

ReadiVoice® API

Developer's Guide



Trademark Information

Polycom®, the Polycom logo design, ReadiVoice®, InnoVox®, and the Voyant logo are registered trademarks of Polycom, Inc. Voyant Technologies™ is a trademark of Polycom, Inc. All other trademarks are the property of their respective owners.

Patent Information

The accompanying product is protected by one or more U.S. and foreign patents and/or pending patent applications held by Polycom, Inc.

Catalog No. 3725-70047-009E1 (10/2006)

v. 3.0

© 2006 Polycom, Inc. All rights reserved.

Polycom Inc.

1765 West 121st Avenue

Westminster, CO 80234-2301 U.S.A.

No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Polycom, Inc. Under the law, reproducing includes translating into another language or format.

As between the parties, Polycom, Inc. retains title to, and ownership of, all proprietary rights with respect to the software contained within its products. The software is protected by United States copyright laws and international treaty provision. Therefore, you must treat the software like any other copyrighted material (e.g. a book or sound recording).

Every effort has been made to ensure that the information in this manual is accurate. Polycom, Inc. is not responsible for printing or clerical errors. Information in this document is subject to change without notice.

Contents

About this Manual	vii
Purpose	vii
Document Conventions	viii
Customer Support	ix

1 Introducing the ReadVoice APIs

CAPI Overview	1
ReadVoice and CAPI Application Architecture	1
Key Features of CAPI	2
AAPI Overview	3
Admin API Architecture	3
Getting Started	3
Installing the SDK and Its Documentation	4
Software Not Included in the SDK	4
Software Included in the SDK	4
ReadVoice SDK Installation	5
Accessing the Documentation	6

2 Developing a CAPI Application

Writing Your CAPI Application	8
Understanding CAPI Architecture	8
Understanding CAPI Core Communication Modes: Synchronous and Asynchronous	10
Understanding How Ports Are Used	11
Pull Mode and Ports	11
Push Mode and Ports	11
Ports and Firewalls	12
Transport Layer Security	13
Understanding Connections, Sessions, and Logins	14
Overview of Connections, Sessions, and Logins	14
Application, Moderator, and Participant Sessions	15
High-Level Overview of the Conferencing Process	16

Relationships Among Connections, Sessions, and Logins	16
Benefits and Drawbacks	19
CAPI SDK Components	20
Understanding the CAPI Events	20
Understanding the CAPI Java Classes	21
Programming Guidelines	29
How the ODPROC and the CSC Work Together	29
Application Design and Configuration	30
Programming Fundamentals	31
Programming Tips	33
Using ACM	36
How ACM Works	36
Two Types of ACM Applications	37
Basics of ACM Implementation	37
ACM Events	38
Communications Between Readivoice and ACM Applications	39
ACM CDRs	39
Setting Up an ACM Application in the Readivoice System	40
Conference User Data Feature	43
Commonly Used Conferencing Procedures	43
Additional Sources of Information	43
Procedure Sequence	43
Procedure Notes	44
Initial Procedures	45
Conference Procedures	52
Conference-End Procedures	67
Sample Moderator Application	69
SimplePushModerator	69
SimplePushModeratorHandler	88

3 Developing an AAPI Application

Port Usage	95
Connections, Sessions, and Logins	95
Connection URL for Readivoice	96
Admin API SDK Components	97
Associating the AAPI Events and the Java Utility Classes	98
Auto-generated helper classes	99
Admin API Responses	99

A Migrating MAPI Applications to CAPI

Migration Overview	101
Compatibility Notes	102
MAPI-to-CAPI Sample Application Comparison	103

Index	137
--------------------	------------

About this Manual

This introduction provides a brief overview of the *ReadiVoice API Developer's Guide*, describes the conventions used in this manual, and explains how to get additional information or support.

Purpose

This manual is for developers of a client application for the ReadVoice Conferencing API (CAPI) or Administration API (AAPI). It includes information about:

- CAPI/AAPI architecture and process flow
- Writing a conferencing application using CAPI
- Writing a system administration application using AAPI
- Migrating MAPI applications to the new CAPI

Since conferencing is much more complex and involved, and since AAPI is implemented much like CAPI, this manual focuses mainly on CAPI. The Admin API shares the same architecture and uses some of CAPI's core Java classes, so learning how CAPI works is a prerequisite for learning about AAPI.

CAPI/AAPI applications interact with a ReadVoice system. To write applications using CAPI or AAPI, you need to understand the ReadVoice system, its access type options, its conferencing features and options, and how provisioners, operators, and other users use the system. We recommend that you review the information in the following documents:

- *ReadiVoice Provisioning Guide*
- *ReadiVoice Operator Guide*
- *ReadiVoice Subscriber Guide*
- *ReadiVoice Administration and Maintenance Guide*

Document Conventions

This document uses the following typographical conventions.

Typeface	Usage
bold	Names of fields, screens, windows, dialog boxes, and other user interface elements; for example: <ol style="list-style-type: none"> 1 Type the number into the Phone Number field and click Dial. 2 Click Cancel to close the dialog box.
<i>italics</i>	New terms, book titles, or emphasis; for example: According to the <i>VERITAS Cluster Server User Guide</i> , crash tolerant applications are sometimes referred to as <i>cluster friendly</i> applications.
code	Computer output, command references within text, and filenames; for example: Performs the initial configuration and reads the .vcsrc file
code, bold	Command line entries, for example: >> Type cp ../default_group.ini group.ini.
code, bold & italics	Command line variables, for example: >> Type cp ../default_group.ini group<i>nn</i>.ini replacing <i>nn</i> with the subscriber group number.
SMALL CAPS	Specific keys on the keyboard, for example: >> Move the cursor by pressing TAB or SHIFT+TAB.

Customer Support

Recognizing that technology alone cannot solve today's complex challenges, Polycom Global Services provides the industry's best technical support staff and programs to let you concentrate on the task at hand. ReadVoice users can select from a variety of support solutions to obtain the level of support that best meets their needs.

Before contacting your Polycom Global Services representative for technical assistance, gather as much information as possible about your situation. Any information you can provide helps us assess the problem and develop an appropriate solution.

Polycom Global Services Telephone and Email

If you have comments or questions about ReadVoice or if you need technical assistance, contact:

- Polycom Global Services in U.S.A 800-827-7782
- Denver metro or outside U.S.A. 303-223-5223
- Email: techsupport@polycom.com

Polycom Global Services on the Web

For more information about Polycom Global Services, go to <http://www.polycom.com> and select the Support Solutions link under the Global Services category.

Polycom Technical Publications Department

If you have comments or questions about this or any other ReadVoice documentation, contact:

Polycom Technical Publications Department
1765 West 121st Avenue
Westminster, CO 80234-2301 U.S.A.

Email: TechPubs@polycom.com

Introducing the ReadVoice APIs

This chapter provides a brief overview of the ReadVoice SDK and its two primary components, CAPI and AAPI, and describes its requirements, components, and installation.

CAPI Overview

ReadVoice lets subscribers launch and control conferences at any time, using touchtones or a web interface. It isn't necessary to make reservations ahead of time or restrict time limits. The features users need to set up and control conferences are provided in the conferencing application you write using CAPI, and in the ReadVoice system to which the application connects.

If you're not familiar with conferencing applications or the ReadVoice system, refer to the ReadVoice documentation suite, focusing on the *Provisioning Guide* and *Subscriber Guide*.

ReadVoice and CAPI Application Architecture

The CAPI tools in this product let you write a client application that interacts with a ReadVoice system. A conferencing system based on ReadVoice CAPI has three main parts: the CAPI client application, the CACS that's the core of the conferencing system, and the bridges where the conferences take place.

A CAPI client application connects to the CACS, which interacts with and controls the behavior of the bridges.

Your client application can communicate with the CACS in two ways:

- Synchronous communication, or pull, means a request is sent and the response is sent back immediately on the same connection. Use this method when you need a stateless protocol.
- Asynchronous communication, or push, means a request is sent and either a response is not required, or a response is sent later after the information requested has been obtained, on another connection. Use this method when you need a more robust set of features.

Key Features of CAPI



Beginning with v. 3.0, ReadiVoice no longer supports MAPI.

If you have a MAPI application, use the information in this guide to port that application to CAPI. See [Appendix A](#), “Migrating MAPI Applications to CAPI,” for specific help with this.

CAPI is the latest release of Polycom’s powerful teleconferencing API. Key features include the following:

- CAPI encapsulates the server components from the conferencing application, making it easier to write and maintain CAPI applications
- CAPI is XML-based, allowing developers to write applications in a variety of programming languages. Java utility files and event files are provided, as well.
- A new login type, Application, is available. The Application user is like a super-moderator, and can monitor several conferences at once. For instance, Application can receive notifications of all conferences being created and ended.
- A conference can have multiple subscriber and participant logins.
- Log4j and Apache Common Logging have been used to provide logging functionality.
- CAPI uses XML for all transport. Use of XML allows for better backward-compatibility in future versions.
- An ACM event set is provided to allow CAPI applications to interface with external applications. See “[Using ACM](#)” on page 36 for more information.
- Synchronous and asynchronous functionality is available. Synchronous REQUEST/RESPONSE functionality goes through port 80 on the CACS. Asynchronous event notifications for conference and participants allow for instant notification for any change in participant or conference state. Notification that requires that some information be generated and returned is sent asynchronously on a user-defined port. All asynchronous events will go from server to client.
- A UserData parameter added to the CONF_INFO event and a new SET_CONF_USER_DATA event enable you to assign a user data string to a conference. This user data string persists through resynchronizations and failovers, and remains available until the conference ends. See “[Conference User Data Feature](#)” on page 43 for more information.

A-API Overview

The Admin API provides functionality like that currently found in the Readivoice System Administration web page. This includes administering subscriber groups, number groups, access numbers, access classes, and admin users. For a complete list of available admin requests, please consult the Admin API Javadocs and event reference guide.

Admin API Architecture

The Admin API re-uses the core communication and utility classes of CAPI. It adds a new set of Castor utility and event classes and implements the necessary core interfaces to provide the administration interface into the Readivoice system.

Like CAPI, the Admin API interacts with the Readivoice system through the Apache web server on the target CACS. By default, it uses port 80. API requests are forwarded by Apache to the admin server process, which validates the requests and interacts with the necessary Readivoice components for processing. Once the request is completed, the admin server sends back a response to the client through Apache.

Getting Started

To get started:

- 1 Install the SDK. See [“Installing the SDK and Its Documentation”](#) on page 4.
- 2 If you’re going to develop a CAPI application, study [Chapter 2, “Developing a CAPI Application.”](#) This chapter, combined with the technical reference information in the *Conferencing API Reference* (either the PDF or the Javadoc HTML), provide what you need to know to write a conferencing application. But because A-API is so similar and shares some of the core components with CAPI, it’s also information you need in order to write an administration application.
- 3 If you’re going to develop an Admin API application, first study [Chapter 2, “Developing a CAPI Application.”](#) Because A-API is so similar and shares some of the core components with CAPI, you need to understand the CAPI information in order to understand A-API. Then, study [Chapter 3, “Developing an A-API Application,”](#) and see the *Administration API Reference* (either the PDF or the Javadoc HTML).
- 4 If you’re migrating from MAPI to CAPI, read both [Chapter 2, “Developing a CAPI Application.”](#) and [Appendix A, “Migrating MAPI Applications to CAPI.”](#) Use the technical reference information in the *Conferencing API Reference*, as well.

Installing the SDK and Its Documentation

This section describes the software and its dependencies, the SDK installation, and how to access the documentation.

Software Not Included in the SDK

- Java 1.4 or higher

Software Included in the SDK

ReadiVoice SDK libraries

- `capi`.jar (API independent core classes used to communicate with the ReadiVoice server)
- `capi`api.jar (ReadiVoice Conferencing API classes)
- `admin`api.jar (ReadiVoice Administration API classes)

Open Source libraries

- `xml`-apis (Apache XML Commons APIs)
- `xerces`Impl (Apache Xerces implementation)
- `commons`-lang (Apache Jakarta Commons Lang)
- `commons`-logging (Apache Jakarta Commons Logging)
- `commons`-cli (Apache Jakarta Commons Command Line interface)
- `commons`-net (Apache Jakarta Commons Net)
- `castor`-xml (Castor Project's XML <-> Java data binding framework)
- JUnit
- JUnitPerf (JUnit test decorators for performance measurement)
- GSBASE (Collection of helper classes for JUnit)
- Ifxjdbc (Informix JDBC Driver)
- iText (Java PDF generation library)
- XMLUnit (JUnit extension for testing XML)
- log4j (Log 4 Java logging framework)
- LogKit

Table 1-1 Software dependencies

This:	Requires:	
Core CAPI	xml-apis xercesImpl commons-lang capibase capiapp	commons-logging castor-xml
Simple Moderator application	Core CAPI	log4j
AcmPins	Core CAPI Acm Shared	log4j commons-cli
Acm Shared library	No dependencies	
Admin API	xml-apis xercesImpl commons-lang capibase	commons-logging castor-xml adminapi

ReadVoice SDK Installation

You can simply copy the provided files to any directory on your system. The directory structure is as follows:

```

rvsdk
  <version-number>
    aapi
    capi
    docs
    sharedjlib

```

On Solaris, you can also run the install file within the provided CAPI files. That install program will install the `rvsdk` directory into `/opt`.

The directory contents include:

- `/aapi` – Admin API, including `.jar` files and `.java` source files for the Java layer of the Admin API.
- `/capi` – Conferencing API, including `.jar` files and `.java` source files for the Java layer of CAPI.
- `/docs` – HTML-based javadocs and PDFs for the Admin API, CAPI, and PSPI, including this developer's guide.
- `/sharedjlib` – The `.jar` files necessary to build and run CAPI applications, including the required third-party libraries and some useful third-party libraries (see "[Software Included in the SDK](#)" above).

Accessing the Documentation

If the SDK package is installed in Solaris, the main SDK page can be accessed by pointing a local browser to the following URL:

```
file://opt/rvsdk/<version-number>/docs/rvsdkindex.html
```

If the SDK is installed from the compressed tar file, the main SDK page can be accessed by pointing a local browser to the following URL:

```
file://<install-directory>/rvsdk/<version-number>/docs/rvsdkindex.html
```

The documentation includes:

- *API Developer's Guide* – This manual, provided in PDF form for online use or printing.
- *Conferencing API Javadoc Reference* – Standard Javadoc HTML reference pages generated from the CAPI code.
- *Conferencing API Reference* – CAPI event and enum reference in PDF form.
- Conferencing API code examples – Source files for the sample Moderator application discussed in this manual.
- *Administration API Javadoc Reference* – Standard Javadoc HTML reference pages generated from the AAPI code.
- *Administration API Reference* – AAPI event and enum reference in PDF form.
- *PSPI Reference (PDF)* – Usage and reference information for the Provisioning Stored Procedure Interface for accessing the subscriber database.

Developing a CAPI Application



Beginning with v. 3.0, the ReadiVoice system no longer supports MAPI.

If you have a MAPI application, use the information in this guide to port that application to CAPI. See [Appendix A](#), “Migrating MAPI Applications to CAPI,” for specific help with this.

Use this chapter and the technical information it references to learn how to write a conferencing application using CAPI. You also need to understand much of the information in this chapter in order to write an administration application using AAPI.

This chapter includes the following information:

- [“Writing Your CAPI Application”](#) on page 8
- [“Understanding CAPI Architecture”](#) on page 8
- [“Understanding CAPI Core Communication Modes: Synchronous and Asynchronous”](#) on page 10
- [“Understanding How Ports Are Used”](#) on page 11
- [“Transport Layer Security”](#) on page 13
- [“CAPI SDK Components”](#) on page 20
- [“Programming Guidelines”](#) on page 29
- [“Using ACM”](#) on page 36
- [“Commonly Used Conferencing Procedures”](#) on page 43
- [“Sample Moderator Application”](#) on page 69

If you’re migrating from MAPI to CAPI, see also [Appendix A](#), “Migrating MAPI Applications to CAPI.”

Writing Your CAPI Application

The following list outlines the steps we recommend for learning how to write CAPI applications.

- 1 Be sure you're familiar with conferencing applications. If you aren't, review [Chapter 1, "Introducing the ReadVoice APIs,"](#) and the additional reading recommended there.
- 2 Install the ReadVoice SDK as described in ["Installing the SDK and Its Documentation"](#) on page 4.
- 3 Familiarize yourself with the documentation and code provided with CAPI. See ["Accessing the Documentation"](#) on page 6.
- 4 Review the events and programming information in ["CAPI SDK Components"](#) on page 20, and the technical information in the PDF and HTML (Javadoc) references.
- 5 Read the ["Programming Guidelines"](#) on page 29 to learn how to write your application.
- 6 Review the ["Sample Moderator Application"](#) on page 69.
- 7 Write your own basic application using the sample as a guideline, to ensure that core features like connections and creating a conference are working.
- 8 Add features to the basic application using the Java classes and events provided. Consult the instructions in ["Commonly Used Conferencing Procedures"](#) on page 43, and the event reference (PDF or Javadoc).

Understanding CAPI Architecture

CAPI-specific

A CAPI application interacts with the ReadVoice system. The ReadVoice documentation provides extensive information about the system; but, we've provided a high-level view of it here, and subsequently how ReadVoice relates to CAPI.

Note: We're presenting a view of the whole architecture to ensure that you know all the processes involved. But, when you write CAPI applications, you don't need to deal directly with bridge or port IDs, or understand how the ODPROC or MPLEX processes work. CAPI is set up to encapsulate the back-end processing details so that the programmer only needs to work with the API itself.

A ReadVoice system includes:

- CACS – A Conference Allocation and Control System (CACS), which is a Sun server. A CACS is the key component of a conferencing system. The CACS stores customer and conference data, routes customers to

conferences, and controls the conferencing bridge(s). When a conference is created, the CACS sends the bridge a conference start package including the conference ID.

Subcomponents or modules include:

- ODPROC – ODPROC stands for on-demand processing. It runs on the CACS and is the master shell process for the ReadiVoice system.

ODPROC is a multi-threaded process that handles every aspect of conferencing over the bridge. Each of its threads handles a different aspect of conferencing. There is a BIS (bridge interface server, one for each bridge) that communicates with the bridge, a call routing thread that handles the logistics of choosing a bridge, reserving ports, etc., as well as several others. Events are received, processed, and sent by ODPROC. If ODPROC is not running, the system can't start new processes.

ODPROC knows where the host computer is, where the conference data is stored, and other configuration information.

- CSC – CSC is a module within the CACS. In previous versions of the API, you needed to deal with multiple components within CACS; in CAPI, your conferencing application communicates only with CSC, which then manages communication with the other CACS modules.
- Apache - An Apache Web server handles the IP connections between the CACS and your conferencing application.
- Bridges - Each system includes one or more bridges. Each bridge provides up to several thousand ports. Each port, or *channel*, on the bridge is equivalent to one phone call, or one person. ODPROC interacts with the bridges. Conferences take place on a bridge, using a group of ports.

The overall architecture is shown in [Figure 2-1](#).

Figure 2-1 CAPI architecture

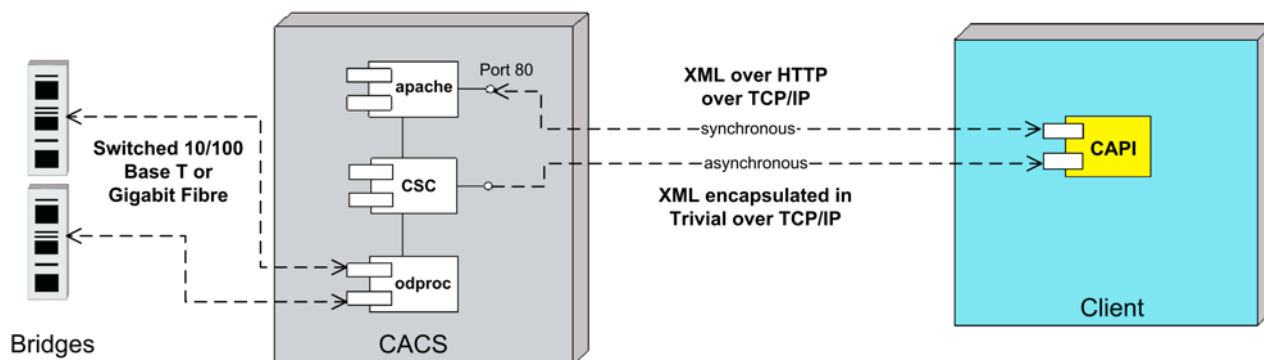


Figure 2-1 shows two lines of communication, one for synchronous and one for asynchronous. They're also known as pull and push mode, respectively. The first, pull, is for quick responses on the same connection. The second, push, is for asynchronous responses on a different connection, typically returning the results of a requested action.

[“Understanding CAPI Core Communication Modes: Synchronous and Asynchronous”](#) on page 10 explains more about the synchronous and asynchronous communication modes.

Understanding CAPI Core Communication Modes: Synchronous and Asynchronous

API-independent

CAPI lets you send requests in two modes: pull (synchronous) and push (asynchronous). The difference is whether the requested action is simply acknowledged as having been sent (pull), or whether a response is immediately sent back (push) satisfying the request.

- Pull mode – The request event is sent, and a reply (often, simply an ACK for success or a NACK for failure) is sent back immediately over the same connection on the same ports.
- Push mode – The request event is sent, and the response is an asynchronous notification sent back later (after the requested action is complete or information has been created) over another connection using different ports.

When you create the first connection from your client to the CACS, you specify the port on which your client expects to receive the asynchronous responses.

Currently only CAPI uses the Push mode; the Admin API (AAPI) uses only Pull mode.

Note: In pull mode the instant reply is sent back over the same connection; in push mode a second connection is created for the asynchronous reply. But, when “a connection” is discussed in CAPI documentation, both are considered one connection.

Understanding How Ports Are Used

Ports for pull mode and push mode are handled differently, as noted in [Figure 2-1](#) on page 9.

Pull Mode and Ports

Sending Port on the Client

The port from which an event is sent is up to you. Typically when you write the conferencing application, you'll request a socket, which will give you a port automatically.

Receiving Port on the Server

The event is received by default on port 80 on the CACS.

Push Mode and Ports

Four ports are used for push (asynchronous) mode:

- The sending port on the client
- The receiving port on the server
- The sending port on the server for the asynchronous response
- The receiving port on the client for the asynchronous response

Sending Port on the Client

This is the same as pull mode. The port from which an event is sent is up to you. Typically when you write the conferencing application, you'll request a socket, which will give you a port automatically.

Receiving Port on the Server

This is the same as pull mode. The event is received by default on port 80 on the CACS.

Sending Port on the Server

The asynchronous response to the event is sent on a port determined by the server; you set this up through ReadiVoice.

Receiving Port on the Client

The asynchronous response to the event is received on the port you specified when you established the connection in the first place. You can use any port you want; if you specify 0 (zero), the system selects an available port.

The asynchronous connection back to the client is created based on the session ID. This lets the response come back to the right client, using the right connection, to the expected port.

