Avaya Proactive Contact 5.0 Software Developer's Kit

Issue 1.1

June 2011

Notice

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

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

Documentation disclaimer

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

Link disclaimer

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

Warranty

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

Copyright

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

Avaya support

Avaya provides a telephone number for you to use to report problems or to ask questions about your product. The support telephone number is 18002422121 in the United States. For additional support telephone numbers, see the Avaya Web site: http://www.avaya.com/support.

Contents

Preface

This section contains the following topics:

Purpose

The purpose of this guide is to provide detailed information about Avaya Proactive Contact.

Audience

This guide is for personnel who develop client applications for Avaya Proactive Contact. The Avaya Proactive Contact SDK provides a CORBA-based interface for monitoring and reporting real-time and historical operations.

Reasons for issue

- TAO is upgraded to 1.6a
- JacORB is upgraded to 2.3.1
- JRE is upgraded to 1.6.0_21
- Added description related to multi unit selection in

  Contents of struct Contents of struct AgentDynDataPerJob ->unitID.
- Certificates are renewed to increase expiry date.
- Added support for Windows 32-bit for Windows Vista, Windows 7, Windows 2008 and 64-bit for Windows XP, Windows Vista, Windows 7, Windows 2008
- Corrected the resolution of the images

Related documents

- Planning and Prerequisites for Avaya Proactive Contact
- Administering Avaya Proactive Contact (Linux-based Interface)
- Using Avaya Proactive Contact Supervisor
- Using Avaya Proactive Contact Agent
- Avaya Proactive Contact Safety and Regulatory Information

Overview

This section contains an overview of Avaya Proactive Contact and the Software Developer's Kit.

This section contains the following topics:

Avaya Proactive Contact fundamentals

Avaya Proactive Contact is a predictive dialing system designed to support both centralized and distributed call centers. It initiates outbound calls and receives and manages inbound calls. Avaya Proactive Contact consists of software and hardware components such as a digital telephone switch, host computer, system cabinet, workstations, and printers.

This section contains the following topics:

Avaya Proactive Contact configurations

There are three basic Avaya Proactive Contact configurations:

Avaya Proactive Contact

In this configuration, the system includes the Avaya PG230 digital switch and the dialer server. The Avaya PG230 digital switch is controlled by the dialing system software.

Avaya Proactive Contact with PG230

In this configuration, the system separates the Avaya PG230 digital switch and the dialer server. The Avaya PG230 digital switch is controlled by the dialing system software.

Avaya Proactive Contact with AES

In this configuration, the Avaya Communication Manager performs outbound calling functions instead of the Avaya PG230 digital switch.

The following figures illustrate the three basic configurations.

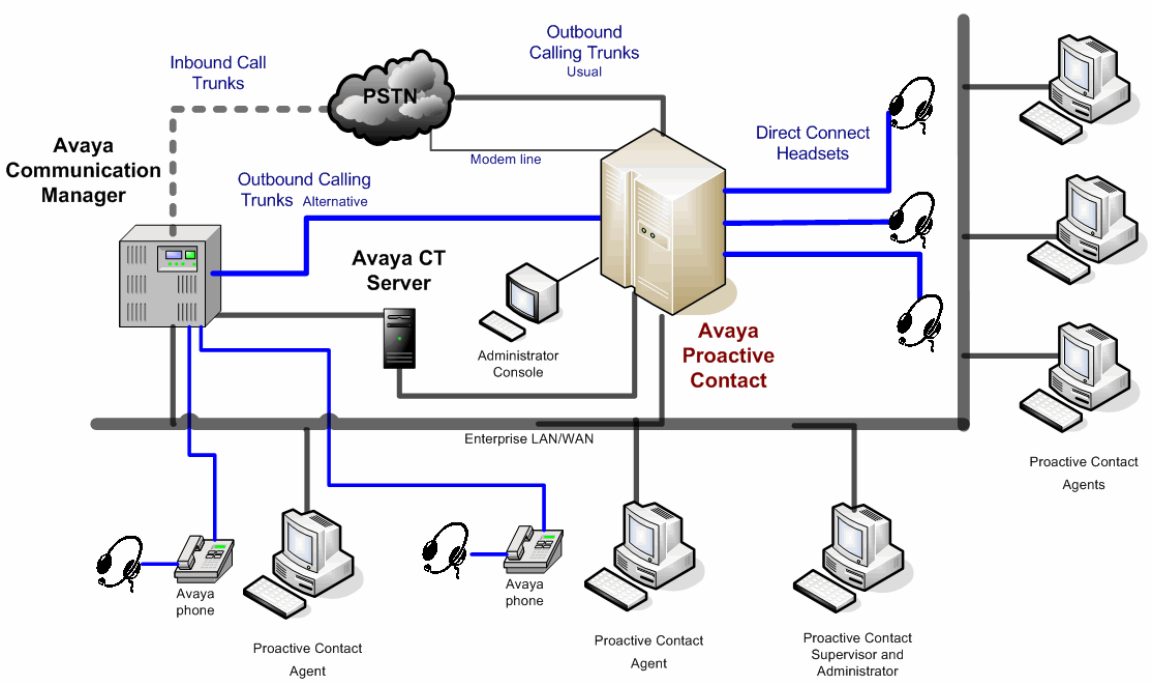


Figure 1:
Avaya Proactive Contact - dialer and dialer server in one cabinet

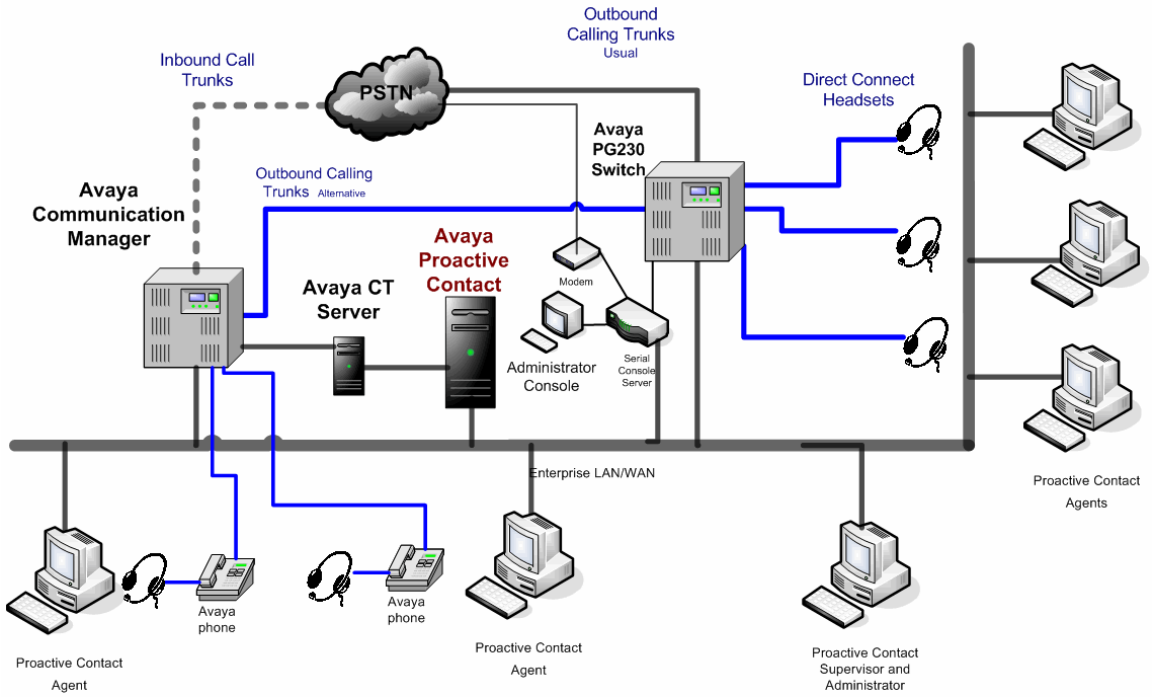


Figure 2:
Avaya Proactive Contact with PG230 - dialer and dialer server in two cabinets

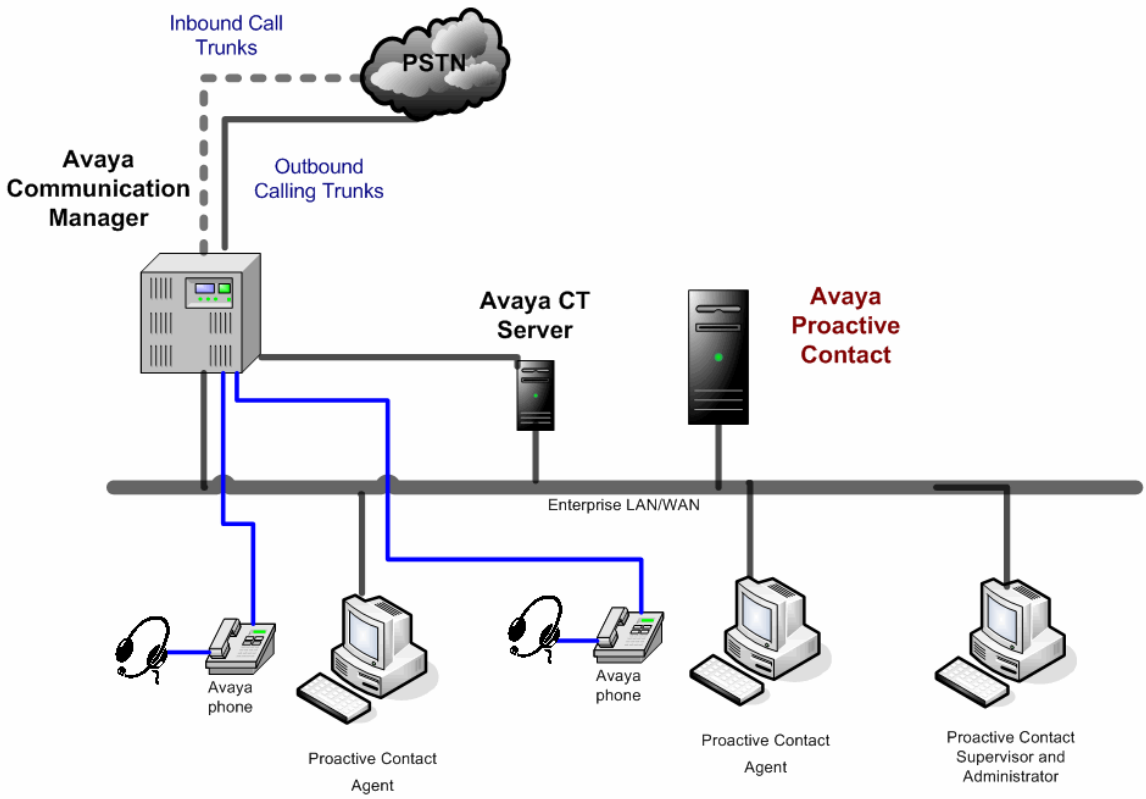


Figure 3:
Avaya Proactive Contact with AES

Blending options

There are two types of blending in Avaya Proactive Contact, Agent Blending and Intelligent Call Blending. Blending mixes inbound and outbound calling activities so that agents can handle either kind of call. Appropriate use of blending helps optimize agent and job effectiveness.

