming Audio RTP stream is played on the Speaker and the audio from Microphone is played onto the outgoing Audio RTP stream. Since different systems have different Sound cards, the sample Speaker and Microphone may need to be modified. Also, the Speaker and Microphone does not support transcoding but this can be added by the programmer. The application also demonstrates the use of media encryption.

3 of 4

Table 23: Sample code (continued)

Application or Class	Registration Mode Used	File path names under cmapi/examples/src/sampleapps	Description
Station class		station/AvayaStation.java (interface) station/AvayaStationImpl.java (implementation) station/AvayaStationListener.java station/AvayaStationAdapter.java	This class shows how to create a higher level abstraction of Terminal Services and Physical Device Services.
Telecommuter Application	Exclusive Control Telecommuter mode	telecommuter/TutorialTeleCommuterApp.java	This application registers an extension with Communication Manager in Telecommuter mode. The application has signaling control while media is delivered to any other phone (on or off the pbx) which is also called the Telecommuter extension. This mode provides remote access to station features of Communication Manager at the same time allowing use of one's regular phone or cell-phone. When a user dials the extension, the application answers the call. Communication Manager then establishes a talk path between the telecommuter extension and the calling party.

4 of 4

Look at the TutorialApp first to learn the basics of writing an application that utilizes both the physical and logical elements of a device. You can compile and run the applications by following the instructions found in:

cmapijava-sdk/examples/bin/Readme.txt

Communication Manager must be properly installed and configured and the AE Services server must be set up before the sample applications can run. Please see the appropriate *Avaya MultiVantage™ Application Enablement Services Installation Guide* for the offer you have purchased (bundled server or software only).

After studying the sample code, you may also want to modify or extend the sample code. Feel free to make the modified code part of your product. However, if you do choose to include any of the sample code in your production code, no guarantees are made by Avaya.

Chapter 3: Writing a client application

This chapter describes how to write an application using the Application Enablement Services Device, Media and Call Control API. It will frequently refer to the details in the Javadoc, so you may wish to have ready access to the Javadoc while reading this chapter. Read Accessing the client API reference documentation on page 38 to find out how to get access to the Javadoc and where to find which kinds of information within the Javadoc.

Your application may have these different parts:

- Setup
  - Imports
  - Catching exceptions
  - Getting access to desired services
  - Session Management
  - Getting device identifiers
  - Requesting notification of events
  - Registering devices
- Telephony Logic for performing actions after various events occur
  - Monitoring and controlling physical elements
  - Recording and playing voice media
  - Detecting and collecting DTMF tones
- Recovery
- Cleanup
  - Stop collecting tones
  - Stop recording or playing
  - Unregister the device
  - Remove the listeners
  - Release the device identifier
  - Disconnect the socket

Each part is described in the sections below.

Setup

This section describes the different setup steps that must be taken by every application.

Imports

The classes that provide the service layer are found in:

- **ch.ecma.csta** - holds interfaces and classes that implement the CSTA standard
- **com.avaya.csta** - holds interfaces and classes that extend the CSTA standard
- **com.avaya.cmapi** - holds interfaces and classes for the Service Provider, a lookup factory

The Java beans used by the service layer are found in:

- **ch.ecma.csta.binding -**holds Java beans for CSTA data elements
- **com.avaya.csta.binding** - holds Java beans for Avaya data elements

The minimum set of imports your application will need to have are:

```
import com.avaya.cmapi.ServiceProvider;
import com.avaya.csta.device.DeviceServices;
import com.avaya.csta.terminal.TerminalServices;
import ch.ecma.csta.physical.PhysicalDeviceServices;
import ch.ecma.csta.errors.CstaException;
import ch.ecma.csta.binding.*; // or specific CSTA beans used
import com.avaya.csta.binding.*; // or specific Avaya beans used
```

Catching exceptions

Each service request may throw an exception; therefore your application must be prepared to catch exceptions with a try/catch block around service requests.

Tip:

Be sure to always catch at least the parent class exception. For most methods, that is `CstaException`.

It is recommended that the application catch and log all possible exceptions since this will be an important source of information for debugging the application. See the Javadoc to determine the types of exceptions that may be thrown by any given method.

Session Management

An application session must be established between your application and the AE Services server for the purpose of exchanging application messages. It is required that an application session be established before application messages can be exchanged. This allows you to recover and reestablish an existing session on a new connection. The application session is established by the `ServiceProvider` class.

Getting access to desired services

The `ServiceProvider` class in the `com.avaya.cmapi` package is the starting point for all applications to:

- indicate what connector server to connect to
- identify the application
- establish an application session
- gain access to all API services

Getting a ServiceProvider instance

You will first get an instance of a `ServiceProvider` object. This creates a socket connection between the client and the AE Services server and starts an application session associated with this socket connection.

Your application will get an instance of `ServiceProvider` using the `ServiceProvider` method called `getServiceProvider:`

```
public static ServiceProvider getServiceProvider(Properties properties)
```

It takes a `Properties` parameter where the properties shown in the following two tables are set as necessary:

Table 24: Required Properties

Property	Description
cmapi.server_ip	Set to IP address or hostname of the AE Services server.
cmapi.server_port	Set to the port number of the AE Services server: • unsecured 4721 • secured 4722
cmapi.username[a]	Set to user name of the application. This property will be authenticated by the Device, Media and Call Control service
cmapi.password[b]	Set to password of the application. This property will be authenticated by the Device, Media and Call Control service

a. In previous releases the username and password properties were required, but were not authenticated. These properties will now be authenticated by default with the User Service. See the *Avaya MultiVantage™ Administration and Maintenance Guide* for the authentication options and directions on administering users.

b. See Footnote a.

Table 25: Optional Properties

Property	Description
cmapi.session_id	Set to the SessionID that was generated by the AE Services server upon session establishment. The SessionID is optional and should only be set if the application is attempting to recover a session.
cmapi.session_duration_timer	Set to the requested amount of time (in seconds) for a keepalive interval. We recommend that you do not override the default of 180 seconds.
cmapi.session_cleanup_delay	Set to the requested amount of time (in seconds) that a session is to be placed in an inactive state before it is cleaned up. We recommend a cleanup delay time of 60 seconds. The default is set to 0.
cmapi.api_version	Set to the api version if you do not want the default of 3.1.
cmapi.application_description	Not currently used, but could be used in future releases to see what applications are running against an AE Services server.
cmapi.trust_store_location[a]	Specify a path to the Java Key Store file if not using the default in the resources directory.
cmapi.secure	Set to "true" if connecting to the secure port, set to "false" otherwise. This will default to "false". We strongly recommend that you use the secure port and set this value to "true".

a. The cmapi.trust_store_location needs to be used only if the application developer wants to supply the keystore. We recommend using the default. By default, the Java SDK tries to load the file avaya.jks from the ../resources directory (relative to where the application was launched). If that file cannot be found when your application starts, set this property to the full path to the file avaya.jks, including the file name.

To use the secure port set up and pass in a `Properties` object like this:

```
Properties properties = new Properties();
properties.setProperty (“cmapi.secure”, "true");
properties.setProperty (“cmapi.server_ip”, "132.7.82.40");
properties.setProperty (“cmapi.server_port”, "4722");
properties.setProperty (“cmapi.username”, "application1");
properties.setProperty (“cmapi.password”, "secretpasswd");
properties.setProperty (“cmapi.session_duration_timer”, "180");
properties.setProperty (“cmapi.session_cleanup_delay”, "60");
properties.setProperty (“cmapi.api_version”, "3.0");
ServiceProvider servProv = ServiceProvider.getServiceProvider(properties);
```

Note:

The secure port is now 4722.

To use the non-secure port set up and pass in a `Properties` object like this:

```
Properties properties = new Properties();
properties.setProperty (“cmapi.secure”, "false");
properties.setProperty (“cmapi.server_ip”, "132.7.82.40");
properties.setProperty (“cmapi.server_port”, "4721");
properties.setProperty (“cmapi.username”, "application1");
properties.setProperty (“cmapi.password”, "secretpasswd");
properties.setProperty (“cmapi.session_duration_timer”, "180");
properties.setProperty (“cmapi.session_cleanup_delay”, "60");
properties.setProperty (“cmapi.api_version”, "3.0");
ServiceProvider servProv = ServiceProvider.getServiceProvider(properties);
```

Note:

In previous releases the `properties` parameter may have been set to null and the properties taken from a properties file (cmapisdk/lib/cmapi-client.properties). In this release the properties file is only supported for backwards compatibility. You may continue to use the properties file method in this release, but it is recommended that you move to using the new `Properties` object method.

Note:

If you want to use the unsecure port, on the server side you must change the OAM page to enable the unencrypted port, and on the application side, set the properties `cmapi.secure` and the `cmapi.server_port` appropriately.

If your application requires access to multiple AE Services servers, then one `ServiceProvider` must be instantiated for each server.

An application that has multiple threads in the same JVM can either call `getServiceProvider()` once for the application or once for each thread. Only the first call will create a connection (socket) to the specified AE Services server. Subsequent calls for the same server result in the first connection being reused. Any of the threads can request services for a device that was registered in one of the threads. Therefore, device identifiers can be shared across threads as long as the threads are in the same JVM.

If an application has multiple JVMs, then each JVM must have its own `getServiceProvider()` call that results in its own connection to a connector server. As a

result, a device that is registered in one JVM cannot be acted upon with services from another JVM.

The following figure shows the sequence of events that occurs between the client application, the API library and the AE Services server during establishment of an application session and subsequent session management.

Figure 1: Session Management

When the application requests an instance of `ServiceProvider` using the `getServiceProvider` method, the API library will create a socket connection between the client and the server and start an application session associated with this socket connection.

The API will send a `StartApplicationSession` request to the server containing the Cleanup Delay, and the Application Session Duration that you specified in the `Properties` object. The Application Session Duration is the amount of time that the server will wait before moving the session to the inactive state. This timer is reset when the server receives a `ResetApplicationSessionTimer` message from the application.The Java API automatically handles sending these keepalive messages to the server on the behalf of the

application at intervals that are approximately 1/3 the session duration. This allows for up to 2 messages to be lost / delayed before the application session is put into an inactive mode.

The subsequent `StartApplicationSessionResponse` sent by the server to the API, will contain a SessionID, the *actual* session duration and the protocol version.

Note:

The AE Services server only allows session durations between 30 seconds and 2 hours. If you request a duration shorter than 30 seconds the duration will automatically be set at 30 seconds and if you request a duration longer than 2 hours it will automatically be set at 2 hours.

If the server session duration timer expires due to not having received the `ResetApplicationSessionTimer` message, it will:

- close the socket
- mark the session as inactive
- start the cleanup timer using the time specified in the Cleanup Delay.

If the application then attempts to send any messages, it will receive a `ServerSessionNotActiveEvent` on their `ServiceProvider` listener. If this happens, the application should then take recovery actions as specified in the recovery section (Recovery on page 82).

Note:

If authenticaion of the username and password was not successful, a `java.lang.SecurityException` will be thrown with a message indicating authentication failure.

Getting an instance of each set of services

The sets of services available in this API include those shown in the following table:

Table 26: Services available through `ServiceProvider.getService` method

Service interface	Found in package	Required?
Asynchronous Services	com.avaya.cmapi.asynchronous	Only required if the application wishes to use the Asynchronous Service to do registration. Not required if the application will use the synchronous Registration Services
Call Control Services	ch.ecma.csta.callcontrol	Only required if the application needs to perform SingleStepConference operations.

1 of 2

Table 26: Services available through `ServiceProvider.getService` method (continued)

Service interface	Found in package	Required?
Call Information Services	com.avaya.cmapi.callinformation	Only required if the application needs detailed information about a call in progress (such as VDN number, agent ID, etc).
DeviceServices	com.avaya.csta.device	Yes
ExtendedVoiceUnitServices	com.avaya.csta.voiceunit	Only required for dubbing and for controlling recording and playing independently
Monitoring Services	ch.ecma.csta.monitor	Always required to receive notification of events. If nothing else, an application needs to watch for the registered event.
PhysicalDeviceServices	ch.ecma.csta.physical	Yes
Registration Services	com.avaya.cmapi.registration	Yes
Snapshot Services	ch.ecma.csta.snapshot	Only required if the application needs to use SingleStepConference, if they would like to see all parties on a call, or all calls on a device
ToneCollectionServices	com.avaya.csta.tonecollection	Only required for collecting a series of DTMF tones
ToneDetectionServices	com.avaya.csta.tonedetection	Only required for detecting DTMF tones
VoiceUnitServices	ch.ecma.csta.voiceunit	Only required for recording and playing messages

2 of 2

Get an instance of each set of services that you plan on using in the application. Every application will need at least the following sets of services:

- `DeviceServices` to get device identifiers
- `RegistrationServices` to register devices
- `PhysicalDeviceServices` to manipulate and monitor the physical device.
- `MonitoringServices` to request notification of events.

Get an instance of each of these services using Service Provider's `getService` method, like this:

```
DeviceServices devServ = (DeviceServices)servProv.getService
(DeviceServices.class.getName());
RegistrationServices regSvcs = (RegistrationServices)servProv.getService
(com.avaya..csta.registration.RegistrationServices.class.getName());
PhysicalDeviceServices physDevServ = (PhysicalDeviceServices)servProv.getService
(PhysicalDeviceServices.class.getName());
```

If an application calls the `getService()` method multiple times for any given service using the same `ServiceProvider`, then each call returns a reference to the same instance of the service. The services are thread safe. Service requests are sent to the connector server and processed one at a time.

Note:

You must `get` all of the services you wish to monitor, before you `get` Monitoring Services.

Getting device identifiers

Set up a device identifier for each Communication Manager phone or extension that your application needs to work with.

In past releases of Device and Media Control, there was only one type of device identifier. With the introduction of third party call control through Call Control Services and Snapshot Services, a new type of device identifier is required. There is now first party device identifiers and third party device identifiers.

First party device identifiers are the types of device identifiers that have been supported in past releases. These device identifiers have the following properties:

- Are exclusive to a session
- Can only be obtained from Device Services
- Can be used to register a virtual softphone and perform device and media control
- Can also be used for third party call control operations, but only if they contain a Communication Manager connection name
- Must be released to be cleaned up

Third party device identifiers are new for the 3.1 release. These device identifiers have the following properties:

- Are not exclusive to session
- Can be obtained from Device Services or may be returned by Call Control Services/ Snapshot Services
- Can only be used with Call Control Services and Snapshot Services
- Need not be released

There are several valid ways to get a first party device ID. It is recommended that the application take advantage of the H.323 Gatekeeper List feature so that both first party device IDs and third party device IDs are acquired in the same way: by giving a switch name and an extension number. This feature requires the administrator to administer through the OAM web pages a list of IP addresses that can be used for H.323 registrations. See the "Administering Switch Connections" section of the "AE Services OAM Administration and CTI OAM Admin" chapter of the *Avaya MultiVantage™ Administration and Maintenance Guide*.

If an application is not using Call Information Services, Call Control Services or Snapshot Services, it is also valid to include only a `switchIPInterface` and extension number in the `getDeviceID` request, and thereby not administer the H.323 Gatekeeper list.

Create a device identifier using the `DeviceServices.getDeviceID` method. For details about this method, see the Javadoc in the `com.avaya.csta.device` package under the `DeviceServices` interface.