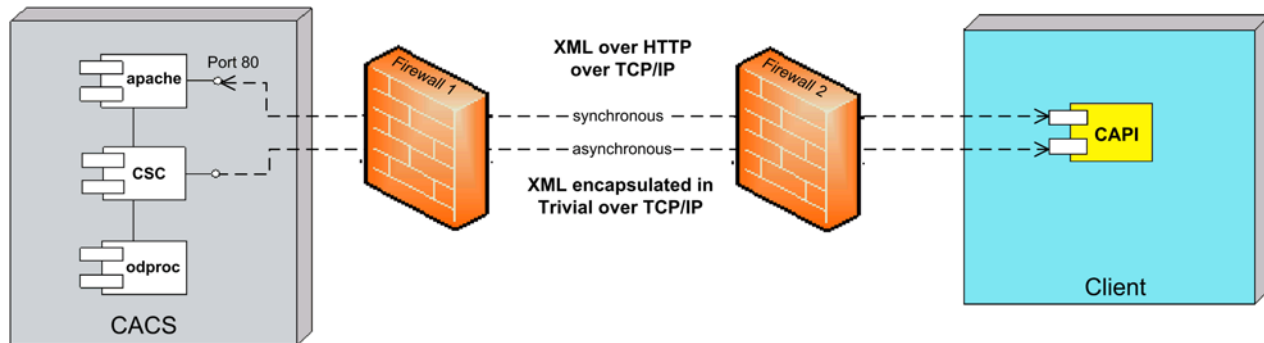
See [“Setting the Port for Asynchronous Responses on the Client”](#) on page 59 or [“Sample Moderator Application”](#) on page 69 for sample code.

Ports and Firewalls

You need to ensure that CAPI communication can get through your firewalls. Typically the pull mode requests to port 80 will get through a firewall without any trouble. Using pull mode doesn't pose any issues regarding firewalls; push mode does involve complications. If you're using firewalls between the client and CACS, you'll need to make some configuration adjustments to accommodate push mode's second connection back to the client.

[Figure 2-2](#) shows a possible firewall configuration, where the client is behind one firewall and the CACS is behind another.

Figure 2-2 Firewalls and CAPI architecture



For push mode in particular, you need to ensure that the asynchronous responses on the additional ports can get through to the client.

Be sure that the data isn't filtered or modified in any way as it goes through the firewalls.

If you want to minimize the number of ports to let communication through, see [“Considerations for Number of Connections to Use”](#) on page 19.

Transport Layer Security

API-independent

The core communication classes allow secure transport between the client application and the Apache web server on the CACS through the use of the `HttpsConnection` class. The connection can be initialized using either the default Java Security SSL socket factory or by providing an SSL socket factory to use.

If a self-signed certificate has been installed in the Apache web server on the CACS, the server certificate must be imported into the Java Security framework of the machine on which the client is installed. To do this:

- 1 Copy the server certificate to the client machine.
- 2 Use the `keytool` utility provided with the Sun Java SDK to create a new keystore with the server certificate:

```
keytool -import -keystore keystore_name -alias trusted_cert_alias -file server_certificate
```

Where:

keystore_name is the name of the keystore to create.

trusted_cert_alias is an alias for the trusted certificate being added.

server_certificate is the full path to the server certificate.

- 3 Copy the newly created keystore to the `lib/security` subdirectory of the Java Runtime Environment installation directory on the client.

If a valid SSL certificate from a well-known, trusted certificate authority has been installed on the Apache web server, the Java Security Framework should already have a public certificate from the certificate authority installed. If it doesn't, follow the above procedure to install the certificate on the client.

For more information on the Java `keytool` utility, please visit the Sun Java website at <http://java.sun.com>.

Once an HTTPS connection has been initialized and passed to the message bus, the client can use it just like a non-secure HTTP connection.

Understanding Connections, Sessions, and Logins

CAPI-specific

This section includes:

- [Overview of Connections, Sessions, and Logins](#)
- [Application, Moderator, and Participant Sessions](#)
- [High-Level Overview of the Conferencing Process](#)
- [Relationships Among Connections, Sessions, and Logins](#)

Overview of Connections, Sessions, and Logins

In ReadiVoice, these terms are used as follows:

- An HTTP *connection* is the networking conduit that enables communication between a port on the client and a port on the server. You specify in the connection creation code whether you're using push or pull mode.

When you're in push mode, there are actually two connections, one from the client to the server, and one back from the server to the client. But they work together, and the code that creates the first one specifies how to create the second one, so the pair are functionally one connection between the client and CACS.

- A *session* is a CAPI process that takes place over the connection. Events are sent through a session.

When your sessions are running over a push mode connection, you get a *session ID*. This ID is stored in a cookie on the client and is used to let the CACS know where to send back the reply value. The session ID distinguishes the requests and responses for one session from those for other sessions.

Multiple sessions can run simultaneously over the same connection. Event requests and responses running over the same connection are handled sequentially by the CACS.

- A *login* creates a session. CAPI has three login types, each with a different set of capabilities. They're described in the next section.

To maintain open connections and sessions, see ["Maintaining Open Sessions and Connections"](#) on page 32.

Application, Moderator, and Participant Sessions

CAPI has three basic types of logins/sessions: Application, Moderator, and Participant. But, there are separate login events for Push Moderator and Pull Moderator, so the effective total is four.

A login creates a session. The login type determines the session type, that is, what capabilities are available within the session, and implicitly what kind of user is controlling the session:

- Participant – An ordinary conference participant user with few capabilities. Use the LOGIN_PART event.

A Participant session is extremely limited in what it can do. It can join the conference and mute its own phone, but very little else.

A Participant session is sometimes also referred to as a One-Click session, since it's the type of session that implements the one-click feature. This feature lets you send a URL to a participant, who clicks it to join the conference.

- Push Moderator – A *subscriber* (conferencing account owner and conference leader or chairperson) with extensive capabilities within the subscriber's own conference, for use in push mode. Use the LOGIN_PUSH_MODERATOR event.

Moderator sessions have extensive control over a single conference. But they can't affect any other conference.

- Pull Moderator – A subscriber with extensive capabilities within the subscriber's own conference, for use in pull mode. Use the LOGIN_MODERATOR event.

- Application – A “super-Moderator” that has all the capabilities of a Moderator, plus the ability to affect multiple conferences. Use the LOGIN_APPLICATION event. Application sessions are push mode.

The Application session provides extended capabilities beyond what a Moderator can do. Its key capabilities are:

- Monitoring multiple conferences at once.
- Receiving a notification whenever a conference starts or stops.
- Running ACM applications.

High-Level Overview of the Conferencing Process

The following steps are for a simple implementation of a conference:

- 1 The CAPI application on the client opens an HTTP connection to the CACS.
- 2 The CAPI application creates a session and logs into the CACS. The login can be an Application or Moderator login type.
- 3 The CAPI application creates a conference for that session; the conference runs on the CACS.
- 4 The CAPI application sends request events to the conference on the CACS, and receives responses back, within the conference's session.

A session ID is used to ensure that requests and responses for a session are tracked correctly.

- 5 When the conference is over, the login controlling the conference ends the conference, ends the session, and logs out of the connection to the CACS.

Relationships Among Connections, Sessions, and Logins

You might think that for each request/response, or for each conference, you create one connection and one session. Actually, you have more flexibility than that, but there are a few restrictions.

Remember, the Application session is for push mode only, but there are both push and pull mode Moderator sessions.

Basic Relationships

- There is a one-to-many relationship between the connection to the server and the sessions that can go across it.
- There is a one-to-one relationship between a login and a session, that is, each successful login creates a session, which is assigned a unique session ID.
- There is a one-to-one relationship between a Moderator session and the conference to be monitored and controlled. That is, a Moderator session is associated with a specific subscriber account and can only work with that subscriber's conference.
- There is a one-to-many relationship between an Application session and the conferences to be monitored and controlled. That is, an Application session is *not* associated with a specific subscriber account and can work with multiple subscribers' conferences.
- There is a many-to-many relationship between sessions and conferences. Multiple Applications, Moderators, and Participants can access a conference at the same time. See the next section.

Session and Conference Combinations

Multiple client applications, or multiple threads of your application, can create multiple sessions that control multiple conferences at the same time. Possible implementations of this include:

- One Application session can control all conferences.
- One Application session can control one or more conferences, while one Moderator session controls a specific conference.
- One Application session can control one or more conferences, while one Moderator session controls a specific conference, and another Moderator session controls another specific conference.
- Two Application sessions can control one or more conferences each, while one Moderator session controls a specific conference, and another Moderator session controls another specific conference.

Session and Connection Combinations

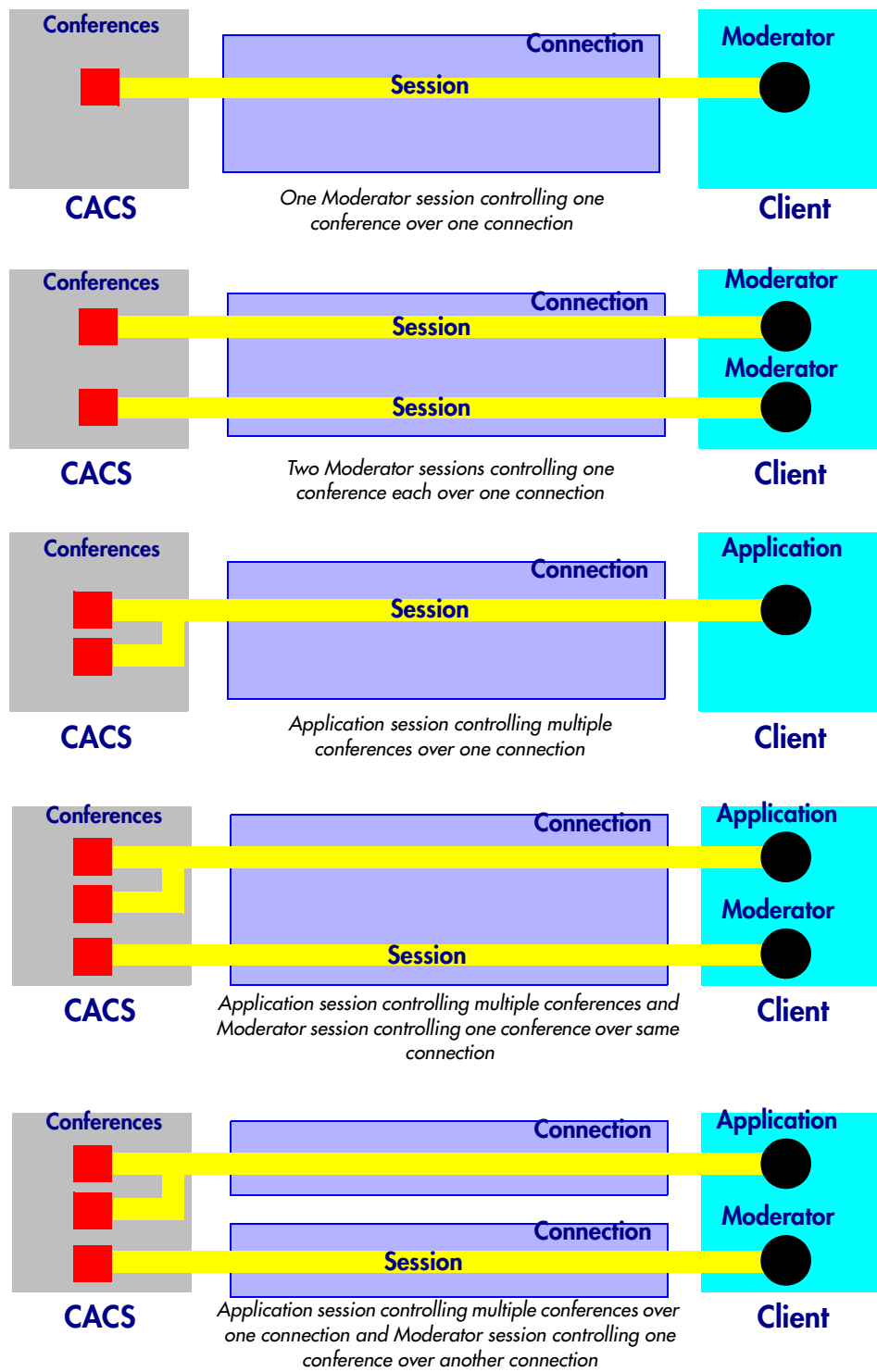
The combinations from the previous section can be grouped over connections in nearly any way you decide:

- All sessions can run over the same connection (for push, the same set of connections, one client to server and the other server to client).
- Some sessions can run over one connection and the rest can run over another.
- Each session can take place on a separate connection.
- Each session can manage multiple conferences (if the session was created by an Application login) or many sessions can manage one conference.

Possible Connection, Session, and Conference Configurations

[Figure 2-3](#) illustrates some possible approaches to connections, sessions, and conferences.

Figure 2-3 Possible approaches to connections and sessions



Benefits and Drawbacks

The choices shown in “[Relationships Among Connections, Sessions, and Logins](#)” on page 16 boil down to a few choices:

- Whether to use more or fewer connections.
- Whether to use an Application session, a Moderator session, or both.

Considerations for Number of Connections to Use

Number of ports kept open

You read in “[Understanding How Ports Are Used](#)” on page 11 that a pull event involves two ports and a push event involves four ports. You also read in “[Ports and Firewalls](#)” on page 12 that you need to ensure for push mode in particular that responses can get back to the client through your firewall.

The more connections you have, the more traffic you need to let through the firewall and the higher the risk that unwanted traffic will get through. This means that if you want to minimize the open ports in your firewalls, use one or very few connections.

Performance

The fewer connections you use, the slower performance will be in general. The events are serialized and processed by the server one at a time within the connection, in the order received. The server can deal with multiple events from multiple connections at the same time, but events within a connection are dealt with sequentially.

Simplicity

Using just one connection for all sessions is simpler.

Considerations for Sessions

An Application session has the ability to manage multiple conferences and to use the CAPI ACM tools to interface with other applications. If these features are important to you, use the Application session.

CAPI SDK Components

CAPI-specific

Review this section to learn about the CAPI events and the CAPI Java utility and event classes.

Understanding the CAPI Events

An event contains information describing some behavior or functionality of the conferencing application or CACS. All control and functionality of CAPI is accomplished by either sending an event to the CACS, or receiving one back from the CACS.

For example, to dial a participant or subscriber, your conferencing application sends a DIAL_CALL event (or class object) to the CACS with the proper data (participant's name, phone number, etc.). The CACS then sends back a DIAL_STARTED event, and then a DIALOUT-CONNECTED event when the person picks up the phone. The reply events contain information such as the conference ID and participant ID used when completing the requested task.

For more information on call flows, see the "Voice Prompts and Call Flows" appendix of the *ReadiVoice Administration and Maintenance Guide*.

Push and Pull Mode

Each event can function in either push or pull mode. Pull mode sends back ACK or NACK responses for success or failure; push mode has a wider variety of responses in the form of asynchronous notifications (see "[Responses Versus Notifications](#)" on page 21). In addition to the replies to the original event, other events can be affected as well. These are noted where appropriate in this document.

Events Can Be Run By Certain Login Types

Not all events can be run by all login types: many can be run by Moderators and Applications, but not Participants. The legal login types for each event are listed in the event reference guide.

Responses Versus Notifications

When you send an event, you can get back a reply. You can also receive notifications. Here's the difference between them:

- In pull mode, you only receive replies. You send an event, and immediately a reply, such as an event containing the requested value, or ACK or NACK, comes back.
- In push mode, you can receive both immediate replies and asynchronous notifications. You might send a request to register to be notified every time a conference starts and stops, for instance:
 - The *reply* to this would tell you that the request was received and you're registered.
 - The *notifications*, later, would tell you that a conference has started or that one has stopped.

Using the Event References

The event references (PDF or Javadoc HTML) included in the SDK contain extensive technical information about how to use each event. Be sure to use them as you write your application.

In the event references, pay particular attention the responses for each event for pull mode and for push mode. Some responses are standard ACK for success and NACK for failure; others send alternate or additional information. Note that NACK is not the failure of the event to get the requested response, but the failure of the event request to be correctly processed.

For instance, if you send a request for an operator (SET_OP_REQ) when an operator isn't available, you might expect a NACK (failure) response. In pull mode, this is the case. But in push mode, the asynchronous response explains the failure to get the operator because one isn't available, and this is expressed not with NACK, but with the OP_REQ_CANCELED event.

The OP_REQ_CANCELED event response returns the enum value CR_NO_OP_AVAILABLE if no operator is available. If the request is canceled by the caller, the event response enum is CR_CALLERS_ACTION.

Understanding the CAPI Java Classes

The Javadoc HTML reference pages were installed with the SDK. Use them to learn more about how and when to use each class, interface, and package.

You can access Javadocs using the main CAPI components navigation page.

Use this section to learn more about the Java classes provided with CAPI.

This manual assumes that you're writing your application using all of the provided Java classes, including the Java event classes generated by Castor from the schema.

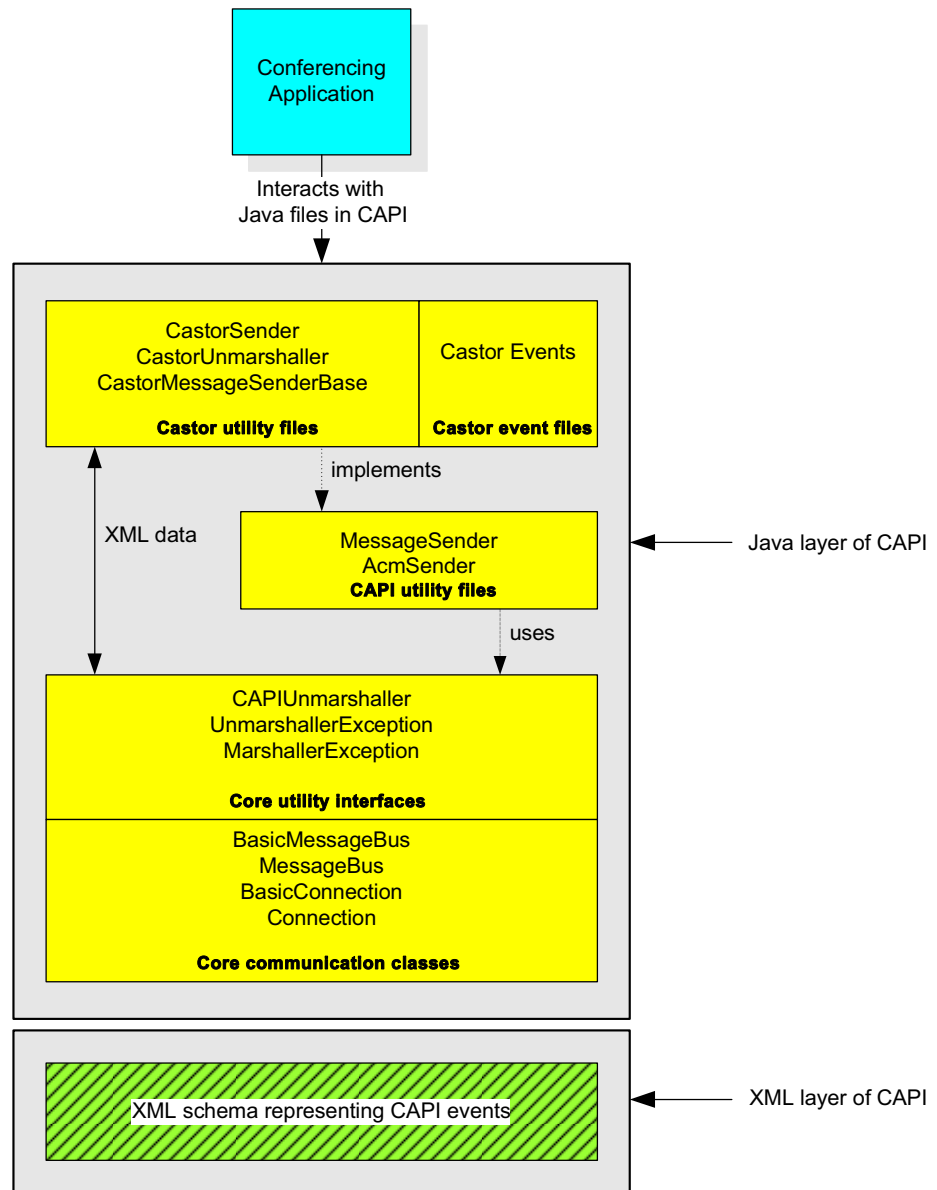
CAPI Java Classes

The new CAPI SDK contains the following components:

- An XML schema defining conferencing events, composing the XML layer of the API. These events are CAPI-specific.
- Five sets of Java classes, built on top of the XML layer, composing the Java layer of the API:
 - Core communication classes, which interact with CACS and the core utility classes. These are generic classes that can be used with either CAPI or AAPI.
 - Core utility classes, generic interfaces and wrappers which isolate the core communication classes from API specifics. These are generic classes and interfaces that can be used with either CAPI or AAPI.
 - CAPI Utilities, which interact with the event classes and implement the core utility interfaces to provide CAPI specific messaging. These classes are specific to CAPI.
 - CAPI Castor utility classes, which interact with the event files and implement the core utility classes. These classes are specific to CAPI.
 - CAPI Castor event files, generated from the CAPI schema using the Castor program (to learn more about Castor, see www.castor.org). These classes are specific to CAPI.

[Figure 2-4](#) shows the different Java CAPI SDK components, and how they relate to each other and the conferencing software you write. A partial list of the Java classes in each category is included.

Figure 2-4 CAPI SDK architecture with sample class names



CAPI Java Class Quick Reference

Table 2-1 provides a quick reference. For technical details, see the Javadocs.

Table 2-1 Java classes provided with CAPI

Category	Java class name	Description
Core Communication (capibase.jar)	AsyncReceiverConnection	An implementation of Connection that uses a trivial protocol. Provides for reading asynchronous data off a unidirectional connection (does not provide a way to write to that connection). Internal read is unbuffered. Trivial protocol is to read 1 integer of data specifying the size of the rest of the data packet, then to read that much more off the connection.
	BasicCookie	Implements the Cookie interface. Usually used to set a cookie on a connection.
	BasicListener	Listener class provides infrastructure for infrastructure for asynchronous/synchronous (pull/push) connections. Contains a ServerSocket that will accept new connections.
	BasicMessageBus	Implements MessageBus. MessageBus objects handle communications with the CACS utilizing the RV Simple XML protocol, and HTTP. If given a Listener, four threads are created: one for reading HTTP events, one for writing HTTP events, one to listen for asynchronous server connections, and one to read from any asynchronous server connections. If not created with a listener only the HTTP reader and writer threads are created.
	Connection	Forms the base for all types of socket based CAPI connections to servers. If not set, will use internal SocketFactory to create a socket if needed, or a factory may be set using the appropriate interface methods.
	Cookie	Cookie provides interface for cookie processing. This interface should be overridden to provide desired functionality.
	HttpConnection	An implementation of Connection that uses the HTTP protocol. Internally uses a BufferedReader for reading data off the connection, and provides for writing data to that connection.
	HttpResponseMessage	Represents the HTTP protocol message received in the server API response.
	HttpsConnection	An extension of the HttpConnection class that adds transport layer security (SSL or TLS) to the connection between the client and server.

Table 2-1 Java classes provided with CAPI (continued)

Category	Java class name	Description
	Listener	A utility interface for network components that listen for new connections.
	MessageBus	A utility interface for main communications.
Core Utility (capibase.jar)	CapiUnmarshaller	HandlerInterface provides interface for handling incoming events. This interface should be overridden to provide desired functionality.
	MarshallerException	Higher-level exception wrapper for any exception thrown during data marshalling. This higher-level exception shields the application from having to deal with the marshalling-specific implementations' exceptions.
	UnmarshallerException	Higher-level exception wrapper for any exception thrown during data unmarshalling. This higher-level exception shields the application from having to deal with the unmarshalling-specific implementations' exceptions.
CAPI Utilities (capiapl.jar)	ACMSender	A utility interface for sending ACM-related events to the CACS. No passed parameter for any method may be null or < 0 unless otherwise specified.
	AddressBookSender	A utility class interface for sending address book related events to the CACS. No passed parameter for any method may be null or < 0 unless otherwise specified.
	MessageSender	A utility interface for sending CAPI events.
CAPI Castor Utilities	CastorAcmSender	Implements ACMSender. A utility class for sending ACM-related events.
	CastorAddressBookSender	Implements AddressBookSender. A utility class for sending address book-related events to the CACS.
	CastorMessageHandlerBase	Implements java.util.Observer. The Castor event specific base for handling all events. Handle methods must be overridden by the application to implement an action to take when a specific event is received.
	CastorSender	Implements MessageSender. The Castor specific utility to send events.
	CastorUnmarshaller	Extends Observable, implements UnmarshallerInterface. The Castor specific implementation to unmarshall data into events.
CAPI Castor Events	All events from schema.	See event reference.

