Media Interaction SDK 7.6

Java

Developer’s Guide

About Genesys

Genesys Telecommunications Laboratories, Inc., a subsidiary of Alcatel-Lucent, is 100% focused on software for call centers. Genesys recognizes that better interactions drive better business and build company reputations. Customer service solutions from Genesys deliver on this promise for Global 2000 enterprises, government organizations, and telecommunications service providers across 80 countries, directing more than 100 million customer interactions every day. Sophisticated routing and reporting across voice, e-mail, and Web channels ensure that customers are quickly connected to the best available resource—the first time. Genesys offers solutions for customer service, help desks, order desks, collections, outbound telesales and service, and workforce management. Visit www.genesyslab.com for more information.

Each product has its own documentation for online viewing at the Genesys Technical Support website or on the Documentation Library DVD, which is available from Genesys upon request. For more information, contact your sales representative.

Notice

Although reasonable effort is made to ensure that the information in this document is complete and accurate at the time of release, Genesys Telecommunications Laboratories, Inc., cannot assume responsibility for any existing errors. Changes and/or corrections to the information contained in this document may be incorporated in future versions.

Your Responsibility for Your System's Security

You are responsible for the security of your system. Product administration to prevent unauthorized use is your responsibility. Your system administrator should read all documents provided with this product to fully understand the features available that reduce your risk of incurring charges for unlicensed use of Genesys products.

Trademarks

Technical Support from VARs

If you have purchased support from a value-added reseller (VAR), please contact the VAR for technical support.

Technical Support from Genesys

If you have purchased support directly from Genesys, please contact Genesys Technical Support at the following regional numbers:

Region	Telephone	E-Mail
North and Latin America	+888-369-5555 or +506-674-6767	support@genesyslab.com
Europe, Middle East, and Africa	+44-(0)-1276-45-7002	support@genesyslab.co.uk
Asia Pacific	+61-7-3368-6868	support@genesyslab.com.au
Japan	+81-3-6361-8950	support@genesyslab.co.jp

Prior to contacting technical support, please refer to the *Genesys Technical Support Guide* for complete contact information and procedures.

Ordering and Licensing Information

Complete information on ordering and licensing Genesys products can be found in the ***Genesys 7 Licensing Guide*.**

Released by

Genesys Telecommunications Laboratories, Inc. www.genesyslab.com

Document Version: 76sdk_dev_ixn_java-media_09-2008_v7.6.101.00

Table of Contents

Preface

Welcome to the *Media Interaction SDK 7.6 Java Developer's Guide*. This document introduces you to the concepts, terminology, and procedures relevant to the Genesys Media Interaction SDK 7.6 product.

This document provides a high-level overview of Genesys Media Interaction SDK 7.6 features and functions, together with software-architecture information and development-planning materials.

This document is valid only for the 7.6 release(s) of this product.

Note: For versions of this document created for other releases of this product, please visit the Genesys Technical Support website, or request the Documentation Library DVD, which you can order by e-mail from Genesys Order Management at `orderman@genesyslab.com`.

This preface provides an overview of this document, identifies the primary audience, introduces document conventions, and lists related reference information:

The Genesys Media Interaction SDK (Software Development Kit) is built around the Media Interaction Layer library, which presents an API for developing third-party media applications. The library provides connectivity with Genesys Multimedia servers, so that your applications can create and manage Open Media interactions.

Intended Audience

This guide, primarily intended for programmers developing Java-based applications for contact center agents, assumes that you have a basic understanding of:

- Network design and operation.
- Your own network configurations.

You should also be familiar with:

- Java programming.
- Genesys Multimedia 7.6 features.

Usage Guidelines

The Genesys developer materials outlined in this document are intended to be used for the following purposes:

- Creation of contact-center agent desktop applications associated with Genesys software implementations.
- Server-side integration between Genesys software and third-party software.
- Creation of a specialized client application specific to customer needs.

The Genesys software functions available for development are clearly documented. No undocumented functionality is to be utilized without Genesys's express written consent.

The following Use Conditions apply in all cases for developers employing the Genesys developer materials outlined in this document:

1. Possession of interface documentation does not imply a right to use by a third party. Genesys conditions for use, as outlined below or in the *Genesys Developer Program Guide,* must be met.
2. This interface shall not be used unless the developer is a member in good standing of the Genesys Interacts program or has a valid Master Software License and Services Agreement with Genesys.
3. A developer shall not be entitled to use any licenses granted hereunder unless the developer's organization has met or obtained all prerequisite licensing and software as set out by Genesys.
4. A developer shall not be entitled to use any licenses granted hereunder if the developer's organization is delinquent in any payments or amounts owed to Genesys.

5. A developer shall not use the Genesys developer materials outlined in this document for any general application development purposes that are not associated with the above-mentioned intended purposes for the use of the Genesys developer materials outlined in this document.
6. A developer shall disclose the developer materials outlined in this document only to those employees who have a direct need to create, debug, and/or test one or more participant-specific objects and/or software files that access, communicate, or interoperate with the Genesys API.
7. The developed works and Genesys software running in conjunction with one another (hereinafter referred to together as the "integrated solutions") should not compromise data integrity. For example, if both the Genesys software and the integrated solutions can modify the same data, then modifications by either product must not circumvent the other product's data integrity rules. In addition, the integration should not cause duplicate copies of data to exist in both participant and Genesys databases, unless it can be assured that data modifications propagate all copies within the time required by typical users.
8. The integrated solutions shall not compromise data or application security, access, or visibility restrictions that are enforced by either the Genesys software or the developed works.
9. The integrated solutions shall conform to design and implementation guidelines and restrictions described in the *Genesys Developer Program Guide* and Genesys software documentation. For example:
   a. The integration must use only published interfaces to access Genesys data.
   b. The integration shall not modify data in Genesys database tables directly using SQL.
   c. The integration shall not introduce database triggers or stored procedures that operate on Genesys database tables.

Any schema extension to Genesys database tables must be carried out using Genesys Developer software through documented methods and features.

The Genesys developer materials outlined in this document are not intended to be used for the creation of any product with functionality comparable to any Genesys products, including products similar or substantially similar to Genesys's current general-availability, beta, and announced products.

Any attempt to use the Genesys developer materials outlined in this document or any Genesys Developer software contrary to this clause shall be deemed a material breach with immediate termination of this addendum, and Genesys shall be entitled to seek to protect its interests, including but not limited to, preliminary and permanent injunctive relief, as well as money damages.

Chapter Summaries

In addition to this opening chapter, this document contains the following chapters:

- Chapter 1, "About the Media Interaction SDK," on page 13. Introduces the Media Interaction SDK, its components, features, and scope of use.
- Chapter 2, "About the Examples," on page 25. Introduces the code examples that accompany this developer's guide.
- Chapter 3, "Simple Media Server Example," on page 37. Explains the `SimpleMediaServer.java` example, a stand-alone application that connects to Genesys servers and submits new Open Media interactions.
- Chapter 4, "Simple Custom Extension Example," on page 45. Explains the `SimpleCustomExtension.java` example used to run the simple media server in custom mode.

Document Conventions

This document uses certain stylistic and typographical conventions—introduced here—that serve as shorthands for particular kinds of information.

Document Version Number

A version number appears at the bottom of the inside front cover of this document. Version numbers change as new information is added to this document. Here is a sample version number:

76fr_ref_09-2005_v7.6.000.00

You will need this number when you are talking with Genesys Technical Support about this product.

Type Styles

Italic

In this document, italic is used for emphasis, for documents' titles, for definitions of (or first references to) unfamiliar terms, and for mathematical variables.

Examples:

- Please consult the *Genesys 7 Migration Guide* for more information.
- *A customary and usual practice* is one that is widely accepted and used within a particular industry or profession.
- Do *not* use this value for this option.
- The formula, x +1 = 7 where x stands for . . .

Monospace Font

A monospace font, which looks like `teletype or typewriter text`, is used for all programming identifiers and GUI elements.

This convention includes the *names* of directories, files, folders, configuration objects, paths, scripts, dialog boxes, options, fields, text and list boxes, operational modes, all buttons (including radio buttons), check boxes, commands, tabs, CTI events, and error messages; the values of options; logical arguments and command syntax; and code samples.

Examples:

- Select the `Show variables on screen` check box.
- Click the `Summation` button.
- In the `Properties` dialog box, enter the value for the host server in your environment.
- In the `Operand` text box, enter your formula.
- Click `OK` to exit the `Properties` dialog box.
- The following table presents the complete set of error messages T-Server® distributes in `EventError` events.
- If you select true for the `inbound-bsns-calls` option, all established inbound calls on a local agent are considered business calls.

