Platform SDK 7.6

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

Genesys, the Genesys logo, and T-Server are registered trademarks of Genesys Telecommunications Laboratories, Inc. All other trademarks and trade names referred to in this document are the property of other companies. The Crystal monospace font is used by permission of Software Renovation Corporation, www.SoftwareRenovation.com.

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)-118-974-7002	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_platform_02-2008_v7.6.001.00

Table of Contents

Preface

Welcome to the *Platform SDK 7.6 Developer's Guide*. This document gives you the information you need to write programs that communicate directly with several of the servers in the Genesys Framework.

This document is valid only for the 7.6 release 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 CD, which you can order by e-mail from Genesys Order Management at orderman@genesyslab.com.

For the latest versions of all SDK documents, please visit the Genesys Developer Zone at http://www.genesyslab.com/developer.

This preface contains these sections:

The Platform SDKs are for developers requiring maximum flexibility. They provide this flexibility by giving you low-level access to the core functionality of the following Genesys servers:

- T-Servers
- Configuration Server
- Statistics Server
- Interaction Server
- Outbound Contact Server
- Universal Contact Server

- Message Server
- Solution Control Server
- Local Control Agents
- Web API Server/Chat Server
- Web API Server/E-Mail Server Java
- Web API Server/Callback Server

By creating objects that communicate with these servers using their own protocols, you will be able to pass messages back and forth with any or all of them. Please note, however, that the added control you get from using the Platform SDKs may bring with it a longer development effort than you might need for software developed with SDKs that offer a higher level of abstraction, such as the Interaction SDK.

Intended Audience

This document, primarily intended for software developers, assumes that you have a basic understanding of:

- Computer-telephony integration (CTI) concepts, processes, terminology, and applications.
- Network design and operation.
- Your own network configurations.

You should also be familiar with:

- Software development using the C# or Java programming languages
- Genesys Framework architecture and functions

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 specialized client applications 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 preface, this document contains the following chapters:

- Chapter 1, "About the Platform SDK," on page 13. This chapter briefly introduces the Platform SDKs and shows how to access the in-depth introductory material that is available with the product.
- Chapter 2, "About the .NET Code Examples," on page 21. This chapter discusses how the .NET code examples that come with the Platform SDK are packaged, how to use them, and how they work.
- Chapter 3, "About the Java Code Examples," on page 53. This chapter discusses how the Java code examples that come with the Platform SDK are packaged, how to use them, and how they work.

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:

75sdk_dev_platform_03-2007_v7.5.000.01

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:

- The Genesys Developer Zone, at http://www.genesyslab.com/developer, which contains the latest versions of all SDK documents, as well as forums and other important sources of developer-related information.
- *Platform SDK 7.6 Deployment Guide,* which contains important configuration and installation information.
- *Platform SDK 7.6 API Reference* for the particular SDK you are using, which provides the authoritative information on methods, functions, and events for your SDK.
- *Platform SDK 7.6 Application Block Guide* for the particular application block you are using. Each *Guide* explains how to use the application block and documents all code used in the application block itself. (Application blocks are production-quality available code.)
- *Platform SDK 7.6 Code Examples* for the particular SDK you are using, which offer illustrative ways to begin using your SDK. These code examples are fully functioning software applications, but are for educational purposes only and are not supported.

- The *Deployment Guides* for the underlying Genesys servers with which you intend to have your Platform SDK applications integrate. For instance, be sure to check the *Framework 7.5 SIP Server Deployment Guide* if you plan on using the Voice Platform SDK and the SIP Endpoint Application Block.
- Other Genesys SDK documentation for extended information on ways to integrate custom applications with Genesys Servers. This includes documents such as the *T-Library SDK 7.2 C Developer's Guide,* which contains detailed information on the `TLIB` protocol and on message exchanges with T-Servers. All Genesys SDK documentation is available from the Technical Support website and is also located on the SDK Documentation CD and on your product CD.
- The *Genesys Technical Publications Glossary,* which ships on the Genesys Documentation Library CD 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 Documentation Library CD, 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 Platform SDK

The Platform SDK includes individual components that allow you to write .NET and Java applications to communicate with Genesys servers in their native protocols. For example, to work with Stat Server, you would write an application that uses the Statistics Platform SDK.

The Platform SDK also includes several *application blocks*. These are production-quality components that you can use to handle commonly-encountered tasks, such as event handling or communicating with more than one server from a single application.

Each of the server-related components comes with an API Reference and each application block comes with an Application Block Guide. These documents include detailed introductory material on the use of the Platform SDK, in addition to material that relates to the individual component.

This chapter gives you a brief outline of the purpose of the individual SDKs and the application blocks that come with them. It also shows where to find the introductory material about the Platform SDK. It contains the following sections:

The Platform SDKs

This section gives you a simple introduction to each of the Platform SDKs. It ends with Table 1 on page 15, which shows the servers each of the Platform SDKs connects with, and gives the names of the protocols that are used to communicate with each server.

Configuration

The Configuration Platform SDK enables you to build applications that add, modify, and delete information in the Configuration Layer of your Genesys environment.

Contacts

The Contacts Platform SDK enables you to write .NET applications that work with the Universal Contact Server.

Management

The Management Platform SDK enables you to write applications that interact with Message Server, Solution Control Server, and Local Control Agents.

Open Media

With the Open Media Platform SDK, you can build applications that feed open media interactions into your Genesys environment, or applications that act as custom media servers to perform external service processing (ESP) on interactions that have already entered it.

Outbound Contact

The Outbound Contact Platform SDK can be used to build applications that allow you to manage outbound campaigns.

Statistics

Use the Statistics Platform SDK to build applications that solicit and monitor statistics from your Genesys environment.

Voice

The Voice Platform SDK enables you to design applications that monitor and handle voice interactions from a traditional or IP-based telephony device.

Web Media

The Web Media Platform SDK can be used to build applications that interact with Chat Server, E-Mail Server Java, and Callback Server by way of the Web API Server.

Table 1: The Platform SDKs

SDK Name	Server	Protocol
Configuration Platform SDK	Configuration Server	CFGLIB
Contacts Platform SDK (.NET only in 7.6)	Universal Contact Server	UCS Protocol
Management Platform SDK	Message Server Solution Control Server Local Control Agents	GMESSAGELIB SCSLIB LCALIB
Open Media Platform SDK	Interaction Server	ITX, ESP
Outbound Contact Platform SDK	Outbound Contact Server	CMLIB OCS-Desktop Protocol
Statistics Platform SDK	Stat Server	STATLIB
Voice Platform SDK	T-Servers	TLIB Preview Interaction Protocol
Web Media Platform SDK	Web API Server/Chat Server Web API Server/E-Mail Server Java Web API Server/Callback Server	MCR Chat Lib MCR E-Mail Lib MCR Callback Lib

The Application Blocks

When you are working with a message-based API, you need to handle events. When you are using an application that needs to communicate with more than one server, you have to manage the connections to each server and keep track of the interactions with each one.

So why should every development team have to write new code to address functionality that other developers have already had to deal with?

Genesys has written reusable production-quality components that carry out these functions and other common development tasks facing Platform SDK developers. We call these components *application blocks.* They have been designed using industry best practices so you can use them without modification. We have also included the source code so you can tailor them if you need to.

This section gives a list of the application blocks that currently ship with the Platform SDK.

Note: If you have suggestions about the application blocks, please contact us in the forums at the Genesys Developer Zone, which can be reached from the DevZone home page at http://www.genesyslab.com/developer.

Message Broker

The Message Broker Application Block makes it easy for your applications to handle events in an efficient way.

Protocol Manager

The Protocol Manager Application Block allows for simplified communication with more than one server. It takes care of opening and closing connections to many different servers, as well as handling the reconfiguration of high availability connections.

Warm Standby

You can use the Warm Standby Application Block to switch to a backup server in case your primary server fails, in cases where you do not need to guarantee the integrity of existing interactions.

Configuration Context

The Configuration Context Application Block makes it easy for your client applications to retrieve information from the Genesys configuration layer.

Configuration Object Model

The Configuration Object Model Application Block provides a consistent and intuitive object model for .NET applications that need to work with Configuration Server objects. (The Java version of this application block can be downloaded from the Genesys Developer Zone at http://www.genesyslab.com/developer.)

SIP Endpoint

The SIP Endpoint application block enables you to add Internet telephony capabilities to your .NET softphone applications, using the Session Initiation Protocol (SIP).

Multi-Channel Communication Model

You can use the Multi-Channel Communication Model Application Block in your .NET applications to work with different types of communication channel in a unified way.

Learning About the Platform SDK for .NET

As mentioned earlier, each server-related component of the Platform SDK comes with an API Reference. Each application block also comes with its own Application Block Guide. In addition to information that is specific to each of these components, these documents also contained detailed introductory material for the Platform SDK. Here is how to find that material if you are using .NET.

First, locate the appropriate document by opening the `Start Menu`. Select `All Programs > Genesys Solutions > Platform SDK for .NET 7.6`. If you need an API Reference, select the appropriate link from the list that is displayed on your screen. If you are looking for an Application Block Guide, select `Application Blocks` and then select the appropriate link.

When you have opened the document, you will see a page with a title like this: “Welcome to the Configuration Platform SDK” or “Welcome to the Configuration Context Application Block.”

This page includes links to the following articles, which will get you started with the Platform SDK:

- Introducing the Platform SDKs
- Architecture of the Platform SDKs
- Connecting to a Server
- Event Handling
- Namespace Structure

Once you have covered this material, you can start learning about the specifics of the component you are working with by following the next link, which has a title like this: “Using the Configuration Platform SDK” or “Using the Configuration Context Application Block.”

Learning About the Platform SDK for Java

As mentioned earlier, each server-related component of the Platform SDK comes with an API Reference. Each application block also comes with its own Application Block Guide. In addition to information that is specific to each of these components, these documents also contain detailed introductory material for the Platform SDK. Here is how to find that material if you are using Java.

If you are using Windows, you can locate the appropriate document by opening the `Start Menu`. Select `All Programs > Genesys Solutions > Platform SDK for Java 7.6`. If you need an API Reference, select the appropriate link from the list that is displayed on your screen. If you are looking for an Application Block Guide, select `Application Blocks` and then select the appropriate link.

When you have opened the document, you will see a page with a title like this: "Welcome to the Configuration Platform SDK" or "Welcome to the Configuration Context Java Application Block 7.6 Guide." Click the `Description` link that appears on that page, as shown in Figure 1.

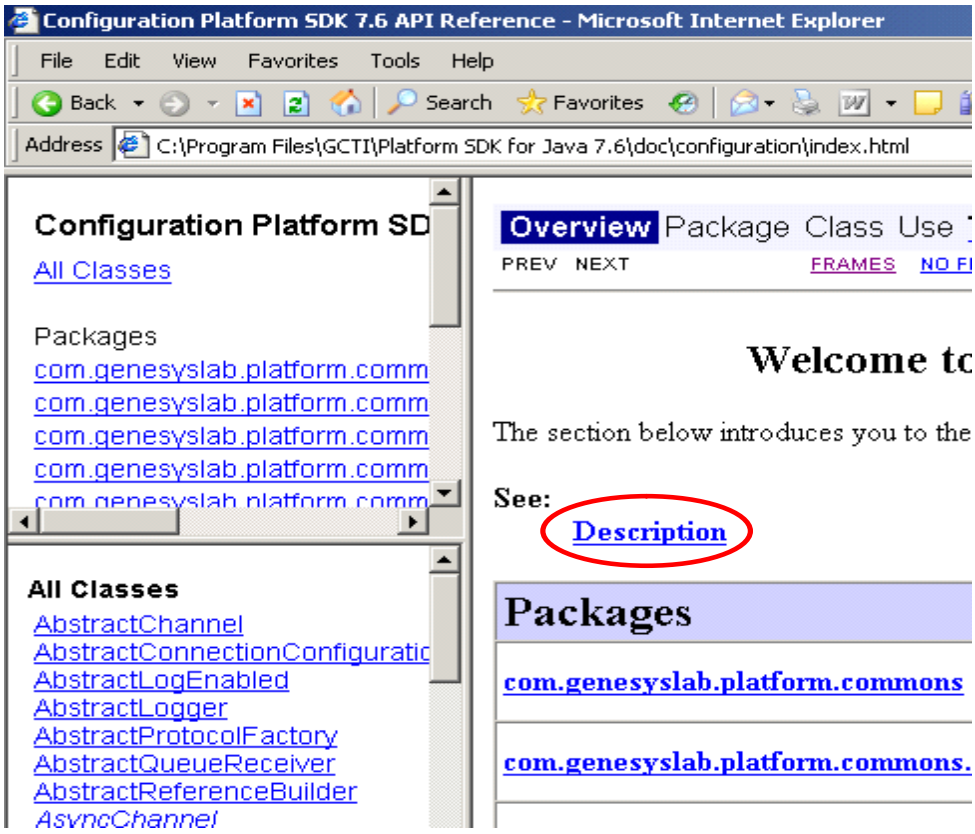

Figure 1: The Description Link on the Welcome Page

When you click this link, the page scrolls to a section that includes links to the following articles, which will get you started with the Platform SDK:

- Introducing the Platform SDKs
- Architecture of the Platform SDKs
- Connecting to a Server
- Event Handling
- Package Structure

Once you have covered this material, you can start learning about the specifics of the component you are working with by following the next link, which has a title like this: “Using the Configuration Platform SDK” or “Using the Configuration Context Application Block.”

