



Avaya Interaction Center
Client SDK Programmer Guide

Release 7.2
May 2013
Issue 1.1

© 2013 Avaya Inc. All Rights Reserved.

Notice

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

Documentation disclaimer

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

Link disclaimer

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

Warranty

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

<http://www.avaya.com/support>

License

USE OR INSTALLATION OF THE PRODUCT INDICATES THE END USER'S ACCEPTANCE OF THE TERMS SET FORTH HEREIN AND THE GENERAL LICENSE TERMS AVAILABLE ON THE AVAYA WEB SITE <http://support.avaya.com/LicenseInfo/> ("GENERAL LICENSE TERMS"). IF YOU DO NOT WISH TO BE BOUND BY THESE TERMS, YOU MUST RETURN THE PRODUCT(S) TO THE POINT OF PURCHASE WITHIN TEN (10) DAYS OF DELIVERY FOR A REFUND OR CREDIT.

Avaya grants End User a license within the scope of the license types described below. The applicable number of licenses and units of capacity for which the license is granted will be one (1), unless a different number of licenses or units of capacity is specified in the Documentation or other materials available to End User. "Designated Processor" means a single stand-alone computing device. "Server" means a Designated Processor that hosts a software application to be accessed by multiple users. "Software" means the computer programs in object code, originally licensed by Avaya and ultimately utilized by End User, whether as stand-alone Products or pre-installed on Hardware. "Hardware" means the standard hardware Products, originally sold by Avaya and ultimately utilized by End User.

License type(s)

Concurrent User License (CU). End User may install and use the Software on multiple Designated Processors or one or more Servers, so long as only the licensed number of Units are accessing and using the Software at any given time. A "Unit" means the unit on which Avaya, at its sole discretion, bases the pricing of its licenses and can be, without limitation, an agent, port or user, an e-mail or voice mail account in the name of a person or corporate function (e.g., webmaster or helpdesk), or a directory entry in the administrative database utilized by the Product that permits one user to interface with the Software. Units may be linked to a specific, identified Server.

Copyright

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

Third-party components

Certain software programs or portions thereof included in the Product may contain software distributed under third party agreements ("Third Party Components"), which may contain terms that expand or limit rights to use certain portions of the Product ("Third Party Terms"). Information identifying Third Party Components and the Third Party Terms that apply to them is available on the Avaya Support Web site:

<http://support.avaya.com/ThirdPartyLicense/>

Preventing toll fraud

"Toll fraud" is the unauthorized use of your telecommunications system by an unauthorized party (for example, a person who is not a corporate employee, agent, subcontractor, or is not working on your company's behalf). Be aware that there can be a risk of toll fraud associated with your system and that, if toll

fraud occurs, it can result in substantial additional charges for your telecommunications services.

Avaya fraud intervention

If you suspect that you are being victimized by toll fraud and you need technical assistance or support, call Technical Service Center Toll Fraud Intervention Hotline at +1-800-643-2353 for the United States and Canada. For additional support telephone numbers, see the Avaya Support Web site:

<http://www.avaya.com/support>

Trademarks

Avaya and the Avaya logo are either registered trademarks or trademarks of Avaya Inc. in the United States of America and/or other jurisdictions.

All other trademarks are the property of their respective owners.

Downloading documents

For the most current versions of documentation, see the Avaya Support Web site:

<http://www.avaya.com/support>

Avaya support

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

<http://www.avaya.com/support>

Contents

Preface	9
Purpose.	9
Audience	9
Reason for Re-issue.	9
Related documents	10
Availability	10
Chapter 1: Introduction	11
Features of the Client SDK	11
Limitations of the Client SDK	12
Architecture of the Client SDK	14
Client components.	15
Custom application	15
Client API.	15
SDK client framework	16
Hierarchical Data Store	16
Server components	17
SDK server bridge	17
Web container	17
Messaging providers	18
Basic services	18
User Object Model	18
Client-side integration.	19
Server-side integration	19
Additional documentation for the Client SDK	19
Chapter 2: Client API object model.	21
Communication through the Client SDK.	21
Delivery of communications to application users.	21
Delivery of outbound communications from application users	22
Work items and the Current concept.	23
Relationship between Current and the Working state	23
Identifying the Current work item.	23
How a work item becomes Current.	24
Effect of makeCurrent()	24
Relationship between Current and Voice Trailing.	25
Events in the Client SDK	26
Operations	26
Method calls	26

Contents

Event registration	27
Overview of the object model.	28
Client API objects	29
Application	29
Session	30
User	33
Channel.	33
VoiceChannel	35
EmailChannel	36
ChatChannel	36
WorkList	37
WorkItem	38
MedialInteraction	41
VoiceMedialInteraction.	42
ChatMedialInteraction	45
Document	48
EmailDocument	48
DraftDocument.	49
EmailDraft	49
Attributes on the WorkItem object	50
Attribute storage in the EDU	50
WorkItem functions for attribute storage	51
Data integration with a custom application	51
Attribute data in the Contact table	51
Example: moving data from a contact routing workflow to the database	52
Chapter 3: Sample clients.	55
Overview	55
Features not supported in the sample clients	56
About the Java sample client	56
User interface of the Java sample client	57
Source code for the Java sample client	57
Design configuration files for the Java sample client.	58
Resources for the Java sample client	58
Dependencies of the Java sample client.	59
Developing a custom Java sample client	59
Running a Java sample client from an SDK server system.	60
Running a Java sample client from a non-SDK server system.	61
About the .NET sample client.	61
User interface of the .NET sample client	62

Code and resources for the .NET sample client	62
Dependencies of the .NET sample client.	63
Developing a custom .NET sample client	63
Running a .NET sample client from an SDK server system	64
Running a .NET sample client from a non-SDK server system.	65
Chapter 4: Guidelines for using the Client API	67
Chat interaction guidelines	68
Methods not allowed for chat interactions.	68
Handle TranscriptLine events for chat interactions.	68
Do not use Datawake method.	69
Voice interaction guidelines	69
Using VoiceChannel.Reset	69
Setting the force multiple calls option on the switch	70
Impact of network recovery on voice interaction	70
Callback guidelines	70
Time and date duration guideline.	72
History API guidelines.	73
Retrieving the history for a WorkItem or Customer	73
WorkItemHistory record.	74
CustomerHistory record.	74
Formatting dates for the history of an object	75
Example: retrieve WorkItem history for Java application.	75
Example: retrieve Customer history for Java application	76
AddressBook API guidelines	76
Retrieving the AddressBook object	76
Implementing Address Book searches.	77
Example: Finding a subset of agents based on criteria.	77
Example: Finding a subset of queues based on criteria	78
WrapupSelection API guideline.	78
Event handling guidelines	79
Register listeners for events before calling Session.Initialize	80
Create a separate listener for each event	80
Avoid blocking operations in event handling	83
Handle ConnectionStatusChange and SessionShutdown events	83
State guidelines	85
Check object for the appropriate state	85
Check status of WorkItem.	86
Log in and log out guidelines.	86
Simultaneous log in and log out for Chat and Email	87

Contents

Check WorkItem status during logout	87
Exception handling guidelines	87
NullPointerException and ArgumentNullException	87
ConnectionException	88
AuthenticationException	88
Operation failure and success guidelines	88
OperationFailed	89
OperationSuccess	89
Null return value guidelines.	89
Customization guidelines	90
Customization directory.	90
Customization files	90
SDKEduAttributesToFilter.properties	91
SDKWorkItemAttributesFilter.properties.	91
SDKSessionAttributesFilter.properties	91
SDKWrapupCodesCategoryGroups.properties	92
SDKSupportedCharsets.properties	92
SDKICPropertiesSections.properties	92
Deploying a configuration file	92
Performance considerations	93
Using WebApplicationContext	93
Configuring the messaging service	95
Chapter 5: Compiling and debugging a custom application	99
Supported compilers	99
Logging.	99
Logging at the module boundaries.	100
Client SDK server logging.	101
Client SDK client logging	104
Logging guidelines	105
Tracing issues through Client SDK logs	106
Sample log messages	107
Error messages	114
Client SDK diagnostic information	114
Using the diagnostic API	115
When the Hierarchical Data Store is updated	115
Viewing the HDS diagnostic information in the sample clients	115
Using the HDS diagnostic information to identify problems	116
Debugging problems found with the HDS diagnostic information.	117
Opening the Diagnostic Viewer.	117

Debugging common problems	117
Custom application cannot communicate with Client SDK server.	118
Chat or email work item is not delivered	118
WorkItem state does not change	119
.NET client encounters socket exception error during log in.	120
Getting support	121
Chapter 6: Localization and internationalization	123
Appendix A: Sample scenarios.	125
Login scenario	126
Logout scenario	128
Agent availability scenario	129
Display channel properties scenario	131
Workitem lifecycle scenario.	133
Workitem collaboration scenario	136
OnHold/OffHold indication scenario	138
Display text message scenario	139
Display email scenario	142
New Outbound email scenario	143
Reply to email scenario	144
Display WorkItem History scenario.	146
Display Customer History scenario	148
AddressBook scenario	150
Retrieving Workitem Contact Attributes scenario.	151
Voice Call scenario	153
Appendix B: Additional sample scenarios.	155
Application object scenario.	155
Password change scenario	156
Session object scenarios	156
Session status scenario.	157
Connectivity status scenario	158
Session shutdown request scenario	159
Enable and disable operational state scenario	160
Channel object scenario	161
Enable and disable channel operational state scenario	161
WorkItem object scenarios	162
Display assigned work items scenario.	162

Contents

Prompt on WorkItem arrival scenario	163
Enable and disable work item operational state scenario	165
Access work item attributes scenario	165
Voice interaction scenario	166
OnHold alert on threshold scenario	166
Chat interaction scenarios	167
Inactivity alert on threshold scenario	167
Language filter scenario	168
Customer-generated alert scenario	169
Email document scenarios	169
Apply signatures scenario	169
Support for attachments scenario	170
Wrapup scenarios	171
Access wrapup codes scenario	171
Wrapup dialog box scenario	172
Use terminate reasons scenario	173
Supervisory scenario	173
Join-Us Scenario.	176
Appendix C: Error messages	179
MajorCodes	179
MinorCodes	180
Index	183

Preface

This section contains the following topics:

- [Purpose](#) on page 9
- [Audience](#) on page 9
- [Reason for Re-issue](#) on page 9
- [Related documents](#) on page 10
- [Availability](#) on page 10

Purpose

The purpose of this guide is to provide detailed information about the Client Software Development Kit (Client SDK) for Avaya Interaction Center 7.2.

Audience

This guide is intended primarily for those who use the Client SDK for Avaya Interaction Center 7.2. You should use this guide as an information source for developing a custom application with the Client SDK.

Reason for Re-issue

Updates to the following chapter:

- [Sample clients](#) on page 55

Related documents

This document provides programming and development information for the Client SDK. Additional information related to the Client API is available in the following Avaya IC documentation.

Client API documentation: The Client API documentation is available in JavaDoc and nDoc formats. The Client API documentation includes detailed information about the object model, such as methods, classes, fields, and constructors. For more information, see [Additional documentation for the Client SDK](#) on page 19.

IC Installation Planning and Prerequisites: This document provides information about the supported third-party platforms and other prerequisites required for the Client SDK, including installation information for those platforms. *IC Installation Planning and Prerequisites* also includes planning and deployment information for the Client SDK.

IC Installation and Configuration: This document provides installation information for all Avaya IC components, including the Client SDK.

Avaya IC Readme: This document provides last-minute information that was not available for the publication of this document.

Avaya IC Readme Addendum: This document provides last-minute information that became available after the release of Avaya Interaction Center 7.2.

Availability

Copies of this document are available on the Avaya support Web site, <http://www.avaya.com/support>.

Note: There is no charge for downloading documents from the Avaya Web site.

Chapter 1: Introduction

The Avaya Interaction Center (Avaya IC) Client Software Development Kit (Client SDK) is a client-side toolkit. Using this toolkit, you can:

- Develop a custom agent desktop application that can access Avaya IC core functions.
- Upgrade the existing custom agent desktop applications to access the Avaya IC core functionality.
- Allow a third-party agent framework or application to work with Avaya IC.
- Integrate your custom application or custom code on clients or on servers.

This section includes the following topics:

- [Features of the Client SDK](#) on page 11
- [Limitations of the Client SDK](#) on page 12
- [Architecture of the Client SDK](#) on page 14
- [Client components](#) on page 15
- [Server components](#) on page 17
- [Client-side integration](#) on page 19
- [Server-side integration](#) on page 19
- [Additional documentation for the Client SDK](#) on page 19

Features of the Client SDK

A custom agent desktop application developed using the APIs (Application Programming Interface) from the client SDK can:

- Save valuable screen real estate. The agent desktop does not need to display an Avaya IC agent application.
- Minimize functionality to the features that only required for a specific implementation. The custom application does not have to include features that are not required for an agent in the contact center.

The Client SDK includes the following features:

Client API: This client-side API includes an application-level protocol that hides communication and session management and provides interfaces that are meaningful to custom applications. Avaya designed the Client API to support future Avaya IC enhancements.

Supported technologies:

- Includes .NET and Java libraries for custom application development.
- Works with NAT, firewalls, and proxy servers.

Security: The Client SDK supports the following security:

- Secure Socket Layer (SSL) for synchronous communication
- Advanced Encryption Standard (AES) for asynchronous communication

Supported integrations: The Client SDK supports server-side and client-side integrations with an Avaya IC core system.

Supported scalability: The Client SDK provides the following support for scalability:

- Up to 500 agents per Client SDK server
- Clustered deployment option for multiple Client SDK servers

Support for internationalization: The Client SDK provides internationalization support for custom applications in languages other than US English. For more information, see [Localization and internationalization](#) on page 123.

Limitations of the Client SDK

The Client SDK has some limitations in Avaya IC 7.2. These limitations might impact the development of custom applications and the interoperability of custom applications with Avaya IC agent desktop applications.

The Client SDK imposes the following limitations on a custom application:

Development limitations of the Client SDK: The Client SDK imposes the following limitations on how you use the Client SDK to develop applications:

- You can not customize Avaya IC agent desktop applications with the Client SDK. For information, see *Avaya Agent Integration* and *Avaya Agent Web Client Customization* guides.
- You do not have a generalized Enterprise Application Integration (EAI) interface to access Avaya IC databases. However, you can use Avaya IC workflows to access data in the Avaya IC databases.

Interoperability limitations with Avaya IC agent desktop applications: The Client SDK does not support more than one agent desktop application that works with Avaya IC on the agent desktop at one time. For example, an application user cannot run a custom application developed with the Client SDK and Avaya Agent on a computer at the same time.

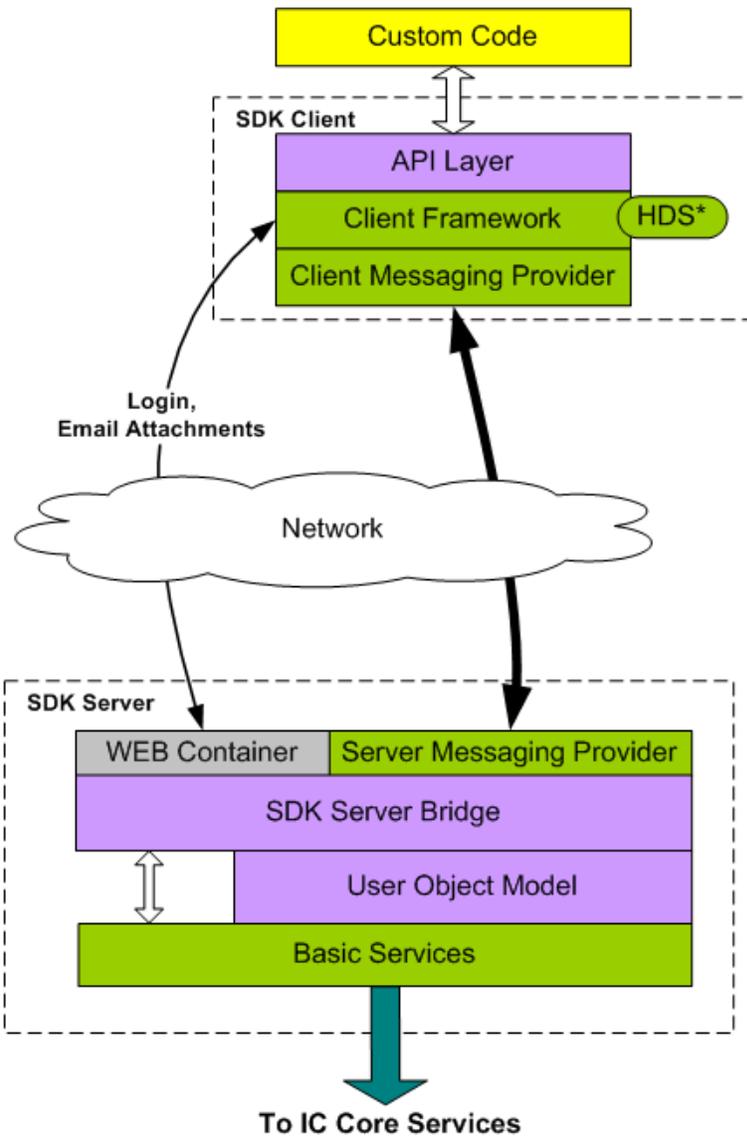
Limitations of the Client API: The Client API does not provide methods to access agent or global resources, email templates, or the suggested responses generated by Avaya Content Analyzer. However, you can develop a custom scheme to store and access the resources in your custom application.

Functionality limitations of custom applications: Custom applications developed with Client SDK do not support the following functionality that is available with the Avaya IC agent desktop applications:

- Ability to monitor queues and telephone events.
- Manual control of the allowable workload for individual channels. For example, the application user cannot change the chat workload.
- Administration of Avaya IC components. For example, you cannot administer agents for Avaya IC through a custom application.
- Ability to monitor real-time statistics.
- Access to Datawake records.
- The following Avaya Agent chat features:
 - Collaborative Form Filling
 - Shared Browsing
 - Datawake

Architecture of the Client SDK

The following diagram shows the Client SDK architecture and how the components work together.



*HDS = Hierarchical Data Store

- | | |
|---|--|
|  Existing Components From IC 7.1 |  Non-Avaya code |
|  Tomcat Component |  Enhanced Version |

Client components

You can use the client components of the Client SDK to:

- Develop custom agent desktop applications.
- Embed an existing agent desktop application at the Avaya IC integration points.

This section includes the following topics:

- [Custom application](#) on page 15
- [Client API](#) on page 15
- [SDK client framework](#) on page 16
- [Hierarchical Data Store](#) on page 16

Custom application

Custom application is the code developed with the Client SDK. You can host the custom code on a client or a server machine. The custom code can create a session for one or more application users. Usually, multiple user sessions can be used if the custom application is hosted on a server machine.

Custom application must use the Client API to access all Avaya IC functions.

Client API

Client API exposes some of the Avaya IC agent functionality that you can use to develop custom agent desktop applications. You can develop a custom application that can use the Client API to access the following functionality of an Avaya IC core system:

- Agent functionality
- Advanced routing and work delivery capabilities

All classes in the Client API use the same object model. The object model includes objects such as Session, Channel, and WorkItem.

Each object include methods that correspond to operations. For example, the Accept method available in the WorkItem object. Calling the Accept method on the WorkItem object means the application user wants to accept the work item that is delivered.

The objects in the Client API determine what happens with a particular client or API request. For example, a request can go directly to the local data store or to a server through messaging.

For more information about the Client API, see [Client API object model](#) on page 21 and the Client API reference documentation.

SDK client framework

The SDK client framework provides the infrastructure for the Client API classes. This infrastructure includes events, interface with communication layers, and error handling.

The SDK client framework lets you:

- Send events to a server.
- Receive events from server.
- Access the Hierarchical data store.

Hierarchical Data Store

The Hierarchical Data Store (HDS) is a client-side data repository or cache for event data received from the Client SDK server components. This cache reduces the number of calls that a custom application needs to make to a server.

The SDK client framework can access data through the Hierarchical Data Store.

The Hierarchical Data Store de-serializes data events from the server into a memory cache.

The Client SDK includes an API that obtains HDS diagnostic information. This diagnostic information provides a snapshot of data of all the objects in the Client API that are cached in the HDS. The Diagnostic Viewer provided with the sample clients displays the data obtained by the Client API. You can use the data provided by the Client API to diagnose errors that occur with your custom applications.

Server components

The Client SDK server is a Web application that is deployed on a Tomcat server. The SDK client components use the Client SDK server to communicate with an Avaya IC core system. The complete server-side code is written using Java.

This section includes the following topics:

- [SDK server bridge](#) on page 17
- [Web container](#) on page 17
- [Messaging providers](#) on page 18
- [Basic services](#) on page 18
- [User Object Model](#) on page 18

SDK server bridge

The SDK server bridge provides the following functionality for the Client SDK:

- Communication between the core Avaya IC services and the custom application through the server messaging provider.
- Interface to interact with the User Object Model (UOM).

The SDK server bridge receives events from the SDK client framework through the server messaging component. Depending on the event received, the SDK server bridge makes the appropriate calls to the UOM. The SDK server bridge uses the Client SDK server objects to determine which calls to make to the UOM.

The SDK server bridge can also receive events from the UOM. The SDK server bridge then uses the client messaging provider to send these events to the Client SDK client framework.

The Client SDK server objects perform all mapping required between the Client SDK and the UOM. The Client SDK server objects are similar to the Client API object model. These objects can:

- Be a wrapper on existing UOM objects.
- Reference the corresponding UOM objects.
- Use their own rules to construct their state based on the UOM objects.

Web container

The Web container is the Tomcat servlet engine and Web server used by the Client SDK server.

Messaging providers

The SDK mediator includes the following messaging providers that are used for asynchronous communication between the client and server:

Client messaging provider: The SDK client components use the client messaging provider to send or receive events.

Server messaging provider: The SDK server components use the server messaging provider to send or receive events.

Basic services

The Client SDK uses the basic services component to communicate with the Avaya IC core system.

User Object Model

The UOM is a representation of an application user and the state of an application user without UI. The UOM consists of a live object model and a well-defined set of access methods. The access methods allow your custom application to retrieve and update the state of all currently connected application users.

The UOM layer is built on the top of VESP Binding and supports the Client SDK. The VESP Binding provides access to the Avaya IC servers.

The UOM is limited strictly to representing the application user and associated services and channels. A separate instance of the UOM exists for each application user connected through the Client SDK. This UOM is assembled when a new application user logs in. The application user receives the channels and services determined in the application user profile. For example, services and channels maintained by a supervisor can be different from those maintained by an agent.

This UOM persists for the duration of an application user session.

The UOM resides in the middle tier and encompasses much of the higher-level functionality behind the Client SDK. For each application user, this layer maintains representations of the following items:

- User
- Channels configured for the user, such as voice, email, or chat
- Media interactions and documents, such as voice, email, or chat, associated with the channels

- Participants associated with the interactions
- Work items that group together related interactions
- Services, such as interaction service and blending service

Client-side integration

For a client-side integration, a single user program creates the integration with the Client SDK and the core Avaya IC system. This user program usually runs on an agent desktop.

In a client-side integration, this user program must:

- Use the Client API.
- Create only one user session per instance of the user program.

Server-side integration

For a server-side integration, a server process creates the integration with the Client SDK and the core Avaya IC system. This server process supports multiple user sessions in a single process. Each user session represents a unique Avaya IC login ID.

Usually, the agent desktop application communicates to this server, either directly or through other servers, to use the Avaya IC functionality.

In a server-side integration, the Client SDK:

- Does not have any presence on the agent desktops.
- Is present only on the server machine.

Additional documentation for the Client SDK

Avaya IC also provides some additional documentation for the Client SDK. You can access this information on the Avaya IC CD-ROMs.

Object model and state models: The following diagrams are available in PDF format:

- Object model: [sdkobjectmodel.pdf](#)
- State models for all stateful Client SDK objects: [sdkstatemodels.pdf](#)

Chapter 1: Introduction

These diagrams are included on the Avaya IC 7.2 Documentation CD-ROM and are installed in the following location:

IC_INSTALL_DIR\IC72\sdk\design

Client API documentation: The Client API documentation is available in JavaDoc and nDoc formats. The Client API documentation includes detailed information about the object model, such as methods, classes, fields, and constructors.

The following table lists the locations where the Client API documentation is installed.

Format	Installation location
Client API documentation for the .NET API	<i>IC_INSTALL_DIR\IC72\sdk\design\dotnet\doc</i>
Client API documentation for the Java API.	<i>IC_INSTALL_DIR\IC72\sdk\design\java\doc</i>

Chapter 2: Client API object model

The Client API is an object oriented Interface. The Client API exposes Avaya IC functionality as classes, methods, and contextual events. The event listener objects consume Avaya IC events.

The Java and .NET classes of the Client API use the same object model. This object model includes all objects, classes, and methods that you can use to determine what happens with a client or API request.

This section includes the following topics.

- [Communication through the Client SDK](#) on page 21
- [Work items and the Current concept](#) on page 23
- [Events in the Client SDK](#) on page 26
- [Overview of the object model](#) on page 28
- [Client API objects](#) on page 29
- [Attributes on the WorkItem object](#) on page 50

Communication through the Client SDK

Communication through the Client SDK is different from communication through the Avaya IC agent desktop applications.

This section includes the following topics:

- [Delivery of communications to application users](#) on page 21
- [Delivery of outbound communications from application users](#) on page 22

Delivery of communications to application users

In the Avaya IC agent desktop applications, the delivery of communications from customers to application users focuses on the voice, email, and chat channels. Each communication is a contact in Avaya Agent or a work item in Avaya Agent Web Client.

The Avaya IC agent desktop applications have a 1:1 ratio of communication to contact or work item delivered to the application user. Each contact or work item represents one communication from a customer. These communications are delivered as communication media to the agent desktop application.

The Client SDK starts with the assumption that all communications are not equal. Communications have different qualities and nuances that need to be considered. Voice takes the largest amount of focus because the communication is interactive and real-time. Chat is also interactive and real-time, but the communication is slower, and a textual transcript allows for context shifts between multiple communications. Email is static in content, not interactive or real-time.

The Client SDK uses a 1:1 ratio of communication to item delivered to an application user.

Instead of delivering a contact or work item as a communication media, the Client SDK delivers a work item in a WorkItem object. A WorkItem can contain the communication as a media interaction or as a static document. The WorkItem allows future enhancements to Avaya IC, such as routing of work that does not initially contain a communication.

In the Client SDK:

- Voice and chat communications are media interactions.
- Email communications are static documents.

For more information about the WorkItem and other objects in the Client API object model, see [Client API object model](#) on page 21.

Delivery of outbound communications from application users

In Avaya IC agent desktop applications and through the Client SDK, application users can initiate work. In Avaya IC 7.2, this work is limited to outbound email messages and voice calls.

Because outbound work is not associated with an existing WorkItem, the custom application must use the channel to create an outbound communication. As a result of this operation, the Client SDK creates a WorkItem. That WorkItem must use the channel that represents the required type of communication.

For example, a MakeCall on the Channel.VoiceChannel object initiates a new outbound voice contact. This MakeCall provides a WorkItem with a VoiceMediaInteraction to represent the new outbound voice communication. Similarly, a NewEmail call on the Channel.EmailChannel creates a WorkItem where a new email draft can be linked.

Note:

In case of callback scenario, an outbound voice call is placed using workitem object, so that both the chat and voice media interactions can be grouped together in a single workitem object.

Work items and the Current concept

The Client SDK uses the concept of Current for work items that are assigned to an application user. The Current concept is also used in the Avaya Agent Web Client, but is not used in Avaya Agent.

To develop an application in the Client SDK, you must understand the concept of Current.

This section includes the following topics:

- [Relationship between Current and the Working state](#) on page 23
- [Identifying the Current work item](#) on page 23
- [How a work item becomes Current](#) on page 24
- [Effect of makeCurrent\(\)](#) on page 24
- [Relationship between Current and Voice Trailing](#) on page 25

Relationship between Current and the Working state

The Current concept was introduced to address the following issues that can occur when the Working state is used to indicate the work item that is being handled by an application user:

Multiple work items can be considered Working at the same time: For example, an application user talks on the telephone to a customer and has an email work item open in the Email application. Because both items are in Working state, Avaya IC cannot determine which work item the application user is actually handling. This problem can skew reporting for that application user.

A work item in Wrapup state cannot be Working: If multiple work items are in the Wrapup state, Avaya IC cannot track which wrapped work item the application user is actually handling. This problem can skew reporting.

Identifying the Current work item

In the Client SDK, a work item is considered Current if that work item has primary focus in the application. Current indicates that the application user is actively handling the work item and that the work item is gathering time for reporting purposes.

Current is not the same as the Working state for work items. A work item need not be working to become a Current workitem. However, a Working work item must be Current or Voice Trailing.

When a work item becomes Current, Avaya IC flags that work item. This flag identifies the work item as the one that has primary focus and is currently being handled.

Only one work item can be Current and considered to be the item that an application user is handling. For example, an application user can have multiple work items in the Wrapped state, but only one of those work items can be Current. You cannot have an additional Current work item, even if a second work item is of a different media type.

How a work item becomes Current

A work item becomes Current when you call `makeCurrent()` on the work item. Usually this call occurs when an application user selects that work item in the application.

Important:

If you want accurate reporting, you must call `makeCurrent()` on the work item at the correct time. For example, if an email does not become Current when opened by an application user, you cannot capture and report on the time the application user worked on that email.

Work items raise a `CURRENT_CONTEXT_CHANGED` event to let consumers know of a change in their Current status.

For example, if WorkItem A is current, and `makeCurrent()` is called on WorkItem B:

1. The Current flag is cleared from WorkItem A, which generates the `WorkItem.CurrentContextChanged` event. An `isCurrent()` call on WorkItem A now returns false.
2. WorkItem B is flagged as the Current work item, which generates a `WorkItem.CurrentContextChanged` event. An `isCurrent()` call on WorkItem B now returns true.

Effect of `makeCurrent()`

When `makeCurrent()` is invoked on WorkItem A, the following events are triggered:

1. If WorkItem B is Current:
 - a. The Current flag is cleared from WorkItem B.
 - b. WorkItem B raises an event that indicates a change in its Current context.
 - c. `isCurrent()` on WorkItem B returns false.
2. If WorkItem B can be deferred or paused:
 - a. The `defer()` method is invoked.
 - b. WorkItem B transitions into the inactive Paused or Deferred state.

- c. A state change event is raised for WorkItem B.
- 3. WorkItem A is flagged as Current:
 - a. WorkItem A raises a WorkItem.CurrentContextChanged event.
 - b. `isCurrent()` returns true.
- 4. If WorkItem A can be activated:
 - a. The `activate()` method is invoked.
 - b. WorkItem A transitions into the Working state.
 - c. A state change event is raised for WorkItem A.

For example, if WorkItem A is inactive when `makeCurrent()` is invoked, WorkItem A is flagged Current and is activated. However, if WorkItem A is in Wrapped state when `makeCurrent()` is invoked, WorkItem A is flagged Current but is not activated. A WorkItem cannot transition from Wrapped state to Working state.