Monospace is also used for any text that users must manually enter during a configuration or installation procedure, or on a command line:

Example:

- Enter `exit` on the command line.

Screen Captures Used in This Document

Screen captures from the product GUI (graphical user interface), as used in this document, may sometimes contain a minor spelling, capitalization, or grammatical error. The text accompanying and explaining the screen captures corrects such errors *except* when such a correction would prevent you from installing, configuring, or successfully using the product. For example, if the name of an option contains a usage error, the name would be presented exactly as it appears in the product GUI; the error would not be corrected in any accompanying text.

Square Brackets

Square brackets indicate that a particular parameter or value is optional within a logical argument, a command, or some programming syntax. That is, the parameter's or value's presence is not required to resolve the argument, command, or block of code. The user decides whether to include this optional information. Here is a sample:

```
smcp_server -host [/flags]
```

Angle Brackets

Angle brackets indicate a placeholder for a value that the user must specify. This might be a DN or port number specific to your enterprise. Here is a sample:

```
smcp_server -host <confighost>
```

Related Resources

Consult these additional resources as necessary:

- *Interaction SDK 7.6 Java Deployment Guide,* which is delivered on the Documentation Library DVD.
- *Genesys Multimedia 7.6 User's Guide*, which contains step-by-step instructions for using the Interaction Workflow Designer (IWD) component of the Multimedia product.
- *Genesys Multimedia 7.6 Deployment Guide*, which outlines how to deploy a Genesys Multimedia solution in your contact center.
- *Genesys Multimedia 7.6 Universal Contact Server Manager Help* documentation, which introduces you to UCS.
- The *Genesys Technical Publications Glossary,* which ships on the Genesys Documentation Library DVD and which provides a comprehensive list of the Genesys and CTI terminology and acronyms used in this document.
- The *Genesys 7 Migration Guide*, also on the Genesys Documentation Library CD, which provides a documented migration strategy from Genesys product releases 5.1 and later to all Genesys 7.x releases. Contact Genesys Technical Support for additional information.
- The Release Notes and Product Advisories for this product, which are available on the Genesys Technical Support website at `http://genesyslab.com/support`.

Information on supported hardware and third-party software is available on the Genesys Technical Support website in the following documents:

- *Genesys 7 Supported Operating Systems and Databases*
- Genesys 7 Supported Media Interfaces

Genesys product documentation is available on the:

- Genesys Technical Support website at `http://genesyslab.com/support`.
- Genesys Developer website at `http://devzone.genesyslab.com`.
- Genesys Documentation Library DVD, which you can order by e-mail from Genesys Order Management at `orderman@genesyslab.com`.

Making Comments on This Document

If you especially like or dislike anything about this document, please feel free to e-mail your comments to Techpubs.webadmin@genesyslab.com.

You can comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this document. Please limit your comments to the information in this document only and to the way in which the information is presented. Speak to Genesys Technical Support if you have suggestions about the product itself.

When you send us comments, you grant Genesys a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

Chapter

1 About the Media Interaction SDK

This chapter introduces the Media Interaction SDK, its components, features, and scope of use. In this chapter you will find the following topics:

Features Overview

The Media Interaction SDK lets you build Java applications to manage third-party media interactions in the Genesys Framework.

The Media Interaction SDK presents a simple Java API, including manager interfaces for developing server applications that:

- Manage third-party media interactions submitted to the Interaction Server.
- Manage third-party media interactions in the Universal Contact Server's database.
- Use the ESP protocol (External Service Protocol) to handle interactions' extensions through the Interaction Server.
- Monitor your application's runmode from the LCA (Local Control Agent) component's point of view.

Components

The Media Interaction SDK comprises the following components:

- The Open Media Commons library, written entirely in the Java language, delivered as a set of `.jar` files on the product CD.
- The Media Interaction Layer (MIL) library, written entirely in the Java language, delivered as a set of `.jar` files on the product CD.
- The *Media Interaction SDK 7.6 Java API Reference*, which is an HTML tree in the `docs/` subdirectory of the installed product directory.
- The *Media Interaction SDK 7.6 Java Code Examples,* a set of code examples that exercise some important features of the API, delivered in `.zip` and `.tar.gz` formats on the documentation CD. For details, see "About the Examples" on page 25.
- *Open Media Interaction Application Blocks for Java,* available on the product CD, include application blocks to develop a media server. For further details, refer to the *Open Media Interaction SDK 7.6 Application Blocks Guide*.

Scope of Use

Contact center agents routinely work with applications that are separate from traditional CRM (Customer Relationship Management) or agent desktop applications. Genesys Open Media allows you to create custom media types that integrate this activity into the contact center workflow.

To use Open Media, you will need to define new interaction types in the Configuration Layer. After you do this, you can use the Media Interaction SDK to build client and server applications that manage them within the Genesys platform. Finally, you will use the Agent Interaction SDK to allow agents to process these interactions.

This section will provide an overview of the kinds of situations that lead application developers to use Open Media interactions. It will then outline some of the things you can do with Open Media using the Media Interaction SDK. Finally, it will describe some real-world Open Media use cases in depth.

Note: For more information on the Agent Interaction SDK, see the *Agent Interaction SDK 7.6 Java Developer's Guide*. For more information on multimedia interactions, see the *Multimedia 7.6 User's Guide* and the *Multimedia 7.6 Open Media Interaction Models Reference Manual*.

Bridging the Contact Center and the Enterprise

There are many occasions on which contact centers could benefit from doing work that is beyond their normal scope. Genesys Open Media makes such task expansion a lot more productive than it would otherwise be.

For example, in the wake of a natural disaster, insurance companies tend to receive high call volumes as people file their claims. At this point, you might want to have claims adjusters and others available to expand the contact center workforce.

But after most of the calls come in, the claims adjusters will have a lot of work to do on these new claims. With Genesys Open Media, you can route calls to adjusters when call volumes are high, and then you can route routine work from the adjusters to the contact center when call volumes are low.

Another common scenario is for the contact center to handle faxed requests when agents are idle. In this case, you could set up an interaction type that includes the fax data for the agent to process. When the interaction is routed to the agent's desktop, the agent can process the data as appropriate and then mark the item as done.

There are several ways you could use Open Media in these situations. In the simplest cases, you might want to use Open Media interactions merely to route work to an agent.

Basic Capabilities

The Media Interaction SDK's typical usage scenarios include:

- Managing third-party media interactions:
  - Creating interactions of third-party media types.
  - Submitting a third-party interaction to Interaction Server.
  - Stopping the processing of third-party interactions in Interaction Server.
  - Managing requests from Interaction Server through ESP (External Service Protocol).
- Managing third-party interaction data in UCS (Universal Contact Server):
  - Saving third-party interaction data in UCS.
  - Updating third-party interaction data in UCS.
  - Finding third-party interaction data in UCS.
- Monitoring changes in the LCA (Local Control Agent) runmode assigned to your application. (Runmodes are described in "LCA" on page 32.)
- Managing connections to Genesys servers, using the Open Media Commons library. Connection services involve the following components and protocols:
  - Interaction Server.
  - Configuration Layer.

- UCS (Universal Contact Server).
- LCA (Local Control Agent).
- ESP (External Service Protocol).

Routing Rejected Orders to an Agent

This example will show how you might use Genesys Open Media interactions simply as a routing mechanism, while having an agent continue to work in an application that is separate from his or her desktop.

Agents for a telephone company might use many applications, including an order management application. When a customer calls in to order DSL (a form of high-speed Internet service), an agent must enter the customer information and submit the order. If the agent makes a mistake on the order form, the system will reject the order and send it back to the agent for correction.

Because the order management system is not linked to the contact routing platform, the rejected order will sit at the agent's desktop—perhaps for hours—until there is a lull in inbound activity that allows the agent to address the problem.

With the addition of Genesys Open Media, the scenario is different. If the agent makes a mistake on an order, the rejected order can be submitted to the Genesys platform as a new activity to be queued and sent back to the agent.

Depending on the priority that you set, the agent may be taken off the phone queue immediately and routed back to the order to make corrections. Here is how you could use Genesys Open Media to create this kind of solution:

1. Define a new interaction type, `DSLOrder`, in the Configuration Layer.
2. Set up the appropriate routing for the new interaction type.
3. Use the Media Interaction SDK to write an application that can receive information about rejected orders from the order management system, and then submit an interaction including this information.
4. Use the Agent Interaction SDK to write agent desktop functionality that allows the agent to mark the `DSLOrder` interaction as done.