Your first party device identifier creation code will look something like this:

```
// Given a switch address & extension, create a first party device ID
request = new GetDeviceId();
request.setSwitchName(switch);
request.setExtension("54444");

DeviceIDResponse response = devServ.getDeviceID(request);
DeviceID deviceID = response.getDevice();
```

Your third party device identifier creation code will look something like this:

```
// Given a switch address & extension, create a third party device ID
GetThirdPartyDeviceId devReq = new GetThirdPartyDeviceId();
devReq.setExtension(deviceToJoin);//set the ext to call
devReq.setSwitchName(switch1);

GetThirdPartyDeviceIdResponse devResp = devSvcs
  .getThirdPartyDeviceID(devReq);
DeviceID calledDevice = devResp.getDevice();
```

Device identifiers can be shared across threads in the same JVM, but cannot be shared across JVMs.

Populating the Switch Name field

For first party device ID's:

There are several ways to populate the `switchName` field:

- Your application can specify a `switchName`, a `switchIPInterface`, and an `extension` when getting a device ID. In this case, administration of the H.323 Gatekeeper list for the switch connection (transport link) is not required.
- Your application can get a `switchName` by using the Gatekeeper List feature of the AE Services server. Your application would specify just an `extension` and `switchIPInterface` as in previous releases of the API. If this is done, it is required

that you administer an H.323 Gatekeeper list for the Switch Connection. The AE Services server, upon receiving the GetDeviceID request, will go to its administration database, and will resolve the given switchIPInterface to a switchName, then populate this value in the DeviceID.

- The AE Services server now offers a feature which will use a round-robin algorithm to distribute softphones to a list of H.323 Gatekeepers. If using this feature, the application need only know a symbolic name for the switch (i.e. switchName), rather than maintaining a list of Gatekeepers with which it wishes to register. To take advantage of this feature, the list of H.323 Gatekeepers must be administered in web OAM for the given Switch Connection. When getting a DeviceID, the application would specify only a switchName and extension. Upon receiving the DeviceID request, the switch will pick the next H.323 Gatekeeper from the list, and register with that gatekeeper.

For third party device ID's:

- Your application must always specify a switchName, and an extension when getting a device ID.
- There is no IPInterface for third party device ID's.

Note:

The switchName field is only required for Call Information Services, Call Control Services and Snapshot Services. If an application is not using one of these services and does not wish to take advantage of the round-robin H.323 Gatekeeper assignment feature, it is not required to administer an H.323 gatekeeper list or specify a switchName in the getDeviceID request. In this case, the switchName field of the DeviceID is simply not populated, and any calls with this DeviceID to Call Information Services, Call Control Services or Snapshot Services will fail.

Requesting notification of events

Your application can choose to be notified of asynchronous events that occur on a device by implementing and adding listeners. Asynchronous events include:

- Telephony events.

  Examples of telephony events are when the lamp state has changed or a DTMF tone has been detected.

- Asynchronous responses to service requests.

  For example, after requesting to register a device using the Asynchronous Services, an event is received indicating whether the request succeeded or failed.

This API uses Java listeners to monitor the occurrence of events and automatically invoke listener callback methods when their associated events occur. CSTA refers to this asynchronous event handling as *monitoring*.

To listen for certain types of events, an application must:

1. Implement a listener.
2. Add the listener.

Note:

Once you receive an event, release the event thread immediately and continue with event processing on a different thread.

The listeners that an application may choose to implement/extend and add are:

Table 27: Listeners per package

Package	Listener	Types of events
`com.avaya.csta.registration`	`RegistrationListener` RegistrationAdapter	Registration events
	`MediaControlListener` MediaControlAdapter	Far-end RTP stream changes
`ch.ecma.csta.physical`	`PhysicalDeviceListener` PhysicalDeviceAdapter	Physical element status change events
`ch.ecma.csta.voiceunit`	`VoiceUnitListener` VoiceUnitAdapter	Media record and play events
`com.avaya.csta.tonedetection`	`ToneDetectionListener` ToneDetectionAdapter	DTMF tone detection event
`com.avaya.csta.tonecollection`	`ToneCollectionListener` ToneCollectionAdapter	DTMF tone retrieval event

Implementing listeners

A listener can be implemented in either of these ways:

- Extend a listener *adapter* in order to implement only a subset of the callback methods and to use the adapter's default "do nothing" methods for the remainder.

  For example, you could extend the `PhysicalDeviceAdapter` class to implement only the `lampUpdated` and `ringerStatusUpdated` methods so that you receive lamp updates and ringer updates and ignore all other physical device events.

- Implement the entire listener *interface* in order to implement all callback methods.

Adding listeners

Listeners should be added for a device before the device is registered so that events are not missed. To add a listener, use the appropriate monitoring “add listener” method. For example, to begin listening for `PhysicalDeviceListener` events for a particular device, you would add your implementation of `PhysicalDeviceListener` with the `MonitoringServices addPhysicalDeviceListener` method, like this:

```
monSvcs.addPhysicalDeviceListener(deviceID, myPhysDevListenerImpl);
```

You are allowed to add multiple implementations of a single kind of listener. For example, you could implement two different versions of `RegistrationListener` and add them both. In this case, each added listener gets called when an event occurs. This may be useful in cases where different objects need to be notified of the same event.

CAUTION:

It is a common mistake to *inadvertently* add the same listener implementation multiple times. This is allowed, but may not be what your application intends. Check your logic to be sure your code adds only the number of listeners it intends.

For more information about implementing a listener, see Telephony Logic on page 68 or see the Javadoc for a particular listener.

Removing listeners

Your application should be removing listeners if they are not being used. Adding listeners without removing them can use up server resources.

Registering devices

To monitor or control a device, an application must register the device with Communication Manager. This tells Communication Manager whether you want exclusive or shared control of the device and what kind of media access you want. Only devices that are softphone enabled on Communication Manager’s **Station** form can be registered.

The list of phone types that can be registered is shown in the Table 28: Telephone configurations controllable by AE Services Device, Media and Call Control API on page 59. For details on how to softphone enable a device, see the appropriate *Avaya MultiVantage™ Application Enablement Services Installation Guide for the offer you have purchased (bundled server or software only)*.

Registration is performed through the `registerTerminal` request of the Registration Services interface found in the `com.avaya.cmapi.registration` package. In the `RegisterTerminal` request you must at a minimum specify which device to register and the password that is administered for the device on the **Station** form. There are also a number of

options to choose from. For more details on setting up a `RegisterTerminal` request, see the programmer's reference (Javadoc).

In previous releases of Device and Media Control, the responses to registration / unregistration requests were simply acknowldgements by the server of receipt of the request. The actual outcome of the request was reflected through events that arrived later.

This has been changed with Registration Services. The responses to registration and unregistration requests indicate the outcome of the request. This can substantially increase the amount of time needed to receive the response, so the application writer may wish to use Asynchronous Services to receive the response asynchronously.

The only registration event supported with Registration Services is the `TerminalUnregistered` event. This event is sent if the AE Services server automatically unregisters the device. The typical cause is when the network or Communication Manager is unresponsive.

If a device becomes automatically unregistered, it is up to the application to re-register. Avaya recommends retrying every 30 seconds.

Some important decisions you will need to make when registering a device include:

- What device control mode to choose
- What media mode to choose
- What codecs to choose
- What media encryption to choose

Controllable telephone types

The AE Services server can control the following types of Communication Manager telephones and extensions when they are administered for softphone access:

Table 28: Telephone configurations controllable by AE Services Device, Media and Call Control API

Telephone configuration	Administered set type	Administered port address	Comments
DCP telephone	Any DCP type that can be administered for softphone access	Physical port address, such as 1B0201.	If an application requests exclusive control, the telephone either goes dead or becomes a TTI set, depending on how it is administered. After the application relinquishes control, control goes back to the DCP telephone. If an application requests shared control, the telephone stays active. (Communication Manager 2.0 and higher)
DCP extension administered without hardware (AWOH)	Any DCP type that can be administered for softphone access	Port address set to “X”.	

1 of 2

Table 28: Telephone configurations controllable by AE Services Device, Media and Call Control API (continued)

Telephone configuration	Administered set type	Administered port address	Comments
IP telephone that is logged in	Any IP type that can be administered for softphone access	Internal software port address, such as S00031, is automatically assigned.	If an application requests exclusive control, the IP telephone is logged off. When the application relinquishes control, the IP telephone must be powered off and on again to be logged back on. If an application requests shared control, the telephone stays active. (An application can gain shared control of an IP phone only if the IP phone is connected to Communication Manager 2.1 or later.) **Note:** Only IP telephones can be controlled. IP softphones can not be controlled by the API.
IP extension without hardware (i.e., no IP telephone logged in using that extension)	Any IP type that can be administered for softphone access	Internal software port address, such as S00031, is automatically assigned.	

2 of 2

Choosing a device control mode

At registration time, the application specifies the desired device control mode. This indicates who controls the device's physical elements and media. The device control choices are:

- **Exclusive control mode**

  Gives all control of the device to the application including control of the media stream. This must be used if the application will use Voice Unit Services, Tone Detection Services, or Tone Collection Services.

- **Shared control mode**

  Gives control to both the telephone and the application. This must be used by applications that need to monitor and control a physical telephone. Cannot be used to monitor or control a softphone. The media always goes to the physical telephone.

Note:

You cannot register an extension in shared control mode if a softphone is already registered with that extension. This is true regardless of whether the softphone itself is in exclusive control or shared control mode.

The following table shows what the physical telephone and application can do in each of the control modes.

Table 29: Device control modes

Capabilities	Shared control		Exclusive control	
	Physical telephone	Application	Physical telephone	Application
Initiate action on device (e.g., press buttons)	Yes	Yes	No	Yes
Be notified of status changes to device (e.g., hear ringback or receive ringback event)	Yes	Yes	No	Yes
Receive incoming media stream and send message or voice via outgoing media stream	Yes	No	No	Yes
Be notified of actions taken on device	No[a]	No[b]	N/A	Sometimes[c]

a. In shared control mode, the user of a telephone is not notified of actions initiated by an application except through resulting status changes to the device's lamps and display.

b. In shared control mode, the application is not notified of actions initiated by a user of the telephone except by status changes to the device's hookswitch, lamps, ringer, and display.

c. In exclusive control mode, even though the application is initiating all actions on the device, the application is not always notified when Communication Manager has completed performing those actions except by status changes to the device's hookswitch, lamps, ringer, and display. More specifically, the application *is* notified when its off hook request has been fulfilled, but it is *not* notified when a button press request has been fulfilled.

The differences between exclusive and shared control are more thoroughly described in the "Device, Media and Call control*"* section of the "Capabilities of the API" chapter of the *Avaya MultiVantage™ Application Enablement Services Overview* .

Media control modes

When a device is registered by an application in exclusive control mode, the application has access to the real-time protocol (RTP) media stream coming into and going out from the softphone. There are three media modes available in exclusive control mode: server media, client media, and telecommuter. When the device is registered in shared device control mode, then only the telephone has control of the RTP stream. The following table, Media control modes, explains the different media modes and the media access for the call control modes.

Table 30: Media control modes

Control mode	Media mode	What handles media	How it works
Exclusive control	Server media	AE Services server	The AE Services server handles the media coming to the softphone and going out from the softphone. In this mode, the application uses the API services called Voice Unit Services to record and play media. The application selects this mode at the time it registers the softphone by letting the local media RTP address default to the AE Services server.
	Client media	Application	The application handles the media coming to the softphone and going out from the softphone. The application selects this mode at the time it registers the softphone by specifying the local media RTP address to where the RTP stream should be sent.
	Telecommuter	Telecommuter phone	The media is given to the real phone (telecommuter phone) only. This phone may be within the Communication Manager domain or outside of the Communication Manager domain.
Shared control	Telephone media	Telephone	The media stream is given to the telephone only. In order to use Shared Control, use the `LoginInfo.setSharedControl` method in `RegisterDevice.setLoginInfo`.

The RTP parameters that an application can control or state preferences for at registration time are:

- Local RTP and RTCP addresses
- Coder/decoder (codec): G.711 A–law, G.711 Mu–law, G.729, G.729A

Choosing a media mode

If the application registers the device with shared control, then the media is handled only by the telephone. However, if the application requests exclusive control, then the application can choose where the media is terminated and who handles the media. The following table shows the media modes available, when to use each, and how to request each:

Table 31: Choosing a media mode

Exclusive-control media modes	When to use	How to set
Server media mode	This mode is used when the application wants the AE Services server to handle media processing or if no media processing is needed at all. The AE Services server handles media with Voice Unit Services and Tone Detection Services or Tone Collection Services. Voice Unit Services is used to record and play messages. Tone Detection Services or Tone Collection Services are used to detect out-of-band DTMF tones. In server media mode, the media stream terminates on the AE Services server. A `MediaControlListener` may be used in this mode to detect changes to the far-end RTP/RTCP parameters.	If you want codecs other than the default, set just the codecs in `LocalMediaInfo`, but do *not* set the RTP/RTCP address in `LocalMediaInfo`. To use the default codecs, do not set anything in `LocalMediaInfo`.
Client media mode	This mode us used when application wants to process the media itself. The RTP stream can be terminated on any machine that can be controlled by the application. A `MediaControlListener` may be used in this mode to detect changes to the far-end RTP/RTCP parameters. `ToneDetectionServices` or `ToneCollectionServices` can be used to detect out-of-band DTMF tones.	Set `LocalMediaInfo`'s RTP/RTCP address to the IP address and port where the media stream is to be terminated. I you want codecs other than the default, set the codecs in `LocalMediaInfo`. May also set media encryption. The media can be terminated on any machine that can be controlled by the application.
Telecommuter mode	This mode is used when an application wants the media to go to a real telephone. The real telephone can be an internal extension to Communication Manager or a PSTN telephone number.	Set `LocalMediaInfo`'s `telecommuter` to the telephone number of the real telephone that you are directing the media to.

The following table shows the media control capabilities of the AE Services server for both server media mode and client media mode.

Table 32: Media Control Capabilities

Media control capabilities	server media mode	client media mode
Record media from a call into a WAV file	provided by the server	
Dub a recording with the contents of another compatible WAV file	provided by the server	
Play a voice announcement or tone from a prerecorded WAV file	provided by the server	
Play a list of prerecorded announcements from separate WAV files	provided by the server	
Stop, pause, or resume outstanding play or record operations	provided by the server	
Detect out-of-band DTMF tones	provided by the server	
Detect inband DTMF tones	provided by the server	
Manage media related event listeners	provided by the server	
Originate and terminate RTP streams on any machine that the application has control of		provided by the application.
Media encryption		provided by the application.[a]

a. Media encryption is provided by the application unless the application is using the Avaya provided client media stack in which case it will be provided by the server.

Choosing a codec

A codec is the algorithm used to encode and decode audio media. For devices being registered in exclusive control mode, the application can optionally specify at registration time what codecs are preferred for the device. The codec options are:

- G.711 A-law (g711A)
- G.711 Mu-law (g711U)

- G.729 (g729)
- G.729 Annex A (g729A)
- G.723 (supported as an option for Client Media mode only)

Specify the set of codecs your application supports and list them in preferential order in the `LocalMediaInfo` parameter of `RegisterTerminal` request. If you do not specify a set of codecs in the `LocalMediaInfo` parameter of the `RegisterTerminal` request, the connector server will default to G.711 A-law as the first choice and G.711 Mu-law as the second choice. If Communication Manager cannot satisfy your request for specific codecs, then calls will still go through, but there will be no media.