This section contains the following sections:

- Agent Blending
- Intelligent Call Blending

Agent Blending

Agent Blending integrates outbound calling activities on your Avaya Proactive Contact with inbound calling activities on your automatic call distributor (ACD). Avaya Proactive Contact provides two types of Agent Blending: Predictive Agent Blending and Proactive Agent Blending.

Both types of Agent Blending systems use a pool of ACD blend agents for outbound calling. The ACD agents log in to the dialer and the ACD. Agent Blending monitors the activity on the ACD to determine when to move agents between inbound and outbound calling activities.

The dialer acquires the pooled agents for outbound calling when the inbound calling activity decreases. The dialer releases the pooled agents to inbound calling when the

inbound calling activity increases. The movement between inbound and outbound calling keeps the ACD blend agents busy and the ACD service level within your prescribed limits.

Predictive Agent Blending - Enables call centers to focus on the inbound mission. Predictive Agent Blending uses events from the ACD to forecast call volume and determine when to move ACD agents between inbound and outbound calling. The dialer predicts when too many agents receive inbound calls. The dialer then acquires agents from the ACD to handle outbound calls until the inbound volume increases.

Proactive Contact Agent Blending - Enables call centers to focus is on outbound calling, but you need to service a low volume of inbound customers. Proactive Agent Blending focuses on outbound calls and releases agents to inbound only when an inbound calls enters a monitored queue on the ACD.

When an ACD agent logs in, the system immediately acquires the agent for outbound calling. When an inbound call arrives in the ACD queue, the dialer releases the agent to handle the call. If inbound calls continue to arrive, the dialer continues to release agents. When the queue is empty, the dialer acquires agents for outbound calls.

Agent Blending is an enhanced system that requires an ACD and AES. If the call center site has an ACD, either Agent Blending or Intelligent Call Blending can be used. However, both methods cannot be used on the same dialer server at the same time.

Intelligent Call Blending

In an Intelligent Call Blending system, blend agents handle outbound calls until there are more inbound calls than available inbound agents. Intelligent Call Blending passes the excess inbound calls to the blend agents. When the call volume decreases, Avaya Proactive Contact returns to passing outbound calls to blend agents.

Job types

Avaya Proactive Contact manages outbound calling and inbound calling through three types of calling jobs: outbound, inbound, and blend. Jobs provide means to organize groups of agents to handle certain customers to support the business goals of the enterprise.

Outbound jobs

When using outbound jobs, Avaya Proactive Contact dials phone numbers and routes outbound calls to agents on Intelligent Call Blending and Agent Blending systems. Avaya Proactive Contact screens for answering machines, phone operator intercepts, busy signals, Interactive Voice Responses (IVR), and no answers. The system reports on the outbound calls and agent activities.

Managed and Cruise Control jobs are special types of outbound jobs. A Managed job is used for Managed Dialing where an agent previews the record of an outbound call before the dialer places the call. A Cruise Control job is an outbound job where the dialer automatically monitors and adjust the dialing pace throughout the oubound job.

Inbound jobs

When using inbound jobs, Avaya Proactive Contact routes inbound calls to agents. If the call center has an Intelligent Call Blending system, the system supervisor uses the

system menus to control the inbound job settings. The system reports on the inbound calls and agent events when handling inbound calls on Avaya Proactive Contact.

If the call center has an Agent Blending system, the system supervisor uses the Agent Blending menus and the ACD to control inbound calling activity. The ACD reports on inbound calls and agent activities when handling inbound calls.

Blend jobs

Blend jobs are jobs on an Intelligent Call Blending system.

When using blend jobs, Avaya Proactive Contact moves agents between outbound and inbound calls. Blend agents receive inbound calls during peak inbound activity and outbound calls when inbound activity decreases. The customer uses Avaya Proactive Contact menus to control the inbound job settings. The system reports on all calls and agent events when handling outbound and inbound calls on Avaya Proactive Contact.

Call management

The following sections provide an overview of how Avaya Proactive Contact manages calling activities to make productive use of agent time and system resources:

- Call activities
- Call
- Device
- Calling lists
- Campaign management
- Monitoring calling activity

Call activities

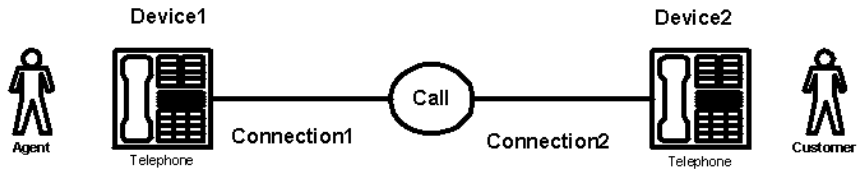
A client application can use calling activity data that the dialing server gathers and stores. The SDK retrieves data from the following perspectives:

- The calls made and received
- The agents who handle the calls
- The job, a collection of calls that fulfills a goal
- The dialer server that conducts the calling activity
- The phone lines used to transmit the calls

The components of a call include:

- The devices that participate in the call
- The connection that defines the relationship between the devices and the call
- The agent who handles the call
- Possibly other agents or personnel involved in call transfers and conferences

- The customer, or far end party that particiates in the call



Call

A call is a communication connection between two or more devices. At the programming level, call object consists of a call identifier and call state. The call identifier is a numeric string of 18 digits. It is composed of the following digits:

DDDPPPJJJHHNNNNNNN

- DDD represents the dialer identifier
- PPP represents the porter number
- JJJ represents the day in the year as a decimal number from 001 to 366
- HH represents the hour when the call was initiated from 01 to 23
- NNNNNNN represents the numeric identifier for the call

The system generates a call identifier in the following instances:

- A job initiates an outbound call
- A job checks for an inbound call on an inbound trunk
- An agent initiates a manual call
- An agent initiates a field call
- An agent initiates a transfer

When an agent transfers a call to a third party, there are three call identifiers (ID). Call ID 1 is the original call ID from the customer. Call ID 2 is the call that the receiving agent makes to connect to a second agent. Call ID 3 is the call created when the agent completes the transfer and disconnects from the call.

Device

A device can be a physical device such as the equipment number for an agent headset. It can also be a logical device identification number, such as a headset number, trunk group number, or line number.

A physical device identifier has a number that is unique within the switch. A logical device identification number is only unique within the device type. For example, you may have headset1, line1, and trunk1 on the same switch.

Calling lists

Avaya Proactive Contact uses a calling list to record call results in customer records.

Avaya Proactive Contact uses outbound calling lists to determine what customers to call. It creates the calling lists by converting host database files into the system format and adding fields. After Avaya Proactive Contact adds the necessary fields, it checks for and flags duplicate records and invalid phone numbers. It can also mark records that have been on the system more than a specified number of days.

Typical added fields include:

- The name of the last agent to speak with the customer
- The date and time of the last call attempt
- The result of the last call attempt
- The number of days the record has been on the system
- The date the record was first downloaded
- The time zone
- The record status

Avaya Proactive Contact with Intelligent Call Blending uses inbound calling lists for inbound and blend jobs. An inbound calling list contains data fields but no customer records.

Avaya Proactive Contact and agents use completion codes to record call results in customer records.

Campaign management

A campaign is a strategy that supervisors design to achieve call center goals. One element of a campaign is a job. Campaigns can include one or more jobs. A job's objective is to accomplish specific campaign goals. A job consists of a calling list, a phone strategy, a record selection, a job definition, job screens, and other settings.

Monitoring calling activity

When a job is active, the Avaya Proactive Contact supervisor can monitor the calling activity and adjust settings to ensure that the job meets call center goals.

Agent activities

Call center agents are responsible for talking with customers and updating customer records. To develop agent applications, you need an understanding of agent tasks. This section provides background information to help you understand the agent functions.

This section contains the following topics:

- Transferring calls
- Placing manual calls
- Placing field calls
- Calling alternative telephone numbers
- Scheduling recalls
- Updating customer records
- Ending calls
- Leaving jobs
- Disconnecting headsets
- Logging out

Agent types

Agents work on specific types of jobs by logging in to the system as a specific type of agent. An agent who logs in to an Intelligent Call Blending system handles outbound and inbound calls during outbound, inbound, and blend jobs. An agent who logs in to an Agent Blending system handles outbound calls on Avaya Proactive Contact and inbound calls on the ACD. The agent type determines which features are available and which jobs the agents can handle.

Agent Type	Description
Outbound Agent	Agents handle calls that Avaya Proactive Contact places during outbound and blend jobs on Intelligent Call Blending systems and outbound jobs on Agent Blending systems.
Inbound Agent	Agents handle calls received from customers during inbound and blend jobs on Intelligent Call Blending systems.
Blend Agent	Agents handle calls received from customers during inbound and blend jobs on Intelligent Call Blending systems.
Person to Person Agent	Agents receive calls only when all other agents on the job are busy during any job on Intelligent Call Blending and Agent Blending systems.
Managed Agent	Agents handle only outbound calls during Managed Dialing jobs on Intelligent Call Blending and Agent Blending systems.

ACD Agent	ACD agents log in to both Avaya Proactive Contact and the ACD. They handle outbound calls from Avaya Proactive Contact when not handling inbound calls from the ACD on Agent Blending systems.

Agent tasks

Call center agents talk with customers and update customer records. Throughout the day, agents work on Avaya Proactive Contact handling calls. This section describes the tasks agents complete while on the system.

Logging in to Avaya Proactive Contact

An agent enters a user name or user identification number (ID) and a password to log in to Avaya Proactive Contact. The system prompts the agent to enter a workstation or headset ID. Avaya Proactive Contact confirms that the ID is valid, displays the agent interface, and establishes a voice connection to the agent headset.

If the agent is an ACD Agent on an Agent Blending system, the agent also logs in to the ACD.

Joining jobs

After establishing workstation and voice connections, the agent joins or attaches to an active job.

Finally, the agent tells Avaya Proactive Contact that he or she is available to take calls.

Handling calls

Depending on the configuration, Avaya Proactive Contact distinguishes between calls answered by a person and calls answered by an electronic device. Avaya Proactive Contact refers to calls answered by a person as "live" calls; calls answered by an electronic device are Autovoice calls .

The supervisor configures the job to determine which call types Avaya Proactive Contact passes to the agents. Supervisors may decide to pass Autovoice calls to agents so that agents can leave messages on the device. Or, they may decide to screen these calls. In addition, the supervisor decides whether to place calls in a wait queue when no agents are available.

Transferring calls

Agents can transfer inbound or outbound calls to another telephone number, a supervisor, or an extension. Avaya Proactive Contact can be configured to allow the agent to transfer both the call and customer record. Avaya Proactive Contact refers to this as Voice and Data Transfer.

Agents can disconnect from the customer who is on hold before the transfer party answers the phone, or they can stay on the line and announce the transfer. While on the call, agents can remain in the on-call state until they disconnect from the transfer or the call. Agents use function keys to disconnect from the transfer.

During a supervised transfer, the agent can disconnect the call transfer by a manual hang up and restore the contact with the customer. An agent may disconnect the call if a call transfer reaches a wrong number, busy signal, answering machine, or an error message.