When an order is rejected, the order entry system can integrate the MIL library and submit a new interaction of type `DSLOrder`. Or it can send information about the order to a separate Media Interaction application, which will create a new interaction of type `DSLOrder`.

The Media Interaction application will submit the MIL interaction to the Genesys servers for processing and routing.

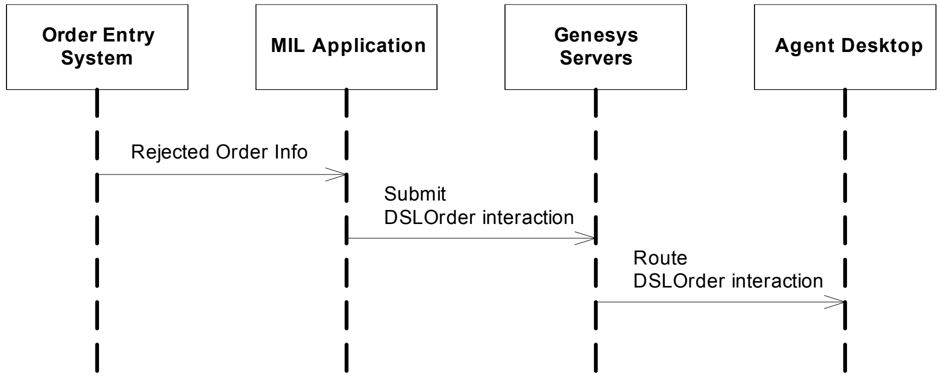

Figure 1: Submitting DSLOrder Interaction for Rejected Order

See “Simple Media Server Example” on page 37 for a code example that shows how to create and submit an Open Media interaction.

Working on a CRM Case

Sometimes it is not enough to use Open Media as a routing and notification mechanism. There are many cases where it makes more sense to write detailed interaction-handling functionality directly into the agent desktop.

For example, you can use Open Media interactions to allow contact center agents to handle incoming fax data for use in a Customer Relationship Management system. This could be done in a way that is very similar to the preceding example, but in this case, you might prefer that the inbound interactions that you create would also carry data for the agent to process right in his or her agent desktop application. Then, as a result of processing this data, the agent might need to fax the customer with data that the agent has filled in.To accomplish this, the agent might create an outbound interaction to carry data that allows the CRM server to handle the outgoing fax.

As in the example above, you would need to define two new interaction types, perhaps called `CRMCaseIn` for handling CRM inbound fax data and `CRMCaseOut` for handling outgoing CRM fax data. You would also need to set up the appropriate routing for these interactions.

Then, you would write a CRM server application that uses the Media Interaction SDK. This application would create `CRMCaseIn` interactions, to which it would attach inbound CRM fax data. In order to safeguard this information, your application would save this interaction, then submit it to the Genesys servers that would route it to the appropriate agent. This process is shown in Figure 2.

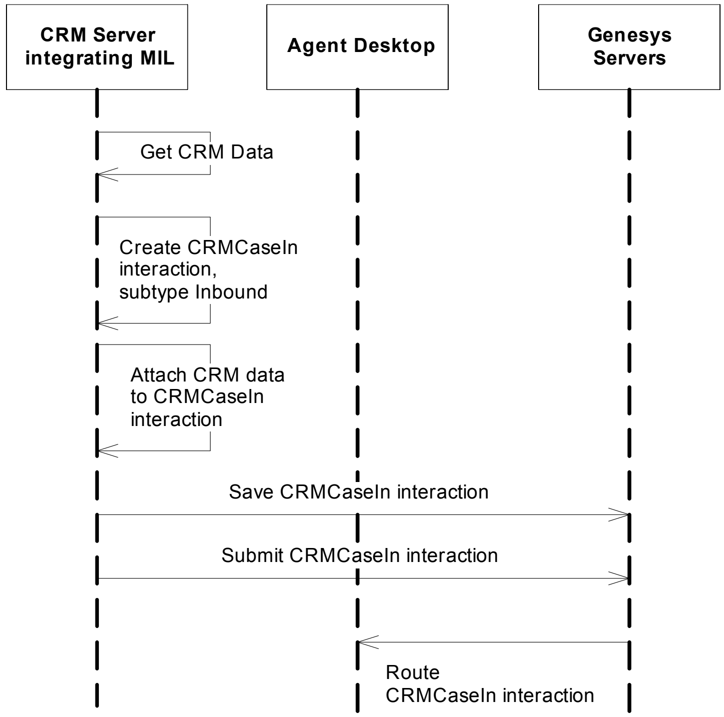

Figure 2: Handling an Inbound CRM Case

As a result of processing the `CRMCaseIn` interaction, the agent would create an `CRMCaseOut` interaction. The parent of this interaction would be the `CRMCaseIn` interaction. Your application would save the outbound CRM interaction and submit it to Genesys servers.

To handle this outbound interaction, you would need to define an external service in the routing strategy defined for the queue to which this interaction would be submitted. This external service would send an ESP (External Service Protocol) request to the CRM server, which would process it by handling an outgoing fax. Then, your application would stop the two CRM interactions, as shown in Figure 3.

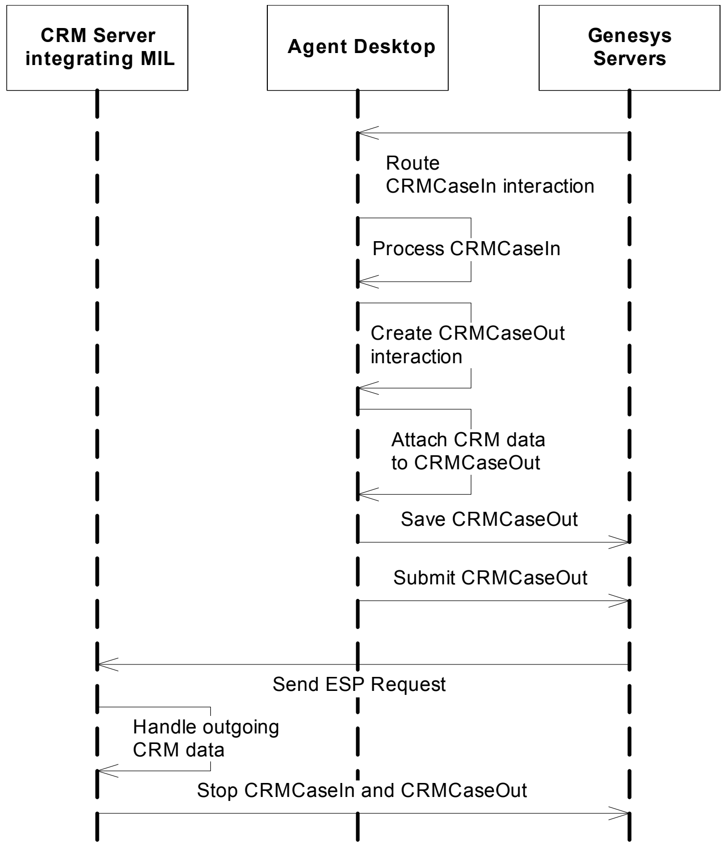

Figure 3: Handling an Outbound CRM Case

See "Simple Custom Extension Example" on page 45 for an example that shows how to manage an ESP request with MIL.

Architecture

The Media Interaction SDK is part of the 7.6 Open Media Interaction SDKs. Accordingly, it works with the Open Media Commons library to manage connections to the Genesys Framework and Genesys servers, as shown in Figure 4.