Note:

For server media mode you cannot specify a mixture of G.711 and G.729 codecs for a single device. This is because there is no conversion offered by the server.

For more information about selecting and administering network regions and their codecs, see Chapter 1 of the *Avaya MultiVantage™ Administration and Maintenance Guide*.

Choosing the media encryption

For devices being registered in exclusive control mode, the application can optionally specify, at registration time, what media encryption is preferred for the device's media stream.

The media encryption options are:

- Advanced Encryption Standard (AES)
- none (i.e. no encryption of the media stream)

Specify the set of encryption options your application supports and list them, in preferrential order, on the `LocalMediaInfo` parameters of the `RegisterTerminal` request. If you do not specify a set of encryption options in the `LocalMediaInfo` parameter, the connector server will default to "none" (no media encryption).

Note:

You will find information on call capacities your application can expect to be able to handle in the "Capacities for calls in Device, Media and Call Control applications*"* section of the "Capacities for types of applications" chapter of the *Avaya MultiVantage™ Application Enablement Services Overview.* This section includes information on how the choice of device control modes, media modes, codecs and encryption can affect the performance of your applications.

Sample registration code

If you are simply requesting that the device be exclusively controlled by the application and the media be handled by the connector server, then your registration code will look something like this:

```
// In the RegisterTerminal request, set only device and password;
// let LoginInfo.sharedControl default to FALSE and
// let LocalMediaInfo default to connector server handling.
RegisterTerminal request = new RegisterTerminal();
request.setDevice(id);
LoginInfo login = new LoginInfo ();
login.setPassword (password);
request.setLoginInfo (login);
RegisterTerminalResponse response = termSvcs.registerTerminal (request);
```

If you want the same as above, but you want shared control of the device, then set shared control to `true` in the LoginInfo bean:

```
login.setSharedControl (Boolean.TRUE);
```

If you setup the request for exclusive control and you want different codecs, then the above exclusive registration code may be prefaced with the setting of the codecs in `LocalMediaInfo` like this:

```
// Set the preferred codecs in LocalMediaInfo
MediaInfo localMediaInfo = new MediaInfo();
String [] codecs = new String [2];
codecs [0] = "g729"; // first choice
codecs [1] = "g729A"; // second choice
localMediaInfo.setCodecs (codecs);
request.setLocalMediaInfo(localMediaInfo);
```

Similarly, you may specify media encryption preference in `LocalMediaInfo` like this:

```
// Set the preferred media encryption in LocalMediaInfo
MediaInfo localMediaInfo = new MediaInfo();
String [] encryption = new String [2];
encryption [0] = "Audio.AES"; // first choice
encryption [1] = "Audio.NOENCRYPTION"; // second choice
localMediaInfo.setEncryptionList (encryption);
request.setLocalMediaInfo(localMediaInfo);
```

If you want to have the media stream terminate at the application or some other application-specified IP address, you must also insert code to set the RTP and RTCP addresses in `LocalMediaInfo`, like this:

```
// Specify the RTP address that will handle the media
InetAddress localAddress = InetAddress.getByName("pcname.yourcompany.com");
IPAddress rtpIPAddress = new IPAddress();
rtpIPAddress.setAddress(localAddress.getAddress());
rtpIPAddress.setPort(9000);
localMediaInfo.setRtpAddress (rtpIPAddress);

//Specify the RTCP address, too
IPAddress rtcpIPAddress = new IPAddress();
rtcpIPAddress.setAddress(localAddress.getAddress());
```

```
rtcpIPAddress.setPort(9001);
localMediaInfo.setRtcpAddress (rtcpIPAddress);
```

For Synchronous registration you send the registration request and wait, like this:

```
// Send the registration request and wait
RegisterTerminalResponse regTermResp=regSvcs.registerTerminal(regRequest);
if(regTermResp.getCode().equals(RegistrationConstants.NORMAL_REGISTER))
    System.out.println("Registration succeeded");
```

To use the Asynchronous Service to send the registration request:

```
// Use the Asynchronous Service to send the registration request.
// Callback will be notified when request succeeds or fails.
asyncSvcs.sendRequest(regRequest, callBack);
```

Here is an example of what an asynchronous callback implementation might look like:

```
private class MyAsyncRegistrationCallback implements AsynchronousCallback{

  public void handleResponse(Object response){
   if(response.getClass().getName().equals(RegisterTerminalResponse.class.getName())){
     RegisterTerminalResponse regResp = (RegisterTerminalResponse) response;
     if (regResp.getCode().equals(RegistrationConstants.NORMAL_REGISTER)){
       System.out.println("registered");
     }
     else{
       System.out.println("registration failed");
     }
   }
   else {
     System.out.println("This is bad. Asynchronous response is not of valid type: " +
       response);
   }
  }

  public void handleException(Throwable exception) {
   System.out.println("Received Asynchronous Exception from Server. The stack trace
     from the Server is as follows");
     exception.printStackTrace();
   System.exit(0);
  }
```

Telephony Logic

Listeners notify your application when specific telephony events occur:

- `RegistrationListener` events indicateunregistration by Communication Manager.
- `PhysicalDeviceListener` events indicate changes to the status of the ringer, display, and lamps.
- `MediaControlListener` events indicate when the media stream parameters have changed.
- `VoiceUnitListener` events indicate the status of recording and playing messages.
- `ToneDetectionListener` events indicate when a DTMF tone is received and when collection begins and ends.
- `ToneCollectionListener` events indicate when specified tone retrieval criteria are met.

Each event triggers a specific callback method in a listener. Listener implementations provide a way for your application to respond to each event.

WARNING:
All listener methods and all methods called from listener methods execute in the context of the client event thread. Therefore, your application will not be notified of other events until the previous event's callback method is complete.

Your logic for what to do with a device begins once it is registered; therefore the `registered` method in `RegistrationListener` is usually the starting point for your application's telephony logic.

Here is an example of what a simple `RegistrationListener` implementation might look like:

```
private class MyRegistrationListener extends RegistrationlAdapter {

    public void terminalUnregistered (TerminalUnregisteredEvent event) {
        try {// Make sure no listeners were left around and do cleanup
            removeListeners()
           cleanup()
       }
        catch (CstaException e){
          System.out.println("Exception when removing listeners or "
                              + "releasing device ID" + e)
       }
     }
}
```

Once a device is registered, your application may begin monitoring or controlling it with any of the services provided by the `ServiceProvider`.

As part of the registration initialization process, multiple physical device events are sent by the Communication Manager indicating the initial states of the lamps, ringer, hookswitch and display. These events may occur before and/or after the `Registered` event.

Monitoring and controlling physical elements

Physical elements of a device are monitored and controlled with the `PhysicalDeviceServices` interface found in the `ch.ecma.csta.physical` package. The set of supported services and events are reflected in Physical Device Services and Events on page 14, and in the Javadoc.

To use physical device services, you must first get `PhysicalDeviceServices` from the `ServiceProvider` as described in the Getting access to desired services on page 47. To monitor for physical device events, you must also implement `PhysicalDeviceListener` and add your listener to `MonitoringServices` as described in the Requesting notification of events on page 55.

Call control can be accomplished with a combination of:

- determining the current status of physical elements on a device, such as requesting the list of buttons administered for the device
- listening for particular physical device events, such as when the phone starts ringing
- activating physical elements of the device, such as going offhook

Knowing what buttons are administered

If your application needs to press any buttons or determine which lamps have changed state, you will need to know what buttons are administered on the device. Buttons are assigned to devices during station administration via the Communication Manager system access terminal (SAT) interface. Your application must use the `getButtonInformation` request in order to the get the list of buttons administered for a device. Each button item in the list includes the following information:

- *button identifier* - indicates the address or location of the button. Its value is one of the constants listed in `ButtonIDConstants` in the `com.avaya.csta.physical` package. Constants are available for both administered buttons and fixed buttons (those buttons that are preset and pre-labeled on a telephone set)
- *button function* - indicates what the button does when pressed. Its value is one of the constants listed in `ButtonFunctionConstants` in the `com.avaya.csta.physical` package .
- *associated extension* - indicates whether there is an extension number associated with this button and what the extension number is
- *associated lamp* - indicates whether there is a lamp associated with the button. If there is, its lamp identifier is the same as the button identifier

Here is an example of how your application might find the call appearance buttons and lamps:

```
// Get the information about all buttons administered on this device.
GetButtonInformation buttonRequest = new GetButtonInformation();
buttonRequest.setDevice(id);
GetButtonInformationResponse buttonResponse =
physSvcs.getButtonInformation(buttonRequest);
ButtonList list = buttonResponse.getButtonList();
ButtonItem[] buttons = list.getButtonItem();

//Loop through each button in the list looking for the call appearances
List callAppearanceIDs = new ArrayList();
for (int i = 0; i < buttons.length; i++) {
     if (ButtonFunctionConstants.CALL_APPR.equals(
             buttons[i].getButtonFunction())) {
        System.out.println("Button: [ID: " + buttons[i].getButton() +
             "] is a call appearance button.");

        // Keep track of this button ID so we know which lamps
        // correspond to call appearances
        callAppearanceIds.add(buttons[i].getButton());
     }
}
```

Note:

There is no direct indication provided by the API to the application of changes to the provisioned information for a monitored device. Thus collecting the device's configuration when the applicaiton initializes is an incomplete solution. A robust application should periodically validate that the current configuration of the device is aligned with the representation of that configuration as derived by the application. In order to do so, an audit of the information should be developed and run at some recurring cycle, or when unexpected feedback to button presses is received. An audit can be realized by utilizing the `getButtonInformation` request and comparing the results with the previously obtained information.

Detecting an incoming call

When a call comes into a device, these three changes occur to the physical device:

- The phone rings.
- A green call appearance lamp flashes.
- The display changes to show caller information.

Therefore, the following events will be sent to an application that has added a `PhysicalDeviceListener` that listens for these events:

- `RingerStatusEvent` - The ring pattern is supplied in the event structure. For a list of possible ring patterns, see the Javadoc for `RingerPatternConstants` in the `com.avaya.csta.physical` package .

- `LampModeEvent` - The identifier of the lamp that has changed and the lamp's mode is supplied in the event structure. Once you know where the call appearance lamps are (see Knowing what buttons are administered on page 69), you can determine if it is a call appearance lamp and if it is flashing by comparing the lamp mode against the `FLASH` constant. For a complete list of the possible lamp modes, see the Javadoc for `LampModeConstants` in the `com.avaya.csta.physical` package.
- `DisplayUpdatedEvent` - The display contents are supplied in the event structure.

You may want your application to key off of just the `LampModeEvent`, or it may want to wait for the other events before responding to an incoming call. The events may come in any order.

Determining that far end has ended the call

If all far-end parties drop on a call, these changes occur on the local device:

- The call appearance green lamp turns off.
- The display is updated based on the current state of the extension. For example:
  - Returns to an idle state showing extension, date and time information
  - Begins showing information about a ringing call at the extension
  - Is not updated and continues to show information relative to an active display feature such as Directory Lookup

Therefore the following events will be sent to an application that has added a PhysicalDeviceListener that listens for these events:

- `LampModeEvent` - The identifier of the lamp that has changed and the lamp's mode is supplied in the event structure. Once you know where the call appearance lamps are (see Knowing what buttons are administered on page 69), you can determine if it is a call appearance lamp and if the lamp is now off by comparing the green lamp mode against the `OFF` constant.

  Note:

  Communication Manager sends lamp updates not only for lamp transitions, but also to refresh lamps. Therefore, some `LampModeEvents` indicate that the lamp is in the same state it was in before the event.

- `DisplayUpdatedEvent` - The display contents are supplied in the event structure.

Making a call

To make a call from a telephone, a person would typically:

1. Go offhook
2. Wait 500 msec
3. Press a sequence of dial pad buttons (0-9, *, #) to initiate a call, such as pressing 5551234, with a 100 msec delay in between each digit

4. Listen for an answer
5. Begin a two-way conversation or listen to a recording

Here is how you might program each of those steps:

1. To go offhook, you could simply use the `setHookswitchStatus` request. However, this approach could cause a conflict with a potential incoming call. That is, if a call came in just before you went offhook, then your dialing attempt would fail and instead you would be connected with the incoming caller.

   The only way to avoid conflicting with an incoming call is to keep track of the lamp transitions of the call appearances. If a lamp goes from off to steady, then you can make an outbound call. But if the lamp goes from off to flashing and then to steady, then you have just picked up an incoming call.

   One method to reduce the chance of conflicting with an incoming call is to begin a call by pressing the *last* call appearance button using the `pressButton` request. This avoids using the same call appearance as an incoming call which comes in to the first available call appearance. To further assure that even the last call appearance is not in use, make sure the lamp is off before you press the button. Observing the sequence of lamp transitions is still necessary even with these precautions.

2. To press the dial pad buttons to dial a number, use the `pressButton` request and the constants defined for dial pad buttons in `ButtonIDConstants`, such as `DIAL_PAD_7` or `DIAL_PAD_POUND`.

   Tip:

   DIAL_PAD_0 through DIAL_PAD_9 have string values of “0” through “9”, therefore using the strings “0” - “9” will work. However, DIAL_PAD_STAR and DIAL_PAD_POUND are *not* set to “*” and “#”; instead they are “10” and “11”, respectively (as specified by CSTA). The `pressButton` request will *not* work with “*” and “#”. Therefore, it is safer to get in the habit of using the constants, not the literals.

3. If your application is handling its own media (as determined by the local RTP address and exclusive control mode set at registration) then you can determine from the media stream that there is media other than ringback coming from the far end. Your application will need an RTP stack and a call progress tone detector. You may use a third party vendor stack, your own stack or the media stack provided by Avaya. Avaya’s media stack is described in the Media Stack API Javadoc. Avaya does not provide a call progress tone detector.

   If the connector server is handling the media, wait for an appropriate period of time before playing a message to the far end or recording the far end’s media stream. This is to allow time for the RTP connection to be made end-to-end.

4. To have a real-time conversation in server media mode, the application must handle the media.

   To play a message to the far end or record the far end’s media stream, use `VoiceUnitServices` (see Recording and playing voice media below for more details).

For an example of how to make a call while taking the precaution of avoiding conflict with an incoming call, see the `Station` class sample source code. In particular, look at the `initCall` request that does not specify a call appearance identifier.

Getting ANI information for a call

There are two ways to get ANI information for a call.

- Using Snapshot Services.

  The simplest way is to use Snapshot Services.

  In order to use Snapshot Services to get ANI information for the call, two service requests will have to be made. First, issue a `SnapshotDevice` request for the device to get a connection identifier for the call. If the response contains multiple calls, find the one that is in the active state. Next, issue a `SnapshotCall` request on the connection identifier retrieved from the `SnapshotDeviceResponse`.

  Note:

  Snapshot Services uses TSAPI and therefore requires a TSAPI license and the appropriate TSAPI configuration.

- Using the conference display button

  If the application wishes to use only device and media control, it can alternatively use the conference display button.

  In order to use the conference display button to get ANI information for the call, the conference display button (conf-dsp) will have to have been administered for that extension in Communication Manager.

  Once an incoming call is received by your application for the extension, your application should press the conference display button of the extension repeatedly to get the ANI information (through the `DisplayUpdatedEvent` of Physical Device Services) for each party on the call.