Relationship between Current and Voice Trailing

With voice trailing, an application user can multitask in a multimedia environment that uses the Current concept. For example, with voice trailing, an application user can reply to an email and continue to speak to a customer on the telephone.

In the Client SDK, only one work item can be current. This limit implies that an application user can have only one Working work item and cannot multitask. For example, if an application user talks on the telephone to a customer and selects an email work item, the Current concept inactivates the voice work item. This inactivation automatically places the telephone call on hold.

Voice trailing avoids this automatic inactivation of a voice work item. With voice trailing, a voice work item can be Working but not Current. With voice trailing, a voice work item can continue to trail after an application user selects an email work item to handle. Both work items remain Working, and the application user can handle them simultaneously. However, the Client SDK considers only the email work item to be Current.

You can configure voice trailing through Avaya IC properties. For more information, see *IC Administration Volume 2: Agents, Customers, & Queues*.

Usually, voice trailing can be determined by the following query:

```
WorkItem.isCurrent() == false and WorkItem.getState() == WorkItem.State.WORKING
```



Important:

Only voice work items can be placed in a voice trailing state. Chat cannot satisfy the above query, except during state transitions, which are transient in nature. When a chat transitions out of the Current state, the work item is paused.

Events in the Client SDK

The Client API defines the events for the objects in the Client SDK.

This section includes the following topics:

- [Operations](#) on page 26
- [Method calls](#) on page 26
- [Event registration](#) on page 27

Operations

The objects in the Client API expose methods. Each method represents an operation that can be performed on these objects. Any operation has the following basic outcomes:

- Succeeded
- Failed

A successful operation can include data that needs to be returned as a result of that operation.

Method calls

With the Client API, method calls can be either synchronous or asynchronous.

Synchronous method calls

If the method call is synchronous, the outcome can be easily determined. For example, a return value can indicate the outcome or return data, or the method can throw an exception. Examples of synchronous communication with the Client API include logging in, and fetching and uploading email attachments.

Asynchronous method calls

If the method call is asynchronous, the outcome or return data can be known only through notifications by the Client SDK. This notification is done through events. All asynchronous operations performed on an object must result in an event. The event indicates whether the operation succeeded or failed.

Asynchronous communications use the client messaging provider and the server messaging provider.

A failure event is always explicitly raised if an operation fails. An exception is an explicit indication of failure. An exception might be thrown if the state of an object is invalid.

A success event can be implicit or explicit.

Implicit indication of success: An implicit success indication occurs when an operation causes a change in the object state that clearly indicates the action succeeded. In this case, the state change of the object is sufficient to determine that the operation was successful.

For example, an implicit indication of a successful operation occurs when an application user tries to release or defer a WorkItem:

- If the WorkItem is successfully released, the WorkItem enters the Wrapped state. If the release fails, an OPERATION_FAILED event is generated.
- If the WorkItem is successfully deferred, the WorkItem enters the Deferred state. If the deferral fails, an OPERATION_FAILED event is generated.

Explicit indication of success: An explicit indication occurs in one of the following cases:

- An operation does not cause a state change.
- An operation involves a process where a state change does occur, but the process cannot uniquely identify that an operation has completed successfully.

In such cases, a unique event is generated that indicates success of that operation.

For example, an explicit indication of a successful operation occurs when you set WorkItem attributes:

- If the attributes are written successfully, an OPERATION_SUCCEEDED event is generated.
- If the write fails, an OPERATION_FAILED event is generated.

Event registration

The Client SDK provides a mechanism for clients to register with objects that generate events. Your custom application must add itself as a listener to objects for all events that matters to the application.

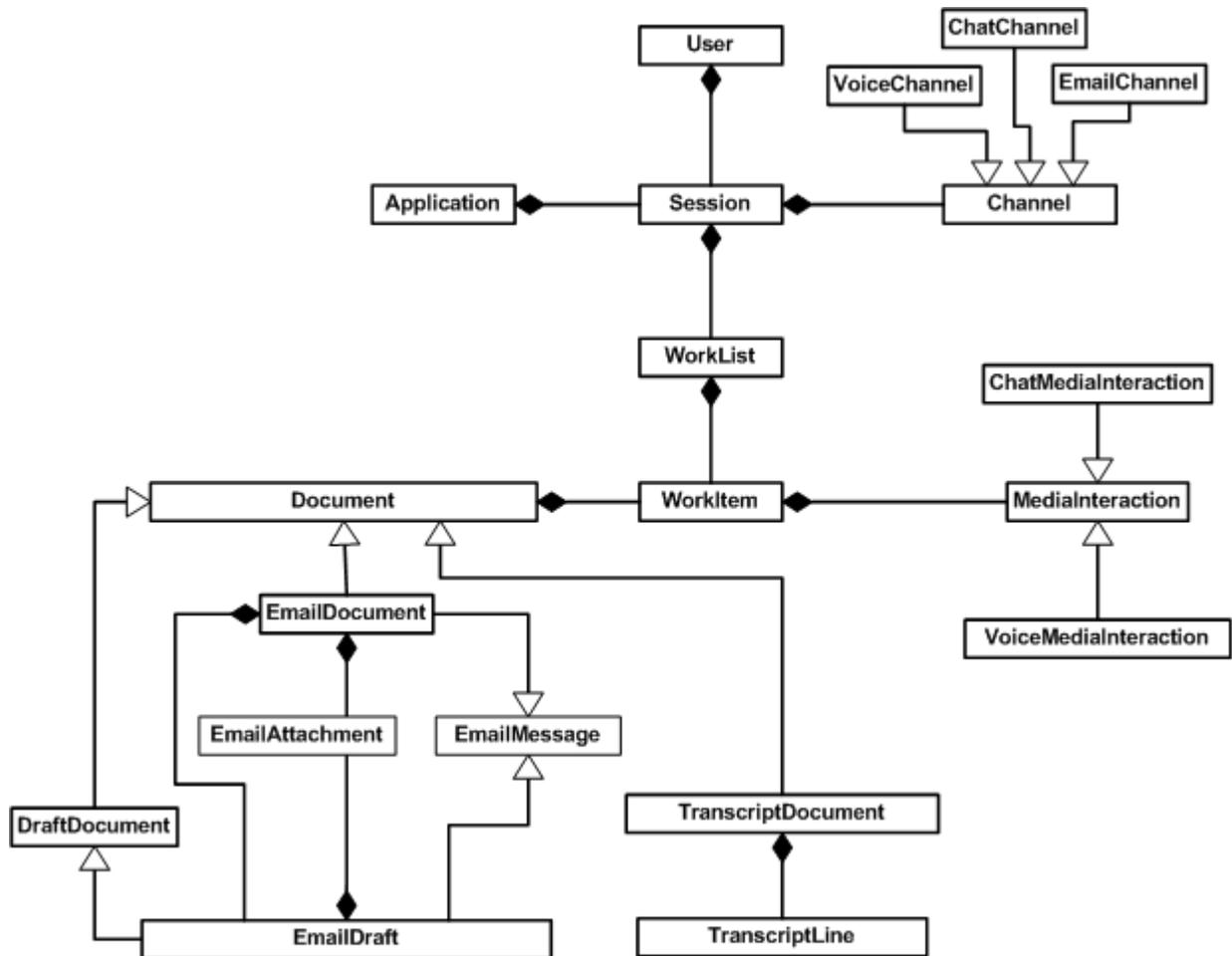
When an event change occurs within an object, the Client SDK generates an event that is propagated to all listeners to that object. Usually, these events are raised as a result of:

- User operations on a particular object
- Changes in the object driven by the core Avaya IC system

Overview of the object model

The following diagram shows the object model that forms the base of the Client API. For more information about each object, see [Client API objects](#) on page 29.

For information about how to access a diagram of the complete object model, see [Additional documentation for the Client SDK](#) on page 19.



Client API objects

This section provides an overview of the objects in the Client API object model. For more information about the classes and methods within these objects, see the API reference documentation provided with the Client API.

This section includes the following topics:

- [Application](#) on page 29
- [Session](#) on page 30
- [User](#) on page 33
- [Channel](#) on page 33
- [VoiceChannel](#) on page 35
- [EmailChannel](#) on page 36
- [ChatChannel](#) on page 36
- [WorkList](#) on page 37
- [WorkItem](#) on page 38
- [MediaInteraction](#) on page 41
- [VoiceMediaInteraction](#) on page 42
- [ChatMediaInteraction](#) on page 45
- [Document](#) on page 48
- [EmailDocument](#) on page 48
- [DraftDocument](#) on page 49
- [EmailDraft](#) on page 49

Application

Description

Application is the starting point for the Client SDK. You can use the Login method of the Application object to provide a [Session](#) object.

Limitations

None.

Sample scenarios

This object is used in the following scenarios:

- [Login scenario](#) on page 126
- [Logout scenario](#) on page 128
- [Password change scenario](#) on page 156
- [Session status scenario](#) on page 157
- [Connectivity status scenario](#) on page 158
- [Session shutdown request scenario](#) on page 159

Session

Description

Session represents the Avaya IC session for a logged in application user. In the Client SDK, Session performs a central role of the primary object that you use to access other objects. You use this object to register for any published events on Client SDK objects. Session also provides the events related to the load, connectivity, and state of an application user.

You can use Session to:

- Register listeners for all published Client SDK object events.
- Receive events regarding agent state, workload, and agent connectivity.
- Manage availability and Auxwork statuses.
- Manage supervisory routines.
- Get system codes for Auxwork, logout, and wrapup.
- Access application user ADU data through the session attribute functions.
- Get character sets and Avaya IC properties.
- Get other objects including [Channel](#) and [WorkList](#).

Limitations

The Session object has the following limitations:

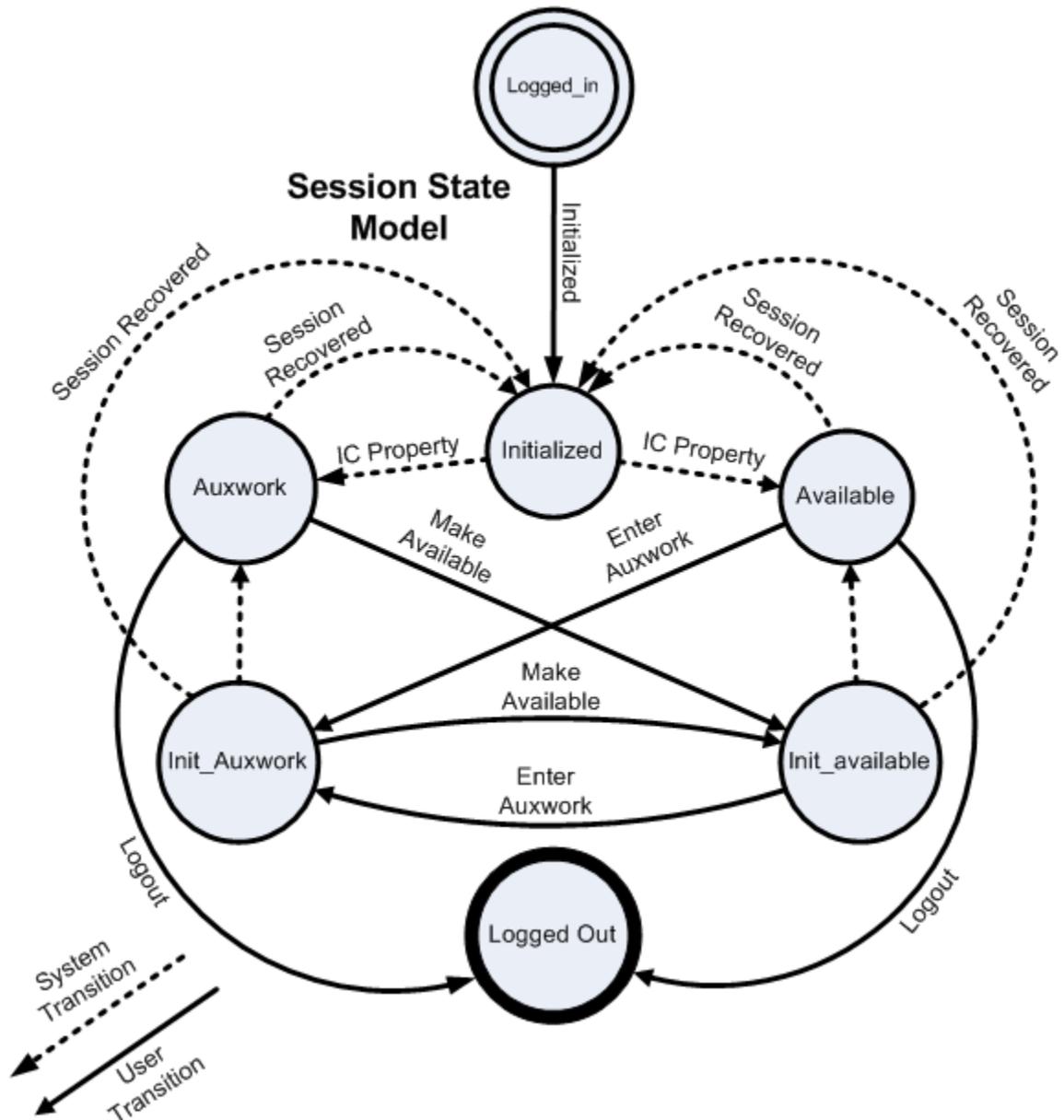
- To use the `GetSessionAttribute` and `SetSessionAttribute` functions, you must have to customize the Client SDK to specify the ADU elements that are available to the function calls.
- To use character sets and Avaya IC properties, you must have to customize the Client SDK to make them available to the function calls.

- To use wrapup codes, Aux codes, and release reason codes, you must have to customize the Client SDK to make them available to the function calls.

For more information, see [Customization guidelines](#) on page 90.

Session state model diagram

The following diagram shows the state model for Session.



State definitions

The following table describes the state definitions included in the state model for Session.

State	Definition
Logged_in	Application user is successfully logged in, but active state has not yet been determined. Logged_in is the initial state reported to clients through the Client SDK.
Initialized	Session was successfully initialized. This state is transitional.
Init_auxwork	Application user submits request to enter Auxwork state. The Avaya IC system uses this state to acknowledge the request and prevent the delivery of new WorkItems.
Auxwork	Application user has successfully entered Auxwork state. This state confirms that all WorkItems are in a state that supports Auxwork. No WorkItems will be delivered until the application user enters Available state.
Init_available	Application user submits request to enter Available state. The Avaya IC system uses this state to acknowledge the request and prepare for the availability of the application user.
Available	Application user has successfully entered Available state. This state confirms that the Avaya IC system completed the request from the application user to become available. The application user can now receive WorkItems on active channels.
Logged_out	Application user has successfully logged out. All WorkItems in the WorkList are in an acceptable logout allowable state, such as deferred. Avaya IC has completed the logout.

Sample scenarios

This object is used in the following scenarios:

- [Login scenario](#) on page 126
- [Agent availability scenario](#) on page 129
- [Session status scenario](#) on page 157
- [Connectivity status scenario](#) on page 158
- [Session shutdown request scenario](#) on page 159
- [Enable and disable operational state scenario](#) on page 160
- [Display channel properties scenario](#) on page 131
- [AddressBook scenario](#) on page 150
- [Prompt on WorkItem arrival scenario](#) on page 163

- [Wrapup dialog box scenario](#) on page 172
- [Use terminate reasons scenario](#) on page 173

User

Description

User represents the application user profile attributes such as name and login ID. You can use the User object to access core application user attributes and retrieve agent information and role.

Limitations

Some channel specific data is unavailable until the [Session](#) is initialized.

Channel

Description

Channel is a generic representation of a path for a communication.

You can use Channel to:

- Access functions specific to a media, independent of an existing [WorkItem](#).
- Receive health and state events specific to the configured media.
- Retrieve configured task loads and ceilings for a given channel.
- Log in to, log out of, or reset a connection for a channel.
- Create a new [WorkItem](#) with outbound intent for voice or email.

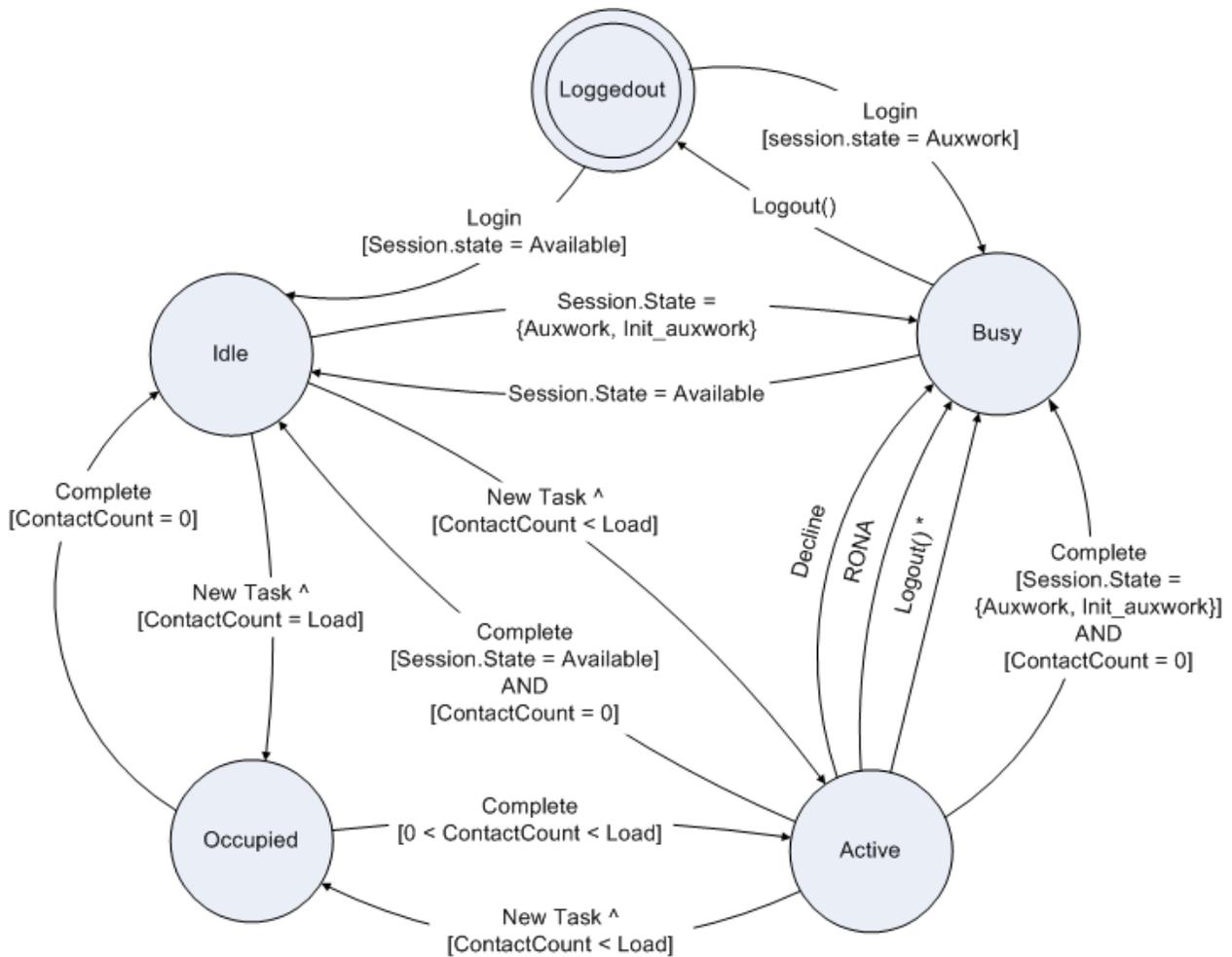
In Avaya IC 7.2, the channels are [VoiceChannel](#), [EmailChannel](#), and [ChatChannel](#).

Limitations

None.

Channel state model diagram

The following diagram shows the state model for Channel.



* **Note:** Logout() from Active forces 2 transitions, Active state to Busy state, then a separate transition from Busy state to Logged out state.

^ New Task transitions consist of either of the following:
 New delivery of a workitem to the agent
 OR
 Agent reactivates a workitem from the deferred state

State definitions

Each ChannelConnection maintains channel state. The state of a channel determines the ability of the system to assign work items to a particular channel. The channel state is triggered by the user.

The following table describes the state definitions included in the state model for Channel.

State	Definition
loggedout	The application user is not currently signed in or logged into the channel.
idle	The application user is available to receive work item on the channel, but does not currently have any work items.
busy	The application user does not want to be available to receive new work items and has set his or her state accordingly. For example, the user is in AuxWork.
occupied	The application user is handling the maximum number of work items permitted by his or her current endpoint setting.
active	The application user is handling at least one work item, but is available to receive more work items.

Sample scenarios

This object is used in the following scenarios:

- [Display channel properties scenario](#) on page 131
- [Enable and disable channel operational state scenario](#) on page 161

VoiceChannel

Description

VoiceChannel represents the channel that you use to work with voice interactions. You can use VoiceChannel to:

- Log in to and log out of the voice channel.
- Initiate new outbound voice communications, resulting in a new [WorkItem](#) with a voice media interaction.
- Reset the connection for the voice channel when the hard phone and soft phone get out of sync.

Limitations

None.

Sample scenarios

This object is used in the following scenarios:

- [Display channel properties scenario](#) on page 131
- [Enable and disable channel operational state scenario](#) on page 161

EmailChannel

Description

EmailChannel represents the channel that you use to work with email documents. You can use EmailChannel to:

- Log in to and log out of the email and chat channels.
- Initiate new outbound email communications, resulting in a new [WorkItem](#) through which the email draft can be composed.
- Reset the connections for the chat and email channels.

Limitations

Resetting, logging in, and logging out of the email channel also affect the chat channel.

Sample scenarios

This object is used in the following scenarios:

- [Display channel properties scenario](#) on page 131
- [Enable and disable channel operational state scenario](#) on page 161

ChatChannel

Description

ChatChannel represents the channel that you use to work with chat interactions. You can use ChatChannel to:

- Log in to and log out of the chat and email channels
- Reset the connections for the chat and email channels

Limitations

ChatChannel has the following limitations:

- ChatChannel works with inbound communications only. An application user cannot initiate a chat.
- Resetting, logging in, and logging out of the chat channel also affect the email channel.

Sample scenarios

This object is used in the following scenarios:

- [Display channel properties scenario](#) on page 131
- [Enable and disable channel operational state scenario](#) on page 161

WorkList

Description

WorkList represents the structure that contains every [WorkItem](#) assigned to an application user. You can use WorkList to:

- Receive notification that a [WorkItem](#) was added to or removed from the work list of an application user.
- Get the [WorkItem](#) objects assigned to an application user.

Limitations

None.

Sample scenarios

This object is used in the following scenarios:

- [Display assigned work items scenario](#) on page 162
- [Prompt on WorkItem arrival scenario](#) on page 163
- [Workitem lifecycle scenario](#) on page 133

WorkItem

Description

WorkItem is the primary processing object for the Client SDK and performs the following tasks:

- Represents the work assigned to an application user without regard to the associated media, allowing a WorkItem to exist without active media.
- Groups together related [MediaInteraction](#) and [Document](#) objects.

You can use WorkItem to:

- Allow an application user to accept, collaborate on, and complete assigned work.
- Track the time spent by an application user on the Current WorkItem.

A collaboration occurs when an application user communicates with another party about an active WorkItem. When an application user starts a collaboration, the application must declare the intent of the collaboration: conference, consultation, or transfer.

A conference completes when all active parties are joined in the same media that was used to initiate the conference. A consultation keeps all participants separate and completes when the new party is released. A transfer is a blind handoff of a WorkItem to another party.

Limitations

The WorkItem object has the following limitations:

- Only one application user can own a WorkItem at any given time.
- Each WorkItem can contain either [VoiceMediaInteraction](#), [ChatMediaInteraction](#), [EmailDocument](#) or both VoiceMediaInteraction and ChatMediaInteraction as related media interactions in case of callback.
- A WorkItem must be associated with a [VoiceMediaInteraction](#), [ChatMediaInteraction](#), or [EmailDocument](#).
- For a conference or consultation, a WorkItem must be associated with an interactive media, such as [VoiceMediaInteraction](#) or [ChatMediaInteraction](#).

State definitions

The following table describes the state definitions included in the state model for WorkItem.

State	Definition
Initiating	An application user initiates a WorkItem. This state occurs when an application user makes an outbound phone call.
Nonviable	<p>The WorkItem is in an unusable state. Usually a WorkItem does not get delivered in this state or is only in this state temporarily.</p> <p>The following scenarios might cause a WorkItem to be delivered in the Nonviable state:</p> <ul style="list-style-type: none"> ● An application user has worked on a Session that contains a WorkItem with an active VoiceMediaInteraction, and that Session terminated abnormally through a process crash. The next time the application user logs in, the application user receives the WorkItem with the VoiceMediaInteraction in a Nonviable state. The application user can release the WorkItem. ● An application user receives a new WorkItem that contains a VoiceMediaInteraction. Auto-accept is not enabled. Therefore, the WorkItem is in the Nonviable state only until the state changes to Alerting. This scenario is usually caused by a time lag after the server receives the call from the Telephony server and before the server receives the attributes of the call from the EDU server. <p>Nonviable is an exception state. You will see a WorkItem in this state for an extended duration only if there is a problem. A WorkItem can be in this state for a transient period of time.</p>
Alerting	The WorkItem was delivered to an application user. The Avaya IC system is waiting for the application user to accept the WorkItem.
Paused	The Workitem was accepted but is not current. This state usually indicates that the application user is working on a different task.
Deferred	<p>The application user has postponed the WorkItem to be worked on later. A WorkItem in the Deferred state does not count against the current workload levels of an application user.</p> <p>This state is only valid for WorkItems that do not contain interactive media, such as email.</p>
Working	The WorkItem is active and gathering time. A WorkItem in the Working state counts against the current workload levels of an application user.
Transferring	The WorkItem is being processed as a blind transfer. Transferring is a transitional state.
Conferencing	The Avaya IC system is establishing a conference with a new party for the WorkItem. The conference includes the party that originated the WorkItem.

State	Definition
Consulting	The Avaya IC system is establishing a conference with a new party for the WorkItem. The conference will not include the party that originated the WorkItem.
Wrapup	The WorkItem was released. The Client SDK integration must call complete to exit this state and have the WorkItem complete.
Completed	Wrapup was completed. Completed is an internal Avaya IC state for cleanup of the WorkItem.

Sample scenarios

The WorkItem object is used in the following scenarios:

- [Session shutdown request scenario](#) on page 159
- [Display assigned work items scenario](#) on page 162
- [Prompt on WorkItem arrival scenario](#) on page 163
- [Prompt on WorkItem arrival scenario](#) on page 163
- [Enable and disable work item operational state scenario](#) on page 165
- [Workitem lifecycle scenario](#) on page 133
- [Workitem collaboration scenario](#) on page 136
- [Access work item attributes scenario](#) on page 165
- [Wrapup dialog box scenario](#) on page 172
- [Use terminate reasons scenario](#) on page 173
- [Display email scenario](#) on page 142
- [Display WorkItem History scenario](#) on page 146
- [Display Customer History scenario](#) on page 148

MediaInteraction

Description

MediaInteraction is a generic representation of real-time, interactive communication with one or more parties. A MediaInteraction must be associated with a [WorkItem](#).

MediaInteraction provides access to generic media events for states and operations. Capabilities and media-specific events are provided through the [VoiceMediaInteraction](#) and [ChatMediaInteraction](#) subclasses.

Chapter 2: Client API object model

For Avaya IC 7.2, the communication paths for MediaInteraction are voice and chat only.

Limitations

None.

Sample scenarios

This object is used in the following scenarios:

- [Display assigned work items scenario](#) on page 162
- [Prompt on WorkItem arrival scenario](#) on page 163
- [Enable and disable work item operational state scenario](#) on page 165
- [OnHold/OffHold indication scenario](#) on page 138

VoiceMediaInteraction

Description

VoiceMediaInteraction represents real-time work on the voice channel. VoiceMediaInteraction provides access to features and events that are specific to voice communication. You can use VoiceMediaInteraction to:

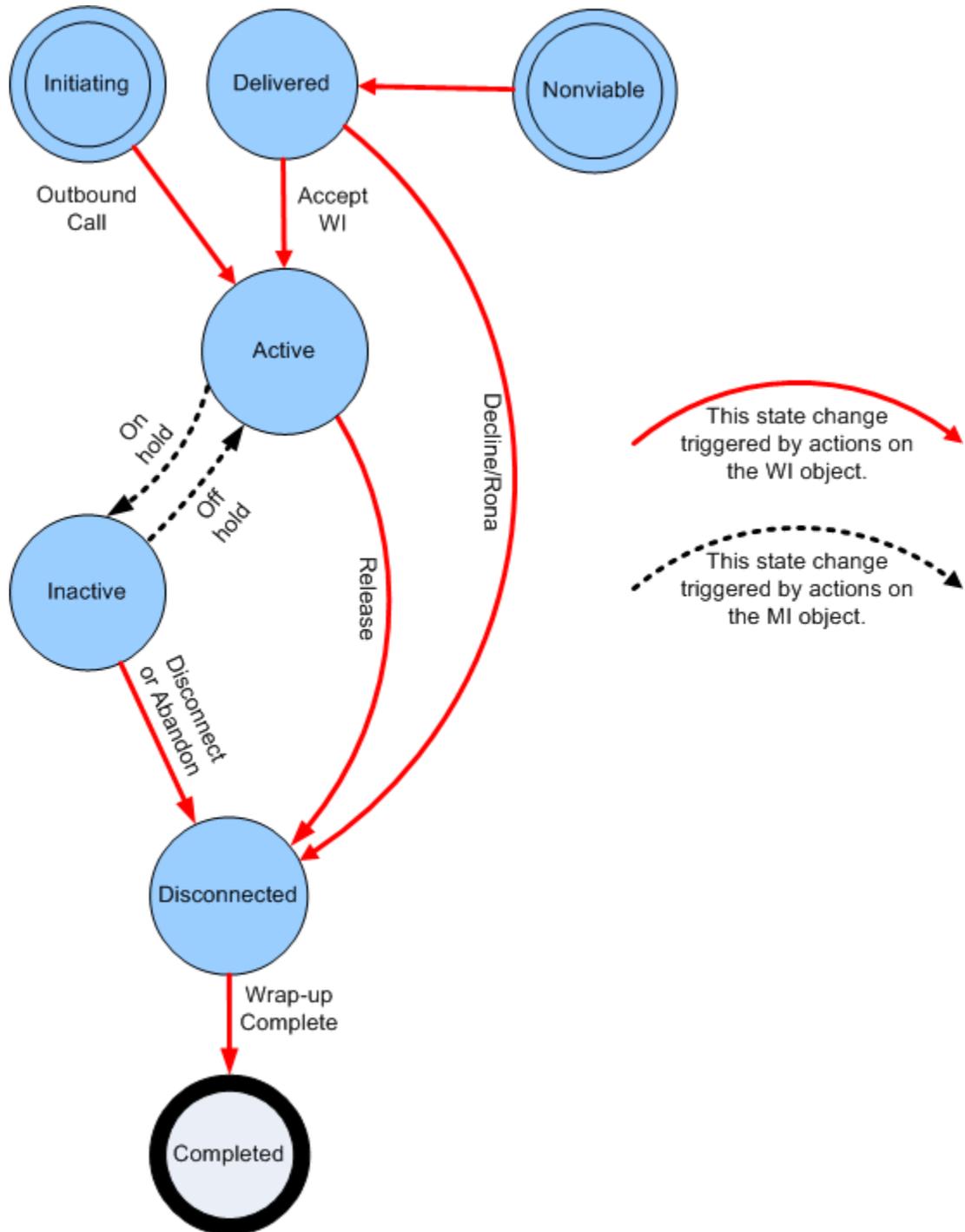
- Put a voice call on hold.
- Reconnect to a voice call.
- Send DTMF digits.
- Receive events that are specific to voice interaction processing.