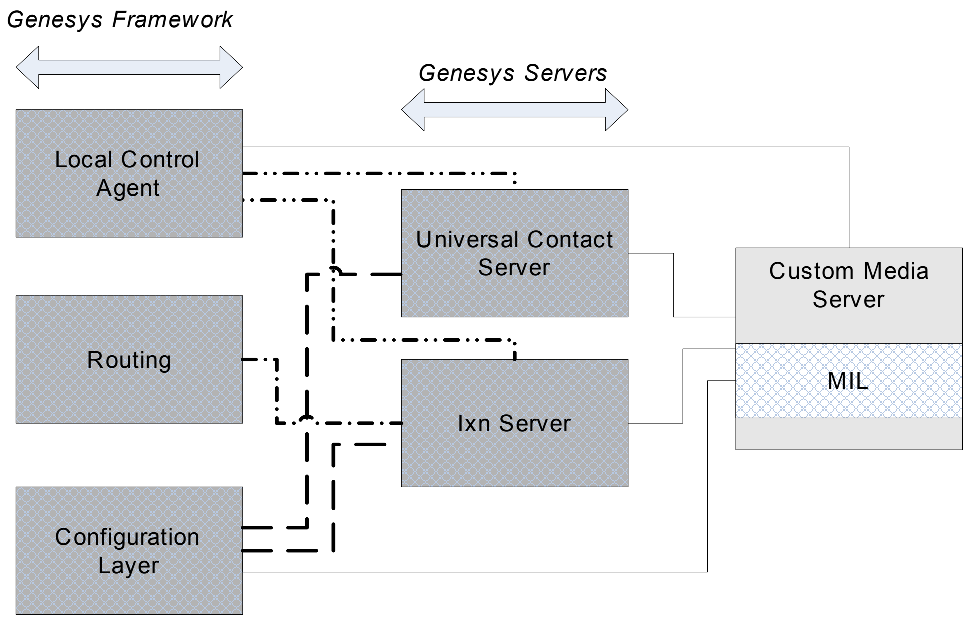

Figure 4: MIL Architectural Overview

Media Interaction (Java API) exposes objects—such as `MILInteraction`, `MILLCAManager`, and `MILUCSManager`—as interfaces that provide access to the information and features available through the connected services.

The MIL library core is responsible for maintaining all TCP/IP connections to servers, for maintaining the context, and for consolidating the data.

Interfaces to Core Objects

You do not access core objects of the Media Interaction SDK library directly. Rather, you get interfaces on them using the `MILFactory` or using another MIL interface.

Your application uses the `MILFactory` interface to initialize and access the internal core factory object. The `MILFactory` object is a singleton. Because of its singleton design, only one instance of the core factory object exists at runtime. All `MILFactory` interfaces refer to this same object.

Connectivity to Other Genesys Components

Connections to Genesys servers are maintained by the library core. There is a mechanism through which the Media Interaction SDK user can be notified of servers' statuses—namely, of the loss of a connection.

The MIL library core is designed to work in a single-tenant environment. See the *Interaction SDK 7.6 Java Deployment Guide* for details.

Performance of a single application built around Media Interaction (Java API) is satisfactory for a relatively large number of clients. However, that number will vary, depending mainly on network (interaction) activity and on platform features.

For example, as a guideline: When the number of simultaneous clients exceeds 200, check to verify that performance is satisfactory. To improve overall performance for your site, you may deploy multiple-server applications.

Configuration Layer

The Genesys Configuration Layer stores configuration information. To run a Media Interaction application, you must define its application parameters in the Configuration Layer. See the *Interaction SDK 7.6 Java Deployment Guide* for details.

Interaction Server

Interaction Server manages interactions and queues. The MIL library core communicates with Interaction Server to manage Open Media interactions in queues.

Additionally, Interaction Server can submit requests to the application integrating the library through External Service Protocol.

Universal Contact Server

Universal Contact Server (UCS) manages contact-related information. It uses the UCS database to store contact data (such as name, telephone number, and so on), and interactions.

Local Control Agent

Local Control Agent (LCA) is a Genesys component for managing the runtime status of Genesys applications. It enables you to build an application that has different runmodes, and to provide facilities to deal with these runmodes.

For further details, see “LCA” on page 32, which gives examples of a pair of modes.

API Overview

The API presents a common programmatic interface to manage your own interactions’ life cycle in the Genesys environment, including data creation and storage in the UCS database. It also provides high flexibility and interactivity with routing strategies, by managing requests and responses to Interaction Server through External Service Protocol (ESP).

Your application design is largely a matter of managing data for interactions that you create, and communicating with Genesys servers using the MIL manager interfaces. MIL also provides packages to monitor services that handle connections to servers. You do this by implementing event listeners on objects. This section provides you with high-level information about the API's features.

Packages

The *Media Interaction SDK 7.6 Java API Reference* (whose home page is `index.html` in the installation directory's `docs/` subdirectory) shows that the API comprises the following packages:

- `com.genesyslab.omsdk.commons`—Exposes the Open Media Commons classes for connecting servers, accessing connection services, and getting application information.
- `com.genesyslab.omsdk.commons.event`—Exposes classes and interfaces related to event notification in connections.
- `com.genesyslab.omsdk.commons.exception`—Exposes exceptions thrown by Open Media Commons API methods.
- `com.genesyslab.omsdk.mil`—Exposes the main MIL API classes.
- `com.genesyslab.omsdk.mil.event`—Exposes classes and interfaces related to event notification in MIL.
- `com.genesyslab.omsdk.mil.exception`—Exposes exceptions specific to MIL API methods.

Events and Listeners

The Media Interaction Layer and Open Media Commons library cores provide the push model through the Observer pattern. For instance, objects such as `OMSDKConnector` and `MILFactory` implement the pattern.

The following sections gives you further details about the Observer pattern and the event-thread implementation that Genesys recommends for MIL.

Event Push Model

The mechanism is that of sending an event on an object to a listener. This permits each object to implement its own set of listeners and methods. Generally, a listener declares only one method, a `handle*Event()` method, which takes an event interface as an inbound parameter. The inbound event interface is highly dependent on the original interface for which it is intended.

Threads and Listeners

Commons service events are time-ordered and should be published in listeners as soon as they occur, to ensure workflow and information consistency. Events occurring for a service cannot block events of another service.

If you want to perform an extended treatment, or a treatment making calls to MIL methods, be sure that your application implements such code in a separate thread, as illustrated in the following code snippet:

```
// Avoid:
public void handleXXXEvent(XXXEvent myEvent){
   ///...
   // my treatment
   ///...
}

// Prefer:
public void handleXXXEvent(XXXEvent myEvent){
   java.lang.Runnable treatEvent = new java.lang.Runnable() {
      public void run() {
         //...
         // my treatment
         ///...
      }
   }
   java.lang.Thread doTreatment = new java.lang.Thread(treatEvent);
   doTreatment.start();
}
```

The above code snippet shows one example of thread implementation. You should choose the thread implementation that best fits your application requirements.

What's Next

The next chapter goes into further details about the examples provided with this SDK. It provides installation instructions and gives a basic explanation of the supported features.

Chapter

2 About the Examples

This chapter introduces the code examples that accompany this developer's guide. It presents essential design considerations, and outlines some initial tasks that an application performs in using Media Interaction (Java API). The chapter contains the following sections:

Overview of the Code Examples

All examples are Java applications that integrate a specific set of features provided with MIL.

- `OpenMediaSDKData`—reads the `OpenMediaSDK.properties` file containing connection data.
- `SimpleConnector`—establishes the connections to servers, based on `OpenMediaSDKData` contents. This example illustrates how to connect, using the Open Media Commons library.
- `SimpleMediaServer`—parses a source directory, and for each file, submits an interaction (containing the file) as attached data to Interaction Server.
- `StartMedia`—launches the `SimpleMediaServer` application.

Installing the Code Examples

In order to develop applications with Media Interaction (Java API), you will need a compiler, such as the one delivered in the JAVA 2 Standard Edition SDK. It must conform to the 1.4.2 or 1.5 release.

In this guide, the JDK 1.4.2 from Sun Microsystems was used to compile and run the code examples.

Before you install and use the examples, install Media Interaction SDK Library. Refer to the *Interaction SDK 7.6 Java Deployment Guide* for details.

Then, set the following environment variables:

- Specify all Media Interaction SDK `.jar` files in the `CLASSPATH` environment variable.
- Specify the location of the Java Runtime Environment in the `JAVA_HOME` environment variable.

Source-Code Examples

On the Genesys Documentation CD, you will find the example source-code files. When you expand the `sdk_exmpl_ixn_java-media` archive file containing the code examples, you will find the following directory structure:

- The top-level directory contains the following files:
  - `README.html` provides instructions for compiling and running the examples.
  - `compile.sh` and `compile.bat` are shell scripts (respectively for UNIX and Windows) that, with a little editing, you can use to compile the examples. They take a single argument, which is the name of the example you want to compile (without the `.java` extension).
  - `go.sh` and `go.bat` are shell scripts (respectively for UNIX and Windows) that, with a little editing, you can use to run the examples. They take no arguments.
  - An `OpenMediaSDK.properties` file (used by the `OpenMediaSdkData` class in `OpenMediaSdkData.java`).
- The `classes/` directory is where the scripts store or access compiled classes.
- Source files are stored in the `media/sdk/java/examples/` directory.
- There is a `doc/` directory containing Javadoc comments for each of the examples.