Recording and playing voice media

If your application needs to record incoming media or play a message on a call, then at registration time:

- You must request exclusive control of the device (in shared control, media goes only to the telephone).
- You can choose to handle the media yourself (client media) or have the AE Services server do it for you (server media). See Choosing a media mode on page 63.

This section describes how to use Voice Unit Services to have the AE Services server record and play media for you. The supported services are described in the Voice Unit Services and Events on page 15. The details about using the services are described in the Javadoc. To see sample code using `VoiceUnitServices`, see the sample application called TutorialApp referenced in Learning from sample code on page 39.

Some basic rules of Voice Unit Services are:

- **Wave files**

  All digital audio files that are created or played using `VoiceUnitServices` are in the Wave Resource Interchange File Format (RIFF). The standard Wave file structure is used for all encoded media types. See http://www.sonicspot.com/guide/wavefiles.html for a description of the Wave structure.

  G.729 formatted files, however, use non-standard field values in some of the standard format chunk fields. They are:

  - Compression code value: 131 (0x0083)
  - Block align value: 10 (0x000A)
  - Bits per sample value: 1 (0x0001)

  An external G.729 converter is required to convert a G.729 Wave file into a standard RIFF Wave file that can be played.

- **Files on connector server**

  All Wave files are assumed to be on the AE Services server machine in the directories specified in the OAM interface under the media properties as the player directory and the recorder directory.

  Note:

  These two directories do not have to be the same. These directories are the root directory of the relative paths specified in the `playMessage` and `recordMessage` requests.

- **Encoding algorithms**

  Files to be played can be encoded in these formats:

  - PCM 8-bit or 16-bit
  - G.711 A-law
  - G.711 Mu-law
  - G.729
  - G.729A

  Recordings can be made of calls in these formats

  - PCM 8-bit or 16-bit
  - G.711 A-law

- G.711 Mu-law
- G.729

Note:

While in server mode, you can find out what codecs have been chosen from the `MediaStartedEvent`, which will contain a list of the codecs.

Note:

Codecs must be specified at registration time and while in server media mode you cannot specify a mixture of G.711 and G.729 codecs for a single device. This is because there is no conversion offered by the server.

- **Conversions between encoding algorithms**

  Files to be played can be converted from any PCM type to any G.711. No other conversions are supported for playing.

  Messages to be recorded can be converted from G.711 to PCM. No other conversions are supported for recording.

- **Using with Tone Detection or Tone Collection Services**

  The Voice Unit Services player and recorder may be setup to detect DTMF tones at the same time Tone Detection or Tone Collection Services is being used. However, there is no guarantee which service will detect a tone first. See Possible race conditions on page 106 for more specifics.

Recording

To record the RTP media stream of a device, use the `VoiceUnitServices.recordMessage` method. This will record only the media coming from other parties on the call to the device not the media that is being played from this device using the `VoiceUnitServices.playMessage` method. Only media packets that are received are recorded: lost packets are not replaced.

The application can specify an alphanumeric filename for the recording or let the filename default to a format of <*timestamp*><*extension*>.wav. The alphanumeric filename may contain a relative directory path. Filenames specified for recorded files must be relative to the configured directory, their directories must already exist, and recordings cannot overwrite an existing file. If it is defaulted, then the resulting filename is returned in the `RecordMessageResponse`.

Since recording is associated with a device rather than a call, a recording could contain the incoming media from multiple calls. Recording continues until one of the following occurs:

- Application explicitly stops the recording with a `VoiceUnitServices.stop` method (stops both playing and recording) or with an `ExtendedVoiceUnitServices.stopRecording` method.

- Application requests that the connector server automatically stop the recording when a specified termination criterion is met. Multiple termination criteria can be specified in which case the first criterion that is met stops the recording. Termination criteria options include:
  - when a DTMF tone is received by the device

    To request this termination criterion, set the `termination` parameter's terminating conditions so that `DTMFDigitDetected` is set to true.
  - when recording reaches a specified duration

    To request this termination criterion, set the `maxDuration` parameter to the maximum number of milliseconds allowed for the recording.

If you wish to record one entire call and only one call, then your application can watch the lamp events to determine when the call has ended and explicitly stop the recording after the call has ended.

Note:

No exception is thrown if the application requests that recording stop when there is no active recording.

Once an active recording on a device has been stopped, a `StopEvent` in the Voice Unit Services listener indicates that the recording has finished and the recorded file is ready for the application.

The recording can also be:

- suspended temporarily - with `VoiceUnitServices.suspend` method (suspends both recording and playing) or with `ExtendedVoiceUnitServices.suspendRecording` method.
- dubbed with another recording - with `ExtendedVoiceUnitServices.startDubbing` and `stopDubbing` methods. See next section for more information on dubbing.
- resumed - with `VoiceUnitServices.resume` method (resumes both recording and playing) or with `ExtendedVoiceUnitServices.resumeRecording` method.
- stopped - with `VoiceUnitServices.stop` method (stops both recording and playing) or the `ExtendedVoiceUnitServices.stopRecording` method.

Dubbing

A recording can be dubbed with another Wave file by using `ExtendedVoiceUnitServices`' `startDubbing` and `stopDubbing` methods. Dubbing records the specified Wave file over the recording repeatedly from the time `startDubbing()` is called until `stopDubbing()` is called. This may be helpful to avoid recording sensitive information such as a spoken password or other private or security-based information.

Since the application must explicitly stop the dubbing, the application must have the logic to know when to stop. It may be based on time, an incoming DTMF tone such as "#", or a manual action by an agent who is listening.

Playing

To play one or more messages to the RTP stream of a call as if the message(s) are coming from the device, use the `VoiceUnitServices.playMessage` method. The message(s) can be played once, multiple times, for a particular duration, or until a DTMF tone is received by the device.

The filename of the file to be played can be alphanumeric. The alphanumeric filename may contain a relative directory path. One or more files can be specified as long as they are of the same encoding type. For an example of how multiple files can be specified, see the Simple IVR application that is provided with this SDK.

Since playing is associated with a device rather than a call, the playing of the message(s) may continue across multiple calls to which the device is a party. Playing continues until one of the following occurs:

- Application explicitly stops the playing with a `VoiceUnitServices.stop` method (stops both playing and recording) or with an `ExtendedVoiceUnitServices.stopPlaying` method.
- A specified termination criterion is met.

  Multiple termination criteria can be specified, in which case the first criterion that is met stops the playing. Termination criteria options include:

  - When a DTMF tone is received by the device.

    To request this termination criterion, set the `termination` parameter's terminating conditions so that `DTMFDigitDetected` is set to true.

  - When playing occurs for a specified duration.

    To request this termination criterion, set the `duration` parameter to the maximum number of milliseconds allowed for the playing.

  - When the message(s) have been repeated a specified number of times with a specified interval in between.

    To request this termination criterion, set `playCount` and `playInterval` in the `extensions` parameter.

If you wish to play message(s) to one entire call and only one call, then your application can watch the lamp events to determine when the call has ended and explicitly stop the playing after the call has ended.

Once active playing on a device has stopped, a `StopEvent` in the Voice Unit Services listener indicates that the playing has finished.

Note:

No exception is thrown and no event is generated if the application requests that playing stop when there is no active message playing.

The playing of the message can also be:

- suspended temporarily- with the `VoiceUnitServices.suspend` method (suspends both recording and playing) or the `ExtendedVoiceUnitServices.suspendPlaying` method.
- resumed - with the `VoiceUnitServices.resume` method (resumes both recording and playing) or the `ExtendedVoiceUnitServices.resumePlaying` method.
- stopped - with the `VoiceUnitServices.stop` method (stops both recording and playing) or the `ExtendedVoiceUnitServices.stopPlaying` method.

Monitoring Voice Unit Events

Your application can receive and respond to Voice Unit events by adding a `VoiceUnitListener`. The events indicate when your `VoiceUnitService` requests have been accepted and processing has begun, and when processing has ended.

Detecting and collecting DTMF tones

If your application needs to detect DTMF tones coming to a device from another party on the call, then you can use `ToneDetectionServices` or `ToneCollectionServices`. To use these services, you must do both of the following:

- Register the device in exclusive control mode (the media goes only to the telephone in shared mode).
- Register the device in server media mode so that the connector server can control the media.

DTMF tones are generated by parties on a call by pressing the dial pad digits 0 through 9 and * and # during the call. If the device that is being monitored is on a call and another party on the call presses a dial pad digit, then `ToneDetectionServices` can be used to report each DTMF tone to the application.

In contrast, `ToneCollectionServices` can be used to buffer the received tones and report them to the application when the specified retrieval criteria are met. The retrieval criteria may be one or more of the following:

- A specified number of tones has been detected.
- A specified tone (called a "flush character") has been detected.
- A specified amount of time (called a "timeout interval") has elapsed.

When at least one of the retrieval criteria is met, the following retrieval steps are performed by the AE Services server:

1. The buffered tones, up to and including the tone which met the retrieval criteria, are removed from the buffer.

2. The retrieval criteria are cleared (optional).
3. The application is notified of the retrieved tones with the `TonesRetrievedEvent`.

If more than one retrieval criterion is specified, the first one to occur causes the retrieval criteria to be met.

The supported services are summarized in Terminal Services and Events on page 27 and Tone Collection Services and Events on page 27. The details about using the services are described in the Javadoc.

Some basic rules of these services are:

- **Touch tone detection mode**

  Both sets of services can detect in-band and out-of-band DTMF tones. In-band tones are transmitted within the media stream. Out-of-band tones are transmitted in the signalling channel. The AE Services server always detects out-of-band tones. If the application wishes to also detect in-band tones (in-band tone detection is not recommended), then the tone detection mode must be explicitly set to in-band.

  Set the tone detection mode before the AE Services server is started up. The mode is provisioned in the AE Services server OAM interface under the media properties as the ttd_mode.

  To detect only out-of-band, set the mode to `OUT_BAND`. To detect both in-band and out-of-band, set to `IN_BAND`. See the see the appropriate *Avaya MultiVantage™ Application Enablement Services Installation Guide* for the offer you have purchased (bundled server or software only) for instructions on how to choose between in-band and out-of-band for the connector server and for Communication Manager, and how to setup the `ttd_mode` property.

- **Using with Voice Unit Services tone detection**

  The Voice Unit Services player and recorder may be set up to detect DTMF tones at the same time Tone Detection or Tone Collection Services is being used. However, there is no guarantee which service will detect a tone first. See Possible race conditions on page 106 for more specifics.

Detecting individual tones

To detect DTMF tones one at a time, use the `ToneDetectionServices` methods in this order:

1. Implement the `toneDetected` method in `ToneDetectionListener` to handle each tone that is detected. More than one `ToneDetectionListener` implementation can be created, if needed.
2. Add the listener(s) to the device with the `MonitoringServices.addToneDetectionListener` method.

3. Now each time a tone is detected by the connector server, the application's callback method, `ToneDetectionListener.toneDetected`, is called. If more than one listener was added, all of their callback methods are called.
4. When you no longer wish to be notified of detected tones, remove the listener with the `MonitoringServices.removeToneDetectionListener` method.

You may want to set up interdigit timers limiting the maximum amount of time the application will wait between tones or a duration timer limiting the maximum amount of time before stopping the tone detection.

Collecting multiple tones

To have the AE Services server collect multiple DTMF tones and report them to the application based on specified retrieval criteria, use the `ToneCollectionServices` methods in this order:

1. Implement the `tonesRetrieved` method in `ToneCollectionListener` to handle a retrieved set of tones. The `TonesRetrievedEvent` parameter that is passed to the `tonesRetrieved` callback method supplies the set of tones that were retrieved and the cause of the event. The cause of the event may be any of the following:
   - BUFFERFLUSHED - when the `flushBuffer` method is called
   - CHARCOUNTRECEIVED - when the number of tones specified in the retrieval criteria is received
   - FLUSHCHARRECEIVED - when the tone specified in the retrieval criteria is received
   - TIMEOUT - when the amount of time specified in the retrieval criteria has elapsed
2. Add the listener to the device with the `MonitoringServices.addToneCollectionListener` method.

   Note:

   Adding the listener does *not* start the buffering of tones.
3. Start tone collection with the `ToneCollectionServices.startToneCollection` method. This causes each detected tone to be put in a buffer.
4. Set the retrieval criteria with the `ToneCollectionServices.setRetrievalCriteria` method.
5. When one of the retrieval criteria is met, the application's callback method, `ToneCollectionListener.tonesRetrieved`, is called and optionally, the retrieval criteria are cleared. The connector server continues buffering detected tones.
6. If you wish to be notified of more tones, call the `ToneCollectionServices.setRetrievalCriteria` method again. There is no need to stop and restart the collection.
7. You can add and remove listeners at any time during collection.

8. If for any reason you wish to flush the buffer during collection, use the `ToneCollectionServices.flushBuffer` method. This will report the tones collected since the last time the buffer was flushed in the `TonesRetrievedEvent`. You may want to flush the buffer at the end of a call.
9. When you no longer wish to be notified of collected tones, stop tone collection with the `ToneCollectionServices.stopToneCollection` method and remove the listener with the `MonitoringServices.removeToneCollectionListener` method.

To see sample code using `ToneCollectionServices`, see the sample application called SimpleIVR referenced in Learning from sample code on page 39.

Determining when far-end RTP media parameters change

Applications that control their own media (client media mode) will need to use the `MediaControlListener` to determine when the far-end RTP media parameters change. Here is the sequence of media control events an application should be prepared to receive:

1. When media is first established for a call, the application receives a `MediaStartEvent`. However, it does not guarantee that the call has been established end-to-end yet.
2. If Communication Manager changes the far-end RTP parameters for a call, the application receives a `MediaStopEvent`. At that point the current far-end RTP parameters should no longer be used. A `MediaStopEvent` could indicate that the call has ended, but do not depend on that event alone to determine the end of a call. The call appearance lamp will also change if it is the end of a call.
3. If there are new far-end RTP parameters, then the application will subsequently receive a `MediaStartEvent`.

One scenario in which a `MediaStopEvent` and then a `MediaStartEvent` may be received is when Communication Manager *shuffles* a call. Shuffling occurs when Communication Manager changes the path of the media. For example, if the media is going from calling party A to Communication Manager and then to called party B, Communication Manager may choose to change the media path such that it goes directly between endpoints A and B. In this case, Communication Manager would tell both A and B to stop using the Communication Manager address as the far-end address and to start using the other endpoint as the far-end address. In other words, A would be notified that B is now the far end and B would be notified that A is the far end. Later Communication Manager may choose to change the path again due to some change in the call, such as a conference or transfer. Each time the path changes, the endpoints are notified via media control events to stop using the current far-end address and to start using the new far-end address.

In server media mode all of this is usually transparent to the user, unless the codec happens to change. In this case you may need to play a different wav file in the codec of the new form. For this reason it is recommended that you give a single codec when registering devices.

Recovery

The application session protocol that was implemented for the 3.0 release has several recovery benefits for the Device, Media and Call Control application. The keepalive messages that are sent periodically by the API to the AE Services server provide a way for the application to verify that the AE Services server is operational even if there is no other activity from the application. In addition, the session cleanup delay provides a mechanism for the application to reestablish its session after a short network interruption without having to reestablish their state (e.g. register devices again). This section tells you how to take advantage of these new features to design a more robust Device, Media and Call Control application.