Limitations

None.

VoiceMediaInteraction state model diagram

The following diagram shows the state model for VoiceMediaInteraction.



State definitions

The following table describes the state definitions included in the state model for VoiceMediaInteraction.

State	Definition
Initiating	An application user initiates a WorkItem. This state occurs when an application user makes an outbound phone call.
Nonviable	<p>The WorkItem associated with the VoiceMediaInteraction is in an unusable state. Usually a WorkItem does not get delivered in this state or is only in this state temporarily.</p> <p>The following scenarios might cause a WorkItem to be delivered in the Nonviable state:</p> <ul style="list-style-type: none"> ● An application user has worked on a Session that contains a WorkItem with an active VoiceMediaInteraction, and that Session terminated abnormally through a process crash. The next time the application user logs in, the application user receives the WorkItem with the VoiceMediaInteraction in a Nonviable state. The application user can release the WorkItem. ● An application user receives a new WorkItem that contains a VoiceMediaInteraction. Auto-accept is not enabled. Therefore, the WorkItem is in the Nonviable state only until the state changes to Alerting. This scenario is usually caused by a time lag after the server receives the call from the Telephony server and before the server receives the attributes of the call from the EDU server. <p>Nonviable is an exception state. You will see a WorkItem in this state for an extended duration only if there is a problem. A WorkItem can be in this state for a transient period of time.</p>
Delivered	The WorkItem associated with the VoiceMediaInteraction was delivered to an application user. The Avaya IC system is waiting for the application user to accept the WorkItem that contains the VoiceMediaInteraction.
Active	The VoiceMediaInteraction is active with the application user and the initial party.
Inactive	The call associated with the VoiceMediaInteraction is on hold. Usually, this state occurs because the application user needs to complete another activity, or initiate a transfer, conference or consult.
Disconnected	The call associated with the VoiceMediaInteraction was disconnected.
Completed	Completed is an internal Avaya IC state for cleanup of VoiceMediaInteractions.

Sample scenarios

This object is used in the following scenarios:

- [Workitem collaboration scenario](#) on page 136
- [OnHold/OffHold indication scenario](#) on page 138
- [OnHold alert on threshold scenario](#) on page 166

ChatMediaInteraction

Description

ChatMediaInteraction represents real-time work on the chat channel. ChatMediaInteraction provides access to features and events specific to chat communication. You can use ChatMediaInteraction to:

- Access the chat transcripts as documents associated with the [WorkItem](#).
- Receive events that are specific to chat interaction processing.
- Send text or URLs to chat participants.
- Retrieve callback information and using [WorkItem](#) object to place voice call back to customer.
- Create joinus handle for involving additional people in ongoing chat communication.
- Monitor chat interactions of agents.

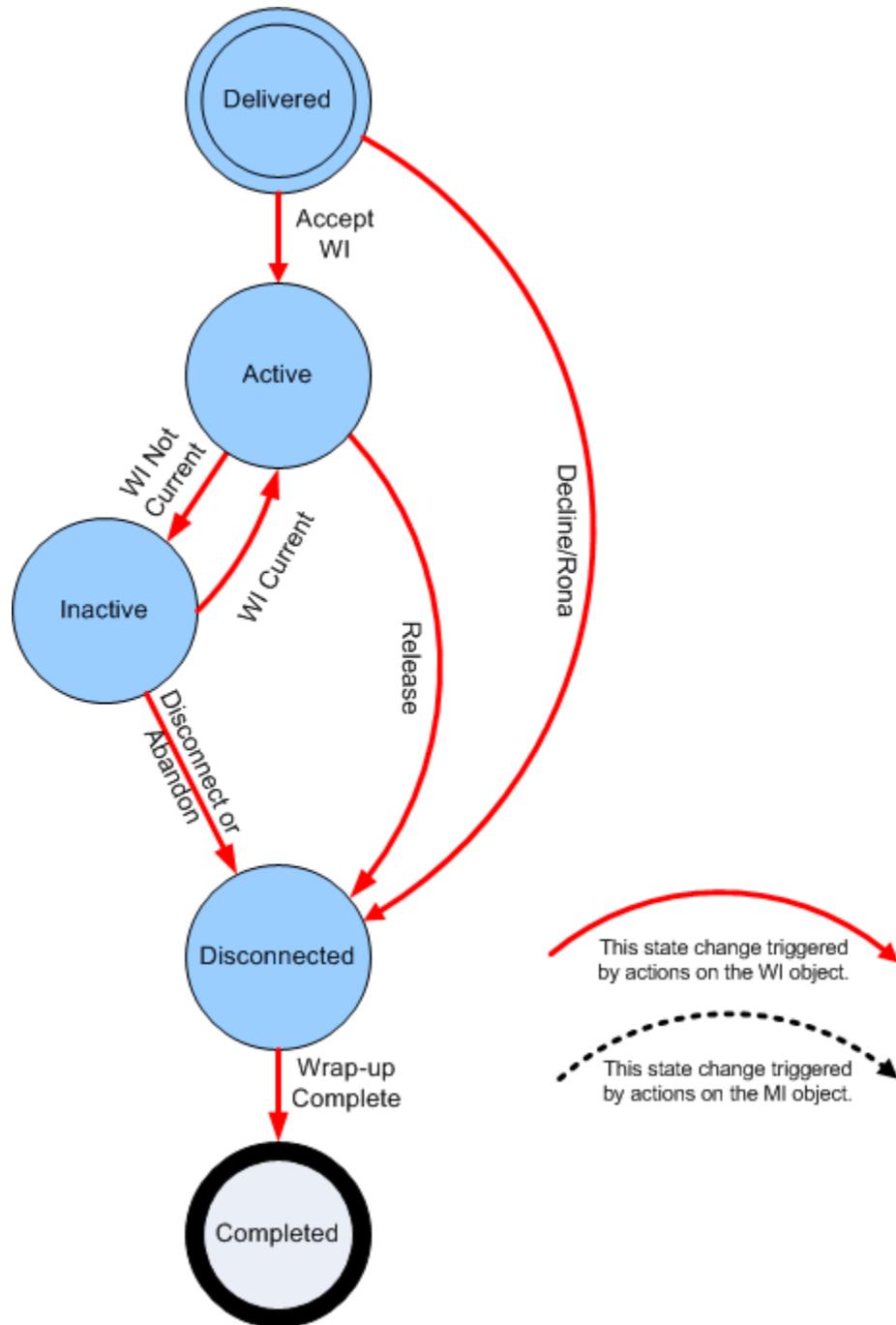
Limitations

ChatMediaInteraction has the following limitations:

- Outbound chat initiated by an application user is not supported.
- Datawake is not supported.

ChatMediaInteraction state model diagram

The following diagram shows the state model for ChatMediaInteraction.



State definitions

The following table describes the state definitions included in the state model for ChatMediaInteraction.

State	Definition
Delivered	The ChatMediaInteraction was delivered to an application user. The Avaya IC system is waiting for the application user to accept and activate the ChatMediaInteraction.
Active	The application user is in the chat room, actively handling the chat work item. The WorkItem associated with the ChatMediaInteraction is in a Working state.
Inactive	The application user is not actively handling the chat work item. The WorkItem associated with the ChatMediaInteraction is not in a Working state. Usually, this state occurs because the application user has switched to work on a different WorkItem.
Disconnected	The chat associated with the ChatMediaInteraction was disconnected.
Completed	Completed is an internal Avaya IC state for cleanup of ChatMediaInteractions.

Sample scenarios

This object is used in the following scenarios:

- [Display text message scenario](#) on page 139
- [Inactivity alert on threshold scenario](#) on page 167
- [Language filter scenario](#) on page 168
- [Customer-generated alert scenario](#) on page 169

Document

Description

Document is a container for noninteractive information. From the Document object, you can perform the following operations:

- Get the contents of the document.
- Get the WorkItem object associated with the document.
- Get the document type.
- Receive events on changes to the document.

A Document must be associated with a [WorkItem](#).

Limitations

None.

Sample scenarios

This object is used in the following scenario:

- [Enable and disable work item operational state scenario](#) on page 165

EmailDocument

Description

EmailDocument represents work on the email channel. You can use EmailDocument to:

- Deliver the contents of an email to the requesting custom application.
- Access the contents and attributes of the email, such as the header fields, language, body, and attachments.
- Create an [EmailDraft](#) related to the existing email, such as reply or forward.

Limitations

None.

Sample scenarios

This object is used in the following scenarios:

- [Display email scenario](#) on page 142
- [New Outbound email scenario](#) on page 143
- [Reply to email scenario](#) on page 144
- [Apply signatures scenario](#) on page 169
- [Support for attachments scenario](#) on page 170

DraftDocument

Description

DraftDocument is a generic representation of an outbound document. Drafts can be saved for later access by an application user. However, a draft document is private and cannot be shared with other application users.

The structure of DraftDocument is purposefully generic to enable future support of other delivery mechanisms.

Limitations

The `DraftDocument` object supports only [EmailDraft](#) in Avaya IC 7.2.

EmailDraft

Description

EmailDraft represents an outbound email. EmailDraft includes several types of drafts, based on the type of outbound message the draft will create. These types of drafts include:

- Reply
- Forward
- Reply Request
- Subject Matter Expert (SME) Request
- Request for additional information

Limitations

EmailDraft supports only one draft of each type at any given time for [EmailDocument](#). For example, an application user can have a draft reply, a draft forward to another application user, and a draft request for information from an external agent simultaneously. However, an application user cannot have two draft replies.

Sample scenarios

This object is used in the following scenarios:

- [Reply to email scenario](#) on page 144
- [Apply signatures scenario](#) on page 169

Attributes on the WorkItem object

The Client SDK provides a structure around attributes that improves the integration of the Client SDK with the Avaya IC system, IC Repository database, and with other systems. This structure is provided by an EDU container that holds all Client SDK attributes for a work item. You must use this EDU container to access and set data associated with a work item.

This section includes the following topics:

- [Attribute storage in the EDU](#) on page 50
- [WorkItem functions for attribute storage](#) on page 51
- [Data integration with a custom application](#) on page 51
- [Attribute data in the Contact table](#) on page 51
- [Example: moving data from a contact routing workflow to the database](#) on page 52

Attribute storage in the EDU

The EDU includes the `contact_attr.<name>` container that stores attributes for a work item.

The `contact_attr` container has only one entry for each named attribute. You can manipulate every element in the container with the `ContactAttribute` functions on the `WorkItem` object.

The `contact_attr` container can maintain one value for each attribute on a work item. If the container already has a value for an attribute, the Client SDK will overwrite the existing value with a new value.

WorkItem functions for attribute storage

The following table lists the functions on the WorkItem object that can access the contact_attr container in the EDU.

Function	Description
SetContactAttribute(NameValueList)	Takes a NameValueList object that contains the name value pairs to be inserted in the contact_attr container.
GetContactAttribute	Returns a NameValueList object containing all entries in the contact_attr container.

Data integration with a custom application

The Client SDK and a custom application perform most data lookups in the Avaya IC system through processes, such as workflows. However, a custom application must have access to and be able to manipulate, display, or reference data.

To write data for the Client SDK into the contact_attr container, a workflow or other Avaya IC process must prefix the key with contact_attr. Then, the WorkItem ContactAttribute functions can update and fetch the data.

Attribute data in the Contact table

When a user completes a work item, the Avaya IC system retires the associated EDU. The Report server then processes that EDU and saves the EDU data to the Contact table in IC Repository. By default, this EDU data does not include the contact_attr container. To add data from the contact_attr container into the database, you must create field expression records in the mapping rules structures for each EDU attribute you want to save.

Note:

Avaya recommends that you store the data in separate fields. Do not share a field with mixed elements.

To save attribute data in the Contact table of IC Repository, you must:

1. Add the required new fields to the Contact table in IC Repository.
2. Create new field expression records for the contact creation rule.

Example: moving data from a contact routing workflow to the database

This example shows how a custom application can:

- Obtain customer information from the database through a contact routing workflow.
- Return information to the database after the application user completes the work item.

Columns required in the Contact table

This example requires that you add the following columns to the Contact table with Database Designer:

- Contact fieldname
- account_no
- customer_value
- promised_payment
- promised_date

Field expression records required for the contact create rule

This example requires that you add the following field expression records for the contact create rule:

Create rule	Field name	Field value
Contact	account_no	contact_attr.account_no
Contact	customer_value	contact_attr.customer_value
Contact	promised_payment	contact_attr.promised_payment
Contact	promised_date	contact_attr.promised_date

How the data moves through the Avaya IC system

1. The contact routing workflow performs a database lookup on the customer for the following information:
 - account_no
 - customer_value
 - account_balance
2. The database lookup returns the following information to the workflow:

Field name	Value
account_no	1234567A
customer_value	Gold
account_balance	\$300

3. Because the custom application needs this information, the workflow writes the data to the contact_attr container of the EDU, as follows:

Container element	Value
contact_attr.account_no	1234567A
contact_attr.customer_value	Gold
contact_attr.account_balance	\$300

4. The workflow routes the work item to an application user through the Client SDK.
5. To use the information in the contact_attr container in a screen pop or other function, the custom application:
 - a. Calls `workitem.GetContactAttribute`.
 - b. Receives a `NameValueCollection` with the following data:

Name	Value
account_no	1234567A
customer_value	Gold
account_balance	\$300

6. During the conversation between the application user and the customer, the customer makes a promise to pay \$200 on March 13, 2006.

Chapter 2: Client API object model

7. To record this information for historical data, the custom application:
 - a. Calls `workitem.SetContactAttribute`.
 - b. Passes a `NameValueCollection` that contains:

Name	Value
promised_payment	\$200
promised_date	March 13, 2006

8. The Client SDK updates the `contact_attr` container of the EDU, as follows:

Container element	Value
<code>contact_attr.account_no</code>	1234567A
<code>contact_attr.customer_value</code>	Gold
<code>contact_attr.account_balance</code>	\$300
<code>contact_attr.promised_payment</code>	\$200
<code>contact_attr.promised_date</code>	March 13, 2006

9. After the processing is complete, the EDU retires to the Report server.
10. Because the columns already exist in the Contact table, and the rows already exist in the field expression table, the Report server automatically writes the data to the Contact table.

Chapter 3: Sample clients

The Client SDK includes two sample clients that you can use in a test or development environment to test functionality that can be accessed through the Client API.

This section includes the following topics:

- [Overview](#) on page 55
- [Features not supported in the sample clients](#) on page 56
- [About the Java sample client](#) on page 56
- [About the .NET sample client](#) on page 61

Overview

The Client SDK includes following sample clients:

- A sample client written in Java.
- A sample client written in C# for .NET.

The above sample clients are not designed for use in a production environment.



Important:

Avaya does not support the use of the sample clients in a production environment. Avaya will respond only to sample client issues in a test or development environment, if you use the sample client to test functionality that can be accessed through the Client API.

The user interfaces of the sample clients have a very different look and feel. They provide two examples of the types of custom applications that you can develop with the Client SDK.

You can access the source code of both sample clients. The source code is extensively commented to provide assistance with the Client API and to show examples of best practices. You can modify sections of the source code for use in your custom applications.

Features not supported in the sample clients

The sample clients do not support all of the features available with the Client SDK.

The following feature is not supported by either sample client:

- HTML Email

Features supported in the Java sample client but not supported in the .NET sample client:

- Login and logout of channels
- New Outbound email
- Push URL in the Chat window

About the Java sample client

The Java sample client is developed using Swing in Java. The Java sample client is packaged in JAR files.

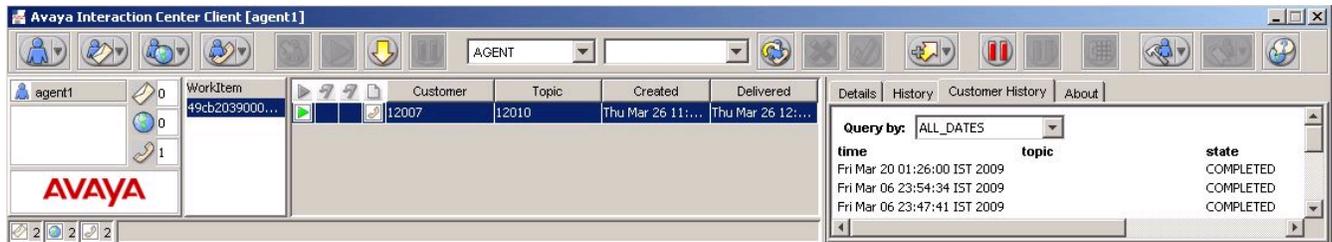
This section includes the following topics:

- [User interface of the Java sample client](#) on page 57
- [Source code for the Java sample client](#) on page 57
- [Dependencies of the Java sample client](#) on page 59
- [Developing a custom Java sample client](#) on page 59
- [Running a Java sample client from an SDK server system](#) on page 60
- [Running a Java sample client from a non-SDK server system](#) on page 61

User interface of the Java sample client

The interface of the Java sample client is based on the Avaya Agent Web Client interface. The Java sample client takes up minimal space on the desktop when the pop-up windows are closed. The pop-up windows include an Email application and a Chat application.

The following figure shows the Java sample client with an active outbound voice work item.



Source code for the Java sample client

You can find the source code for the Java sample client in the following directory:

`IC_INSTALL_DIR\IC72\sdk\design\java\sample\src`

The following table describes the files that contain the source code for the Java sample client.

Package name	Contents
<code>com.avaya.ic.sdk.sample</code>	Sample client class that is the entry point to the application
<code>com.avaya.ic.sdk.sample.ui</code>	PresentationSession that forms the glue between the model, view, and controller components
<code>com.avaya.ic.sdk.sample.ui.command</code>	Code that manages application user action commands
<code>com.avaya.ic.sdk.sample.ui.components</code>	Code for all the user interface and view components
<code>com.avaya.ic.sdk.sample.ui.event</code>	Classes that manage eventing between the view and controller components
<code>com.avaya.ic.sdk.sample.ui.model</code>	Classes that form the model for the more complex user interface elements, such as list
<code>com.avaya.ic.sdk.sample.ui.controllers</code>	Classes that provide a communication bridge between view components and the Client SDK.

Package name	Contents
com.avaya.ic.sdk.sample.ui.resources	Resource management classes
com.avaya.ic.sdk.sample.util	Some utility classes

Design configuration files for the Java sample client

You can find the design configuration files for the Java sample client in the following directory:

`IC_INSTALL_DIR\IC72\sdk\design\java\sample\config`

The following table describes the files that contain the design for the Java sample client.

File name	Contents
Application.properties	Layout information for application components, such as toolbars and lists.
ClientActions.properties	Information on all actions defined for the application. You can tie an action to multiple user interface elements, such as toolbar buttons, toolbar menus, and hot keys.
ClientInterface.properties	Information that ties property names to images used by the application.
ClientMessages.properties	Messages that the sample client displays in the user interface.

Resources for the Java sample client

The resources used by the Java sample client are images. All images for the Java sample client are installed in the following directory:

`IC_INSTALL_DIR\IC72\sdk\design\java\sample\images`

Dependencies of the Java sample client

The Java sample client depends on the following libraries that are installed with the Client SDK design files:

- avaya-common.jar
- avayaiccommon.jar
- common-base.jar
- avaya-ic-sdk-client.jar
- avaya-ic-sdk-common.jar
- commons-logging.jar
- commons-logging-api.jar
- log4j-1.2.8.jar
- mail.jar

These libraries are installed in the following directory:

```
IC_INSTALL_DIR\IC72\sdk\design\java\lib
```

Developing a custom Java sample client

To develop a custom Java sample client using the Eclipse application:

1. From the system where you configured the SDK server, copy the \\ <Avaya_IC72_HOME>\sdk\design\Java directory.
2. Start the Eclipse application.
3. In Eclipse, create a new Java project and create a new Java project:
 - Project name
 - Location
4. Right-click the new Java project and select **Configure Build Path**.
5. In the **Properties for Custom Client** dialog box, click **Libraries**.
6. Click **Add External JARs**.
7. From the `.lib` directory, select and add all the `.jar` files.
8. In the navigation pane, click **Java Compiler**.
9. In the right pane, select the **Enable project specific settings** check box.
10. In the **Click compliance settings** field, click JRE version 1.6.

11. Click **OK**. The Eclipse application rebuilds the complete project.
12. Right-click **Project** and select **Import File System**.
13. Click **Next**.
14. In the **File system** dialog box, click **Browse** and select the `Java/sample/images` directory.
15. Click **Finish**.
16. Right-click **Project** and select **Import File System**.
17. Click **Next**.
18. In the **File system** dialog box, click **Browse** and select the `Java/sample/config` directory.
19. Click **Finish**.
20. Right-click **Project** and navigate to `com.avaya.ic.sdk.sample`.
21. Right-click **Sampleclient.java** and select **Run as Java application**.
22. In the **Log in** dialog box, enter the following:
 - User Name
 - Password
 - SDK Server URL
23. Click **Log in**.
24. Verify that the sample client starts and you can log in from the Java sample client.

Running a Java sample client from an SDK server system

1. Navigate to the `IC_INSTALL_DIR\IC72\sdk\design\Java\sample\bin` directory.
2. Double-click the `RunClient.bat` file.
3. In the **Login** dialog box, enter the following:
 - User Name
 - Password
 - Server URL. For example, `http://<SDK Server name or IP address>:9700/icsdk`.
4. Click **Log in**. You are logged in the Avaya IC system from the Java sample client.

Running a Java sample client from a non-SDK server system

1. From the system where you configured the SDK server, copy the `IC_INSTALL_DIR\IC72\sdk\design\Java` directory to the system where you want to run the Java sample client.
2. From the `\Java` directory, open the `RunClient.bat` file and replace the value for `DIR` with `<JRE_HOME>\bin`. Save the `RunClient.bat` file.
3. In the `\Java` directory, double-click the `RunClient.bat` file.
4. In the **Login** dialog box, enter the following:
 - User Name
 - Password
 - Server URL. For example, `http://<SDK Server name or IP address>:9700/icsdk`.
5. Click **Log in**. You are logged in the Avaya IC system from the Java sample client.

About the .NET sample client

The .NET sample client is written in C#. The .NET sample client is packaged in assembly and DLL files.

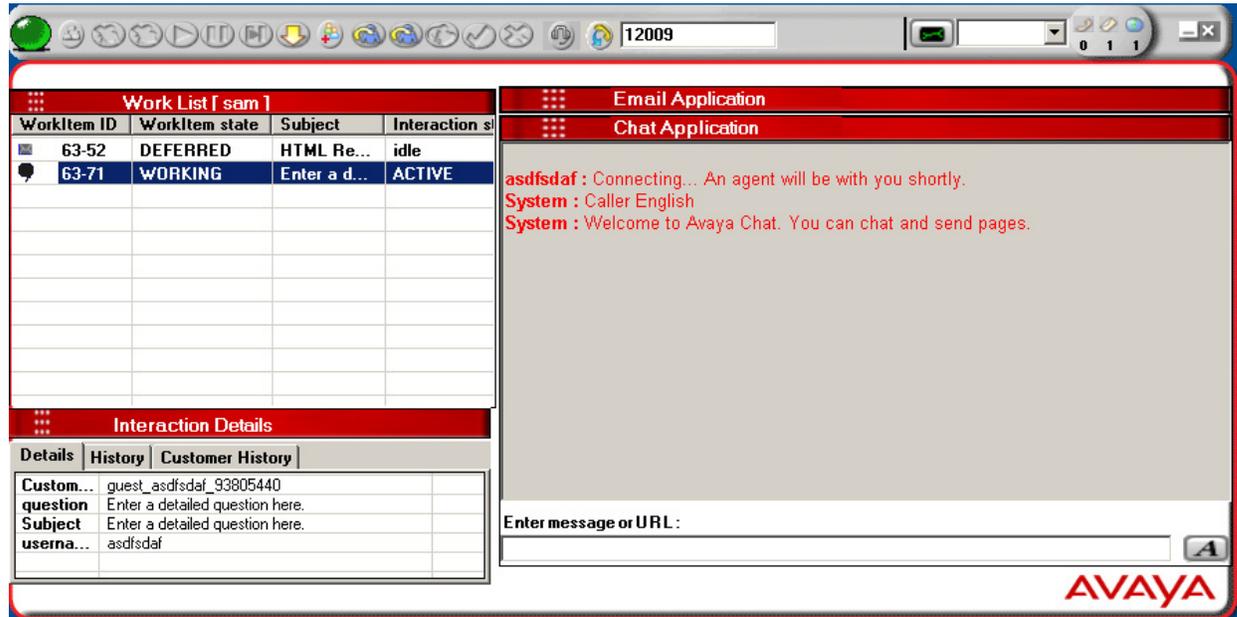
This section includes the following topics:

- [User interface of the .NET sample client](#) on page 62
- [Code and resources for the .NET sample client](#) on page 62
- [Dependencies of the .NET sample client](#) on page 63
- [Developing a custom .NET sample client](#) on page 63
- [Running a .NET sample client from an SDK server system](#) on page 64
- [Running a .NET sample client from a non-SDK server system](#) on page 65

User interface of the .NET sample client

The .NET sample client can be positioned anywhere on the agent desktop. All components of the application including the work list opens within the application.

The following figure shows the .NET sample client.



Code and resources for the .NET sample client

The following topics describe the files that contain the code and resources for the .NET sample client:

- [Com.Avaya.Ic.Sdk.Sampleclient.Ui](#) on page 62
- [Com.Avaya.Ic.Sdk.SampleClient.Controller](#) on page 63

Com.Avaya.Ic.Sdk.Sampleclient.Ui

Description: Contains the forms and the supporting code for the forms. All images are embedded in the forms.

Installation location: The Com.Avaya.Ic.Sdk.Sampleclient.Ui files are installed in the following directory:

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\  
Com.Avaya.Ic.Sdk.Sampleclient\Ui
```

Com.Avaya.Ic.Sdk.SampleClient.Controller

Description: Contains the Controller classes. The Controller classes do the following:

- Contain all of the Client SDK related code.
- Maintain a reference to the view.
- Update the view when events are received from the Client API.

Installation location: The Com.Avaya.Ic.Sdk.SampleClient.Controller files are installed in the following directory:

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\  
Com.Avaya.Ic.Sdk.Sampleclient\Controller
```

Dependencies of the .NET sample client

The .NET sample client depends on the following DLLs that are installed with the Client SDK design files:

- Com.Avaya.Util.dll
- Com.Avaya.Util.Messaging.dll
- ICSharpCode.SharpZipLib.dll
- log4net.dll
- AvayaICSDKClient.dll

These libraries are installed in the following directory:

```
IC_INSTALL_DIR\IC72\sdk\dotnet\sample\bin
```

Developing a custom .NET sample client

To develop a custom .NET sample client using Microsoft Visual Studio:

1. From the system where you configured the SDK server, copy the \\<AVAYA_IC72_HOME>\sdk\design\dotnet directory.
2. Navigate to the \\<AVAYA_IC72_HOME>\sdk\design\dotnet\sample\src directory.
3. Open the SampleClient.solution directory in Microsoft Visual Studio.
4. In the left navigation pane, right-click **Reference** and select **Add Reference**.
5. In the **Add Reference** dialog box, click **Browse**.

6. In the **Look in** field, select the directory where you copied the `\\<AVAYA_IC72_HOME>\sdk\design\dotnet` directory.
7. Navigate to the `\\<AVAYA_IC72_HOME>\sdk\design\dotnet\lib` directory and select all the `.dll` files and click **OK**.
8. To start the sample client in release mode:
 1. In Microsoft Visual Studio, from the drop-down list, select **Release**.
 2. Click **Start Debugging**. The system displays the dialog box with a message indicating that the you are running the .NET sample client in the release mode.
 3. Click **OK**.
9. To start the sample client in the Debug mode:
 1. In Microsoft Visual Studio, from the drop-down list, select **Debug**.
 2. To debug a specific area of the .NET sample client code, enter the debug point in the code and then run the sample client.
 3. In the **Login** dialog box, enter the following:
 - Login
 - Password
 - SDK server URL. For example, `http://<SDK_Server>:9700/icsdk`.
10. Click **Login**. The system displays the .NET sample client interface.

Running a .NET sample client from an SDK server system

1. Navigate to the `\\IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\bin` directory.
2. Double-click the `RunClient.bet` file.
3. In the **Login** dialog box, enter the following:
 - User Name
 - Password
 - Server URL. For example, `http://<SDK Server Name or IP address>:9700/icsdk`.
4. Click **Login**. You are logged in the Avaya IC system through a .NET sample client.

Running a .NET sample client from a non-SDK server system

Important:

Ensure that you have .NET 3.5 or later on the non-SDK system.

1. From the system where you configured the SDK server, copy the `IC_INSTALL_DIR\IC72\sdk\design\dotnet` directory to the system where you want to run the .NET sample client.
2. In the `\dotnet` directory, double-click the `RunClient.bat` file.
3. In the **Login** dialog box, enter the following:
 - User Name
 - Password
 - Server URL. For example, `http://<SDK Server name or IP address>:9700/icsdk`.
4. Click **Log in**. You are logged in the Avaya IC system from the .NET sample client.

Chapter 4: Guidelines for using the Client API

Avaya recommends that you consider the following guidelines when you develop a custom application or integration with the Client API.

This section includes the following topics:

- [Chat interaction guidelines](#) on page 68
- [Voice interaction guidelines](#) on page 69
- [Callback guidelines](#) on page 70
- [Time and date duration guideline](#) on page 72
- [History API guidelines](#) on page 73
- [AddressBook API guidelines](#) on page 76
- [WrapupSelection API guideline](#) on page 78
- [Event handling guidelines](#) on page 79
- [State guidelines](#) on page 85
- [Log in and log out guidelines](#) on page 86
- [Exception handling guidelines](#) on page 87
- [Operation failure and success guidelines](#) on page 88
- [Null return value guidelines](#) on page 89
- [Customization guidelines](#) on page 90
- [Performance considerations](#) on page 93



Important:

The code provided in this section is for example only. Last minute changes to the Client SDK may result in differences between the samples in this section and the actual code in the Client SDK. Before you implement any of these guidelines, review the Client API documentation and code in the Client SDK.

Chat interaction guidelines

Avaya recommends that you should be aware of the following guidelines when you develop your custom code:

- [Methods not allowed for chat interactions](#) on page 68
- [Handle TranscriptLine events for chat interactions](#) on page 68
- [Do not use Datawake method](#) on page 69

Methods not allowed for chat interactions

The following methods are not allowed for chat interactions:

- `collaborationComplete` for conference and consult
- `collaborationCancel` for conference and consult

Handle TranscriptLine events for chat interactions

If your custom application includes chat, ensure your code:

- Monitors the application user and customer idle time.
- Generates idleness and other alerts as needed by the custom application.