New in this Release

This section summarizes the changes between the 7.6 release of the Platform SDK and prior releases of this product.

Contacts Platform SDK for .NET

The Platform SDK for .NET now includes the Contacts Platform SDK for .NET, which allows for integration with the Universal Contact Server. For more information, see the *Contacts Platform SDK API Reference.*

Configuration Object Model Application Block for Java

The Configuration Object Model Application Block is now available for Java. To obtain this application block, download it from the Genesys Developer Zone at `http://www.genesyslab.com/developer`.

Host Information in Management Platform SDK

A new request has been added to the Management Platform SDK. `RequestGetHostInfo` enables applications to request host information from Solution Control Server.

New Java Code Examples

The Platform SDK now includes several Java code examples.

These examples use the Protocol Manager and Message Broker Application Blocks, which employ industry best practices to facilitate server connections and event handling.

See Chapter 3, "About the Java Code Examples," on page 53 of this Developer's Guide for more information.

New .NET Code Examples

There is now a Complex .NET example that shows how to work with two servers in a single application. Also, the Configuration and Statistics examples have been rewritten to use the Protocol Manager and Message Broker Application Blocks, which employ industry best practices to facilitate server connections and event handling.

See Chapter 2, "About the .NET Code Examples," on page 21 of this Developer's Guide for more information.

New Default Value for Users Accounts

Starting with release 7.6, Configuration Server does not automatically assign new users to any Access Groups. Each new user:

- Is created with no privileges.
- Cannot log in to any interface.
- Cannot use a daemon application.

Each new user must be added to an Access Group by an Administrator or a user with permissions to update user accounts. For details, see the chapter "No Default Access for New Users" in the *Genesys 7.6 Security Deployment Guide*.

Enhanced documentation

The documentation for all API References, Application Blocks, and samples has been enhanced. See the Genesys Developer Zone at http://www.genesyslab.com/developer for a complete listing.

Chapter

2 About the .NET Code Examples

This chapter introduces the .NET code examples that accompany this developer's guide. It contains the following sections:

Setup for Development

When you install the Platform SDK, be sure that you have the required tools, environment-variable values, and configuration data. See the *Platform SDK 7.6 Deployment Guide* for details.

The Platform SDK includes all Genesys libraries and third-party libraries needed for proper operation, while the Genesys Developer Documentation CD includes the Platform SDK code examples and a PDF version of this developer's guide.

Source-Code Examples

This chapter refers to the supplied source-code examples. These examples illustrate the use of some common features of the Platform SDKs. They are provided on the Genesys Documentation CD in a `.zip` archive file.

The examples are not tested and are not supported by Genesys.

Note: Applications that use the Platform SDK normally receive events from the servers they work with. Genesys recommends that you use the Message Broker Application Block to handle these messages. However, there may be times when this is not possible. In that case, you will probably need to set up separate threads for message-handling. "Voice Examples" on page 46 shows one way to do this.

Required Third-Party Tools

To develop .NET applications that use the Platform SDK, you will need .NET Framework 2.0. If you are using Visual Studio, you will need Visual Studio 2005 or higher.

In writing this guide, Visual Studio 2005 was used to compile and run all of the code examples.

.NET Environment Setup

In order to compile and run these examples, Visual Studio needs access to the Platform SDK libraries. These libraries are located in the `bin/` subdirectory of the Platform SDK product installation directory.

Add references to all of these libraries to your Visual Studio projects.

If you want to use any of the application blocks that ship with the Platform SDK, you also need to add their libraries to your projects. To create these libraries, build each application block according to the instructions in the appropriate Application Block Guide.

Building the Examples

To build one of the C# examples described in this document:

- Open the example in Visual Studio
- Add references to the Platform SDK libraries and any necessary application block libraries, as mentioned in the section on environment setup
- Build the example

Configuration Data

For the examples provided for this document to work, they need valid configuration data, including connections to servers and configuration objects such as `Place`, `Dn`, `Agent`, and so on.

See the *Platform SDK 7.6 Deployment Guide* for configuration details.

Configuration Example

This example allows you to retrieve data from Configuration Server about objects of a specific type.

Note: Before you try this example, you should review the introductory material in the *Configuration Platform SDK 7.6 API Reference*. The article on "Connecting to a Server" will help you understand how to use the Protocol Manager Application Block to connect to Configuration Server, while the article on "Event Handling" will show how to use the Message Broker Application Block to handle messages coming in from Configuration Server. This example uses both of these application blocks.

The user interface for this example consists of a single button and a text window, as shown in Figure 2.

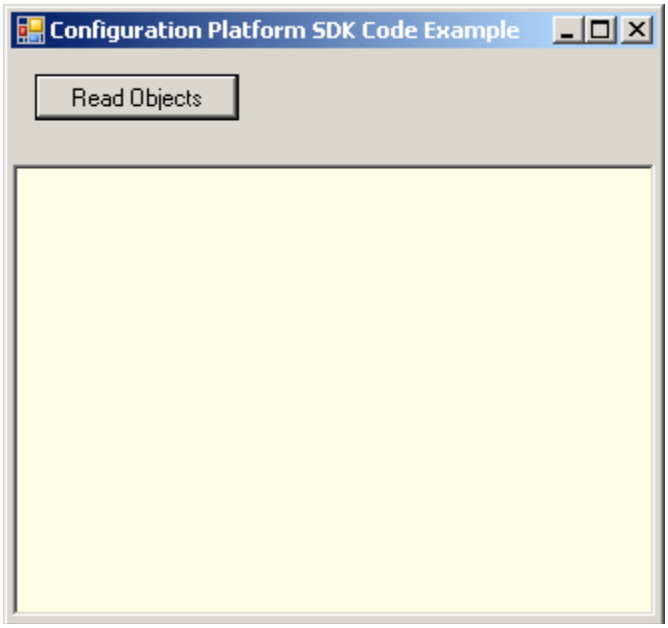

Figure 2: Configuration Example User Interface

To use this application, you need to set up some information about your environment. Find the following statements in the source code for `Form1.cs`, and enter values that match your Genesys Configuration Layer:

```
// Enter configuration information here:
private string confServerHost = "<host>";
private int confServerPort = <port>;
private string clientName = "<clientName>";
private string userName = "<userName>";
private string password = "<password>";
// End of configuration information.
```

Then you need to determine which object type you want to retrieve information for. The example is set to ask for information about `Filter` objects by default.

Once you have entered your configuration information, run the sample and click the `Read Objects` button. If you have a `Filter` object in your configuration layer, you should receive a response that looks something like this:

```
This is the response:

<?xml version="1.0" encoding="UTF-8"?>
<ConfData>
  <CfgFilter>
    <DBID value="101" />
    <tenantDBID value="101" />
    <name value="Test_Filter" />
    <formatDBID value="104" />
    <state value="1" />
  </CfgFilter>
</ConfData>
```

If you don't have any filters set up, you can try a different object type, for example, `Person`. Look in the `ReadObjectsButton_Click` method to see the code that generates the query. All you need to do is change this line of code:

```
(int)ConfServerObjectType.Filter,
```

To this:

```
(int)ConfServerObjectType.Person,
```

If you do try querying on `Person` objects, you may get more information than you wanted. In that case, you might want to filter out most of the `Person` objects. To do that, let's take a look at how this query was generated.

The first thing to notice is that you obtain Configuration Layer data by issuing a `RequestReadObjects`. This is done using a `Create` method. In this case, we are using the `ProtocolManagementService.Send` method to get the request to the server, as explained in the introductory article on event handling in the *Configuration Platform SDK API Reference*:

```
KeyValueCollection filterKey = new KeyValueCollection();
RequestReadObjects requestReadObjects =
RequestReadObjects.Create(
   (int)ConfServerObjectType.Filter,
   filterKey);
protocolManagementService["Config_Server_App"].
   Send(requestReadObjects);
```

But notice that before you can create the `Request` object, you have to set up a `KeyValueCollection` as a filter for your query. This collection can be empty, as shown above, but you must create one.

Now try changing the object type to `Person`. Run the sample and examine the response from Configuration Server. You probably got back a lot of data!

To filter out most of this data, you can add one or more key-value pairs to the filter key, as shown in the bolded line:

```
KeyValueCollection filterKey = new KeyValueCollection();
filterKey.Add("user_name", "<user_name>");
RequestReadObjects requestReadObjects =
   RequestReadObjects.Create(
      (int)ConfServerObjectType.Person,
      filterKey);
confServerProtocol.Send(requestReadObjects);
```

This request will only return data about one `Person`.

Connecting to Configuration Server

As mentioned previously, this code example uses the Protocol Manager Application Block to connect to Configuration Server. This section will briefly review the things you need to do to use Protocol Manager in an application of this type.

First off, you must have the proper `using` statement, and then declare a new `ProtocolManagementService` object:

```
using Genesyslab.Platform.ApplicationBlocks.Commons.Protocols;
.
.
.
ProtocolManagementService protocolManagementService;
```

Now you need to instantiate the Service object; set up a `ConfServerConfiguration` object; and configure the Configuration object with the appropriate settings, at which point you can register the Configuration object with your Protocol Management Service:

```
protocolManagementService = new ProtocolManagementService();

ConfServerConfiguration confServerConfiguration =
   new ConfServerConfiguration("Config_Server_App");
confServerConfiguration.Uri = confServerUri;
confServerConfiguration.ClientType = ConfServerClientType.SCE;
confServerConfiguration.ClientName = clientName;
```

```
confServerConfiguration.UserName = userName;
confServerConfiguration.UserPassword = password;
protocolManagementService.Register(confServerConfiguration);
```

You still need to open the connection to the server, but before doing that, let's take a look at the event-handling code you need to set up first.

Event Handling

As shown in the article on event handling in the *Configuration Platform SDK API Reference,* you only need to do a few things to set up Message Broker so that it will work with Protocol Manager.

Just like with Protocol Manager, you will start by adding a `using` statement and declaring the `EventBrokerService` object:

```
using Genesyslab.Platform.ApplicationBlocks.Commons.Broker;
.
.
.
EventBrokerService eventBrokerService;
```

At that point, you can instantiate the service and register the appropriate Configuration Server events with the event handlers you have set up in your code. (We will discuss the event handlers in just a few moments.)

```
eventBrokerService =
   BrokerServiceFactory.CreateEventBroker
   (protocolManagementService.Receiver);

eventBrokerService.Register(this.OnEventObjectsRead,
   new MessageIdFilter(EventObjectsRead.MessageId));
eventBrokerService.Register(this.OnEventObjectsSent,
   new MessageIdFilter(EventObjectsSent.MessageId));
eventBrokerService.Register(this.OnEventError,
   new MessageIdFilter(EventError.MessageId));
```

Now you are ready to open the connection to the server:

```
protocolManagementService["Config_Server_App"].Open();
```

In this example, we are using three event handlers. The first method handles `EventError` messages:

```
private void OnEventError(IMessage theMessage)
{
   EventError eventError = theMessage as EventError;
   if (eventError == null)
```

```
   {
      writeToLogArea("EventError!\n");
      return;
   }

   writeToLogArea("EventError:\n"
      + eventError + "\n\n");
}
```

There is also an event handler for `EventObjectsSent`. You will receive this event when all of the data you requested has been sent from Configuration Server. If you are receiving large amounts of data, this is the event that will tell you it has all arrived. Here is the `EventObjectsSent` handler from the example:

```
private void OnEventObjectsSent(IMessage theMessage)
{
   EventObjectsSent objectsSent = theMessage as EventObjectsSent;
   if (objectsSent == null)
   {
      writeToLogArea("EventObjectsSent!\n");
      return;
   }

   writeToLogArea("EventObjectsSent:\n"
      + objectsSent + "\n\n");
}
```

For the simple purposes of the sample, the `EventObjectsRead` handler acts as if all of the data from the server has arrived in one response. The incoming message is cast to `EventObjectsRead` and the result is written out to an XML document:

```
private void OnEventObjectsRead(IMessage theMessage)
{
   EventObjectsRead objectsRead = theMessage as EventObjectsRead;
   if (objectsRead == null)
   {
      writeToLogArea
      ("There are no objects of the requested type\n
      in the Genesys Configuration Layer.\n\n");
      return;
   }

   XmlDocument resultDocument = new XmlDocument();
   resultDocument =
   (XmlDocument)objectsRead.ConfObject;
```

```
   StringBuilder xmlAsText = new StringBuilder();
   StringWriter stringWriter = new StringWriter(xmlAsText);
   XmlTextWriter xmlTextWriter =
   new XmlTextWriter(stringWriter);
   xmlTextWriter.Formatting = Formatting.Indented;

   resultDocument.WriteContentTo(xmlTextWriter);

   writeToLogArea("This is the response:\n\n"
      + xmlAsText.ToString() + "\n\n");
}
```

After the XML information is received, the example application receives an `EventObjectsSent` message from Configuration Server, indicating that all of the requested data has been sent.