Associating the CAPI Events and the Java Utility Classes

Typically, you'll use the procedures in "[Commonly Used Conferencing Procedures](#)" on page 43 and the event reference section to figure out which events you need to use. From there, though, you need to determine which Java classes to use to implement those features. Use these tips to make the connection between events and Java classes.

Where to Find Event Methods

The methods that send events are in classes and interfaces with Sender in the name. The methods that read, or handle, events, are in classes and interfaces with Handler in the name.

You can also look at "[Sample Moderator Application](#)" on page 69 to see how the code sends and reads events.

Sender methods from the CastorSender class:

- `public void appRegisterConfsActivity(java.lang.String passBack, java.lang.String sessionId)`
Corresponds to the APP_REGISTER_CONFS_ACTIVITY event.
- `public void dialCall(java.lang.String passBack, java.lang.String sessionId, int confId, java.lang.String callTo, boolean promptToConnect, java.lang.String name, java.lang.String partTypeStr, java.lang.String callflowStr, boolean delayConf)`
Corresponds to the DIAL_CALL event.
- `public void loginParticipant(java.lang.String passBack, java.lang.String accessNumber, java.lang.String accessCode)`
Corresponds to the LOGIN_PART event.

Handler methods, from the `CastorMessageHandlerBase` class:

- `public void handle(RVAPIACK message)`
Corresponds to the ACK event.
- `public void handle(RVAPIANNOUNCEMENT_CHANGED message)`
Corresponds to the ANNOUNCEMENT_CHANGED event.
- `public void handle(RVAPICALL_DISCONNECTED message)`
Corresponds to the CALL_DISCONNECTED event.

Names of Java Classes That Castor Generates for Each Event

In addition to the Java utility classes written for CAPI, a Java class is automatically generated from the schema for each event by the Castor program. The name of each automatically generated Java class is based on the event name in the event reference guide, but with some differences. These differences are part of how Castor works and ensure uniqueness for the generated code:

- The names of events are all caps with underscores. Some examples are `CONF_INFO_CHANGED`, `SUBSCR_INFO`, and `CALL_DISCONNECTED`.
- But, the Castor program attaches the namespace to the front of each generated class and type. Thus, when you directly reference the event files generated by Castor, you need to use the name Castor assigns to the Java classes: `RVAPICONF_INFO_CHANGED`, `RVAPISUBSCR_INFO`, and `RVAPICALL_DISCONNECTED`. Here's an example:

```
public void handle( RVAPICONF_INFO_CHANGED message )
```

[Table 2-2](#) covers the naming changes that Castor makes when it creates the Castor Events files.

Table 2-2 Event naming conventions

Item type	Schema name	Castor Java class name	Package	Comments
Events	EVENTNAME Example: ACM_CALL_LEAVE SET_CONTACT_INFO	NAMESPACEEVENTNAME Examples: RVAPIACM_CALL_LEAVE RVAPISET_CONTACT_INFO	events	Complex types go in this directory. RVAPI is a namespace.
Types	EVENTNAME Example: STRING	EVENTNAME Example: STRING	events.types	Naming is identical in name and case.
Enums	EVENTNAME Example: NACK_REASON	NAMESPACEEVENTNAME Example: RVAPINACK_REASON	events	Naming is the same as for events.
Enum values	<i>EVENTNAME</i> Example: NACK_REASON	NAMESPACESIMPLE_ <i>EVENTNAME</i> Example: RVAPISIMPLE_NACK_ REASON	events.types	The enumerated value is wrapped in the generated enum.

Programming Guidelines

This section contains important information about designing and writing your CAPI application.

How the ODPROC and the CSC Work Together

This section explains what happens when ODPROC resyncs with the CSC. It's important that you know how this works to be prepared for the events that are called and sent.

See [“Understanding CAPI Architecture”](#) on page 8 for information on the ODPROC and the CSC.

- Bridge logout – When a bridge logs out (or ODPROC is restarted), each push Moderator session that has an active conference on the bridge receives the BRIDGE_STATUS_CHANGED event with the BS_DOWN value. This notification is also sent to all Application sessions that are registered to receive notifications of conference starts and stops.
- Bridge login – When a bridge logs in (after a logout or ODPROC restart), each push Moderator session that had an active conference on the bridge before the logout receives the BRIDGE_STATUS_CHANGED event with the BS_UP value. This notification is also sent to all Application sessions that are registered to receive notifications of conference starts and stops.
- Bridge login – When a bridge logs in (after a logout or ODPROC restart), each push Moderator session receives CONF_INFO_CHANGED for its conference if it is still active. This notification is also sent to all Application sessions that are registered to receive notifications of conference starts and stops. If the Application session is not registered for that conference's subscription, it receives the CONF_STARTED event.
- Nothing is received for conferences that went away while the bridge was out.
- No participant events are received in any case.

In general, sessions interested in a conference are notified that the conference is active, and they must ask for more information if they need that.

Application Design and Configuration

Review this section before writing your application and adhere to specified information.

Choosing a Programming Language

This document assumes in general that you are programming in Java using the provided Java classes.

We recommend that you use Java and the Java classes provided with CAPI. Java 1.4 or higher is required.

See the Javadocs and [“Understanding the CAPI Java Classes”](#) on page 21 to learn more about using the Java classes provided with CAPI.

If you're programming in another language, you'll need to use the schema as the basis for your program. If the existing extension architecture isn't suitable for your needs, you can use only the schema. For instance, you can use the schema as a foundation for building a translation server application that translates events between the CACS Java API and other languages or platforms.

You can use any language with a socket utility so that you can open the Web server port of the CACS. You can use C++, Perl, Java, or Visual Basic, among others. All events must comply with HTTP 1.1.

Object Model Layer

We recommend that you set up an object-model layer between your code and the code provided by CAPI. This was provided in MAPI, and is likely to be provided in the future by CAPI, but is not currently available.

Storing Configuration Information

The `gui.ini` file and the `Config` object were used in MAPI to store information such as ports and server IP addresses. In CAPI, this file and object are no longer used. You are responsible, for CAPI applications, for determining how to store this information; but, you have the flexibility to select the approach you prefer.

Configuration information you might store depends on your system. But we recommend that you ensure that the following are easily available to your application:

- IP addresses and ports on CACS
- Frequency of the heartbeat you send to keep connections open (see [“Maintaining Open Sessions and Connections”](#) on page 32)
- IDs for ACM applications such as PINS

Programming Fundamentals

Review this section before writing your application and adhere to specified guidelines.

Push and Pull Mode

As stated in [“Understanding CAPI Core Communication Modes: Synchronous and Asynchronous”](#) on page 10, CAPI lets you send requests in two modes: push (asynchronous) and pull (synchronous). You can specify that any given session will be in push or pull mode.

Event Return Values for Push and Pull

The push mode responses to an event (success and failure) are listed in the event reference section. For instance, the `REMOVE_CONTACT_GROUP` event returns `CONTACT_GROUP_REMOVED` for success and `NACK` for failure.

The pull mode responses to an event are also listed. For the same event, `REMOVE_CONTACT_GROUP`, the responses are `ACK` for success and `NACK` for failure. `ACK` simply indicates that the request was received.

`ACK` and `NACK` are common responses for pull mode. But, sometimes information is sent back for the success response. For the event `REQ_CONTACT_GROUP_LIST`, which simply asks for information instead of for a task to be performed, the response for pull mode is `CONTACT_GROUP_LIST` for success and `NACK` for failure (the same responses as for push mode).

If information can simply be retrieved to fulfill a request, pull mode returns the requested value for success. If a task needs to be performed to fulfill the request, pull mode returns an `ACK` for success. Both pull and push return `NACK` for failure.

Choosing Push Versus Pull

Push is a more robust, powerful approach. Pull is used when you’re working with a stateless protocol such as HTTP. Push does require you to manage firewalls to ensure that the response coming back on the second connection can get through. Generally, push is the better choice unless you have specific issues with state or firewalls.

Specifying Push or Pull

The type of connection to the CACS that you create specifies whether you’re using push or pull. For push, you need to create a listener and need to specify the client port that the connection for the response should come back to. For pull, you don’t need either of these components.

The code for creating a connection is in [“Connecting to the CACS”](#) on page 46 and [“Sample Moderator Application”](#) on page 69.

For pull, use a Pull Moderator to manage the conference; for push, use a Push Moderator or Application.

It's possible to create a pull connection and then log in with a Push Moderator, or vice versa; the type of login, not the original connection, determines the session type. But your application won't work correctly if you do this.

Push and Firewalls

The second connection that push mode uses can require specific firewall configuration. See ["Ports and Firewalls"](#) on page 12.

Login Types and Sessions

A session determines what types of capabilities can be used and, implicitly, what type of user is controlling or monitoring the conference(s).

A session and a login function together; a login creates a session.

There are three types of logins: Application, Moderator, and Participant. But, there is both a Push Moderator and a Pull Moderator, so the effective total is four. For detailed descriptions, see ["Application, Moderator, and Participant Sessions"](#) on page 15.

Sessions not only specify the capabilities, and thus the events that can be used, but the session ID is used in push mode to let the server know where to send the response on the asynchronous connection. You get a session ID when you create a push session, and it's stored in a cookie on the client.

Creating a Connection

The code for creating a connection is included in the sample CAPI Moderator application, in the SimplePushModerator class. See ["Sample Moderator Application"](#) on page 69.

Maintaining Open Sessions and Connections

A session takes place over a connection. Both need to stay open to make sure the conferencing application works.

- A connection will timeout by default after four minutes without a session over it. This timeout is not configurable.
- A session will timeout by default after two minutes without any traffic. (You can change this value by editing the `.odprocr` file.) To keep the session open, you need to have some traffic over it. The simplest way to ensure traffic over the session is to use the `SESSION_HEARTBEAT` event.

A code example is shown in ["Maintaining an Open Session"](#) on page 48.

Programming Tips

This section contains more specific, task-related tips to follow when you write your application.

Notes on Conference, Participant, and Subscriber IDs

Conference and Participant IDs Must Be Saved and Sent

MAPI contained an object model, so the conference and participant IDs were saved in the conference object and sender objects. That layer does not exist currently in CAPI, so you need to be sure to save the values, and pass them in when needed.

Here's an example of the subscriberId and conferenceId.

```
/*
 * We need to save these for future communications.
 */
private int subscriberId = -1;
private int conferenceId = -1;
```

Conference and Subscriber IDs

It's important to understand the distinction between SubscrId and ConfId, the subscriber ID and the conference ID. Even though they're typically assigned the same number right now, they're two different items, and you should distinguish between them. In CAPI, they're used consistently as follows:

- SubscrId – Used before a conference has been created, to identify a subscriber.
- ConfId – Used as soon as a conference has been created, to identify a physical conference.

Database Conference Information Updates

CSC is a module within CACS that manages conferences. The information it manages is divided into two types: database conference information, generally static settings used to set up the conference, and runtime information, more dynamic information exchanged during the conference.

When you change one, the same information in the other is not necessarily changed. You could change an attribute for the current conference session, but the same information in the database would remain the same. Typically, there are two events to change a piece of information, one to change it in the conference session and one to change it in the database.

Prerequisites: Connection, Session, Conference, and the Associated IDs

Before you run most tasks, such as dialing a participant, you need to create a connection and log in (start a session). Before you run some tasks, you also need to start a conference.

You also need to be sure that you've saved and can read in the values for the session ID, conference ID, and subscriber ID.

Verifying That Events Have Occurred

Sending a request and receiving an acknowledgment are not sufficient to ensure the action requested by an event was carried out. We recommend that you verify each event is both acknowledged and carried out. For example, if you send a request to start a conference:

- Verify that you receive an ACK, which indicates the system has received the request, found no problems, and will attempt to process the request.
- Poll using REQ_SUBSCR_INFO to verify that ConfRunning is XTRUE (which means that the conference ID returned is valid).

Multiple Requests of Same Event and the Passback parameter

To send multiple requests of the same event, you can change the passback number of each request to make it easy to correlate each acknowledgment (ACK) response with its request. ACK contains a passback field parameter, so when an ACK comes back, you can match it with the correct request.

Here's an example. Dialing participants uses the DIAL_CALL event, and the corresponding dialCall method. The Javadocs show that the syntax for the method is as follows, and the first parameter is passBack.

```
public void dialCall(java.lang.String passBack, java.lang.String sessionId, int confId, java.lang.String callTo, boolean promptToConnect, java.lang.String name, java.lang.String partTypeStr, java.lang.String callflowStr, boolean delayConf)
```

The following example uses the passback number when dialing to differentiate dialing the first participant and the second participant. The ACK response for the first request will contain the value Dial1Passback, and the ACK for the second request will contain Dial2Passback.

```

try
{
    if ( dialCount < 1 )
    {
        log.info("Dialing part one");

        sender.dialCall("Dial1Passback", getSessionId(), getConferenceId(),
            phoneNum1, false, partName1, "PT_SUBSCRIBER", "CF_BLAST",
            false);

        dialCount++;
    }
    else if ( dialCount < 2 )
    {
        log.info("Dialing part two");

        sender.dialCall("Dial2Passback", getSessionId(), getConferenceId(),
            phoneNum2, false, partName2, "PT_PARTICIPANT", "CF_BLAST",
            false);

        dialCount++;
    }
    else
    {
        log.error("all parts already dialed");
    }
}
catch (MarshallerException e)
{
    log.fatal("Marshal error in login, exiting", e);
    shutdown();
}
}

```

Implementing the Conference Security Code

The implementation of the conference security code (when SET_CONF_SECURITY_CODE is sent after JOIN_CONF is sent) has been updated. CAPI implementation changes the order and requires that SET_CONF_SECURITY_CODE be sent before JOIN_CONF. This approach means that SET_CONF_SECURITY_CODE is optional.

Response to Malformed XML and HTTP

Malformed XML and HTTP code is responded to when possible. When it's too malformed to respond to, it's silently ignored.

Dial String Requirements

Input for the dial string (usually a telephone number) can include parentheses and hyphens. For example, (303) 555-1212 would be accepted. CAPI strips the extra non-numeric characters out automatically; it also translates alphabetic characters to telephone numbers so that, for example, A,B, and C are translated to the number 2, based on the standard telephone keypad.

But, you must make sure the following requirements are implemented by programming the user interface or post-processing:

- **Telephone extensions** – Extension must be preceded by a sufficient delay – specified by commas. For example, (303) 555-1212,,,,2000 would successfully dial extension 2000 if it existed.
- **Delay** – Any required delays to allow for the switch to route the call must be added by programming the user interface or post-processing. To add delay, commas (,) need to be inserted into the dial string. Each comma creates a one-second delay. Usually four commas (,,,,) are sufficient.

Using ACM

ACM (Application Control Mode) is an interface that enables external applications to interact with callers or conference participants. This makes it possible for you to develop and implement new functionality and features.

How ACM Works

The ReadVoice ACM controller removes a channel (caller or conference participant) from the normal call flow or conference and temporarily hands it over to an ACM application. Once the ACM application has finished with the channel (and assuming there are no additional ACM applications waiting to take control), the ACM controller puts the channel back into the conference or returns it to the call flow at the point where it was removed.

The ReadVoice ACM controller supports:

- Playing .wav files to a channel (the files must be in a location accessible to the bridge).
- Playing DTMF tones to a channel.
- Playing verbal representations of a string of numbers or a list of digits to a channel.
- Collecting DTMF digits from a channel.
- Writing CDR data to the database in the form of <key>:<value> pairs.
- Facilitating communication between a ReadVoice client and an ACM application.

In addition to these ACM-specific features, an ACM application logs in as an Application user and can use all of the CAPI functionality available in an Application session.

Caution!

If your application performs additional processing-intensive functions (such as accessing an external database), it should be run on a server other than the CACS in order to avoid negatively impacting the core ReadiVoice conferencing functions.

Two Types of ACM Applications

You can develop two kinds of ACM applications:

- **Call flow (or pre-conference) applications** are designed to interact with callers before they reach the conference. The ACM controller removes callers from the inbound call flow for an interactive session with one or more external applications. This happens automatically and can't be manually initiated or skipped.

The ACM controller removes callers from the call flow just before they're placed into conference. This is the only point in the call flow at which ACM applications can interact with callers.

ACM supports two pre-conference modes, one for interacting with all callers and one for interacting only with the subscriber. You specify the appropriate mode for your application when you set up the application in the ReadiVoice System Administration interface.

The ReadiVoice SDK includes a sample call flow application named ACM PINs. It prompts each caller for a PIN, verifies it, and writes it to the CDR.

- **In-conference applications** are designed to interact with participants who are in conference. The ACM controller removes participants from the conference for an interactive session with the external application.

Unlike call flow applications, these applications aren't invoked automatically. The ACM controller takes control of the channel only in response to a CAPI, Moderator, or DTMF command (depending on implementation) and turns it over to the external application linked to that command.

Basics of ACM Implementation

Before you can enable an ACM application, the ReadiVoice system has to be set up to use it. See ["Setting Up an ACM Application in the ReadiVoice System"](#) on page 40.

Once the system is configured properly, the ACM application can:

- Log in to create an Application session.
- Register its existence, using the ACM_REGISTER event. The application's ID number must match the one assigned to it in the ReadiVoice system.

- Send a heartbeat (quick data packet) every few seconds. Failure to do so will result in the application being unregistered. The timeout period before the application is unregistered is specified in the `.odprocr` file. Refer to the *ReadiVoice Administration and Maintenance Guide* for information on configuring the `.odprocr` file.

If the ACM application is unregistered, callers will be totally unaware of its existence, even if the subscriber has selected the option to use it.

Call flow (pre-conference) ACM applications have a priority number associated with them (see [“Setting Up an ACM Application in the ReadVoice System”](#) on page 40), which determines the order in which multiple applications get control of a channel. You can use this capability to chain together multiple ACM applications, which enables you to develop smaller, simpler applications with more opportunities for customization.

ACM Events

The ACM-related command events are:

`ACM_CALL_ENTER` – Used by Moderator or Application sessions to place a specific channel into ACM mode and have control passed to an ACM application.

`ACM_REGISTER` – Used by an Application session to become an ACM application.

`ACM_UNREGISTER` – Used by an ACM application to inform the ReadVoice system that it's no longer available.

`ACM_CALL_LEAVE` – Used by an ACM application to pass control of a channel back to the ACM controller. The controller returns the channel to the point from which it entered ACM mode (call flow or conference).

`ACM_PLAY_WAV` – Used to play a `.wav` file, DTMF tones, or a spoken number or digits to a channel.

`ACM_GET_DTMF` – Used to collect DTMF tones from a channel.

`ACM_CDR` – Used to write a CDR record to the CDR database as a `<key>:<value>` pair.

`ACM_HEARTBEAT` – Sent by an ACM application to inform the ReadVoice system that it's still present and available.

`ACM_QUERY` – Used to facilitate communication between a ReadVoice application and an ACM application.

`ACM_QUERY_RESP` – Used to facilitate communication between a ReadVoice application and an ACM application.

`ACM_GET_OPTIONS` – Used by any Moderator or Application session to see what ACM options are provisioned for a specific subscriber.

For full descriptions of events and parameters, see the *Conferencing API Reference* (PDF) or *Conferencing API Javadoc Reference* (HTML).

Communications Between ReadVoice and ACM Applications

ReadVoice and ACM provide a convenience feature that facilitates communication between ReadVoice clients and ReadVoice-developed ACM applications. This feature makes direct connections between the ReadVoice clients and ACM applications unnecessary by having the ReadVoice application itself act as a proxy between the two.

A ReadVoice client (a moderator, for example) sends ACM_QUERY events to the ReadVoice system with the ID of an ACM application. If that ACM application is active, then the ReadVoice system sends the query event to it. The ACM application parses it and responds with an ACM_QUERY_RESP. The ReadVoice system receives this response event and returns it to the original requesting ReadVoice client.

By using this mechanism, and the ReadVoice NACK/error code infrastructure, you don't have to add code to an application in order for it to interact with and monitor partnered ACM applications.

An example of how this works can be seen with the ACM PINs sample application. The ACM PINs application collects DTMF digits from a channel and stores them. A Moderator can then send a query to the ACM application, by way of the ReadVoice system, asking for the DTMF sequence collected from a specific channel. It will get back an ACM_QUERY_RESP event with either the desired information from the ACM PINs application or some type of error code (for example, the ACM application is not currently active and the query could not be delivered).

ACM CDRs

CDRs are manually written using the ACM_CDR event. This event contains an enumeration parameter specifying the type of ACM application it originated from (in-conference or pre-conference), the unique ID of the originating ACM application, what conference the channel belonged to, along with channel-identifying information, time stamps, and a user-defined <key>:<value> pair string.

This user-defined string is a single string with the key and the value parameters separated by a colon (:). All of this information is written into the cdr_acm_data table in the ReadVoice CDR database. The user-defined string is parsed on the colon and is inserted into the table as:

```
acm_name = <key>
acm_value = <value>
```

The key and the value each have a maximum size of 50 characters.

In addition to this optional and more user-defined CDR, the regular cdr_post_state record for a channel also show when it entered and left ACM mode. The ACM data will be stored until purged.

Setting Up an ACM Application in the ReadVoice System

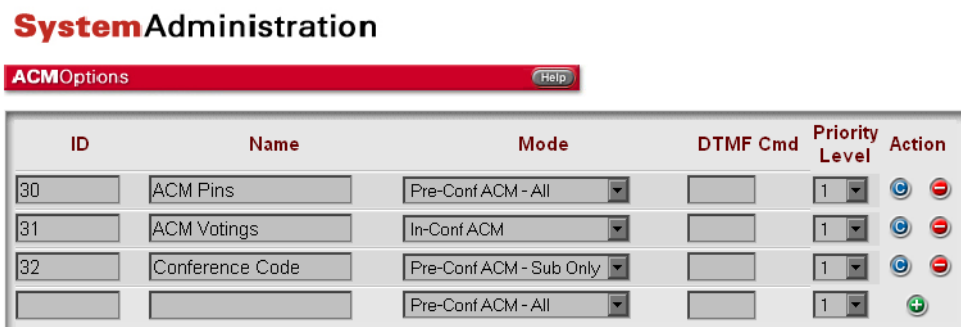
To set up an ACM application, you must add it to the ReadVoice system in the ACM Options page of the System Administration interface. If it's an in-conference application, you may also need to edit the `ive.ini` file to enable the DTMF command for invoking the application.

Adding an ACM Application in System Administration

To add an ACM application to the ReadVoice system:

- 1 In the System Administration navigation bar, click **ACM Options**.
The ACM Options page appears. If any ACM applications have already been added, they appear in the list. Below that, a set of blank input fields and an associated **Add** button let you add ACM applications.