The best way to monitor for problems with a session is to add a `ServiceProviderListener` to the application's `ServiceProvider` instance. The following is a list of events the application can receive via this listener, and the proper recovery actions to take in the case of each event.

- `ServerConnectionDownEvent`. This event is sent if the socket to the AE Services server has gone down for some reason. The recovery action to take for this event is to call `reconnect` on the application's `ServiceProvider` instance. This method will attempt to open a new socket to the AE Services server, and to reestablish the existing session with a new socket.

  If successful, the application should be able to resume operations without reestablishing its state although your application may have missed some events. To be sure that you know the current state of the lamps, hookswitch and display, you will have to query for those states using `getLampState`, `getHookswitch` and `getDisplay` to update the states.

  The operation may fail if the network is down, or if the session has been terminated on the AE Services server because the cleanup timer expired. If a `SessionCleanedUp` exception is received on the call to `reconnect`, it should be processed as a `ServerSessionTerminated` event would be. If any other exception is received, the application should set a timer and try to reconnect again at some later time. Any requests that the application was trying to send that timed out in the interval will generate exceptions.

  Note:

  Although the Click2Call sample application show session recovery, for simplicity, it does not attempt to automatically reconnect if an exception is thrown during `getServiceProvider`.

- `ServerSessionNotActiveEvent`. This event is sent if a message was received by the AE Services server but the session has timed out and been placed in the inactive state. A session enters the inactive state if the Application Session Duration expires before a `ResetApplicationSessionTImer` message is received. Upon receiving this event, an application must first call `disconnect` on the application's `ServiceProvider` instance, then take the same recovery actions as for `ServerConnectionDownEvent`.

  Note:

  Sending `ResetApplicationSessionTimer` messages is handled automatically by the API for Java applications.

- `ServerSessionTerminatedEvent`. This event is sent if a message was received by the AE Services server but the session cleanup timer expired after the session entered the inactive state. The cleanup timer is 0 by default for backwards compatibility. If an application wants to have an opportunity to recover from a short network outage without reestablishing state, it should ensure that it sets the `cmapi.session_cleanup_delay` property. If this event is received, it is impossible to recover the previous session, and a new session must be established. In order to do this, a new `ServiceProvider` instance must be instantiated by calling `getServiceProvider`. In this case do not set the `cmapi.session_id` property, because a new session is desired. Handles to services and DeviceIDs must be reacquired, and stations must be re-registered.

  Note:

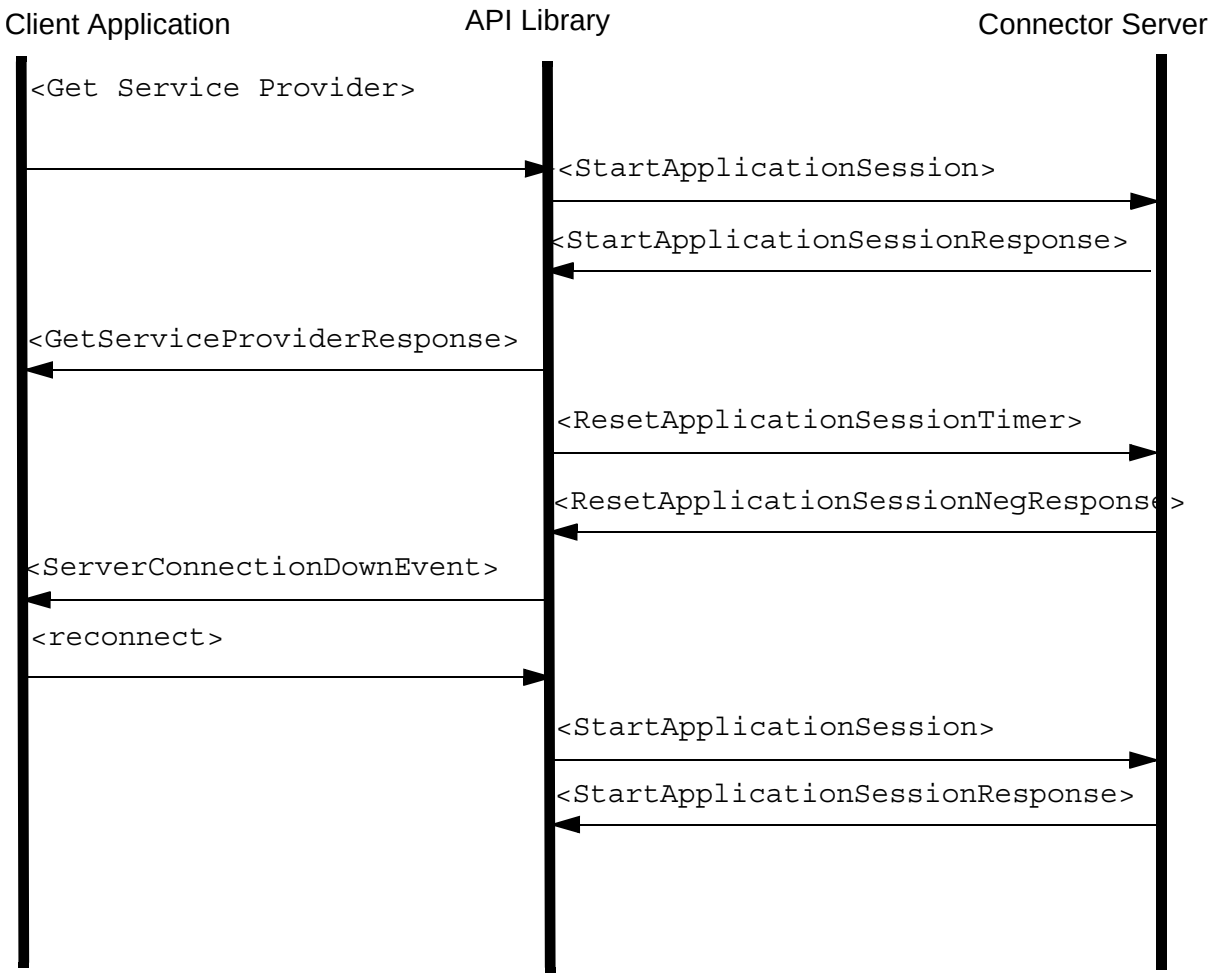
  The Click2Call sample application shows how to build session recovery into an application.

  Note:

  If a `ServerConnectionDownEvent` or `ServerSessionTerminatedEvent` is received as a result of the application sending a request to the AE Services server (as opposed to a `ResetApplicationSessionTimer` message automatically being sent) the application will receive an exception in addition to the event. Recovery actions should be initiated upon receiving the event, but not upon receiving the exception. This is to enable the application to keep the calls to its recovery code in one place, rather than sprinkling it throughout the rest of the code every time a request is issued.

The following figure shows the sequence of events that occurs between the client application, the API library and the connector server during establishment of an application session, and an instance of a `ServerConnectionDownEvent` occurring during subsequent session management.

Figure 2: Session Management Recovery



Cleanup

If the cleanup session expires resources are reclaimed. It is important to free resources when they are no longer needed. This is most likely to occur when your application:

- detects the end of a call
- is finished with a device
- is about to exit

Cleanup should occur in this order:

1. **Stop collecting tones**

   At the end of a call, you can choose to stop collecting DTMF tones for the device. Alternatively, you can let the collection and the retrieval criteria continue across calls. In that case you might just flush the buffer at the end of each call.

   When finished with a device, stop the tone collection for that device.

   When the application is about to exit, stop tone collection on all devices.

2. **Stop recording or playing**

   At the end of a call, you can choose to stop recording or playing or let the recording or playing continue across calls on the device.

   When finished with a device, stop both recording and playing on that device.

   When the application is about to exit, stop both recording and playing on all registered devices.

   Both recording and playing can be stopped on a device using `VoiceUnitServices.stop` method or can be individually stopped using `ExtendedVoiceUnitServices.stopRecording` method and `ExtendedVoiceUnitServices.stopPlaying`

3. **Unregister the device**

   When finished with a device, unregister it using the `RegistrationServices.unregisterTerminal` method.

   When the application is about to exit, unregister each registered device.

4. **Remove the listeners**

   When your application no longer needs to receive events for a device, remove the listeners that were added. For example if a `PhysicalDeviceListener` was added with the `MonitoringServices.addPhysicalDeviceListener` method, then use the `MonitoringServices.removePhysicalDeviceListener` method to remove it.

5. **Release the device identifier**

   When finished with a device identifier, release it using the `DeviceServices.releaseDeviceID` method.

   When the application is about to exit, release each device identifier.

6. **Disconnect the socket**

   Cleanup the application session and disconnect the socket by calling the `disconnect` method on the application's `ServiceProvider` instance.

Media Encryption

Application Enablement Services release 3.1 offers the user the ability to encrypt the voice RTP streams between the CMAPI softphone and the far end of the call.

For Application Enablement Services release 3.1, the only encryption scheme available is the Advanced Encryption Standard (AES) media encryption.

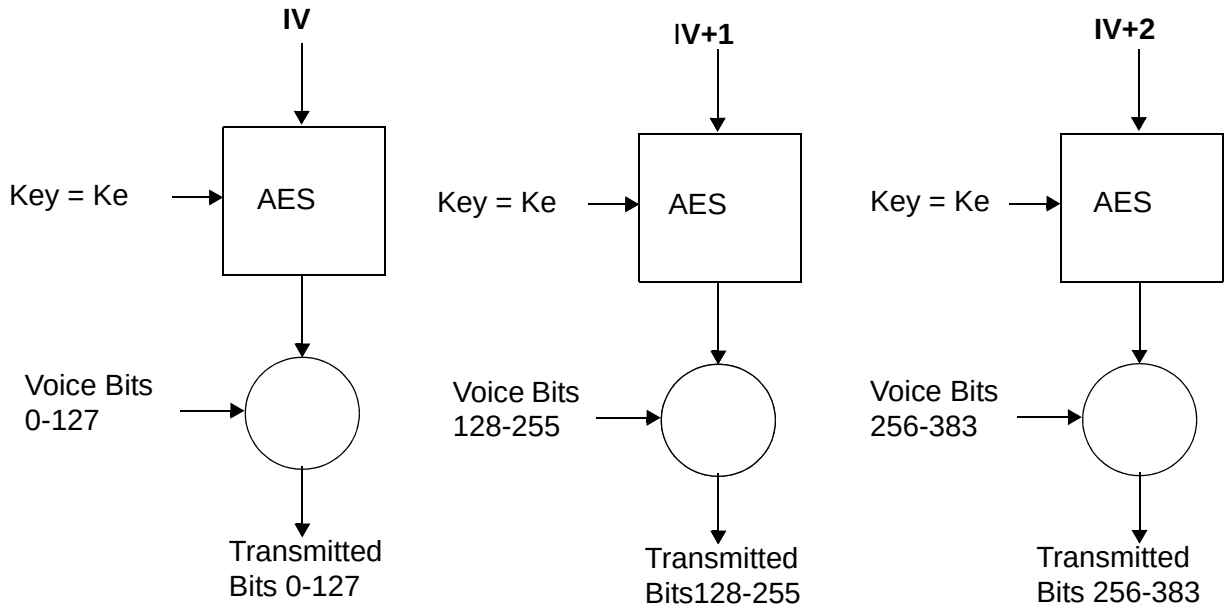
The AES Encryption Scheme

1. Encrypting the Voice Stream

   To transmit voice information over a digital medium, the analog voice signal is sampled at discrete intervals. Each sample is represented as an 8-bit number or 8-bit byte. Samples (bytes) are transmitted sequentially as a stream of bits. If a G.711 codec is used to generate the samples and if 20 milliseconds worth of data is sent in each Internet Protocol (IP) packet, then each packet will contain 160 bytes which equals 1280 bits. (8000 samples/ second for 20 milliseconds).

   To send the voice data, the stream of voice bits is exclusive OR'd with a second stream of bits before being transmitted. This second stream of bits is generated cryptographically and the resulting transmitted stream is said to be "encrypted". The two bit streams are processed in fixed "chunks" of 128 bits as illustrated in Figure 3

Figure 3: Encryption of the Voice Stream



A 128 bit initialization vector (IV) is encrypted with key KE (128 bits) using the AES encryption algorithm to produce 128 bits of output. Those 128 bits are exclusive OR'd with the first 128 bits of the voice packet. The initialization vector is incremented by one, and the process repeated for the next 128 bits. Ten repetitions are required to send one 20 millisecond packet which contains 1280 bits of voice data. This means that the AES encryption engine is run 10 times to send one packet. This mode of using AES is called counter mode (because the IV acts as a counter).

On the receiving end, the same process is used to recover the original voice data. The receiver must have the exact same key (KE) and the same initialization vector (IV).

2. Generating Key Material

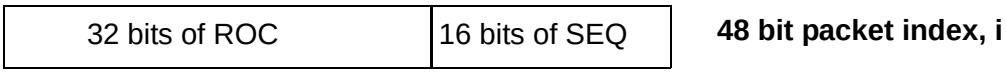
   In order to generate the encryption key, initialization vector, and other keys to be seen shortly, the following operations are performed. Note that these computations are performed anew for each RTP packet.

   Let the packet index "i" be defined as:

   i = (32-bit ROC) || (SEQ for RTP)

   where ROC is the roll over counter, SEQ is the 16-bit sequence number from the RTP packet and || indicates concatenation. This is shown in Figure 4.

Figure 4: Structure of the Packet Index



Let

r = i DIV key_derivation_rate

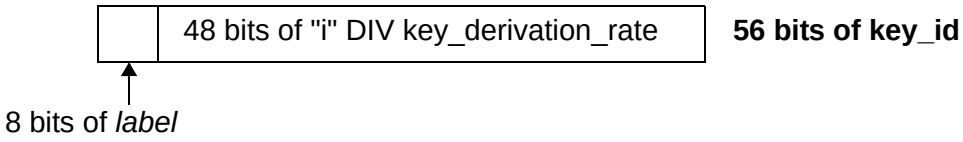
where DIV denotes integer division rounded down with the convention that dividing by 0 equals 0. The SRTP algorithm supports changing the keys periodically, even while the voice stream is active. The key derivation rate is the rate of this change. A value of zero indicates that the keys are not changed periodically. When the rate is zero, “r” is also zero (48 bits). Note that in the first computation of “r”, the value of SEQ used in the computation of “i” is the initial value at the beginning of the media stream.

Let

key_id = <label> || r

where <label> = 0x00 for RTP packet encryption and 0x02 for the salting key used to generate the IV as illustrated in Figure 5.