Using the Code Examples

The examples are designed to run with the Genesys Configuration Layer, Interaction Server, and Universal Contact Server.

To run successfully, the examples provided for this document need valid configuration data. This includes connections to servers, and such configuration objects as business attributes for open media interactions (interaction type, subtype, and so on).

For configuration details, see the *Interaction SDK 7.6 Java Deployment Guide*.

Introducing the Media Interaction Code Examples

The code examples were designed to isolate API-related code from presentation-related code as much as possible. This design should make it easier for you to learn the Media Interaction SDK's functionality.

In order to isolate the API code, separate classes have been set up to read properties information, as shown in Figure 5 on page 28. As you are learning the API functionality, you can ignore the `OpenMediaData` class.

The examples include a single class that connects your application to the servers. This class is called `SimpleConnector` and will be explained in the next section.

The examples also include the `SimpleMediaServer` class, which is explained in Chapter 3, "Simple Media Server Example," on page 37. This class implements a simple media server that demonstrates the submission of new open media interactions.

Another important class is the `SimpleCustomExtension` class, which manages ESP requests. This class is discussed in Chapter 4, "Simple Custom Extension Example," on page 45.

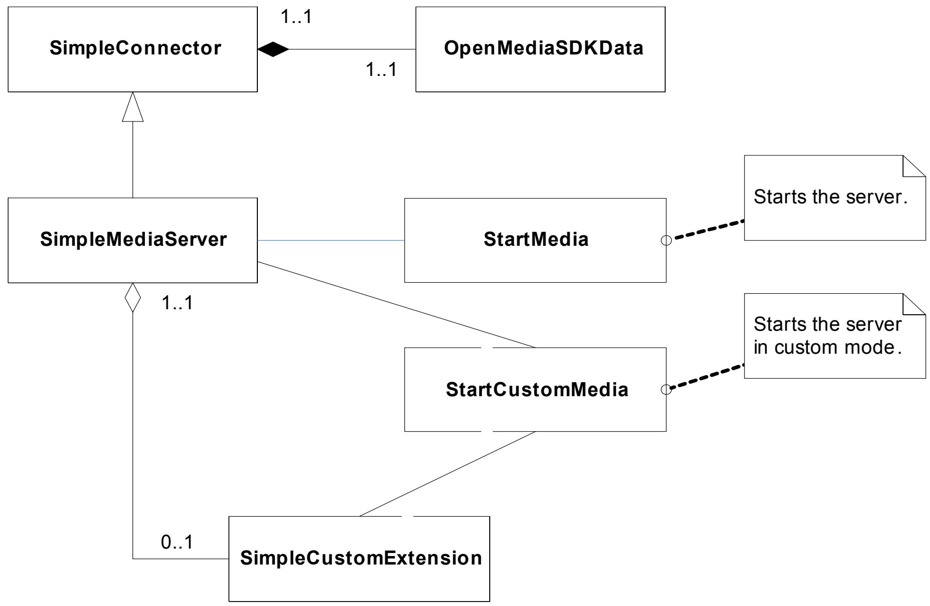

Figure 5: Architectural Overview of the Media Interaction Examples

If you have comments on these examples, please contact Genesys, as described in "Making Comments on This Document" on page 11.

Open Media Commons

The Open Media Commons library includes all material to instantiate connections to the Genesys servers. This section discusses the Open Media Commons library's contents, including API connection features that are essential for every application.

This discussion refers to the `SimpleConnector.java` example in the `sdk_exmpl_ixn_java-media/media/sdk/java/samples/` directory. This example is not a stand-alone application: the `SimpleMediaServer` class uses it to handle all connection-related tasks.

Before running the examples, be sure to edit the `OpenMediaSDK.properties` file to specify the actual data in your Configuration Layer (host, port, application name, and so on.) You will also need to compile `OpenMediaSDKData.java` and `SimpleConnector.java`, in addition to compiling `SimpleMediaServer.java`.

`SimpleConnector.java` is a very simple class that shows how to initialize connections using the `OMSDKConnector` interface.

The remainder of this section focuses mostly on classes and interfaces of the `com.genesyslab.omsdk.commons.*` packages.

Commons with Connection

Every MIL application must use the `OMSDKConnector` class to initialize the Open Media Commons library, passing correct configuration data arguments.

To make connections possible and start the Open Media Common library, your application will need to do two basic things:

- Set initialization parameters.
- Initialize the library.

To set initialization parameters, you need to create and fill in an `InitializationParameters` instance. Before that, you must set or obtain the following minimum configuration data:

- Configuration layer host name.
- Configuration layer port.
- Reconnection period.
- Maximum number of reconnection attempts.

In the `SimpleConnector` example, getting this information is a task passed on to the `OpenMediaSdkData` class. All the following code snippets are from the `SimpleConnector` constructor:

```
//Getting parameters for connecting Configuration Layer
OpenMediaSdkData data = new OpenMediaSdkData();

//Creating the required InitializationParameters
InitializationParameters ip = new InitializationParameters(
            data.getConfigServerPrimaryHost(),
            data.getConfigServerPrimaryPort(),
            data.getConfigServerBackupHost(),
            data.getConfigServerBackupPort(),
            data.getApplicationName(),
            data.getReconnectionPeriod(),
            data.getReconnectionAttempts());
```

Then, to properly fill in this object, you must specify the correct list of services that your application will use. Therefore, you get an `InitializationServices` instance by calling the static `MILFactory.getInitializationServices()` method. The returned object contains all the services you need, and you can add it to your `InitializationParameters` object, as shown here:

```
ip.addInitializationServices(MILFactory.getInitializationServices() );
OMSDKConnector.initialize(ip);
```

Once you have initialized the Open Media Commons library through the `OMSDKConnector` interface, you can get a reference to this component to take advantage of its features, and you are ready to proceed to the initialization of the MIL library.

```
connector = OMSDKConnector.getInstance();
System.out.println("Open Media SDK Connector
                    initialized successfully.");
```

Commons with Services

Connections are represented as services available through the `OMSDKConnector` interface. When connecting, your application uses service features to monitor connections' states, and to take into account possible disconnections.

Types of Services

Service features are available for several types of services—see `ServiceType` for further details:

- `CONFIGURATION`—Connection to the Configuration Layer. This connection lets your application access configuration information.
- `INTERACTION_SERVER`—Connection to Interaction Server. This connection lets your application manage open media interactions.
- `UCS`—Connection to Universal Contact Server. This connection lets your application manage interactions in the UCS database.
- `LCA`—Connection to Local Control Agent. This connection lets your application update its status depending on LCA events.
- `ESP`—Connection to Interaction Server using ESP (External Service Protocol). This connection lets your application manage requests from Interaction Server.

Service Interfaces

The `OMSDKConnector` interface is the entry point to service features:

- It accesses each `ServiceInfo` instance that associates a `ServiceStatus` with a `ServiceType`.
- It lets your application associate a `ServiceListener` listener with a service type, in order to track status changes, which are propagated in `ServiceEvent` events.

For further details about interfaces, see the *Media Interaction SDK 7.6 Java API Reference*.

Media Interaction (Java API)

The Media Interaction Java (API) includes all material to manage open media interactions. This section discusses the main features available through the API, including MIL initialization needed for every application.

All interfaces and atomic methods provided with MIL for handling interactions allow you to build your own interaction workflow and manage interaction-related information (status, attached data, and so on).

As a consequence, when developing your applications, pay attention to interaction data synchronization through servers, such as Interaction Server and UCS.

This section focuses on classes and interfaces of the `com.genesyslab.omsdk.mil.*` packages.

MILFactory

The `MILFactory` is the entry point to the Media Interaction Layer API. You need this interface to initialize the library, then access other MIL interfaces.

The discussion refers to the `SimpleMediaServer.java` example in the `71_OpenMedia/media/sdk/java/examples/` directory.

Before initializing the MIL library, you must enable connections to servers with the `OMSDKConnector` class. See "Commons with Connection" on page 29 for further details.

Then, initialization is straightforward, as shown in this code snippet:

```
MILFactory.initialize(new MILInitializationParameters(null));
```

Then, call the `MILFactory.getMILFactory()` method to access MIL managers—for instance, `MILInteractionManager` to perform Interaction Server's actions, `MILUCSManager` to perform UCS actions, and so on.