Statistics Example

This example shows how to subscribe to a statistic and have it updated periodically. The user interface of this example is very simple, with a single button that lets you retrieve statistics information based on values that are hard-coded in the application. Figure 3 shows the user interface.

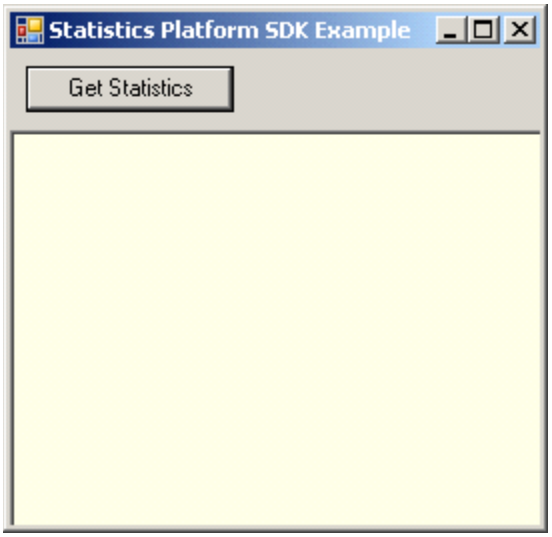

Figure 3: Statistics Example User Interface

After clicking the `Get Statistics` button, you will receive an `EventPackageOpened`. The application will display information similar to that shown here:

```
EventPackageOpened:
'EventPackageOpened' ('2048')
message attributes:
```

```
XAL_PCKG_REQ_ID [int] = 2
XAL_PCKG_ID [int] = 132
XAL_PCKG_USER_REQ_ID [int] = -1
```

You will also receive information similar to the following. This information will be repeated every 15 seconds:

```
Statistic Metric is: StatisticType=TotalNumberInboundCalls
I=
TimeProfile=CollectorDefault
TimeRange=Range0-120
Filter=VoiceCall

Statistic Object is: Tenant=MyTenant
ObjectId=1234@theSwitch
ObjectType=RegularDN

Statistic IntValue is: 0
Statistic StringValue is: 0
Statistic ObjectValue is:
Statistic ExtendedValue is:
Statistic Tenant is: MyTenant
Statistic Type is: RegularDN
Statistic Id is: 1234@theSwitch
Statistic TimeProfile is: CollectorDefault
Statistic StatisticType is: TotalNumberInboundCalls
Statistic TimeRange is: Range0-120
```

Now that you have seen what the sample does, let's look at how it works.

First of all, in order to use this example, you will have to enter the appropriate configuration information in the following lines of `Form1.cs`:

```
// Enter configuration information here:
private string statServerHost = "<host>";
private int statServerport = <port>;
private string tenantName = "<tenantName>";
private string dn = "<DN>@<switch>";
// End of configuration information.
```

The structure of this example is similar to the structure of the previous one. You start out by declaring and instantiating `ProtocolManagementService` and `EventBrokerService` objects, which also requires the appropriate `using` statements:

```
using Genesyslab.Platform.ApplicationBlocks.Commons.Broker;
using Genesyslab.Platform.ApplicationBlocks.Commons.Protocols;
.
.
.
```

```
EventBrokerService eventBrokerService;
ProtocolManagementService protocolManagementService;
.
.
.
protocolManagementService = new ProtocolManagementService();
eventBrokerService =
   BrokerServiceFactory.
   CreateEventBroker(
      protocolManagementService.Receiver);
```

You also need to register your event handlers:

```
eventBrokerService.Register(this.OnEventPackageOpened,
   new MessageIdFilter(EventPackageOpened.MessageId));
eventBrokerService.Register(this.OnEventPackageError,
   new MessageIdFilter(EventPackageError.MessageId));
eventBrokerService.Register(this.OnEventPackageInfo,
   new MessageIdFilter(EventPackageInfo.MessageId));
```

Once you have set up the `StatServerConfiguration` object, you can register it with the `ProtocolManagementService`. At that point, you are ready to open the connection to Stat Server:

```
StatServerConfiguration statServerConfiguration =
   new StatServerConfiguration("Stat_Server_App");
statServerConfiguration.Uri = statServerUri;
protocolManagementService.Register(statServerConfiguration);

protocolManagementService["Stat_Server_App"].Open();
```

Now you are ready to create a statistic. As explained in the article on "Using the Statistics Platform SDK" in the *Statistics Platform SDK API Reference,* you must first create a `StatisticObject`. This includes information about the object you want to monitor:

```
StatisticObject objectDescription =
   new StatisticObject(tenantName,
      dn,
      StatisticObjectType.RegularDN);
```

Then you create a `StatisticMetric`, which describes the information you need and how you want it collected:

```
StatisticMetric statisticMetric =
   new StatisticMetric("TotalNumberInboundCalls");
statisticMetric.TimeProfile = "CollectorDefault";
```

```
statisticMetric.TimeRange = "Range0-120";
statisticMetric.Filter = "VoiceCall";
```

Then you combine the `StatisticObject` and the `StatisticMetric` into a `Statistic` and add this new `Statistic` to a `StatisticsCollection`:

```
Statistic inboundCalls =
   new Statistic(objectDescription, statisticMetric);
StatisticsCollection statisticsCollection =
   new StatisticsCollection();
statisticsCollection.AddStatistic(inboundCalls);
```

Before you can send a request to Stat Server, you have to tell it how you want to be notified. In this case, we are asking to be notified every 15 seconds:

```
Notification notification =
   Notification.Create(NotificationMode.Periodical, 15);
```

You are now ready to request that Stat Server open a package, which means "Please send me the requested information at the specified interval":

```
RequestOpenPackage requestOpenPackage =
   RequestOpenPackage.Create(
      132,
      StatisticType.Historical,
      statisticsCollection,
      notification);

protocolManagementService["Stat_Server_App"].
   Send(requestOpenPackage);
```

When the package has been opened, you will receive two messages from the server. The first is an `EventPackageOpened`. This message indicates that Stat Server is ready to start sending information in response to your request. The sample uses the `OnEventPackageOpened` method to handle this message:

```
private void OnEventPackageOpened(IMessage theMessage)
{
   EventPackageOpened packageOpened = theMessage as
EventPackageOpened;
   if (packageOpened != null)
   {
   writeToLogArea("EventPackageOpened:\n"
      + packageOpened + "");
   }
}
```

The second message is an `EventPackageInfo`, which is the server's way of delivering the information you requested. The `OnEventPackageInfo` method that handles this message in the application simply prints the information to the pane at the bottom of the user interface:

```
private void OnEventPackageInfo(IMessage theMessage)
{
   EventPackageInfo packageInfo = theMessage as EventPackageInfo;
   if (packageInfo != null)
   {
      foreach (Statistic statistic in packageInfo.Statistics)
      {
         writeToLogArea("\n\nStatistic Metric is: "
            + statistic.Metric.ToString());
         writeToLogArea("\nStatistic Object is: "
            + statistic.Object);
         writeToLogArea("\nStatistic IntValue is: "
            + statistic.IntValue);
         writeToLogArea("\nStatistic StringValue is: "
            + statistic.StringValue);
         writeToLogArea("\nStatistic ObjectValue is: "
            + statistic.ObjectValue);
         writeToLogArea("\nStatistic ExtendedValue is: "
            + statistic.ExtendedValue);
         writeToLogArea("\nStatistic Tenant is: "
            + statistic.Object.Tenant);
         writeToLogArea("\nStatistic Type is: "
            + statistic.Object.Type);
         writeToLogArea("\nStatistic Id is: "
            + statistic.Object.Id);
         writeToLogArea("\nStatistic TimeProfile is: "
            + statistic.Metric.TimeProfile);
         writeToLogArea("\nStatistic StatisticType is: "
            + statistic.Metric.StatisticType);
         writeToLogArea("\nStatistic TimeRange is: "
            + statistic.Metric.TimeRange + "\n");
      }
   }
}
```

The code example also includes a handler for processing any errors you might receive from Stat Server:

```
private void OnEventPackageError(IMessage theMessage)
{
   EventPackageError packageError = theMessage as EventPackageError;
   if (packageError != null)
   {
      writeToLogArea("EventPackageError:\n"
         + packageError + "\n\n");
   }
}
```

When you run the example, you should receive information similar to that shown at the beginning of this section.

Complex Example

This example shows how to work with multiple servers in a single application, using the services of the Protocol Manager and Message Broker Application Blocks. It makes use of concepts you have learned from the previous two examples to connect with Configuration Server and Stat Server, but there is one important difference, which will be explained shortly.

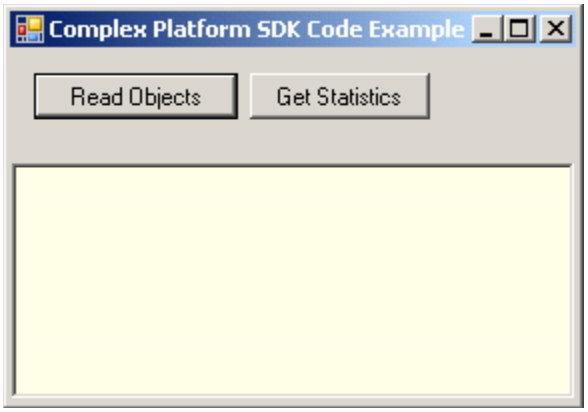

Figure 4: Complex Example User Interface

When you click the `Read Objects` button in this sample, you will receive configuration information that is similar to what you saw in the Configuration Example. Likewise, when you click the `Get Statistics` button, you will receive periodic statistics information similar to what you saw in the Statistics Example.

In order to run this sample, you need to enter the appropriate configuration information, just as you did in the two previous examples. You also need to set up `ProtocolManagementService` and `EventBrokerService` objects, and the appropriate server Configuration objects.

The main difference between this example and the previous ones is that you are now receiving events from more than one server. However, not all of the events that are used in the Platform SDK have unique names.

For example, most of the servers have an `EventError`. There may be times when you need to handle an error from Configuration Server differently from one created by Stat Server. Some servers send an `EventAck`, which may also be subject to special handling, depending on the server it came from.

This code example shows how to set up message filters that discriminate based on the source of the message. In order to do this, you first need to set up a protocol object for each server, as shown here:

```
ConfServerProtocol confServerProtocol =
   protocolManagementService["Config_Server_App"]
      as ConfServerProtocol;
```

You are doing this so you can get a description of that protocol. This description will be used by the message filter to determine which server to listen to. Here is the code to obtain the `ProtocolDescription` object for Configuration Server:

```
ProtocolDescription configProtocolDescription = null;
if (confServerProtocol != null)
{
   configProtocolDescription =
      confServerProtocol.ProtocolDescription;
}
```

And here is how to get the protocol description for Stat Server:

```
StatServerProtocol statServerProtocol =
   protocolManagementService["Stat_Server_App"]
      as StatServerProtocol;
ProtocolDescription statProtocolDescription = null;
if (statServerProtocol != null)
{
   statProtocolDescription =
      statServerProtocol.ProtocolDescription;
}
```

When you register the event handlers for this application, the code is almost identical to the code used in the previous two examples. However, in this case, there is an extra argument for each message filter constructor. This extra argument uses a `ProtocolDescription` object to make sure the event handler will only pay attention to events from the indicated server.

Only two of these registration statements are shown here, but the rest use the same technique:

```
eventBrokerService.Register(
   this.OnStatEventPackageOpened,
      new MessageIdFilter(
         statProtocolDescription,
         EventPackageOpened.MessageId));
...
eventBrokerService.Register(
   this.OnConfEventObjectsRead,
      new MessageIdFilter(
         configProtocolDescription,
         EventObjectsRead.MessageId));
```

There is one more technique demonstrated in this example. Protocol Manager allows you to use either a synchronous or an asynchronous open. The asynchronous method will open all of your server connections with a single statement. There is also an asynchronous close method which will close every connection with a single statement.

Note: If you use the asynchronous open method, you must be sure to wait for all of the connections to open before sending or receiving messages. Likewise, if you use the asynchronous close method, you must also wait for all connections to close before carrying out any processing that relies on a closed connection.