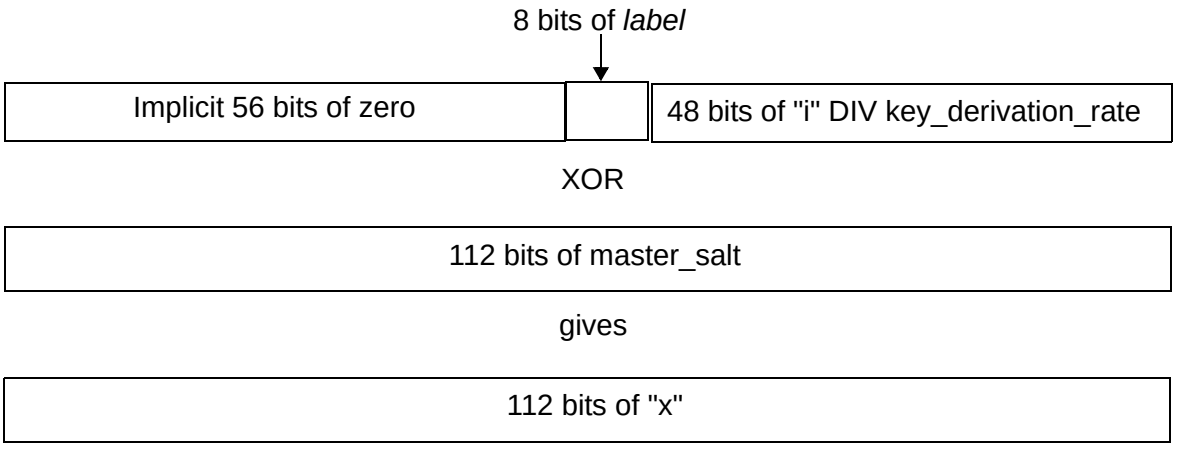
Figure 5: key_id Structure



Let `x = key_id XOR master_salt`

This is shown in Figure 6:

Figure 6: Computation of '"x"

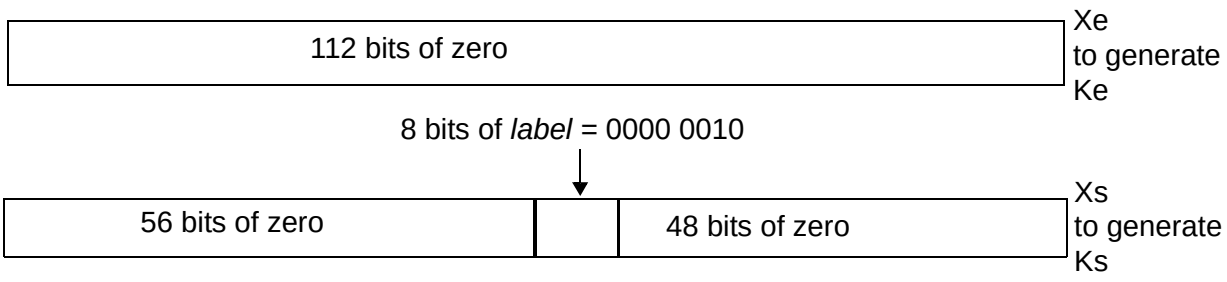


The keys for encryption and IV generation are the result of encryption of ("x" * 2^{16}) with a key called the master key. (The master key is distributed by the Media Gateway Controller for each voice IP media link as the link is established.)

The values of "x" for Avaya's implementation are shown in figure 6. Note that two values of "x" are computed, one (xe) is used to compute the value of KE and a second value of "x" (xs) is used in the computation of the IV.

For Avaya's implementation: `Key_derivation_rate = 0 Master_salt = 0`

Figure 7: Values of "x" for Avaya's Implementation



3. Creating the Initialization Vector

   The Initialization Vector (IV) changes for each packet (1280 voice bits for 20ms G.711) according to the following equation: $IV = (SSRC * 2^{64}) \text{ XOR } (KS * 2^{16}) \text{ XOR } (i * 2^{16})$

   where KS is known as the session salting key and SSRC is the synchronization source from the RTP packet currently being encrypted. Note that “i” contains the packet sequence number SEQ and therefore **the IV must be recalculated for each RTP packet**.

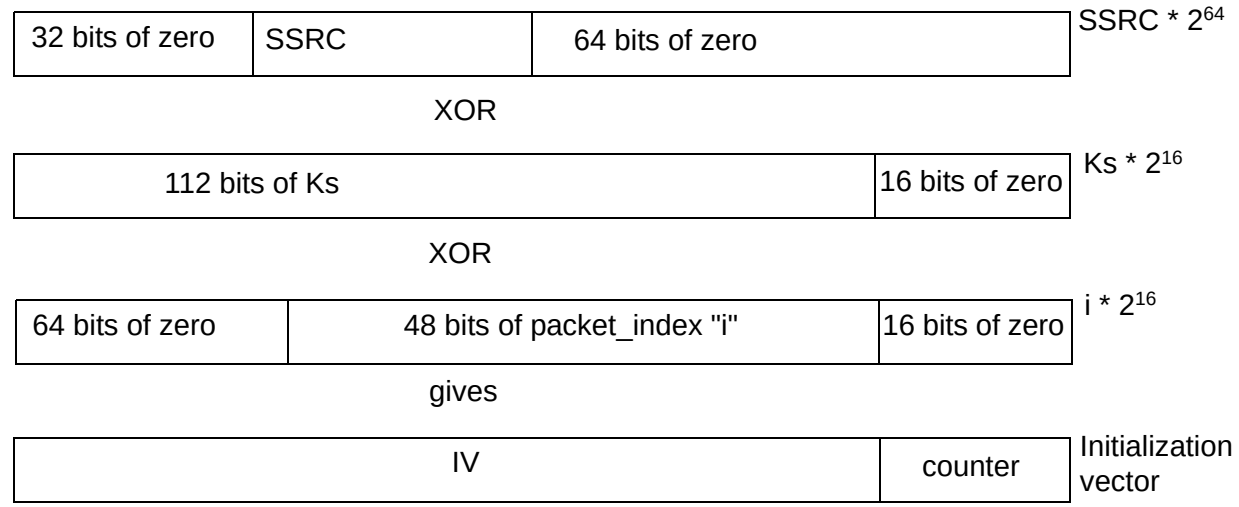
   The 112 bit KS is computed from xs using the pseudo random function (PRF) as follows.

   KS = PRF_112 (master_key, xs * 2^{16})

   The pseudo random function is defined to be AES in counter mode with its output stream truncated as necessary (left most bits are retained).

   The process for IV generation is shown in Figure 8.

Figure 8: 128-bit IV Generation



4. Creating the Encryption Key KE

   The process for generating the session key (KE) uses the pseudo random function (AES in counter mode) to produce a 128 bit value as follows: $KE = PRF\_128 \text{ (master\_key, } xE * 2^{16})$

Specifying the Devices' Encryption Capability

You may control whether your CMAPI softphone will support media encryption or not by specifying the supported encryption types in the local `MediaInfo` structure:

```
// specify either AES or no encryption for this device (let CM choose)

String [] encryption = {MediaConstants.AES, MediaConstants.NOENCRYPTION};

MediaInfo localMediaInfo = new MediaInfo();

localMediaInfo.setEncryptionList(encryption);

station.register (password, false, localMediaInfo, new MyAsyncRegistrationCallback());
```

This specifies that the CMAPI softphone will support both AES encryption and no media encryption. In this case, the decision to encrypt the media stream is left up to the Communication Manager (as specified in the CM's "change ip-codec" form).

Alternatively, you may force AES media encryption to be chosen by specifying a supported encryption type of only AES:

```
// must use AES encryption

String [] encryption = {MediaConstants.AES};
```

and, of course, you may force no encryption to be chosen by specifying:

```
// encryption not supported

String [] encryption = {MediaConstants.NOENCRYPTION};
```

MediaStartEvent Handling

If you have chosen to receive and handle the media stream as part of your application (you have chosen client-media mode), you will receive the media encryption information in the `MediaStartEvent`. In addition to the usual "RTP address", "RTP port", "codec" etc., the `MediaStartEvent` will also contain an "Encryption" object containing the encryption protocol and keys chosen by Communication Manager.

When you start to process the RTP stream, you need to pass the encryption information to your RTP stream read/write methods to enable them to do the media encryption/decryption. If you are using Avaya's client-media-stack, you may call the Audio "start" method (as usual) passing the encryption information as an extra parameter:

```
MediaEncryption encryption = new MediaEncryption();

encryption.setProtocol(startEvent.getEncryption().getProtocol());

encryption.setTransmitKey(startEvent.getEncryption().getTransmitKey());

encryption.setReceiveKey(startEvent.getEncryption().getReceiveKey());
```

```
encryption.setPayloadType(startEvent.getEncryption().getPayloadType().intValue());
audio.start(rtpAddress, rtcpAddress, codec, packetSize, encryption);
```

Media Encryption Information

The Encryption object in the `MediaStartEvent` contains the following information:

- Encryption protocol
- Separate media encryption transmit and receive keys
- Payload type

For release 3.1, the encryption keys and the payload type are only required if the encryption protocol is “MediaConstants.AES”.

For SDK compatibility reasons, the transmit and receive keys are formatted as Strings. For example, the transmit key may be in the form:

```
String transmitKey = “{38,4F,0B,34,DF,00,2A,BE,F0,C7,55,80,1D,1D,33,A8}”
```

and similarly for the receive key. The curly braces and commas are actually part of the String. In order to be used by the encryption/decryption routines, the keys need to be converted to byte arrays of the form (for example):

```
byte[] txKey = { 0x38, 0x4F, 0x0B, 0x34, 0xDF, 0x00, 0x2A, 0xBE, 0xF0, 0xC7, 0x55, 0x80, 0x1D, 0x1D, 0x33, 0xA8 };
```

If you use Avaya’s client-media-stack, the Audio “start” method will automatically do the String to byte[] conversion for you.

Encrypting and Decrypting the RTP Stream

The encryption transmit and receive keys, along with the roll over counter (ROC) plus the RTP header’s SSRC and sequence number, are used to calculate the Initialization Vector.

Roll Over Counter (ROC)

The ROC (initially set to zero) is a 32-bit unsigned integer which records how many times the 16-bit RTP sequence number (SEQ) has been reset to zero withing the same SSRC (after incrementing up through 65,535) Unlike the sequence number (SEQ), which your secure RTP implementation (SRTP) extracts from the RTP packet header, the ROC is maintained by the header of each RTP packet. The ROC must also be knowledgeable of the SSRC that is included in the header of each RTP packet. The SSRC is a 32 bit randomly chosen value in an RTP packet that is used to represent the synchronization source (RFC1889). From one `MediaStartEvent` to the next `MediaStopEvent`, the SSRC will remain the same. If the SSRC changes this will be an indication that a new RTP stream has started. When this situation occurs the ROC must be reset to zero.

```
// Increment the ROC whenever the sequence number rolls over
incomingReadSSRC = rtpHdr.ssrc;
incomingSeqNum = rtpHdr.seqNum;
if (currentReadSSRC == incomingReadSSRC) {
   if (incomingSeqNum > currentSeqNum) {
      // Do nothing
   } else if (incomingSeqNum < (currentSeqNum - 100)) {
      // Sequence number has probably rolled over
      ++readROC;
   } else {
      // out of sequence RTP packet - ignore it
      return 0;
   }
} else if (incomingReadSSRC == prevReadSSRC) {
   // very late RTP packet from previous call - ignore it
   return 0;
} else {
   // New SSRC (i.e. new call) - reset ROC
   readROC = 0;
   prevReadSSRC = currentReadSSRC;
   currentReadSSRC = incomingReadSSRC;
}
currentSeqNum = incomingSeqNum;
```

and similarly for `writeROC`.

Creating the Encryption Keys Using the Pseudo Random Function

The pseudo random function PRF_n(key,x) produces a bit string of length “n” from a string “x” which is encrypted using the encryption key named “key”. The AES Symmetric algorithm mode is “ECB” with no padding.

```
private status final byte Xe{} = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,}
// Set up and initialize JCE Engine
SymmetricEncryptionEngineIF see =
       SymmetricFactory.getInstance().createFactory(SymmetricFactory.AES);
see.initializeEngine("ECB","NoPadding");
// Generate the symmetric key using the JCE Engine and the readMasterKey from the
// MediaStart event and then use the AVAYA XE value to calculate the Ke-Rx value
see.generateKey(readMasterKey);
Ke-Rx = see.encryptBlock((Xe);
```

And, similarly for Ke-Tx based on the writeMasterKey.

Once the media receive and transmit encryption keys (Ke-Rx and Ke-Tx) are created, they will be used within the AES algorithm to encrypt and decrypt the RTP stream.

Creating the Initialization Vectors (IV)

The ROC, together with the media encryption keys from the `MediaStartEvent`, the SSRC and the RTP header sequence number are used to calculate the IV for each direction (transmit & receive):

```
private static final byte Xs-Rx[] = {0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0};
byte[] Ks-Rx = new byte[14];
ByteBuffer ssrcBuffer = ByteBuffer.allocate(16);
ByteBuffer KsBuffer = ByteBuffer.allocate(16);
ByteBuffer iBuffer = ByteBuffer.allocate(16);

// Convert the RTP header sequence number to bytes
byte[] seqHex = convert2Bytes(seq);
// Clear the local 16-byte buffers used in the IV calculation
ssrcBuffer.clear();
KsBuffer.clear();
iBuffer.clear();

// Softphone PRF version
// Set up and initialize JCE Engine
SymmetricEncryptionEngineIF see =
      SymmetricFactory.getInstance().createFactory(SymmetricFactory.AES)
see.initializeEngine("ECB","NoPadding");

// Generate the symmetric key using the JCE Engine and the readMasterKey from the
// MediaStart event and then use the Acaya Xs value to calculate the Ks-Rx value
see.generateKey(readMasterKey);
Ks-Rx = see.encrypBlock(Xs);
KsBuffer.put(Ks-Rx, 0, Ks-Rx.length-2);

And, similarly for Ks-Tx based on the writeMasterKey
// Grab the SSRC from the RTP header and populate the SSRCbuffer
ssrcBuffer.position(4);
ssrcBuffer.putInt(ssrc);

// Setup and populate the iBuffer - this will currently
// make the roll over counter to be zero always
iBuffer.position(8);
iBuffer.putInt(readROC);
iBuffer.put(seqHex[2]);
iBuffer.put(seqHex[3]);

byte[] ssrcBytes = ssrcBuffer.array();
byte[] ksBytes = KsBuffer.array();
byte[] iBytes = iBuffer.array();
byte[] iv-Rx = new byte [16];
iBuffer.put(seqHex[2]);
iBuffer.put(seqHex[2]);

// XOR all 3 buffers
for (int ii = 0; ii < iv-Rx.length; ii++) {
    iv-Rx[ii] = (byte)(ssrcBytes[ii] ^ ksBytes[ii] ^ iBytes[ii]);
```

and similarly for calculating the write buffer IV (iv-Tx).

Decrypting the Media Payload

In the draft SRTP specification, the encryption algorithm is defined as AES in counter mode (CTR and NoPadding). The generated IVs, along with the Ke-Rx or Ke-Tx, can be used to secure the RTP stream during each transmit and receive operation.

```
// get the RTP packet from the read socket
ByteBuffer dst = rtpPacket.getPacket();

// Point to payload start and obtain the encrypted payload
int hLength = 12; (RTP header size)
pLength = dst.limit() – hLength; (RTP payload size)

dst.position(hLength);
byte[] buf2 = new byte[pLength];
dst.get(buf2, 0, pLength);

// Decrypt payload
byte[] buf3 = decryptEngine.decryptBlock(buf2, iv-Rx, Ke-Rx);
```

The decryptBlock will look something like the following:

```
// decryptBlock sample code
Public byte[] decryptBlock(buf2, iv-Rx, Ke-Rx){
// Set up cipher engine
  SymmetricEncryptionEngineIF see =
SymmetricFactory.getInstance().createFactory(SymmetricFactory.AES);
see.initializeEngine("CTR", "NoPadding");

// Pass to JCE etc. . . .
see. decrypt(buf2, iv-Rx, Ke-Rx);
}
```

Test Data

In order to validate your code, we present here some test data against which you may run your decipher code. Following that is the expected output.