The Client SDK provides an event stream that you can use to monitor idle times. However, your custom code must include timers to monitor the events against the threshold settings to ensure that alerts are generated.

Reason

Idleness alerts ensure that application user in a chat room is alerted if the participants are waiting for a response to a message.

For example, an application user handles two chats with two different customers. After sending a message to the first customer, the application user starts a chat with the second customer. If the first customer responds, the application user needs to be alerted that a message has arrived in the first chat room. If the application user does not respond within the specified time, your custom application generates an alert. The alert notifies the application user that the customer is still waiting for a response.

Coding information

When a message is sent or received by an application user, the Client SDK:

1. Creates a TranscriptLine object.
2. Provides the TranscriptLine object to the consumer through a TRANSCRIPT_LINE_ADDED event on the TranscriptDocument.

The TranscriptLine object represents a message sent into the chat room. TranscriptLine includes the OriginatorHandle property that provides the originator of the message. This originator can be agent, caller, system, or unknown.

Use these objects and the OriginatorHandle property when you code the alerts in your custom application. You can create, start, and reset timers based on the context carried in those objects.

For more information about TranscriptLine, see the Client API documentation.

Do not use Datawake method

The Client SDK does not support Datawake.

Do not use the `getDatawakeURL()` method in your custom application. That method will return a NULL.

Voice interaction guidelines

Avaya recommends that you should be aware of the following guidelines when you develop your custom code:

- [Using VoiceChannel.Reset](#) on page 69
- [Setting the force multiple calls option on the switch](#) on page 70
- [Impact of network recovery on voice interaction](#) on page 70

Using VoiceChannel.Reset

VoiceChannel.Reset does not hang up any telephone calls. The API is supposed to terminate all the EDUs associated with the softphone and set the agent into "ready" or "notready" state depending on the API call.

Only use VoiceChannel.Reset if the softphone state is out of sync with the hardphone operation. Do not use this API to hang up a telephone call.

Setting the force multiple calls option on the switch

If an application user accepts a voice call and is made busy from the softphone, the user can still receive a second voice call. The switch is unaware that the user is busy and may send a second call. This problem occurs only when the force multiple calls option is enabled on the hunt group where the user has logged in.

The task load and ceilings set on Avaya IC for the voice channel are not enforced on the voice channel.

Impact of network recovery on voice interaction

If a network recovery that requires an application user to log in back occurs when an application user is handling a voice interaction on the hardphone, the voice channel will not connect to the Telephony server. The user will not have access to any softphone controls, because the softphone is out of sync.

Reason

The voice channel does not automatically connect to the Telephony server because this connection can drop the current call on the hardphone.

User expectations

Under these circumstances, the Client SDK expects the application user to perform the following steps *in sequence*:

1. End the call on the hardphone.
2. Wrapup and complete the contact that corresponds to the EDU associated with the call.

The voice channel should automatically log in as soon as all hardphone contacts have been wrapped and cleared from the task list.

Callback guidelines

Chat and Callback feature allows the customer to specify the callback number while initiating a chat from IC website. On receiving the workitem, agent can retrieve the callback number from `ChatMediaInteraction` object and can place an outbound call to the customer. In this case, the voice call gets associated with chat interaction and is grouped together in the same workitem.

This section will detail out the implementation considerations, guidelines and limitations of callback feature.

Note:

From the feature implementation perspective the Chat and Callback, Web Schedule Callback, and Chat and VoIP are same.

Implementation considerations

1. In order to associate the Chat and Voice media interactions:
 - Report Server must be present either in the Web domain or in its failover domain.
 - A EDU Server must be present either in the domain of the Report server or in its failover domain.
2. For a workitem with chat and voice media interactions grouped together, SDK treats chat as **parent** media interaction and voice as **child** media interaction.
3. For collaborating a multimedia interaction, agent must be searched using the agent ID or the address book and not using the station ID.

Guidelines for implementing chat and callback

This scenario requires the following steps:

1. Register listeners for the `WorkItemAdded` and `WorkItemRemoved` events.
2. Register listeners for the `MediaInteractionAdded` and `MediaInteractionRemoved` events.
3. Create handlers to receive above events.
4. Update the user interface to display workitem and media interaction separately.
5. Update the user interface to display callback number after the chat media interaction is received.
6. Provide a mechanism to place a voice callback on the `Workitem` object.

Guidelines for collaborating chat and callback

As chat and voice media interactions are grouped together in a single workitem, you need to ensure that the collaborated agent receive both the interactions as a single workitem.

Following are the high level steps:

1. Register listeners for `AudioSourceChanged` and `PartyAdded` events.
2. Provide handlers for these events.
3. Conference or consult voice media interaction by invoking `collaborationBegin()` API on a `Workitem` object, with `VoiceMediaInteraction` as a parameter.
4. On receiving `AudioSourceChanged` event, collaborate Chat media interaction by invoking `collaborationBegin()` API on a `Workitem` object.

5. On receiving `PartyAdded` event, invoke `collaborationComplete()` API on a `Workitem` object.

Guidelines for completing chat and callback

As mentioned earlier, for a workitem with chat and voice media interactions grouped together, SDK treats chat as **parent** media interaction and voice as **child** media interaction. Therefore invoking the `complete()` API on a workitem removes both chat and voice interactions.

If only voice interaction needs to be removed, invoke `complete()` API on the `VoiceMediaInteraction` object.

Collect the wrapup codes when the parent media interaction is removed.

Limitations

Collaborating of chat and callback interaction to a queue is not supported. Doing so will raise `OperationFailed` event.

Time and date duration guideline

The SDK server does not calculate specific durations. The Client SDK provide APIs that allow a custom application to perform these calculations if needed.

If your custom application requires the calculation of a specific duration, use the create date value or the delivered date value to do the required calculation. For example, your custom code can calculate the age of a particular work item in milliseconds by subtracting the current time in ms from the create date value.

Use the following APIs in your calculations:

- `WorkItem.getDeliveredDate()`
- `WorkItem.getCreateDate()`
- `WorkItem.getQueueTime()`
- `Document.getCreateDate()`

History API guidelines

Avaya recommends that you should be aware of these guidelines when you use the History API in your custom code:

- [Retrieving the history for a WorkItem or Customer](#) on page 73
- [WorkItemHistory record](#) on page 74
- [CustomerHistory record](#) on page 74
- [Formatting dates for the history of an object](#) on page 75
- [Example: retrieve WorkItem history for Java application](#) on page 75
- [Example: retrieve Customer history for Java application](#) on page 76

Retrieving the history for a WorkItem or Customer

When a WorkItem arrives at a custom application, the Client SDK does not automatically send the WorkItemHistory or the CustomerHistory for that WorkItem. This process ensures that the Client SDK can deliver the WorkItem to the application user as quickly as possible, and avoid the time-consuming look ups required for WorkItemHistory and CustomerHistory.

The Client SDK is optimized to allow your custom application to retrieve the WorkItemHistory or CustomerHistory. The retrieval of the most recent WorkItemHistory and CustomerHistory is asynchronous. When the history data arrives at the custom application, the Client SDK automatically caches the data on the client.

For the CustomerHistory, your custom application must specifically request every piece of information that you want to display to the user.

When you retrieve the history, always follow this two-step process:

1. Request the history.
2. Retrieve the history data.

WorkItemHistory record

The Client SDK obtains information for a WorkItemHistory record from the EDU for a VoiceMediaInteraction and a ChatMediaInteraction.

For an EmailDocument, the WorkItemHistory record is the same as the TrackingHistory. The Client SDK server retrieves that information from the IC Email server, which retrieves the information from the database. This information does not include the time spent by an email in queue.

Each WorkItemHistory record has a type of NameValueList. Each field in the record has a type of NameValue. By default, each WorkItemHistory record contains the following fields:

- starttime
- origin
- description

CustomerHistory record

Your custom application can potentially retrieve more information for a CustomerHistory record. Each CustomerHistory record corresponds to a previous interaction that the customer had with the contact center. This interaction can be a single historical MediaInteraction or a single historical EmailMessage exchanged between the customer and the contact center.

Data provided with a CustomerHistory record: In addition to the data provided with a WorkItemHistory, the CustomerHistory can also include:

- WrapupRecords for all types of WorkItems
- WorkItemHistory for all types of WorkItems
- Transcripts for ChatMediaInteractions
- EmailMessages for EmailDocuments

Fields in a CustomerHistory record: Each CustomerHistory record has a type of NameValueList. Each field in the record has a type of NameValue. By default, each CustomerHistory record contains the following fields:

- type
- state
- topic
- time

Rules for database queries: The CustomerHistory record fields represent the summary information available for the CustomerHistory record. This summary information is obtained through a database query. The query criteria must obey the following rules:

- If a customerkey is present in the EDU for the MediaInteraction or Document, the query must use the customerkey to perform the customer history query.
- If no customerkey is present in the EDU, the query must use the following query criteria:
 - ANI for VoiceMediaInteractions
 - Email Address for EmailDocuments
 - Chat Handle for ChatMediaInteractions

Formatting dates for the history of an object

The Client SDK does not provide a formatted string for the date. The History API returns a string that contains a millisecond value. For Java, the millisecond value is from January 1, 1970. For .NET, the millisecond value is from January 1, 1900.

Your custom application needs to apply a locale specific pattern when parsing the date string.

To use the date format in your custom application:

1. Convert the millisecond value to a Long.
2. Add the Long to a date object.
3. Retrieve the formatted string from the date object.

Example: retrieve WorkItem history for Java application

Note:

This example uses the API names for the Java API of the Client SDK. A .NET application should follow the same steps. For information about the corresponding API calls for .NET, see the .NET API documentation provided with the Client SDK.

For more information, see [Display WorkItem History scenario](#) on page 146.

To retrieve the history for a WorkItem requires the following steps:

1. Call `workitem.requestHistory` to request the WorkItem history.
2. Wait until the `Workitem.RequestHistoryResponse` event is received.
 - This event indicates that the WorkItem history is received and available in the HDS.
3. Retrieve the WorkItem object from the `Workitem.RequestHistoryResponse` event.
4. Call `workitem.getHistory` to retrieve the history data.

Example: retrieve Customer history for Java application

Note:

This example uses the API names for the Java API of the Client SDK. A .NET application should follow the same steps. For information about the corresponding API calls for .NET, see the .NET API documentation provided with the Client SDK.

For more information, see [Display Customer History scenario](#) on page 148.

To retrieve the history for a Customer requires the following steps:

1. Call `Workitem.requestCustomerHistory` with the appropriate search criteria to request the customer history.
2. Wait until the `Workitem.RequestCustomerHistoryResponse` event is received. This event indicates that the customer history is received and available in the HDS.
3. Retrieve the `WorkItem` object from the `Workitem.RequestCustomerHistoryResponse` event.
4. Call `Workitem.getCustomerHistory` to retrieve the customer history data.

AddressBook API guidelines

Avaya recommends that you should be aware of the following guidelines when you develop your custom code:

- [Retrieving the AddressBook object](#) on page 76
- [Implementing Address Book searches](#) on page 77
- [Example: Finding a subset of agents based on criteria](#) on page 77
- [Example: Finding a subset of queues based on criteria](#) on page 78

Retrieving the AddressBook object

When an application user initiates an Avaya IC session, the system does not send AddressBook data to the application by default. However, you can use AddressBook APIs in the Client API to retrieve agents and queues on the AddressBook object.

Use the `Session.getAddressBook()` API to retrieve the AddressBook object. The AddressBook object contains all of the addressable agents and queues in the Avaya IC system.

Implementing Address Book searches

You can implement your custom application to do either of the following AddressBook searches:

- Perform a default search for AddressBook data with the `findAgents` or `findQueues` methods.
- Collect query information from the application users and use that information in your AddressBook search.

These searches return either a `FindAgentsResponse` or `FindQueuesResponse` event that contains the relevant records.

**Tip:**

You can use search criteria to specify and retrieve a subset of agents or queues for the Address Book, instead of all agents or queues in the Avaya IC system. For information about the search criteria, see the Client API documentation.

Example: Finding a subset of agents based on criteria

Note:

This example uses the API names for the Java API of the Client SDK. A .NET application should follow the same steps. For information about the corresponding API calls for .NET, see the .NET API documentation provided with the Client SDK.

For an example scenario with more detailed information, see [AddressBook scenario](#) on page 150.

To retrieve a subset of agents based on criteria requires the following steps:

1. Call `Session.getAddressBook` to retrieve the AddressBook object.
2. Call `AddressBook.findAgents` with the appropriate criteria to request the subset of agents.
3. Wait until the `AddressBook.FindAgentsResponse` is received.
This event indicates that the agent subset is received and available in the HDS.
4. Call `AddressBook.FindAgentsResponse.getAgentRecords` to retrieve the subset of agents.

Example: Finding a subset of queues based on criteria

Note:

This example uses the API names for the Java API of the Client SDK. A .NET application should follow the same steps. For information about the corresponding API calls for .NET, see the .NET API documentation provided with the Client SDK.

For an example scenario with more detailed information, see [AddressBook scenario](#) on page 150.

To retrieve a subset of queues based on criteria requires the following steps:

1. Call `Session.getAddressBook` to retrieve the `AddressBook` object.
2. Call `AddressBook.findQueues` with the appropriate criteria to request the subset of queues.
3. Wait until the `AddressBook.FindQueuesResponse` is received.
This event indicates that the queue subset is received and available in the HDS.
4. Call `AddressBook.FindQueuesResponse.getQueueRecords` to retrieve the subset of queues.

WrapupSelection API guideline

The `WrapupSelectionList` represents a list of `WrapupSelections` that has been set on a `WorkItem`.

If you use the `WrapupSelection` API do not expect the `WrapupSelectionList` method `addWrapupSelection` to add the wrapup selection to the list. `addWriapup Selection` adds the `WrapupSelection` to a cache.

Recommended sample code

```
public void foo{
    ...
    WrapupSelectionList list = wi.getWrapupSelectionList();
    list.addWrapupSelection(new WrapupSelectionImpl(categoryCode, reasonCode, outcomeCode);
    wi.setWrapupSelectionList(list); //add the cached wrapupselection to the list. This call
    sends an event to the server. You now need to wait for OperationSucceeded event before
    calling getWrapupSelections to see if the wrapup selection was added.
}
public static void WorkItemOperationSucceededListener implements SessionListener{
    public void onEvent(Event evt){
        WorkItem.OperationSucceeded success = (WorkItem.OperationSucceeded) evt;
        if(success.getOperationName() ==
WorkItem.Operation.SETWRAPUPSELECTIONLIST.toString()){
            wi.getWrapupSelectionList().getWrapupSelections(); //this will contain the newly
added wrapup selection.
        }
    }
}
```

Sample of code to avoid



Important:

Do not follow the practice shown in the below sample code when you create listeners.

```
public void foo(){
    ....
    WrapupSelectionList list = wi.getWrapupSelectionList();
    list.addWrapupSelection(new WrapupSelectionImpl(categoryCode, reasonCode, outcomeCode);
    list.getWrapupSelections(); //returned collection will not have
    ...
}
```

Event handling guidelines

Avaya recommends that you consider the following guidelines when you develop event handling in your custom code:

- [Register listeners for events before calling Session.Initialize](#) on page 80
- [Create a separate listener for each event](#) on page 80
- [Avoid blocking operations in event handling](#) on page 83
- [Handle ConnectionStatusChange and SessionShutdown events](#) on page 83

Register listeners for events before calling `Session.Initialize`

Register all or most of your listeners between the call to `Application.Login` and `Session.Initialize`.

Reason

If you do not register listeners before `Session.Initialize`, timing reasons can cause events to arrive before the listeners are registered. An event that arrives before registration of the appropriate listener might be lost.

Recommended sample code

.NET sample code

```
...
ISession _session = _app.Login(user, password);
_session.SessionStateChanged += new SessionStateChangedHandler(HandleSessionStateChanged);
..
..
_session.Initialize();
```

Java sample code

```
Session _session = application.login(user, pass);
_session.registerListener(Session.StateChanged.TYPE, new SessionStateChangedListener());
...
...
_session.initialize();
```

Create a separate listener for each event

Do not use the same listener to handle more than one event.

Note:
.NET enforces this behavior automatically.

Reason

The Client API does not prevent you from using the same listener to handle more than one event. However, if you use the same listener for multiple events, your code must repeatedly instruct the Client API to perform the internal check for which listener to call. This practice results in inefficient, spaghetti handler code.

Recommended sample code

Java sample code

```
//Registration code
_session.registerListener(Session.StateChanged.TYPE, new SessionStateChangedListener());
_session.registerListener(WorkList.WorkItemAdded.TYPE, new WorkItemAddedListener());
_session.registerListener(Channel.StateChanged.TYPE, new ChannelStateChangedListener());

//Handler code
private static class SessionStateChangedListener implements SessionListener{
public void onEvent(Event evt){

//handle the event here
}
}
private static class WorkItemAddedListener implements SessionListener{
public void onEvent(Event evt){

//handle the event here
}
}
private static class ChannelStateChangedListener implements SessionListener{
public void onEvent(Event evt){

//handle the event here
}
}
}
```

.NET sample code

```
_session.SessionStateChanged += new SessionStateChangedHandler(HandleSessionStateChanged);
_session.WorkListWorkItemAdded += new WorkListWorkItemAddedHandler(HandleWorkItemAdded);
_session.ChannelStateChanged += new ChannelStateChangedHandler(HandleChannelStateChange);

//Handler code
private void HandleSessionStateChanged(SessionStateChangedEventArgs earg)

{

}

private void HandleWorkItemAdded(WorkListWorkItemAddedEventArgs earg)

{

}

private void HandleChannelStateChange(ChannelStateChangedEventArgs earg)

{

}
```

Sample of code to avoid



Important:

Do not follow the practice shown in the below sample code when you create listeners.

```
//Registration code
_listener = new AllEventListener();

_session.registerListener(Session.StateChanged.TYPE, _listener);
_session.registerListener(WorkList.WorkItemAdded.TYPE, _listener);
_session.registerListener(Channel.StateChanged.TYPE, _listener);

//Handler code

private static class AllEventListener implements EventListener{
public void onEvent(Event evt){
    if(evt instanceof SessionStateChanged){
        handleSessionStateChanged();
    } else if(evt instanceof WorkListWorkItemAdded){
        handleWorkItemAdded();
    } else if(evt instanceof ChannelStateChanged){
        handleChannelStateChanged();
    }
}
}
```

Avoid blocking operations in event handling

Do not use blocking operations or operations that require a significant amount of processing in the event handlers. Examples of such operations are:

- Painting application windows
- Implementing I/O
- Playing media
- Writing a loop that loops for a long time

Reason

All events from the Client API arrive on a single thread that processes all inbound messages from the server. For server-side integration, a blocking operation or an operation that requires a significant amount of processing can cause message queues to back up and impact performance.

Handle `ConnectionStatusChange` and `SessionShutdown` events

Do not use Client API objects in your code after a `ConnectionStatusChange` event is received and the `ConnectionStatus` is equal to `FAILED`. Always use a logout operation after that event.

Reason

After a custom application receives a `SessionShutdown` event or the `ConnectionStatus` of the `Session` changes to `Failed`, the `Session` object and all other objects that belong to that session become unusable. A logout is the only valid operation after these events.

All custom code that uses the Client API must register for these events and gracefully log out of the session after the events occur.

If your custom code uses Client API objects after a `Connection` failure event or a `SessionShutdown` event, the `Communication` component will usually throw an error. The Client SDK throws a `CommunicationException` when the connection is impaired or disabled. For more information about this custom exception, see the API documentation.

These errors are runtime exceptions in Java. The Client API does not expect application users to handle these exceptions. Therefore, your custom code must handle these events properly to provide an acceptable application user experience.

Recommended sample code

Java sample code

```
//Registration code
_session.registerListener(Session.Shutdown.TYPE, new SessionShutdownListener());
_session.registerListener(Session.ConnectionStatusChange.TYPE, new
ConnectionStatusChangeListener());

//Handler code
private static class SessionShutdownListener implements SessionListener{
public void onEvent(Event evt){
    //Check the reason for shutdown and display message to user (or) do additional handling

    //at the end just logout the user
    _application.logout(_session.getUser().getLoginId());
}
}

private static class ConnectionStatusChangeListener implements SessionListener{
public void onEvent(Event evt){
    Session.ConnectionStatusChange csc = (Session.ConnectionStatusChange) evt;
    if(csc.getConnectionStatus() == ConnectionStatus.FAILED){
        _application.logout(_session.getUser().getLoginId());
    }
}
}
```

.NET sample code

```

_session.SessionShutdown += new SessionShutDownHandler(HandleSessionShutDown);
_session.SessionConnectionStatusChanged += new
SessionConnectionStatusChangedHandler(HandleSessionConnectionStatusChanged);

//Handler code

private void HandleSessionShutDown(SessionShutDownEventArgs earg)
{
    //Check the reason for shutdown and display message to user (or) do additional handling
    //at the end just logout the user
    _application.logout(_session.User.LoginId);
}

private void HandleSessionConnectionStatusChanged(SessionConnectionStatusChangedEventArgs
earg)
{
    if (earg.ConnectionStatus == SessionConnectionStatus.FAILED){
        _application.logout(_session.getUser().getLoginId());
    }
}

```

State guidelines

Avaya recommends that you consider the following guidelines with respect to object states in your custom code:

- [Check object for the appropriate state](#) on page 85
- [Check status of WorkItem](#) on page 86

Check object for the appropriate state

For all objects with defined state models, your custom code must:

1. Check for the appropriate state.
2. If the object is in the required state, make a Client API call on that object.

Reason

The Client SDK performs state validations on the SDK server. If the custom code does not check for the state of the object before making a Client API call, your custom application will have to perform unnecessary round trip checks to the server.

Objects with state models

The following table shows those objects that have a state model.

Object	State model
Session	Session state model diagram on page 31
WorkItem	WorkItem state model diagram on page 39
Voice MediaInteraction	VoiceMediaInteraction state model diagram on page 43
ChatMediaInteraction	ChatMediaInteraction state model diagram on page 46

Check status of WorkItem

For all operations on a WorkItem and the media interaction or document contained in that WorkItem, your custom code must check whether the WorkItem is Current before making most life cycle calls. For more information, see [WorkItem state model diagram](#) on page 39.

Reason

Some lifecycle operations on WorkItem are valid only if the WorkItem is the Current WorkItem. Your custom code must therefore:

1. Check if a WorkItem is Current.
2. If the WorkItem is Current, perform a lifecycle operation.

Log in and log out guidelines

Avaya recommends that you consider the following guidelines with respect to logging in and logging out of the application in your custom code:

- [Simultaneous log in and log out for Chat and Email](#) on page 87
- [Check WorkItem status during logout](#) on page 87

Simultaneous log in and log out for Chat and Email

An application user must log in or out of the Chat and Email channel simultaneously. A user cannot perform a single channel log in to or log out of the Chat channel or the Email channel.

Reason

In Avaya IC, the chat and email channels are not exclusive. Logging in or out of one of those channels will always log the user in or out of the other channel.

Check WorkItem status during logout

When an application user logs out of your custom application, your custom code must check the state of all WorkItems in the WorkItemList. If the list includes an active or Current WorkItem, your custom code must prevent the user from logging out.

Reason

Neither the Avaya IC system nor the Client SDK prevent an application user from logging out if the user has active work items.

Exception handling guidelines

Avaya recommends that you avoid the following exception in your custom code:

- [NullPointerException and ArgumentNullException](#) on page 87

Avaya recommends that you handle the following exceptions in your custom code:

- [ConnectionException](#) on page 88
- [AuthenticationException](#) on page 88

NullPointerException and ArgumentNullException

Your custom code must avoid NullPointerException and ArgumentNullException.

These exceptions occur if both of the following are true:

- Null parameters are passed to the Client API.
- Methods used in the custom code did not expect null parameters.

ConnectionException

Your custom code must handle ConnectionException, according to the requirements of the programming language:

- In Java, ConnectionException is a checked exception.
- In C#, your custom code must handle the exception. C# does not enforce this exception.

Reason

This exception occurs during the following events:

- Log in
- Log out
- Add or get an email attachment

AuthenticationException

Your custom code must handle AuthenticationException, according to the requirements of the programming language:

- In Java, AuthenticationException is a checked exception.
- In C#, your code must handle the exception. C# does not enforce this exception.

Reason

This exception occurs if an application user provides:

- An invalid username or password
- An expired password

Operation failure and success guidelines

Several objects in the Client API have OperationFailed or OperationSucceeded events. Avaya recommends that you consider the following guidelines when you handle these events in your custom code:

- [OperationFailed](#) on page 89
- [OperationSuccess](#) on page 89

OperationFailed

Your custom code must handle an `OperationFailed` event where available.

Reason

An operation failure event can indicate a problem with Avaya IC. This event can occur for operations that are asynchronous in Avaya IC.

An `OperationFailed` event can occur when certain calls fail at Avaya IC servers. For example, a `CollaborationBegin` call can raise an `OperationFailed` event if the wrong destination is provided.

An `OperationFailed` event also occurs if an application user tries to execute an invalid operation.

OperationSuccess

An `OperationSuccess` event occurs rarely. However, your custom code must handle an `OperationSuccess` event where available.

Reason

An `OperationSuccess` event occurs in rare cases in which the application user needs feedback on the success of an operation.

For example, an `EmailDraft.Send` event will raise an `OperationSuccess` event to acknowledge that an outbound email was sent.

Null return value guidelines

The Client API has many scenarios where a method call can return a null value. Your custom code must expect null return values. Avaya recommends that you code carefully to avoid unexpected `NullPointerExceptions`.

For example, `NullPointerExceptions` can be raised:

- Because an Email Cc is not a mandatory method, `getCc()` might return a null value.
- If an application user is not enabled for all channels, the `Session` method `getChannel()` might return a null value.

Customization guidelines

You can configure the Client SDK to customize access to the following items:

- Avaya IC agent properties
- EDU and ADU attributes
- Wrapup codes

This section includes the following topics:

- [Customization directory](#) on page 90
- [Customization files](#) on page 90
- [Deploying a configuration file](#) on page 92

Customization directory

Avaya IC installs the Client SDK customizing files in the following directory on a computer that hosts the Client SDK server:

```
IC_INSTALL_DIR\IC72\sdk\server\icsdk\custom\config\sdk\
```

Customization files

Important:

If your custom application does not require customizing, do not modify the entries in these files. Filter and send data to a custom application only if the data is absolutely necessary. For example, do not send all Avaya IC property sections or cache and filter all EDU data. If you send more than required data to the custom application, server performance might be severely impacted.

This section includes the following topics that describe the customizing files provided with the Client SDK:

- [SDKEduAttributesToFilter.properties](#) on page 91
- [SDKWorkItemAttributesFilter.properties](#) on page 91
- [SDKSessionAttributesFilter.properties](#) on page 91
- [SDKWrapupCodesCategoryGroups.properties](#) on page 92
- [SDKSupportedCharsets.properties](#) on page 92
- [SDKICPropertiesSections.properties](#) on page 92

SDKEduAttributesToFilter.properties

Description: Determines which EDU fields are sent as MediaInteraction and Document attributes.

Methods: Use the following methods to access the EDU fields:

- `MediaInteraction.getAttributes()`
- `Document.getAttributes()`

Note:

The EDU fields must be available at the SDK server bridge before they can be filtered. Define the EDU fields to be made available at the SDK server bridge in the `EDUFieldsToCache.properties` file. This properties file is in the configuration directory.

SDKWorkItemAttributesFilter.properties

Description: Determines which EDU fields are sent as WorkItem attributes.

Methods: Use the `WorkItem.getAttributes()` method to access the EDU fields.

Note:

The EDU fields must be available at the SDK server bridge before they can be filtered. Define the EDU fields to be made available at the SDK server bridge in the `EDUFieldsToCache.properties` file. This properties file is in the configuration directory.

SDKSessionAttributesFilter.properties

Description: Determines which ADU fields are sent as Session attributes.

Methods: Use the `Session.getAttributes()` method to access the EDU fields.

Note:

The ADU fields must be available at the SDK server bridge before they can be filtered. Define the ADU fields to be made available at the SDK server bridge in the `ADUFieldsToCache.properties` file. This properties file is in the configuration directory.

SDKWrapupCodesCategoryGroups.properties

Description: Determines which wrapup codes are sent to the custom application.

Methods: Use the `Session.getWrapupCodes()` method to access the wrapup codes.

SDKSupportedCharsets.properties

Description: Determines which character sets are:

- Supported by Avaya IC
- Exposed to the custom application
- Used when emails are sent

Methods: Use the `Session.getSupportedCharsets()` method to access the character sets.

SDKICPropertiesSections.properties

Description: Determines which Avaya IC agent property sections are available to the custom application.

Methods: Use the `Session.getICProperties()` method to access the properties.



Tip:

For information about the `IC_INSTALL_DIR\IC72\sdk\server\icsdk\custom\config\ICPropertiesSections.properties` file that you can use to customize the UOM, see *Avaya Agent Web Client Customization*.

Deploying a configuration file

After you update a configuration, place a copy of the file in the following directory on the machine that hosts the Client SDK server:

```
/WEB-INF/classes/com/avaya/ic/sdk/customization
```

Performance considerations

You can configure and tune the Client SDK to improve the performance of a custom application.

This section includes the following topics:

- [Using WebApplicationContext](#) on page 93
- [Configuring the messaging service](#) on page 95

Using WebApplicationContext

This configuration uses WebApplicationContext and SDKAppContext. Avaya recommends that you create and configure a single instance of the WebApplicationContext for each Client SDK process or custom application.

This section includes the following topics:

- [Creating and configuring WebApplicationContext](#) on page 93
- [Using the setter methods of WebApplicationContext](#) on page 94

Creating and configuring WebApplicationContext

You can use the following methods to create and configure an instance of WebApplicationContext:

WebApplicationContext(`URI appUri`): Allows an application to connect to the server directly with specified URI. This mechanism requires the following:

- The URI must contain the Web application context root.
- The server address and port specified in the URI must be directly accessible from the client machine.
- All of the ports specified in the server application deployment descriptor (`web.xml`) with the `messaging.listener.port.range` parameter must be directly addressable.

WebApplicationContext(`URI appUri`, `URI proxyUri`): Allows an application to connect to the server indirectly through a proxy server. This mechanism requires the following:

- The URI must be provided as a second parameter to the proxy server.
- Only the address and port of the proxy server can be directly addressable from the client machine.
- The proxy server must meet all requirements listed for [WebApplicationContext\(`URI appUri`\)](#).

WebAppContext(Uri appUri, boolean useSystemProxy): Allows an application to connect to the server indirectly through a proxy server. This mechanism uses the system properties to specify the proxy server.

For .NET, this mechanism uses the proxy specified in the Internet Explorer configuration.

Using the setter methods of WebAppContext

The following setter methods of the WebAppContext class provide additional tuning:

setMessageOutboundPoolSize(int messagingOutboundPoolSize): Specifies the number of threads in the thread pool that process outbound messages by forwarding them to the TCP transport layer. These threads perform network-related activities for the outbound traffic. The default value is 1.

setMessageInboundPoolSize(int messagingInboundPoolSize): Specifies the number of threads in the thread pool that process inbound messages by notifying all message listeners. The default value is 1.