Figure 2-5 The ACM Options page in ReadVoice System Administration



- 2 In the **ID** field, enter the unique ID number that this application will use to identify itself when it registers with ReadVoice.
- 3 In the **Name** field, enter the name as you want it to appear in the Provisioning interface.
- 4 Under **Mode**, select the ACM mode for this application from the list.
- 5 If the mode is **InConf ACM** and the application can be initiated by a touchtone command in conference, enter the command in the **DTMF Cmd** field. Otherwise, leave this field empty.
- 6 Under **Priority Level**, choose a priority level for this application. A higher number bestows a higher priority. This setting only applies to call flow (pre-conference) applications.

If there are multiple ACM applications with the same mode (**Pre-Conf ACM - All** or **Pre-Conf ACM - Sub**), this setting determines the order in which they are invoked.

For example, assume that application A has priority 1, and application B has priority 2, and that both are **Pre-Conf ACM - All** mode. When the

ACM controller removes an inbound call from the call flow, it turns the channel over to application B first.

Assigned priorities only apply within the modes. A **Pre-Conf ACM - Sub** application always precedes a **Pre-Conf ACM - All** application.

7 Click the **Add** button.

ReadiVoice confirms that the ACM application has been added and provides a link for returning to the ACM Options page.

8 Click the link to reload the ACM Options page.

The new entry appears at the end of the list, above the blank row.

9 To add additional ACM applications, repeat the preceding steps.

Defining DTMF Commands for ACM Applications

When you add an InConf ACM application on the ACM Options page, you can assign it a DTMF command. But to enable that command, you must also define it in the [DTMF_CMDS] section of the `ive.ini` file for the system.

For general rules and guidelines and detailed procedures, see “Customizing Touchtone Commands” in the *ReadiVoice Administration & Maintenance Guide*. But, contrary to the warning there, to add an ACM application command, you must add a whole new row, including the command name to the right of the equals sign, to the [DTMF_CMDS] section.

The valid command names for activating ACM mode are `eCMD_ENTER_ACM_1`, `eCMD_ENTER_ACM_2`, `eCMD_ENTER_ACM_3`, and so on through `eCMD_ENTER_ACM_9`.

These commands are all functionally identical, and you could simply use `eCMD_ENTER_ACM_1` for every ACM DTMF command you define. Using a different command name for each is purely a matter of clarity and cosmetics.

[Figure 2-6](#) shows an example of a standard [DTMF_CMDS] section with two ACM application commands added.

When a conference participant presses either #4 or #9, the ACM controller removes the channel from conference and turns it over to the application identified by the command. If there is none (for instance, no application has been added to the ACM Options page with that command, or the application isn't registered), then the ACM controller simply returns the channel to conference.

Figure 2-6 Adding ACM DTMF commands to the `ive.ini` file

```
[DTMF_CMDS]
** = eCMD_HELP
*0 = eCMD_OP_REQUEST
00 = eCMD_CONF_REQUEST
*1 = eCMD_DIAL_OUT
*2 = eCMD_CONF_RECORD
*3 = eCMD_CONF_CONFIG
*4 = eCMD_CONF_LOCK
*5 = eCMD_CONF_UNLOCK
*6 = eCMD_MUTE
*7 = eCMD_UNMUTE
*8 = eCMD_CONF_CONTINUE
*9 = eCMD_ROLL_CALL_PRIV
*# = eCMD_PART_COUNT
## = eCMD_MUTE_ALL
99 = eCMD_UNMUTE_ALL
#1 = eCMD_LISTEN_ONLY
#2 = eCMD_UNLISTEN_ONLY
#5 = eCMD_WR_PROCESS
#4 = eCMD_ENTER_ACM_1 // #4 will now activate ACM mode for a channel pressing it.
#6 = eCMD_WR_ANNOUNCE_TOGGLE
#8 = eCMD_CONF_END_CONF
#9 = eCMD_ENTER_ACM_2 // #9 will now activate ACM mode for a channel pressing it.
1 = eCMD_WR_PART_TO_CONF
2 = eCMD_WR_PART_IGNORE
3 = eCMD_WR_PART_DISCONNECT
* = eCMD_WR_SUB_TO_CONF
[] // DTMF_CMDS
```

Conference User Data Feature

A new feature has been added to ReadVoice that, although not ACM-specific, may be useful for developers of ACM applications. This feature adds a “user data” string field to active conferences in the ReadVoice system.

Any ReadVoice client with an Application or Moderator login (including ACM applications) can set the user data field for a conference by using the SET_CONF_USER_DATA event. This user data string persists through resynchronizations and failovers, and remains available until the conference ends.

The UserData parameter has been added to the CONF_INFO event, so the user data string is available to any ReadVoice client that receives the CONF_INFO event for the conference.

When a ReadVoice client sets the user data for a conference, ReadVoice sends immediate CONF_INFO notifications to all clients registered for the conference, as well as to the original setting client (as a direct notification).

Once a conference’s user data field has been set, all CONF_INFO notifications for that conference contain this user data string until it’s either reset by another SET_CONF_USER_DATA command or the conference ends.

Commonly Used Conferencing Procedures

This section contains a limited list of the more commonly used procedures. To write any procedure in CAPI, you can use the event reference section to find an event that performs the task. Read the descriptions and other information to determine how to use it.

Additional Sources of Information

For more information on how the features in this section work, refer to the ReadVoice documentation suite. The following in particular are recommended:

- *ReadVoice Provisioning Guide*
- *ReadVoice Operator Guide*
- *ReadVoice Subscriber Guide*
- *ReadVoice Administration and Maintenance Guide*

Procedure Sequence

When you start a conference, you need to perform certain steps in a certain order. For instance, you need to create a connection to the CACS, then log into it, then create a conference, then dial participants, and so on.

The procedures in this section are presented in the order in which they should be used.

- 1 Perform [“Initial Procedures”](#) on page 45 in the order in which they are listed.
- 2 Perform [“Conference Procedures”](#) on page 52 only after you’ve completed the initial procedures.

If a procedure logically needs to take place when a conference is already running, be sure you’ve first performed [“Starting a Conference”](#) on page 52. Some tasks don’t need a conference running first; you wouldn’t need to be in a conference for [“Registering the Application Login for Conference Start and Stop Notifications”](#) on page 53, for instance. A Moderator also could simply log in, make some changes to settings, and log out, without creating a conference. But, before you perform [“Dialing Participants”](#) on page 55, you must have a conference running already so that there was a conference for them to join.

- 3 Perform [“Conference-End Procedures”](#) on page 67 to end a conference.

A typical order for a set of tasks in a simple application is:

- 1 Create a connection
- 2 Log in
- 3 Start a conference
- 4 Dial participants
- 5 End the conference
- 6 Log out

Procedure Notes

This section covers important information about what data is provided, prerequisites, and other tips.

Content of Procedures in This Section

The appropriate information for each event depends on whether it’s an administrative procedure such as creating a connection, or a simpler CAPI procedure that uses a straightforward event and response. We’ve provided one or the other for each procedure, and both where possible.

Please refer also to the [“Sample Moderator Application”](#) on page 69 to see working CAPI sample code.

Push Mode Versus Pull Mode

CAPI has two modes, push and pull. (See [“Understanding CAPI Core Communication Modes: Synchronous and Asynchronous”](#) on page 10 and [“Push and Pull Mode”](#) on page 31.) The responses you’ll get for sending an event in push mode are different from the ones in pull mode. We’ve written these procedures for push mode, since push mode is more robust and somewhat more complex, and summarized the differences.

If you want to write a procedure in pull mode, read the push procedure, and then make changes as necessary for the different pull responses, which are listed in the event reference guide.

Application Login Prerequisites

For any procedures involving the Application login, be sure that the Application ID and password are set up in the REDIvoice system. Refer to the *REDIvoice Administration and Maintenance Guide* for more information.

Application login is for push mode only, so be sure to create a push mode connection to the server.

Initial Procedures

This section includes procedures you perform in order to get the conference started.

Setting Up Logging

Apache Log4J and Apache Commons logging are available and included in the SDK.

Sample Code

Here’s an example of setting up logging. Refer to [“Sample Moderator Application”](#) on page 69 to see the sample code in context.

```
import org.apache.commons.lang.Validate;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.PatternLayout;
import org.apache.log4j.RollingFileAppender;
import org.apache.log4j.TTCCLayout;
```

...

```
/*
```

```
* Set up the Log4J system
*/
RollingFileAppender appender = new RollingFileAppender(new TTCCLayout(),
    "SimplePushModerator.log", true);

PatternLayout pl = new PatternLayout();
pl.setConversionPattern("%d{ISO8601} %-5p [%t]: %m%n");

appender.setLayout(pl);

BasicConfigurator.configure(appender);
}
catch( IOException e )
{
    BasicConfigurator.configure();
}

Logger logger = Logger.getLogger("com");
logger.setLevel(Level.toLevel(logLevel));
```

Connecting to the CACS

You need to create a connection to the CACS before you can do anything else in the application. You specify in the connection code whether the connection will be push or pull.

The connection is a simple `HttpConnection`; for push, you also need to create a listener and specify the client port that the asynchronous responses should come back to.

Sample Code

Refer to [“Sample Moderator Application”](#) on page 69 to see the sample code in context.

```
public SimplePushModerator( String remoteHost, int remotePort, String
    localAsyncHost,
    int localAsyncPort )
{
    Validate.notEmpty(remoteHost);
    Validate.isTrue(remotePort > 0);
    Validate.notEmpty(localAsyncHost);
    Validate.isTrue(0 <= localAsyncPort);
```

```
// The following listener code is used only for push connections.
try
{
    /*
     * This is the listener for asynchronous event communication. When the server
     * contacts for asynchronous communication on the specified port, this object
     * will create a connection for the MessageBus.
     */
    listener = new BasicListener(localAsyncPort);

// The following cookie code is used only for push connections.

    /*
     * The server uses the cookie to find out the details on how to establish
     * the asynchronous connection with us.
     */
    Cookie cookie = new BasicCookie(localAsyncHost, listener.getLocalPort());

// The rest of the code is used for both push and pull connections.

    /*
     * We need to establish a basic HTTP connection to send commands and queries.
     * In push-mode, responses will come back over the asynchronous connection.
     */
    httpConnection = new HttpConnection("SimpleModHttp", remoteHost, remotePort,
        true,
        Connection.DEFAULT_URL, cookie);

    /*
     * This is a handler for all events from the server/CACS.
     */
    handler = new SimplePushModeratorHandler(this);

    /*
     * This objects handles conversion of the raw data off a connection into
     * POJO representations of the CAPI events.
     */
    unmarshaller = new CastorUnmarshaller(handler);

    /*
     * Once the CAPIUnmarshaller, the command sending HTTP connection, and the
     * asynchronous connection listening service are set up, we give them to a
     * MessageBus so it can use them to finish any further connection setup, and
     * begin those communications as needed.
     */
    messageBus = new BasicMessageBus(unmarshaller, httpConnection, listener);
```

```

        /*
        * This is the utility class that provides methods for sending the CAPI events
        * to the server/CACS. These methods will handle marshalling of the passed
        * arguments into the POJO representations of those events. It will then
        * marshall those POJO representations into raw data and use the
        * provided MessageBus to send to the server/CACS.
        */
        sender = new CastorSender(messageBus);
        shouldRun = true;
        /*
        * Start the main thread
        */
        start();
    }
    catch( IOException e )
    {
        log.fatal("Error in setup, exiting", e);
        shutdown();
    }
}

```

Maintaining an Open Session

Once you've created a connection and a session, you need to have traffic going to keep both open. The session will timeout by default after two minutes without traffic; the connection will timeout after four minutes without a session.

Event

SESSION_HEARTBEAT

Any event can be used to maintain the connection by requesting and receiving information regularly; however, SESSION_HEARTBEAT was created for that purpose.

Response

Pull: N/A

Push: SESSION_HEARTBEAT_ACK

Prerequisites

["Creating a Connection"](#) on page 32.

Sample Code

Refer to [“Sample Moderator Application”](#) on page 69 to see the sample code in context.

```
/*
 * Application is responsible for sending heartbeats. This counter
 * is just used to limit the number of heartbeats to reduce network traffic.
 */
private static final int HEART_BEAT_COUNTER_DEFAULT = 30;
private int heartBeatCounter = HEART_BEAT_COUNTER_DEFAULT;

public synchronized void heartBeat()
{
    if( heartBeatCounter == 0 )
    {
        log.debug("sending connection heartbeat...");
        try
        {
            sender.sessionHeartbeat("SimpModPassback", sessionId, sHeartBeatSeq++);
        }
        catch( MarshallerException e )
        {
            e.printStackTrace();
            System.exit(1);
        }
        heartBeatCounter = HEART_BEAT_COUNTER_DEFAULT;
    }
    else
    {
        heartBeatCounter--;
    }
}
```

Logging Into the CACS

Event

LOGIN_APPLICATION, LOGIN_PUSH_MODERATOR,
LOGIN_MODERATOR, LOGIN_PART

Response to LOGIN_APPLICATION

Pull: N/A

Push: ACK for success, NACK for failure.

Response to LOGIN_PUSH_MODERATOR

Pull: N/A

Push: SUBSCR_INFO for success, NACK for failure.

Response to LOGIN_MODERATOR

Pull: MODERATOR_SESSION_INFO for success, NACK for failure.

Push: MODERATOR_SESSION_INFO for success, NACK for failure.

Response to LOGIN_PART

Pull: N/A

Push: PART_SESSION_INFO for success, NACK for failure.

Prerequisites

[“Connecting to the CACS”](#) on page 46, [“Maintaining an Open Session”](#) on page 48.

Be sure you create a push mode connection if you'll use the Application or Push Moderator login, and a pull mode connection if you'll use the Pull Moderator login.

Notes

Use the Pull Moderator login for pull mode; use the Application or Push Moderator login for push mode.

You need to log into CACS before you can run any conferencing tasks. Use an Application or Moderator login, since the Participant has very few capabilities.

The Application login is new in this release of CAPI. Use the Application login to monitor multiple concurrent conferences, or for ACM functionality.

Sample Code

See [“Sample Moderator Application”](#) on page 69 to see the sample code in context.

```
/**
 * Sends a push mode Moderator login.
 */
public synchronized void login()
{
    log.debug("log in");

    if ( !shouldRun )
    {
        return;
    }

    Validate.isTrue(!loggedIn);

    try
    {
        sender.loginPushModerator("PassbackLogin", accessNumber, accessCode,
            subPassCode);
    }
    catch (MarshallerException e)
    {
        log.fatal("Marshal error in login, exiting", e);
        shutdown();
    }
}
```

Conference Procedures

This section contains the majority of the procedures; they can be performed once the initial procedures are completed.

Starting a Conference

Event

START_CONF

Response

Pull: ACK for success, NACK for failure.

Push: CONF_INFO_CHANGED with CIR_CONF_STARTED for success, NACK for failure.

Prerequisites

[“Initial Procedures”](#) on page 45.

Notes

A conference can have multiple Application, Moderator, and Participant logins monitoring or controlling it.

Sample Code

Refer to [“Sample Moderator Application”](#) on page 69 to see the application.

```
public synchronized void startConf()
{
    log.debug("startConf");
    if( !shouldRun )
    {
        return;
    }
    Validate.isTrue(loggedIn);
    Validate.notEmpty(sessionId);
    Validate.isTrue(!confStarted);
    try
```

```

    {
        sender.startConference("PassbackStartConf", getSessionId(),
                               getSubscriberId());
    }
    catch( MarshallerException e )
    {
        log.fatal("Marshal error in login, exiting", e);
        shutdown();
    }
}

```

Saving Conference and Subscriber ID

MAPI contained an object model, so the conference and participant IDs were saved in the conference object and sender objects. That layer doesn't exist currently in CAPI, so you need to be sure to save the values once you've created the conference and pass them in when needed. (The conference ID is passed back when you create a conference; take that value and save it.)

Sample Code

Here's an example of the subscriberId and conferenceId. Refer to ["Sample Moderator Application"](#) on page 69 to see the code in context.

```

/*
 * We need to save these for future communications.
 */
private int subscriberId = -1;
private int confenceId = -1;

```

Registering the Application Login for Conference Start and Stop Notifications

The Application login was created as the mechanism for several new features in CAPI; the primary one is to receive conference start and stop notifications for multiple concurrent conferences.

Event

APP_REGISTER_CONFS_ACTIVITY to begin getting notifications.

APP_UNREGISTER_CONFS_ACTIVITY to stop getting notifications.

Response

Registering: ACK for success, then one CONF_STARTED for each active conference; NACK for failure.

Unregistering: No response.

Prerequisites

“[Initial Procedures](#)” on page 45 and “[Application Login Prerequisites](#)” on page 45.

Notes

The response to this procedure is ACK to indicate that the registration was received and performed successfully. This is followed by a CONF_STARTED notification for each active conference. After that, when a conference starts or stops, CONF_STARTED or CONF_ENDED will be sent as a notification.

This procedure is different from an Application simply registering for a conference, using the APP_REGISTER_CONF and APP_UNREGISTER_CONF events.

The Application login isn't associated with a particular conference, since its primary functions are to monitor multiple conferences and ACM applications.

Application is a push login, so this procedure can be used only in push mode.

See “[Application, Moderator, and Participant Sessions](#)” on page 15 for more information about Application.

Setting Conference Security

Setting a conference security code is optional in CAPI.

Event

SET_CONF_SECURITY_CODE

Response

Pull: ACK for success, NACK for failure.

Push: CONF_SECURITY_CODE_CHANGED success, NACK for failure.

Prerequisites

The implementation of the conference security code (when SET_CONF_SECURITY_CODE is sent after JOIN_CONF is sent) has been updated. CAPI implementation changes the order and requires that SET_CONF_SECURITY_CODE be sent *before* JOIN_CONF.

See also “[Procedure Sequence](#)” on page 43.

Notes

The security code must be implemented before participants join a conference. When you implement the security code first, the participants who then join the conference will be prompted for the security code. If you do it after they've joined, then they're already in the conference, so they won't be prompted to enter the security code.

Security applies to anyone dialing in, and to some participants reached by dialing out. If the dialout is set to the Default, then it's assumed that the person dialing out is a subscriber or moderator, and therefore the person being called doesn't need to be asked for the security code. For a One-Click dialout, however, the participant being called is asked for the security code.

Dialing Participants

Note: For more information on call flows, refer to the "Voice Prompts and Call Flows" appendix of the *ReadiVoice Administration and Maintenance Guide*.

Dialing participants so they can join the conference is one of the core tasks in a conferencing application. CAPI provides several ways to dial participants:

- Standard dialing – Participants are placed into call flows as if they had dialed into the system. They enter a conference code, record their name, and so on.

One of the options with standard dialing is to dial out to a recording device in addition to the participants; the device records the conference.

- One-click dialing – Participants are placed into a conference as if they had dialed into the system. In a One-Click Conference, participants have a link or icon, such as in an e-mail invitation, on which they can click. The ReadVoice system identifies the conference from the link's URL and returns a Web page asking for the participant's phone number. When the system gets this number, it calls and puts the connection in conference (or on music hold, if the subscriber isn't present and Quick Start is off). The subscriber can use the same link, since the Web page provides a checkbox to indicate subscriber and a field for entering the subscriber password.
- Blast dialing – A group of participants are dialed at once; this is a good way to contact a large group of conference participants. When participants pick up the phone, they're automatically joined to the conference.
- Instant dialing – Participants are placed directly into conference without prompts. A participant is dialed and when he or she picks up the phone, is instantly joined to the conference.

Event

DIAL_CALL

- Standard dialing – Use the CF_DEFAULT enum for the CallFlow parameter
- Blast dialing – Use the CF_BLAST enum for the CallFlow parameter
- One-click dialing – Use the CF_ONECLICK enum for the CallFlow parameter
- Instant dialing – Use the CF_INSTANT enum for the CallFlow parameter
- Dialing out to a recorder – Use the CF_DEFAULT enum for the CallFlow parameter, set the Part_Type to PT_RECORDER.

Response

Pull: ACK for success, NACK for failure.

Push: DIAL_STARTED or DIALOUT-CONNECTED for success; NACK for failure.

Prerequisites

See “[Procedure Sequence](#)” on page 43.

General Notes

Anytime a participant's status changes, such as when they join a conference, the PART_INFO_CHANGED message is sent. When a participant joins the conference, the message's value is PIR_CONNECTED_TO_CONF, the participant's new state. If you want to track the number of participants in the conference, you can read the PART_INFO_CHANGED message value and increment the number of people in the conference.

Notes on Conference Recording

Refer to the *ReadiVoice Administration and Maintenance Guide* for information on setting up a system for conference recording. You need to set up values in the .odprocr file, including the recorderPhone value. The recorderPhone value is used by default unless you pass a different phone number when you send the DIAL_CALL event.

Notes on Joining Participants for Different Call Flow Types

There are four types of enums for the call flow type of the CALL_FLOW event. The simple part of handling the code is to specify the correct enum type, as in this example:

```
sender.dialCall("DiallPassback", getSessionId(),  
getConferenceId(), phoneNum1, false, partName1, "PT_SUBSCRIBER",  
"CF_INSTANT", false);
```

The main difference is more complex, and involves how you handle the code. Depending on the call flow type, you need to send a JOIN_PART_TO_CONF event for that participant to put them into conference.

- If you want the user to complete an action to join the conference, such as clicking a button, send JOIN_PART_TO_CONF event for that participant once they complete the action. This applies to the standard call flow for which you use the CF_DEFAULT enum.
- If you want the user to be automatically included in the conference, use the auto-joining call flow types: blast, one-click, and instant. For these, you don't need to send a join event since that functionality is already built into the listed events.

Whether you send the join event explicitly or the join happens automatically because of the call flow, the response is a `PART_INFO_CHANGED` event containing one of several enum values. If the participant joins the conference, the value is `PIR_CONNECTED_TO_CONF`, the participant's new state.

Sample Code for Blast Dial

See ["Sample Moderator Application"](#) on page 69 to see the sample code in context.

```

/**
 * Dials up to two participants. Session must be active (logged in), and the
 * conference started. Each call to this function will dial a participant
 * until the maximum/desired number of dial attempts have been made, then it
 * will just log an error and return.
 */
public synchronized void dialPart()
{
    if ( !shouldRun )
    {
        return;
    }
    Validate.isTrue(loggedIn);
    Validate.notEmpty(sessionId);
    Validate.isTrue(confStarted);
    Validate.notEmpty(phoneNum1);
    Validate.notEmpty(phoneNum2);
    try
    {
        if ( dialCount < 1 )
        {
            log.info("Dialing part one");

            sender.dialCall("Dial1Passback", getSessionId(), getConferenceId(),
                phoneNum1, false, partName1, "PT_SUBSCRIBER", "CF_BLAST",
                false);
            dialCount++;
        }
        else if ( dialCount < 2 )
        {
            log.info("Dialing part two");
            sender.dialCall("Dial2Passback", getSessionId(), getConferenceId(),
                phoneNum2, false, partName2, "PT_PARTICIPANT", "CF_BLAST",
                false);

            dialCount++;
        }
    }
    else

```

```

        {
            log.error("all parts already dialed");
        }
    }
    catch (MarshallerException e)
    {
        log.fatal("Marshal error in login, exiting", e);
        shutdown();
    }
}

```

Joining a Conference

Event

JOIN_PART_TO_CONF

Once a user is connected, send a join to bring them into the conference.

Reply

Pull: ACK for success, NACK for failure.

Push: CALL_CONNECTED_TO_CHANGED, PART_INFO_CHANGED with PIR_CONNECTED_TO_CONF for success, NACK for failure.

Prerequisites

See [“Procedure Sequence”](#) on page 43.