Here is a sample of how to use the asynchronous open method. Note that this statement is commented out in the example.

```
protocolManagementService.BeginOpen();
```

And here is the asynchronous close method:

```
protocolManagementService.BeginClose();
```

Open Media Examples

There are two Open Media examples. The first example is a simple media server that submits a new Open Media interaction. The second example is a client application that can accept an Open Media interaction for processing. Once an interaction has been accepted, the application allows an agent to read information about the interaction and mark it as completed.

Note: These Open Media samples do not use the Protocol Manager and Message Broker Application Blocks. If you are writing production code that uses the functionality explained in these examples, you should use these two application blocks, as explained in the previous examples in this chapter, and in the articles on "Connecting to a Server" and "Event Handling" in the *Open Media Platform SDK API Reference*.

Open Media Server Example

This example is very simple. The user interface has a single button that submits an Open Media interaction using information that has been hard-coded in the application. It also has a window that displays information about the interaction, as shown in Figure 5 on page 36.

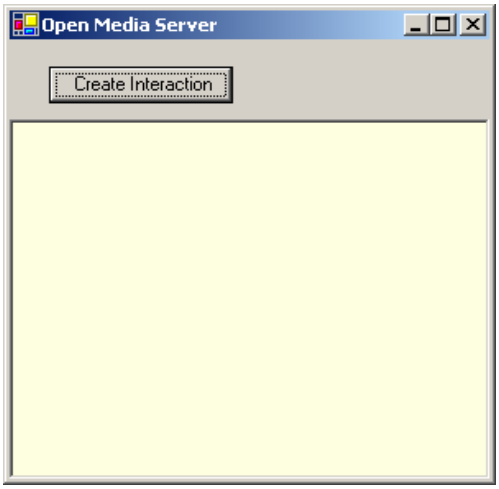

Figure 5: Open Media Server Example User Interface

To use this application, you need to set up some information about your environment. Find the following statements in the source code for `Form1.cs`, and enter values that match your Genesys Configuration Layer:

```
// Enter configuration information here:
private string interactionServerName = "<server name>";
private string interactionServerHost = "<host>";
```

```
private int interactionServerPort = <port>;
private int tenantId = 101;
private string inboundQueue = "<queue>";
private string mediaType = "<media type>";
// End of configuration information.
```

Once you have entered this information, you can run the program. Click the `Create Interaction` button, and if the interaction was successfully created, you should soon see this message:

```
Response: EventAck
```

Now that you have seen what the program does, let's take a look at how it works.

Since this custom media server is such a simple program, almost all of the code you need to understand is in the `InteractionButton_Click` method, which is called when you click the `Create Interaction` button. The first thing this method does is create a URI, using the server and host information you entered in the configuration section of the program. It then uses that information to create a new `InteractionServerProtocol` object, as shown here:

```
interactionServerUri = new Uri("tcp://"
   + interactionServerHost + ":"
   + interactionServerPort);
InteractionServerProtocol interactionServerProtocol =
   new InteractionServerProtocol(
      new Endpoint(
         interactionServerName,
         interactionServerUri));
```

Now the code specifies the client name and type. This is also a good time to set up some user data for the new interaction. In this case, the code shows how to add a subject to the new interaction.

```
interactionServerProtocol.ClientName = "EntityListener";
interactionServerProtocol.ClientType =
   InteractionClient.MediaServer;

KeyValueCollection userData =
   new KeyValueCollection();

userData.Add("Subject", "New Interaction Created by a Custom Media
Server");
```

With this basic setup accomplished, it is time to open the `Protocol` object, and then submit a new interaction, as you see here:

```
try
{
   interactionServerProtocol.Open();

   RequestSubmit requestSubmit = RequestSubmit.Create(
      inboundQueue,
      mediaType,
      "Inbound");
   requestSubmit.TenantId = tenantId;
   requestSubmit.InteractionSubtype = "InboundNew";
   requestSubmit.UserData = userData;
   IMessage response =
      interactionServerProtocol.Request(requestSubmit);
```

Once the `Request` has been processed, you will receive a message from Interaction Server, which will then be printed to the information pane of your application:

```
   LogAreaRichTextBox.Text = LogAreaRichTextBox.Text
      + "Response: " + response.Name + ".\n\n";
```

Open Media Client Example

This example enables an agent to receive an Open Media interaction, accept it for processing, and mark it done. As you can see in Figure 6, the example has buttons to log an agent in and out and to make the agent ready or not ready. Once the agent has logged in, he or she can click the `Receive` button to receive an interaction, and then click the `Accept` button to accept it.

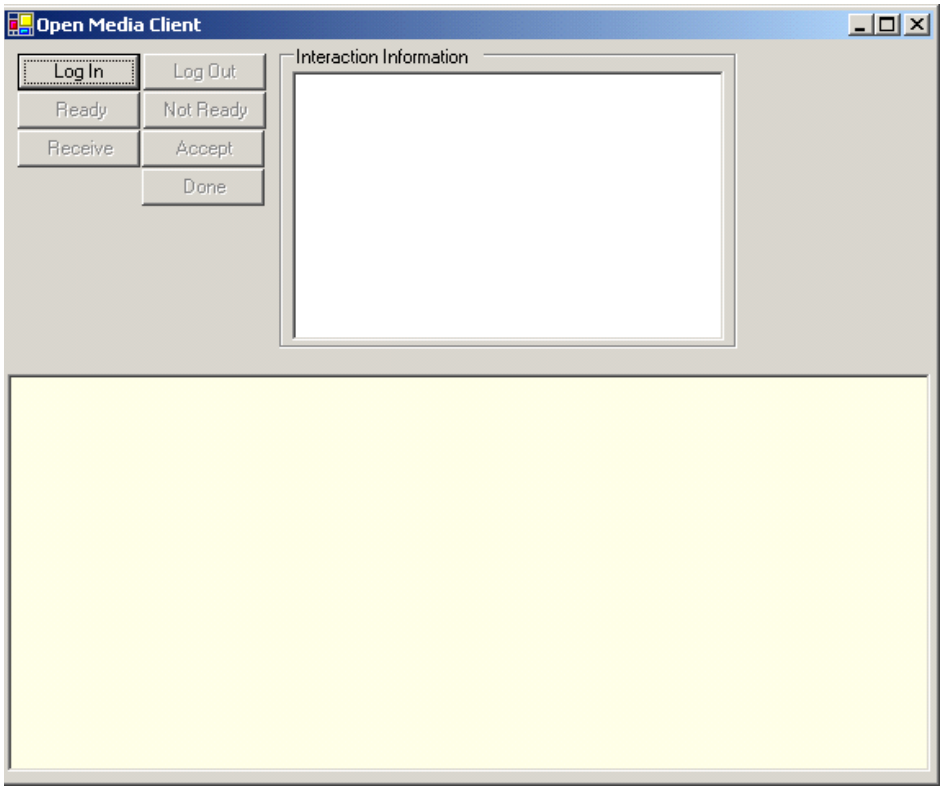

Figure 6: Open Media Client Example User Interface

Figure 7 shows the information supplied by the application after an agent has accepted an interaction and marked it done.

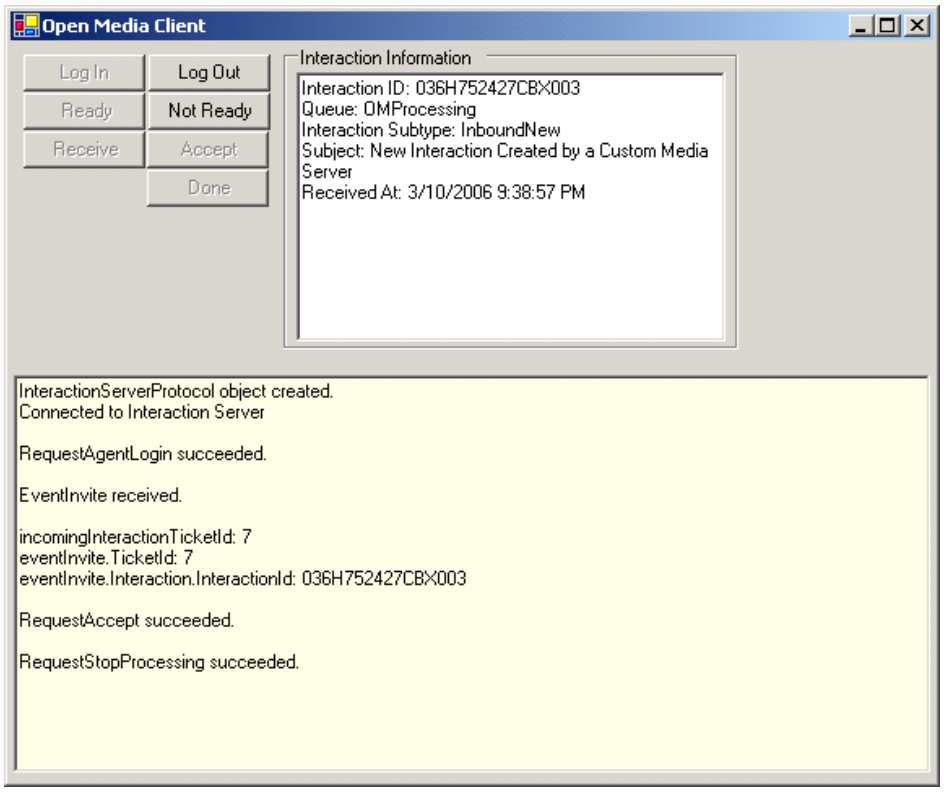

Figure 7: A Completed Interaction

To use this application, you need to set up some information about your environment. Find the following statements in the source code for `Form1.cs`, and enter values that match your Genesys Configuration Layer:

```
// Enter configuration information here:
private string mediaType = "<media type>";
private string placeId = "<place ID>";
private string interactionServerName = "<Interaction Server>";
private string interactionServerHost = "<host>";
private int interactionServerPort = <port>;
// End of configuration information.
```

Once you have set up the configuration information, you can run the application. Here is what it does.

When the application starts, the `Form1` constructor code sets the state of the buttons in the user interface. Then it adds your media type to a list of the media types your agent will use while logged in. Finally, it builds the URI for the Interaction Server:

```
LoginButton.Enabled = true;
ReadyButton.Enabled = false;
LogoutButton.Enabled = false;
NotReadyButton.Enabled = false;
ReceiveInviteButton.Enabled = false;
AcceptInteractionButton.Enabled = false;
DoneButton.Enabled = false;
mediaList.Add(mediaType, 1);
interactionServerUri = new Uri("tcp://"
   + interactionServerHost + ":"
   + interactionServerPort);
```

Once the application is running, your agent can log in. Clicking the `Log In` button will run the `LoginButton_Click` method, which first checks to see whether the `InteractionServerProtocol` has been created. If not, the following code is executed, creating a new protocol object and opening a connection to Interaction Server:

```
interactionServerProtocol =
   new InteractionServerProtocol(
      new Endpoint(
         interactionServerName,
         interactionServerUri));
interactionServerProtocol.ClientType
   = InteractionClient.AgentApplication;
interactionServerProtocol.Open();
writeToLogArea("InteractionServerProtocol object created."
   + "\nConnected to "
   + interactionServerUri + "\n\n");
protocolObjectCreated = true;
```

At this point, the agent can be logged in, using the media list created in the constructor call. The first step is to create the login request and set up the media list:

```
try
{
   RequestAgentLogin requestAgentLogin =
      RequestAgentLogin.Create(
      101,
      placeId,
      null);
   requestAgentLogin.MediaList = mediaList;
```

Now the message can be sent to the server. The `Request` method does the sending and then waits for a response:

```
   IMessage respondingEvent
      = interactionServerProtocol.Request(requestAgentLogin);
```

When the response comes in, the code checks to see whether it was successful, and then updates the user interface to reflect the agent's new status:

```
   if (checkReturnMessage(requestAgentLogin.Name, respondingEvent))
   {
      LoginButton.Enabled = false;
      ReadyButton.Enabled = false;
      LogoutButton.Enabled = true;
      NotReadyButton.Enabled = true;
      ReceiveInviteButton.Enabled = true;
   }
}
catch (ProtocolException protocolException)
{
   MessageBox.Show("Protocol Exception!\n" + protocolException);
}
```

Note that the first line of this snippet calls the `checkReturnMessage` method. This method handles return messages for several of the requests issued in the sample application. As you can see from its signature, this method uses the request type and the response from the server as input. It returns a Boolean value indicating whether the request was successful.

```
private bool checkReturnMessage(string requestType,
   IMessage respondingEvent)
```