Interaction Server

MIL provides several interfaces to manage your open media interactions through Interaction Server:

- `MILInteraction`—represents an open media interaction.
- `MILInteractionManager`—accesses and manages `MILInteraction` objects handled by Interaction Server.

Use the `MILInteraction` interface to set your interaction parameters and data. Mandatory parameters for submitting an `MILInteraction` are passed when calling the `MILInteractionManager.createInteraction()` method.

The MILInteractionManager interface provides synchronous and asynchronous methods to perform requests on interactions handled by Interaction Server:

- Submit an interaction with its interaction parameters.
- Stop processing interactions.
- Change submitted interactions' parameters.

Note: You cannot use MIL to manage Genesys multimedia interactions, that is, chat and email interactions.

For further details about these interfaces, see the *Media Interaction SDK 7.6 Java API Reference*.

The SimpleMediaServer example demonstrates the use of these interfaces. See Chapter 3, "Simple Media Server Example," on page 37.

UCS

MIL provides UCS features that enable your application to:

- Save open media interactions before or after their submission.
- Get or search for open media interactions saved in the UCS database.
- Stop processing or delete open media interactions saved in the UCS database.

These features are available through the MILUCSManager interface. When saving a MILInteraction object, you should pay attention to its MILUCSInteractionParameters.

Depending on options set in UCS, if you set some parameters with UCS keys, UCS can add a contact ID to the interaction or UCS can create a new contact. These keys are the following: LastName, FirstName, PhoneNumber, EmailAddress.

For logical reasons, some parameters are set only once. An example is CanBeParent, which specifies whether or not the interaction can be a parent interaction of other interactions saved in UCS. For further details about these interfaces, see the *Media Interaction SDK 7.6 Java API Reference*.

Genesys recommends that when your application modifies open media interactions' data through Interaction Server, your application should save these modifications in the UCS database.

LCA

Local Control Agent (LCA) is a daemon component that monitors, starts, and stops Genesys server applications as well as third-party server applications that you have configured in the Genesys configuration environment.

If the use-lca option defined in the Configuration Layer is true for your application, you can get an MILLCAManager interface able to communicate with LCA.

LCA sends a `MILLCAEvent` propagating your application's runmode in the Genesys environment. The predefined MIL LCA runmodes are enumerated types of the `MILLCARunMode` class.

The LCA feature enables your application to work in different runmodes and update its state within LCA events. Depending on your application's architecture, you can choose to implement one or several of these modes, so that your application can easily integrate into your Genesys environment.

For example, you can implement a MIL server application running in two modes, `PRIMARY` and `BACKUP`, so that you can run two application instances concurrently.

Due to a Management Layer limitation, LCA does not notify applications of type `Third Party Server`— including MIL applications— about runmode changes at runtime. If a primary fail-over occurs, your application receives LCA events for `EXIT` and `PRIMARY/BACKUP` switch-over, but MIL runmode does not change.

For further details about implementing listeners and using the `MILLCAManager`, see the *Media Interaction SDK 7.6 Java API Reference*.

ESP

External Service Protocol (ESP) is a Genesys protocol that MIL can use to communicate with Interaction Server. Its purpose is to let Interaction Server send requests to your application depending on external services defined in your routing strategies.

Refer to your *Multimedia 7.6 and Universal Routing 7.6* documentation for details about implementing external services.

To take advantage of the ESP feature, define an extension package and extension classes that inherit the `MILESPExtension` class, as shown in this code snippet:

```
package com.genesyslab.examples.mil.extensions;
public class ServiceName implements MILESPExtension
{
   public ServiceName(){
   }
   public void initialize(){
      //Implement initialization
      //...
   }
   public void shutdown(){
      //Implement release
      //...
   }
   //Name this method as you wish
```

```
    //MILESPRequest parameter is mandatory
    public void methodName(MILESPRequest request){
        //Managing the request
        //...
    }
}
```

If you define an external service `com.genesyslab.examples.mil.extensions.ServiceName` in your routing strategies, Interaction Server will be able to make calls to `ServiceName.methodName()`, passing data in `MILESPRequest` objects:

- Request parameters—key-value pairs specified in the external service.
- User data—properties (also called attached data) of the interaction involved with the request.

MIL also includes `MILESPResponse` interfaces that let your application provide request results to Interaction Server in a map added to interaction properties.

For further details, see the *Media Interaction SDK 7.6 Java API Reference*.

Bootstrapper

`MILBootstrapper` is a simple MIL server application that connects, then exits when its LCA runmode changes to `MILLCARunMode.EXIT`. The `StartMediaServer.cmd` command file delivered with MIL enables you to launch `MILBoostrapper` and make a connection to the Genesys environment.

You can use this class as a container for a first MIL application. For further details about this interface, see the *Media Interaction SDK 7.6 Java API Reference*.

Stop Interactions

All interfaces and atomic methods provided with MIL for handling interactions allow you to build your own interaction workflow.

Your application can terminate interactions when they reach the end of your workflow by calling the `MILInteractionManager.stopProcessing()` method.

To save in the UCS database those MIL interactions that need to be archived upon termination, synchronize the interaction status in UCS by calling the `MILUCSManager.stopProcessing()` method.

To avoid de-synchronization in case that both or one of the corresponding requests fails, your application design should define a strategy to manage the sequence of calls to the `MILInteractionManager.stopProcessing()` and `MILUCSManager.stopProcessing()` methods. These methods make calls to two distinct servers, so their results do not collid.

For instance, the following scenario updates the interaction status in UCS with the results of the requests sent to the Interaction Server. First, it modifies the interaction status in UCS, then it tries to stop the interaction in the Interaction Server. If the request to the Interaction Server fails, the code snippet again modifies the interaction status in UCS.

```
// Creating a new inbound interaction
MILInteraction milInteraction =
isManager.createInteraction("Inbound",
"InboundNew","thirdPartyMedia");
//...
//change the interaction status in UCS
milUCSManager.stopProcessing(milInteraction.getId(), "Terminated");
//...
try
{
  MILInteractionManager.stopProcessing();
}
catch(RequestFailedException e)
{
  //Test exception
  milUCSParam.setstatus( MILUCSInteractionStatus.UNKNOWN);
  MILUCSManager.saveInteraction(milInteraction);
}
```

What’s Next

The next chapter goes into further details about the `SimpleMediaServer` example provided with this SDK. It explains how to use MIL to implement a simple server that creates open media interactions.

Chapter

3 Simple Media Server Example

This chapter explains the `SimpleMediaServer.java` example, a stand-alone application that connects to Genesys servers and submits new Open Media interactions.

This chapter includes the following sections:

Before you read the "Simple Media Server" section, Genesys recommends that you read the concepts and techniques discussed in the sections "Open Media Commons" on page 28 and "Media Interaction (Java API)" on page 31.

Prerequisites

To follow the discussion in this chapter, you will need the *Media Interaction SDK 7.6 API Reference* (which is located in the `doc/` subdirectory of your Media Interaction SDK installation directory) and the source code for the `SimpleConnector.java`, `startMedia.java`, `OpenMediaSDKData.java`, and `SimpleMediaServer.java` examples. See Chapter 2 for more information about how to use the examples.

More Application Essentials

Now that you have been introduced to the Media Interaction Layer, it is time to outline the steps you will need to make a very basic media server work. There are three basic things this example does:

- **Connect to Servers.** This example uses the `SimpleConnector` class to connect to Genesys servers, as explained earlier and shown in this constructor:

```
SimpleConnector connector = new SimpleConnector();
```

- **Create Independent Threads.** This example uses synchronous methods of the MIL interface. Because it creates several interactions simultaneously, separate threads are in charge of these interactions' management and these threads make use of MIL managers.
- **Submit a new Open Media Interaction.** This example demonstrates how to use MIL managers and main interfaces to create, attach data to, save, and submit a new Open Media interaction.

The examples have been designed to make these steps stand out so that you can quickly learn to write your own real-world applications. The following sections explore how they are implemented in the `SimpleMediaServer` example.

SimpleMediaServer

The `SimpleMediaServer` example provides a stand-alone application that creates and submits new Open Media interactions, each of which contains a file in its attached data.

This simple server periodically parses a source directory. For each file found in this directory, the server creates a dedicated thread that submits a new Open Media interaction to Interaction Server. The file attached to this interaction is removed from the directory.

This section focuses on the steps to complete the submission of a new Open Media interaction.

Here is how `SimpleMediaServer` carries out the three basic steps in writing an MIL application.