Avaya recommends that you gradually increase a number of threads in this thread pool as the number of concurrent agent sessions grows. Use this method for server-side integration only.

setMessageSelectorPoolSize(int messagingSelectorPoolSize): Specifies the number of threads that service transport activity for asynchronous socket communications. For values of 1 or greater, a pool of specified number of threads are created and each agent session uses individual asynchronous duplex connections to the server. If the agent session communicates through a proxy, two separate asynchronous connections are established per agent. The default value is 1.

You can also set a value of 0. If you use a value of 0, no thread pool is used for the transport. Each agent session creates its own listener thread to service inbound traffic. Two separate synchronous connections are established to the server per agent.

setMessageSchedulerPoolSize(int messagingSchedulerPoolSize): Specifies the number of threads in the thread pool, in addition to the timer thread, that assists with scheduled tasks. The primary task is network status monitoring.



Important:

Do not use setMessageSchedulerPoolSize(int messagingSchedulerPoolSize) for .NET. Avaya recommends that you leave the value at 0. With that value, no thread pool is created in addition to the timer main thread.

Configuring the messaging service

You can configure the Client SDK messaging server. Some parameters in this section are already included in the application deployment descriptor (web.xml). You can specify the additional parameters in that file if needed.

This section includes the following properties:

- [Service/transport properties](#) on page 95
- [Workload capacity properties](#) on page 96
- [Functional properties](#) on page 97

Service/transport properties

avaya.ic.webclient.url: Binds the messaging service to the correct IP address on the server machine. If you specify the IP address incorrectly, the server raises a binding exception during startup.

messaging.listener.port.range: Specifies the server ports that you want the Messaging Service to bind. This property works in conjunction with [avaya.ic.webclient.url](#). The server requires the address and the port to perform binding. If the specified port is already occupied by another process, the binding algorithm will repeat an attempt with the next value specified with the range.

The binding algorithm will also repeat attempts if the Messaging Service has multiple listeners to start. The algorithm fails only if no ports are available in the specified range.

The default value is 8000-9000. This value provides a range of 1000 subsequent ports. Other examples of acceptable port ranges are:

- 8001: a single port
- 8001,8010,9001,9010: a comma separated list of ports for a total of 4 ports
- 8001-8010,9001-9010: a comma separated list of subranges of ports for a total of 20 ports



Important:

Your server must have the specified ports available and open for external access by the end-clients or proxy server.

messaging.listener.config: Specifies the number of listeners to start so that inbound and outbound activities can be serviced. You can set this property to only 1 or 2.

The default value is 2. This value means that two listeners will be started. One listener will service client connections for inbound work. The second listener will service client connections for outbound work. All connections are established and maintained by the client.

If you set the value to 1, the Messaging Service starts the inbound listener only. That listener must accept all client connections. If the custom application opens only one connection per agent session for inbound and outbound work, the inbound listener will accept all connections.

messaging.listener.backlog: Specifies the maximum queue length for incoming connection indications. Each indication is a request to connect. If a connection indication arrives when the queue is full, the server refuses the connection.

The backlog value must be a positive value greater than 0. If the value passed is equal or less than 0, then the Java platform default value will be assumed. The default value is -1.

The Java default on Windows is 50 for the socket backlog. To be able to handle a high number of concurrent requests to connect, set the value of the socket backlog at a much higher value than the default. 1024 or even higher is a recommended value for the Messaging Service listeners.

messaging.socket.timeout: Specifies a time-out for socket operations. The default value is -1. This value means that the socket operations will never expire, providing a time-out of infinity. This property applies only to synchronous socket operations.

messaging.inbound.capacity: Specifies the number of threads that will service transport activity for client connections for inbound work. This property is similar to the `messagingSelectorPoolSize` property of the SDK client. The default value is 1.

Note:

This property does not create a thread pool. The Client SDK uses a simple round-robin algorithm to assign work to the threads.

messaging.outbound.capacity: This property is useful if the SDK clients work through a proxy server and need to establish separate connections for outbound and inbound work. The default value is 1.

This property is not useful if you configure only an inbound listener configured. This property also does not have any value if all clients establish only one asynchronous duplex connection per agent session.

Workload capacity properties

messaging.inbound.thread.pool.size: Specifies the number of threads in the thread pool that will process inbound messages by notifying all message listeners. This property is the same as the `messagingInboundPoolSize` property of the SDK client.

messaging.outbound.thread.pool.size: Specifies the number of threads in the thread pool that will process outbound messages by forwarding them to the TCP transport layer. These threads will perform network related activities for outbound traffic. This property is the same as the `messagingOutboundPoolSize` property of the SDK client.

Functional properties

messaging.session.timeout: Specifies the duration of time after which session connectivity will be considered lost. For example, this time period can be invoked after the Client SDK detect a loss of client-server connectivity. After this time period expires, the system will send a shutdown signal. This signal is symmetric for both client and server. The default value is 600 seconds, or 10 minutes.

messaging.session.heartbeat: Specifies the ping interval for monitoring client-server connectivity. The default value is 60 seconds.

Chapter 5: Compiling and debugging a custom application

This section includes the following topics.

- [Supported compilers](#) on page 99
- [Logging](#) on page 99
- [Error messages](#) on page 114
- [Client SDK diagnostic information](#) on page 114
- [Debugging common problems](#) on page 117
- [Getting support](#) on page 121

Supported compilers

The following table shows the compilers that the Client SDK supports for each client platform.

Client platform	Supported compiler
.NET	.NET 2.0
Java	Java 1.6.0_10

Logging

The Client SDK logging generates several logs. Wherever possible, these logs:

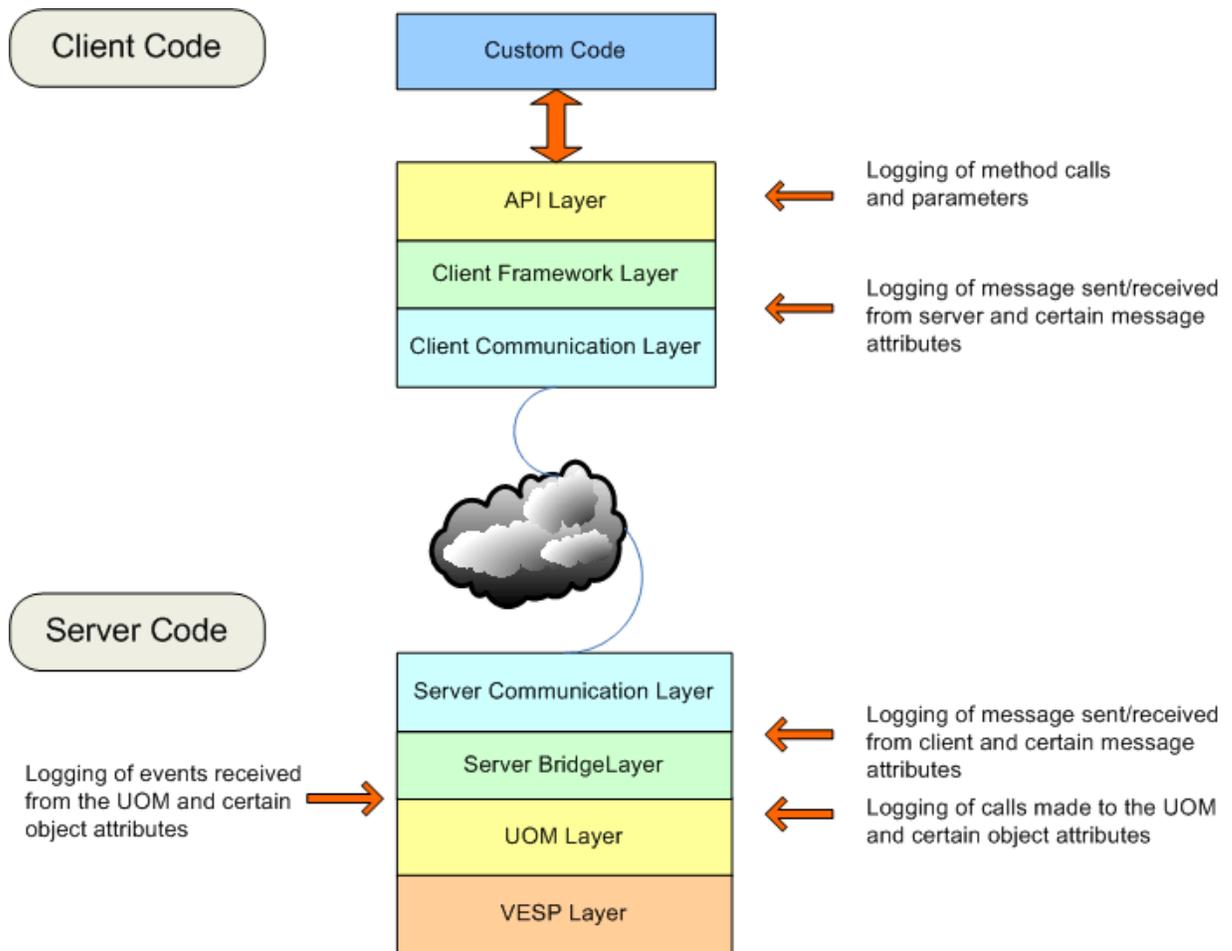
- Show the execution path in the Client SDK.
- Distinguish whether a problem occurred in a component of the Client SDK or in the code of a custom application.

This section includes the following topics:

- [Logging at the module boundaries](#) on page 100
- [Client SDK server logging](#) on page 101
- [Client SDK client logging](#) on page 104
- [Logging guidelines](#) on page 105
- [Tracing issues through Client SDK logs](#) on page 106
- [Sample log messages](#) on page 107

Logging at the module boundaries

The following diagram shows the module boundaries where the Client SDK performs logging. For more information about how to trace issues across the module boundaries, see [Tracing issues through Client SDK logs](#) on page 106.



Client SDK server logging

The Client SDK server uses log4j for logging. For information about how to use the log4j configuration file to change the logging level for the Client SDK server, see [Changing the level of Client SDK server logging](#) on page 103.

This section includes the following topics:

- [When server logging occurs](#) on page 101
- [Server log files](#) on page 101
- [Location of log4j configuration files](#) on page 102
- [Location of Client SDK server log files](#) on page 102
- [Changing the log4j file name](#) on page 102
- [Changing the level of Client SDK server logging](#) on page 103

When server logging occurs

The Client SDK server logs when:

- A message is received from the client. Logging includes the ID attribute of the message.
- A call is made into the UOM and Basic Services as a result of an arriving message. Logging can include the ID of the object on which the method is invoked.
- An event is received from the UOM. Logging can include the ID of the UOM object that fired that event.
- A message is sent to the client. Logging includes the message name. For a data message, logging also shows the dataid and rootcause attributes of the message.

Server log files

The Client SDK server includes the following log configuration files:

Log files	Description
log4j.xml.basic	Info level logging for standard Client SDK server logging
log4j.xml.comm	Info level logging and debug level logging for the communication and messaging module
log4j.xml.debug	Debug logging for reporting issues

Location of log4j configuration files

The log4j configuration files are provided in the `IC_INSTALL_DIR/IC72/sdk/server/icsdk.war` file.

When you start the SDK server for the first time, the server extracts the `icsdk.war` file into the following folder: `IC_INSTALL_DIR/IC72/sdk/server/icsdk`

The log4j configuration files are extracted in the following folder: `IC_INSTALL_DIR/IC72/sdk/server/icsdk/WEB-INF/classes`.

Location of Client SDK server log files

By default, the Client SDK server log files are stored in the following folder:

`IC_INSTALL_DIR/IC72/tomcat/logs/`

Changing the log4j file name

By default, the Client SDK server uses `log4j.xml` for the log4j configuration file name. You can change the file name, if necessary.

To change the file name for the log4j configuration file, perform the steps in the following table:

Operating system	Steps
Windows	<ol style="list-style-type: none">1. Open the following file for editing: <code>IC_INSTALL_DIR\IC72\tomcat\bin\icsdk.bat</code>2. Change the log4j configuration file name.3. Save the <code>icsdk.bat</code> file.4. Stop the Client SDK service in the Window Services control panel.5. Configure the Client SDK services with the Configuration Tool, as described in <i>IC Installation and Configuration</i>, to recreate the Client SDK service. <p>Note: You must re-configure the Client SDK services to make the renamed log4j file effective.</p>

Operating system	Steps
Solaris or AIX	<ol style="list-style-type: none"> 1. Open the following file for editing: <code>IC_INSTALL_DIR/IC72/tomcat/bin/icsdk.sh</code> 2. Change the log4j configuration file name. 3. Save the icsdk.sh file. 4. Navigate to the <code>IC_INSTALL_DIR/IC72/bin</code> directory. 5. Execute the following command to stop and restart the Client SDK server: nohup ./ictomcat.sh [stop start] SDK <p>Note: You must re-configure the Client SDK services to make the renamed log4j file effective.</p>

Changing the level of Client SDK server logging

The following items control the level of logging for the Client SDK server:

- The value of the Avaya IC property Agent/Desktop/WebClient/LogLevelServer
- The logging configuration file

If the log level setting in the Agent/Desktop/WebClient/LogLevelServer property is lower than that in the log4j.xml file, the property overrides the setting in the file. For example:

- If the log level in the server log4j.xml is debug and LogLevelServer is set to info, all logging for the Client SDK server occurs at info level.
- If the log level in the server log4j.xml is info and the LogLevelServer is set to debug, all logging for the Client SDK server occurs at the info level. Logging ignores the LogLevelServer property.

To change the logging level for the Client SDK server:

1. In IC Manager, set the required value for the Agent/Desktop/WebClient/LogLevelServer property.

For information on how to set this property, see *IC Administration Volume 2: Agents, Customers, & Queues*.

2. Select the server log configuration file with the correct level of logging.
3. Rename the selected file to log4j.xml.

For example, to set Client SDK server logging at the info level, rename log4j.xml.basic to log4j.xml.

Client SDK client logging

The Client SDK uses log4j for the Java sample client and log4net for the .NET sample client. To optimize consistent logging, the Client SDK uses log4j and log4net internally.

This section includes the following topics:

- [When Client SDK client logging occurs](#) on page 104
- [Logging in the .NET sample client](#) on page 104
- [Changing the level of logging for the .NET sample client](#) on page 105
- [Logging in the Java sample client](#) on page 105
- [Configuring the Java sample client to log to a file](#) on page 105

When Client SDK client logging occurs

On the client-side, the Client SDK logs when:

- The code calls a method of the Client API. Logging shows the ID of the object that was called and all the parameters of the method.
- A component is about to send a message to the server. Logging shows the ID attribute associated with the message.
- The server receives a message. Logging shows the name of the message. If the message is a data message, logging also shows the dataid and rootcause attributes of the message.

Logging in the .NET sample client

The .NET sample client uses the log4net logger module.

The .NET sample client has a log4net configuration file named CSharptester.exe.log4net. By default, this configuration file creates a debug.log in the current directory.

For information about logging with log4net, see the following Website:

<http://logging.apache.org/log4net>

Changing the level of logging for the .NET sample client

To change the level of logging for the .NET sample client, modify the CSharpTester.exe.log4net file to log at the required logging level.

Logging in the Java sample client

The Java sample client uses commons-logging. By default, the Java sample client logs everything to console. You can configure the Java sample client to log according to the log4j.xml specifications.

For information about configuring log4j, see the following Website:

<http://logging.apache.org/log4j/docs>

Configuring the Java sample client to log to a file

To configure the Java sample client to log to a file:

1. Copy the sample log4j.xml file:
 - From `IC_INSTALL_DIR\IC72\sdk\design\java\sample\config`
 - To `IC_INSTALL_DIR\IC72\sdk\design\java\sample\bin`
2. In a text editor, open the sample log4j.xml file for editing.
3. Uncomment the second appender and comment the first appender.
4. Save the log4j.xml file.

Setting the logging level for the Java sample client

To set the logging level for the Java sample client, modify the log4j.xml file to log at the desired level of logging.

Logging guidelines

Avaya recommends you to consider the following guidelines when you configure logging for the Client SDK:

Use prepackaged logging configurations for custom applications: To effectively log with the Client SDK, you must be familiar with the logging toolkit used for the platform of your custom application. For best results, Avaya recommends you to use the prepackaged configurations that are provided with the Client SDK for your custom applications.

Set default logging at info level: At info level, the Client SDK logs the information required to assist you in understanding how the Client SDK functions. Logging at Info level includes events that occur at the boundaries of the Client SDK.

Use debug level only when problems occur: At debug level, the Client SDK logs additional information that Avaya DevConnect and Support require to debug the problem.

Configure Avaya IC logging properties: Some Avaya IC properties control the configuration of logging in the log4j.xml file. For more information about those properties, including recommended settings, see *IC Administration Volume 2: Agents, Customers, & Queues*.

Tracing issues through Client SDK logs

In Client SDK logs, you can:

- Use the ID of an object to trace what happens when a single method of the Client API is called.
- Trace the events propagated by Avaya IC, such as WorkItem delivery.



Tip:

If the problem was initiated in the Avaya IC core system, verify that no issues occurred in that system before the object or event was passed to the Client SDK.

To trace an issue through the Client SDK logs:

1. Review the Client SDK server log to verify that the UOM fired the appropriate event for the object or method call.
 - If the log includes this event, the object or method call has traversed through the UOM and has reached the SDK server bridge. Continue with Step 2.
 - If the log does not include this event, the UOM might have a problem. Increase the log level to debug on the Client SDK server, reproduce the issue, and then contact Avaya for support.
2. Review the Client SDK server log to verify that the SDK server bridge sent a message over the Communication layer.
 - If the log includes this message, the object or method call has traversed through the SDK server bridge. Continue with Step 3.
 - If the log does not include this message, the SDK server bridge might have a problem. Increase the log level to debug on the Client SDK server, reproduce the issue, and then contact Avaya for support.
3. Review the Client SDK client log to verify that the Client SDK client framework received the message.
 - If the log includes this message, the object or method call traversed successfully over the Communication Layer. Continue with Step 4.

- If the log does not include this message, the Communication Layer might have a problem. Increase the log level to debug on the Client SDK client, reproduce the issue, and then contact Avaya for support.
4. In the Diagnostic Viewer, verify that the appropriate data was added to the Hierarchical Data Store.
 - If the Diagnostic Viewer shows the data, the Client SDK client framework and Hierarchical Data Store likely did not cause the problem.
 - If the Diagnostic Viewer does not show the data, the Hierarchical Data Store or the Client SDK client framework might have a problem. Increase the log level to debug on the Client SDK client, reproduce the issue, and then contact Avaya for support.

Sample log messages

Logging in the Client SDK is symmetrical. The Client SDK logs always include one or both of the following:

- Application user for which the item was logged
- Session for which the item was logged

This section describes the seven messages that the Client SDK client and server logs when the `makeAvailable` method is called on the `Session` object. The client logs message types 1 through 3. The server logs message types 4 through 6.

Note:

This section provides sample messages for only one scenario. However, Client SDK logs will include similar messages for other events and method calls.

This section includes the following topics:

- [Log message type 1](#) on page 108
- [Log message type 2](#) on page 109
- [Log message type 3](#) on page 110
- [Log message type 4](#) on page 111
- [Log message type 5](#) on page 112
- [Log message type 6](#) on page 113

Log message type 1

Description

Message type 1 is logged when a Client API call is made. The Client SDK client logs this message.

Sample message

```
2005-09-26 18:56:11,921|INFO|com.avaya.ic.client.sdk.impl.SessionImpl|
AWT-EventQueue-0| || |{agent1}.makeAvailable()| []
```

Message explanation

Message section	Description
2005-09-26 18:56:11,921	The date and time when the message was logged.
INFO	The log level at which the message was logged.
com.avaya.ic.client.sdk.impl.SessionImpl	The Logger Name that logged this message.
AWT-EventQueue-0	The ID of the thread on which this message was logged.
{agent1}.makeAvailable()	<p>The text of the log message. The text uses the following format:</p> <pre>{ (<objectid>).<methodName> (&param1=value 1....&paramN..valueN)</pre> <p>where</p> <ul style="list-style-type: none"> ● objectid is a unique identifier for the object on which the method was called. ● methodName is the name of the method that was called. ● paramN is the name of the parameter. ● valueN is the value of the parameter. If the value is an object reference, the log includes the output of the toString method on the object.

Log message type 2

Description

Message type 2 is logged just before the Client SDK client sends the method call as a message over the communication layer. The Client SDK client logs this message.

Sample message

```
2005-09-26 18:56:11,921|INFO |com.avaya.client.sdk.framework.impl.
SDKMessagingConnectorImpl|AWT-EventQueue-0|||Sending Message : Name =
session.makeavailable&Object Id = agent1|[]
```

Message explanation

Message section	Description
2005-09-26 18:56:11,921	The date and time when the message was logged.
INFO	The log level at which the message was logged.
com.avaya.client.sdk.framework.impl. SDKMessagingConnectorImpl	The Logger Name that logged this message.
AWT-EventQueue-0	The ID of the thread on which this message was logged.
Sending Message : Name = session.makeavailable&Object Id = agent1	<p>The text of the log message. The text uses the following format:</p> <pre>Sending Message : Name = <interfaceName>.<methodName> &Object Id = <objectid></pre> <p>where</p> <ul style="list-style-type: none"> • interfaceName is the name of the interface for which the message is being sent to the server. • methodName is the name of the method for which the message is being sent to the server. • objectid is a unique identifier for the object on which the method was called.

Log message type 3

Description

Message type 3 is logged when the communication layer delivers an event to the client framework. The Client SDK client logs this message.

Sample message

```
2005-09-26 18:56:12,015|INFO |com.avaya.client.sdk.framework.impl.
SDKMessagingConnectorImpl|MessagingClientRF_agent1-InboundExecutor_Thread-1|[[[[
Received Message : Name = data.modify&Data Id = agent1&Root Cause =
[LiveUserImpl.INIT_AVAILABLE], reason=[Setting a new user state. The old state was
AUXWORK.]][]
```

Message explanation

Message section	Description
2005-09-26 18:56:12,015	The date and time when the message was logged.
INFO	The log level at which the message was logged.
com.avaya.client.sdk.framework.impl. SDKMessagingConnectorImpl	The Logger Name that logged this message.
MessagingClientRF_agent1-Inbound Executor_Thread-1	The ID of the thread on which this message was logged.
Received Message : Name = data.modify&Data Id = agent1&Root Cause = [LiveUserImpl.INIT_AVAILABLE], reason=[Setting a new user state. The old state was AUXWORK.]	<p>The text of the log message. The text uses the following format:</p> <pre>Received Message : Name = <messageName>&Data Id = <fullyQualifiedObjectId>&Root Cause = <reasonServerSentMessage></pre> <p>where</p> <ul style="list-style-type: none"> ● messageName is the name of the message sent by the server. ● fullyQualifiedObjectId is a unique identifier for the object on the client. ● reasonServerSentMessage is the reason that the server sent the message.

Log message type 4

Description

Message type 4 is logged when a message handler on the SDK server bridge is invoked. The Client SDK server logs this message.

Sample Message

```
2005-09-26 19:16:51,312|INFO| com.avaya.ic.sdk.bridge.controllers.session.
SessionMessageHandlers$SessionMakeAvailableHandler|MessagingServiceRF-InboundExecutor
_Thread-6|||onMessage(): Received message for agent1: [session.makeavailable]||
```

Message explanation

Message section	Description
2005-09-26 19:16:51,312	The date and time when the message was logged.
INFO	The log level at which the message was logged.
\$SessionMakeAvailableHandler	The Logger Name that logged this message.
MessagingServiceRF-InboundExecutor_Thread-6	The ID of the thread on which this message was logged.
onMessage(): Received message for agent1: [session.makeavailable]	<p>The text of the log message. The text uses the following format:</p> <pre>onMessage(): Received message for <objectid>: [<interfaceName>.<methodName>]</pre> <p>where</p> <ul style="list-style-type: none"> ● objectid is a unique identifier for the object on which the method was called. ● interfaceName is the name of the interface for which the message is being sent to the server. ● methodName is the name of the method for which the message is being sent to the server.

Log message type 5

Description

Message type 5 is logged when an event is received from the UOM. The Client SDK server logs this message.

Sample Message

```
2005-09-26 19:16:51,327|INFO |com.avaya.ic.sdk.bridge.controllers.session.
SessionController$MyUserListener|UOMEventQueueExecutor_Thread-7|agent1|x3FFJip6aNZ0dO
HHFbInpfd||Received uom event for agent1: [STATE_CHANGED[INIT_AVAILABLE]]|[]
```

Message explanation

Message section	Description
2005-09-26 19:16:51,327	The date and time when the message was logged.
INFO	The log level at which the message was logged.
\$MyUserListener	The Logger Name that logged this message.
UOMEventQueueExecutor_Thread-7	The ID of the thread on which this message was logged.
Received uom event for agent1: [STATE_CHANGED [INIT_AVAILABLE]]	<p>The text of the log message. The text uses the following format:</p> <p>Received uom event for <objectid>: <reason></p> <p>where</p> <ul style="list-style-type: none"> objectid is a unique identifier for the object on which the method was called. reason explains why the UOM sent the event.

Log message type 6

Description

Message type 6 is logged just before the SDK server bridge sends a message over the Communication Layer. The Client SDK server logs this message.

Sample Message

```
2005-09-26 19:16:51,327|INFO |com.avaya.ic.sdk.bridge.controllers.session.
SessionController|UOMEventQueueExecutor_Thread-7|agent1|x3FFJip6aNZ0dOHHFbInpfd|Send
ing message to client: data id = agent1; rootcause = [LiveUserImpl.INIT_AVAILABLE],
reason=[Setting a new user state. The old state was AUXWORK.]|[]
```

Message explanation

Message section	Description
2005-09-26 19:16:51,327	The date and time when the message was logged.
INFO	The log level at which the message was logged.
com.avaya.ic.sdk.bridge.controllers.session.SessionController	The Logger Name that logged this message.
UOMEventQueueExecutor_Thread-7	The ID of the thread on which this message was logged.
Sending message to client: data id = agent1; rootcause = [LiveUserImpl.INIT_AVAILABLE], reason=[Setting a new user state. The old state was AUXWORK.]	<p>The text of the log message. The text uses the following format:</p> <pre>Sending message to client: data id = <fullyQualifiedObjectId> ; rootcause = < reasonServerSendingMessage></pre> <p>where</p> <ul style="list-style-type: none"> fullyQualifiedObjectId is a unique identifier for the object on the client. reasonServerSendingMessage explains why the server sent the message.

Error messages

All the error, warning, and information messages generated by Client SDK components are delivered as events. Each message includes:

- Message code
- Message description in English

The messages are not localized into non-English languages. Each message has an error code that you can use to customize or translate the message. For a list of messages and error codes, see [Error messages](#) on page 179.

Client SDK diagnostic information

The Client SDK includes an API that can provide a snapshot of the Hierarchical Data Store (HDS). You can use this snapshot information to debug issues and view the state of a sample client or custom application at any given point.

For more information about the HDS, see [Hierarchical Data Store](#) on page 16.

This section includes the following topics:

- [Using the diagnostic API](#) on page 115
- [When the Hierarchical Data Store is updated](#) on page 115
- [Viewing the HDS diagnostic information in the sample clients](#) on page 115
- [Using the HDS diagnostic information to identify problems](#) on page 116
- [Debugging problems found with the HDS diagnostic information](#) on page 117
- [Opening the Diagnostic Viewer](#) on page 117

Using the diagnostic API

You can use the diagnostic API to send HDS diagnostic information into a file after a specified user action or trigger. You can then analyze this information to debug issues with your custom application.

The sample clients display the output of the diagnostic API in the Diagnostic Viewer.



Important:

Use the diagnostic API for diagnostic purposes with your custom application in debug mode only.

To use the diagnostic API, note the following:

1. The diagnostic API is defined on the Session object.
2. Use the appropriate call from the following table.

Operating system	API call
.NET	<code>string DumpDiagnosticInfo()</code>
Java	<code>String dumpDiagnosticInfo()</code>

When the Hierarchical Data Store is updated

The Hierarchical Data Store is updated when one of the following messages arrives in the Client SDK client:

- `data.modify`
- `data.add`
- `data.remove`
- `event.notification`

When data is updated in the Hierarchical Data Store, your custom code can trigger an event.

Viewing the HDS diagnostic information in the sample clients

You can use the Diagnostic Viewer to view the HDS diagnostic information in the sample clients.

The user interface of the Diagnostic Viewer in the .NET sample client is not exactly the same as that of the Java sample client. However, both versions provide the same view of the folders and their contents.

Data elements in the Diagnostic Viewer

The Diagnostic Viewer displays the values returned by the `DumpDiagnosticInfo` method.

Changed elements in the Diagnostic Viewer

Changes to the structure of data elements in the Diagnostic Viewer and the relationships between the elements correlate:

- Changes in the Hierarchical Data Store
- Events that derive from the Client API

For example, in the diagram in [Viewing the HDS diagnostic information in the sample clients](#) on page 115, if the state of an agent1 changes, you can expect a `Session.StateChanged` event to occur.

Using the HDS diagnostic information to identify problems

You can use the diagnostic API to obtain a snapshot of the HDS at any given point in time and display the snapshot information in a user interface or log file for debugging purposes.

The Diagnostic Viewer in the sample clients is an example of how to display and view HDS diagnostic information. For example, with the Diagnostic Viewer in the sample clients you can:

- View the data in the Hierarchical Data Store.
- Determine whether messages arrived and related events were generated as expected.

If one or more of these messages do not arrive in the Client SDK client, the Hierarchical Data Store is not updated. This problem usually indicates that there is an issue with the Client SDK client framework or the Hierarchical Data Store.

If the Hierarchical Data Store was updated but an event triggered by that update did not generate, this problem usually indicates one of the following issues:

- The Hierarchical Data Store did not notify the Client SDK client framework of the change in data.
- The Client SDK client framework did not implement the event generation.
- A problem in the Client SDK client framework caused the event not to generate.

Debugging problems found with the HDS diagnostic information

If the HDS diagnostic information reveals a problem, do the following:

1. Review the logs to determine whether an exception is logged.
2. If an exception is logged, change the log configuration to debug, and reproduce the problem. Then send the logs to Avaya support.

Opening the Diagnostic Viewer

To open the Diagnostic Viewer from the .NET sample client, click the **Diagnostic Viewer** button.

To open the Diagnostic Viewer from the Java sample client, select **HDS Diagnostic** from the **Tools** menu.

Debugging common problems

This section includes the following topics:

- [Custom application cannot communicate with Client SDK server](#) on page 118
- [Chat or email work item is not delivered](#) on page 118
- [WorkItem state does not change](#) on page 119
- [.NET client encounters socket exception error during log in](#) on page 120

**Tip:**

If you encounter a problem that is not included in this section, review the sequence of steps in [Tracing issues through Client SDK logs](#) on page 106 to identify the component that triggered the issue.

Custom application cannot communicate with Client SDK server

Problem: Your custom application cannot communicate with the Client SDK server.