Notes

You need to use this event when you use standard dialing; using CF_DEFAULT value of the DIAL_CALL event's CallFlow parameter. When you use blast, instant, or one-click dialing, the conference join is “built into” that type of call flow so the conference join doesn't need to be done explicitly.

Using the Waiting Room

A user is placed into the waiting room when the waiting room is active and the conference is locked. (If the waiting room is not active, the user will be disconnected rather than put in the waiting room.)

The waiting room allows a Moderator to keep participants in a separate location after they have dialed in, but before they have joined the conference. For more information, refer to the *ReadiVoice Subscriber Guide* and to Appendix A of the *ReadiVoice Administration and Maintenance Guide*.

Event

Once a participant is in the waiting room, you can use the following events:

- ONE_TO_ONE - Puts the specified participant in a direct, or “point-to-point” conversation with the conference moderator. You can also use this event to take the participant out of the conversation and put him or her back in the waiting room. For the Connect parameter, use True to connect and False to put back in the waiting room.
- JOIN_PART_TO_CONF - Puts the specified participant in the conference.
- DISCONNECT_CALL - Disconnects the specified participant.

ONE_TO_ONE Response

Pull: ACK for success, NACK for failure.

Push: CALL_STATUS_CHANGED or CALL_CONNECTED_TO_CHANGED for success, NACK for failure.

JOIN_PART_TO_CONF Response

Pull: ACK for success, NACK for failure.

Push: CALL_CONNECTED_TO_CHANGED, PART_INFO_CHANGED with PIR_CONNECTED_TO_CONF for success, NACK for failure.

DISCONNECT_CALL Response

Pull: SUBSCR_INFO for success.

Push: N/A

Prerequisites

See “[Procedure Sequence](#)” on page 43.

Setting the Port for Asynchronous Responses on the Client

Four ports are used for push (asynchronous) mode:

- The sending port on the client
- The receiving port on the server
- The sending port on the server, for the asynchronous response
- The receiving port on the client, for the asynchronous response

The ports used are selected automatically or set to port 80 (receiving port on the server). See “[Understanding CAPI Core Communication Modes: Synchronous and Asynchronous](#)” on page 10. You can specify the receiving port on the client for the asynchronous response.

You can specify any port you like; a simple approach is to pass the value 0 when you send the event, and the system will in turn pick a locally available port on the client.

The port is used per connection; you can have several sessions over the same connection.

Sample Code

See [“Sample Moderator Application”](#) on page 69 to see the sample code in context.

The following code example shows how to set the port to 0; note the lines in bold. There are two sections of code.

```
public SimplePushModerator( String remoteHost, int remotePort, String localAsyncHost,
    int localAsyncPort )
{
    Validate.notEmpty(remoteHost);
    Validate.isTrue(remotePort > 0);
    Validate.notEmpty(localAsyncHost);
    Validate.isTrue(0 <= localAsyncPort);

    try
    {
        /*
         * This is the listener for asynchronous event communication. When the server
         * contacts for asynchronous communication on the specified port, this object
         * will create a connection for the MessageBus.
         */
        listener = new BasicListener(localAsyncPort);

        /*
         * The server uses the cookie to find out the details on how to establish
         * the asynchronous connection with us.
         */
        Cookie cookie = new BasicCookie(localAsyncHost, listener.getLocalPort());

        /*
         * We need to establish a basic HTTP connection to send commands and queries.
         * In push-mode, responses will come back over the asynchronous connection.
         */
        httpConnection = new HttpConnection("SimpleModHttp", remoteHost, remotePort,
            true, Connection.DEFAULT_URL, cookie);

        /*
         * This is a handler for all events from the server/CACS.
         */
        handler = new SimplePushModeratorHandler(this);
    }
}
```

```

/*
 * This objects handles conversion of the raw data off a connection into
 * POJO representations of the CAPI events.
 */
unmarshaller = new CastorUnmarshaller(handler);

/*
 * Once the CAPIUnmarshaller, the command sending HTTP connection, and the
 * asynchronous connection listening service are set up, we give them to a
 * MessageBus so it can use them to finish any further connection setup, and
 * begin those communications as needed.
 */
messageBus = new BasicMessageBus(unmarshaller, httpConnection, listener);

/*
 * This is the utility class that provides methods for sending the CAPI events
 * to the server/CACS. These methods will handle marshalling of the passed
 * arguments into the POJO representations of thoes events. It will then
 * marshall those POJO representations into raw data and use the
 * provided MessageBus to send to the server/CACS.
 */
sender = new CastorSender(messageBus);

shouldRun = true;

/*
 * Start the main thread
 */
start();

}
catch( IOException e )
{
    log.fatal("Error in setup, exiting", e);
    shutdown();
}
}

... // handler class

public static void main( String[] args )
{
    String cacsIp = "192.168.56.104";
    int cacsPort = 80;
    String localAsyncIp = "10.33.48.231";
    int localAsyncPort = 0;
    String logLevel = "ALL";

    try
    {

```

```
    /*
    * Set up the Log4J system
    */
    RollingFileAppender appender = new RollingFileAppender(new TTCCLayout(),
        "SimplePushModerator.log", true);

    PatternLayout p1 = new PatternLayout();
    p1.setConversionPattern("%d{ISO8601} %-5p [%t]: %m%n");

    appender.setLayout(p1);

    BasicConfigurator.configure(appender);
}
catch( IOException e )
{
    BasicConfigurator.configure();
}

Logger logger = Logger.getLogger("com");
logger.setLevel(Level.toLevel(logLevel));
/*
* Create a SimplePushModerator. It should take care of the rest.
*/
SimplePushModerator moderator = new SimplePushModerator(cacsIp, cacsPort,
    localAsyncIp, localAsyncPort);

if( moderator.isAlive() )
{
    System.out.println("moderator activated");
}
}
}
```

Sending Events

Event

The event you want to send; see code example and sample application.

Prerequisites

See [“Procedure Sequence”](#) on page 43.

Notes

The Java utility classes with “sender” in their names have extensive examples of sending events.

Sample Code

See [“Sample Moderator Application”](#) on page 69 to see the sample code in context. Note the lines in bold.

```
public synchronized void login()
{
    log.debug("log in");
    if( !shouldRun )
    {
        return;
    }
    Validate.isTrue(!loggedIn);
    try
    {
        sender.loginPushModerator("PassbackLogin", accessNumber, accessCode,
            subPassCode);
    }
    catch( MarshallerException e )
    {
        log.fatal("Marshal error in login, exiting", e);
        shutdown();
    }
}
```

Reading Events

To read events using the Java utility classes, you can use the methods in the CastorMessageHandlerBase class.

Event

Any event you want to read; see code example and sample application.

Prerequisites

See [“Procedure Sequence”](#) on page 43.

Notes

The `CastorMessageHandlerBase` utility class and `SimplePushModeratorHandler` sample class have extensive examples of reading events.

Sample Code

See [“Sample Moderator Application”](#) on page 69 to see the sample code in context.

```
public void handle( RVAPICONF_INFO_CHANGED message )
{
    log.debug("Conf Info Changed...");

    /*
     * If this event is telling us that a conference has started inform the
     * moderator accordingly
     */
    if( message.getCiReason().getContent().getType() ==
RVAPISIMPLE_CI_REASON.CIR_CONF_STARTED_TYPE )
    {
        if( !moderator.isConfStarted() )
        {
            moderator.setConferenceId(message.getConfId().getContent());
            moderator.setConfStarted(true);
        }
    }
}
```

Identifying Talkers

Identify the people talking in a specified conference.

Event

REGISTER_TALKER_UPDATE and UNREGISTER_TALKER_UPDATE

Response

Pull: N/A (only available to Push Moderator and Application login)

Push: TALKER_UPDATE event containing the participant IDs of the three lines (the three talkers)

Prerequisites

See [“Procedure Sequence”](#) on page 43.

You can register or unregister for talker updates anytime after a conference has started (you need the conference ID) and before the conference has ended.

Notes

REGISTER_TALKER_UPDATE is a request to register a specified conference for Talker Update information. The response is the TALKER_UPDATE event containing the talker information.

Sample Code

Refer to [“Sample Moderator Application”](#) on page 69 to see the code in context.

```

/*
 * You can register or unregister for talker updates anytime after a
 * conference has started (you need the conference ID)and before the
 * conference has ended.
 */
if ( confStarted && !talkerRegisterSent )
{
    registerTalkers();
}

```

Using ACM to Interface With External Applications

ACM lets CAPI applications interface with external applications. Use the Application login. See also [“Using ACM”](#) on page 36.

Event

ACM events in the event reference section.

Prerequisites

See [“Procedure Sequence”](#) on page 43.

Notes

The message passed through the sample code in `acmSender.acmQuery` depends entirely on the type of messages that the ACM application understands. You need to ensure that the message is understood by the ACM application, and handle any response that it sends.

Sample Code

The following example requests information from an ACM application.

Refer to [“Sample Moderator Application”](#) on page 69 to see the code in context.

```
/**
 * Requests Acm data from an ACM application.
 * @param acmAppId the application ID to request data from
 * @param chanHndl the ID used to identify which channel's data to get
 */
public synchronized void getAcmData( int acmAppId, int chanHndl )
{
    Validate.isTrue(loggedIn);
    Validate.notEmpty(sessionId);

    try
    {
        /**
         * The ACM application ID needs to be known in order to query for data.
         * This is the value that is configured in the RV ACM system
         * administration. The query and response string formats and content
         * are not specified in the API (CAPI) and thus must be the result of
         * some kind of agreement between the two applications. This is because
         * the server (CACS) only provides a convenience communication mechanism
         * between Moderator and ACM applications, so that the two applications
         * don't have to communicate directly. This request/response proxy
         * feature is only there for convenience, and does not preclude well
         * designed applications from direct communication outside of the CACS
         * and CAPI framework.
         */
        acmSender.acmQuery("SimpModPassback", sessionId, acmAppId,
            ACM_QUERY_STRING_CHANHNDL + chanHndl + ","
            + ACM_QUERY_STRING_CONFID + conferenceId);
    }
    catch (MarshallerException e)
    {
        log.fatal("Marshal error in talker registration, exiting", e);
        shutdown();
    }
}
```

Conference-End Procedures

Use these procedures when you're ready to end a conference.

Ending a Conference

A conference will end automatically when there are no participants. Here's how to end it explicitly.

Event

END_CONF

Response

Pull: ACK for success, NACK for failure.

Push: CONF_ENDED for success, NACK for failure.

Prerequisites

See ["Procedure Sequence"](#) on page 43.

Sample Code

Refer to ["Sample Moderator Application"](#) on page 69 to see the sample code in context.

```
public synchronized void shutdown()
{
    log.info("shutting down");

    if( shouldRun )
    {
        shouldRun = false;
        if( confStarted )
        {
            try
            {
                sender.endConference("SimpModPassback", sessionId, conferenceId);
            }
            catch( MarshallerException e )
            {
                e.printStackTrace();
                System.exit(1);
            }
        }
    }
}
```

Logging Out

Logging out is the last step to complete when a conference is over.

Event

LOGOUT

Response

Pull: N/A

Push: N/A

Prerequisites

See [“Procedure Sequence”](#) on page 43.

Sample Code

Refer to [“Sample Moderator Application”](#) on page 69 to see the sample code in context.

```
if( loggedIn )
{
    try
    {
        /*
         * This is not specifically needed since the server will kill
         * the logout eventually, but it is good to send if possible.
         */
        sender.logout("SimpModPassback", sessionId);
    }
    catch( MarshallerException e )
    {
        e.printStackTrace();
        System.exit(1);
    }
}

if( null != messageBus )
{
    messageBus.shutdown();
}
}
```

```

else
{
    if( null != messageBus )
    {
        messageBus.shutdown();
    }

    System.exit(1);
}
}

```

Sample Moderator Application

This section presents the CAPI sample application, which you can use as a starting point for writing your own application.

The sample application contains two classes:

- `SimplePushModerator`, which extends `Thread`. This class is the main thread of the sample application. This class sets up an asynchronous push-mode Moderator communication system. With a `SimplePushModeratorHandler` (the second class in the sample application), it will log in, start a conference if one is not already started, blast-dial two participants, and wait for the conference to end before exiting.
- `SimplePushModeratorHandler`, which extends `CastorMessageHandlerBase`. This class is the main message-handling class for all events coming to the application.

SimplePushModerator

```

1  ///////////////////////////////////////////////////////////////////
2  //Copyright (c) 2005 Polycom, Inc
3  ///////////////////////////////////////////////////////////////////
4  package com.polycom.readivoice.capi.simplemoderator;
5
6  import java.io.IOException;
7
8  import org.apache.commons.lang.Validate;
9  import org.apache.commons.logging.Log;
10 import org.apache.commons.logging.LogFactory;
11 import org.apache.log4j.BasicConfigurator;
12 import org.apache.log4j.Level;

```

```
13 import org.apache.log4j.Logger;
14 import org.apache.log4j.PatternLayout;
15 import org.apache.log4j.RollingFileAppender;
16 import org.apache.log4j.TTCCLayout;
17
18 import com.polycom.readivoice.capi.AcmSender;
19 import com.polycom.readivoice.capi.BasicCookie;
20 import com.polycom.readivoice.capi.BasicListener;
21 import com.polycom.readivoice.capi.BasicMessageBus;
22 import com.polycom.readivoice.capi.CapiUnmarshaller;
23 import com.polycom.readivoice.capi.Connection;
24 import com.polycom.readivoice.capi.Cookie;
25 import com.polycom.readivoice.capi.HttpConnection;
26 import com.polycom.readivoice.capi.Listener;
27 import com.polycom.readivoice.capi.MarshallerException;
28 import com.polycom.readivoice.capi.MessageBus;
29 import com.polycom.readivoice.capi.MessageSender;
30 import com.polycom.readivoice.capi.castor.CastorAcmSender;
31 import com.polycom.readivoice.capi.castor.CastorSender;
32 import com.polycom.readivoice.capi.castor.CastorUnmarshaller;
33
34 /**
35  * The main thread of our example program. This class will set up an
36  * asynchronous push-mode Moderator communication system. With a
37  * {@link SimplePushModeratorHandler} it will log in, start a conference (if one
38  * is not already started), blast dial 2 participants, and wait for the
39  * conference to end before exiting.
40  */
41 public class SimplePushModerator extends Thread
42 {
43     private static Log log = LogFactory.getLog(SimplePushModerator.class);
44
45     /*
46     * These are the needed objects to establish communications.
47     */
48     private Listener listener = null;
49     private SimplePushModeratorHandler handler = null;
50     private CapiUnmarshaller unmarshaller = null;
51     private Connection httpConnection = null;
52     private MessageBus messageBus = null;
53     private MessageSender sender = null;
54     private AcmSender acmSender = null;
55     private String sessionId = null;
```

```
56
57  /*
58   * We need to save these for future communications.
59   */
60  private int subscriberId = -1;
61  private int conferenceId = -1;
62
63  /*
64   * Application is responsible for sending heartbeats. This counter is just
65   * used to limit the number of heartbeats to reduce network traffic.
66   */
67  private static final int HEART_BEAT_COUNTER_DEFAULT = 30;
68  private int heartBeatCounter = HEART_BEAT_COUNTER_DEFAULT;
69
70  /*
71   * Internal non-API specific variables to implement program logic.
72   */
73  private boolean shouldRun = false;
74  private boolean loggedIn = false;
75  private boolean confStarted = false;
76  private boolean talkerRegisterSent = false;
77  private int dialCount = 0;
78  private int dialConnectCount = 0;
79  private String accessNumber = "1111";
80  private String accessCode = "1111";
81  private String subPassCode = "1111";
82  private String phoneNum1 = "5129";
83  private String phoneNum2 = "5129";
84  private String partName1 = "Part1";
85  private String partName2 = "Part2";
86
87  private static final int THREAD_SLEEP_TIME = 1000; // in ms
88  private static final int LOGIN_COUNTER_DEFAULT = 10;
89  private static final int CONF_START_COUNTER_DEFAULT = 30;
90  private static final int PARTS_CONNECTED_COUNTER_DEFAULT = 120;
91  private static final String CONFERENCE_SECURITY_NUMBER = "1234567890";
92  private static final String ACM_QUERY_STRING_CHANHNDL = "CHANHNDL=";
93  private static final String ACM_QUERY_STRING_CONFID = "CONFID=";
94  private static int sHeartBeatSeq = 0;
95
96  /**
97   * Constructor. Sets up communication system and starts the main thread.
98   *
```

```

99     * @param remoteHost
100    *         server/CACS' host name (IP address or name)
101    * @param remotePort
102    *         server/CACS' port where commands/queries will be sent
103    * @param localAsyncHost
104    *         host where asynchronous connections should be listened for
105    * @param localAsyncPort
106    *         port where asynchronous connections should be listened for,
107    *         usually port 80 unless otherwise configured on the server/CACS
108    */
109    public SimplePushModerator(String remoteHost, int remotePort,
110                               String localAsyncHost, int localAsyncPort)
111    {
112        Validate.notEmpty(remoteHost);
113        Validate.isTrue(remotePort > 0);
114        Validate.notEmpty(localAsyncHost);
115        Validate.isTrue(0 <= localAsyncPort);
116
117        try
118        {
119            /*
120             * This is the listener for asynchronous event communication. When the
121             * server contacts for asynchronous communication on the specified
122             * port, this object will create a connection for the MessageBus.
123             */
124            listener = new BasicListener(localAsyncPort);
125
126            /*
127             * The server uses the cookie to find out the details on how to
128             * establish the asynchronous connection with us.
129             */
130            Cookie cookie = new BasicCookie(localAsyncHost,
131                                           listener.getLocalPort());
132
133            /*
134             * We need to establish a basic HTTP connection to send commands and
135             * queries. In push-mode, responses will come back over the
136             * asynchronous connection.
137             */
138            httpConnection = new HttpConnection("SimpleModHttp", remoteHost,
139                                               remotePort, true, Connection.DEFAULT_URL, cookie);
140
141            /*