The body of the method sets the Boolean to `false`, initializes the message that will be printed to the information pane, and determines which event message was received from the server. If an acknowledgement message was received, the Boolean is set to `true`. In any case, a message is written to the information pane:

```
bool success = false;
string messageText = "";
switch(respondingEvent.Id)
{
   case EventAck.MessageId: // 125
      messageText = requestType + " succeeded.\n\n";
      success = true;
      break;
```

```
   case EventError.MessageId: // 126
      messageText = requestType + " failed with an error: \n"
         + respondingEvent.ToString() + "\n";
      break;
   default:
      messageText = "Unexpected Event: "
         + respondingEvent.Name + "\n"
         + "Responding Event ID = " + respondingEvent.Id + "\n"
         + respondingEvent.ToString() + "\n";
      break;
}
writeToLogArea(messageText);
return success;
```

Now the agent can click the `Receive` button. This triggers the `ReceiveInviteButton_Click` method. This method issues a `Receive` method, which waits for a response from the server:

```
IMessage unsolicitedEvent = interactionServerProtocol.Receive();
```

If the method times out, the code indicates to the agent that the queue is empty:

```
if (unsolicitedEvent == null)
{
   LogAreaRichTextBox.Text = LogAreaRichTextBox.Text
      + "The queue is empty!\n\n";
}
```

Otherwise, the code determines whether the server's response is an invitation to process an interaction:

```
switch(unsolicitedEvent.Id)
{
   case EventInvite.MessageId: // We are invited...
      LogAreaRichTextBox.Text = LogAreaRichTextBox.Text
         + "EventInvite received.\n\n";
      invitationReceived = true;
      break;
   case EventError.MessageId: // 126
      LogAreaRichTextBox.Text = LogAreaRichTextBox.Text
         + "EventInvite failed with an error: \n"
         + unsolicitedEvent.ToString() + "\n";
      break;
```

```
   default:
      LogAreaRichTextBox.Text = LogAreaRichTextBox.Text
         + "Unexpected Event: " + unsolicitedEvent.Name + "\n"
         + "Responding Event ID = " + unsolicitedEvent.Id + "\n"
         + unsolicitedEvent.ToString() + "\n";
      break;
}
```

If the agent has been invited to process an interaction, the code prints information about it in the information window at the bottom of the user interface:

```
if (invitationReceived)
{
   EventInvite eventInvite = (EventInvite) unsolicitedEvent;
   incomingInteractionTicketId = eventInvite.TicketId;
   incomingInteractionProperties = eventInvite.Interaction;
   incomingInteractionId = eventInvite.Interaction.InteractionId;
   AcceptInteractionButton.Enabled = true;
   LogAreaRichTextBox.Text = LogAreaRichTextBox.Text
   + "TicketId: "
   + incomingInteractionTicketId + "\n"
   + "InteractionId: "
   + incomingInteractionId + "\n\n";
}
```

Here is a sample of what might be printed there:

```
TicketId: 5
InteractionId: 036H752427CBX002
```

Now the agent can click the `Accept` button, which triggers a request to accept the interaction:

```
RequestAccept requestAccept =
   RequestAccept.Create(
   incomingInteractionTicketId,
   incomingInteractionName);
```

The code then checks the response, and if the response is favorable, it updates buttons on the user interface and prints information about the interaction in the `Interaction Information` pane:

```
IMessage respondingEvent =
   interactionServerProtocol.Request(requestAccept);
```

```
if (checkReturnMessage(requestAccept.Name, respondingEvent))
{
   ReceiveInviteButton.Enabled = false;
   AcceptInteractionButton.Enabled = false;
   DoneButton.Enabled = true;
   InteractionInfoRichTextBox.Text =
      "Interaction ID: " + requestAccept.InteractionId + "\n"
      ...
      + "\n";
}
```

Here is a sample of what might be printed about an interaction created by the Open Media Server example:

```
Interaction ID: 036H752427CBX002
Queue: OMProcessing
Interaction Subtype: InboundNew
Subject: New Interaction Created by a Custom Media Server
Received At: 3/10/2006 9:33:19 PM
```

The agent can view this interaction, and then click the `Done` button.

```
RequestStopProcessing requestStopProcessing =
   RequestStopProcessing.Create(
   incomingInteractionProperties.InteractionId,
   null);
IMessage respondingEvent =
   interactionServerProtocol.Request(requestStopProcessing);
if (checkReturnMessage(requestStopProcessing.Name, respondingEvent))
{
   DoneButton.Enabled = false;
}
```

At this point, the agent can log out, triggering the following code, which sends the logout request and checks the response. If the response is successful, the buttons on the user interface are updated:

```
RequestAgentLogout requestAgentLogout =
   RequestAgentLogout.Create();
IMessage respondingEvent =
   interactionServerProtocol.Request(requestAgentLogout);
if (checkReturnMessage(requestAgentLogout.Name, respondingEvent))
{
   LoginButton.Enabled = true;
   ReadyButton.Enabled = false;
   LogoutButton.Enabled = false;
```

```
    NotReadyButton.Enabled = false;
    ReceiveInviteButton.Enabled = false;
    AcceptInteractionButton.Enabled = false;
    DoneButton.Enabled = false;
}
```

Voice Examples

These examples demonstrate how to log an agent in and out, and how to make him or her ready and not ready. They also show how to make and receive a call, how to put the call on hold and then retrieve it, and how to release the call. The user interface for these examples contains several groups of buttons and an information pane, as shown in Figure 8.

Note: These Voice samples do not use the Protocol Manager and Message Broker Application Blocks. If you are writing production code that uses the functionality explained in these examples, you should use these two application blocks, as explained in the previous examples in this chapter, and in the articles on "Connecting to a Server" and "Event Handling" in the *Voice Platform SDK API Reference*.

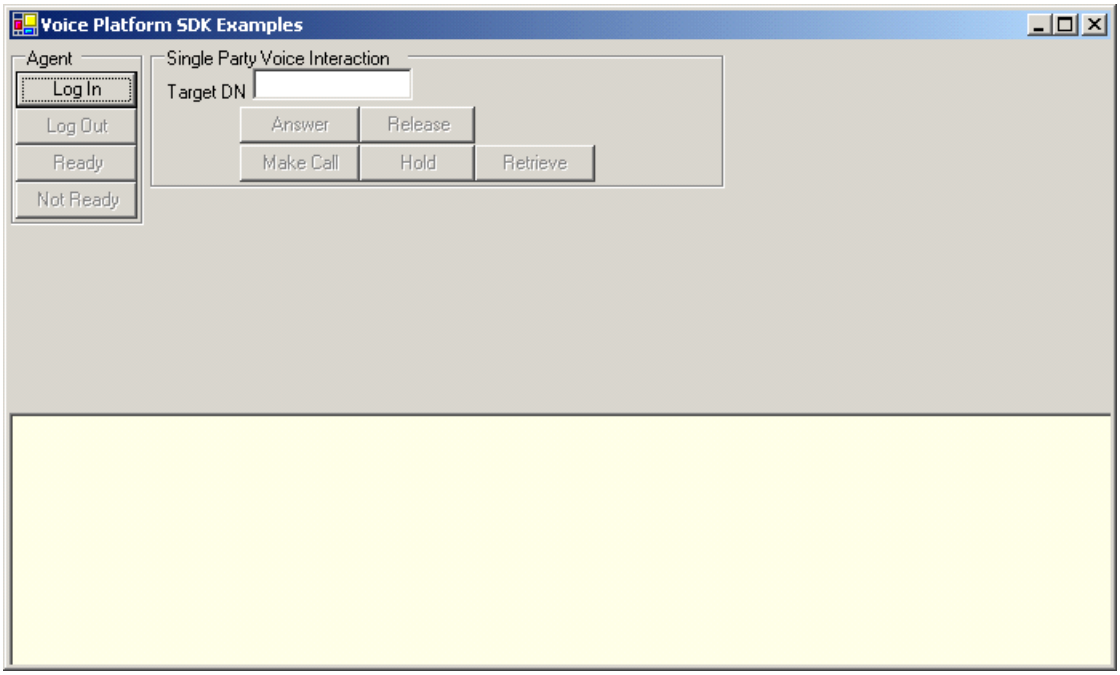

Figure 8: Voice Examples User Interface

To use this example, you will have to set up configuration information in the following lines of `Form1.cs`. Note that not all switches require a user name and password, but they are shown here in case your switch does require them:

```
// Enter configuration information here:
private string tServerName = "<T-Server>";
private string tServerHost = "<host>";
private int tServerport = <port>;
private string thisDn = "<DN>";
// Your switch may not require a user name and password:
private string userName = "<user name>";
private string password = "<password>";
private string thisQueue = "<queue>";
// End of configuration information.
```

Once you have set up the configuration information, you can run the example. When you start the application, the constructor creates the T-Server URI and sets up a thread that will receive messages from the server. It also sets the state of buttons in the user interface, the code for which is not shown here:

```
tServerUri = new Uri("tcp://" + tServerHost + ":" + tServerport);
receiver = new Thread(new ThreadStart(this.ReceiveMessages));
```

When the application starts, you can log in. After you log in, your user interface will look something like Figure 9. As you can see, the `Log In` button is disabled, while the `Log Out` and `Not Ready` buttons are now enabled.

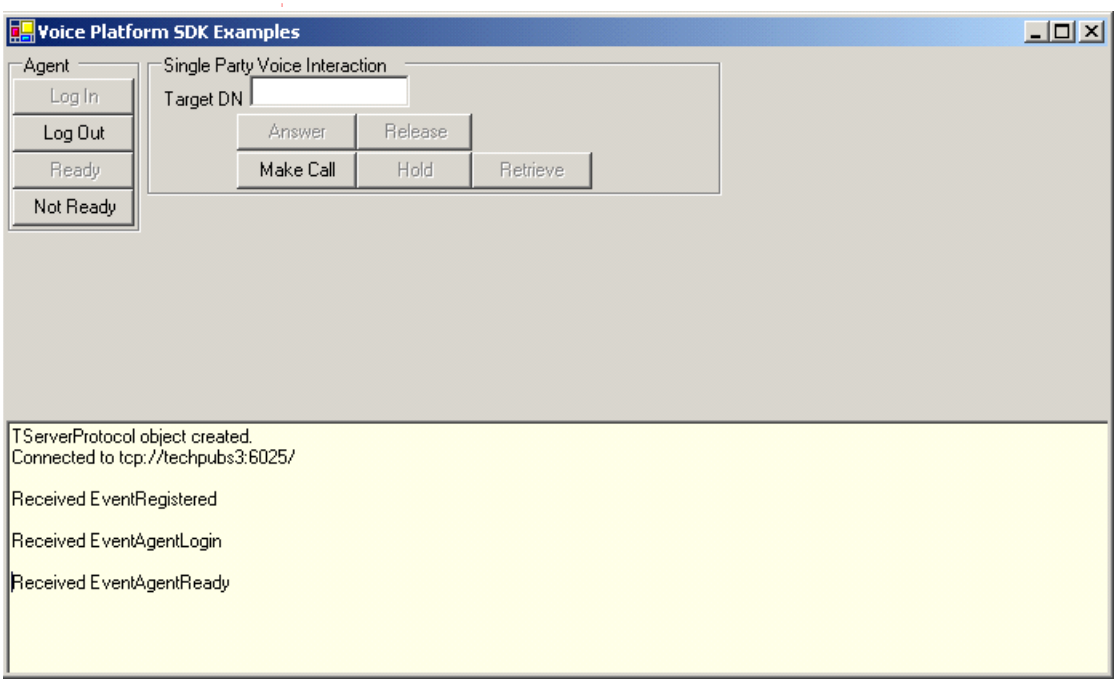

Figure 9: After Logging In

When you click the `log in` button, several things happen. First the code checks to see if a `TServerProtocol` object has been created. If not, it executes the `createProtocolObject()` method.

This method creates a new protocol object and then sets a flag to indicate that the application is running. This flag is used by the message-handling thread. Then the method opens a connection to the server:

```
tServerProtocol =
   new TServerProtocol(
             new Endpoint(
            tServerName,
            tServerUri));
isRunning = true;
receiver.Start();
tServerProtocol.Open();
writeToLogArea("TServerProtocol object created. "
   + "\nConnected to " + tServerUri + "\n\n");
protocolObjectCreated = true;
```

After creating the protocol object, the application issues a request to register the DN you will be using in your session. This must happen before you can log in:

```
RequestRegisterAddress requestRegisterAddress =
   RequestRegisterAddress.Create(
       thisDn,
       RegisterMode.ModeShare,
       ControlMode.RegisterDefault,
       AddressType.DN);
IMessage response = tServerProtocol.Request(requestRegisterAddress);
```

After sending this request, the application checks the response from the server and sends a request to log in. After checking the response, the application updates the state of several buttons in the user interface:

```
checkReturnMessage(response);
RequestAgentLogin requestAgentLogin =
   RequestAgentLogin.Create(
       thisDn,
       AgentWorkMode.AutoIn);
requestAgentLogin.ThisQueue = thisQueue;
// Your switch may not need a user name and password:
requestAgentLogin.AgentID = userName;
requestAgentLogin.Password = password;
response = tServerProtocol.Request(requestAgentLogin);
if (checkReturnMessage(response))
{
   LoginButton.Enabled = false;
   ReadyButton.Enabled = false;
   LogoutButton.Enabled = true;
```

```
   NotReadyButton.Enabled = true;
   MakeCallButton.Enabled = true;
}
```

Since the login request also makes the agent ready, the `Make Call` button is now enabled. Before making a call, let's take a look at the message-handling code in the sample application.