Placing manual calls

When placing a manual call, the agent may get a busy signal, answering machine, or no answer. The agent must enter the appropriate completion code in the customer's record.

Placing field calls

The agent screen displays fields that contain telephone numbers. The agent selects a customer telephone number and tells Avaya Proactive Contact to place the call. The system calls the number.

When placing a field call, the agent may get a busy signal, answering machine, or no answer. The agent must enter the appropriate completion code in the customer's record.

Calling alternative telephone numbers

The agent screen displays fields that contain telephone numbers. The agent selects a customer telephone number and tells Avaya Proactive Contact to place the call. The system then calls the number.

The agent may tell Avaya Proactive Contact to dial a different phone number. If the agent chooses to dial digit by digit, the system prompts the agent for each digit. Otherwise, the system prompts the agent for the entire telephone number.

Scheduling recalls

Avaya Proactive Contact can be set up to allow the agent to schedule recalls. A scheduled recall allows the agent to specify the phone number and set the time and date to place a phone call to the customer. On systems with the Agent Owned Recall feature, agents can specify that they handle the call. When Avaya Proactive Contact places an agent owned recall, it attempts to locate the agent from the list of agents logged in to the system. If the agent is not logged in, Avaya Proactive Contact postpones the call.

Updating customer records

Agents update the customer records on their screens and send the updated information to Avaya Proactive Contact or the host.

Ending calls

When the conversation ends, the agent may be finished with the customer's record and ready for another call, or the agent may need additional time to finish customer record updates.

If the agent needs additional update time, the agent releases the telephone line and then finishes updating the customer record.

When the agent is ready to take another call, the agent assigns a completion code that identifies the conversation's outcome. Avaya Proactive Contact saves the record

update, the agent releases the record. Avaya Proactive Contact then records that the agent is available for another call.

Leaving jobs

Agents typically leave a job or go offline for the following reasons:

- To transfer to another job
- To stop handling calls temporarily during the day to take a break or attend a meeting
- To log out at the end of the day

To leave a job, the agent requests to go on "break". If the system was about to pass a call to the agent when it receives the request, the system may complete the connection to the agent.

Avaya Proactive Contact stops placing calls when all agents leave a job. However, if there are calls in the wait queue, the last agent to leave a job continues to receive calls until the call queue is empty.

Disconnecting headsets

When the agent stops work for the day, the agent logs out. Avaya Proactive Contact disconnects from the agent's headset and clears the memory buffer reserved for that headset identification. On most Avaya Proactive Contact agent interfaces, the logout process automatically disconnects the headset.

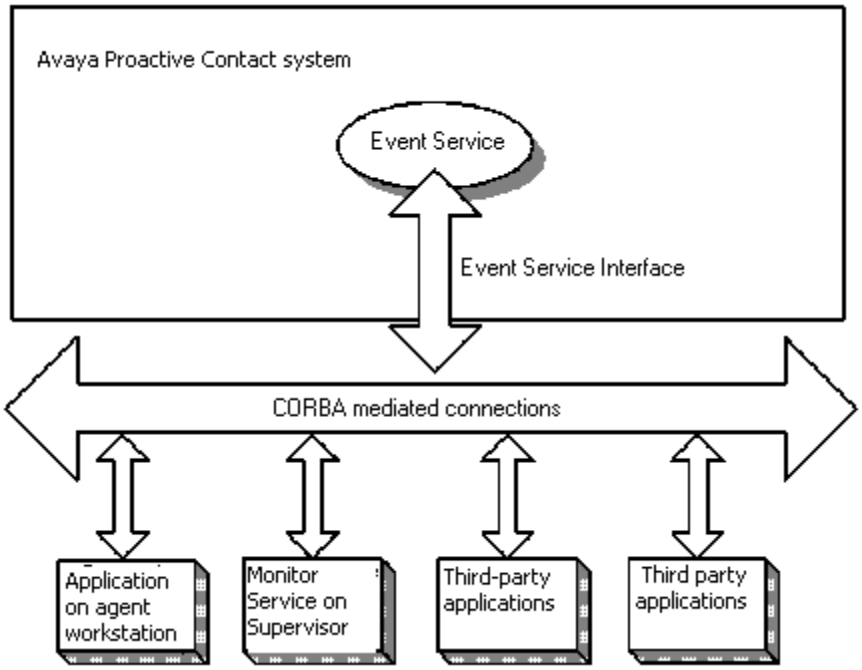
Logging out

Agents log out of Avaya Proactive Contact from the agent menu. Avaya Proactive Contact logs the agent out of the system and returns the workstation to the Avaya Proactive Contact log in prompt.

If the agent is an ACD agent working on an Agent Blending system, the agent must log out of the ACD before logging out of Avaya Proactive Contact.

If the workstation also connects to a host system, the agent may need to log out of the host system after logging out of Avaya Proactive Contact.

The SDK is a Common Object Request Broker Architecture (CORBA) based service that provides an Event Interface for Avaya Proactive Contact. The SDK gathers and processes Avaya Proactive Contact events and statistics. This information can be integrated with client or third-party applications to store or display data and to make decisions about call center processes.



The SDK communicates with client applications over a CORBA interface. This interface is defined in Interface Definition Language (IDL) files. CORBA hides the underlying communication between client applications and Avaya Proactive Contact, provides scalable services, and helps manage the interfaces between client applications and servers.

Working in a CORBA environment

CORBA defines a framework for developing object-oriented, distributed applications. With CORBA, applications can communicate with one another no matter where they are located or how they are designed. Developers can create distributed applications that interact as though the applications were implemented in one programming language on one computer.

The Avaya Proactive Contact event SDK CORBA interface is implemented using a CORBA 2.6, IIOP 1.2 compliant interface. For information on CORBA see http://www.omg.org/gettingstarted/corbafaq.htm. The CORBA implementation used by Avaya Proactive Contact is the ACE ORB (TAO) version 1.6a; for information on TAO see: http://www.theaceorb.com/

Developing client applications

The SDK provides access to real-time Avaya Proactive Contact event and statistical information. Customers can develop monitoring and tracking applications that receive these events from the SDK. This information is helpful when a customer is making decisions concerning operations, staffing, and business goals.

The application developer begins by developing a client application that implements or instantiates a callback object to receive the event notifications. The callback object

implements the methods the SDK uses to deliver events and statistics. For more information on creating callback objects, see Using the Avaya Proactive Contact SDK.

A user name and password is required to access the event API. For more details, see Logon and SetPasswd.

Sample applications

The developer's kit includes sample applications that illustrate how you can implement SDK client applications. For more information about each application, refer to the appendices.

The Java Sample Application demonstrates how a CORBA complaint Object Request Broker (ORB) can be used to connect to the SDK. This application demonstrates how to test the SDK interface. This sample code was written for customers who want to implement client applications using Java.

Using the Avaya Proactive Contact SDK

The Avaya Proactive Contact SDK was developed using the Common Object Request Broker Architecture (CORBA) to allow interoperability between distributed applications. This application-to-application communication is independent of the programming language, the operating system it runs on, and the network location.

This section contains the following topics:

Working with interface definition language

The Avaya Proactive Contact SDK Interface Definition Language (IDL) files define the interface between client and server applications. The IDL files, however, do not contain all information a client application needs to use Avaya Proactive Contact SDK.

The client application submits requests to Avaya Proactive Contact SDK using operations that are defined in the IDL files. You can implement the interface in various programming languages, such as C++, Visual Basic, and Java, according to the language mapping defined by the Object Management Group (OMG).

When the Avaya Proactive Contact SDK server receives a logon request from the client, it creates an object that implements the Avaya Proactive Contact SDK interface as defined by the IDL and returns a reference to that object to the client application.

When the Avaya Proactive Contact SDK server receives a logoff request from a client with the object, it reclaims the object and performs cleanup work.

Connecting to the Avaya Proactive Contact SDK

A client application can connect to the Avaya Proactive Contact SDK using standard CORBA methods to locate and connect to CORBA objects, choosing either the standard name of the Naming Service or the Interoperable Object Reference (IOR) of the Naming Service to do so.

This section contains the following topics:

- Connect using the Naming Service IOR
- Connect to the Avaya Proactive Contact SDK

Connect using the Naming Service

The Naming Service allows a client to locate object references based on abstract, programmer defined object names.

A CORBA server assigns a name to an object and registers the name and the object with the Naming Service. Before a client application can use a CORBA object, the application gets a reference to the object, called an object reference.

To locate an object using the Naming Service, the client application must connect to the Naming Service and query it for an object reference that corresponds to the name of the same object. The client application connects to the object named dialer.*ServerHostName*.eventserver, where the *ServerHostName* is the host on which the Avaya Proactive Contact SDK runs. The Naming Service returns an object reference to the Avaya Proactive Contact SDK object. The client application then uses this object to connect to and to log in to the Avaya Proactive Contact SDK.

The following Visual C++ code illustrates how to use the Naming Service to locate an object. Begin your client application with this code:

```
#include <fstream.h>
#include <time.h>
#include "ace/Get_Opt.h"
#include "CosNamingC.h"
#include "EventClientS.h"
#include "EventServerC.h"
#include "StringConvert.h"
#include "ace/SString.h"

using namespace EventServer;
using namespace ESType;
using namespace Common;

#include "EventClient.h"
#include "EventClient_i.h"

client_global_t * g;

// forward declarations:

void   usage (void);

void   init (int, char **);

void   initSigHandlers (void);

static void sigIntHandler (int);

long dt_to_abs_time(const char *ts);

CORBA::Object_ptr import_object (CORBA::ORB_ptr, const char *);

/*-
 * Function:    main
```

```
 *
 * Description:
 *
 * Usage:
 *
 * Return Values:
 *
 * Side Effects:
 *
 * Process:
 *
 * Calls:
 */

int
main (int argc, char ** argv)
{

  // Create the client global object:

  client_global_t client_global_obj;

  g = &client_global_obj;

  CORBA::ORB_var orb = CORBA::ORB::_nil ();

  try
  {
    cout << "initializing the ORB" << endl;

    orb = CORBA::ORB_init (argc, argv, "TAO");

  }
  catch (...)
  {
    cout << "Error: ORB_init failed" << endl;
    exit (1);
  }

  init (argc, argv);
  initSigHandlers ();

  setbuf (stdout, 0);
  setbuf (stderr, 0);

  PortableServer::POA_var child_poa;

  PortableServer::POAManager_var poa_manager;

  try
  {
    CORBA::Object_var objV;
    CosNaming::NamingContext_var rootContext;

    // Initialize the ORB:

    CORBA::Object_var poa_obj = orb->resolve_initial_references ("RootPOA");
```

```
// Narrow the Root POA to POA_var:

PortableServer::POA_var root_poa =
  PortableServer::POA::_narrow (poa_obj);

assert (! CORBA::is_nil (root_poa));

// Share the root POAs POA Manager:

poa_manager = root_poa->the_POAManager ();

// Create the Empty Sequence for Default POA Policies:

CORBA::PolicyList policyList;

policyList.length (0);

child_poa = root_poa->create_POA ("child", poa_manager, policyList);

objV = orb->resolve_initial_references ("NameService");

try
{
  rootContext = CosNaming::NamingContext::_narrow (objV);
}
catch (CORBA::SystemException &e)
{
  cerr << "Unexpected system exception" << &e << endl;
  exit (-1);
}

if (CORBA::is_nil (rootContext))
{
  cout << "rootContext is nil" << endl;
  exit (1);
}

if (g->using_IOR_flag)
{
  objV= import_object (orb, g->IOR_file);
}
else
{
  // Construct a Name:

  CosNaming::Name serverNameV;

  serverNameV.length (5);

  serverNameV[(unsigned long) 0].id   = CORBA::string_dup ("PDS");
  serverNameV[(unsigned long) 0].kind = CORBA::string_dup ("");

  serverNameV[(unsigned long) 1].id   = CORBA::string_dup ("dialers");
  serverNameV[(unsigned long) 1].kind = CORBA::string_dup ("");

  serverNameV[(unsigned long) 2].id   = CORBA::string_dup (g->server_host);
  serverNameV[(unsigned long) 2].kind = CORBA::string_dup ("");
```

```
      serverNameV[(unsigned long) 3].id   = CORBA::string_dup ("eventserver");
      serverNameV[(unsigned long) 3].kind = CORBA::string_dup ("");

      serverNameV[(unsigned long) 4].id   = CORBA::string_dup ("v2_0");
      serverNameV[(unsigned long) 4].kind = CORBA::string_dup ("");

      // Now resolve with name server

      objV = rootContext->resolve (serverNameV);
    }

    g->serverV = DialerEventServerIF::_narrow (objV);

    if (CORBA::is_nil (g->serverV))
    {
      cout << "Error: failed to narrow object reference" << endl;

      return 1;
    }

  }
  catch (CORBA::SystemException &e)
  {
      cerr << "Unexpected system exception" << &e << endl;
      exit (-1);
  }
  catch (...)
  {
    // an error occurred while trying to bind to the object.
    cerr << "Error: Bind to object failed" << endl;
    exit (1);
  }
```