Solution: Verify the following items:

1. On the **SDK Server** tab of the Configuration Tool, confirm that the **SDK Server Machine** field includes the fully-qualified domain name of the computer that hosts the Client SDK server machine.
2. If necessary, rerun to the Configuration Tool to configure the Client SDK services, as described in *IC Installation and Configuration*.

Chat or email work item is not delivered

Problem: The custom application or sample client does not receive a chat or email work item sent to an application user.

Solution: Verify the following steps:

1. Review the WebACD Administration pages in IC Manager to determine who is the current owner of the work item:
 - If the work item is assigned to an Avaya IC agent, the Avaya IC core system has routed the work item correctly. Continue with Step 2.
 - If the work item is not visible in the WebACD Administration pages or has not been routed to an Avaya IC agent:
 - Verify the configuration of all Avaya IC agents.
 - Verify the configuration of the Web Management servers.
 - Contact Avaya support.
2. Review the Client SDK server log to verify that the UOM has generated a ContactAdded event.
 - If the log includes this event, the work item has traversed through the UOM and has reached the SDK server bridge. Continue with Step 3.
 - If the log does not include this event, the UOM might have a problem. Increase the log level to debug on the Client SDK server, reproduce the issue, and then contact Avaya support.
3. Review the Client SDK server log to verify that the SDK server bridge sent a message over the Communication layer.
 - If the log includes this event, the work item has traversed through the SDK server bridge. Continue with Step 4.

- If the log does not include this event, the SDK server bridge might have a problem. Increase the log level to debug on the Client SDK server, reproduce the issue, and then contact Avaya support.
4. Review the Client SDK client log to verify that the SDK client framework received the message.
 - If the log includes this message, the work item traversed successfully over the communication layer. Continue with Step 5.
 - If the log does not include this message, the Communication Layer might have a problem. Increase the log level to debug on the SDK client, reproduce the issue, and then contact Avaya support.
 5. In the Diagnostic Viewer, verify that the data was added to the Hierarchical Data Store.
 - If the Diagnostic Viewer shows the data, the SDK client framework and Hierarchical Data Store likely did not cause the problem.
 - If the Diagnostic Viewer does not show the data, the Hierarchical Data Store or the SDK client framework might have a problem. Increase the log level to debug on the SDK client, reproduce the issue, and then contact Avaya support.

WorkItem state does not change

Problem: The state of a WorkItem does not change as expected.

Solution: Verify the following steps:

1. Review the Client SDK server log to verify that the UOM has generated the state change event.
 - If the log includes this event, the state change call has traversed through the UOM and has reached the SDK server bridge. Continue with Step 2.
 - If the log does not include this event, the UOM might have a problem. Increase the log level to debug on the Client SDK server, reproduce the issue, and then contact Avaya support.
2. Review the Client SDK server log to verify that the SDK server bridge sent a message over the Communication layer.
 - If the log includes this event, the state change call has traversed through the SDK server bridge. Continue with Step 3.
 - If the log does not include this event, the SDK server bridge might have a problem. Increase the log level to debug on the Client SDK server, reproduce the issue, and then contact Avaya support.

3. Review the Client SDK client log to verify that the SDK client framework received the message.
 - If the log includes this message, the state change call has traversed over the communication layer. Continue with Step 4.
 - If the log does not include this message, the communication layer might have a problem. Increase the log level to debug on the SDK client, reproduce the issue, and then contact Avaya support.
4. In the Diagnostic Viewer, verify that the data was added to the Hierarchical Data Store.
 - If the Diagnostic Viewer shows the data, the SDK client framework and Hierarchical Data Store likely did not cause the problem.
 - If the Diagnostic Viewer does not show the data, the Hierarchical Data Store or the SDK client framework might have a problem. Increase the log level to debug on the SDK client, reproduce the issue, and then contact Avaya support.

.NET client encounters socket exception error during log in

Problem: The following socket exception error is generated when you log in to a .NET client:

```
System.Net.Sockets.SocketException: An operation on a socket could not be performed because the system lacked sufficient buffer space or because a queue was full at System.Net.Sockets.Socket.InitializeSockets() at System.Net.Sockets.Socket..ctor()
```

Solution: This socket exception error is a known .NET 1.1 issue with network bindings. You can encounter this issue with a .NET client, especially if you use VMWare or during the development phase.

According to Microsoft, the socket exception error occurs if the number of protocol bindings exceeds 50, and you use the IPAddress class directly or indirectly.

For more information about this .NET 1.1 issue, including information about the recommended solution, see the following Microsoft Knowledge base article:

<http://support.microsoft.com/default.aspx?scid=kb;en-us;815209>

Getting support

Development support for the Client SDK is available only through the Avaya DeveloperConnection Program (DevConnect). You must follow the DevConnect guidelines to obtain support for the Client SDK.

To obtain support or participate in testing opportunities, use the DevConnect Portal on the following Website:

<http://www.devconnectprogram.com>

Innovator, Premier, or Strategic level members of the DevConnect Program are entitled to technical support. Support is unavailable to Registered level members of the program. If you are a Registered level member and require support, you can apply for a higher level of membership. Avaya determines membership status at all levels. Membership at the Innovator, Premier, or Strategic level is not open to all companies.

When you contact DevConnect to request technical support, provide the following information:

- Configuration settings, including the SDK server bridge configuration parameters
- Usage scenario, including all steps required to reproduce the issue
- Any custom code that exercises the Client API
- Screen shots, if the issue occurs in a sample client or is visible in your custom application
- Copies of all logs related to the issue, including all client and server logs
- All other information that you gathered when you attempted to resolve the issue

Chapter 6: Localization and internationalization

The Client SDK is internationalized and includes support for non-English custom applications. You can use the Client SDK to build internationalized custom applications or to integrate into existing internationalized applications.

The Client SDK does not include localized versions of the sample clients or error messages. They are provided in English only. The error messages and other strings in the Client SDK are configured to make them available for you to customize and translate. Each error message has a error code associated with them. For a list of messages and error codes, see [Error messages](#) on page 179.

All strings in the Client SDK are in Unicode format. Avaya IC internally uses UTF-8 format. However, Avaya IC and the Client SDK perform the conversions between Unicode and UTF-8 formats. Your custom application does not need to perform the conversions.

For information about how to customize the character sets that are used in your custom application, see [SDKSupportedCharsets.properties](#) on page 92.

Appendix A: Sample scenarios



Important:

The code provided in this section is for example only. The actual code in the sample clients may differ with the samples provided in this section. Always review the code in the sample clients before you implement any of the scenarios.

This section includes the following topics.

- [Login scenario](#) on page 126
- [Logout scenario](#) on page 128
- [Agent availability scenario](#) on page 129
- [Display channel properties scenario](#) on page 131
- [Workitem lifecycle scenario](#) on page 133
- [Workitem collaboration scenario](#) on page 136
- [OnHold/OffHold indication scenario](#) on page 138
- [Display text message scenario](#) on page 139
- [Display email scenario](#) on page 142
- [New Outbound email scenario](#) on page 143
- [Reply to email scenario](#) on page 144
- [Display WorkItem History scenario](#) on page 146
- [Display Customer History scenario](#) on page 148
- [AddressBook scenario](#) on page 150
- [Retrieving Workitem Contact Attributes scenario](#) on page 151
- [Voice Call scenario](#) on page 153

Login scenario

This scenario provides the information needed to use the Client SDK to log in to the Avaya IC system.

Primary objects used in scenario

This scenario uses the following objects: [Application](#), `ApplicationFactory`, `WebApplicationContext`, [Session](#).

Implementation considerations

Before you write your custom code, consider the following:

When can you consider an application user to be logged in? An application user can be considered logged in when the application user was authenticated, and a valid `Session` object is available through the `Application`.

At this point, the `Session` object is in the `logged_in` state. However, `logged_in` is a transitional state, and the session is still not initialized. A session is not useful before initialization. Do not allow the application user to perform any session-based operations until the `Session` object is in a stable state of `Auxwork` or `Available`.

The Avaya IC property `Agent/Desktop/AuxWorkOnLogin` determines whether the initial stable state is `Auxwork` or `Available`.

How are log in failures propagated? Log in failures are propagated either by exceptions during the log in request, or through the `Session.OperationFailed` event.

Why are authentication and initialization not a single-step process? Authentication and initialization are two-step processes. These processes allow the application user to register for the event feed from the client library prior to the session initialization. At this point, the Client SDK starts to receive events from the Client SDK server.

Therefore, at the end of a successful authentication, the application user receives a valid session. You can use this valid session to register for events. After registration, invoke `initialize()` on the `Session` object to initialize the session.

Can you log in an application user twice with the same Tomcat instance? An application user cannot have two active Avaya IC sessions. If an application user logs in from a different Client SDK session, Avaya IC terminates the first session.

Can you log in an application user with two different Tomcat instances? Since the back-end Avaya IC system is the same, an application user cannot have two active Avaya IC sessions. If an application user logs in from a different Client SDK session on another Tomcat instance, Avaya IC terminates the first session.

High-level steps

This scenario requires the following steps:

1. Set an instance of the Application object using the ApplicationFactory.
2. For the authentication process, use the Application to log in and get access to the Session.
3. Use the Session object to register listeners to receive events.
4. Initialize the Session to complete the process.

Event changes

This scenario involves the following event changes for the objects:

1. During initialization, the Session moves to the transitional Initialized state.
2. The Session moves to a stable Auxwork or Available state.
3. As part of the initialization, channels are also logged in and move to a stable state of Busy or Idle, depending on whether the application user is in Auxwork or Available.

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\  
PresentationSessionImpl.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\  
Controller\UIController.cs
```

Logout scenario

This scenario provides the information needed to gracefully log out an application user from the Avaya IC system with the Client SDK.

Primary objects used in scenario

This scenario uses the following object: [Application](#).

Implementation considerations

Before you write your custom code, consider the following:

How is the session impacted when logout is invoked on the application? When you invoke logout on the application, the session state changes to `logged_out`.

What event stream can you expect? The session state will be `logged_out`.

What determines that an application user is logged out? The following items determine that an application user is logged out:

- The session state is `logged_out`.
- Any operation on the session reference generates an `OperationFailed` event.
- Any attempt to access the session with `Application.getSession(id)` with the ID of that session returns a null.

What happens if the server host process is terminated without logging out? Your custom application gets a `ConnectionStatusChanged` event. The following table shows what this event indicates.

Event	Description
FAILED	Indicates the session connection has failed permanently, and the connection cannot be restored. For all practical purposes, the session is no longer valid.
IMPAIRED	Indicates the session connection is impaired, and the Client SDK will try to reconnect.

Note:

All user operations that need server involvement need to be disabled if the connection status is `IMPAIRED` or `FAILED`. These user operations include session availability and workitem lifecycle operations. If possible, disable all user interface elements when the connection is impaired. If the connection cannot be restored, ensure the application logs out.

High-level steps

This scenario requires the following steps:

1. Access the Application object.
2. Use the Application object to log out.

Event changes

This scenario involves the following event change for the object:

- Session moves to the Logged_out state.

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\
PresentationSessionImpl.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\
Controller\UIController.cs
```

Agent availability scenario

This scenario provides the information needed to enable an application user to set himself or herself:

- As Available to start receiving work
- In Auxwork to stop receiving work

Primary objects used in scenario

This scenario uses the following object: [Session](#)

Implementation considerations

Before you write your custom code, consider the following:

Which operations need to be invoked? Invoke the following operations:

- `Session.enterAuxwork()`
- `Session.makeAvailable()`

Appendix A: Sample scenarios

Which events need to be acted on? Who needs these events? Both operations affect the state of the Session. Therefore, your custom application must listen to and act on `Session.StateChanged` events.

Which server-side responsibilities are involved? When invoked, these methods change the state of the session and affect the state of the logged in channels. Therefore, also listen to channel state change events.

Similarly, after channels are available, they can receive work. The Avaya IC core servers might start to deliver queued items. If this occurs, listen to `WorkList.WorkItemAdded` events.

Do you want to rely on an `OperationSuccess` message or a `State Change`? Most session operations lead to session state changes. If the state change indicates that the operation succeeded, the Client SDK does not generate an additional `OperationSuccess` event.

Operations that do not lead to a state change generate an explicit `OperationSuccess` event to indicate the operation was successful. This event is not limited to the Session object but also occurs for operations exposed by all other Client API objects.

Which API call can manually change endpoints and load? None! The Avaya IC 7.2 release of the Client SDK does not support manual endpoint manipulation. This release supports only blended mode. Therefore, the Client API does not expose any objects or methods that can manually change endpoints and load.

High-level steps

This scenario requires the following steps:

1. Access the Session object.
2. Through Session, invoke the operation that enables an application user to receive work.
3. Listen to necessary events that indicate that the operation was successful.
4. Through Session, invoke the operation that enables an application user to stop receiving work.
5. Listen to necessary events that indicate that the operation was successful.

Event changes

This scenario involves the following event changes for the objects:

- For Available, Session object moves from `Init_available` to `Available`.
- For Auxwork, Session object moves from `Init_auxwork` to `Auxwork`.

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\
controllers\SessionControllerImpl.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\
Controller\UIController.cs
```

Display channel properties scenario

This scenario provides the information needed to ensure that the full view of each channel is available in the custom application and to ensure that:

- The application displays the channel state and ceiling in the status bar.
- The application displays health status changes through different icons.
- The channel background displays the delivery status. For example, the icon is:
 - White when work items can be delivered.
 - Black when work items cannot be delivered.

Note:

The Java sample client implements this user interface specification. The .NET client has a different user interface view and does not follow this scenario.

Primary objects used in scenario

This scenario uses the following objects: [Session](#), [Channel](#), [VoiceChannel](#), [EmailChannel](#), [ChatChannel](#).

Implementation considerations

Before you write your custom code, consider the following:

What are the potential uses of the health status? The health status indicates the ability of the channel to service the interactions and documents associated with that channel. If the channel health is impaired or failed, certain operations on the interaction or document might not be possible. Therefore, use the health status to determine the enablement or disablement of user interface buttons that are associated with WorkItem, Interaction, or Document operations that depend on a healthy channel.

Appendix A: Sample scenarios

How can you make channels available and unavailable? The only way to enable channels to receive or not receive work is to toggle session availability. Use the following methods:

- `Session.enterAuxwork()` to make all channels busy
- `Session.makeAvailable()` to make all channels available to receive work

You cannot individually modulate channel endpoints. Avaya IC allows that operation only in manual mode. The Client SDK does not support manual mode.

What mechanism can you use to notify the application of health and delivery status, state changes, and other relevant events? Use the following explicit event callbacks to be notified of a change in the health or delivery status:

- `HealthStatusChanged`
- `DeliveryStatusChanged`

High-level steps

This scenario requires the following steps:

1. Access the Session object.
2. Register listeners for Channel events.
3. After the handlers receive an event:
 - a. Identify the event.
 - b. Make the necessary changes in the user interface.

Event changes

This scenario involves the following event changes for the objects:

- `Channel.HealthStatusChanged`
- `Channel.DeliveryStatusChanged`

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\
controllers\ChannelViewControllerImpl.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
core\StatusBarPanel.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
core\AgentSummaryPanel.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\
Controller\UIController.cs
```

Workitem lifecycle scenario

This scenario provides the information needed to exercise the lifecycle operations of a delivered work item from delivery to completion.

Note:

This scenario assumes that AutoAccept is disabled through the Avaya IC properties. With AutoAccept disabled, you must accept the work item after delivery.

Primary objects used in scenario

This scenario uses the following objects: [WorkList](#), [WorkItem](#)

Implementation considerations

Before you write your custom code, consider the following:

How do you check the values of PromptOnArrival and WrapUpEnabled Avaya IC properties? You can access Avaya IC properties through appropriate methods exposed on the Session object. However, by default, only a subset of the Avaya IC properties is sent to the client and accessible through the session. The SDKICpropertiesSections.properties file defines the property sections that are sent to the client. To access additional Avaya IC properties, you must customize this file. For more information, see [SDKICPropertiesSections.properties](#) on page 92.

Which events need to be considered? Which entities generate these events? For example, what are the required WorkItem and WorkList state change events? The WorkList object generates WorkItem delivery and removal events. These events indicate the delivery and removal of a work item. The `WorkItemAdded` event provides access to the most recently delivered `WorkItem` object.

WorkItem operations trigger most lifecycle changes on the Workitem and lead to corresponding changes in the WorkItem state. These operations include:

- `accept()`
- `decline`
- `makeCurrent()`
- `defer()`
- `release()`
- `complete()`

Note:

All work items must go through the Wrapup state. This requirement ensures that you can take all necessary custom actions before calling the `WorkItem.complete()` API. Therefore, even if an application user declines a work item or a RONA timeout occurs before an application user accepts, the work item enters the Wrapup state. For scenarios in which the application user does not accept a work item, and no wrapup data needs to be collected, the `WorkItem.getTerminateReason()` provides the context to identify what triggered the Wrapup state.

You must look at these TerminateReason codes to determine which of the following operations needs to occur next:

- Launch a wrapup sequence.
- Invoke immediately the `complete()` API to terminate the WorkItem.

Usually, you should immediately invoke the `WorkItem.complete()` when the state changes to Wrapup and TerminateReason is not equal to NORMAL. Your code should invoke the `WorkItem.complete()` API no matter how you configure the `WrapupEnabled` Avaya IC property.

When and why must you override the `WrapupEnabled=true` setting? The Avaya IC property called `WrapupEnabled` controls whether wrapup occurs after an application user releases a work item. Your custom application must check this value to determine whether to launch the wrapup sequence after a work item enters the Wrapup state.

However, in some scenarios, you might need to suppress wrapup even if the `WrapupEnabled` property is set to true. Most of these scenarios are captured in the TerminateReasons. Your custom code should check the TerminateReason to determine whether to ignore the value of the `WrapupEnabled` property.

Why is the client responsible for completing the work item? Based on our experience, integrators and developers frequently customize wrapup or take special steps after a workitem enters the Wrapup state. Therefore, the Client SDK ensures that each work item goes through the Wrapup state. The Client SDK developer must invoke `complete()` on the `WorkItem` object to terminate and to remove the work item from the work list of an application user. Your client can perform these operations after the client collects wrapup data or immediately if the TerminateReason indicates a special scenario.

Where are the reasons for entering wrapup defined? The TerminateReasons are defined on the `WorkItem` object. These reasons are accessible through the `WorkItem.getTerminateReason()` method.

High-level steps

This scenario requires the following steps:

1. Register listeners for WorkItem and WorkList events.

2. Depending upon the values of Avaya IC properties, prompt on arrival if the WorkItem is in Alerting state.
3. After the work item is accepted, hook in operations that manipulate state changes.
4. Set the event handlers that update the user interface. For example, buttons to reflect operational state, or a list widget to display the latest state and currentness.
5. After the work item is released, check the value of TerminateReason and the wrapupEnabled property to determine if your application must display the **Wrap Up** dialog box.
6. If the wrapupEnabled property is disabled or the TerminateReason is not NORMAL, complete the work item after the state changes to wrapup.

Event changes

Depending on the operation invoked, you get state and event changes for the objects. These changes include:

- WorkItem.StateChanged
- WorkItem state changes from Alerting to Working to Wrapup

For more information, see the implementation considerations in this section.

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\
controllers\WorkListViewControllerImpl.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
core\InteractionListPanel.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\model\
action\helper\WorkItemActionHelperFactory.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\
Controller\UIController.cs
```

Workitem collaboration scenario

This scenario provides the information needed to transfer a WorkItem that contains a VoiceMediaInteraction to another voice agent.

Primary objects used in scenario

This scenario uses the following objects: [WorkItem](#), [VoiceMediaInteraction](#).

Implementation considerations

Before you write your custom code, consider the following:

How do you disable prompt on arrival and wrapupEnabled, with auto-accept set to true?

You disable these actions through Avaya IC properties. When auto-accept is set to true, the WorkItem is automatically accepted into the application user worklist on delivery and made Current by the Client SDK server. The application user can then start to handle the work item immediately.

When can an application user perform WorkItem operations? Usually, an application user can perform operations only on a current work item. An agent must accept a work item before that work item can become current. To reflect this sequence, you must ensure that all buttons associated with WorkItem operations are tied into the `WorkItem.isCurrent()` boolean value. You only need to tie Accept and Decline to the WorkItem state value of Alerting to enable those buttons.

Where do the collaboration methods reside? Are the prerequisites for collaboration satisfied? For example, is the required channel available? Is this reflected in the button state? Collaboration methods reside on the WorkItem, not the channel. Collaboration includes conferences and consults. In the Client SDK, you use a particular channel to collaborate a work item. As a result, you cannot collaborate a WorkItem that does not have an associated channel.

When your custom code enables or disables user interface buttons associated with the collaborative operations, consider the requirements of collaboration.

Which events indicate a successful transfer or a failed transfer? What actions does the custom application need to take after an event is received? If a state change can uniquely identify that a transfer was successful, the Client API does not raise an `OperationSuccess` event. Therefore, you will not see an `OperationSuccess` event after the operation of `tWorkItem.transfer()` API. If the transfer succeeds, the WorkItem goes into the Wrapup state. If the transfer fails, the Client SDK raises an `OperationFailed` event, which indicates the reason for the failure.

Note:

Especially in collaboration and transfer scenarios, you can get a state change after you invoke a collaborate or transfer operation. This state change occurs because the WorkItem can enter a transitional CONFERENCING/TRANSFERRING/CONSULTING state, and then return to the previous state if the conference or transfer fails. You will receive state changes for this operation. However, those state changes do not indicate that the operation was successful.

Are reasons available if the transfer fails? The Client SDK provides these reasons in the OperationFailed event. At this time, no enumerations are defined.

From a state change perspective, what will happen if a collaborative request fails? You receive state change events. A rollback from the transitional state to the previous stable state causes these state change events.

Is collaboration with a chat media interaction a two-step process? No, collaboration with a chat media interaction is not a two-step process. Specifically for chat media interactions, collaboration is a single-step process. After the collaboration is successful, collaboration immediately enters the collaboration complete phase. For a conference, this step confirms that the chat is successfully conferenced and all parties are in the call.

This single step process means:

1. In the case of conference, the chat is successfully conferenced, and all parties are in the call.
2. Consultative collaboration is not currently supported for chat media because this collaboration is essentially a `transfer()`.
3. `collaborationComplete(CONFERENCE/CONSULT)` is invalid for chat media based work items.

Can a chat transfer request be cancelled after initiation? Yes. You can cancel the chat transfer by invoking the `transferCancel()` API on the `Workitem` object.

High-level steps

This scenario requires the following steps:

1. Register listeners for WorkItem events.
2. Read the destination from the text box, and determine the `destinationType`. In this case, the `destinationType` is `agent`.
3. Invoke the collaboration methods and wait for events that indicate a success or failure.

Event changes

This scenario involves the following event changes for the objects:

- WorkItem state change events from Working to Conferencing to Working
- OperationFailed and Operation Success events

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\
controllers\WorkListViewControllerImpl.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
core\InteractionListPanel.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\model\
action\helper\WorkItemActionHelperFactory.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\
Controller\UIController.cs
```

OnHold/OffHold indication scenario

This scenario provides the information needed to display the work list and update the toolbar when a voice media interaction:

- Is on hold.
- Has been reconnected.

Primary objects used in scenario

This scenario uses the following object: [VoiceMediaInteraction](#)

Implementation considerations

Before you write your custom code, consider the following:

Which operations allow an application user to place a voice interaction on hold? Use the `hold()` operation exposed on the `VoiceMediaInteraction` object.

Which event notifies that the voice interaction is on hold or reconnected?

`VoiceMediaInteraction` state changes indicate whether the interaction is on hold or reconnected.

Can a work item be Current or Working when a voice interaction is on hold? Yes. An application user can work on a work item even if the voice call is on hold. Therefore, the Client SDK supports this scenario. You must invoke life cycle operations to change the currentness of a work item. For example, you can invoke the `defer()` API on a work item that contains an email, or the application user can make another work item current.

High-level steps

This scenario requires the following steps:

1. Provide a mechanism to invoke the appropriate methods.
2. Implement handlers that react to state change events.
3. Update the user interface based on the state change events.

Event changes

This scenario involves the following event changes for the objects:

- VoiceMediaInteraction.State.ACTIVE to VoiceMediaInteraction.State.INACTIVE
- VoiceMediaInteraction.State.INACTIVE to VoiceMediaInteraction.State.ACTIVE

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\
controllers\MediaInteractionViewControllerImpl.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\
Controller\UIController.cs
```

Display text message scenario

This scenario provides the information needed to implement a solution where:

- An application user can exchange messages with a customer.
- The transcript highlights the messages exchanged. For example you can highlight the customer name with blue and the application user name with red.

Primary objects used in scenario

This scenario uses the following objects: [ChatMediaInteraction](#), TranscriptDocument.

Implementation considerations

Before you write your custom code, consider the following:

Which method do you use to navigate to the chat interaction and access the transcript?

A Document class named TranscriptDocument represents the transcript. To access the transcript, do the following:

- Access ChatMediaInteraction from MediaInteractionList.
- MediaInteractionList is contained within WorkItem.
- Access TranscriptDocument from ChatMediaInteraction.

After the ChatMediaInteraction is available, access the transcript with the `ChatMediaInteraction.getTranscript()` method. You can access a collection of all TranscriptLine objects with the `transcriptDocument.getTranscriptLines().getAll()` method.

Which mechanism do you use to register for events associated with message updates?

The TranscriptDocument generates a TranscriptLineAdded event when a message is received from the associated chat room. The following example shows a mechanism to register for these events:

```
session.registerListener(TranscriptDocument.TranscriptLineAdded.TYPE, new  
TranscriptLineAddedListener());
```

Note:

The context of a TranscriptDocument is available in the `TranscriptDocument.TranscriptLineAdded` Event object.

Which user interface transcript element needs to be updated on receiving messages?

Usually, the window that displays the real-time message exchange has an embedded listener to receive these event callbacks.

Where are the message types defined? The different message types are defined on the TranscriptLine that represents a message. Currently, the two types of messages that can be exchanged are TEXT and URL.

How do you identify between different sources of the same type? The TranscriptLine defines the MessageType and an OriginType that indicates the originator. The following table shows who can originate a message in the chat room.

Message originator	Description
CALLER	The customer
AGENT	The application user
SYSTEM	A broadcast message from the chat subsystem

A chat room can host multiple agents or customers. For example, messages can arrive from two different AGENT originator types. `TranscriptLine.getOriginatorHandle()` uniquely identifies the chat handle for the originator of a message.

Can you identify parties? For example, who is the source of a particular message? The `OriginType` identifies parties.

High-level steps

This scenario requires the following steps:

1. On `WorkItem` delivery:
 - a. Access the transcript document for the `ChatMediaInteraction`.
 - b. Use the transcript document to populate the transcript window.
2. Implement handlers that react to transcript events.
3. Update the user interface based on the transcript events.

Event changes

This scenario involves the following event change for the objects:

- `TranscriptLineAdded` event on `TranscriptDocument`

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\
controllers\MediaInteractionViewControllerImpl.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\model\
TextEditorPane.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
chat\ChatTranscriptPanel.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\
Controller\UIController.cs
```

Display email scenario

This scenario provides the information needed to:

- Receive a WorkItem that contains an EmailDocument.
- Display the contents of that email in the email viewer.

Primary objects used in scenario

This scenario uses the following objects: [WorkItem](#), [EmailDocument](#).

Implementation considerations

Before you write your custom code, consider the following:

Does EmailDocument have a state model? No, EmailDocument does not have a state model. The Client SDK considers a Document object to be static. Therefore Document and related objects do not have any state.

What is the significance of the different email types? How does your custom application know the type of an email? The Client SDK defines email types in the EmailMessage object. The types define the different categories of e-mails that Avaya IC tracks.

To access the type of an email document, use the `Document.getType()` API. The type of a Document can be Email, ChatTranscript, or Draft.

High-level steps

This scenario requires the following steps:

1. When a WorkItem is delivered, determine whether the work item contains an EmailDocument.
2. Display the fields of the EmailDocument in the email viewer.

Event changes

This scenario does not involve any event changes for the objects.

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
document\DocumentListPanel.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
document\EmailView.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
document\EmailMessagePreviewTab.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
document\AttachmentPanel.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\
controllers\EmailViewControllerImpl.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\
Controller\UIController.cs
```

New Outbound email scenario

This scenario provides the information needed to create and send new outbound email.

Primary objects used in scenario

This scenario uses the following objects: [Session](#), [EmailChannel](#), [WorkItem](#), [EmailDocument](#), and [EmailDraft](#).

Implementation considerations

Before you write your custom code, consider the following:

Which operation sends the email? `EmailDraft.send()` sends an outbound email.

High-level steps

This scenario requires the following steps:

1. Register listeners and create handlers for `WorkItemAdded`, `EmailDraftOperationSucceeded`, and `EmailDraftOperationFailed` events.
2. Get access to `EmailChannel` object through `Session` object.
3. Invoke `createNewOutboundEmail()` API on the `EmailChannel` object. On successful invoking of the API, SDK server raises the `WorkItemAdded` event, which will have email document representing a new outbound email as a draft.

Appendix A: Sample scenarios

4. Get access to the `EmailDocument` object from the `WorkItemAdded` event.
5. Invoke the `GetDraft()` API with the `EmailDraftType.NEW_OUTBOUND` parameter on the `EmailDocument` object to get an access to the `EmailDraft` object.
6. Set `OutboundAccount` on `EmailDraft` object. You can retrieve registered outbound account list in IC using the `getOutboundAccountList()` API on `EmailChannel` object.
7. Invoke the `Send()` API on the `EmailDraft` object, which sends a new outbound email.
8. Determine the success or failure status of an outbound email from the `EmailDraftOperationSucceeded` and `EmailDraftOperationFailed` events.

Event changes

This scenario involves the following event changes:

- `WorkItemAdded`, `EmailDraftOperationSucceeded` and `EmailDraftOperationFailed`.

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\command\
SampleClientCommand.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
document\EmailMessageEditPanel.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
document\EmailView.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\
controllers\EmailViewControllerImpl.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\model\
action\UIActionListFactory.java
```

Note:

This scenario is not implemented for the .NET sample client.

Reply to email scenario

This scenario provides the information needed to:

- Create a normal reply to a delivered workitem that contains an email document.
- Use the same charset for the reply if a character set exists.
- Use ISO-8859-1 for the reply if no character set exists.

Primary objects used in scenario

This scenario uses the following objects: [EmailDocument](#), [EmailDraft](#), [EmailMessage](#).

Implementation considerations

Before you write your custom code, consider the following:

How do you create a draft to prepopulate the composition window? Which draft type do you need to specify? The application user determines whether the type of draft to be created is available. For example, the application user can click a button or make a selection from a drop-down list. When the type of draft is available, the Client SDK passes the appropriate enumeration value in the `createDraft(EmailDraft.Type type)` method on the `EmailDocument`. This method returns an `EmailDraft` object that has all the necessary headers set. These headers determine the email type when the email is sent to ensure that Avaya IC can appropriately track the work item.