First off, there is a method that is waiting to receive messages as long as the `isRunning` switch is true. When a message is received from the server, this method calls the `checkReturnMessage()` method:

```
private void ReceiveMessages()
{
   while (isRunning)
   {
      if(tServerProtocol.State != ChannelState.Opened)
      {
         System.Threading.Thread.Sleep(500);
         continue;
      }

      IMessage response = tServerProtocol.Receive();

      if(response != null)
      {
         checkReturnMessage(response);
      }
   }
}
```

The `checkReturnMessage()` method contains a switch statement that writes information to the user interface and returns a Boolean indicating whether it received an event it knows about. In some cases, only the Boolean is set, while in others, some slightly more complicated processing takes place. For instance, if the method processes an `EventDialing` message, it remembers the call's Connection ID:

```
private bool checkReturnMessage(IMessage response)
{
   bool knownEvent = false;
   string messageText = "";
   switch(response.Id)
   {
      case EventACK.MessageId:
         knownEvent = true;
         break;
      case EventAgentLogin.MessageId:
         knownEvent = true;
         break;
```

```
      case EventDialing.MessageId:
         EventDialing eventDialing = response as EventDialing;
         connId = eventDialing.ConnID;
         knownEvent = true;
         break;
.
.
.
      case EventError.MessageId: // 126
         messageText = "EventError Received: \n"
            + response.ToString() + "\n";
         break;
      default:
         messageText = "Unexpected Event: " + response.Name + "\n"
            + "Responding Event ID = " + response.Id + "\n"
            + response.ToString() + "\n";
         break;
   }
   if (knownEvent)
   {
      writeToLogArea("Received " + response.Name + "\n\n");
   }
   else
   {
      writeToLogArea(messageText);
   }
   return knownEvent;
}
```

Now that you have seen how messages are processed, here is how to make a call. First you have to fill in the `Target DN` field. Then click the `Make Call` button. At this point, the application requests the call. If the request is successful, the user interface is updated:

```
RequestMakeCall requestMakeCall =
   RequestMakeCall.Create(
       thisDn,
       SinglepartyTargetDNTextBox.Text,
       MakeCallType.DirectAgent);
IMessage response = tServerProtocol.Request(requestMakeCall);
if (checkReturnMessage(response))
{
   MakeCallButton.Enabled = false;
   HoldButton.Enabled = true;
   ReleaseButton.Enabled = true;
}
```

You can place this call on hold, then retrieve it. You can also release the call when you are finished.

These examples also allow you to answer a call. When you receive an `EventRinging`, the `checkReturnMessage()` method saves the Connection ID and then checks to see whether the call is inbound. If it is, the `Answer` button is enabled:

```
case EventRinging.MessageId:
   EventRinging eventRinging = response as EventRinging;
   connId = eventRinging.ConnID;
   if (eventRinging.ThisDN == thisDN)
   {
      AnswerButton.Enabled = true;
   }
   knownEvent = true;
   break;
```

When you click the `Answer` button, the application issues a request to answer the inbound call. Then it checks the response from the server and updates the user interface:

```
RequestAnswerCall requestAnswerCall =
   RequestAnswerCall.Create(
       thisDn,
       connId);
IMessage response = tServerProtocol.Request(requestAnswerCall);
if (checkReturnMessage(response))
{
   AnswerButton.Enabled = false;
   MakeCallButton.Enabled = false;
   HoldButton.Enabled = true;
   ReleaseButton.Enabled = true;
}
```

Chapter

3 About the Java Code Examples

This chapter introduces the Java code examples that accompany this developer's guide. It contains the following sections:

Setup for Development

When you install the Platform SDK, be sure that you have the required tools, environment-variable values, and configuration data. See the *Platform SDK 7.6 Deployment Guide* for details.

The Platform SDK includes all Genesys libraries and third-party libraries needed for proper operation, while the Genesys Developer Documentation CD includes the Platform SDK code examples and a PDF version of this developer's guide.

To download the latest version of this document, or additional code examples, visit the DevZone website at http://www.genesyslab.com/developer.

Source-Code Examples

This chapter refers to the Java source-code examples and application blocks supplied with Platform SDK. These examples are provided on the Genesys Documentation CD in a `.zip` archive file.

The code examples illustrate the use of some common features of the Platform SDKs. The examples are not tested and are not supported by Genesys.

Required Third-Party Tools

To develop Java applications that use the Platform SDK, you will need a Java compiler (such as the one delivered in the Java 2 Platform Standard Edition Development Kit (JDK), version 1.5.x, from Sun Microsystems.

Note: The Sun JDK does not support all platforms. Your platform vendor can tell you how to obtain a JDK that supports your platform.

For this guide, Eclipse was used to compile and run all of the Java code examples.

Java Environment Setup

In order to compile and run these examples, the compiler or the JVM needs access to the libraries of the Platform SDK. They are located in the `lib/` subdirectory of the Platform SDK product installation directory.

Additionally, any code examples that use Genesys application blocks also need access to third-party components located in the `thirdparty\jwsdp-1.6\` subdirectory of the Platform SDK product installation directory.

Set the following environment variables:

- Specify the location of the Java Runtime Environment in the `JAVA_HOME` environment variable.
- Specify all Platform SDK `.jar` files in the `CLASSPATH` environment variable.
- For samples that make use of the Genesys application block, include all third component `.jar` files in the `CLASSPATH` environment variable.

Building the Examples

To build one of the Java examples described in this document:

1. Set your required environment variables, as described in "Java Environment Setup" on page 54.
2. Use the `javac <source>.java` command to compile the example source file into a `.class` file.
3. Use the `java <class>` command to run the code example.

Depending on your development configuration, there may be other ways to build the Java example. For example, if you have the Eclipse Integrated Development Environment installed you could simply import the sample code as a new project, set your Java Build Path to include all necessary libraries, and then run the project as a Java Application.

Configuration Data

For the examples provided for this document to work, they need valid configuration data, including connections to servers and configuration objects such as `Place`, `Dn`, `Agent`, and so on.

See the *Platform SDK 7.6 Deployment Guide* for configuration details.

Genesys Application Blocks

Application Blocks are production-ready components that use industry best practices to offer functionality needed by a broad range of Genesys customers.

To make development of Platform SDK applications easier, Genesys includes a number of these application blocks for common tasks such as managing protocol connections and basic message handling. When writing applications that use the Platform SDKs, Genesys recommends that you use these application blocks whenever possible.

Note: For more information on the application blocks included with the Platform SDK, see "The Application Blocks" in Chapter 1.

Both the Protocol Manager and Message Broker application blocks are used within the Java samples, as described in Table 2.

Table 2: Java Code Examples that use Application Blocks

Example Name	Application Blocks Used
Open Media Server Example	(none)
Open Media Client Example	Protocol Manager
Statistics Example	Protocol Manager Message Broker
Configuration Example	Protocol Manager Message Broker

Note: The Open Media examples show how you can handle Platform SDK messages without the Message Broker application block.

Normally, you would set up and manage separate threads for message-handling if not using the Message Broker application block, but these examples are so simple that they carry out all message-handling activity in a single thread.

For a full list of application blocks that are included with the 7.6 release of Platform SDK, along with a brief description of each application block, see your *Platform SDK 7.6 Deployment Guide*.

Using the Protocol Manager Application Block

The Protocol Manager application block is used to handle Platform SDK protocol objects within your applications. It allows you to easily open and manage multiple protocols. Although each of these examples is limited to a single protocol, this application block is very helpful for custom applications that must work with more than one Genesys server.

Since all Platform SDKs require you to create and work with protocol objects to access the underlying Genesys servers, being able to efficiently manage those protocol objects is an important function.

Note: For more information on the application blocks included with the Platform SDK, see "The Application Blocks" in Chapter 1.

To use the Protocol Manager application block, your Java build path must include the `protocolmanagerappblock.jar` library for your project. You also need to import the appropriate application block packages in your code, as shown by the following line:

```
import
   com.genesyslab.platform.applicationblocks.commons.protocols.*;
```

As part of implementing the Protocol Manager application block, each sample uses a `ProtocolManagementServiceImpl` object.

```
ProtocolManagementServiceImpl pmsi;
```

In addition to the Protocol Manager object, you also will use a server-specific configuration object with information about the Genesys server that you are connecting to.

Steps for configuring this object will depend on exactly which Genesys server you are connecting to. A connection to Configuration Server is shown here as an example:

```
ConfServerConfiguration confServerConfiguration;

confServerConfiguration = new ConfServerConfiguration(protocolName);
confServerConfiguration.setClientType(ConfServerClientType.SCE);
confServerConfiguration.setClientName("default");
confServerConfiguration.setUserName(confServerUserName);
confServerConfiguration.setUserPassword(confServerPassword);
try{
   confServerConfiguration.setUri(new URI("tcp://" + confServerHost
+ ":" + confServerPort));
}
catch(URISyntaxException e){
   // add error handling
   e.printStackTrace();
}
```

Once the server configuration is ready, you can register it with Protocol Manager and then open and use that protocol:

```
pmsi = new ProtocolManagementServiceImpl();
pmsi.register(confServerConfiguration);
try{
   pmsi.getProtocol(protocolName).open();
}
catch(Exception e){
   // add error handling
   e.printStackTrace();
}
```

Using the Message Broker Application Block

The Message Broker application block is designed to handle events that are received in response to the various requests sent to Genesys servers. Since the Platform SDK is message based, efficient event handling is an important function is almost all custom applications.

For example, the Configuration Example issues a `RequestReadObjects`, which can receive one of two events in response. These events are `EventObjectsRead` and `EventError`.

To use the Protocol Manager application block, you will need to add the `protocolmanagerappblock.jar` library to your project. You also need to import the appropriate application block packages, as shown in this line from the code examples:

```
import
   com.genesyslab.platform.applicationblocks.commons.protocols.*;
```

To use the Message Broker Application Block, add the `messageborkerappblock.jar` library to your project, and import the appropriate application block packages, as shown in this line from the code examples:

```
import com.genesyslab.platform.applicationblocks.commons.broker.*;
```

As part of implementing the Message Broker Application Block, this sample uses an `EventBrokerService` object to handle the two types of messages that are expected. This service is declared with the other class members:

```
EventBrokerService mEventBroker;
```

The `InitializeBroker()` method in the sample includes code to register the two event-handling methods:

```
mEventBroker =
   BrokerServiceFactory.CreateEventBroker(pmsi.getReceiver());
mEventBroker.register(new OnEventError(),
   new MessageIdFilter(EventError.ID));
mEventBroker.register(new OnEventRead(),
   new MessageIdFilter(EventObjectsRead.ID));
```

Finally, there are two classes that do the event handling, as just mentioned. The first class handles `EventError` messages:

```
class OnEventError implements Action<Message> {
   public void handle(Message obj) {
      if (obj != null)
         appendText("Error returned.");
   }
}
```

The `OnEventRead` method is similar, and uses the same principles:

```
class OnEventRead implements Action<Message> {
   public void handle(Message obj) {
      if (obj == null)
         appendText("There are no objects of the requested type\n" +
         "in the Genesys Configuration Layer.");
      else {
         appendText("Response to RequestReadObjects:\n\n" +
               obj.toString());
      }
   }
}
```

Configuration Example

This example allows you to retrieve data from Configuration Server about objects of a specific type. The user interface is shown in Figure 10.