```

```
142     * This is a handler for all events from the server/CACS.
143     */
144     handler = new SimplePushModeratorHandler(this);
145
146     /*
147     * This objects handles conversion of the raw data off a connection
148     * into POJO representations of the CAPI events.
149     */
150     unmarshaller = new CastorUnmarshaller(handler);
151
152     /*
153     * Once the CAPIUnmarshaller, the command sending HTTP connection, and
154     * the asynchronous connection listening service are set up, we give
155     * them to a MessageBus so it can use them to finish any further
156     * connection setup, and begin those communications as needed.
157     */
158     messageBus = new BasicMessageBus(unmarshaller, httpConnection,
159         listener);
160
161     /*
162     * This is the utility class that provides methods for sending the CAPI
163     * events to the server/CACS. These methods will handle marshalling of
164     * the passed arguments into the POJO representations of those events.
165     * It will then marshall those POJO representations into raw data and
166     * use the provided MessageBus to send to the server/CACS.
167     */
168     sender = new CastorSender(messageBus);
169
170     /*
171     * An AcmSender and CastorAcmSender are children of base MessageSender
172     * and CastorSender and as such contain all the base's functionality.
173     * We are using both here just for clarity, otherwise this ACM sender
174     * would be more than sufficient to deal with all our application's
175     * sending needs.
176     */
177     acmSender = new CastorAcmSender(messageBus);
178
179     shouldRun = true;
180
181     /*
182     * Start the main thread
183     */
184     start();
```

```
185
186     }
187     catch (IOException e)
188     {
189         log.fatal("Error in setup, exiting", e);
190         shutdown();
191     }
192 }
193
194 public synchronized final String getSessionId()
195 {
196     return sessionId;
197 }
198
199 public synchronized final void setSessionId( String sessionId )
200 {
201     this.sessionId = sessionId;
202 }
203
204 public synchronized final boolean isConfStarted()
205 {
206     return this.confStarted;
207 }
208
209 public synchronized final void setConfStarted( boolean confStarted )
210 {
211     this.confStarted = confStarted;
212
213     /*
214      * You can register or unregister for talker updates anytime after a
215      * conference has started (you need the conference ID)and before the
216      * conference has ended.
217      */
218     if ( confStarted && !talkerRegisterSent )
219     {
220         registerTalkers();
221     }
222 }
223
224 public synchronized final boolean isLoggedIn()
225 {
226     return this.loggedIn;
227 }
```

```
228
229 public synchronized final void setLoggedIn( boolean loggedIn )
230 {
231     this.loggedIn = loggedIn;
232 }
233
234 public synchronized final boolean shouldRun()
235 {
236     return this.shouldRun;
237 }
238
239 public synchronized final void setShouldRun( boolean shouldRun )
240 {
241     this.shouldRun = shouldRun;
242 }
243
244 public synchronized final int getSubscriberId()
245 {
246     return this.subscriberId;
247 }
248
249 public synchronized final void setSubscriberId( int subscriberId )
250 {
251     this.subscriberId = subscriberId;
252 }
253
254 public synchronized final int getConferenceId()
255 {
256     return this.conferenceId;
257 }
258
259 public synchronized final void setConferenceId( int confenceId )
260 {
261     this.conferenceId = confenceId;
262 }
263
264 public synchronized final int getDialConnectCount()
265 {
266     return this.dialConnectCount;
267 }
268
269 public synchronized final void setDialConnectCount( int dialConnectCount )
270 {
```

```
271     this.dialConnectCount = dialConnectCount;
272 }
273
274 public synchronized final int getDialCount()
275 {
276     return this.dialCount;
277 }
278
279 public synchronized final void setDialCount( int dialCount )
280 {
281     this.dialCount = dialCount;
282 }
283
284 /**
285  * Sends a push mode modeartor login.
286  */
287 public synchronized void login()
288 {
289     log.debug("log in");
290
291     if ( !shouldRun )
292     {
293         return;
294     }
295
296     Validate.isTrue(!loggedIn);
297
298     try
299     {
300         sender.loginPushModerator("PassbackLogin", accessNumber, accessCode,
301             subPassCode);
302     }
303     catch (MarshallerException e)
304     {
305         log.fatal("Marshal error in login, exiting", e);
306         shutdown();
307     }
308 }
309
310 /**
311  * Requests subscriber information from server.
312  */
313 public synchronized void checkSubscription()
```

```
314 {
315     log.debug("checkSubscription");
316
317     if ( !shouldRun )
318     {
319         return;
320     }
321
322     Validate.isTrue(loggedIn);
323     Validate.notEmpty(sessionId);
324
325     try
326     {
327         sender.requestSubscriberInfo("SimpleModPassback", getSessionId(),
328             getSubscriberId());
329     }
330     catch (MarshallerException e)
331     {
332         log.fatal("Marshal error in login, exiting", e);
333         shutdown();
334     }
335
336 }
337
338 /**
339  * Starts a conference only if application is successfully logged in and a
340  * conference has not already been started.
341  */
342 public synchronized void startConf()
343 {
344     log.debug("startConf");
345
346     if ( !shouldRun )
347     {
348         return;
349     }
350
351     Validate.isTrue(loggedIn);
352     Validate.notEmpty(sessionId);
353     Validate.isTrue(!confStarted);
354
355     try
356     {
```

```
357     sender.startConference("PassbackStartConf", getSessionId(),
358         getSubscriberId());
359     }
360     catch (MarshallerException e)
361     {
362         log.fatal("Marshal error in login, exiting", e);
363         shutdown();
364     }
365 }
366
367 /**
368  * Dials up to 2 participants. Session must be active (logged in), and the
369  * conference started. Each call to this function will dial a participant
370  * until the maximum/desired number of dial attempts have been made, then it
371  * will just log an error and return.
372  */
373 public synchronized void dialPart()
374 {
375     if ( !shouldRun )
376     {
377         return;
378     }
379
380     Validate.isTrue(loggedIn);
381     Validate.notEmpty(sessionId);
382     Validate.isTrue(confStarted);
383     Validate.notEmpty(phoneNum1);
384     Validate.notEmpty(phoneNum2);
385
386     try
387     {
388         if ( dialCount < 1 )
389         {
390             log.info("Dialing part one");
391
392             sender.dialCall("DialPassback", getSessionId(), getConferenceId(),
393                 phoneNum1, false, partName1, "PT_SUBSCRIBER", "CF_BLAST",
394                 false);
395
396             dialCount++;
397         }
398         else if ( dialCount < 2 )
399         {
```

```
400         log.info("Dialing part two");
401
402         sender.dialCall("Dial2Passback", getSessionId(), getConferenceId(),
403             phoneNum2, false, partName2, "PT_PARTICIPANT", "CF_BLAST",
404             false);
405
406         dialCount++;
407     }
408     else
409     {
410         log.error("all parts already dialed");
411     }
412 }
413 catch (MarshallerException e)
414 {
415     log.fatal("Marshal error in login, exiting", e);
416     shutdown();
417 }
418 }
419
420 /**
421  * Sends a heartbeat event with an incrementing sequence number.
422  */
423 public synchronized void heartBeat()
424 {
425     if ( heartBeatCounter == 0 )
426     {
427         log.debug("sending connection heartbeat...");
428         try
429         {
430             /*
431              * The server doesn't really care what the sequence number is, it is
432              * more of a specific passback incase the client would like to match
433              * up heartbeats and heartbeat acknowledgements . This is a good
434              * mechanism for the client
435              */
436             sender.sessionHeartbeat("SimpModPassback", sessionId,
437                 sHeartBeatSeq++);
438         }
439         catch (MarshallerException e)
440         {
441             e.printStackTrace();
442             System.exit(1);
```

```
443     }
444     heartBeatCounter = HEART_BEAT_COUNTER_DEFAULT;
445 }
446 else
447 {
448     heartBeatCounter--;
449 }
450 }
451
452 /**
453  * Registers for talker update notifications. The session must be active and
454  * the conference started for this function to work.
455  */
456 public synchronized void registerTalkers()
457 {
458     /*
459     * Talker registration must have an active conference to register for,
460     * otherwise there are no timing concerns.
461     */
462     Validate.isTrue(loggedIn);
463     Validate.notEmpty(sessionId);
464     Validate.isTrue(confStarted);
465
466     try
467     {
468         sender.registerTalkers("SimpModPassback", sessionId, conferenceId);
469         talkerRegisterSent = true;
470     }
471     catch (MarshallerException e)
472     {
473         log.fatal("Marshal error in talker registration, exiting", e);
474         shutdown();
475     }
476 }
477
478 /**
479  * Sets a security code for the conference. The session must be active, and
480  * the conference must be running.
481  */
482 public synchronized void setSecurityCode()
483 {
484     /*
485     * The conf security code is now optional. We are sending here just to
```

```
486     * illustrate its use. It applies to active conferences' parts only after
487     * the event is received (new participants dialing in from that point on,
488     * or being dialed out with certain call flows such as CF_CALLFLOW). If a
489     * join event is manually sent for any part not yet in conference, it will
490     * bypass the conference security code (and other call flow for that
491     * matter) and join them to the conference. The code can be altered at any
492     * time (of the conference's life) by resending the event with the new
493     * value.
494     */
495     Validate.isTrue(loggedIn);
496     Validate.notEmpty(sessionId);
497     Validate.isTrue(confStarted);
498
499     try
500     {
501         sender.setConfSecurityCode("SimpModPassback", sessionId, conferenceId,
502             CONFERENCE_SECURITY_NUMBER);
503     }
504     catch (MarshallerException e)
505     {
506         log.fatal("Marshal error in setting conf security number, exiting", e);
507         shutdown();
508     }
509 }
510
511 /**
512  * Requests ACM data from an ACM application.
513  * @param acmAppId the application ID to request data from
514  * @param chanHndl the ID used to identify which channel's data to get
515  */
516 public synchronized void getAcmPidData( int acmPid, int chanHndl )
517 {
518     Validate.isTrue(loggedIn);
519     Validate.notEmpty(sessionId);
520
521     try
522     {
523         /*
524          * The ACM application ID needs to be known in order to query for data.
525          * This is the value that is configured in the RV ACM system
526          * administration. The query and response string formats and content
527          * are not specified in the API (CAPI) and thus must be the result of
528          * some kind of agreement between the two applications. This is because
```

```

529     * the server (CACS) only provides a convenience communication
530     * mechanism between Moderator and ACM applications, so that the two
531     * applications don't have to communicate directly. This
532     * request/response proxy feature is only there for convenience, and
533     * does not preclude well designed applications from direct
534     * communication outside of the CACS and CAPI framework.
535     */
536     acmSender.acmQuery("SimpModPassback", sessionId, acmAppId,
537         ACM_QUERY_STRING_CHANHNDL + chanHndl + ",",
538         + ACM_QUERY_STRING_CONFID + conferenceId);
539     }
540     catch (MarshallerException e)
541     {
542         log.fatal("Marshal error in talker registration, exiting", e);
543         shutdown();
544     }
545 }
546
547 /**
548  * Shuts down this program, ending the conference if running, logging out if
549  * needed. The termination of the conference is only done here to illustrate
550  * how it is done, not to mandate that when a CAPI application logs out it
551  * should end a conference as a general rule. Typically logging out of a
552  * client should not terminate the conference, unless that is the actual
553  * desired behavior. The unregistrations for talkers and the logging out are
554  * not required per the CAPI spec (as the CACS server will take care of this
555  * on session termination) but it should be done, and thus is illustrated
556  * here.
557  */
558 public synchronized void shutdown()
559 {
560     log.info("shutting down");
561
562     if ( shouldRun )
563     {
564         shouldRun = false;
565
566         if ( confStarted )
567         {
568             try
569             {
570                 /*
571                  * This will automatically be done at conf end, but to be nice

```

```
572         * unregister for the talkers
573         */
574         sender.unregisterTalkers("SimpModPassback", sessionId,
575             conferenceId);
576
577         /*
578         * The Program is done, end the conference if it is still active.
579         */
580         sender.endConference("SimpModPassback", sessionId, conferenceId);
581
582     }
583     catch (MarshallerException e)
584     {
585         e.printStackTrace();
586         System.exit(1);
587     }
588 }
589
590 if ( loggedIn )
591 {
592     try
593     {
594         /*
595         * This is not specifically needed since the server will kill the
596         * logout eventually, but it is nice of us to send if we can.
597         */
598         sender.logout("SimpModPassback", sessionId);
599     }
600     catch (MarshallerException e)
601     {
602         e.printStackTrace();
603         System.exit(1);
604     }
605 }
606
607 if ( null != messageBus )
608 {
609     messageBus.shutdown();
610 }
611 }
612 else
613 {
614     if ( null != messageBus )
```

```

615     {
616         messageBus.shutdown();
617     }
618
619     System.exit(1);
620 }
621 }
622
623 /**
624  * This function only contains internal program methods and variables needed
625  * to accomplish the sample's goal, and has nothing particularly CAPI
626  * specific in it except for the fact that it must initiate the heartbeat
627  * periodically to keep the connection alive. If no traffic is sent to the
628  * server/CACS for a configurable (at the server) amount of time the session
629  * is invalidated and the application will have to log back in to continue.
630  */
631 public void run()
632 {
633     log.debug("Main Thread running...");
634
635     int notLoggedInCounter = LOGIN_COUNTER_DEFAULT;
636     int confNotStartedCounter = CONF_START_COUNTER_DEFAULT;
637     int partsNotConnectedYet = PARTS_CONNECTED_COUNTER_DEFAULT;
638     boolean setupDone = false;
639
640     login();
641
642     while (shouldRun)
643     {
644         try
645         {
646             if ( !setupDone )
647             {
648                 if ( !isLoggedIn() )
649                 {
650                     log.debug("not logged in yet counter left["
651                         + notLoggedInCounter + "]);
652
653                     notLoggedInCounter--;
654                     if ( notLoggedInCounter < 0 )
655                     {
656                         log.fatal("Did not log in before timeout");
657                         shutdown();

```

```
658         }
659     }
660     else
661     {
662         if ( !isConfStarted() )
663         {
664             log.debug("Conf not Started, counter["
665                 + confNotStartedCounter + "]);
666
667             confNotStartedCounter--;
668             if ( confNotStartedCounter < 0 )
669             {
670                 log.fatal("Conf did not start before timeout");
671                 shutdown();
672             }
673         }
674     else
675     {
676         int dialCount = getDialCount();
677         int dialsConnected = getDialConnectCount();
678
679         log.debug("Conf Started, dialCount[" + dialCount
680             + "] dialsConnected[" + dialsConnected + "]);
681
682         if ( dialCount < 2 )
683         {
684             dialPart();
685         }
686
687         if ( dialCount > 0 && (dialCount != dialsConnected) )
688         {
689             log.debug("dials not yet equal counter left["
690                 + partsNotConnectedYet + "]);
691
692             partsNotConnectedYet--;
693             if ( partsNotConnectedYet < 0 )
694             {
695                 log.fatal("All parts did not connect");
696                 shutdown();
697             }
698         }
699
700         if ( dialCount > 0 && (dialsConnected >= dialCount) )
```

```
701         {
702             log.info("All parts are connected");
703
704             /*
705              * Arbitrarily sent here. It is valid anytime after a
706              * conference is running.
707              */
708             setSecurityCode();
709
710             setupDone = true;
711         }
712     }
713 }
714 }
715
716 /*
717  * All of our parts were dialed and connected, and now we notice
718  * that no one is left in conf, end the program.
719  */
720 if ( setupDone && getDialConnectCount() <= 0 )
721 {
722     shutdown();
723 }
724
725 heartBeat();
726
727     sleep(THREAD_SLEEP_TIME);
728 }
729 catch (InterruptedException e)
730 {
731     log.fatal("sleep was interrupted", e);
732 }
733 }
734
735     log.debug("exit main while");
736
737 }
738
739 /**
740  * The main method for program execution. Sets up IP addresses and ports, as
741  * well as the application level logging.
742  *
743  * @param args
```

```
744     */
745     public static void main( String[] args )
746     {
747         String cacsIp = "192.168.56.104";
748         int cacsPort = 80;
749         String localAsyncIp = "10.33.48.231";
750         int localAsyncPort = 0;
751         String logLevel = "ALL";
752
753         try
754         {
755             /*
756              * Set up the Log4J system, refer to the Log4J manual for more
757              * information. Sets up a single rolling log file called
758              * SimplePushModerator.log
759              */
760             RollingFileAppender appender = new RollingFileAppender(
761                 new TTCCLayout(), "SimplePushModerator.log", true);
762
763             PatternLayout p1 = new PatternLayout();
764             p1.setConversionPattern("%d{ISO8601} %-5p [%t]: %m%n");
765
766             appender.setLayout(p1);
767
768             BasicConfigurator.configure(appender);
769         }
770         catch (IOException e)
771         {
772             BasicConfigurator.configure();
773         }
774
775         /*
776          * Sets the logging level for all packages in CAPI, both
777          * this application, and base CAPI classes. This could
778          * be made specific per class file per the Log4J and Apache
779          * commons logging interface manuals.
780          */
781         Logger logger = Logger.getLogger("com");
782         logger.setLevel(Level.toLevel(logLevel));
783
784         /*
785          * Create a SimplePushModerator. It should take care of the rest.
786          */
```

```
787     SimplePushModerator moderator = new SimplePushModerator(cacsIp, cacsPort,
788         localAsyncIp, localAsyncPort);
789
790     if ( moderator.isAlive() )
791     {
792         System.out.println("moderator activated");
793     }
794 }
795 }
796
```

SimplePushModeratorHandler

```
1  //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2  //Copyright (c) 2005 Polycom, Inc
3  //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
4  package com.polycom.readivoice.capi.simplemoderator;
5
6  import org.apache.commons.logging.Log;
7  import org.apache.commons.logging.LogFactory;
8
9  import com.polycom.readivoice.capi.castor.CastorMessageHandlerBase;
10 import com.polycom.readivoice.capi.castor.events.RVAPIACM_QUERY_RESP;
11 import com.polycom.readivoice.capi.castor.events.RVAPICALL_DISCONNECTED;
12 import com.polycom.readivoice.capi.castor.events.RVAPICONF_ENDED;
13 import com.polycom.readivoice.capi.castor.events.RVAPICONF_INFO;
14 import com.polycom.readivoice.capi.castor.events.RVAPICONF_INFO_CHANGED;
15 import com.polycom.readivoice.capi.castor.events.RVAPINACK;
16 import com.polycom.readivoice.capi.castor.events.RVAPIPART_INFO_CHANGED;
17 import com.polycom.readivoice.capi.castor.events.RVAPISESSION_HEARTBEAT_ACK;
18 import com.polycom.readivoice.capi.castor.events.RVAPISUBSCR_INFO;
19 import com.polycom.readivoice.capi.castor.events.RVAPITALKER_UPDATE;
20 import com.polycom.readivoice.capi.castor.events.types.RVAPISIMPLE_ACM_REASON_CODE;
21 import com.polycom.readivoice.capi.castor.events.types.RVAPISIMPLE_CI_REASON;
22 import com.polycom.readivoice.capi.castor.events.types.RVAPISIMPLE_PI_REASON;
23 import com.polycom.readivoice.capi.castor.events.types.RVSIMPLE_API_EV_TYPE;
24
25
26 /**
27  * Main message handling class for all events coming to the application.
28  */
```

```
29 public class SimplePushModeratorHandler extends CastorMessageHandlerBase
30 {
31     private static Log log = LoggerFactory.getLog(SimplePushModeratorHandler.class);
32
33     private static final int ACPINS_APP_ID = 30;
34
35     private SimplePushModerator moderator;
36
37     /**
38      * Constructor. Takes a {@link SimplePushModerator} as a parameter
39      * so that it can set values inside of it, and call methods. Using a proper
40      * object model, this would not be needed.
41      */
42     public SimplePushModeratorHandler( SimplePushModerator moderator )
43     {
44         this.moderator = moderator;
45     }
46
47     public void handle( RVAPINACK message )
48     {
49         log.debug("Got ACK to message" + message);
50
51         switch( message.getEvent().getContent().getType() )
52         {
53             /*
54              * The login failed for some reason, we can't do anything else,
55              * exit the application.
56              */
57             case RVSIMPLE_API_EV_TYPE.ET_LOGIN_PUSH_MODERATOR_TYPE:
58                 moderator.setLoggedIn(false);
59                 log.fatal("Could not login, NACK received reason["
60                     + message.getNackReason().getContent().toString() + "] Message["
61                     + message.getMessage().getContent() + "]" );
62
63                 moderator.shutdown();
64                 break;
65
66             case RVSIMPLE_API_EV_TYPE.ET_START_CONF_TYPE:
67                 moderator.setConfStarted(false);
68                 log.warn("Could not start conference, NACK received reason["
69                     + message.getNackReason().getContent().toString() + "] Message["
70                     + message.getMessage().getContent() + "]" );
71                 break;
```

```

72
73     /*
74     * You could attempt a re-login here instead of failing
75     */
76     case RVSIMPLE_API_EV_TYPE.ET_SESSION_HEARTBEAT_TYPE:
77         log.fatal("Session heartbeat failed, shutdown");
78         moderator.shutdown();
79         break;
80
81     default:
82         break;
83     }
84 }
85
86 public void handle( RVAPISUBSCR_INFO message )
87 {
88     log.debug("Got a SUBSCR_INFO: " + message.toString());
89
90     /*
91     * If this event is a reply to a login request, mark us as logged in,
92     * setting our session and subscription ID values.
93     */
94     if( message.getEvent().getContent().getType() ==
RVSIMPLE_API_EV_TYPE.ET_LOGIN_PUSH_MODERATOR_TYPE )
95     {
96         if( !moderator.isLoggedIn() )
97         {
98             moderator.setSessionId(message.getSessionId().getContent());
99             moderator.setSubscriberId(message.getSubscrId().getContent());
100            moderator.setLoggedIn(true);
101        }
102    }
103
104    if( !moderator.isConfStarted() )
105    {
106        if( message.getConfRunning().getContent() )
107        {
108            moderator.setConferenceId(message.getConfId().getContent());
109            moderator.setConfStarted(true);
110        }
111        else
112        {
113            moderator.startConf();

```

```
114     }
115   }
116 }
117
118 public void handle( RVAPICONF_INFO_CHANGED message )
119 {
120     log.debug("Conf Info Changed...");
121
122     /*
123      * If this event is telling us that a conference has started, inform the
124      * moderator accordingly
125      */
126     if( message.getCiReason().getContent().getType() ==
RVAPISIMPLE_CI_REASON.CIR_CONF_STARTED_TYPE )
127     {
128         if( !moderator.isConfStarted() )
129         {
130             moderator.setConferenceId(message.getConfId().getContent());
131             moderator.setConfStarted(true);
132         }
133     }
134
135 }
136
137 /*
138 * This may be sent periodically notifying us of conference information. We
139 * will only get this if the conf is running, so we should inform the
140 * moderator as such if it isn't already.
141 */
142 public void handle( RVAPICONF_INFO message )
143 {
144     log.debug("Got conf info..." + message.toString());
145
146     if( !moderator.isConfStarted() )
147     {
148         moderator.setConferenceId(message.getConfId().getContent());
149         moderator.setConfStarted(true);
150     }
151 }
152
153 public void handle( RVAPICONF_ENDED message )
154 {
155     log.debug("Got Conf ended..." + message.toString());
```

```

156     moderator.shutdown();
157 }
158
159 public void handle( RVAPIPART_INFO_CHANGED message )
160 {
161     if( message.getPiReason().getContent().getType() ==
RVAPISIMPLE_PI_REASON.PIR_CONNECTED_TO_CONF_TYPE )
162     {
163         moderator.setDialConnectCount(moderator.getDialConnectCount() + 1);
164         moderator.getAcmData(ACMPINS_APP_ID, message.getPartId().getContent());
165     }
166 }
167
168 public void handle( RVAPICALL_DISCONNECTED message )
169 {
170     log.debug("got part disconnected msg..." + message.toString());
171
172     moderator.setDialConnectCount(moderator.getDialConnectCount() - 1);
173 }
174
175 public void handle( RVAPISESSION_HEARTBEAT_ACK message )
176 {
177     log.debug("got session heartbeat ack..." + message.toString());
178 }
179
180 public void handle(RVAPITALKER_UPDATE message )
181 {
182     int talker1 = message.getTalker1().getContent();
183     int talker2 = message.getTalker2().getContent();
184     int talker3 = message.getTalker3().getContent();
185
186     log.debug("Got a talker update, T1[" + talker1 + "] T2[" + talker2
187         + "] T3[" + talker3 + "]");
188 }
189
190 public void handle( RVAPIACM_QUERY_RESP message )
191 {
192     if( message.getAcmReason().getContent().getType() ==
RVAPISIMPLE_ACM_REASON_CODE.AR_NO_APP_TYPE)
193     {
194         log.error("There is no application currently registered with an ID of[" +
195             message.getApplicationId().getContent() + "]);
196     }
197     else

```

```
198     {
199         log.debug("ACM response data[" + message.getRespString().getContent() + "]);
200     }
201 }
202
203 }
204
```

Developing an AAPI Application

This chapter presents an overview of the Admin API, which allows you to write an application to access the system administration functions of the ReadVoice system.



Study [Chapter 2](#), “Developing a CAPI Application,” before continuing. Much of the information about CAPI is relevant and necessary for using the Admin API.

Port Usage

Currently, the Admin API uses only the synchronous pull mode of the core communication classes. It doesn't support asynchronous (push) mode. For specifics on the CAPI pull mode, see [“Understanding How Ports Are Used”](#) on page 11.

Connections, Sessions, and Logins

The HTTP or HTTPS connection between the client admin application and the ReadVoice CACS operates in the same fashion as it does for CAPI. See [“Understanding Connections, Sessions, and Logins”](#) on page 14 and [“Creating a Connection”](#) on page 32.

When initializing the message bus via the `com.polycom.readivoice.capi.BasicMessageBus` class, be sure to use the proper constructor for pull mode.

A connection times out after four minutes without a session over it. This is not configurable.

An Admin session is an Admin API process that takes place over the client-server connection. Each instance of an application using the Admin API has its own session through which it communicates to the ReadVoice system.

As with CAPI, multiple sessions can run simultaneously over a connection. Event requests and responses running over the same connection are handled sequentially by the CACS.

An Admin API login creates a session. The Admin API has only one login type, `admin`, which provides access to all of the Admin API capabilities. When the Admin API login request is sent to the admin server, it creates a new session and returns a unique session ID to the client. Subsequent requests from the client must provide this session ID in order for the request to be processed.

Admin API sessions time out after two minutes without any traffic. To keep a session open and active when no requests are being made, send the Admin API `SESSION_HEARTBEAT` event to the CACS periodically.

Connection URL for ReadVoice

When creating a connection to the ReadVoice Admin server, use the URL `"/api/admin/"`. This value is also defined as public static field `DEFAULT_URL` in the `com.polycom.readivoice.adminapi.AdminMessageSender` class.

For example:

```
HttpConnection conn = new HttpConnection(
    "Test Connection", "192.168.55.100", 80, true,
    "/api/admin/");
```

Admin API SDK Components

As mentioned previously, the Admin API re-uses the core communication and utility classes from CAPI for communication with the ReadVoice system. These core classes are located in the `capibase` jar file.

A separate jar file, `adminapi`, contains the classes specific to the Admin API. The noteworthy classes are listed in [Table 3-1](#).

Table 3-1 Java classes provided with Admin API

Category	Class name	Description
AAPI Utilities	AdminMessageSender	A utility interface for sending all Admin API requests to the ReadVoice system. Use this class to isolate your application from Castor specifics.
	AdminApiUnmarshaller	Handler interface providing the interface for handling responses to the API requests. This interface must be overridden to provide the desired application functionality.
AAPI Castor Utilities	CastorAdminSender	Castor-specific class that implements the AdminMessageSender for sending events to the ReadVoice system.
	CastorAdminMessageHandlerBase	Castor-event-specific base class for handling all events. Each handle method must be overridden by the application to implement an action to take when a specific event is received. Implements <code>java.util.Observer</code> .
	CastorAdminUnmarshaller	Extends <code>Observable</code> . Implements the AdminApiUnmarshaller interface.
AAPI Castor Events	All events from the Admin API schema	See Admin API event reference.

Associating the AAPI Events and the Java Utility Classes

The Admin API sender methods (the methods used to send requests to the ReadVoice Admin server) create XML-based messages over HTTP (or HTTPS) to the ReadVoice CACS.

The available sender methods are listed in the Javadocs. See the `AdminMessageSender` class in the `com.polycom.readivoice.adminapi` package. Details of the associated event and response to the request can be found in the Admin API Reference.

For each sender method, there is an associated event with a similar name. For example, the `addAccessNumber` sender method sends the `ADD_ACCESS_NUMBER` event.

The event reference for each request event lists the possible response events from the ReadVoice system. For example, the `ADD_ACCESS_CLASS` event returns an `ACCESS_CLASS_ADDED` event when processed successfully.