Connect using the Naming Service IOR

An IOR is a data structure that encodes a particular object. A CORBA client can use the IOR to locate and connect directly to that object. An alternative method is to do the following:

- Connect to the Avaya Proactive Contact SDK using the Naming Service IOR to connect to the Naming Service.
- Query the Naming Service to obtain a reference to the Avaya Proactive Contact SDK.

You can view an IOR as a string using certain tools or as a hex string. The Naming Service IOR contains the information that uniquely identifies the object to the network. Clients can use the IOR to locate and connect directly to that object.

If you have command line access to the Naming Service, invoke the following command to retrieve the IOR file:

```
/opt/ACE_wrappers/TAO/bin/Naming_Service -o <myIORfile>
```

If you don't have command line access, contact your Avaya Customer Support representative to arrange to have the IOR file sent to you.

The following Java code illustrates how to use the Naming Service IOR to locate an object.

```
FileInputStream fis = new FileInputStream("NSIOR");
DataInputStream dis = new DataInputStream(fis);
String ior = dis.readLine();
System.out.print("IOR: ");
System.out.println(ior);
org.omg.CORBA.Object obj = orb.string_to_object(ior);
NamingContext ncRef = NamingContextHelper.narrow(obj);
NameComponent nc0 = new NameComponent("PDS", "");
NameComponent nc1 = new NameComponent("eventserver", "");
NameComponent path[] = new NameComponent[2];
     path[0] = nc0;
     path[1] = nc1; DialerEventServer_IF desRef =
DialerEventServer_IFHelper.narrow(ncRef.resolve(path));
     return desRef;
} catch(Exception e)
     {
     System.out.println("ERROR : " + e);
     e.printStackTrace(System.out);
     throw e;
}
```

Connect to the Avaya Proactive Contact SDK

After you set up the Naming Service or IOR, complete the following steps to develop a client application and connect with the Avaya Proactive Contact SDK:

To develop the client application:

1. Choose a development language and Object Request Broker (ORB).
2. Connect to the Avaya Proactive Contact SDK using the method appropriate to the ORB and development language you have chosen.
3. Write application code that connects and logs on to the Avaya Proactive Contact SDK as illustrated in the sample applications discussed in the appendices in this guide.
4. Write the application code that invokes appropriate requests on the server according to the interface defined in the server side IDL.
5. Write the application code that implements a callback object according to the interface defined in the client side IDL.

To set up a run-time environment on the client run-time host:

1. Obtain an ORB for the client application's machine.
2. Install and configure the client machine.
3. Complete any network configuration required to establish communication between the client and server machines.

To ensure the client application talks to the Avaya Proactive Contact SDK:

1. Install the client application.
2. Log in to the client application.
3. Register for events or statistics.

4. Process the event or statistical information received through the callback object methods.

After you log in to the Avaya Proactive Contact SDK, you receive a confirmation that the service object was created.

After you receive this confirmation, you can register to receive events. Use the following sections to help you determine which events you want to receive.

Call event state diagrams

This section provides state diagrams for the following call types supported in Avaya Proactive Contact:

- Inbound calling
- Outbound calling , including both predictive and managed calls.

The circles in the state diagrams represent the states that the calls can be in.

The arcs are the state transitions. They are labeled with the names of call events described in Commands and notification events. Call events represent the possible actions that can be taken on a call to progress it through its life cycle.

The state diagrams represent the valid sequence of states and events that a call can progress through. Equivalently stated, follow the directional arcs to progress through the possible scenarios of a call.

The possible call states and their meanings are given in the following table.

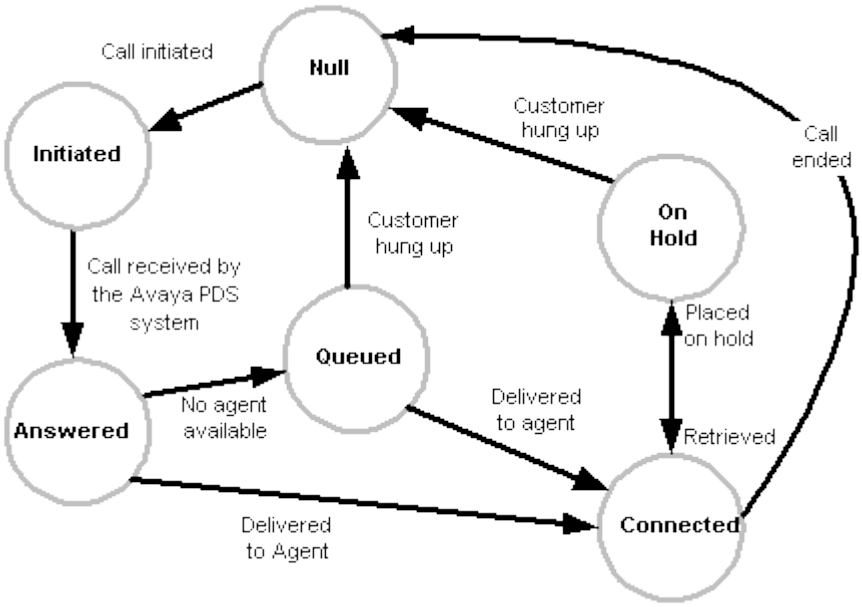
Call state	**Meaning**
Answered	An outbound call is answered by a far end party. An inbound call is answered by an agent.
Connected	The agent is capable of controlling an answered call.
Consulted (1)	The call is connected between the agent and a consulted-to party.
Dialed	An outbound call is in the process of being dialed and monitored for answer.
Disconnected	The call connection no longer exists.
Initiated	The dialer is in the process of preparing to create a new outbound call, or it is preparing to deliver an inbound call to an agent.
On hold	The call is on hold.

null state	A state diagram is entered with a solid circle, and exits with a clear circle with a contained solid circle. These are both considered to be null states. A null state means the entity being modeled, in this case a call, does not yet exist or no longer exists.
Previewed	The customer data for a managed call is being presented to and previewed by an agent.
Queued	The call is in a wait queue and a message may be played to the far end party. This occurs because there is no agent available to handle the call. This can happen to inbound and outbound calls, although proper tuning of the dialer can avoid the latter.
Transferred	A call is in the process of being transferred. Certain call states are not actual call states within the dialer. Certain call events happen in an immediate sequence but do not cause a real state transition, yet to be represented as transitions on a state diagram they have to go through a state. Thus the use of the imputed states.

To facilitate the implementation of your client application, use the state diagrams to help identify the appropriate sequences of call events described in Commands and notification events.

Inbound calling

An inbound call scenario starts when a customer places an inbound call to Avaya Proactive Contact. If an agent is available, the system passes the call to the agent. If no agent is available, the system passes the call to an inbound wait queue. When an agent is available, the system connects the call and agent. The agent may place the call on hold and retrieve it to continue the conversation. The caller may hang up and disconnect the call while the call is in the wait queue or on hold.



Outbound calling

This section covers the state behaviors of predictive and managed outbound calling.

This section includes the following:

- A brief description of the types of calling covered
- The state diagram for outbound calling
- A description of each of the possible state transitions

Predictive outbound calling

Avaya Proactive Contact determines when to place an outbound call to the customer. The system selects a trunk, and then dials the customer's telephone number.

If the customer answers the call, he or she will normally be connected to an agent. If an agent is not available, the system places the dialed call in the wait queue. When an agent later becomes available, the system connects customer to the agent.

If the call is not answered, the call attempt is ended. Also, depending on job configuration, if an answering machine is detected, the call may be routed to an agent, or to a recorded message, or merely ended.