During the creation of the `EmailDraft`, the Client SDK prepopulates certain fields based on the original document, such as the To, Cc, and Subject fields. However, you must set the content. For example, when an application user replies to an email, you can embed the text of the original email in the message body. Because this content might require special formatting, the client developer must set the content. The Client SDK does not provide any special handling of content when the email draft is created.

Which properties need to be set on the EmailDraft? Most of the basic fields need to be set on the `EmailDraft`. The most important property is the character set if the original email does not include a character set. If a character set is available from the original email document, the email draft uses that character set by default. However, you can override this setting.

Important:

You must provide a character set for all `EmailDraft`s. A character set is required for new outbound email drafts, where no default character set can be obtained from an inbound email document with a valid character set.

Which operation sends the email? `EmailDraft.send()` sends out an email.

How do draft types differ from email types? This information is available in the API documentation installed with the Client SDK design files.

High-level steps

This scenario requires the following steps:

1. Get access to the `EmailDraft` object of the appropriate type through the Email Document.
2. Create the composition window, and populate the window with the email draft contents.
3. When the composition is ready to be sent:
 - a. Collect the email data from the composition window.

Appendix A: Sample scenarios

- b. Set the appropriate fields on the EmailDraft, including the charset and content type.
- c. Send the reply.
- d. Check for OperationSucceeded event.

Event changes

This scenario involves the following event changes for the EmailDraft.Send/Cancel/Save operations:

- OperationSucceeded
- OperationFailed

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
document\EmailView.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
document\EmailMessageEditPanel.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\
controllers\EmailViewControllerImpl.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\
Controller\UIController.cs
```

Display WorkItem History scenario

This scenario provides the information needed to retrieve the history of a WorkItem.

Primary objects used in scenario

This scenario uses the following objects: [WorkItem](#), WorkItemHistory

Implementation considerations

Before you write your custom code, consider the following:

Which method do you use to request the history information? Use

`WorkItem.requestHistory()`. This method makes a request on the server side to get the history. When the History is received, a `WorkItem.RequestHistoryResponse` is sent to notify the listeners. You can access a collection of all history objects with the `workItem.getHistory().getAll()` method.

Which mechanism do you use to register for events associated with history updates?

The WorkItem generates a `RequestHistoryResponse` event when the History is received from the server. The following example shows a mechanism to register for these events:

```
session.registerListener( WorkItem.RequestHistoryResponse.TYPE, new
WorkItemHistoryResponseListener ());
```

Note:

The Client SDK uses different mechanisms in Java and .NET to register listeners. For more information, see [Event handling guidelines](#) on page 79.

When should an application retrieve the WorkItem history? Avaya recommends that you retrieve the WorkItem history when a WorkItem arrives in the custom application, or when an application user selects the WorkItem and makes it current.

High-level steps

This scenario requires the following steps:

1. Implement an event listener to handle `WorkItemRequestHistoryResponse` events.
2. When the application user selects a WorkItem, send a request to retrieve the WorkItem history information.
3. Update the display based on the WorkItem history information returned from the `WorkItemRequestHistoryResponse` events.

Event changes

This scenario involves an event change for the `RequestHistoryResponse` event on `WorkItem`.

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\
controllers\SessionControllerImpl.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
pane>ContactHistoryPanel.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
pane\AbstractContactDataPanel.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\Com.Avaya.Ic.Sdk.Sampleclient\  
Controller\UIController.cs  
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\Com.Avaya.Ic.Sdk.Sampleclient\Ui\  
WorkItemDetailsHelper.cs
```

Display Customer History scenario

This scenario provides the information needed to retrieve the history for a Customer.

Primary objects used in scenario

This scenario uses the following objects: [WorkItem](#), CustomerHistory, RecordsList, NameValueCollection.

Implementation considerations

Before you write your custom code, consider the following:

Which method do you use to request the history information? Use

`WorkItem.requestCustomerHistory(CustomerHistory.QueryCriteria criteria)`. This method makes a request on the server side to get the history. When the History is received from the server, `WorkItem.RequestCustomerHistoryResponse` event will be sent to notify the listeners. You can access a collection of all history objects with the `workItem.getCustomerHistory().getAll()` method.

Which mechanism do you use to register for events associated with history updates?

The `WorkItem` generates a `RequestCustomerHistoryResponse` event when the History is received from the server. The following example shows a mechanism to register for these events:

```
session.registerListener( WorkItem.RequestCustomerHistoryResponse.TYPE, new  
CustomerHistoryResponseListener());
```

Note:

The Client SDK uses different mechanisms in Java and .NET to register listeners. For more information, see [Event handling guidelines](#) on page 79.

When should an application retrieve the Customer history? Avaya recommends that a custom application provide Customer history information only on demand.

When an application user selects a WorkItem and makes it current, your custom application should retrieve a summary of the CustomerHistory record and populate the user interface with that summary.

When the application user double clicks on an item in that summary list, your custom application should retrieve the details of that record. For example, your custom application can retrieve WrapupRecords, WorkItemHistory and other additional data as applicable for the CustomerHistoryRecord from the Client SDK server.

High-level steps

This scenario requires the following steps:

1. Implement event listeners to handle the following events:
 - `WorkItemRequestCustomerHistoryResponse`
 - `CustomerHistoryRequestWrapupRecordsResponse`
 - `CustomerHistoryRequestWorkItemHistoryResponse`
 - `ChatMediaInteractionRecordRequestTranscriptResponse`
 - `EmailMessageRecordRequestEmailMessageResponse`
2. Send a request to retrieve the customer history information when a Workitem is selected.
3. Update the display based on the customer history information returned from the events,

Event changes

This scenario involves an event change for the `RequestCustomerHistoryResponse` event on `WorkItem`.

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\
controllers\SessionControllerImpl.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
pane\CustomerHistoryPanel.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
pane\AbstractContactDataPanel.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\Com.Avaya.Ic.Sdk.Sampleclient\
Controller\UIController.cs
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\Com.Avaya.Ic.Sdk.Sampleclient\Ui\
WorkItemDetailsHelper.cs
```

AddressBook scenario

This scenario provides the information needed to retrieve addressable agents and queues for the Address Book in a custom application.

Primary objects used in scenario

This scenario uses the following objects: [Session](#), AddressBook, AddressBookAgentQuery, AddressBookQueueQuery, AddressBookEntry.

Implementation considerations

Before you write your custom code, consider the following:

How can you get the AddressBook object? You can get the AddressBook object from the Session object with the `Session.getAddressBook()` method.

How can you get the addressable agents? Make a request to get agents with the `AddressBook.findAgents(AddressBookAgentQuery query)` method. When the information is returned, the `AddressBook.FindAgentsResponse` event is sent to notify the listeners. You can access a collection of all returned addressable agents with the `AddressBook.FindAgentsResponse.getAgentRecords().getAll()` method.

How can you get the addressable queues? Make a request to get queues with the `AddressBook.findQueues(AddressBookQueueQuery query)` method. When the information is returned, the `AddressBook.FindQueuesResponse` event is sent to notify the listeners. You can access a collection of all returned addressable queues with the `AddressBook.FindQueuesResponse.getQueueRecords().getAll()` method.

Which mechanism do you use to register for events associated with addressable entry updates? The AddressBook generates a `FindAgentsResponse` event when the agent type response is received or a `FindQueuesResponse` event when the queue type response is received from the server. The following example shows a mechanism to register for these events:

```
session.registerListener( AddressBook.FindAgentsResponse.TYPE, new
FindAgentsResponseListener());
session.registerListener( AddressBook.FindQueuesResponse.TYPE, new
FindQueuesResponseListener());
```

High-level steps

This scenario requires the following steps:

1. Retrieve the AddressBook object from the Session object.
2. Register for the following events:
 - AddressBook.FindAgentsResponse
 - AddressBook.FindQueuesResponse
3. Implement event listeners to handle AddressBook events.
4. Make requests to get information about the addressable agents and queues.
5. Update the user interface based on the AddressBook events.

Event changes

This scenario involves event changes for the `FindAgentsResponse` and `FindQueuesResponse` events on `AddressBook`.

Sample code

Location of code in Java sample client

```

IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\
controllers\SessionControllerImpl.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
uad\SimpleUADDialog.java

```

Location of code in .NET sample client

```

IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\Com.Avaya.Ic.Sdk.Sampleclient\
Controller\UIController.cs
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\Com.Avaya.Ic.Sdk.Sampleclient\Ui\
frmAddressBook.cs

```

Retrieving Workitem Contact Attributes scenario

This scenario provides information about configuring the SDK to monitor the `contact_attr` container.

Primary objects used in scenario

This scenario uses the following objects: `WorkItem`

Implementation Considerations

Before you write your custom code, consider the following:

What Edu fields needs to be monitored? Monitor only those EDU fields which are in your business requirement. This way SDK server can fetch only the required fields from EDU container.

Will you be notified with an event in case the fields are modified? Yes, SDK server raises `ContactAttributeChanged` event for each field that you have added or modified.

High-level Steps

This scenario requires following steps:

1. Add the following line to the `EDUFieldsToCache.properties` located at `...\Avaya\IC72\sdk\server\icsdk\custom\config` directory
`all_1=contact_attr.*`
2. Copy the `EDUFieldsToCache.properties` file to the following directory:
`...\Avaya\IC72\sdk\server\icsdk\WEB-INF\classes\com\avaya\ic`
3. Ensure that file `SDKWorkItemAttributesFilter.properties` located at the `...\Avaya\IC72\sdk\server\icsdk\custom\config\sdk\` directory contains an entry for `contact_attr.*`.
For example: `field_9=contact_attr.*`
4. Copy the `SDKWorkItemAttributesFilter.properties` file to the following directory:
`...\Avaya\IC72\sdk\server\icsdk\WEB-INF\classes\com\avaya\ic\sdk\customization`
5. Restart the SDK Server to reflect the changes that you have made in both the properties files. The SDK server starts monitoring all changes to EDU fields in the `contact_attr` container.

Event changes

This scenario involves the following event changes:

`ContactAttributesChanged`

Sample code

Location of code in Java sample client:

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\
controllers\WorkListViewControllerImpl.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\  
Controller\UIController.cs
```

Voice Call scenario

This scenario provides the information required for outbound and inbound voice call.

Primary objects used in scenario

This scenario uses the following objects: [Session](#), [VoiceChannel](#), [WorkList](#), [WorkItem](#), [MediaInteractionList](#), [VoiceMediaInteraction](#).

Implementation considerations

Consider the following before you write your custom code:

Which events signify the delivery or removal of workitems and VoiceMediaInteraction?

- WorkList.WorkItemAdded
- WorkList.WorkItemRemoved
- MediaInteractionList.MediaInteractionAdded
- MediaInteractionList.MediaInteractionRemoved

Which events you can consider for voice call?

- WorkItem.StateChanged
- WorkItem.CurrentContextChanged
- WorkItem.OperationSucceeded
- WorkItem.OperationFailed
- VoiceMediaInteraction.AudioSource
- VoiceMediaInteraction.AudioSourceChanged
- VoiceMediaInteraction.DestinationBusy
- VoiceMediaInteraction.DestinationConnected
- VoiceMediaInteraction.Diverted
- VoiceMediaInteraction.OperationFailed

Which objects generate events for voice call?

- WorkList

Appendix A: Sample scenarios

- WorkItem
- MediaInteractionList
- VoiceMediaInteraction

How to place outbound voice call? Invoke the `makeCall()` API with required parameters on `VoiceChannel` object.

High-level steps

Outbound voice call requires the following steps:

1. Register listeners and create handlers for the appropriate `WorkList`, `MediaInteractionList` and `VoiceMediaInteraction` object's events.
2. Access the `VoiceChannel` object through `Session` object.
3. Invoke `makeCall()` API on the `VoiceChannel` object. On success, SDK server raises the `WorkItemAdded` event of `WorkList` object.
4. Retrieve `WorkItem` object from `WorkItemAdded` event and update the user interface accordingly.
5. SDK server raises `MediaInteractionAdded` event when voice media is added to a `workitem`.
6. Retrieve `VoiceMediaInteraction` from `MediaInteractionList.MediaInteractionAdded` event and update the user interface accordingly.

Note:

In Avaya IC 7.2, apart from the `WorkItemAdded` and `WorkItemRemoved` events, two new events, `MediaInteractionAdded` and `MediaInteractionRemoved` are raised.

The `MediaInteractionAdded` event is raised when a media is added to a `workitem` and the `MediaInteractionRemoved` event is raised when a media is removed from the `workitem mediainteraction` list.

These events are applicable even for single interaction.

For steps for handling inbound voice call, see [WorkItem object scenarios](#) on page 162.

Appendix B: Additional sample scenarios

This section includes some additional high-level descriptions of scenarios that you might want to consider including in your custom application. These scenarios do not contain the same detailed descriptions provided in [Sample scenarios](#) on page 125.



Important:

Some of these scenarios might not be fully implemented in the sample clients.

This section includes the following topics.

- [Application object scenario](#) on page 155
- [Session object scenarios](#) on page 156
- [Channel object scenario](#) on page 161
- [WorkItem object scenarios](#) on page 162
- [Voice interaction scenario](#) on page 166
- [Chat interaction scenarios](#) on page 167
- [Email document scenarios](#) on page 169
- [Wrapup scenarios](#) on page 171
- [Supervisory scenario](#) on page 173
- [Join-Us Scenario](#) on page 176

Application object scenario

This section includes the following scenario related to logging in to and out of a custom application:

- [Password change scenario](#) on page 156

Password change scenario

The sample code in this scenario shows how to implement a solution that:

- Enables a new application user to change the assigned Avaya IC password on the first login.
- Ensures that subsequent logins use the new password for authentication.

Primary objects used in scenario

This scenario uses the following object: [Application](#)

Implementation considerations

Before you write your custom code, consider the following:

- Have you created an Avaya IC agent account that requires a password change?
- What indicates a password change is needed?
- What are the potential causes of login failure, and how can they be identified?
- Where are the causes documented?
- What is the impact of the Avaya IC rules for changing passwords? Do you need to validate the new password?
- Can you authenticate against external resources, such as LDAP?

High-level steps

This scenario requires the following steps:

1. Create an instance of the Application object.
2. Log in to the application.
3. On failure, determine a reason and, based on that, prompt the application user for a new password.
4. For subsequent logins, use the new password to log in to the application, as described in [Login scenario](#) on page 126.

Session object scenarios

This section includes the following scenarios related to session availability:

- [Session status scenario](#) on page 157

- [Connectivity status scenario](#) on page 158
- [Session shutdown request scenario](#) on page 159
- [Enable and disable operational state scenario](#) on page 160

Session status scenario

This scenario provides the information needed to:

- Specify how your custom application reacts to Session availability.
- Warn the custom application that the Session is unavailable.
- Display a warning in the status bar on unavailable status.
- Display a dialog box for session failure before a forced shutdown.
- Appropriately disable application operations.

Primary objects used in scenario

This scenario uses the following objects: [Application](#), [Session](#).

Implementation considerations

Before you write your custom code, consider the following:

- What behavior is allowed or not allowed during session unavailability and failure?
- What must happen if the custom application does not exit during session failure?
- How will you notify about session unavailability?
- Which user interface changes must occur on session unavailability? For example, should the code disable the Agent Available button?

High-level steps

This scenario requires the following steps:

1. Access the Session object.
2. Enable listeners for session status change events.
3. On status changes:
 - a. Disable the user interface on unavailability.
 - b. Force a logout on failure.

Connectivity status scenario

This scenario provides the information needed to ensure that your custom application can handle network connectivity failures.

To test your custom application, remove the network cable, and verify that:

- The status bar indicates that the connection was broken.
- If the network cable is reconnected within 30 seconds, or a specified length of time:
 - Your application displays a dialog box that notifies the application user that the application needs to be shut down.
 - When the application user selects **OK** in the dialog box, the application shuts down.

Primary objects used in scenario

This scenario uses the following objects: [Application](#), [Session](#)

Implementation considerations

Before you write your custom code, consider the following:

- What mechanism do you use to notify the application of connectivity status, such as impaired and failed?
- Which application interface changes must occur on connectivity status changes? For example, should the code display messages in the status bar, change connectivity icons, or disable interface elements?
- Is the threshold configurable? Where is this specified?
- What will happen if the custom application does not shut down? How will the Client API react? Will the Client API force a shutdown?

High-level steps

This scenario requires the following steps:

1. Access the Session object.
2. Register Connectivity status event listeners.
3. Listen to the Connectivity Status changed events.
4. Take appropriate actions to a Connectivity Status changed event.

Session shutdown request scenario

This scenario provides the information needed to ensure that your custom application shuts down gracefully during a server-driven forced shutdown request. When a shutdown request is received:

- Your application displays a dialog box that notifies the application user about the reasons that the application needs to be shut down.
- When the application user selects **OK** in the dialog box, force a graceful shutdown.

Primary objects used in scenario

This scenario uses the following objects: [Session](#), [WorkItem](#), [Application](#).

Implementation considerations

Before you write your custom code, consider the following:

- Which events and shutdown reasons must initiate a shutdown request?
- How will you force the application user to complete certain ongoing tasks before shutdown?
- How will you initiate the shutdown?
- How can you ensure that your session does not time-out?

High-level steps

This scenario requires the following steps:

1. Register for the event that indicates forced logout.
2. After the event is received:
 - a. Determine the reason for the shutdown.
 - b. Based on the reason, your custom application must take the necessary steps before a logout. For example, automatically complete work items, or disable the Agent Available button.
3. Use the Application object to initiate a logout.

Enable and disable operational state scenario

This scenario provides the information needed to implement the user interface to ensure that the button states correctly reflect the state of the relevant operations.

Primary objects used in scenario

This scenario uses the following object: [Session](#)

Implementation considerations

Before you write your custom code, consider the following:

- What are the dependencies of the different operations on the items exposed by the Session object, such as state, status, and properties?
- What support does the Client SDK provide if an incorrect operation is invoked? For example, OperationFailed events.

For more information, see [Guidelines for using the Client API](#) on page 67.

High-level steps

This scenario requires the following steps for each button on the user interface:

- Enable when the operation that the button is expected to invoke is possible, based on operational state.
ELSE
- Disable to prevent illegal operations from being invoked.

Channel object scenario

This section includes the following scenario related to channels:

- [Enable and disable channel operational state scenario](#) on page 161

Enable and disable channel operational state scenario

This scenario provides the information needed to implement the user interface to ensure that the button states correctly reflect the state of the relevant channel operations.

Primary objects used in scenario

This scenario uses the following objects: [Channel](#), [VoiceChannel](#), [EmailChannel](#), [ChatChannel](#).

Implementation considerations

Before you write your custom code, consider the following:

- What are the dependencies of the different operations on the items exposed by the Channel objects, such as state, status, and properties?
- Why are the chat and email channels linked? How should your custom code handle them? For example, can you make an application user available for Email only?
- What support does the Client SDK provide if an incorrect operation is invoked? For example, OperationFailed events.
- What is the purpose of the Reset method on the channels? How do they work? Do you need to make this method available to the application user?

High-level steps

This scenario requires the following steps for each button on the user interface:

- Enable when the operation that the button is expected to invoke is possible, based on operational state.
ELSE
- Disable to prevent illegal operations from being invoked.

WorkItem object scenarios

This section includes the following scenarios related to work items:

- [Display assigned work items scenario](#) on page 162
- [Prompt on WorkItem arrival scenario](#) on page 163
- [Enable and disable work item operational state scenario](#) on page 165
- [Access work item attributes scenario](#) on page 165

Display assigned work items scenario

This scenario provides the information needed to:

- Update the list widget in the user interface with details of the assigned work item. For example, have the list widget display one or more of the following for each work item:
 - "Current" status of the work item
 - State of the work item
 - Type of media that the work item contains: voice, email, or chat
 - Origin of the work item
 - State of the media interaction: connected or disconnected
 - Topic for escalation
- Update the Channel user interface to display the total number of media interactions and documents.

Primary objects used in scenario

This scenario uses the following objects: [WorkItem](#), [MediaInteraction](#), [WorkList](#), [MediaInteractionList](#), [DocumentList](#).

Implementation considerations

Before you write your custom code, consider the following:

- Which events signify the delivery of work items?
- Which event handlers and actions update the list widget?
- Which methods return the specified data?
- What user interface elements, such as icons, do you need to represent various states?
- Which objects do you need to listen to events?

- How is being Current different from the Working state of a work item? Why does a work item need to be current?

For more information about the Current concept for work items, see [Work items and the Current concept](#) on page 23.

High-level steps

This scenario requires the following steps:

1. Register listeners for the appropriate WorkList events.
2. Create handlers to receive events.
3. Update the user interface to reflect work item delivery.

Note:

In Avaya IC 7.2, apart from the `WorkItemAdded` and `WorkItemRemoved` events, two new events are raised. The `MediaInteractionAdded` event, which is raised when a media is added to a workitem, and the `MediaInteractionRemoved` event, which is raised when a media is removed from the workitem `mediainteraction` list. These events are applicable even for single interaction.

You must add appropriate listeners for both these events and update the UI accordingly.

Prompt on WorkItem arrival scenario

This scenario provides the information needed to:

- Using the `PromptOnArrival` property, display a prompt to the application user that asks if the application user wants to accept the work item.
- If the application user selects:
 - Yes, the application accept the work item.
 - No, the application declines the work item only if `AutoAccept` is set to `False` for the associated channel.

Note:

SDK does not support declining incoming voice calls.

Appendix B: Additional sample scenarios

Primary objects used in scenario

This scenario uses the following objects: [Session](#), [WorkItem](#), [MediaInteraction](#), [WorkList](#), [MediaInteractionList](#), [DocumentList](#).

Avaya IC properties used in scenario

This scenario uses the following properties:

- PromptOnArrival
- AutoAccept

Implementation considerations

Before you write your custom code, consider the following:

- Which operations allow you to accept or decline a work item?
- Which events signify successful accept or decline?
- Which actions need to be taken if the operation is successful?
- Do any other Avaya IC properties, such as RONA, play a role in the dialog behavior?

High-level steps

This scenario requires the following steps:

1. Implement the Accept/Decline dialog box.
2. Use Avaya IC properties to determine behavior.
3. Hook in appropriate operations to the button clicks.
4. Implement event handlers.
5. Update the user interface accordingly.

Event changes

This scenario involves at least one of the following event changes for the objects:

- Worklist.workitemRemoved
- Wrapped state with TerminateReason
- workitem.stateChanged away from alerting

Enable and disable work item operational state scenario

This scenario provides the information needed to ensure that the button states correctly reflect the state of the relevant work item operations.

Primary objects used in scenario

This scenario uses the following objects: [WorkItem](#), [MediaInteraction](#), [Document](#).

Implementation considerations

Before you write your custom code, consider the following:

- What are the dependencies of the different operations on the items exposed by the WorkItem object, such as state, status, and properties?
- Does a workitem being current impacts workitem operations?

For more information about the Current concept for work items, see [Work items and the Current concept](#) on page 23.

- What support does the Client SDK provide if an incorrect operation is invoked? For example, OperationFailed events.

High-level steps

This scenario requires the following steps for each button on the user interface:

- Enable when the operation that the button is expected to invoke is possible, based on operational state.
ELSE
- Disable to prevent illegal operations from being invoked.

Access work item attributes scenario

This scenario provides the information needed to:

- Display work item attributes in the attributes viewer.
- Set up the customization so that the attributes viewer displays two custom fields in the EDU that start with contact_attr.

Primary objects used in scenario

This scenario uses the following objects: [WorkItem](#).

Implementation considerations

Before you write your custom code, consider the following:

- Which methods do you use to access the attributes?
- Which user interface widget displays the attributes?
- How are attributes set through the Client SDK?
- Does the custom application get an event when an attribute is changed?
- What do the attributes represent? Where does the data come from?

For more information, see [Attributes on the WorkItem object](#) on page 50.

High-level steps

This scenario requires the following steps:

1. Access the attributes list through the Client API call.
2. Iterate through the attributes to display the key value pairs.

Voice interaction scenario

This section includes the following scenario related to voice interactions:

- [OnHold alert on threshold scenario](#) on page 166

OnHold alert on threshold scenario

This scenario provides the information needed to implement a solution where:

- An agent is alerted whenever a voice interaction was placed on hold for longer than a configured threshold.
- Reconnecting clears the alert.

Primary objects used in scenario

This scenario uses the following object: [VoiceMediaInteraction](#)

Implementation considerations

Before you write your custom code, consider the following:

- Which operations allow your custom application to place a voice interaction on hold?
- Which event notifies that the voice interaction is on hold or reconnected?

- Which timers are needed?
- Which user interface elements do the event handlers need to update?

High-level steps

This scenario requires the following steps:

1. Provide a mechanism to invoke the appropriate methods.
2. Implement handlers that react to state change events.
3. Update the user interface based on the state change events.

Chat interaction scenarios

This section includes the following scenarios related to chat interactions:

- [Inactivity alert on threshold scenario](#) on page 167
- [Language filter scenario](#) on page 168
- [Customer-generated alert scenario](#) on page 169

Inactivity alert on threshold scenario

This scenario provides the information needed to notify an application user if:

- A customer has sent a chat message, but the application user has not responded in a certain configurable threshold.
- The customer was idle for a certain configurable threshold.

Primary objects used in scenario

This scenario uses the following object: [ChatMediaInteraction](#).

Implementation considerations

Before you write your custom code, consider the following:

- Which event notifies that the transcript is added?
- What determines that the line added from Customer or Agent?
- What timers are needed?
- Which user interface elements do the event handlers need to update?

Appendix B: Additional sample scenarios

High-level steps

This scenario requires the following steps:

1. Implement handlers that react to TranscriptLineAdded events.
2. Start a timer.
3. Update the user interface based when timer expires.
4. Reset the timer.

Language filter scenario

This scenario provides the information needed to:

- Pass all text sent by an application user through a language filter, where foul language is replaced with "#" before the text is sent to the customer.
- Use a foul language word list that is customizable and can be read from a file.

Primary objects used in scenario

This scenario uses the following object: [ChatMediaInteraction](#).

Implementation considerations

Before you write your custom code, consider the following:

- How do you use a file I/O, word-list file, and in-memory caching at startup?
- How do you filter the implementation to parse and compare content with the word list prior to sending the message through the Client SDK?

High-level steps

This scenario requires the following steps:

1. Provide a mechanism to invoke the appropriate methods.
2. Implement handlers that react to state change events.
3. Update the user interface based on the state change events.

Customer-generated alert scenario

This scenario provides the information needed to give a customer the ability to dynamically force the attention of an application user by:

- Causing the user interface to pop-up to the front on the agent desktop.
- Using color signals to indicate that the customer has requested urgent help.

Primary objects used in scenario

This scenario uses the following object: [ChatMediaInteraction](#).

Implementation considerations

Before you write your custom code, consider the following:

- How do you create a text protocol that can be encoded within the message exchange?
- Which user interface elements do the event handlers need to update?
- Which classes need to listen to the transcript lines to act on the encoded protocol?

High-level steps

This scenario requires the following steps:

1. Add listeners for the transcriptLineAdded event.
2. Implement the protocol decoder on receipt of the message.
3. Based on the meta-message, update the user interface.

Email document scenarios

This section includes the following scenarios related to email documents:

- [Apply signatures scenario](#) on page 169
- [Support for attachments scenario](#) on page 170

Apply signatures scenario

This scenario provides the information needed to:

- Create a signature file that contains a signature named signature.txt.

Appendix B: Additional sample scenarios

- Append the signature to each reply sent out by the application user.

Primary objects used in scenario

This scenario uses the following objects: [EmailDocument](#), [EmailDraft](#), EmailMessage.

Implementation considerations

Before you write your custom code, consider the following:

- How do you create the draft? Which user interface elements should you populate with the draft data?
- How do you use I/O to read from the signature file?
- Which event sends the composition?

High-level steps

This scenario requires the following steps:

1. Get access to the EmailDraft object of the appropriate type through the Email Document.
2. Create the composition window, and populate the window with the email draft contents.
3. When the composition is ready to be sent:
 - a. Collect the email data from the composition window.
 - b. Set the appropriate fields on the EmailDraft, including the charset and content type.
4. Append the signature to the end of the content.
5. Send the email.

Support for attachments scenario

This scenario provides the information needed to provide attachment support for:

- Viewing attachments.
- Sending attachments during a composition.

Primary objects used in scenario

This scenario uses the following objects: [EmailDocument](#), [EmailDraft](#), EmailAttachment

Implementation considerations

Before you write your custom code, consider the following:

- How do you access the EmailAttachment object?
- How do you use the attachment ID to identify an attachment?

- How must your application react to attachment-related events?
- Which synchronous operations are involved?

High-level steps

This scenario requires the following steps:

1. Get access to the EmailDraft object of the appropriate type through the Email Document.
2. Create the composition window, and populate the window with the email draft contents.
3. When the composition is ready to be sent:
 - a. Collect the email data from the composition window.
 - b. Set the appropriate fields on the EmailDraft, including the charset and content type.
4. Add the attachment to the email.
5. Send the email.

Wrapup scenarios

This section includes the following scenarios related to wrapup:

- [Access wrapup codes scenario](#) on page 171
- [Wrapup dialog box scenario](#) on page 172
- [Use terminate reasons scenario](#) on page 173

Access wrapup codes scenario

This scenario provides the information needed to implement a solution in which an application user can select and set wrapup codes before completion of a WorkItem.

Primary objects used in scenario

This scenario uses the following objects: [Session](#), [WorkItem](#), WrapupSelection, CategoryCodesList, CategoryCode, ReasonCode, OutcomeCode, CodesList.

Implementation considerations

Before you write your custom code, consider the following:

- Have wrapup codes been configured in Avaya IC?
- How do you access the wrapup codes with the Client API?

Appendix B: Additional sample scenarios

- How do you use the Client API to specify a wrapup code?

High-level steps

This scenario requires the following steps:

1. Create an application user interface for wrapup.
2. Implement a handler for the WorkItem state change event.
3. Listen to the Wrapup state change event.
4. Implement the code that populates the user interface, as appropriate.

Wrapup dialog box scenario

This scenario provides the information needed to display a wizard dialog box when a work item is released, as follows:

- The first page of the wizard displays the category codes.
- The second page of the wizard displays the relevant reason codes.
- The third page of the wizard displays the relevant outcome codes.

Primary objects used in scenario

This scenario uses the following objects: [Session](#), [WorkItem](#), WrapupSelection, CategoryCodesList, CategoryCode, ReasonCode, OutcomeCode, CodesList.

Implementation considerations

Before you write your custom code, consider the following:

- Have wrapup codes been configured in Avaya IC?
- How do you access the wrapup codes with the Client API?
- How do you use the Client API to specify a wrapup code?

High-level steps

This scenario requires the following steps:

1. Create the wizard dialog box.
2. Implement a handler for the WorkItem state change event.
3. Implement the code that populates the user interface, as appropriate.

Use terminate reasons scenario

This scenario provides the information needed to ensure that the normal wrapup mechanism is not invoked when terminate reason is not Normal.

Primary objects used in scenario

This scenario uses the following objects: [Session](#), [WorkItem](#)

Implementation considerations

Before you write your custom code, consider the following:

- Is this mandatory?
- What terminate reasons do you want to use?
- Where do you define the terminate reasons?