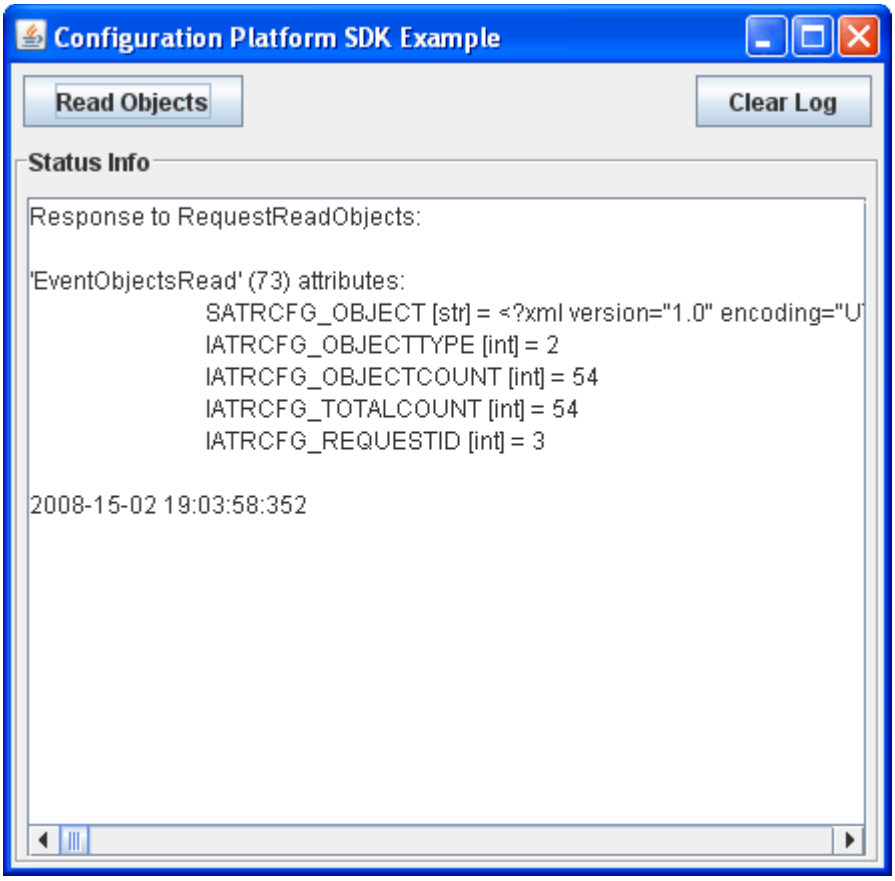

Figure 10: Configuration Example with User Interface

To use this application, you need to set up some information about your environment. Find the following statements in the source code for `ConfigMessageBrokerExample.java`, and enter values that match your Genesys Configuration Layer:

```
// Beginning of Configuration Server information:
private String protocolName = "<protocol>"; //protocol name
private String confServerHost = "<host name>"; //host
private int confServerPort = <port>; //<port>
private String confServerUserName = "<username>"; //username
private String confServerPassword = "<password>"; //password
// End of Configuration Server information.
```

This sample issues a `RequestReadObjects`, which can receive one of two events in response. These events are `EventObjectsRead` and `EventError`. Once you have entered the configuration information above, you are ready to run the program.

Click the `Read Objects` button to create and send a `RequestReadObjects` request to Configuration Server. If the request was successful, then the Event Broker will display a message that lists the event attributes:

```
Response to RequestReadObjects:

'EventObjectsRead' (73) attributes:
   SATRCFG_OBJECT [str] = <?xml version="1.0" encoding="UTF-8"
        standalone="no"?><ConfData>...</ConfData>
   IATRCFG_OBJECTTYPE [int] = 2
   IATRCFG_OBJECTCOUNT [int] = 54
   IATRCFG_TOTALCOUNT [int] = 54
   IATRCFG_REQUESTID [int] = 3

2008-26-02 10:48:15:719
```

Now that you have seen what the program does, let's take a look at how it works.

When this code example begins running, it immediately uses the Protocol Manager and Event Broker application blocks, as described in the sections above. This establishes a connection to the Configuration Server, and provides behavior for various events that are returned.

After the basic setup is accomplished, the `createReadRequest` method provides code that creates and sends a new Read Object request, which is called when you click the `Read Objects` button, as you see here:

```
KeyValueCollection filterKey = new KeyValueCollection();
RequestReadObjects requestReadObjects = RequestReadObjects.create(
      ConfServerObjectType.DN.asInteger(), filterKey);
try{
   pmsi.getProtocol(protocolName).send(requestReadObjects);
}
catch (ProtocolException e){
   e.printStackTrace();
}
```

Once the request has been processed, an event is returned containing the information about one person from Interaction Server. The class registered with your `EventBroker` object then handles the event

Statistics Example

This example shows how to subscribe to a statistic and have it update periodically. The user interface for this example is very simple, with one button that lets you subscribe to (and begin retrieving) statistics information based on hard-coded values within the application, a second button to cancel the current subscription, and a third button to clear the log window. Figure 11 shows the user interface.

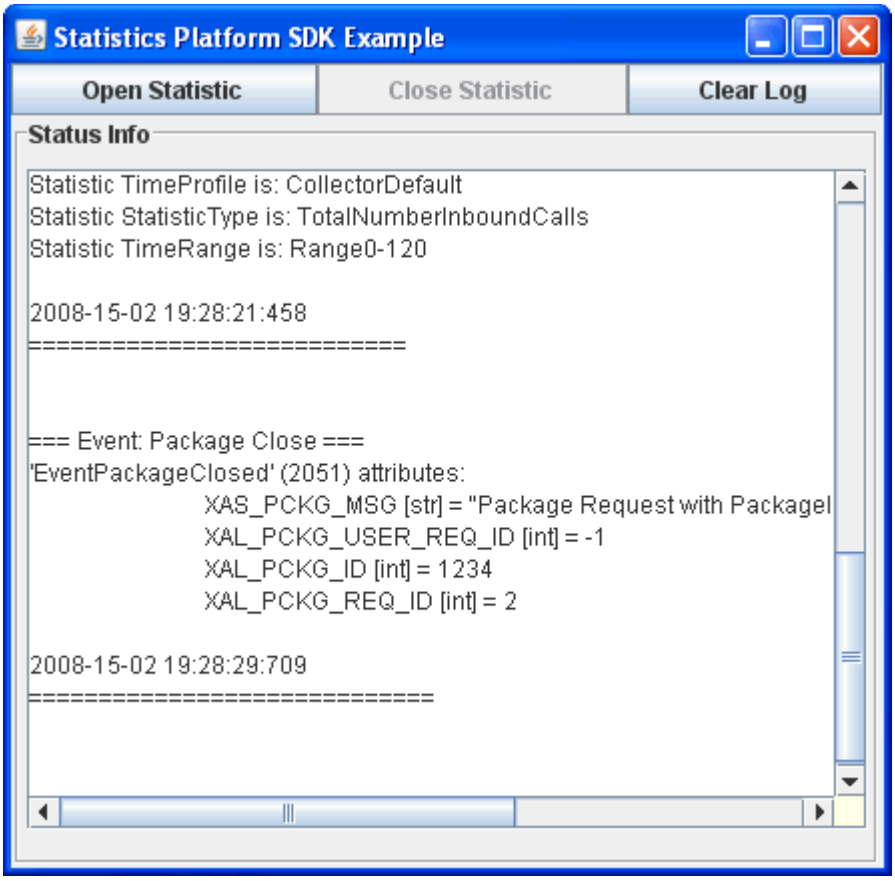

Figure 11: Statistics Example User Interface

To use this example, you will have to set up configuration information in the following lines of `StatisticsPlatformSDKExample.java`:

```
// Beginning of Statistic information:
private int statKey = 1234; // unique key for this Statistic request
private int refreshTime = 10; // notification refresh time (in
seconds)
private String statObjTenantId = "<tenant ID>";
private String statObjId = "<object ID>";
private String statMetricName = "<metric name>";
private String statMetricTimeProfile = "<metric time profile>";
private String statMetricTimeRange = "<metric time range>";
private String statMetricFilter = "<metric filter>";
// End of Statistic information.
```

When this code example begins running, it immediately uses the Protocol Manager and Event Broker application blocks, as described in the sections above. This establishes a connection to the Statistics Server, and provides behavior for the various events that are returned.

At this point, it's time to define some statistics. To do this, the `createStatisticsCollection` method first creates and defines metrics for the statistic:

```
// build a new StatisticMetric
StatisticMetric statisticMetric = new
StatisticMetric(statMetricName);
statisticMetric.setTimeProfile(statMetricTimeProfile);
statisticMetric.setTimeRange(statMetricTimeRange);
statisticMetric.setFilter(statMetricFilter);
```

Then, a new statistics object is created that identifies the Configuration Layer object to collect information about the object's type, and the tenant housing the object. In this case, the object is a DN:

```
// build a new StatisticObject
StatisticObject objectDescription = new
StatisticObject(statObjTenantId,
     statObjId, StatisticObjectType.RegularDN);
```

Once the statistics object and metrics have been defined, they are used to create a new statistic. Then a new statistics collection is instantiated, and the new statistic is added to the collection:

```
Statistic statistic = new Statistic(objectDescription,
statisticMetric);
// create a new StatisticCollection
StatisticsCollection statisticsCollection = new
StatisticsCollection();
statisticsCollection.addStatistic(statistic);
```

This statistic will be used in the `checkStatistic` method. This method creates a `Notification` object, which gives the notification interval, creates a request to open a statistics package, and then sends the request to the server:

```
Notification notification = Notification.create(
     NotificationMode.Periodical, refreshTime);
RequestOpenPackage requestOpenPackage =
RequestOpenPackage.create(statKey,
     StatisticType.Historical, createStatisticsCollection(),
notification);
try {
mProtocolManager.getProtocol(
     protocolName).send(requestOpenPackage);
```

When the package has been opened, a message comes in from the server with statistics information. This message is handled by the `OnEventPackageOpened` class, which was registered with Event Broker:

```
class OnEventPackageOpened implements Action<Message> {
   public void handle(Message obj) {
      if (obj == null)
         System.out.println("nothing returned");
      else
         // handle EventPackageOpened
         appendLogMessage("=== Event: Package Open ===\n"
            + obj.toString()
            + "\n\n" + createTimeStamp()
            + "\n============================");
   }
}
```

Open Media Examples

There are two Open Media examples. The first example is a simple media server that submits a new Open Media interaction. The second example is a client application that can accept an Open Media interaction for processing. Once an interaction has been accepted, the application allows an agent to read information about the interaction and mark it as completed.

Open Media Server Example

This example is very simple. The user interface has a single button that submits an Open Media interaction using information that has been hard-coded in the application. It also has a window that displays information about the interaction, as shown in Figure 12.

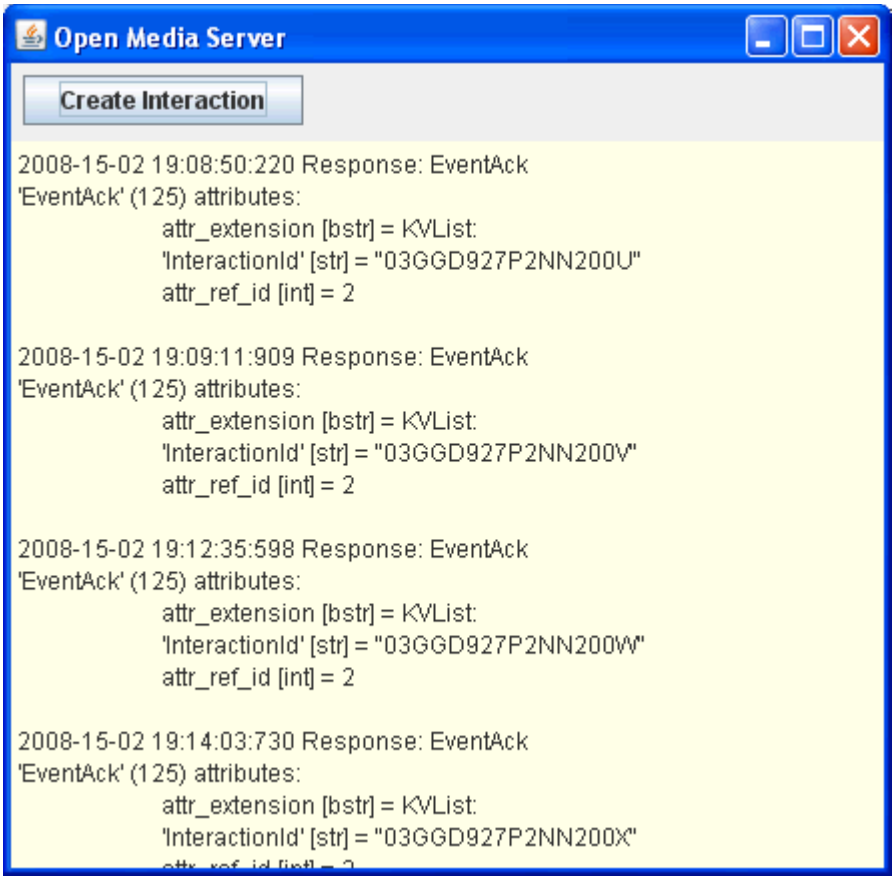

Figure 12: Open Media Server Example User Interface

To use this application, you need to set up some information about your environment. Find the following statements in the source code for `ConfigMessageBrokerExample.java`, and enter values that match your Genesys Configuration Layer:

```
private String protocolName = "<protocol>"; //protocol name
private String interactionServerHost = "<host name>"; //host
private int interactionServerport = <port>; //<port>
private Integer tenantId = new Integer(101);
private String inboundQueue = "<queue>"; //queue
private String mediaType = "<media type>"; //media type
```

Once you have entered this information, you can run the program. Click the `Create Interaction` button, and if the interaction was successfully created, you should soon see this message:

```
Response: EventAck
```

Now that you have seen what the program does, let's take a look at how it works.