Managed outbound calling

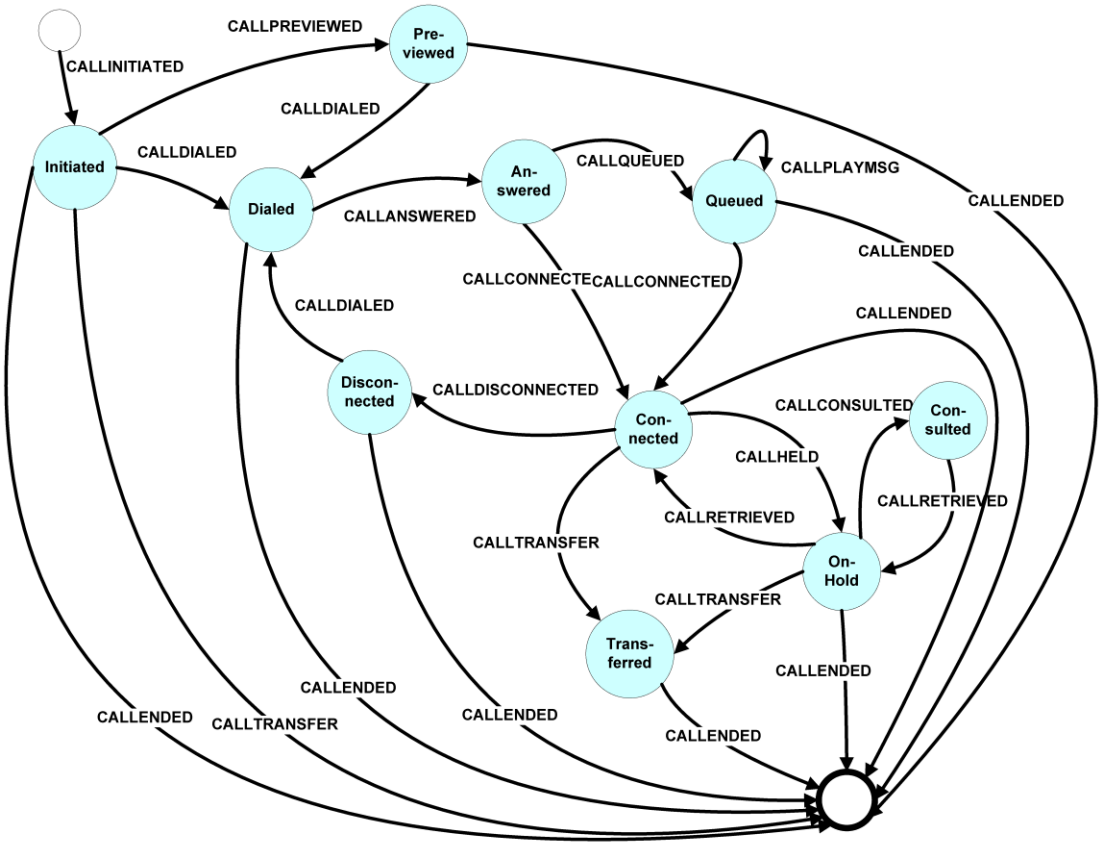
Avaya Proactive Contact first selects a customer, and the agent application presents information on that customer to the agent. The agent can preview this information for a time interval which is specified in the job configuration.

If the preview time is zero, the system attempts the call immediately. If the preview time is greater than zero, the agent may cancel or call that customer within that time interval. If the agent waits longer than the preview time, the system will start the call attempt.

Depending on configuration, either the system or the agent will determine whether a connection is made to a live party, an answering machine, or was not successful.

In all of the above cases, when an agent is connected to a customer, the agent may then place a call on hold, and later retrieve it to continue the conversation. The caller may hang up and disconnect the call while the call is on hold. The agent may also transfer a call, or hang up on the call, which is not a desirable agent behavior.

In the following diagram, the circles represent the outbound calling states, and the directed lines represent the state transitions. Each transition is labeled with the appropriate call event; the call events are described in detail in Commands and notification events.



The following table describes the possible state transitions:

Original state	**Call event**	**New state**	**Meaning**
null (starting)	CALLINTIATED	Initiated	The dialer is starting the process creating a call, using customer information from the calling list of a job.
Initiated	CALLPREVIEWED	Previewed	Information about a managed call is presented to an agent and the agent previews it.
Initiated	CALLENDED	null	Call initiation failed due to an equipment or internal resource error.
Initiated	CALLTRANSFER	null	Completion of a trunk-to-trunk transfer.
Initiated	CALLDIALED	Dialed	A predictive call is being dialed.

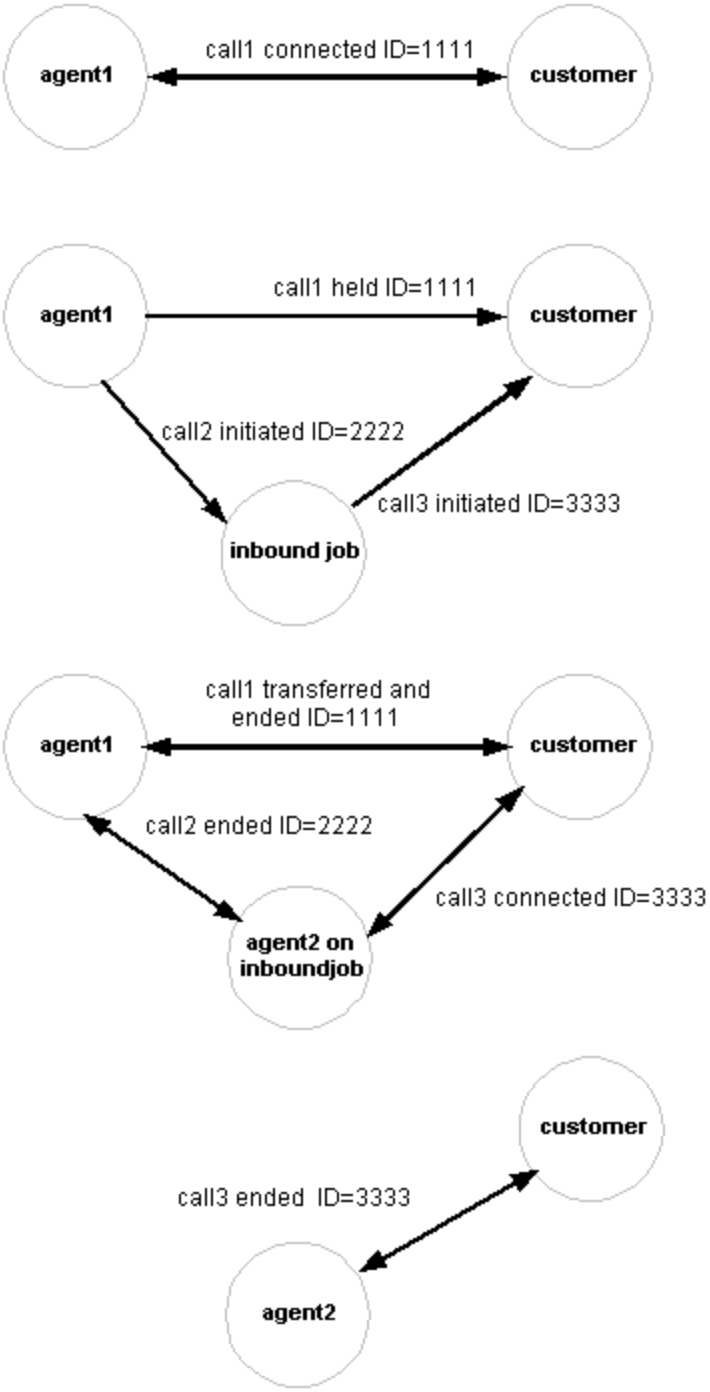
Previewed	CALLDIALED	Dialed	A managed call is being dialed. The agent has accepted to pursue a managed call, or a time-out was reached whereupon the dialer initiated dialing. The connection to the desired far end phone number is being established.
Previewed	CALLENDED	null	The agent has chosen to cancel a managed call. "Cancel" means that the agent determined that some preview data for the desired far end party indicated it was not appropriate to call at this time, and decided to not pursue the managed call.
Dialed	CALLANSWERED	Answered	The device at the far end party is now part of a physical call. For managed calls without automated call progress analysis (CPA), this is the result of the far end being answered. It could be answered by a person, an answering device, be a network or switch message or special information tone (SIT), or due to some other means. For predictive calls, which always use CPA, the CPA has determined that an agent is needed to talk to a live party, or to leave a message if answering machine detection is supported and the job is appropriately configured.
Dialed	CALLENDED	null	For managed calls without CPA, the call was hung up on by the agent before it was answered. (This is how an agent treats a busy indication.) For predictive calls, the CPA determined that the far end was not answered by a live person or an answering machine; or it could have been answered by an answering machine but the job is configured to not accept such calls.
Answered	CALLCONNECTED	Connected	The agent is in control of the call. Note, CALLANSWERED indicates that the physical call has been established.

Answered	CALLQUEUED	Queued	The call is in a wait queue and a message may be played to the far end party. This occurs because there is no agent available to handle the call. This can happen to inbound and outbound calls, although proper tuning of the dialer can avoid the latter.
Answered	CALLENDED	null	A call was answered by an answering machine or other electronic voice device, and, per job configuration, the call was not sent to an agent.
Queued	CALPLAYMSG	Queued	A message is played to a call in queue. This can happen repeatedly as long as the call is queued.
Queued	CALLCONNECTED	Connected	An agent has become available to take a queued call, the call is delivered to the agent, and the agent is in control of the call. Note, CALLANSWERED indicates that the physical call has been established.
Queued	CALLENDED	null	The call has ended because the far end party abandoned it while waiting in queue.
Connected	CALLENDED	null	The call has been ended. The agent may or may not be in a wrap up mode. That is determined from Agent states, described inAgent st.
Connected	CALLDISCONNECTED	Disconnected	A call has been physically disconnected, but the agent has not released the line. This allows the agent to make follow up calls related to the initial call.
Connected	CALLHELD	OnHold	The agent puts a call on hold. This allows the agent to transfer the call to another party.
Connected	CALLTRANSFER	Transferred	The call was transferred.
OnHold	CALLCONSULTED	Consulted	

OnHold	CALLENDED	null	The call has ended because the far end party abandoned it.
Consulted	CALLRETRIEVED	OnHold	
Transferred	CALLENDED	null	
Disconnected	CALLDIALED	Dialed	The agent is making a manual call as a follow up to a prior call.
Disconnected	CALLENDED	null	

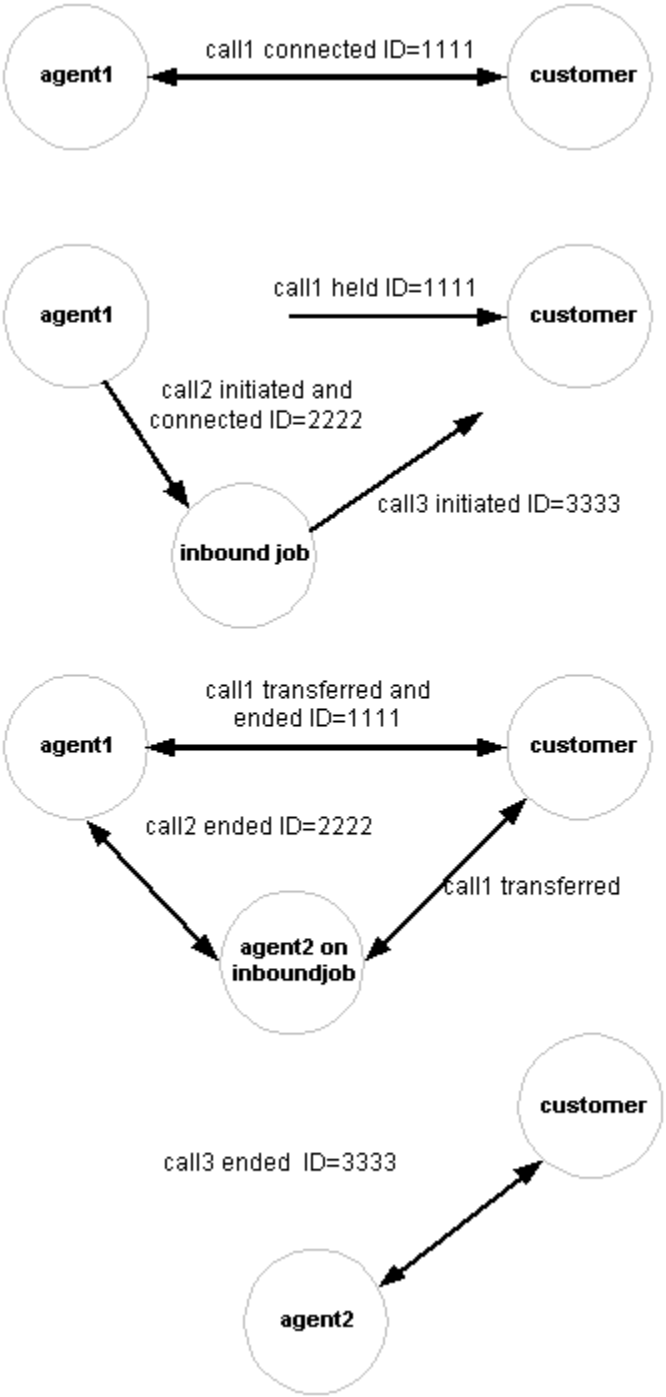
Blind native voice transfer

Blind native voice transfer disconnects the transferring agent before transferring the call and customer record. Avaya Proactive Contact places the customer call in the queue for the new job or connects the customer to an available agent. Use this diagram to help identify the Blind Native Voice transfer to implement through your client application.