High-level steps

This scenario requires the following steps:

1. Implement a handler for the WorkItem state change event.
2. Check the reason code that accompanies this event.
3. Call complete on WorkItem when the reason code is not Normal.

Supervisory scenario

This scenario provides information required for supervisory operations. In this scenario Supervisor can perform the following operations:

- Begin and END session
- Monitor Chat
- Changing Supervisor Visibility

Primary objects used in scenario

This scenario uses the following primary objects:

- Session
- User
- Workitem

Appendix B: Additional sample scenarios

- ChatMediaInteraction

Implementation considerations

Before you write your custom code, consider the following:

How to identify the supervisor role? If the logging-in agent has a role of supervisor, you can provide options to begin and end supervisor session. To check the role of agent invoke the `User.isSupervisor()` API.

Which interactions supervisor can monitor? Supervisor can monitor only chat media interactions. Supervisor cannot monitor voice interactions. Email Documents can only be viewed by supervisor.

Which events need to be acted on? Who needs these events? If the logging-in agent is supervisor and invokes `beginSupervising()` or `endSupervising()` API on the `Session` object, SDK server raises `SupervisorSessionStatusChanged` event indicating session began or ended. This event is primarily required to an agent who is logging in as supervisor so that custom application can act and change the display accordingly, if required.

Will the agents (belonging to this supervisor monitoring workgroup) be notified when supervisor begins or ends session? Yes, SDK server raises the `SupervisorStatusChanged` event stating Supervisor is Online or Offline.

How will agent information be notified to supervisor? After the supervisor begins a supervising session, SDK server raises the `AgentStatusChanged` event when agent, belonging to this supervisor monitoring workgroup, login or logout. You need to register for this event to get agent information.

How will supervisor monitor agent workitems? Which event will be received? To get workitems handled by agents, you need to invoke the `requestWorkitems()` API on the `Session` object passing the agent ID. Once the `requestWorkitems()` API is invoked on a particular agent you don't need to call that API again. SDK server takes care of getting the workitems for that agent whenever any new workitems gets added or removed for that agent. SDK server raises the `WorkItemAdded` event, similar to when a normal workitem is received to an agent.

Similarly, SDK server raises the `WorkItemRemoved` event whenever agent wraps the workitem.

How to identify the owner of workitem? How to identify that a workitem is supervised workitem? You can invoke the `getOwnerLoginId()` API on the `Workitem` object to identify the agent who is handling this interaction. You can invoke the `isWorkitemSupervised()` API on the `Workitem` object to identify whether a workitem is supervised or not.

What all operations are not allowed for supervisor? Supervisor cannot perform following operations on supervised workitems.

- accept

- release
- complete
- decline
- collaborationBegin
- collaborationCancel
- collaborationComplete
- transfer
- makecurrent
- defer
- setJoinUsHandle on ChatMediaInteraction

How will supervisor monitor Chat interactions? To monitor a chat session, supervisor has to invoke the `monitor()` API on the `ChatMediaInteraction` object. On Invoking this API, SDK Server joins the supervisor to chat room but in an invisible mode. To become visible to chat room, supervisor has to invoke `setVisibility()` API on the `ChatMediaInteraction` object.

High-level steps

This scenario requires the following steps:

1. Access the Session object and get the user object.
2. Check the role of an agent. If role is supervisor, provide an option to begin or end the supervisor session.
3. Listen for `SupervisorSessionStatusChanged` and `AgentStatusChanged` events to get agent information.
4. To monitor workitems handled by a particular agent, invoke `requestWorkitems()` API which triggers SDK server to retrieve supervised workitems by raising `WorkitemAdded` event.
5. Listen to this event, retrieve the owner of this workitem by invoking the `getOwnerloginId()` API.
6. If workitem has chat media interaction and needs to be monitored, invoke the `monitor()` API on the `ChatMediaInteraction` object. You will start receiving transcript events for this chat media.
7. In order to make supervisor visible or invisible to a chat room, invoke the `setVisibility()` API on the `ChatMediaInteraction` object.

Event changes

The following events are involved in this scenario:

1. The `Session.SupervisorSessionStatusChanged` event raised for agent whose role is supervisor, whenever supervisor do begin or end supervisor session.
2. The `Session.SupervisorStatusChanged` event raised for agents belonging to this supervisor monitored workgroup, if supervisor is logged in/out.
3. The `Session.AgentStatusChanged` event raised when agent logs in or logs out.
4. The `ChatMediaInteraction.MonitoringStatusChanged` event raised when supervisor starts/ends monitoring chat interaction.
5. The `ChatMediaInteraction.VisibilityModeChanged` event raised when supervisor becomes visible/invisible to chat room.

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
core\SupervisorPanel.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
core\WorklistViewControllerImpl.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\components\
core>MainToolBarPanel.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\command\
SampleClientCommand.java
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ ui\model\
action\UIActionListFactory.java
IC_INSTALL_DIR\IC72\sdk\sample\src\com\avaya\ic\sdk\sample\ui\model\action\helper\
AgentActionHelperFactory.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\
Controller\UIController.cs
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\Ui \
frmMain.cs
```

Join-Us Scenario

You may occasionally have a customer who would like to invite one or more people to join the chat session. This type of session is called a Join-Us Conference.

Before initiating a Join-Us Conference, create a Join-Us handle (user name) and send the handle with a join-us URL to the person who is going to enter the Join-Us session.

This scenario provides information about setting the Join-Us handle.

Primary objects used in scenario

This scenario uses the `ChatMediaInteraction` object.

Implementation considerations

Before you write your custom code, consider the following:

Which events indicate a success or failure of setting JoinUs Handle? Client SDK raises the `TranscriptLineAdded` event that provides information about success. In case of failure, you will get `OperationFailed` event with reason.

Can Supervisor set Join-Us handle? No, this operation is not supported for Supervisor. The `OperationFailed` event is raised if supervisor performs this operation.

Under what circumstances can set Join-Us handle fails? Under the following situation, the set Join-Us handle fails and the `OperationFailed` event is raised:

- If the chat session doesn't exist
- If the join-us handle already exist for another chat session. It is recommended to use the system generated handle to avoid duplication.

High-level steps

- Register Listener for `TranscriptLineAdded` event and create a handler to receive this event.
- Provide a mechanism to create a Join-Us handle and invoke the `setJoinUsHandle()` API on the `ChatMediaInteraction` object.

Appendix B: Additional sample scenarios

Sample code

Location of code in Java sample client

```
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\command\  
SampleClientCommand.java  
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\  
components\chat\ChatView.java  
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\  
components\uad\JoinUsDialog.java  
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ui\  
controllers\WorklistViewControllerImpl.java  
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ ui\model\  
action\helper\ActionHelper.java  
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ ui\model\  
action\helper\ChatActionHelperFactory.java  
IC_INSTALL_DIR\IC72\sdk\design\java\sample\src\com\avaya\ic\sdk\sample\ ui\model\  
action\UIActionListFactory.java
```

Location of code in .NET sample client

```
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\  
Controller\UIController.cs  
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\Ui\  
frmMain.cs  
IC_INSTALL_DIR\IC72\sdk\design\dotnet\sample\src\Com.Avaya.Ic.Sdk.Sampleclient\Ui\  
frmJoinUs.cs
```

Appendix C: Error messages

The Client SDK error messages include error codes and error strings. Each error message has a MajorCode and a MinorCode.

Client SDK users receive these errors as part of OperationFailed events.

This section includes the following topics:

- [MajorCodes](#) on page 179
- [MinorCodes](#) on page 180

MajorCodes

The MajorCode describes the high level area to which the error belongs.

Error code	Error string
0	"UNKNOWN"
100	"SDK_BRIDGE"
200	"UOM"
300	"VESP"
400	"IC"

MinorCodes

The MinorCode provides specific information about an error.

This section includes the following topics:

- [General or common error codes](#) on page 180
- [Session error codes](#) on page 180
- [Common channel error codes](#) on page 181
- [Email Channel error codes](#) on page 181
- [WorkItem error codes](#) on page 181
- [EmailDraft error codes](#) on page 181
- [Common media interaction error codes](#) on page 182
- [Voice Media interaction error codes](#) on page 182

General or common error codes

Error code	Error string
0	"UNKNOWN"
1	"Unable to retrieve corresponding UOM object"
2	"Operation currently not allowed on this object"
3	"Invalid destination passed"
4	"Invalid destination type passed"

Session error codes

Error code	Error string
101	"Invalid attribute name passed - should start with " + <"custom"> Note: The string "custom" is the name of the EDU container.
102	"Session object not found"
103	"AuxReasonCodes not found"
104	"AuxReasonCodes not found"

Common channel error codes

Error code	Error string
202	"Channel object not found"

Email Channel error codes

Error code	Error string
231	"Invalid To Address passed"

WorkItem error codes

Error code	Error string
301	"Invalid collaboration intent passed - null or invalid"
302	"Invalid media interaction passed for collaboration"
303	"Collaboration may be in progress - intent from previous collaboration not cleared - can't honout new request"
304	"Invalid wrapup selection list passed"
305	"Invalid attribute name passed - should start with " + + <"custom"> Note: The string "custom" is the name of the EDU container.
306	"Collaboration intent was not set"
307	"Media interaction was not set"
308	"Invalid release reason passed"
309	"Workitem object not found"

EmailDraft error codes

Error code	Error string
401	"Operation not supported on this draft"
402	"Invalid attachment id passed"

Appendix C: Error messages

Error code	Error string
403	"Attachment id not found"
404	"Invalid pool information passed - null or invalid"

Common media interaction error codes

Error code	Error string
501	"media interaction object not found"

Voice Media interaction error codes

Error code	Error string
531	"Invalid digits passed"

Index

Symbols

.NET	
compilers	99
diagnostic API	115
libraries	12
socket exceptions.	120
.NET client	
about	61
log4Net	104
logs	104
open Diagnostic Viewer	117
setting log levels	105
.NET Framework.	99

A

accept()	133
access work item attributes scenario	165
access wrapup codes scenario	171
Active state	44 , 47
active state	35
active work items	23
AddressBook	76
event changes	151
implementation considerations.	150
object	150
objects	150
sample code	151
scenario	150
steps.	151
AddressBook.FindAgentsResponse	150
AddressBook.FindQueuesResponse	150
AddressBookAgentQuery	150
AddressBookEntry	150
AddressBookQueueQuery	150
ADU	
access data	30
customizing attributes	91
AES.	12
agent availability	
event changes	130
implementation considerations.	129
object	129
sample code	131
scenario	129
steps.	130
agent properties, using	92

Alerting state	40
API	
AddressBook	76 , 78
diagnostic.	115
History	73
see Client API	
Application	
connectivity status	158
log in	126
log out	128
object.	29
password change	156
Session shutdown	159
Session status	157
Application.Login	80
ApplicationFactory	126
apply signatures scenario	169
architecture	14
ArgumentNullException	87
asynchronous communication	
AES	12
method calls	26
attributes	
about	50
ADU	91
contact_attr	50
ContactAttribute functions	50
data integration	51
EDU	91
functions	51
saving data	51
Session.	91
storage in EDU	50
WorkItem	91
AuthenticationException	88
AutoAccept.	164
Auxwork state	32
Available state	32
Avaya DeveloperConnection Program	121

B

basic services	18
best practices	
AddressBook API	76
chat interactions.	68
Client API.	67
customization	90

Index

event handling	79
exception handling	87
History API	73
log in and out	86
logging	105
null return values	89
operations	88
performance considerations	93
states	85
time and date duration	72
TranscriptLine events	68
voice interactions	69
WrapupSelection API	78
blocking operations	83
busy state	35

C

CategoryCode	171 , 172
CategoryCodesList	171 , 172
changing log4j file name	102
Channel	
active state	35
busy state	35
display properties	131
enable and disable	161
idle state	35
object	33
occupied state	35
operational state scenario	161
states	34
Channel.HealthStatusChanged	132
character sets, customizing	92
chat	
debugging delivery	118
log in and out	87
unsupported methods	68
chat interaction guidelines	68
ChatChannel	36 , 131 , 161
ChatInteraction	
see ChatMediaInteraction	
ChatMediaInteraction	
about	45
active state	47
completed state	47
delivered state	47
disconnected state	47
inactive state	47
scenarios	139 , 167 , 168 , 169
states	47
ChatMediaInteraction.getTranscript()	140
checking	
object states	85
WorkItem status	86
Client API	
about	15
documentation	10 , 20
guidelines	67
model diagrams	19
server objects	17
client messaging provider	18
Client SDK	
architecture	14
features	11
limitations	12
messaging provider	18
Web container	17
Client SDK server bridge	
see SDK server bridge	
clients	
.NET	61
Client API	15
components	15
custom	15
framework	16
Hierarchical Data Store	16
Java	56
logging	104
client-side	
integration	19
log files	104
logging	104
codes, error messages	180
CodesList	171 , 172
collaboration, work item	136
collaborationCancel	68
collaborationComplete	68 , 137
common problems	117
commons-logging	105
communication	
asynchronous	26
synchronous	26
compilers	
.NET	99
Java	99
complete()	133 , 134
Completed state	41 , 44 , 47
components	
basic services	18
client	15
Client API	15
Client API server objects	17
client framework	16
custom client	15
Hierarchical Data Store	16
messaging provider	18
sample clients	55
SDK server bridge	17
server	17
User Object Model	18

Web container	17
Conferencing state	40
configuration file	92
configuration files for log4	102
configuring Client SDK	90
ConnectionException	88
ConnectionStatusChange	83
connectivity status scenario	158
Consulting state	41
contact_attr	50
ContactAttribute	50
container, EDU	50
createEmailDraft	145
creating listeners	80
CSharpTester.exe.log4net	104
current work item	23
currentness	23
custom application	15
custom client	15
customer-generated alert scenario	169
CustomerHistory	
data provided	74
database queries	75
display	148
fields	74
record	74
retrieving	73
when to retrieve	149
customizing	
ADU attributes	91
agent properties	92
character sets	92
Client SDK	90
deploying configuration file	92
EDU attributes	91
files	90
Session attributes	91
WorkItem attributes	91
wrapup codes	92

D

data	
changes	116
diagnostic	114
elements	116
integration	51
saving	51
updating in HDS	115
viewing	115 , 116
data store, hierarchical	16
database queries, CustomerHistory	75
date	
formatting	75
guideline	72

millisecond value	75
debugging	
.NET socket exceptions	120
common problems	117
communication	118
diagnostic API	115
diagnostic information	114
Diagnostic Viewer	117
error messages	114 , 179
logging	99
work item delivery	118
work item state	119
decline	133
defer()	133
Deferred state	40
Delivered state	44 , 47
DeliveryStatusChanged	132
deliveryStatusChanged	132
description	74
DevConnect	121
diagnostic API	
about	115
using	116
diagnostic information	116
Diagnostic Viewer	
about	16
API	115
changed elements	116
data elements	116
debugging with	117
open	117
view of data	115 , 116
directory, customization	90
Disconnected state	44 , 47
display assigned work items scenario	162
display channel properties	
event changes	132
implementation considerations	131
objects	131
sample code	132
scenario	131
steps	132
display Customer history	
event changes	149
implementation considerations	148
sample code	149
scenario	148
steps	149
display email	
event changes	142
implementation considerations	142
objects	142
sample code	143 , 144
scenario	142
steps	142

Index

display text message	
event changes	141
implementation considerations	140
objects	139
sample code	141
scenario	139
steps	141
display WorkItem history	
event changes	147
implementation considerations	147
objects	146 , 148
sample code	147
scenario	146
steps	147
Document	48 , 165
Document.getAttributes()	91
Document.getCreateDate()	72
Document.getType()	142
documentation	
Client API	10 , 20
object model	19
related	10
state models	19
DocumentList	162 , 164
DraftDocument	49
duration guideline	72

E

EDU	
contact_attr	50
customizing attributes	91
saving attribute data	51
WorkItem attributes	50
email	
debugging delivery	118
log in and out	87
EmailAttachment	170
EmailChannel	36 , 131 , 161
EmailDocument	48 , 142 , 145 , 170
EmailDraft	49 , 145 , 170
EmailDraft.send()	145
EmailMessage	142 , 145 , 170
encryption	12
error messages	
about	114 , 179
MajorCode	179
MinorCode	180
event changes	
AddressBook	151
agent availability	130
display channel properties	132
display Customer history	149
display email	142
display text message	141

display WorkItem history	147
log in	127
log out	129
OnHold/OffHold indication	139
reply to email	146
work item collaboration	137
work item lifecycle	135
event handling	
blocking operations	83
ConnectionStatusChange	83
create listeners	80
guidelines	79
register listeners	80
SessionShutdown	83
TranscriptLine	68
events	
AddressBook.FindAgentsResponse	150
AddressBook.FindQueuesResponse	150
FindAgentsResponse	150
FindQueuesResponse	150
RequestHistoryResponse	147
WorkItem.RequestCustomerHistoryResponse	148
examples	
access work item attributes	165
access wrapup codes	171
AddressBook	150
agent availability	129
apply signatures	169
channel operational state	161
connectivity status	158
customer-generated alert	169
display assigned work items	162
display channel properties	131
display Customer history	148
display email	142
display text message	139
display WorkItem history	146
inactivity alert on threshold	167
language filter	168
log in	126
log out	128
OnHold alert on threshold	166
OnHold/OffHold indication	138
operational state	160
password change	156
prompt on WorkItem arrival	163
reply to email	144
saving attribute data	51
session shutdown request	159
Session status	157
support for attachments	170
use terminate reasons	173
voice call	153
work item collaboration	136
work item lifecycle	133

work item operational state	165
wrapup code dialog box	172
exception handling	87

F

features	11
files, customization	90
files, log	107
FindAgentsResponse	150
FindQueuesResponse	150
firewalls	12
flag, current	24
force multiple calls option	70
formatting, date for history	75
framework, client	16
functions	
attributes	51
ContactAttribute	50

G

GetContactAttribute	51
getting support	121
guidelines	
AddressBook API	76
chat	68
chat interaction	68
Client API	67
customization	90
event handling	79
exception handling	87
History API	73
logging	105
logging in	86
logging out	86
null return values	89
operations	88
performance considerations	93
states	85
time and date duration	72
voice interaction	69
WrapupSelection API	78

H

handling	
events	79
TranscriptLine events	68
HDS	
see Hierarchical Data Store	
HealthStatusChanged	132
Hierarchical Data Store	
about	16

diagnostic information	114
Diagnostic Viewer	16
updating	115
History API guidelines	
about	73
CustomerHistory	149
CustomerHistory record	74
formatting date	75
retrieving history	73
WorkItemHistory	147
WorkItemHistory record	74

I

identifying problems	116
idle state	35
implementation considerations	
AddressBook	150
agent availability	129
display channel properties	131
display Customer history	148
display email	142
display text message	140
display WorkItem history	147
log in	126
log out	128
OnHold/OffHold indication	138
reply to email	145
voice call	153
work item collaboration	136
work item lifecycle	133
inactive state	44, 47
inactivity alert on threshold scenario	167
Init_auxwork state	32
Init_available state	32
Initialized state	32
Initiating state	40, 44
integration	
client-side	19
data	51
server-side	19
integrations	
supported	12
internationalization	12
about	123
character sets	92
error messages	179

J

Java	12
commons-logging	105
compilers	99
diagnostic API	115
log4j	105

Index

Java client	
about	56
log to a file	105
logs	105
open Diagnostic Viewer	117
setting log levels	105
JavaDoc	20
JDK	99
join us handle	177

L

language filter scenario	168
lifecycle, work item	133
limitations	
Client SDK	12
sample clients	56
listeners	
creating	80
registering	80
localization	
about	123
character sets	92
error messages	179
location	
customization files	90
log4j configuration files	102
log files	
client-side	104
server	101
log in	
chat	87
ConnectionException	88
email	87
event changes	127
guidelines	86
implementation considerations	126
objects	126
sample code	127 , 176
scenario	126
steps	127
log levels	
.NET client	105
Java client	105
server	103
log out	
chat	87
ConnectionException	88
email	87
event changes	129
guidelines	86
implementation considerations	128
objects	128
sample code	129
scenario	128

steps	129
WorkItem status	87
log4j	
changing file name	102
configuration files	102
server logging	101
log4j.xml.basic	101
log4j.xml.comm	101
log4j.xml.debug	101
log4net logger module	104
Logged_in state	32
Logged_out state	32
loggedout state	35
logging	
.NET client	104
about	99
changing log4j file name	102
client	104
client-side	104
Java client	105
log files	107
log4j	101
module boundaries	100
recommendations	105
sample clients	104
server	101
setting	
.NET client	105
Java client	105
server	103
logs	
Client SDK	107
sample client	104

M

MajorCode	179
makeCurrent()	133
MediaInteraction	41 , 162 , 164 , 165
MediaInteraction.getAttributes()	91
MediaInteractionList	140 , 162 , 164
messages, error	114 , 179
MessageType	140
messaging provider	
client	18
Client SDK	18
server	18
method calls	
about	26
asynchronous	26
synchronous	26
methods, chat	68
millisecond value, date	75
MinorCode	180
module boundaries	100

monitoring chat sessions [175](#)

N

NameValue [74](#)
 NameValueList [74](#), [148](#)
 NAT. [12](#)
 nDoc [20](#)
 Nonviable state [40](#), [44](#)
 null return values [89](#)
 NullPointerException [87](#), [89](#)

O

object model diagram [19](#)
 objects
 Application [29](#)
 Channel [33](#)
 ChatChannel [36](#)
 ChatMediaInteraction [45](#)
 checking state [85](#)
 Document [48](#)
 DraftDocument [49](#)
 EmailChannel [36](#)
 EmailDocument [48](#)
 EmailDraft [49](#)
 history [73](#)
 MediaInteraction [41](#)
 server [17](#)
 Session [30](#)
 User [33](#)
 User Object Model [18](#)
 VoiceChannel [35](#)
 VoiceMediaInteraction [42](#)
 WorkItem [38](#)
 WorkList [37](#)
 occupied state [35](#)
 OnHold alert on threshold scenario [166](#)
 OnHold/OffHold indication
 event changes [139](#)
 implementation considerations [138](#)
 objects [138](#)
 sample code [139](#)
 scenario [138](#)
 steps [139](#)
 opening Diagnostic Viewer [117](#)
 operational state scenario [160](#)
 OperationFailed [89](#)
 operations
 about [26](#)
 blocking [83](#)
 failure [88](#)
 success [88](#)
 OperationSuccess [89](#)
 origin [74](#)

OriginatorHandle [69](#)
 OriginType [140](#)
 OutcomeCode [171](#), [172](#)

P

password change scenario [156](#)
 Paused state [40](#)
 performance considerations [93](#)
 prompt on WorkItem arrival scenario [163](#)
 PromptOnArrival [163](#)
 properties, agent [92](#)
 proxy servers [12](#)

Q

queries, CustomerHistory [75](#)

R

ReasonCode [171](#), [172](#)
 recommendations
 AddressBook API [76](#)
 chat interactions [68](#)
 Client API [67](#)
 customization [90](#)
 event handling [79](#)
 exception handling [87](#)
 History API [73](#)
 log in and out [86](#)
 logging [105](#)
 null return values [89](#)
 operations [88](#)
 performance considerations [93](#)
 states [85](#)
 time and date duration [72](#)
 voice interactions [69](#)
 WrapupSelection API [78](#)
 records
 CustomerHistory [74](#)
 WorkItemHistory [74](#)
 registering listeners [80](#)
 related documentation [10](#)
 release() [133](#)
 reply to email
 event changes [146](#)
 implementation considerations [145](#)
 objects [145](#)
 sample code [146](#)
 scenario [144](#)
 steps [143](#), [145](#)
 RequestHistoryResponse event [147](#)
 retrieving object history [73](#)
 return, null [89](#)

Index

S

sample clients	
.NET	61
about	55
Java	56
logging	104
unsupported features	56
sample code	
AddressBook	151
agent availability	131
create listeners	81
display channel properties	132
display Customer history	149
display email	143 , 144
display text message	141
display WorkItem history	147
log in	127 , 176
log out	129
OnHold/OffHold indication	139
register listeners	80
reply to email	146
work item collaboration	138
work item lifecycle	135
WrapupSelection	79
saving attribute data	51
scalability	12
scenarios	
access work item attributes	165
access wrapup codes	171
AddressBook	150
agent availability	129
Application object	155
apply signatures	169
Channel object	161
channel operational state	161
chat interaction	167
connectivity status	158
customer-generated alert	169
display assigned work items	162
display channel properties	131
display Customer history	148
display email	142
display text message	139
display WorkItem history	146
email document	169
inactivity alert on threshold	167
join us scenario	176
language filter	168
log in	126
log out	128
OnHold alert on threshold	166
OnHold/OffHold indication	138
operational state	160
password change	156
prompt on WorkItem arrival	163
reply to email	144
Session object	156
session shutdown request	159
Session status	157
supervisory scenario	173
support for attachments	170
use terminate reasons	173
voice call	153
voice interaction	166
work item collaboration	136
work item lifecycle	133
work item operational state	165
WorkItem object	162
wrapup	171
wrapup code dialog box	172
SDK client framework	16
SDK server	
components	17
debugging communication	118
SDK server bridge	17
SDKAppContext	93
SDKEducAttributesToFilter.properties	91
SDKICPropertiesSections.properties	92
SDKICPropertiesSections.properties file	133
SDKSessionAttributesFilter.properties	91
SDKSupportedCharsets.properties	92
SDKWorkItemAttributesFilter.properties	91
SDKWrapupCodesCategoryGroups.properties	92
security	12
server	
basic services	18
components	17
log files	101
logging	101
messaging provider	18
SDK server bridge	17
server objects	17
setting log levels	103
User Object Model	18
Web container	17
server bridge, SDK	17
server messaging provider	18
server objects	17
server-side integration	19
Session	
about	30
AddressBook	150
agent availability	129
attributes	91
channel properties	131
connectivity status	158
enable and disable	160
log in	126

prompt on arrival [164](#)
 shutdown request. [159](#)
 states [32](#)
 status [157](#)
 terminate reasons [173](#)
 voice call [153](#)
 wrapup. [171](#), [172](#)
 session shutdown request scenario [159](#)
 Session status scenario [157](#)
 Session.enterAuxwork() [129](#), [132](#)
 Session.getAddressBook() [150](#)
 Session.Initialize [80](#)
 Session.makeAvailable() [129](#), [132](#)
 SessionShutdown [83](#)
 SetContactAttribute [51](#)
 SSL [12](#)
 starttime. [74](#)
 state model
 Channel [34](#)
 ChatMediaInteraction [46](#)
 diagrams [19](#)
 Session [31](#)
 VoiceMediaInteraction [43](#)
 WorkItem [39](#)
 states
 Active [44](#), [47](#)
 active [35](#)
 Alerting [40](#)
 Auxwork [32](#)
 Available [32](#)
 busy [35](#)
 Channel [34](#)
 ChatMediaInteraction [47](#)
 check WorkItem status [86](#)
 checking object. [85](#)
 Completed [41](#), [44](#), [47](#)
 Conferencing [40](#)
 Consulting [41](#)
 Deferred [40](#)
 Delivered. [44](#), [47](#)
 Disconnected. [44](#), [47](#)
 guidelines [85](#)
 idle [35](#)
 Inactive [44](#), [47](#)
 Init_auxwork [32](#)
 Init_available [32](#)
 Initialized. [32](#)
 Initiating [40](#), [44](#)
 Logged_in [32](#)
 Logged_out [32](#)
 loggedout [35](#)
 Nonviable [40](#), [44](#)
 occupied [35](#)
 Paused [40](#)
 Session [32](#)

Transferring [40](#)
 VoiceMediaInteraction [44](#)
 Working [40](#)
 WorkItem [40](#)
 Wrapup [41](#)
 String dumpDiagnosticInfo() [115](#)
 string DumpDiagnosticInfo() [115](#)
 supervisory operations [174](#)
 supervisory role [174](#)
 support for attachments scenario [170](#)
 support, getting. [121](#)
 supported compilers [99](#)
 synchronous communication
 method calls [26](#)
 SSL [12](#)

T

technologies [12](#)
 time guideline [72](#)
 TranscriptDocument [139](#)
 transcriptDocument.getTranscriptLineList() [140](#)
 TranscriptLine [68](#), [140](#)
 TranscriptLine.getOriginatorHandle(). [141](#)
 TranscriptLineAdded [141](#)
 transfer(). [137](#)
 Transferring state. [40](#)
 troubleshooting
 API [114](#)
 common problems. [117](#)
 diagnostic information [114](#)
 error messages [179](#)

U

UOM. [18](#)
 updating Hierarchical Data Store [115](#)
 use terminate reasons scenario [173](#)
 User [33](#)
 User Object Model [18](#)
 using
 agent properties. [92](#)
 diagnostic API. [115](#)
 diagnostic information [116](#)

V

values, null return. [89](#)
 viewing data [115](#), [116](#)
 voice call
 implementation considerations [153](#)
 objects [153](#)
 scenario [153](#)
 steps [154](#)

Index

voice interaction guidelines	69
voice trailing	25
VoiceChannel	35 , 131 , 161
VoiceChannel.Reset	69
VoiceInteraction	
see VoiceMediaInteraction	
VoiceMediaInteraction	
about	42
active state	44
completed state	44
delivered state	44
disconnected state	44
inactive state	44
initiating state.	44
nonviable state	44
scenario	136 , 138 , 166
states	44
VoiceMediaInteraction.State.ACTIVE	139
VoiceMediaInteraction.State.INACTIVE	139

W

Web container	17
WebApplicationContext	93 , 126
work item collaboration	
event changes	137
implementation considerations.	136
objects	136
sample code	138
scenario	136
steps.	137
work item lifecycle	
event changes	135
implementation considerations.	133
objects	133
sample code	135
scenario	133
steps.	134
work item operational state scenario	165
work items	
active	23
current.	23
debugging delivery	118
debugging state	119
voice trailing	25
workflows	51
Working state	40
WorkItem	
about	38
access.	165
alerting state	40
attributes	50
check status	86
collaboration	136
completed state	41

conferencing state	40
consulting state	41
contact_attr container	50
ContactAttribute functions	50
customizing attributes	91
deferred state	40
display assigned	162
display customer history	148
display email	142
display history.	146
enable and disable	165
functions	51
initiating state	40
lifecycle.	133
log out status	87
paused state	40
prompt on arrival	164
session shutdown	159
states.	40
terminate reasons	173
transferring state	40
working state	40
wrapup codes	171
wrapup dialog	172
wrapup state	41
WorkItem.complete()	134
WorkItem.CurrentContextChanged	24
WorkItem.getCreateDate()	72
WorkItem.getDeliveredDate()	72
WorkItem.getQueueTime()	72
WorkItem.getTerminateReason()	134
WorkItem.isCurrent()	136
WorkItem.RequestCustomerHistoryResponse event	148
WorkItem.StateChanged	135
workitem.stateChanged	164
WorkItemHistory	146
about	74
retrieving	73
when to retrieve	147
WorkList	37 , 133 , 162 , 164
Worklist.workitemRemoved	164
wrapup code dialog box scenario	172
wrapup codes, customizing	92
wrapup scenarios.	171
Wrapup state.	41
WrapupEnabled	134
WrapupSelection	78 , 171 , 172
WrapupSelectionList	78