Connect to Servers

As explained earlier, the example uses the `SimpleConnector` class to establish the all-important connection with the Genesys servers and to initialize the `OMSDKConnector`. For more information on how this is done, you can refer to

"Open Media Commons" on page 28. For the purposes of this example, the following code snippet shows all you need to do:

```
SimpleConnector connector = new SimpleConnector();
```

Create Independent Threads

In the `run()` method of the main thread, `SimpleMediaServer` parses the directory containing the inputs and process the file list by creating a thread for each file.

```
public void run()
{
   int i= 0;
   while(i<1)
   {
      getSourceFiles();
      processFileList();
      try {
         Thread.sleep(1000);
      } catch (InterruptedException e) {
         e.printStackTrace();
         stop();
      }
   }
}
//...
public synchronized void processFileList()
{
   for(int i=0; i<filesToSend.length; i++)
   {
      //creates a thread for each file to send
      System.out.println(filesToSend[i].getName());
      SubmitThread p = new SubmitThread(filesToSend[i]);
      p.start();
   }
}
```

Submit a New Open Media Interaction

This section shows how the server application manages an interaction submission. This task is handled in the `SubmitThread` class. This separate thread makes calls to the main MIL managers in its `run()` method.

It carries out the following steps:

- Read the file to be attached to a new interaction.
- Create a new Open Media interaction and set its properties.
- Attach the file to the created interaction
- Save the interaction in UCS (Universal Contact Server.)

- Submit the interaction to Interaction Server.

If any of these steps fails, an exception is thrown and the thread stops.

Reading the Input File

In this example, the input files' type has no importance, and the example server will read any file of a size up to `50 KB` as a `byte[]` array.

```
FileInputStream fileInputStream = new FileInputStream(fileToSend);
byte[] b = new byte[51200];
System.out.println(createTimeStamp()+
      "Processing "+fileToSend.getName()+
      " - Read:"+fileInputStream.read(b));
fileInputStream.close();
```

Before creating the interaction, the thread saves the file in the `savePath` directory and removes the file from the `sourcePath` directory,

```
// Saving the file
// If the interaction submission fails,
// the saved file is not deleted
File saveFile =
   File.createTempFile(fileToSend.getName(), ".sav",savePath);
FileOutputStream fileOutputStream = new FileOutputStream(saveFile);
fileOutputStream.write(b);
fileOutputStream.close();

//Deleting the original file, so that it is submitted once only
System.out.println(createTimeStamp()
   +fileToSend.getName()+" Deleting.");
if(! fileToSend.delete())
   System.out.println(createTimeStamp()+" Deletion aborted.");
```

This way, if any step involving MIL interfaces fails, the source file is preserved and the media server does not keep on processing the same file of the source directory.

Create a New Open Media Interaction

For the creation of each interaction, constructor parameters are mandatory. All these parameters must be available through the Configuration Layer, or else the interaction's submission will fail.

```
//Creating the interaction
MILInteraction myInteraction =
     ixnManager.createInteraction(interactionType,interactionSubtype,mediaType);
```

Now it is time to set additional properties that provide further details about this interaction. Here, the thread assigns a date, a queue, and an ID to the interaction. The provided ID must be unique, otherwise Interaction Server will throw an exception and the submission will fail. In this example, the `incrementID()` method is synchronous and prevents threads from creating identical IDs.

```
//Setting Interaction properties
myInteraction.setReceivedAt(new GregorianCalendar().getTime());
String ID = IDRoot+incrementID();
myInteraction.setID(ID);
myInteraction.setQueueName(queueName);
```

Attach the File and Save the New Interaction

To attach the file to the created interaction, the thread adds it in the interaction parameters. There are two types of interaction parameters:

- `MILUCSInteractionParameters`—Parameters for UCS. When saving your interaction in UCS, these parameters are also saved.
- `MILISInteractionParameters`—Parameters for Interaction Server. When submitting the interaction to Interaction Server, these parameters are also submitted.

This thread adds the file to send to both UCS and IS (Interaction Server) interaction parameters, so that the file is saved and sent with the interaction.

UCS and IS interaction parameters are independant. You can set IS interaction parameters after having saved the interaction in UCS, as shown in this code snippet:

```
//Attaching the file to UCS user data
MILUCSInteractionParameters ixnUCSparam = myInteraction.getUCSParameters();
ixnUCSparam.setProperty("FileName", fileToSend.getName());
ixnUCSparam.setProperty("FileBody", b);
ixnUCSparam.setProperty("Subject", "MIL interaction from Simple Media Server");
ixnUCSparam.setStatus(MILUCSInteractionStatus.IN_PROCESS);

//Printing object content
//...

//Saving the interaction
ucsManager.saveInteraction(myInteraction);

MILISInteractionParameters ixnISParam = myInteraction.getISParameters();

//Attaching the file to IS user data
ixnISParam.setProperty("FileName", fileToSend.getName());
ixnISParam.setProperty("FileBody", b);
ixnISParam.setProperty("Subject", "MIL interaction from Simple Media Server");
```

Submit the New Interaction

Submitting the interaction to Interaction Server is the easiest step. It requires a single call to the `MILInteractionManager.submit()` method. This method is synchronous, and does not return until the interaction submission succeeds or fails.

```
//Submitting the interaction
ixnManager.submit(myInteraction);
```

Note: The `MILInteractionManager` also provides asynchronous methods, including methods for interaction submission. See the *Media Interaction SDK 7.6 Java API Reference* for further details.

Wrap Up

If you can master the preceding steps, you will have the foundation for writing your own MIL servers. However, there is also some code in the `SimpleMediaServer` constructor that you might be curious about. In order to make it easier to understand this example, here is a brief explanation of how the `SimpleMediaServer` constructor performs the setup tasks.

The constructor parameters define data for interaction creation. The following parameters should all be defined in the Configuration Layer: queue name, media type, interaction type, and interaction subtype. Refer to the *Interaction SDK 7.6 Java Deployment Guide* for further details.

After calling its superconstructor (handling `SimpleConnector`), `SimpleMediaServer` initializes the MIL factory, and get references on main managers required to handle interaction creation, as shown here:

```
try {
   MILFactory.initialize(new MILInitializationParameters(null));

   //Getting Managers
   ixnManager = MILFactory.getMILFactory().getInteractionManager();
   ucsManager = MILFactory.getMILFactory().getUCSManager();
catch (MILInitializationException e) {
   e.printStackTrace();
   stop();
}
```

Runtime

Now that you understand the basics of the `SimpleMediaServer` application, you can start running it in your environment.

To do so, compile and run the `StartMedia.java` example with appropriate arguments (which are described in the examples' Javadoc.)

Once `SimpleMediaServer` is started, each time you copy files from the source directory, the server removes them, after having sent them to Interaction Server in an Open Media interaction.

Chapter

4 Simple Custom Extension Example

This chapter explains how the `SimpleMediaServer` example makes use of the `SimpleCustomExtension` class to manage ESP (External Service Protocol) requests. The simple media server demonstrated in this example connects to Genesys servers and submits new Open Media interactions. When these interactions go through a strategy that includes an external service, Universal Routing Server (URS) sends an ESP request, referring to the simple custom extension, through Interaction Server.

This chapter includes the following sections:

Before you read the "Simple Custom Extension" section, Genesys recommends that you read the concepts and techniques discussed in the sections "Open Media Commons" on page 28, "Media Interaction (Java API)" on page 31, and "Simple Media Server Example" on page 37.

Prerequisites

To follow the discussion in this chapter, you will need:

- The *Media Interaction SDK 7.6 Java API Reference* (which is located in the `doc/` subdirectory of your Media Interaction SDK installation directory)
- The source code for the `SimpleConnector.java`, `startCustomMedia.java`, `OpenMediaSDKData.java`, `SimpleMediaServer.java`, and `SimpleCustomExtension.java` examples.

See Chapter 2 for more information about how to use the examples.

Before you start with this example, you must define an ESP strategy in the Interaction Routing Designer and create an External Service. In the `External Service Property` text box, specify:

- The fully qualified name of the custom extension for the service name: `media.sdk.java.examples.SimpleCustomExtension`
- The method to call during script execution: `doProcessRequest`

Save the strategy and open Configuration Manager. In the `Properties` dialog box of your URS application there, add the application defined for your simple media server to the `Connections` tab. Click `OK`.

You are now ready to launch the `StartCustomMedia` script. For further details, refer to the `Readme.html` file delivered with the samples.