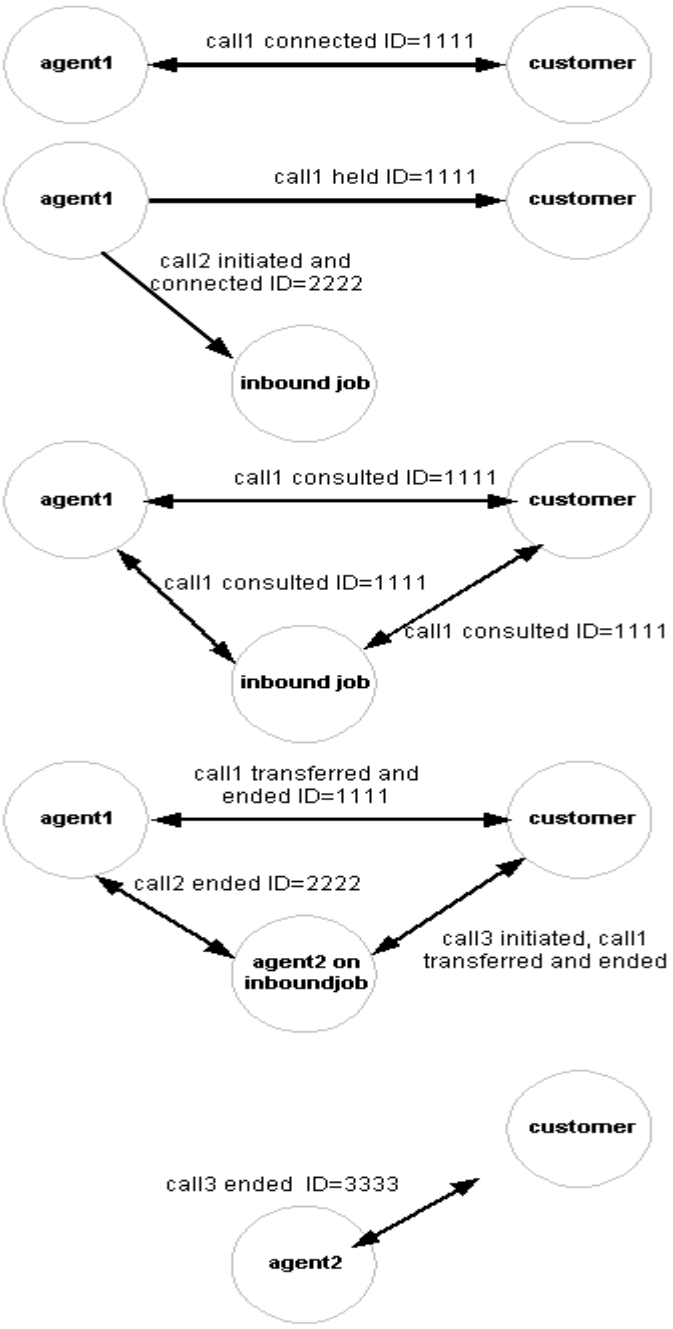
Supervised native voice transfer

Supervised native voice transfer transfers a telephone call and the customer record to another job, maintaining the transferring agent's telephone connection. The transferring agent stays connected to the telephone number during the transfer. The system places the customer on hold. Use this diagram to help identify the supervised native voice transfer to implement through your client application.



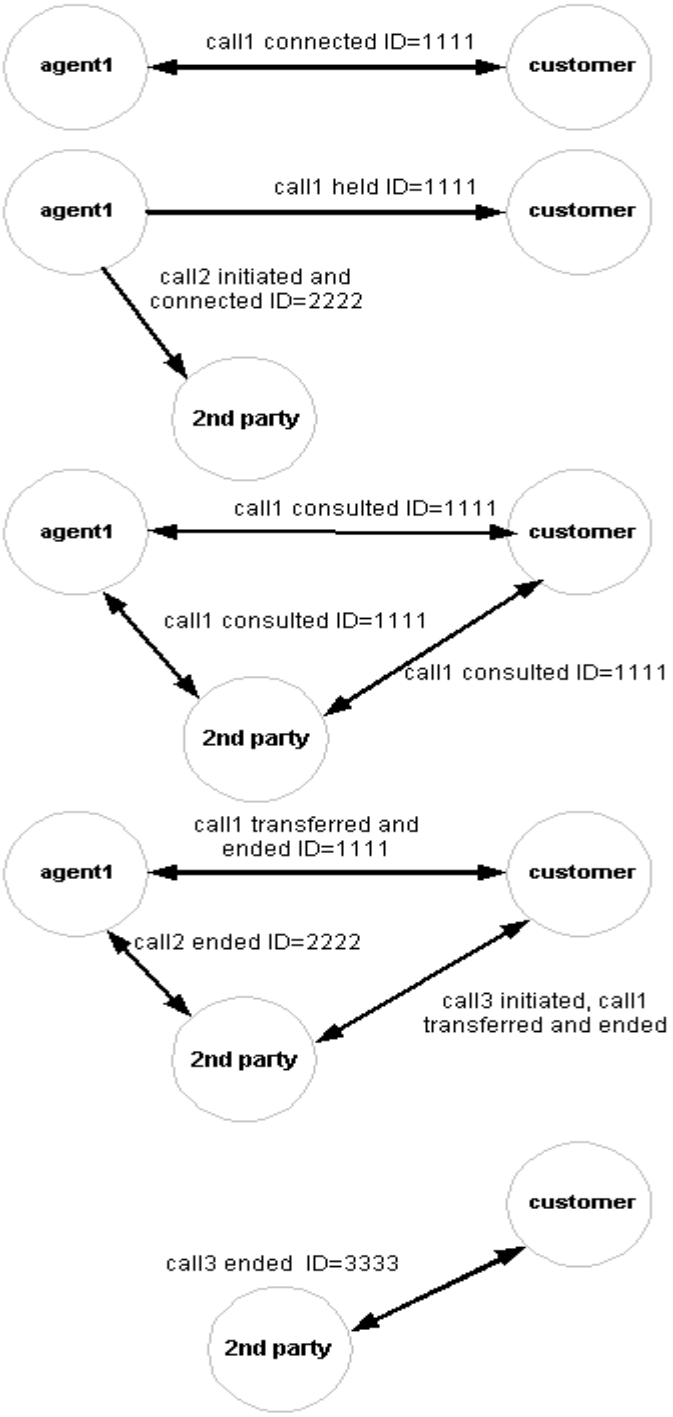
Supervised native voice transfer with consulted

Supervised native voice transfer with consulted transfers a telephone call and the customer record to another job, maintaining the transferring agent's telephone connection. The transferring agent does not stay connected to the telephone number during the transfer. The system places the customer on hold. Use this diagram to help identify the supervised native voice transfer with consulted to implement through your client application.



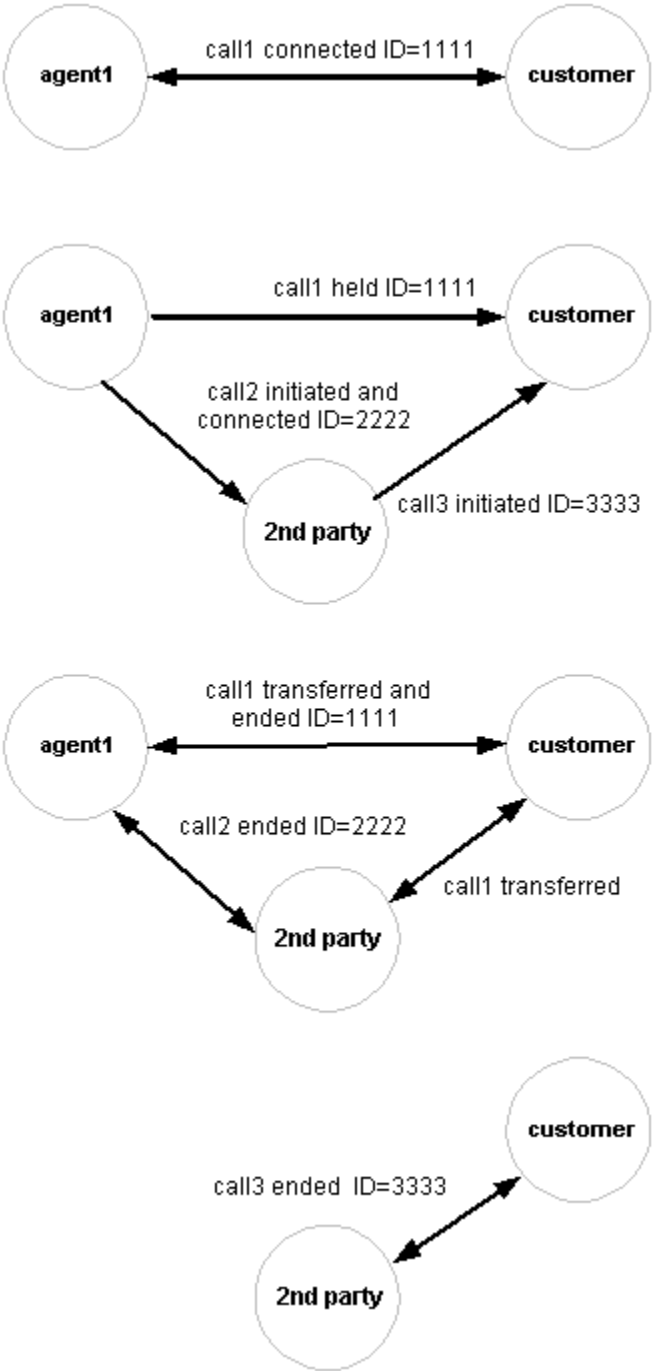
Trunk-to-trunk transfer with consulted

Trunk-to-trunk transfer with consulted transfers a configured customer call. The Avaya Proactive Contact places the customer on hold, then uses a telephone configured as a trunk to place a manual call to another job. Use this diagram to help identify the trunk-to-trunk transfer to implement through your client application.