Since this custom media server is such a simple program, almost all of the code you need to understand is in the `createInteraction` method, which is called when you click the `Create Interaction` button. The first thing this method does is create an `InteractionServerProtocol` object, using the server and host information you entered in the configuration section of the program.

```
interactionServerProtocol = new InteractionServerProtocol(
        new Endpoint(protocolName,
                interactionServerHost,
                interactionServerport));
```

Now the code specifies the client name and type. This is also a good time to set up some user data for the new interaction. In this case, the code shows how to add a subject to the new interaction.

```
interactionServerProtocol
        .setClientName("EntityListener");
interactionServerProtocol
        .setClientType(InteractionClient.MediaServer);

KeyValueCollection userData = new KeyValueCollection();

userData.addString("Subject",
                "New Interaction Created by a Custom Media Server");
```

With this basic setup accomplished, it is time to open the `Protocol` object, and then submit a new interaction, as you see here:

```
try {
    interactionServerProtocol.open();

    RequestSubmit requestSubmit = RequestSubmit
            .create(inboundQueue, mediaType,
                    "Inbound");
    requestSubmit.setTenantId(tenantId);
    requestSubmit
            .setInteractionSubtype("InboundNew");
    requestSubmit.setUserData(userData);
```

Once the `Request` has been processed, you will receive a message from Interaction Server, which will then be printed to the information pane of your application:

```
Message response = interactionServerProtocol
            .request(requestSubmit);
```

```
String message = createTimeStamp()
            + " Response: " + response.messageName()
            + "\n" + response.toString() + "\n\n";
    writeLogMessage(message, "regular");
```

Open Media Client Example

This example enables an agent to receive an Open Media interaction, accept it for processing, and mark it done. As you can see in Figure 13, the example has buttons to log an agent in and out and to make the agent ready or not ready. Once the agent has logged in, he or she can click the `Receive` button to receive an interaction, and then click the `Accept` button to accept it.

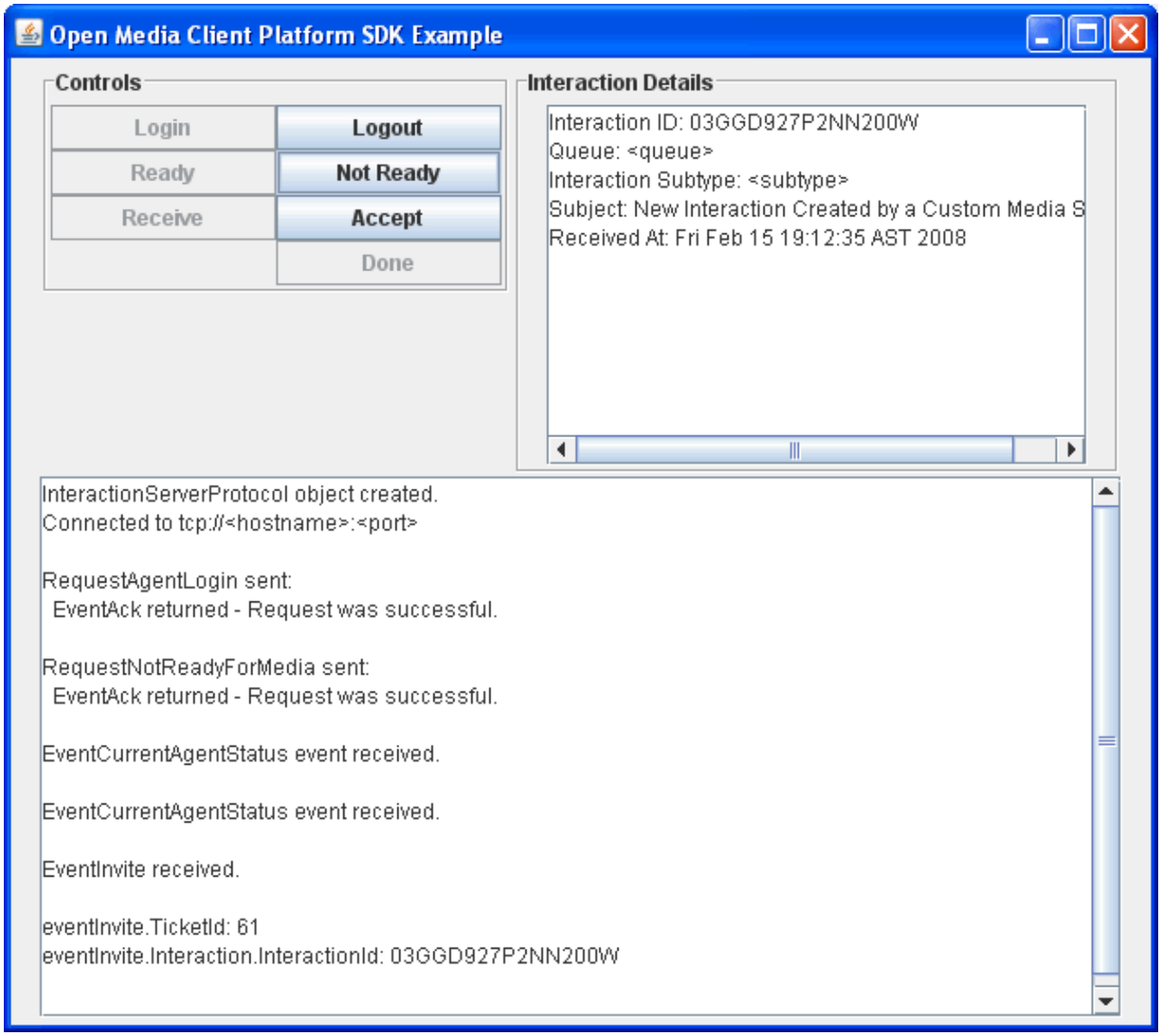

Figure 13: Open Media Client Example User Interface

To use this application, you need to set up some information about your environment. Find the following statements in the source code and enter values that match your Genesys Configuration Layer:

```
// Enter your Interaction Server information here:
private String protocolName = "<protocol>";
private String ixnServerHost = "<host name>";
private int ixnServerPort = <port>;
private String mediaType = "<media type>";
private String placeId = "<place name>";
private long timeoutLength = 10000;
// End of Interaction Server information.
```

Once you have set up the configuration information, you can run the application. Here is what it does.

When this code example begins running, it immediately uses the Protocol Manager and Event Broker application blocks, as described in the sections above. This establishes a connection to Open Media Server, and provides behavior for the various events that are returned.

Once the application is running, your agent can log in. Clicking the `Log In` button will run the `logicLogin` method. At this point, the agent can be logged in, using the media list created in the constructor call. The first step is to create the login request and set up the media list:

```
try {
   RequestAgentLogin reqAgentLogin = RequestAgentLogin.create(101,
placeId, null);
   KeyValueCollection kvc = new KeyValueCollection();
   kvc.addPair(new KeyValuePair(mediaType, mediaType));
   reqAgentLogin.setMediaList(kvc);
```

Now the message can be sent to the server. The `Request` method does the sending and then waits for a response:

```
   Message msgResponse =
mProtocolManager.getProtocol(protocolName).request(reqAgentLogin);
```

When the response comes in, the code checks to see whether it was successful, and then updates the user interface to reflect the agent's new status:

```
   if (checkReturnMessage(msgResponse, "RequestAgentLogin
sent:\n")){
      logoutButton.setEnabled(true);
      readyButton.setEnabled(false);
      readyButton.setEnabled(true);
      receiveButton.setEnabled(true);
   }
```

```
   else {
      loginButton.setEnabled(true);
      appendLogMessage("Could not login. Null message returned on
RequestAgentLogin attempt.\n\n");
   }
}
catch (ProtocolException e) {
   e.printStackTrace();
}
```

Note that the first line of this snippet calls the `checkReturnMessage` method. This method handles return messages for several of the requests issued in the sample application. As you can see from its signature, this method uses the request type and the response from the server as input. It returns a Boolean value indicating whether the request was successful.

```
private boolean checkReturnMessage(string requestType,
   Message respondingEvent)
```

The body of the method sets the Boolean to `false`, initializes the message that will be printed to the information pane, and determines which event message was received from the server. If an acknowledgement message was received, the Boolean is set to `true`. In any case, a message is written to the information pane.

Now the agent can click the `Receive` button. This triggers the `logicReceive` method. This method issues a `Receive` method, which waits for a response from the server:

```
Message unsolicitedEvent =
mProtocolManager.getReceiver().receive(timeoutLength);
```

If the method times out, the code indicates to the agent that the queue is empty. Otherwise, the code determines whether the server's response is an invitation to process an interaction. If the agent has been invited to process an interaction, the code prints information about it in the information window at the bottom of the user interface:

```
switch(unsolicitedEvent.messageId())
{
   case EventInvite.ID: // We are invited...
      EventInvite eventInvite = (EventInvite) unsolicitedEvent;
      // save there parameters so that we can accept the invitation
      incomingInteractionTicketId = eventInvite.getTicketId();
      incomingInteractionProperties = eventInvite.getInteraction();
      incomingInteractionName =
incomingInteractionProperties.getInteractionId();
```

```
      // show details of the interaction message
      logMessage = "Interaction ID: " + incomingInteractionName
         + "\nQueue: " +
incomingInteractionProperties.getInteractionQueue()
         + "\nInteraction Subtype: " +
incomingInteractionProperties.getInteractionSubtype()
         + "\nSubject: " +
incomingInteractionProperties.getInteractionUserData().getString("S
ubject")
         + "\nReceived At: " +
incomingInteractionProperties.getInteractionReceivedAt();
      ixnTextArea.setText(logMessage);
      // display message
      logMessage = "EventInvite received.\n\n"
         + "eventInvite.TicketId: "
         + incomingInteractionTicketId + "\n"
         + "eventInvite.Interaction.InteractionId: "
         + incomingInteractionName + "\n\n";
      acceptButton.setEnabled(true);
      break;
   case EventCurrentAgentStatus.ID:
      logMessage = "EventCurrentAgentStatus event received.\n\n";
      receiveButton.setEnabled(true);
      break;
   case EventError.ID: // 126
      logMessage = "EventError received. Details follow:\n"
         + unsolicitedEvent.toString() + "\n\n";
      receiveButton.setEnabled(true);
      break;
   default:
      logMessage = "Unexpected Event: " +
unsolicitedEvent.messageName() + "\n"
         + "Responding Event ID is "
         + unsolicitedEvent.messageId() + "\n"
         + unsolicitedEvent.toString() + "\n\n";
      receiveButton.setEnabled(true);
      break;
}
```

Now the agent can click the `Accept` button, which triggers a request to accept the interaction:

```
RequestAccept requestAccept =
   RequestAccept.create(incomingInteractionTicketId,
incomingInteractionName);
Message respondingEvent =
mProtocolManager.getProtocol(protocolName).request(requestAccept);
```

The code then checks the response, and if the response is favorable, it updates buttons on the user interface and prints information about the interaction in the `Interaction Information` pane:

```
if (checkReturnMessage(respondingEvent, "RequestAccept sent:\n")) {
appendLogMessage("Interaction ID: " +
requestAccept.getInteractionId() + "\n"
     + "Queue: " +
incomingInteractionProperties.getInteractionQueue() + "\n"
     + "Interaction Subtype: " +
incomingInteractionProperties.getInteractionSubtype() + "\n\n"
     + "User Data: \n" +
incomingInteractionProperties.getInteractionUserData().toString() +
"\n\n"
     + "Received At: " +
incomingInteractionProperties.getInteractionReceivedAt().toString()
+ "\n\n");
doneButton.setEnabled(true);
```

The agent can view this interaction, and then click the `Done` button.

```
RequestStopProcessing reqDone =
RequestStopProcessing.create(incomingInteractionProperties.getInter
actionId(), null);
Message msgResponse =
mProtocolManager.getProtocol(protocolName).request(reqDone);
if (checkReturnMessage(msgResponse, "RequestStopProcessing
sent:\n")){
   ixnTextArea.setText("");
   receiveButton.setEnabled(true);
}
else
   doneButton.setEnabled(true);
```

At this point, the agent can log out, triggering the following code, which sends the logout request and checks the response. If the response is successful, the buttons on the user interface are updated:

```
RequestAgentLogout reqAgentLogout = RequestAgentLogout.create();
Message msgResponse =
mProtocolManager.getProtocol(protocolName).request(reqAgentLogout);
```

```
if (checkReturnMessage(msgResponse, "RequestAgentLogout sent:\n")){
   loginButton.setEnabled(true);
   readyButton.setEnabled(false);
   notReadyButton.setEnabled(false);
   receiveButton.setEnabled(false);
   acceptButton.setEnabled(false);
   doneButton.setEnabled(false);
   ixnTextArea.setText("");
}
else
   loginButton.setEnabled(true);
```

Index

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X