- Input Data

```
// From the MediaStart event, we get
byte[] readMasterKey = { (byte)0xaa,(byte)0xaa,(byte)0xaa,(byte)0xaa,
                         (byte)0xaa,(byte)0xaa,(byte)0xaa,(byte)0xaa,
                         (byte)0xaa,(byte)0xaa,(byte)0xaa,(byte)0xaa,
                         (byte)0xaa,(byte)0xaa,(byte)0xaa,(byte)0xaa
};

// From the RTP packet, we get
 int ssrc = 987011809;
 int seq = 3;
 int roc = 0;
```

```
// And the data to decipher is:
  cipherData = {(byte)0xbb,(byte)0xbb,(byte)0xbb,(byte)0xbb };
```

- Expected Output

```
Ke-Rx:
(byte)0xba, (byte)0xeb, (byte)0xc6, (byte)0x18, (byte)0xa5, (byte)0x5c, (byte)0x35,
(byte)0x1f, (byte)0x25, (byte)0xce, (byte)0xdf, (byte)0x37, (byte)0xbf, (byte)0x70,
(byte)0xf3, (byte)0x90
Ks-Rx:
(byte)0x98, (byte)0x12, (byte)0xf4, (byte)0x3c, (byte)0x17, (byte)0xc5, (byte)0xd4,
(byte)0x0e,   (byte)0xe3, (byte)0x8f, (byte)0x09, (byte)0xe1, (byte)0x7f, (byte)0xa8,
(byte)0xba, (byte)0xb7
IV-Rx:
(byte)0x98, (byte)0x12, (byte)0xf4, (byte)0x3c, (byte)0x2d, (byte)0x11, (byte)0x4e,
(byte)0xef,   (byte)0xe3, (byte)0x8f, (byte)0x09, (byte)0xe1, (byte)0x7f, (byte)0xab,
(byte)0x00, (byte)0x00

     // And the deciphered data should be:
     plainData = {(byte)0x05, (byte)0x43, (byte)0x2a, (byte)0x3d };
```

Security considerations

Your application development organization has the responsibility of providing the appropriate amount of security for your particular application and/or recommending appropriate security measures to your application customers for the deployment of your application. Therefore you should be aware of the security measures that AE Services Device, Media and Call Control API already takes and what risks are known.

AE Services Device, Media and Call Control API provides these security measures:

- The station password is required to register a device.
- Filenames specified for recorded files must be relative to the configured directory, their directories must already exist, and recordings cannot overwrite an existing file.
- Only files within the configured recorder directory can be deleted using the `VoiceUnitServices.deleteMessage()` method.

Application Enablement Services release 3.1 offers the user the ability to encrypt the voice RTP streams between the IP softphone and the far end of the call. See Media Encryption on page 86 for more information.

If you are using encryption, AE Services Device, Media and Call Control API provides these additional security measures:

- The signaling and bearer channels are encrypted.
- XML messages transmitted between the client application and the AE Services server software are encrypted.

- The station password is passed encrypted.
- Username and password are encrypted.
- Username and password are authenticated by the Device, Media and Call Control service.

WARNING:

If you do not use encryption on the client link, the signaling and bearer channels are unencrypted, there is no encryption of XML messages transmitted between the client application and the AE Services server software, the station password is passed unencrypted and the username and password are sent in the clear.

For a complete discussion of the security guidelines for AE Services, see *White-paper on Security in Application Enablement Services for Bundled and Software only solutions*. This white paper is available on the Avaya support site along with the customer documents.

Chapter 4: Compiling and Debugging

This chapter describes:

Compiling

In order to compile your application, you need to ensure that the api.jar file from the AE Services Device, Media and Call Control SDK is in your classpath. The Linux compile command may look something like this:

```
javac -d $BUILD_DIR -classpath $SDK_PATH/lib/api.jar MyApp.java
```

and the equivalent Windows compile command may look like this:

```
javac -d %BUILD_DIR% -classpath %SDK_PATH%/lib/api.jar MyApp.java
```

where:

BUILD_DIR	is set to the directory in which compiled class files should be placed.
SDK_PATH	is set to the path where the AE Services Device, Media and Call Control API SDK has been installed (e.g. `/opt/sdk` or `c:\sdk`).

You may wish to create a .jar file containing your compiled classes. See Sun Microsystem's Java documentation for details on this process.

Deploying your application on a test machine

Your application can be deployed and run on the connector server machine or on a separate application machine that is on the same LAN as the connector server. An *application machine* refers to a computer that is used to run your application(s).

To deploy your application follow these steps:

1. Set up your application machine on page 100
2. Configure your application on page 100
3. Run your application on page 101

Set up your application machine

When you go to deploy your application on a machine, you will have to ensure that the following files from the SDK have been deployed on the application machine along with your compiled application:

- Java runtime environment (JRE 1.5)
- All .jar files in $SDK_PATH/lib
- $SDK_PATH/lib/cmapi-client.properties
- $SDK_PATH/lib/proxy-client-configuration.xml
- Your compiled application
- Any application-specific configuration files as described in Provide application-specific properties below.

Configure your application

Provide application-specific properties

You may have some application-specific information that must be administered for your application to work with the connector server. For example, you can supply an application-specific properties file that allows users or administrators of the application to supply this information. At a minimum, you will need to get the following information to register an extension through the connector server with Communication Manager.

- IP address of the Communication Manager media server
- Extension number that you will be using to register
- Password for the above extension number

For an example of how these values can be supplied through a properties file, see how the TutorialApp does it. The TutorialApp uses this properties file:

cmapisdk/examples/resources/tutorial.properties

You do not have to use this file name. You can name it whatever you want.

If you do not wish to use a properties file, you will need to have some alternate means of getting this information from the user, such as a GUI.

Set up and start the connector server

If you have not already done so, install, administer, run and verify the connector server software as described in the appropriate *Avaya MultiVantage™ Application Enablement Services Installation Guide* for the offer you have purchased (bundled server or software only).

Run your application

You may wish to use ant to run your applications. You will have to ensure that all the files listed above are included in the classpath when launching your application. In addition, if you are using an application-specific properties file, and you are using the classloader mechanism to load this file into your application, you will have to include in your classpath the directory in which your properties file resides.

The following example assumes that all necessary files (including your compiled application `MyApp` and your optional application-specific properties file) are in the $APP_DIR/lib directory. The critical line that launches your Java application in a Linux `run.sh` file might look like the following:

```
java -classpath $APP_DIR/lib:$APP_DIR/lib/api.jar:$APP_DIR/lib/proxy.jar MyApp
```

On Windows the equivalent `run.bat` file might look like this:

```
java -classpath %APP_DIR%/lib;%APP_DIR%/lib/api.jar;%APP_DIR%/lib/proxy.jar MyApp
```

Debugging

This section assumes that you have verified the operation of your connector server using the verification application that comes with the connector server RPM file. Please refer to the appropriate *Avaya MultiVantage™ Application Enablement Services Installation Guide* for the offer you have purchased (bundled server or software only) for troubleshooting procedures if this verification application does not function properly.

In debugging your application, you will rely on:

1. **Exceptions** that your application catches and logs. See Catching exceptions on page 46.
2. **Server-side logs** found at /opt/mvap/logs. See the *Avaya MultiVantage™ Administration and Maintenance Guide* guide to learn more about the server's logs.

Note:

The AE Services Device, Media and Call Control SDK has a very limited logging capability on the client side based on the java.util.logging package, available in version 1.4 of the JDK. The SDK log messages will likely not be very useful when debugging applications. These loggers can be turned off by adding the following line to the logging.properties file:

```
com.avaya.level=OFF
```

The logging.propertire file is, by default, under $JAVA_HOME/jre/lib

The remainder of this section helps you with:

- Common exceptions on page 103
- Possible race conditions on page 106
- Improving performance on page 107

Common exceptions

Common exceptions that you may encounter are listed below along with their potential solutions:

Table 33: Common exceptions

Exception	Potential causes and solutions
`java.net.ConnectException: Connection refused`	This means that the connector server IP address your application used to establish a connection to the connector server is a valid IP address, but it cannot be connected to. This may occur: • if the connector server software is not actively running or if the wrong address was provided. Check that the connector server software is running and that the correct IP address is provided for the server. • if there are network issues between the application machine and the connector server. Check to see if you can ping the server. • The connector server software is not running on the connector server. Try restarting it. If you have checked all of these things and are still having a problem, you may also want to check the /etc/hosts file on the connector server and verify that you have a line in that file that explicitly lists the IP address of the connector server, in addition to the localhost line.
`java.nio.channels.UnresolvedAddress Exception`	This likely means that your application used an invalid IP address to establish a connection to the AE Services server. Correct the address. Do *not* use local host address 127.0.0.1; use the actual IP address.

Table 33: Common exceptions (continued)

Exception	Potential causes and solutions
`ch.ecma.csta.errors.CstaException <stack trace> Caused by: java.nio.channels.UnresolvedAddressException`	The IP address for Communication Manager that was specified in the Device Services `GetDevice` request is incorrect, or the address for the H.323 gatekeeper was administered wrong on the OAM web page. This exception is not sent until you first try to use the `deviceId` in a Registration Services `RegisterTerminal` request. Ensure that you are using the correct IP address for Communication Manager.
`InvalidConnectionIDException: Player unavailable for this session: Note: Player will be unavailable if the device is using G729 or G729A codec <`***`connection id`***`>`	Device specified in the Voice Unit Services `PlayMessage` request is unregistered. It may have been automatically unregistered due to a loss of communication with Communication Manager. See Possible race conditions below. Check the health of Communication Manager and the network and then try again.
`InvalidConnectionIDException: invalid mediasession or Recorder <`***`connection id`***`>`	Device specified in the Voice Unit Services `RecordMessage` request is unregistered. It may have been automatically unregistered due to a loss of communication with Communication Manager. See Possible race conditions below. Check the health of Communication Manager and the network and then try again.
`NotAbleToPlayException: File already being played.`	Two different threads of the application may be trying to play messages to the same device at the same time. See Possible race conditions below. Modify the application so that this does not occur.
`ch.ecma.csta.errors.CstaException: Request with info=com.avaya.mvcs.proxy.XmlGatewayClient$RequestInfo@fa70a4 timed out`	Connector server did not respond in time so the client proxy timed out. Look at the connector server logs to determine the cause.
`InvalidSessionException`	The session has timed out and been placed in an inactive state.
`com.avaya.mvcs.proxy.SessionNoLongerValidException: Request: session[null] ch.ecma.csta.binding.start.StartApplicationSession@1d9e2c7 canceled`	If this exception is thrown when the client application is attempting to connect to the AE Services Server, and the property `cmapi.secure` is set in the client application, then the client may be attempting to connect to an unencrypted port on the server. Verify that the client application is configured to connect to the secure port of the AE Services Server.

2 of 4

Table 33: Common exceptions (continued)

Exception	Potential causes and solutions
`com.avaya.mvcs.proxy.SessionNoLongerValidException: Request: session[null] ch.ecma.csta.binding.start.StartApplicationSession@5c98f3 canceled`	If this exception is thrown when the client application is attempting to connect to the AE Services Server, and the property `cmapi.secure` is not set in the client application, then the client may be inadvertently configured to connect to an encrypted port on the server. Verify that the client application is configured to connect to the unencrypted port of the AE Services Server.
`javax.net.ssl.SSLException: SSL handshake failed`	This exception can be thrown when the server's certificate was not signed by a trusted certificate authority. Verify that the AE Services Server's certificate was signed by a Certificate Authority whose public key exists in the client's trust store.
`javax.net.ssl.SSLHandshakeException: general SSLEngine problem Caused by : sun.security.validator.ValidatorException: No trusted certificate found`	This exception can be thrown when the Java Key Store name avaya.jks is missing from the resources directory. If the file has been renamed, rename it to avaya.jks. If the file has been removed, reinstall it.
`java.lang.IllegalArgumentException: No file is stored in trusted certificates location (resources/my.jks)`	The property `cmapi.trust_store_location` is set and the file it refers to (in this example my.jks) does not exist. Have `cmapi.trust_store_location` refer to avaya.jks
`java.io.IOException: invalid keystore format`	If the property `cmapi.trust_store_location` is set, then the file it refers to is either not a Java Key Store or it is a corrupt Java Key Store. If the property `cmapi.trust_store_location` is not set, then the file resources/avaya.jks is corrupt. Reinstall avaya.jks.
`IllegalArgumentException: Trusted certificates location is an empty string`	The property `cmapi.trust_store_location` is set to an empty string. Either don't set the property (preferred in 3.1) or set the property to the path to the java key store.

3 of 4

Table 33: Common exceptions (continued)

Exception	Potential causes and solutions
`SessionCleanedUpException`	This indicates that the session in question is no longer valid.
`NotValidatedException`	This is an unlikely Exception type to get, but it is possible. The only time a session will be in this state is when the positive response to the `StartApplicationSession` message has not yet been returned. Validating the session is part of the processing that is done during the processing of the `StartApplicationSession` message. This should not require recovery because the session will be validated once the processing of the message is completed.
`ch.ecma.csta.errors.CstaException:Unexpected CSTA error code:<error code>`	Call Control Services has received an unexpected CSTA error code from TSAPI. See Appendix C: TSAPI Error Code Definitions on page 115 for the definition of the error code. Consult the TSAPI Programmers's Guide for more details.
`ch.ecma.csta.errors.CstaException:Unexpected ACS error code:<error code>`	Call Control Services has received an unexpected ACS error code from TSAPI. See Appendix C: TSAPI Error Code Definitions on page 115 for the definition of the error code. Consult the TSAPI Programmers's Guide for more details.

4 of 4

Possible race conditions

You should be aware of some scenarios in which two different threads may be acting in opposition to one another against a single device. Some known race conditions are described below:

- When the connector server detects through its "keep-alive" mechanism that it can no longer communicate with a Communication Manager C-LAN (CLAN) or processor C-LAN (PROCR) that has devices registered to it (possibly due to a network failure or congestion), the connector server automatically unregisters the devices; any media sessions for those devices are cleaned up; and an `UnregisterEvent` is sent to all the listeners listening to those devices. If, at the same time, an application is in the middle of sending a media request to play a file, start tone detection, or start recording on one of those automatically unregistered devices, an `InvalidConnectionIDException` is thrown indicating that the media session is unavailable.

- If one thread is actively playing a message to a device and a second thread attempts to play a message to the same device, the second thread will receive a `NotAbleToPlayException`.
- If an application simultaneously uses both of the following:
  - `VoiceUnitServices` to request that playing or recording terminate when a specified DTMF tone is detected
  - `ToneDetectionServices` or `ToneCollectionServices` to listen for DTMF tones

  there is no guarantee of which of the following occurs first when the termination tone is received:
  - the `VoiceUnitServices` player/recorder terminates and generates a `StopEvent`
  - the `ToneDetectionServices`/`ToneCollectionServices` generates a `ToneDetectedEvent`/`TonesRetrievedEvent`

  If the application assumes the player or recorder has stopped playing/recording when the `ToneDetectedEvent`/`TonesRetrievedEvent` is received and requests another play/record request, an exception may be thrown if the play/record is still in progress on that device. If the application wants to be certain that the player/recorder is finished, it should wait for the `StopEvent` before making another play/record request.