The handler method, defined in the `CastorAdminMessageHandlerBase` class of the `com.polycom.readivoice.adminapi.castor` package, uses the naming convention: `handle(RVADMINresponse_event_name)`

In addition to the Java utility classes created for AAPI, a java class is automatically generated from the schema for each event by the Castor program. The name of each auto-generated class is based on the event name, but with a few minor differences that ensure uniqueness for the generated code:

- The names of events are all caps with underscores. Some examples are `ADD_ACCESS_CLASS`, `ADD_ACCESS_NUMBER`, and `ACCESS_CLASS_ADDED`.
- The Castor classes representing these events attach the Admin API namespace (`RVADMIN`) to the front of each generated class and type. Thus, when you directly reference the classes generated by Castor, you need to use the Castor-assigned name: `RVADMINADD_ACCESS_CLASS`, `RVADMINADD_ACCESS_NUMBER`, and `RVADMINACCESS_CLASS_ADDED`.

[Table 3-2](#) shows the schema and Castor naming conventions.

Table 3-2 Admin API event-naming conventions

Item type	Schema name	Castor Java class name	Package ^a
Events	EVENTNAME Example: ADD_ACCESS_CLASS ADD_ACCESS_NUMBER	NAMESPACEEVENTNAME Examples: RVADMINADD_ACCESS_CLASS RVADMINADD_ACCESS_NUMBER	events
Types	EVENTNAME Example: STRING	EVENTNAME Example: STRING	events.types
Enums	EVENTNAME Example: NACK_REASON	NAMESPACEEVENTNAME Example: RVADMINNACK_REASON	events
Enum values	EVENTNAME Example: NACK_REASON	NAMESPACESIMPLE_ EVENTNAME Example: RVADMINSIMPLE_NACK_REASON	events.types

a. Each package is relative to the `com.polycom.readivoice.adminapi` package.

Auto-generated helper classes

The Admin API also includes a set of helper classes generated from the XML event set. These helper classes are used in the sender methods. Each non-event class and enumeration has an associated helper class located in the `com.polycom.readivoice.adminapi.generated` package.

These classes are used when sending requests to the Readivoice system. Note that the Castor-generated enums must be used when handling the responses back from the Readivoice system.

Admin API Responses

When an request is received and processed by the Admin server, the sending client may receive one of two possible responses. If the request was processed successfully, the Admin server sends the client an `RVADMINACK` event containing the response event.

When using the `CastorAdminMessageHandlerBase`, the response event is automatically extracted from the `RVADMINACK` event and passed to the appropriate handler method.

If the Admin server rejected the request, it sends back an RVADMINNACK event containing a NACK reason for the failure.

Table 3-3 lists the different generic NACK_REASON values. Included in the RVADMINNACK event is a text based message (in the Message field) that provides a more detailed error description.

Table 3-3 Possible NACK_REASON values from Admin server

NACK_REASON Enum Value	Description
BAD_SESSION_ID	The specified session doesn't exist.
BAD_LOGIN	The login information is incorrect.
BAD_ENTRY	One of the specified parameters either doesn't exist or has invalid format (such as an invalid number group ID, empty subscriber group name, or alpha characters in dial-out postfix).
DUP_ENTRY	One of the specified parameters contains a duplicate value (for example, adding an access phone number that already exists).
INT_DB_ERROR	An error occurred while accessing the database.
ERROR	The system received an event that it does not handle (bad event).
INV_OPERATION	The requested action isn't permitted (such as deleting a system row or specifying an empty list of attributes).
INV_STATE	The data is correct, but the action is not permitted at this time (for example, you can't delete an in-use access number, or change a hidden number if the system isn't configured to use hidden numbers).

Migrating MAPI Applications to CAPI

Use this appendix and the technical information it references to migrate MAPI applications to CAPI. It contains:

- [“Migration Overview”](#) on page 101
- [“Compatibility Notes”](#) on page 102
- [“MAPI-to-CAPI Sample Application Comparison”](#) on page 103

Migration Overview

The migration process is primarily dependent upon how you implemented your current conferencing application, using either the Java or XML version of MAPI. We recommend that you use the provided resources as guides, and then update and/or rewrite your application as necessary.

The primary technical information resources for migration are:

- *Conferencing API Javadoc Reference* – Standard Javadoc HTML reference pages generated from the CAPI code.
- *Conferencing API Reference* – CAPI event and enum reference in PDF form.
- Conferencing API code examples – Source files for the sample Moderator application discussed in this manual.
- This guide – [Chapter 2](#), “Developing a CAPI Application,” provides detailed information about how to use CAPI.
- MAPI-to-CAPI comparison – [“MAPI-to-CAPI Sample Application Comparison”](#) on page 103 shows the sample applications for MAPI and CAPI side by side. Use this section to see how to implement features in CAPI.

Compatibility Notes

Your MAPI client is not compatible with the new release of the ReadiVoice software. You must port your MAPI application to CAPI in order to use it with a ReadiVoice v. 3.0 system. If you have a CAPI client developed for ReadiVoice v. 2.56.0 or v. 2.60.0, you can use it with ReadiVoice v. 3.0.

MAPI-to-CAPI Sample Application Comparison

This section covers how to migrate the sample Moderator application provided with MAPI to CAPI. Both are valid running applications and include key procedures such as creating a conference and adding participants. Use these examples and the CAPI event reference to migrate your application.

We've provided the source code for the MAPI version and the CAPI version of the application. The code is provided side by side to demonstrate, where possible, how MAPI and CAPI implemented similar functions. The same functions are not provided side by side in all locations; the differences in API and in architecture mean that there is not a simple one-to-one comparison of a MAPI implementation versus a CAPI implementation. Where appropriate we provide cross-references in the MAPI code to the CAPI implementation that is similar or equivalent.

Note: The CAPI application here is a simplified version of the one referenced elsewhere in this document and included with the CAPI SDK. The version provided here excludes features not available in MAPI, to provide a clearer old-to-new comparison where possible.

Table A-1 MAPI to CAPI comparison: Simple Moderator

Simple_Moderator (MAPI)	SimplePushModerator (CAPI)
Import statements	
<pre>import java.io.*; import java.lang.*; import com.voyanttech.cnow.mapi.events.*; import com.voyanttech.cnow.mapi.types.*; import com.voyanttech.cnow.mapi.utils.*; import com.voyanttech.cnow.mtools.*; import com.voyanttech.cnow.resources.*;</pre>	<pre>package com.polycom.readivoice.capi.apps; import java.io.IOException; import org.apache.commons.lang.Validate; import org.apache.commons.logging.Log; import org.apache.commons.logging.LogFactory; import org.apache.log4j.BasicConfigurator; import org.apache.log4j.Level; import org.apache.log4j.Logger; import org.apache.log4j.PatternLayout; import org.apache.log4j.RollingFileAppender; import org.apache.log4j.TTCCLayout; import com.polycom.readivoice.capi.BasicCookie; import com.polycom.readivoice.capi.BasicListener; import com.polycom.readivoice.capi.BasicMessageBus; import com.polycom.readivoice.capi.CapiUnmarshaller; import com.polycom.readivoice.capi.Connection; import com.polycom.readivoice.capi.Cookie; import com.polycom.readivoice.capi.HttpConnection; import com.polycom.readivoice.capi.Listener; import com.polycom.readivoice.capi.MarshallerException; import com.polycom.readivoice.capi.MessageBus; import com.polycom.readivoice.capi.MessageSender; import com.polycom.readivoice.capi.castor.CastorSender; import com.polycom.readivoice.capi.castor.CastorUnmarshaller;</pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
Main thread of program	
<pre> public class Simple_Moderator extends Thread { //// // Initial Setup Section //// //// // Message sending and receiving (to/from ReadVoice) objects //// MESSAGE_BUS msg_bus = null; MESSAGE_SENDER msg_sender; SIMP_MESSAGE_HANDLER simp_mess_hand; int dialOutSequenceId = 0; </pre>	<pre> /** * The main thread of our example program. This class will set up an * asynchronous push-mode Moderator communication system. With a * {@link SimplePushModeratorHandler} it will log in, start a conference (if one * is not already started), blast dial 2 participants, and wait for the * conference to end before exiting. */ public class SimplePushModerator extends Thread { private static Log log = LogFactory.getLog(SimplePushModerator.class); /* * These are the needed objects to establish communications. */ private Listener listener = null; private SimplePushModeratorHandler handler = null; private CapiUnmarshaller unmarshaller = null; private Connection httpConnection = null; private MessageBus messageBus = null; private MessageSender sender = null; private String sessionId = null; </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
<pre> //// // Conference object that stores all information needed to have a conference. // (Bridges, ports, addresses, etc.) // This is the key to starting a conference. //// Conference conf; <i>// Used to store the information now stored in subscriberId and confenceld at right</i> //// // In this case the state of whether the conference has been started or not is set by the message handler //// boolean conf_start = false; int has_dialed = 0; PART_INFO_WITH_NAME participant1; PART_INFO_WITH_NAME participant2; </pre>	<pre> /* * We need to save these for future communications. */ private int subscriberId = -1; private int confenceld = -1; /* * Application is responsible for sending heartbeats. This counter * is just used to limit the number of heartbeats to reduce network traffic. */ private static final int HEART_BEAT_COUNTER_DEFAULT = 30; private int heartBeatCounter = HEART_BEAT_COUNTER_DEFAULT; /* * Internal non-API specific variables to implement program logic. */ private boolean shouldRun = false; private boolean loggedIn = false; private boolean confStarted = false; private int dialCount = 0; private int dialConnectCount = 0; private String accessNumber = "1111"; private String accessCode = "1111"; private String subPassCode = "1111"; private String phoneNum1 = "5129"; private String phoneNum2 = "5129"; private String partName1 = "Part1"; private String partName2 = "Part2"; private static final int THREAD_SLEEP_TIME = 1000; //in ms private static final int LOGIN_COUNTER_DEFAULT = 10; private static final int CONF_START_COUNTER_DEFAULT = 30; private static final int PARTS_CONNECTED_COUNTER_DEFAULT = 120; private static int sHeartBeatSeq = 0; </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
<p>Constructor, methods, creating a connection</p> <p>The methods are included later in CAPI than in MAPI for a cleaner design. Most methods have the same name in each, however.</p>	
<pre> //// // Constructor Section //// Simple_Moderator() { Config cfg = new Config(null); // Config information no longer stored inobject; see page 124 </pre>	<pre> /** * Constructor. Sets up communication system and starts the main thread. * * @param remoteHost * server/CACS' host name (IP address or name) * @param remotePort * server/CACS' port where commands/queries will be sent * @param localAsyncHost * host where asynchronous connections should be listened for * @param localAsyncPort * port where asynchronous connections should be listened for, * usually port 80 unless otherwise configured on the server/CACS */ public SimplePushModerator(String remoteHost, int remotePort, String localAsyncHost, int localAsyncPort) { Validate.notEmpty(remoteHost); Validate.isTrue(remotePort > 0); Validate.notEmpty(localAsyncHost); Validate.isTrue(0 <= localAsyncPort); </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
<pre>try{ simp_mess_hand = new SIMP_MESSAGE_HANDLER(this); msg_bus = new MESSAGE_BUS(simp_mess_hand, (Thread.currentThread()).getThreadGroup()); msg_sender = new MESSAGE_SENDER(msg_bus);</pre>	<pre>try { /* * This is the listener for asynchronous event communication. When the server * contacts for asynchronous communication on the specified port, this object * will create a connection for the MessageBus. */ listener = new BasicListener(localAsyncPort); /* * The server uses the cookie to find out the details on how to establish * the asynchronous connection with us. */ Cookie cookie = new BasicCookie(localAsyncHost, listener.getLocalPort()); /* * We need to establish a basic HTTP connection to send commands and queries. * In push-mode, responses will come back over the asynchronous connection. */ httpConnection = new HttpConnection("SimpleModHttp", remoteHost, remotePort, true, Connection.DEFAULT_URL, cookie); /* * This is a handler for all events from the server/CACS. */ handler = new SimplePushModeratorHandler(this); /* * This objects handles conversion of the raw data off a connection into * POJO representations of the CAPI events. */ unmarshaller = new CastorUnmarshaller(handler); /* * Once the CAPIUnmarshaller, the command sending HTTP connection, and the * asynchronous connection listening service are set up, we give them to a * MessageBus so it can use them to finish any further connection setup, and * begin those communications as needed. */ messageBus = new BasicMessageBus(unmarshaller, httpConnection, listener);</pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
<pre> // This information is the Access Number/Subscriber's number, // Subscriber's password, SP, and SPPW, and // an instance number for this moderator (you could login // as several moderators using different instance // numbers ex. ULONG(3), ULONG(4), etc.) msg_sender.guiModeratorLogin("1111/0007", "0007", "ServiceProvider", "SPPassWord", new ULONG(0)); //Begin thread start(); } //End Try catch(IOException e) { System.out.println(e.getMessage()); } } //End constructor for Simple_Moderator </pre>	<pre> /* * This is the utility class that provides methods for sending the CAPI events * to the server/CACS. These methods will handle marshalling of the passed * arguments into the POJO representations of those events. It will then * marshall those POJO representations into raw data and use the * provided MessageBus to send to the server/CACS. */ sender = new CastorSender(messageBus); shouldRun = true; /* * Start the main thread */ start(); } catch(IOException e) { log.fatal("Error in setup, exiting", e); shutdown(); } } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
<p><i>// Continued on page 119</i></p>	<pre> public synchronized final String getSessionId() { return sessionId; } public synchronized final void setSessionId(String sessionId) { this.sessionId = sessionId; } public synchronized final boolean isConfStarted() { return this.confStarted; } public synchronized final void setConfStarted(boolean confStarted) { this.confStarted = confStarted; } public synchronized final boolean isLoggedIn() { return this.loggedIn; } public synchronized final void setLoggedIn(boolean loggedIn) { this.loggedIn = loggedIn; } public synchronized final boolean shouldRun() { return this.shouldRun; } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
	<pre> public synchronized final void setShouldRun(boolean shouldRun) { this.shouldRun = shouldRun; } public synchronized final int getSubscriberId() { return this.subscriberId; } public synchronized final void setSubscriberId(int subscriberId) { this.subscriberId = subscriberId; } public synchronized final int getConfenceld() { return this.confenceld; } public synchronized final void setConfenceld(int confenceld) { this.confenceld = confenceld; } public synchronized final int getDialConnectCount() { return this.dialConnectCount; } public synchronized final void setDialConnectCount(int dialConnectCount) { this.dialConnectCount = dialConnectCount; } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
	<pre> public synchronized final int getDialCount() { return this.dialCount; } public synchronized final void setDialCount(int dialCount) { this.dialCount = dialCount; } public synchronized void login() { log.debug("log in"); if(!shouldRun) { return; } Validate.isTrue(!loggedIn); try { sender.loginPushModerator("PassbackLogin", accessNumber, accessCode, subPassCode); } catch(MarshallerException e) { log.fatal("Marshal error in login, exiting", e); shutdown(); } } public synchronized void checkSubscription() { log.debug("checkSubscription"); </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
	<pre> if(!shouldRun) { return; } Validate.isTrue(loggedIn); Validate.notEmpty(sessionId); try { sender.requestSubscriberInfo("SimpleModPassback", getSessionId(), getSubscriberId()); } catch(MarshallerException e) { log.fatal("Marshal error in login, exiting", e); shutdown(); } } public synchronized void startConf() { log.debug("startConf"); if(!shouldRun) { return; } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
	<pre> Validate.isTrue(loggedIn); Validate.notEmpty(sessionId); Validate.isTrue(!confStarted); try { sender.startConference("PassbackStartConf", getSessionId(), getSubscriberId()); } catch(MarshallerException e) { log.fatal("Marshal error in login, exiting", e); shutdown(); } } public synchronized void dialPart() { if(!shouldRun) { return; } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
	<pre> Validate.isTrue(loggedIn); Validate.notEmpty(sessionId); Validate.isTrue(confStarted); Validate.notEmpty(phoneNum1); Validate.notEmpty(phoneNum2); try { if(dialCount < 1) { log.info("Dialing part one"); sender.dialCall("Dial1Passback", getSessionId(), getConfenceld(), phoneNum1, false, partName1, "PT_SUBSCRIBER", "CF_BLAST", false); dialCount++; } else if(dialCount < 2) { log.info("Dialing part two"); sender.dialCall("Dial2Passback", getSessionId(), getConfenceld(), phoneNum2, false, partName2, "PT_PARTICIPANT", "CF_BLAST", false); dialCount++; } else { log.error("all parts already dialed"); } } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
	<pre> catch(MarshallerException e) { log.fatal("Marshal error in login, exiting", e); shutdown(); } } public synchronized void heartBeat() { if(heartBeatCounter == 0) { log.debug("sending connection heartbeat..."); try { sender.sessionHeartbeat("SimpModPassback", sessionId, sHeartBeatSeq++); } catch(MarshallerException e) { e.printStackTrace(); System.exit(1); } heartBeatCounter = HEART_BEAT_COUNTER_DEFAULT; } else { heartBeatCounter--; } } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
	<pre> /** * Shuts down this program, ending the conference if running, logging out if * needed. */ public synchronized void shutdown() { log.info("shutting down"); if(shouldRun) { shouldRun = false; if(confStarted) { try { sender.endConference("SimpModPassback", sessionId, confencId); } catch(MarshallerException e) { e.printStackTrace(); System.exit(1); } } } } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
	<pre> if(loggedIn) { try { /* * This is not specifically needed since the server will kill * the logout eventually, but it is nice of us to send if we can. */ sender.logout("SimpModPassback", sessionId); } catch(MarshallerException e) { e.printStackTrace(); System.exit(1); } } if(null != messageBus) { messageBus.shutdown(); } else { if(null != messageBus) { messageBus.shutdown(); } System.exit(1); } } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
<pre> public void run() { while(true) { try { sleep(3000); } } } </pre>	<pre> /* * This function only contains internal program methods and variables needed * to accomplish the sample's goal, and has nothing CAPI specific in it except * for the fact that it must initiate the heartbeat to keep the connection * alive. If no traffic is sent to the server/CACS for a configurable amount * of time the session is invalidated and the application will have to log * back in to continue. */ public void run() { log.debug("Main Thread running..."); int notLoggedInCounter = LOGIN_COUNTER_DEFAULT; int confNotStartedCounter = CONF_START_COUNTER_DEFAULT; int partsNotConnectedYet = PARTS_CONNECTED_COUNTER_DEFAULT; boolean setupDone = false; login(); } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
<pre> if(!conf_start) { msg_sender.startConference(); sleep(3000); continue; } //End if a Conf has not started if(conf_start) { ///// // Initialize the participant objects with information. // This can be done dynamically in the run() method after a conference // has started by using new: // PART_INFO_WITH_NAME participant1 = new PART_INFO_WITH_NAME("Name", "Phone", etc...), // or statically outside of run() ///// participant1 = new PART_INFO_WITH_NAME("Name1", "25129", conf.getConfId(), new PART_TYPE(PART_TYPE.PT_SUBSCRIBER)); participant2 = new PART_INFO_WITH_NAME("Name2", "21129", conf.getConfId(), new PART_TYPE(PART_TYPE.PT_PARTICIPANT)); if(has_dialed == 0) { ///// // This is very important. Set the port for the participant. // The '-1' that is passed is a requested initial port ID. // When this person is called and placed in a conference // the actual port on the bridge will be returned. ///// participant1.setCallerPortId(CONSTANTS.UNKNOWN_PORT_ID); } </pre>	<pre> while(shouldRun) { try { if(!setupDone) { if(!isLoggedIn()) { log.debug("not logged in yet counter left[" + notLoggedInCounter + "]"); notLoggedInCounter--; if(notLoggedInCounter < 0) { log.fatal("Did not log in before timeout"); shutdown(); } } } else { if(!isConfStarted()) { log.debug("Conf not Started, counter[" + confNotStartedCounter + "]"); confNotStartedCounter--; if(confNotStartedCounter < 0) { log.fatal("Conf did not start before timeout"); shutdown(); } } } } } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
<pre> //// // Set sequence number to this dial out, so we can trace it when PORT_KNOW arrives //// participant1.setDialOutSequence(dialOutSequenceId++); //// // This adds the participant to the list of participants for the conference. //// conf.addPart(participant1); System.out.println("Participant1 has been added to conf list"); //// // This dials the participant. See CAPI dialCall on page 115 //// msg_sender.dialCall(participant1); has_dialed++; } if(has_dialed == 1) { sleep(3000); //// // Set portId to unknown //// participant2.setCallerPortId(CONSTANTS.UNKNOWN_PORT_ID); //// // Set sequence number to this dial out, so we can trace it when PORT_KNOW arrives //// participant2.setDialOutSequence(dialOutSequenceId++); </pre>	<pre> else { int dialCount = getDialCount(); int dialsConnected = getDialConnectCount(); log.debug("Conf Started, dialCount[" + dialCount + "] dialsConnected[" + dialsConnected + "]); if(dialCount < 2) { dialPart(); } if(dialCount > 0 && (dialCount != dialsConnected)) { log.debug("dials not yet equal counter left[" + partsNotConnectedYet + "]); partsNotConnectedYet--; if(partsNotConnectedYet < 0) { log.fatal("All Parts did not connect"); shutdown(); } } } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
<pre> //// // Add Participant to conference list //// conf.addPart(participant2); //// // Dial Participant //// msg_sender.dialCall(participant2); has_dialed++; } } //end if(conf_start) } //End Try </pre>	<pre> if(dialCount > 0 && (dialsConnected >= dialCount)) { log.info("All parts are connected"); setupDone = true; } } } } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
<pre> //// // Note: this simplified version does nothing with exceptions if they are encountered. //// catch(OD_EXCEPTION e) { System.out.println(e.getMessage()); } catch(InterruptedException e) { System.out.println(e.getMessage()); } } //End while } //End conference Driver method </pre>	<pre> // All of our parts were dialed and connected, and now // we notice that no one is left in conf, end the program. if(setupDone && getDialConnectCount() <= 0) { shutdown(); } heartBeat(); sleep(THREAD_SLEEP_TIME); } catch(InterruptedException e) { log.fatal("sleep was interrupted", e); } } log.debug("exit main while"); } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
Main method	
<pre> public static void main(String[] args) { //// // Makes a new Simple_Moderator object, and calls its constructor // which in turn starts the thread. //// Simple_Moderator sm = new Simple_Moderator(); } //End main } //End class Simple_Moderator </pre>	<pre> public static void main(String[] args) { String cacsIp = "192.168.56.104"; int cacsPort = 80; String localAsyncIp = "10.33.48.231"; int localAsyncPort = 0; String logLevel = "ALL"; try { /* * Set up the Log4J system */ RollingFileAppender appender = new RollingFileAppender(new TTCCLayout(), "SimplePushModerator.log", true); PatternLayout pl = new PatternLayout(); pl.setConversionPattern("%d{ISO8601} %-5p [%t]: %m%n"); appender.setLayout(pl); } } </pre>

Table A-1 MAPI to CAPI comparison: Simple Moderator (continued)

Simple_Moderator (MAPI) (continued)	SimplePushModerator (CAPI) (continued)
	<pre> BasicConfigurator.configure(appender); } catch(IOException e) { BasicConfigurator.configure(); } Logger logger = Logger.getLogger("com"); logger.setLevel(Level.toLevel(logLevel)); /* * Create a SimplePushModerator. It should take care of the rest. */ SimplePushModerator moderator = new SimplePushModerator(cacsIp, cacsPort, localAsyncIp, localAsyncPort); if(moderator.isAlive()) { System.out.println("moderator activated"); } } </pre>