Trunk-to-trunk transfer without consulted

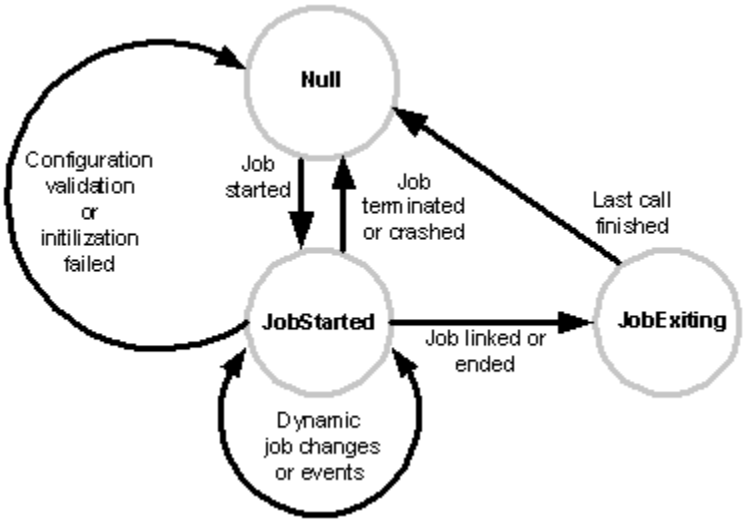
Trunk-to-trunk transfer without consulted transfers a configured customer call. Use this diagram to help identify the trunk-to-trunk transfer without consult to implement through your client application.



Job state scenarios

During calling activity, the Avaya Proactive Contact starts, stops, links, and shuts down jobs. Between the time the Avaya Proactive Contact starts and stops a job, the job progresses through several different states.

Use this diagram to help identify the job events to implement through your client application.



Agent state scenarios

During a job, an agent logs in to the Avaya Proactive Contact, selects a job, performs job setup activities, joins a job, and handles calls. When the job is finished, the agent logs out of Avaya Proactive Contact. Between login and logout an agent progresses through a number of different states.

The following diagram identifies the agent events and corresponding agent states. Agent states can include:

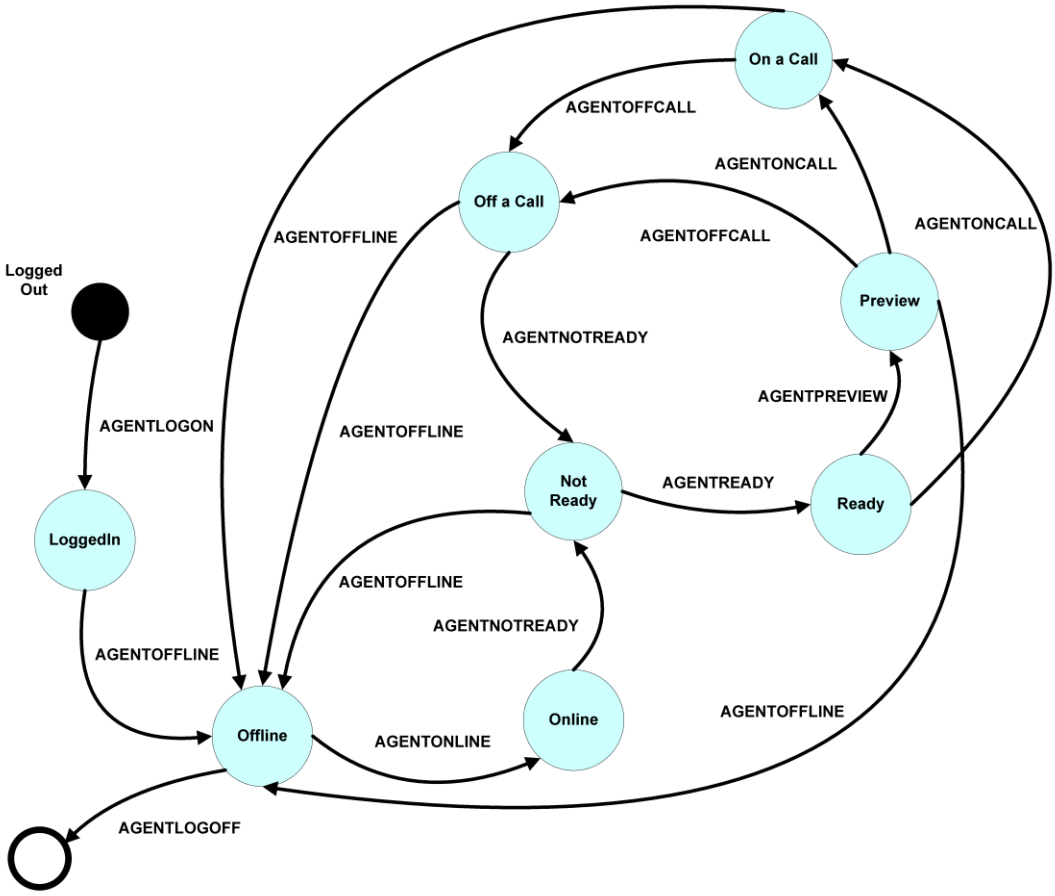
- Not logged in to Avaya Proactive Contact
- Logged in to Avaya Proactive Contact but not attached to a job
- Attached to a job but not available for work
- Attached to a job, available for work, but not ready for a call
- Waiting for a call
- On a call

When an agent logs in, he or she establishes an agent identification. After an agent establishes a headset connection, he or she is in the online state.

On an Intelligent Call Blending system, an agent's headset connection is always open including when outbound and inbound calls are routed through an ACD. The agent is always in the online state.

On an Agent Blending system, an agent's headset connection is broken when the Avaya Proactive Contact releases an ACD agent to inbound calling on the ACD. To the Avaya Proactive Contact, the ACD agent is offline but logged in to the Avaya Proactive Contact.

Use the following diagram to help identify the agent states to implement through your client application.



Commands and notification events

The Avaya Proactive Contact SDK consists of two major services, real-time events and real-time statistics. The Avaya Proactive Contact SDK uses the Common Object Request Broker (CORBA) Interface Definition Language (IDL) to define interfaces for network objects used by client applications.

The SDK IDL files consist of a client interface definition, server interface definition, and common data types. The EventClient.idl and EventServer.idl files use the data types defined in the EventTypes Common.idl and EventTypes.idl to display Avaya Proactive Contact events and statistics. Client applications implement the EventClient.idl, which is used by the Avaya Proactive Contact server application to return statistics and events data to the client application. The system's server application implements the EventServer.idl.

This chapter describes the common data types and data structures used to represent the real-time events and real-time statistics provided in the Common module of the Common.idl file, and the ESType module of the EventTypes.idl file. The real- time events and real-time statistics descriptions are organized by the corresponding method defined in the EventClient.idl.

This section contains the following topics:

SDK common data types

The SDK defines common data types to handle event and statistical information. The following table defines these data types.

Data Type	Parameters	Description
JobType	OUTBOUNDJOB INBOUNDJOB MANAGEDJOB BLENDJOB	Identifies the types of jobs available on the SDK: • Outbound jobs generate outbound calls • Inbound jobs receive inbound calls • Managed jobs is a special outbound job used for Managed Dialing where an agent previews the record of an outbound call before teh dialer places the call • Blend jobs generate outbound calls and receive inbound calls

AgentType	OUTBOUNDAGENT INBOUNDAGENT MANAGEDAGENT BLENDAGENT ACDAGENT PTPAGENT	Identifies the agent log in types available on the SDK: ● Outbound agents handle outbound calls only ● Inbound agents handle inbound calls only ● Managed agents can preview records for outbound calls before the number is dialed ● Blend agents handle both inbound and outbound calls ● ACD agents on Agent Blending systems handle outbound calls on Avaya Proactive Contact and inbound calls on the ACD ● Person to Person agents handle outbound calls when outbound or blend agents are unavailable
EventDataType Common Data	JOBNAME JOBNUMBER TYPEOFJOB DEVICEID EQUIPNUM EXITCODE	Identifies the data that is passed during agent event activity: ● Whether an agent goes online to join a job or be acquired from the ACD to handle Avaya Proactive Contact outbound calls ● The unique identifier for the call or for the agent headset ● The equipment number for the agent headset
EventDataType Call Data	DNIS ANI HEADSETEXTENSION TYPEOFAGENT AGENTNAME MESSAGEID PVLENGTH TRANSFERTYPE UCID	Identifies the following data that is passed during call event activity: ● Number dialed ● Number answered ● Agent type ● Agent name ● Identifier of the message played during a call ● Time an agent previews a record ● Type of transfer that occurred ● Universal Call Identifier

EventDataType Job Data	RECORDSELECT RECALLSELECT AVAILOUTBLINES AVAILINBLINES TOTALJOBTIME PARAM VALUE TYPEOFCALL USERDATA	Identifies the type of data that is passed during job event activity: ● Records selected or recalls selected for calling ● Number of available outbound and inbound lines ● Total time the job is active ● Whether the call is inbound or outbound ● User-defined data set up by the customer
EventDataType Agent Data	REASON CALLID AGENTDEVICEID AGENTEQUIPNUM SWITCH ID	Identifies the data that is passed during agent event activity: ● Whether an agent goes on line to join a job or be acquired from the ACD to handle Avaya Proactive Contact outbound calls ● Unique identifier for the call or for the agent headset ● Device identifier for the agentheadset. ● Equipment number for the agent headset ● Identifier for the switch

Generic data types

Avaya Inc. created a generic data type, LocText, to allow for localization because CORBA does not support the Registering data type. This data type is identified as LocText in the Common module.

LocText simply consists of a sequence of 8-bit octets you can use to store any type of character set, typically MBCS and ASCII. Text data passed to or from the Event Server will be sent using the LocText CORBA data type. The internal contents may be either standard ASCII strings or MBCS strings.

The KeywordValue and KeywordValueSeq lists consist of keywords and their corresponding values. Combine these basic data structures with other data structures to form composite data types.

The Keyword Value structure creates the tag value pair.

Type	**Variable**	**Definition**
LocText	keyword	The server defined keyword for the value.
LocText	value	The data associated with the keyword

The KeywordValueSeq list creates the sequenceofKeywordValue structs:

```
typedef sequence<KeywordValue> KeywordValueSeq;-.
```

There is an EventList sequence defined as a sequence of EVENT types:

```
enum EVENT {DIALERSTATUS, PASSWDSTATUS}
        typedef sequence<Common::EVENT>EventList;
```

There is an ID List sequence defined as a sequence of eventRegistration types:

Type	Variable
unsigned long	Id
Common::EVENT	eventName

Real-time events

The SDK generates real-time events when changes occur in the state of Avaya Proactive Contact agents, calls, and jobs. The SDK delivers the events to callback objects created by the client application.

The EventNotify methods for calls, agents, and jobs have unique headers and event type enumerations and structures.

This section identifies the data structures used to deliver the events to the method for the callback object. It also describes the structure of the method header and data segments.

This section contains the following methods:

- Call Event Notify
- Agent Event Notify
- Job Event Notify

CallEventNotify

The callEventNotify method definition is located in the EventClient module of the EventClient.idl.