Improving performance

Many different factors may potentially affect the performance of your system. The system has three main parts that may be affected:

- The AE Services server
- Communication Manager
- The network

An excessive load on any of these may slow down the overall system. Here are other factors to check that also may affect your system performance.

On the AE Services server:

- Ensure that your AE Services server machine meets the minimum requirements specified in the appropriate *Avaya MultiVantage™ Application Enablement Services Installation Guide* for the offer you have purchased (bundled server or software only).
- Avoid running any other applications on the AE Services server machine.
- Check that the AE Services server's Linux operating system resources are tuned correctly for your application needs. The server software makes no assumptions concerning your application needs and therefore does not tune the kernel for you. The "Appendix C: Manually configuring Linux and installing/configuring third-party software" chapter in the appropriate *Avaya MultiVantage™ Application Enablement Services Installation Guide* for the offer you have purchased (bundled server or software only) provides some guidance in

tuning Linux. See also the Linux documentation found at http://www.redhat.com/docs/manuals/linux/RHL-9-Manual.

- Update the Linux kernel with the latest updates available.

On Communication Manager:

- Ensure that Communication Manager is properly configured for your network and business needs. Misconfigured Communication Manager elements can lead to performance issues.

On the network:

- Ensure that your network traffic is properly balanced. One way to do this professionally is to ask Avaya to perform a network assessment. There is also a *VoIP Readiness Guide* available from the Avaya Support Centre (http://www.avaya.com/support). For more information about improving the performance of your network, see the “Network Quality and Management” section of *Administration for Network Connectivity for Avaya Communication Manager* (555-233-504).

Getting support

Development support is only available through Avaya's DeveloperConnection Program (DevConnect) at this time. As an Innovator/Premier/Strategic level member of the DevConnect Program, technical support questions can be answered through the DevConnect Portal at www.devconnectprogram.com.

As a Registered member of the program, support is not available. If you require support as a Registered member, you can apply for a higher level of membership that offers support and testing opportunities through the DevConnect Portal. Membership at the Innovator/Premier/Strategic level is not open to all companies. Avaya determines membership status above the Registered level.

Appendix A: Communication Manager Features

Here is a list of key features in Communication Manager that you may wish to take advantage of in developing your applications. This is not an exhaustive list - just a subset of features most likely to be used in applications. For descriptions of these features, see any of the Communication Manager administration guides.

- AAR/ARS Partitioning
- Abbreviated Dialing
- Abbreviated and Delayed Ringing
- Abbreviated Programming
- Administration Without Hardware
- Authorization Codes
- Automatic Alternate Routing (AAR)
- Automatic Call Distribution (ACD) Features
  - Announcements
  - Automatic Answering With Zip Tone
  - Multiple Call Handling
  - Service Observing
    - Observe Digital Sets/IP Phones
    - Observe Logical Agent IDs
- Automatic Call Distribution (ACD) Features:
  - Basic Hunt Group Call
  - Agents in Multiple Splits
  - Agent Login/Out
  - Display - Agent Terminal
- Automatic Callback (on Busy)
- Automatic Callback on Don’t Answer
- Automatic Route Selection (ARS)
- Bridged Call Appearance (single-line, multi-appearance)
  - Hunt Group Redirect Coverage
  - Multiple Coverage Paths
  - Temporary Bridged Appearance
  - Remove Temporary Bridged Appearance

- Call Coverage Features
  - Call Coverage Consult with Conference/Transfer
- Call Coverage Features
  - Consult
  - Coverage Paths
  - Send All Calls
  - Temporary Bridged Appearance
- Call Forwarding / Busy Don’t Answer @ Call Vectoring
  - VDN of Origin Announcements
- Call Forwarding by Service Observer
- Call Forwarding by Service Observed
- Call Forwarding Features:
  - Call Forwarding All Calls
  - Call Forwarding - Busy and Don’t Answer
  - Call Forwarding - Don’t Answer
  - Call Forwarding - Off Net
- Call Park
- Call Pickup
  - Directed Call Pickup
  - Remove Temporary Bridged Appearance
  - Remove Auto-Intercom
- Call Vectoring:
  - VDN of Origin Display
- Consult
- Coverage of Calls Redirected Off Net
- Drop (button operation)
- Group Paging
- Hold (single-line, multi-appearance)
- Hold - Automatic
- Hold (single-line, multi-appearance)
- Hold - Automatic [from IR1V4 WCC]
- Hunting/Hunt Groups

- IP Trunks
- Last Number Dialed
- Leave Word Calling - Switch
- Malicious Call Trace
- Message Waiting Indication
- Music-on-Hold Access
  - Held Calls
  - Conference-Terminal Calls
  - Transferred Trunk Calls
- Personalized Ringing
- Personal Station Access
- Priority Calling
- Recorded Announcement
- Terminal Translation Initialization (TTI)
- Tone on Hold
- Transfer (single-line, multi-appearance)
- Voice Terminal Display
  - Calling Number Display (SID/ANI/Extn ID)
  - Called Number Display (internal & DCS)
  - Stored Button Display

Appendix B: Migrating Communication Manager API 2.1 Applications to Application Enablement Services

Migrating from Communication Manager API 2.1 to AE Services 3.0

Consider the following when migrating Communication Manager 2.1 applications to AE Services.

- AE Services is compatible with existing Communication Manager API 2.1 Java applications only if they are run with the Device and Media Control SDK. (No code changes or recompiles are necessary, but new Jar files must be installed on all clients.)
- Device and Media Control provides the same functionality as Communication Manager API 2.1 except for session management and the use of the Gatekeeper list in populating the switch name. These features have been described in the following sections in this document and are fully backwards compatible:

  Session Management on page 47

  Populating the Switch Name field on page 54

- The Communication Manager API Service Call Information Service (formerly DAPI link) now requires an AE Services transport link to be administered both on Communication Manager and AE Services. This API now supports links from a single AE Services server to multiple Communication Managers. See the *Avaya MultiVantage™ Application Enablement Services Administration and Maintenance Guide* to learn how to administer this link.
- When deploying a Device and Media Control application, you must now specify the device and media control server configuration properties in the AE Services server OAM pages instead of in a configuration properties file.

Migrating from AE Services 3.0 to AE Services 3.1

Consider the following when migrating AE Services Device and Media Control 3.0 applications to Device, Media and Call Control 3.1.

- The application link is now authenticated. While applications were previously required to populate the username and password fields, these properties were not validated. In the 3.1 release, the properties are now authenticated with the User Service. The application will have to provide a valid username and password in order to establish a session. For more information and example code see Getting a ServiceProvider instance on page 47.
- The clear-text application link port (4721) is now disabled by default. If it is desired to run your application with no modifications, you can enable this non-secure port through the Web OA&M. However, we strongly recommend that the application be modified to take advantage of this secure connection. In order to do so, it is necessary to change the cmapi.port property to 4722, and specify a new `cmapi.secure="true"` property when calling `getServiceProvider`. For more information and example code see Getting a ServiceProvider instance on page 47.
- It is necessary to deploy the application with the 3.1 SDK jar files.

Appendix C: TSAPI Error Code Definitions

This appendix lists all of the values for the TSAPI error codes.

There are two major classes of TSAPI error codes:

- CSTA universal Failures
- ACS Universal Failures

CSTA Universal Failures

CSTA Universal Failures are error codes returned by CSTAException:Unexpected CSTA error code. The following table lists the definitions for the CSTA error codes. Consult the TSAPI Programmer's Guide for the definition of the numeric error code.

Table 34: CSTA Error Definitions

Error	Numeric Code
genericUnspecified	0
genericOperation	1
requestIncompatibleWithObject	2
valueOutOfRange	3
objectNotKnown	4
invalidCallingDevice	5
invalidCalledDevice	6
invalidForwardingDestination	7
privilegeViolationOnSpecifiedDevice	8
privilegeViolationOnCalledDevice	9
privilegeViolationOnCallingDevice	10
invalidCstaCallIdentifier	11
invalidCstaDeviceIdentifier	12
invalidCstaConnectionIdentifier	13
invalidDestination	14

1 of 4

Table 34: CSTA Error Definitions (continued)

Error	Numeric Code
invalidFeature	15
invalidAllocationState	16
invalidCrossRefId	17
invalidObjectType	18
securityViolation	19
genericStateIncompatibility	21
invalidObjectState	22
invalidConnectionIdForActiveCall	23
noActiveCall	24
noHeldCall	25
noCallToClear	26
noConnectionToClear	27
noCallToAnswer	28
noCallToComplete	29
genericSystemResourceAvailability	31
serviceBusy	32
resourceBusy	33
resourceOutOfService	34
networkBusy	35
networkOutOfService	36
overallMonitorLimitExceeded	37
conferenceMemberLimitExceeded	38
genericSubscribedResourceAvailability	41
objectMonitorLimitExceeded	42
externalTrunkLimitExceeded	43
outstandingRequestLimitExceeded	44

2 of 4

Table 34: CSTA Error Definitions (continued)

Error	Numeric Code
genericPerformanceManagement	51
performanceLimitExceeded	52
unspecifiedSecurityError	60
sequenceNumberViolated	61
timeStampViolated	62
pacViolated	63
sealViolated	64
genericUnspecifiedRejection	70
genericOperationRejection	71
duplicateInvocationRejection	72
unrecognizedOperationRejection	73
mistypedArgumentRejection	74
resourceLimitationRejection	75
acsHandleTerminationRejection	76
serviceTerminationRejection	77
requestTimeoutRejection	78
requestsOnDeviceExceededRejection	79
unrecognizedApduRejection	80
mistypedApduRejection	81
badlyStructuredApduRejection	82
initiatorReleasingRejection	83
unrecognizedLinkedidRejection	84
linkedResponseUnexpectedRejection	85
unexpectedChildOperationRejection	86
mistypedResultRejection	87
unrecognizedErrorRejection	88

3 of 4

Table 34: CSTA Error Definitions (continued)

Error	Numeric Code
unexpectedErrorRejection	89
mistypedParameterRejection	90
nonStandard	100
	4 of 4

ACS Universal Failures

ACS Universal Failures are error codes returned by CSTAException:Unexpected ACS error code. The following table lists the definitions for the ACS error codes. Consult the TSAPI Programmer's Guide for the definition of the numeric error code

Table 35: ACS Error Definitions

Error	Numeric Code	Description
ACSERR_APIVERDENIED	-1	This return indicates that the API Version requested is invalid and not supported by the existing API Client Library.
ACSERR_BADPARAMETER	-2	One or more of the parameters is invalid.
ACSERR_DUPSTREAM	-3	This return indicates that an ACS Stream is already established with the requested Server.
ACSERR_NODRIVER	-4	This error return value indicates that no API Client Library Driver was found or installed on the system.
ACSERR_NOSERVER	-5	This indicates that the requested Server is not present in the network.
ACSERR_NORESOURCE	-6	This return value indicates that there are insufficient resourcesto open a ACS Stream.
ACSERR_UBUFSMALL	-7	The user buffer size was smaller than the size of the next available event.
ACSERR_NOMESSAGE	-8	There were no messages available to return to the application.
		1 of 2

Table 35: ACS Error Definitions (continued)

Error	Numeric Code	Description
ACSERR_UNKNOWN	-9	The ACS Stream has encountered an unspecified error.
ACSERR_BADHDL	-10	The ACS Handle is invalid.
ACSERR_STREAM_FAILED	-11	The ACS Stream has failed due to network problems. No further operations are possible on this stream.
ACSERR_NOBUFFERS	-12	There were not enough buffers available to place an outgoing message on the send queue. No message has been sent.
ACSERR_QUEUE_FULL	-13	The send queue is full.

2 of 2

Glossary

A

AE Used as a “shorthand” term in this documentation for Application Enablement.

AES Stands for Advanced Encryption Sheme.

API Application Programming Interface. A “shorthand” term in this documentation for the Java interface provided by the Application Enablement Services. See also connector client API library

application machine The hardware platform that the connector client API library and the client application are running on

B

BHCC busy hour call capacity

C

client application An application created using the Device, Media and Call Control API

CMAPI softphone Application Enablement Services Device, Media and Call Control API software objects that represent softphone-enabled, Communication Manager telephones or extensions

Application Enablement Services Device, Media and Call Control API The product name. This includes the server-side runtime software (see connector server software) and the connector client API library. This term is never used to reference only the client API library.

connector This describes the function of Application Enablement Services Device, Media and Call Control API.

In this context, “connector” means software and communications protocol(s) that allow two disparate systems to communicate. Often used to provide open access to a proprietary system. In the case of Application Enablement Services Device, Media and Call Control API, the connector enables applications running on a computing platform to incorporate telephony functionality through interaction with Communication Manager.

connector client API library The Application Enablement Services Device, Media and Call Control Java API, also referred to as the AE

connector server machine The hardware platform that the connector server software is running on. In these documents, the term “connector server” by itself never refers to the connector server machine. See connector server software.

connector server software	The Application Enablement Services server-side runtime software, often referred to as the “connector server” in these documents
CSTA	Computer-Supported Telecommunications Applications
D	
DMA	Direct memory access
E	
ECMA	European Computer Manufacturers Association. A European association for standardizing information and communication systems in order to reflect the international activities of the organization.
H	
hold time	The total length of time in minutes and seconds that a facility is used during a call
J	
JDK	Java Developers Kit
J2SE	JavaTM 2 Platform, Standard Edition
JMX	JMX (Java Management Extensions) is a set of specifications for application and network management in the J2EE development and application environment. JMX defines a method for Java developers to integrate their applications with existing network management software by dynamically assigning Java objects with management attributes and operations
JSW	Java Service Wrapper
JVM	Java Virtual Machine. Interprets compiled Java binary code for a computer’s processor so that is can perform a Java program’s instructions
O	
OAM	Operations, Administration and Maintenance
R	
RPM	Red Hat Package Manager
S	
SAT	System Access Terminal (for Communication Manager)
SDK	Software Development Kit. An SDK typically includes API library, software platform, documentation, and tools.

T

TCP Transmission Control Protocol. A connection-oriented transport-layer protocol, IETF STD 7. RFC 793, that governs the exchange of sequential data. Whereas the Internet Protocol (IP) deals only with packets, TCP enables two hosts to establish a connection and exchange streams of data. TCP guarantees delivery of data, and also guarantees that packets are delivered in the same order in which the packets are sent.

TSAPI Telephony Services Application Programming Interface. A service that provides 3rd party call control.

TTI Terminal Translation Initialization. This is a feature in Communication Manager that allows administrators, when initially administering new DCP stations, to not initially bind the extension number to a port. When the technician is installing the stations, they then use the TTI feature access code to bind the extension number to the station.

V

VoIP Voice over IP. A set of facilities that use the Internet Protocol (IP) to manage the delivery of voice information. In general, VoIP means to send voice information in digital form in discrete packets instead of in the traditional circuit-committed protocols of the public switched telephone network (PSTN). Users of VoIP and Internet telephony avoid the tolls that are charged for ordinary telephone service.

X

XML Extensible Markup Language

XSD XML Schema Definition. Specifies how to formally describe the elements in an Extensible Markup Language (XML) document. This description can be used to verify that each item of content in a document adheres to the description of the element in which the content is to be placed. This protocol uses these instead of DTD's.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X