Table A-2 MAPI to CAPI comparison: Simple Moderator Handler

SIMP_MESSAGE_HANDLER (MAPI)	SimplePushModeratorHandler (CAPI)
<pre>import java.io.*; import java.util.*; import java.net.*; import java.lang.*; import com.voyanttech.cnow.resources.*; import com.voyanttech.cnow.mapi.events.*; import com.voyanttech.cnow.mapi.types.*; import com.voyanttech.cnow.mapi.utils.*; import com.voyanttech.cnow.mtools.*;</pre>	<pre>package com.polycom.readivoice.capi.apps; import org.apache.commons.logging.Log; import org.apache.commons.logging.LogFactory; import com.polycom.readivoice.capi.castor.CastorMessageHandlerBase; import com.polycom.readivoice.capi.castor.events.RVAPICALL_DISCONNECTED; import com.polycom.readivoice.capi.castor.events.RVAPICONF_ENDED; import com.polycom.readivoice.capi.castor.events.RVAPICONF_INFO; import com.polycom.readivoice.capi.castor.events.RVAPICONF_INFO_CHANGED; import com.polycom.readivoice.capi.castor.events.RVAPINACK; import com.polycom.readivoice.capi.castor.events.RVAPIPART_INFO_CHANGED; import com.polycom.readivoice.capi.castor.events.RVAPISSESSION_HEARTBEAT_ACK; import com.polycom.readivoice.capi.castor.events.RVAPISUBSCR_INFO; import com.polycom.readivoice.capi.castor.events.types.RVAPISIMPLE_CI_REASON; import com.polycom.readivoice.capi.castor.events.types.RVAPISIMPLE_PI_REASON; import com.polycom.readivoice.capi.castor.events.types.RVSIMPLE_API_EV_TYPE;</pre>

Table A-2 MAPI to CAPI comparison: Simple Moderator Handler (continued)

SIMP_MESSAGE_HANDLER (MAPI) (continued)	SimplePushModeratorHandler (CAPI) (continued)
Main message-handling class and constructor	
<pre> // OD_SERVER_BASE can handle all events from CACS system // This class is just a wrapper to give specific functionality // to some of the possible events (messages) for the user interface // Simple_Moderator.java. It can contain only empty method or have // code for each, depending on the user interface. ////// class SIMP_MESSAGE_HANDLER extends OD_SERVER_BASE { Simple_Moderator mod; //////////// Message Handler Constructor //////////// public SIMP_MESSAGE_HANDLER(Simple_Moderator mod) { //// // Associate mod in this class with the actual running user interface //// super(); this.mod = mod; } </pre>	<pre> /** * Main message handling class for all events coming to the application. */ public class SimplePushModeratorHandler extends CastorMessageHandlerBase { private static Log log = LogFactory.getLog(SimplePushModeratorHandler.class); private SimplePushModerator moderator; /** * Constructor. Takes a {@link SimplePushModerator} as a parameter * so that it can set values inside of it, and call methods. Using a proper * object model, this would not be needed. */ public SimplePushModeratorHandler(SimplePushModerator moderator) { this.moderator = moderator; } </pre>

Table A-2 MAPI to CAPI comparison: Simple Moderator Handler (continued)

SIMP_MESSAGE_HANDLER (MAPI) (continued)	SimplePushModeratorHandler (CAPI) (continued)
Main part of code	
<p>Note: The API has changed considerably, so the two implementations don't correspond closely. Refer to the migration reference for information about how each MAPI event has changed. The following shows the syntax differences between handling events in MAPI and CAPI.</p>	
<p>MAPI method: public void handleCONF_STARTED(CONF_STARTED event) throws OD_EXCEPTION</p>	
<p>CAPI version: public void handle(RVAPICONF_INFO_CHANGED message)</p>	
<p>MAPI method: public void handleMOD_LOGIN_ACK(MOD_LOGIN_ACK event) throws OD_EXCEPTION</p>	
<p>CAPI version: public void handle(RVAPINACK message)</p>	
<pre> //////////////////////////////////// MOD LOGIN ACK ////////////////////////////////////// // Corresponds to SUBSCR_INFO message on page 129 public void handleMOD_LOGIN_ACK(MOD_LOGIN_ACK event) throws OD_EXCEPTION { //// // Store addresses of OP_QUEUE_MGR, CALL_ROUTER, and self address // This information is needed for a conference. Each conference object // will store this information individually. //// mod.msg_sender.setMyAddress(event.getDestination()); mod.msg_sender.setOqmAddress(event.getSource()); mod.msg_sender.setCrAddress(event.getCallRouterAddr()); //// // Create conference object //// mod.conf = new Conference(event); //// // Set a conference reference to MESSAGE_SENDER object //// mod.msg_sender.setConference(mod.conf); } </pre>	<pre> public void handle(RVAPINACK message) { log.debug("Got ACK to message" + message); switch(message.getEvent().getContent().getType()) { /* * The login failed for some reason, we can't do anything else, * exit the application. */ case RVSIMPLE_API_EV_TYPE.ET_LOGIN_PUSH_MODERATOR_TYPE: moderator.setLoggedIn(false); log.fatal("Could not login, NACK recieved reason[" + message.getNackReason().getContent().toString() + "] Message[" + message.getMessage().getContent() + "]); moderator.shutdown(); break; case RVSIMPLE_API_EV_TYPE.ET_START_CONF_TYPE: moderator.setConfStarted(false); log.warn("Could not start conference, NACK recieved reason[" + message.getNackReason().getContent().toString() + "] Message[" + message.getMessage().getContent() + "]); break; } } </pre>

Table A-2 MAPI to CAPI comparison: Simple Moderator Handler (continued)

SIMP_MESSAGE_HANDLER (MAPI) (continued)	SimplePushModeratorHandler (CAPI) (continued)
<pre> ////////////////////////////////// LOGIN NACK //////////////////////////////////// // Corresponds to NACK message on page 128 // for each "void handle$methodname$ method" here, look for an equivalent "public void handle" in CAPI public void handleLOGIN_NACK(LOGIN_NACK event) { //// // Login event has been rejected // look for reson code. //// System.out.println("Login failed: " + event.toString()); } </pre>	<pre> /* * You could attempt a re-login here instead of failing */ case RVSIMPLE_API_EV_TYPE.ET_SESSION_HEARTBEAT_TYPE: log.fatal("Session heartbeat failed, shutdown"); moderator.shutdown(); break; default: break; } } public void handle(RVAPISUBSCR_INFO message) { log.debug("Got a SUBSCR_INFO: " + message.toString()); /* * If this event is a reply to a login request, mark us as logged in * setting our session and subscription ID values. */ if(message.getEvent().getContent().getType() == RVSIMPLE_API_EV_TYPE.ET_LOGIN_PUSH_MODERATOR_TYPE) { if(!moderator.isLoggedIn()) { moderator.setSessionId(message.getSessionId().getContent()); moderator.setSubscriberId(message.getSubscrId().getContent()); moderator.setLoggedIn(true); } } } </pre>

Table A-2 MAPI to CAPI comparison: Simple Moderator Handler (continued)

SIMP_MESSAGE_HANDLER (MAPI) (continued)	SimplePushModeratorHandler (CAPI) (continued)
<pre> ////////////////////////////////////// CONF STARTED // // Corresponds to CONF_INFO_CHANGED message on page 131 public void handleCONF_STARTED(CONF_STARTED event) throws OD_EXCEPTION { // // The conference is already running as a result of START_CONF or dial in. // // // This is where the boolean value from the user interface Simple_Moderator.java // is set. // mod.conf_start = true; // // If a conference is started the below information needs to be set for that conference // This is all required information. // // // 1. Store addresses of BRDG_INTER_SRV // mod.msg_sender.setBrAddress(event.getBridgeAddr()); // // 2. Set Bridge address to Conference information // mod.conf.setBridgeAddress(event.getBridgeAddr()); </pre>	<pre> if(!moderator.isConfStarted()) { if(message.getConfRunning().getContent()) { moderator.setConfenceld(message.getConfId().getContent()); moderator.setConfStarted(true); } else { moderator.startConf(); } } } </pre>

Table A-2 MAPI to CAPI comparison: Simple Moderator Handler (continued)

SIMP_MESSAGE_HANDLER (MAPI) (continued)	SimplePushModeratorHandler (CAPI) (continued)
<pre> //// // 3. Set Bridge Id to Conference information //// mod.conf.setBridgeld(event.getBridgeld()); //// // 4. Register for this conference. We will received updates about the conference // whenever there is a change (new participant, mute, lock, etc.). //// mod.msg_sender.registerConf(); } ////////// CONF CANNOT STARTED ////////// // Corresponds to NACK message on page 128 public void handleCONF_CANNOT_STARTED(CONF_CANNOT_STARTED event) throws OD_EXCEPTION { //// // Cannot start the conference //// System.out.println("Conference failed to start: " + event.toString()); } </pre>	<pre> public void handle(RVAPICONF_INFO_CHANGED message) { log.debug("Conf Info Changed..."); /* * If this event is telling us that a conference has started, inform the * moderator accordingly */ if(message.getCiReason().getContent().getType() == RVAPISIMPLE_CI_REASON.CIR_CONF_STARTED_TYPE) { if(!moderator.isConfStarted()) { moderator.setConfenceld(message.getConfId().getContent()); moderator.setConfStarted(true); } } } /* * This may be sent periodically notifying us of conference information. We * will only get this if the conf is running, so we should inform the * moderator as such if it isn't already. */ public void handle(RVAPICONF_INFO message) { log.debug("Got conf info..." + message.toString()); if(!moderator.isConfStarted()) { moderator.setConfenceld(message.getConfId().getContent()); moderator.setConfStarted(true); } } // Continued on page 135 </pre>

Table A-2 MAPI to CAPI comparison: Simple Moderator Handler (continued)

SIMP_MESSAGE_HANDLER (MAPI) (continued)	SimplePushModeratorHandler (CAPI) (continued)
<pre> ////////////////////////////////// PORT KNOWN //////////////////////////////////// public void handlePORT_KNOWN(PORT_KNOWN event) throws OD_EXCEPTION { //// // If the participant's port is known, assign the proper values // //// System.out.println("Getting Caller's by seq num"); //// // Get part info //// PART_INFO_WITH_NAME part_info = mod.conf.getPartBySequenceId(event.getSequenceNumber().getValue()); part_info.setCallerPortId(event.getCallerPortId()); part_info.setBridgId(event.getBridgId()); System.out.println("Updating Participant"); //// // Update participant. This actually assigns the information obtained. //// if(!mod.conf.updatePartNamePortId(part_info.getCallerPortId(), part_info.getName(), true)) { mod.conf.updateDiscPartByPhone(part_info.getCallerPortId(), part_info.getPhone().getValue(), part_info.getName()); } System.out.println("Participant's Port is assigned. Ready to GO."); } </pre>	

Table A-2 MAPI to CAPI comparison: Simple Moderator Handler (continued)

SIMP_MESSAGE_HANDLER (MAPI) (continued)	SimplePushModeratorHandler (CAPI) (continued)
<pre> //////////////////////////////////// CALL CONNECTED ////////////////////////////////////// public void handleCALL_CONNECTED(CALL_CONNECTED event) { //// // If the caller we dialed connects this will join them to the conference.// //// try { PART_INFO_WITH_NAME part1 = mod.conf.getPartByPortId(event.getCallerPortId()); System.out.println("Your Call is connected. Joining participant to conference. *****"); mod.msg_sender.joinConf(part1); } catch (Exception e) { System.out.println(e.getMessage()); } } //////////////////////////////////// CALL FAILED ////////////////////////////////////// public void handleCALL_FAILED(CALL_FAILED event) { System.out.println("Call failed"); } </pre>	

Table A-2 MAPI to CAPI comparison: Simple Moderator Handler (continued)

SIMP_MESSAGE_HANDLER (MAPI) (continued)	SimplePushModeratorHandler (CAPI) (continued)
<pre> ////////////////////////////////// PARTICIPANT INFO //////////////////////////////////// public void handlePARTICIPANT_INFO(PARTICIPANT_INFO event) throws OD_EXCEPTION { //// //For setting up participant. In this case there is no password for participants. //// try { PART_INFO_WITH_NAME part1 = mod.conf.getPartByPortId(event.getCallerPortId()); //// // Set the conference password if this is the subscriber (actually no password in this case). //// if (mod.conf.isChair(part1) && event.getPartStatus().getValue() == PART_STATUS.PS_CONF) { //// // * means no security code //// mod.msg_sender.setSecurity(""); } } catch (Exception e) { System.out.println(e.getMessage()); } } </pre>	

Table A-2 MAPI to CAPI comparison: Simple Moderator Handler (continued)

SIMP_MESSAGE_HANDLER (MAPI) (continued)	SimplePushModeratorHandler (CAPI) (continued)
<pre> ////////////////////////////////// PARTICIPANT DISCONNECT //////////////////////////////////// // Corresponds to CALL_DISCONNECTED at right public void handlePARTICIPANT_DISCONNECT(PARTICIPANT_DISCONNECT event) throws OD_EXCEPTION { System.out.println("Participant Has left *****"); } ////////////////////////////////// CONFERENCE END //////////////////////////////////// // Corresponds to CONF_ENDED at right public void handleCONFERENCE_END(CONFERENCE_END event) throws OD_EXCEPTION { // This is not needed, but it is nice to stop the program when the conference stops // rather than ctrl-c breaking out of the program System.out.println("The conference has stoped *****"); System.exit(1); } // unimplemented methods are omitted </pre>	<pre> // Conference ends public void handle(RVAPICONF_ENDED message) { log.debug("Got Conf ended..." + message.toString()); moderator.shutdown(); } public void handle(RVAPIPART_INFO_CHANGED message) { if(message.getPiReason().getContent().getType() == RVAPISIMPLE_PL_REASON.PIR_CONNECTED_TO_CONF_TYPE) { moderator.setDialConnectCount(moderator.getDialConnectCount() + 1); } } public void handle(RVAPICALL_DISCONNECTED message) { log.debug("got part disconnected msg..." + message.toString()); moderator.setDialConnectCount(moderator.getDialConnectCount() - 1); } public void handle(RVAPISESSION_HEARTBEAT_ACK message) { log.debug("got session heartbeat ack..." + message.toString()); } } </pre>

Index

A

AAPI

- connecting to Admin server 96
- events 98
- helper classes 99
- introduction 95
- overview 3
- responses 99
- SDK components 97

about this manual vii

ACM

- application IDs, configuring 30
- application types 37
- CDRs 39
- communications 39
- conference user data 43
- events 38
- implementation basics 37
- setting up application 40
- using 65

ACMSender class 25

AddressBookSender class 25

AdminApiUnmarshaller class 97

AdminMessageSender class 97

Apache 9, 45

API

- overview 1

APP_REGISTER_CONFS_ACTIVITY 53

APP_UNREGISTER_CONFS_ACTIVITY 53

Application login 15

Application session 15

architecture

- CAPI Java classes 23
- object model layer 30
- overview 8

AsyncReceiverConnection class 24

B

BasicCookie class 24

BasicListener class 24

BasicMessageBus class 24

blast dialing 55, 58

bridge interface server 9

bridges 9

C

CACS

- connecting to 46
- defined 9
- logging into 50
- logging out 68

Call Flow ACM application 37

CALL_CONNECTED_TO_CHANGED 58

callflows

- dialing participants 55
- overview 55

CAPI

- architecture 8
- communication modes 10
- conferencing procedures 43
- developing application 7
- development guidelines 29
- development steps 8
- events overview 20
- Java classes diagram 23
- Java classes overview 21
- Java classes quick reference 24
- language to use 30
- migrating MAPI to 101
- sample application 69
- SDK components 20

CAPI overview 1

CapiUnmarshaller class 25

Castor

- event files overview 22
- Java utility classes 22, 24, 98
- naming conventions 27, 98

CastorAcmSender class 25

CastorAddressBookSender class 25

CastorAdminMessageHandlerBase class 97

CastorAdminSender class 97

- CastorAdminUnmarshaller class 97
 - CastorMessageHandlerBase class 25, 63, 69
 - CastorSender class 25
 - CastorUnmarshaller class 25
 - CF_BLAST 55, 58
 - CF_DEFAULT 55
 - CF_INSTANT 55
 - CF_ONECLICK 55
 - classes
 - A-API helper classes 99
 - A-API quick reference 97
 - CAPI quick reference 24
 - naming conventions 27, 98
 - overview 21
 - communication modes 10
 - conference IDs 33, 53, 64
 - conference information updates 33
 - conference user data 43
 - conferences
 - creating 16, 52
 - ending 67
 - joining 58
 - recording 56
 - security 35
 - security code 54
 - start and stop notifications 53
 - conferencing procedures 43
 - connecting to CACS 46
 - dialing participants 55
 - ending conference 67
 - identifying talkers 64
 - initial 45
 - joining someone to 58
 - logging 45
 - logging into CACS 50
 - logging out 68
 - main 52
 - maintaining session 48
 - notes 44
 - reading events 63
 - saving IDs 53
 - sending events 63
 - sequence 43
 - setting push response port 59
 - starting conference 52
 - using ACM 65
 - waiting room 58
 - Config object 30
 - configuration information 30
 - Connection class 24
 - connections
 - and sessions 17
 - creating 46
 - defined 14
 - keeping open 48
 - number to use 19
 - overview 14
 - relationships to sessions and logins 16
 - conventions used in manual viii
 - Cookie class 24
 - core communication classes 22
 - CSC 9, 33
 - customer support ix
- D**
- database updates 33
 - developing a CAPI application 7
 - developing an A-API application 95
 - development
 - CAPI procedures 43
 - conferencing procedures 43
 - events, reading 63
 - events, sending 63
 - guidelines 29
 - language to use 30
 - migration 103
 - object model layer 30
 - passback field 34
 - prerequisites to conference tasks 34
 - push and pull, using 31
 - sample application 69
 - verifying events 34
 - development steps, CAPI 8
 - dial string requirements 36
 - DIAL_CALL 55
 - dialing
 - callflows overview 55
 - procedure 55
 - DISCONNECT_CALL 59
 - document
 - conventions viii
 - purpose vii
 - documentation
 - accessing 6
 - overview 6
- E**
- END_CONF 67
 - enumerated types
 - naming conventions 27, 98
 - event references 21

- events
 - A-API and Java utility classes 98
 - CAPI and Java utility classes 26
 - limitations 20
 - naming conventions 27, 98
 - overview 20
 - passback field 34
 - push and pull modes 20
 - reading 63
 - responses and notifications 21, 99
 - sending 63
 - verifying 34
- F**
- firewalls 12
 - connections and 19
 - ports and 12
 - push mode and 32
 - push versus pull 31
 - responses and 19
- G**
- getting help ix
- getting started 3
 - A-API 95
 - CAPI 8
- Global Services ix
- GUL.INI file 30
- H**
- heartbeats
 - configuring 30
 - session, keeping open 48
 - timeout for session and connection 32
- help, getting ix
- helper classes, A-API 99
- HTTP
 - connection 14, 95, 96
 - malformed 35
- HttpConnection class 24
- HttpResponseMessage class 24
- HttpsConnection class 24
- I**
- IDs
 - conference 33, 53
 - participant 33
 - subscriber 33, 53
- In Conference ACM application 37
- installation 4, 5
- instant dialing 55
- introduction 1
 - to A-API 95
 - to CAPI 1
 - to manual vii
- IP address
 - Apache server 9
 - configuring 30
- J**
- Java
 - A-API Java classes quick reference 97
 - CAPI Java classes diagram 23
 - CAPI Java classes overview 21
 - CAPI Java classes quick reference 24
 - naming of utility classes and events 24
 - version required 4
- Java utility classes
 - A-API quick reference 97
 - and A-API events 98
 - and CAPI events 26
 - CAPI quick reference 24
 - naming conventions 24
 - sender files 63
- JOIN_PART_TO_CONF 56, 58, 59
- joining a conference 58
- L**
- Listener class 25
- logging 45
- logging into the CACS 50
- logging out of the CACS 68
- LOGIN_APPLICATION 50
- LOGIN_MODERATOR 50
- LOGIN_PART 50
- LOGIN_PUSH_MODERATOR 50
- logins
 - defined 15
 - events for 15
 - examples 15
 - overview 14
 - relationships to sessions and logins 16
- LOGOUT 68
- M**
- manual
 - conventions viii
 - overview vii
 - purpose vii
- M-API
 - migrating applications to CAPI 101
 - sample application 103

MarshallerException class 25
 MessageBus class 25
 MessageSender class 25
 methods 26
 migration 101–135
 MAPI and CAPI sample applications 103
 overview 101
 Moderator login 15

N

NACK_REASON values from Admin server 99
 naming conventions 27, 98
 notifications
 conference start and stop 53
 responses versus 21
 numbers, telephone 36

O

object model layer 30
 ODPROC
 defined 9
 odprocr file 32, 56
 ONE_TO_ONE 59
 one-click dialing 55
 overview 1
 document vii

P

parameters
 naming conventions 27, 98
 PART_INFO_CHANGED 56, 57, 58
 participant IDs 33
 Participant login 15
 participants
 dialing 55
 passback fields 34
 phone numbers 36
 PIR_CONNECTED_TO_CONF 56, 57, 58
 Polycom Global Services ix
 ports 19
 configuring 30
 firewalls and 12
 how used 11
 push mode and 11
 setting for asynchronous response 59
 prerequisites to conference tasks 34
 procedures
 conferencing 43
 migration 103

programming
 guidelines 29
 PT_RECORDER 55
 pull
 connection, creating 46
 defined 10
 events and 20
 passback field 34
 push versus 31
 specifying in application 31
 using 31

Pull Moderator login 15
 connection type 50

push
 connection, creating 46
 defined 10
 events and 20
 firewalls and 32
 passback field 34
 port, setting for response 59
 ports and 11
 pull versus 31
 specifying in application 31
 using 31

Push Moderator login 15

R

ReadVoice
 architecture 8
 recording conferences 56
 enum and event 55
 REGISTER_TALKER_UPDATE 64
 responses
 event responses, understanding 21
 firewalls and 19
 notifications versus 21
 ports for push mode 12
 session IDs and 14
 setting port for push response 59
 to AAPI events 99
 RVADMIN prefix 98
 RVAPI prefix 27

S

sample application
 MAPI and CAPI comparison 103–135
 simple Moderator 69
 sample code
 ACM, using 65
 blast dialing 57
 conference and subscriber IDs, saving 53
 conference, ending 67

- conference, starting 52
 - connecting to the CACS 46
 - events, reading 64
 - events, sending 63
 - logging into the CACS 51
 - logging out of the CACS 68
 - logging setup 45
 - open connection and sessions, maintaining 49
 - port for asynchronous response 60
 - talkers, identifying 65
- SDK**
- A-API Java classes quick reference 97
 - C-API Java classes quick reference 24
 - components 20
 - documentation included 6
 - installing 4
 - Java classes diagram 23
 - Java classes overview 21
 - overview 1
 - software included 4
- security 13
- conference security code 35, 54
- sender Java classes 63
- server
- logging into 50
 - logging out 68
- SESSION_HEARTBEAT 48
- sessions
- and connections 17
 - defined 14, 15
 - heartbeat, configuration 30
 - keeping open 48
 - overview 14
 - relationship to logins and connections 16
- SET_CONF_SECURITY_CODE 54
- SSL 13
- START_CONF 52
- status, changing participant's 56
- subscriber IDs 33, 53
- support ix
- system requirements 4
- T**
- talkers 64
 - telephone numbers 36
 - Thread class 69
 - timeouts, session and connection 32
 - transport layer security 13
- U**
- UnmarshallerException class 25
 - UNREGISTER_TALKER_UPDATE 64
 - updating conference information 33
 - user data, conference 43
 - utility classes, *see* Java utility classes
- W**
- waiting room procedure 58
- X**
- XML, malformed 35