More Application Essentials

Now that you have been introduced to the Media Interaction Layer and to the simple media server, it is time to outline the steps you will need to make a very basic custom extension work with the media server. There are four basic things this example shows:

- **Define a custom extension.** This example shows which methods to implement when inheriting the `MILESPExtension` interface.
- **Preload the custom extension.** This example explains why and how to preload an extension.
- **Process an ESP request from Interaction Server.** This example demonstrates the use of the `MILESPRequest` object received for each request to process.
- **Send an ESP response to Interaction Server.** This example uses synchronous methods of the MIL interface to send fault or success responses.

The examples have been designed to make these steps stand out so that you can quickly learn to write your own real-world applications. The following sections explore how they are implemented in the `SimpleCustomExtension` sample.

SimpleCustomExtension

The `SimpleMediaServer` example provides a stand-alone application that creates and submits new Open Media interactions, each of which contains a file in its attached data.

When one of these interactions goes through the routing strategy defined for the queue to which it is submitted, URS uses the external service to send an

ESP request to the MIL application you defined. Within this request, it passes the service name and the method to be called by the MIL server.

The simple custom extension parses the text file attached to the submitted interaction. If the file contains a string such as `Gold`, `GOLD`, or `gold`, the extension sends an ESP response including a `Gold Marker` marker set to the `GoldCustomer` value. (In the example's scenario, this indicates a "gold" or first-tier customer.)

URS waits for the ESP response and goes on processing the interaction in the routing strategy.

This section focuses on the steps to process an ESP request and send an ESP response. Here is how `SimpleCustomExtension` carries out the four basic steps in writing an MIL extension.

Define a Custom Extension

As explained earlier, to define a custom extension, you must create a class that inherits the `MILESPExtension` interface and implement at least three methods: `initialize()`, `shutdown()`, and a third class to be called by the external service, as shown here:

```
public class SimpleCustomExtension implements MILESPExtension {
   /** Mandatory */
   public void initialize(){
      System.out.println(this.createTimeStamp() + " Custom Extension initialized " );
   }
   /** Mandatory */
   public void shutdown(){
      System.out.println(this.createTimeStamp() + " Custom Extension is shutdown " );
   }

   /** Mandatory - Method to be called when MIL gets a request for this extension.
    * @param request The ESP request to be processed.
    */
   public void doProcessRequest(MILESPRequest request){
      //...
   }
//...
}
```

When developping your extensions, take into account that MIL uses the extension as a singleton. A unique instance of each extension class is initialized at runtime by calling its `initialize()` method. If the extension is preloaded (see "Preload the Custom Extension" on page 48), the extension is initialized at the MIL library's initialization. Otherwise, the MIL library creates the extension instance at the first ESP request.

Note: The same instance of a custom extension may serve simultaneous requests in separate threads. If your extension defines attributes, your application must take care of data synchronization.

If the extension is preloaded at MIL initialization, then at the time MIL calls the `initialize()` method, the library is not fully initialized. Therefore, Genesys recommends that extensions' `initialize()` methods do not make calls to MIL interfaces and methods.

Whether or not extensions are preloaded, the `shutdown()` method is called when the library is released. Therefore, do not use MIL interfaces in this method.

Preload the Custom Extension

If you preload an extension, the corresponding object is already instantiated when the MIL library receives its first ESP request. If you do not preload the extension, MIL creates it upon the first ESP request involving this extension.

Note: Preloading extensions is a matter of design of your application.

In the `SimpleMediaServer` example, the `SimpleCustomExtension` class is preloaded depending on a boolean value passed at application startup. This `custom` boolean is set to `true` if you launch the `StartCustomMedia` script. Preloading the extension is done in the `SimpleMediaServer` constructor by passing the fully qualified name of the extension to the `MILInitializationParameters` object used for initializing the MIL factory, as shown in the following code snippet.

```
MILInitializationParameters milParam;
/// If you are running StartCustomMedia, custom is true
if( custom == true)
   milParam = new MILInitializationParameters(
                new String[]{"media.sdk.java.examples.SimpleCustomExtension"});
else
   milParam = new MILInitializationParameters(null);

MILFactory.initialize(milParam);
```

Process an ESP Request

When the media server gets an ESP request, it receives the extension's fully qualified name and the name of the method to call. In this example, the method to be called is `SimpleCustomExtension.doProcessRequest()`.

This method has a `MILESPRequest` instance in its parameters. This object contains parameters set in the external service you defined earlier, and all the user data of the relevant interaction—that is, the properties and data submitted with the interaction to Interaction Server.

In this example, the `doProcessRequest()` method creates and starts a thread of class `ProcessThread` to handle the processing of the request, as shown here:

```
public void doProcessRequest(MILESPRequest request){
   ProcessThread p = new ProcessThread(request);
   p.start();
}
```

The `run()` method of this thread is in charge of processing the request and sending a response. This method first makes calls to `MILESPRequest` methods to display information related to the received request.

```
System.out.println(this.createTimeStamp()+ " ESP request "+ request.getID()
             + " interaction ID " + request.getInteractionID()
             + " method " + request.getMethodName()
             + " service (extension) " + request.getServiceName() );
```

Then, it retrieves the file attached to the interaction passed in the user data of the ESP request. The example parses this file to find a `gold` string, as shown in the following code snippet.

```
String fileName = (String) request.getUserData().get("FileName");
System.out.println(this.createTimeStamp() + " Processing Gold search on "+ fileName);

byte[] b = (byte[]) request.getUserData().get("FileBody");
ByteArrayInputStream mystream = new ByteArrayInputStream(b);
DataInputStream d = new DataInputStream(mystream);

boolean gold = false;
while(gold == false){
   try {
      String line = d.readLine();
      if(line!=null &&(line.indexOf("gold")!= -1
         || line.indexOf("Gold")!= -1
         || line.indexOf("GOLD")!= -1))
      {
         gold = true;
      }
   }catch (IOException e) {
      break;
   }
}
```

Send an ESP Response

To create an ESP response, your application calls a `MILESPRequest.create<Type>Response()` method, assigns appropriate values to the response, then sends it by calling the `MILESPResponse.send()` method, as detailed in the following subsections.

Fault Response

Extensions should send a fault response in cases where they did not manage to process the request. In the `SimpleCustomExtension`, this happens if the `ProcessThread` thread did not manage to access user data and received an unexpected exception.

Before the `run()` method sends the fault response, it must set an error code and an error message, as shown here.

```
MILESPFaultResponse faultResponse = request.createFaultResponse();
faultResponse.setFaultCode(0);
faultResponse.setFaultString("No file found for this interaction");

System.out.println(this.createTimeStamp()
      + " Send fault response for ESP Request " +request.getID());
try {
   faultResponse.send();
} catch (MILRequestFailedException e) {
      e.printStackTrace();
}
```

Success Response

In the example, `ProcessThread` sends a success response if it managed to parse the text attached to the interaction. It creates a `MILESPSuccessResponse` instance and adds the result of the search to the user data of this object, as shown below.

```
String goldString = "NotGold";
if(gold)
   goldString = "GoldCustomer";

MILESPSuccessResponse successResponse = request.createSuccessResponse();
successResponse.setUserDataItem("GoldMarker", goldString);

System.out.println(this.createTimeStamp()
   + " Send sucess response gold("+ gold +") for ESP Request " +request.getID());
try {
   successResponse.send();
} catch (MILRequestFailedException e) {
```

```
   e.printStackTrace();
}
```

The created `MILESPResponse` object originally contains an empty map of user data: it does not include the user data and the parameters of the ESP request.

If your application makes calls to the `MILESPResponse.setUserData()` or `MILESPResponse.setUserDataItem()` methods, then as shown in the above code snippet, this user data is sent with the response. URS adds them to the interaction's properties as the value of the `ItemAttachedData` key in the array of attached data.

Runtime

Now that you understand the basics of the `SimpleCustomExtension` application, you can start running it in your environment.

To do so, compile and run the `StartCustomMedia.java` example with appropriate arguments (which are described in the examples' Javadoc.)

Once `SimpleMediaServer` is started, each time you copy files from the source directory, the server removes them, after having sent them to Interaction Server in an Open Media interaction.

When this interaction occurs in URS, Interaction Server sends an ESP request to the media server, and you can see the extension activity in log traces. As a result, if you start an agent application that is built on the Agent Interaction SDK, and if the extension is solicited before the interaction occurs on the agent's place, you should able to see the `GoldMarker` value in the interaction's attached data.

For instance, see the Simple Open Media Interaction code example provided with the Agent Interaction SDK (Java).

Index

T

U

V