The SDK uses this method to notify the client application of call events. Each call event contains a common event header structure, plus each has its own unique data segment information. CallEventNotify does not generate call events for dialback calls or acquire calls.

This section identifies the following call events and their data contents for all call event types found in EventTypes.idl:

- IDL declaration
- Call event header

- Call event types

IDL declaration

Below is the code for IDL Declaration:

```
oneway void callEventNotify(
in ESType:: EventDataSegment callEventData);
in ESType:: eventTypeKind eventTypeKind,
```

Call event header

The eventTypeKind describes the basic information about the call. All call event types share this eventTypeKind structure.

Type	Variable	Definition
CallEventType	eventType	Identifies the type of call event. For details, see Call event types.
long	timeStamp	A number that when converted identifies the date and time that the call event occurred. For example 954951746 converts to 04/05/2000 09:22:26. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970. This is given in time local to where the particular dialer server is sited.
short	dialerID	Identifies the dialer server in which the call event occurred. For example: 1, 2, 3.
CallIDType	callID	Uniquely identifies the call. For example: 000366095090001

Call event types

Each call event may have a different number of data elements in the data segment structure callEventData.

The callEventData data segment is a sequence of name-value pairs, one for each data element, per the following definitions from EventTypes.idl:

```
struct EventData
     {
       EventDataType   keyword;
       Common::LocText value;                                    // The client-
defined values
                                        //   for the Event data types
     };
typedef sequence<EventData> EventDataSeq;
```

For descriptions of the call events and their associated data elements, see:

- CALLINITIATED
- CALLDIALED
- CALLANSWERED
- CALLCONNECTED
- CALLDISCONNECTED
- CALLQUEUED
- CALLHELD
- CALLPREVIEWED
- CALLTRANSFER
- CALLCONSULTED
- CALLENDED
- CALLRETRIEVED
- CALLPLAYMSG

CALLINITIATED

The process of establishing a call between an agent and a far end party has started.

Keyword	Description
DEVICEID	The logical identification number the trunk on which the call is made. For example: 254
EQUIPNUM	The trunk number on which the call is made. For example: 25

ANI	The calling phone number. For example: 2065551212 or 4401753727211
DNIS	The phone number dialed. For example: 7185555555 or 4401777727211
JOBNAME	The name of the job with which the call is associated. For example: 30day
JOBNUMBER	The job number of the job as assigned by the dialer server. For example: 2
TYPEOFJOB	The job type. This can be one of the following types: OUTBOUNDJOB, INBOUNDJOB, MANAGEDJOB, or BLENDJOB. Managed job is a special outbound job used for Managed Dialing where an agent previews the record of an outbound call before the dialer places the call.
USERDATA	The customer's key data associated with this call. For example: IDENT: ACCTNUM 1234567890123456. This information comes from the job's calling list.
TYPEOFCALL	The call type, either OUTBOUNDCALL or INBOUNDCALL.

CALLDIALED

An outbound call has just started dialing.

Keywords	**Description**
DNIS	The phone number dialed. For example: 7185555555 or 4401777727211
UCID	The universal call identifier (UCID) of the associated call at the PBX. This value is valid for Avaya Proactive Contact with AES configurations. Otherwise, the value is zero (0).

CALLANSWERED

A call has been answered.

Keywords	**Description**
ANI	The calling phone number. For example: 2065551212 or 4401753727211

DNIS	The phone number dialed. For example: 7185555555 or 4401777727211
RESPONSECODE	One of the "system" type of completion code numeric values, as defined in the compcode.cfg configuration file. See *Avaya Proactive Contact 5.0 Configuration Reference* for a description of this file.

CALLCONNECTED

The agent is capable of controlling an answered call.

Keywords	**Description**
AGENTNAME	The log in name of the agent connected with the call. For example: jdoe
HEADSETEXTENSION	The headset identifier number for the agent connected to the call. For example: 2
AGENTDEVICEID	The logical identification number of the line that the agent is using. For example: 15
AGENTEQUIPNUM	The line number of the line that the agent is using. For example: 2510
USERDATA	The customer's key data associated with this call. For example: IDENT: ACCTNUM 1234567890123456. This information comes from the job's calling list.
SWITCHID	The switch identification of the switch that the agent is logged in to. Provided only for agents that do agent blending (which have TYPEOFAGENT = ACDAGENT in the AGENTOFFLINE event. For details, see AGENTOFFLINE. If the agent is not of this type, then this parameter is 0.

CALLDISCONNECTED

The call is disconnected.

Keywords	**Description**
none	There are no data segment items associated with this message.

CALLQUEUED

The call is in a wait queue, waiting for an agent to be come available.

Keywords	Description
none	There are no data segment items associated with this message.

CALLHELD

The call is on hold.

Keywords	Description
none	There are no data segment items associated with this message.

CALLPREVIEWED

An agent is previewing a the customer record in preparation for a managed call.

Keywords	Descriptions
AGENTNAME	The log in name of the agent previewing the call. For example: jdoe
HEADSETEXTENSION	The headset identifier number for the agent previewing the call. For example: 2
AGENTDEVICEID	The logical identification number of the line that the agent is using. For example: 15
AGENTEQUIPNUM	The line number of the line that the agent is using. For example: 2510
PVLENGTH	The amount of time, in seconds, that the call can be previewed during a Managed call. For example: 30 (seconds). A value of zero indicates that there is no timeout value.
SWITCHID	• The switch identification of the switch that the agent is logged in to. Provided only for agents that do agent blending. The agents have TYPEOFAGENT = ACDAGENT in the AGENTOFFLINE event that is described in CALLTRANSFER .

CALLTRANSFER

A call is being transferred

Keywords	Description
TRANSFERTYPE	Type of transfer. Values are: BLIND, SUPERVISED or TRUNK. TRUNK indicates a trunk-to-trunk transfer, which is a transfer to an extension, and it is always a supervised transfer. BLIND or SUPERVISED indicate a "Native Voice and Data Transfer", which is a transfer from an agent on an outbound job to an inbound job, where the dialer server picks the next available agent on the inbound job to receive the transfer. As the words indicate, this type of transfer can be either blind or supervised.
DNIS	If TRANSFERTYPE is TRUNK, then DNIS is the phone number to which the call is being transferred. For example, 4251245678. If TRANSFERTYPE is BLIND or SUPERVISED, then DNIS is the name of the job to which the call is being transferred. For example: inboundjobone
CALLID	The CALLID for the first leg of the call, the originating agent to customer.
CALLID	The CALLID for the second leg of the call, agent to agent.
CALLID	The CALLID for the third leg of the call, the receiving agent to customer.
EQUIPNUM	The trunk number that the call is transferred to.

CALLCONSULTED

An agent has established a consult call with another agent.

Keywords	Description
none	There are no data segment items associated with this message.

CALLENDED

The agent, system, or customer ended a call.

Keywords	Description
RESPONSECODE	One of the "system" type completion code numeric values, as defined in the compcode.cfg configuration file. See the document *Avaya Proactive Contact 5.0 Configuration Reference* for a description of this file.

CALLRETRIEVED

A call is retrieved from being on hold.

Keywords	Description
none	There are no data segment items associated with this message.

CALLPLAYMSG

A message is played to an answering machine or customer.

Keywords	Description
MessageID	The identifier of the message to play. For example: 351

AgentEventNotify

The agentEventNotify method definition is located in the EventClient module of the EventClient.idl.

The SDK uses this method to notify the client application of agent events. Each agent event contains a common event header structure, plus each has its own unique data segment information.

This section identifies the following agent events and their data contents for all agent event types found in EventTypes.idl:

- IDL declaration
- Event Type Kind
- Agent event types

IDL declaration

```
oneway void agentEventNotify (
in ESType:: eventTypeKind eventTypeKind,
in ESType:: EventDataSegment agentEventData);
```

Event Type Kind

The eventTypeKind describes the basic information about the agent. All agent event types share this eventTypeKind structure.

Type	Variable	Definition
AgentEventType	eventType	Identifies the type of agent event. For details, see Agent event types.
long	timeStamp	A number that, when converted, identifies the date and time that the agent event occurred, example 954951746 converts to 04/05/2000 09:22:26. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970. This is given in time local to where the particular dialer server is sited.
short	dialerID	Identifies the dialer server in which the agent event occurred. For example: 123.
long	agentID	The user ID of the agent. For example: 1001.
LocText	agentHsExt	The agent headset extension number.

SWITCHID	switchID	The switch identification of the switch that the agent is logged in to. Provided only for agents that do agent blending which have TYPEOFAGENT = ACDAGENT in the AGENTOFFLINE event. See Agent event types.

Agent event types

Each agent event may have a different number of data elements in the data segment structure `agentEventData`.

The agentEventData data segment is a sequence of name-value pairs, one for each data element, per the following definitions from EventTypes.idl:

```
struct EventData
    {
        EventDataType   keyword;
        Common::LocText value;            // The client-defined values
                                          //   for the Event data types
    };
typedef sequence<EventData> EventDataSeq;
```

For descriptions of the agent events and their associated data elements, see:

- AGENTLOGON
- AGENTNOTREADY
- AGENTREADY
- AGENTOFFLINE
- AGENTONLINE
- AGENTPREVIEW
- AGENTONCALL
- AGENTOFFCALL
- AGENTLOGOFF

AGENTLOGON

An agent logged in to Avaya Proactive Contact.

Keyword	Description
AGENTNAME	The UNIX login ID of the agent.

AGENTNOTREADY

The agent is on a job but not ready to receive calls.

Keyword	Description
COMPCODE	Completion code value that an agent assigns when releasing a customer record. This is one of the completion code numeric values, as defined in the compcode.cfg configuration file. See the document *Avaya Proactive Contact 5.0 Configuration Reference* for a description of this file. This may also show a value of -1. This happens the first time the agent reaches the Not Ready state after logging in, or if there is not a completion code to report at that time. This can occur when an ACD agent acquired for outbound is transverses before the agent handles any call.

AGENTREADY

The agent is ready to receive calls.

Keyword	Description
none	There are no data segment items associated with this message.

AGENTOFFLINE

Occurs after AGENTLOGON indicating that the agent headset number is validated.

At other times, this indicates that the agent released the line or has gone on break.

Keyword	Description
TYPEOFAGENT	The agent's type. One of the following types: OUTBOUNDAGENT, INBOUNDAGENT, MANAGEDAGENT, BLENDAGENT, ACDAGENT, or PTPAGENT.
REASON	Provides a reason why the agent is offline. One of the following reasons: Login, Released, NoFUrtherWork, Logoff, or Joblink.

AGENTONLINE

The agent headset is connected and the agent joined a job.

Keyword	Description

TYPEOFAGENT	The agent's type. One of the following agent types: OUTBOUNDAGENT, INBOUNDAGENT, MANAGEDAGENT, BLENDAGENT, ACDAGENT, or PTPAGENT
AGENTDEVICEID	The logical identification number of the line that the agent is using. For example: 15
AGENTEQUIPNUM	The line number of the line that the agent is using. For example: 2510
JOBNAME	Identifies the name of the job to which the agent is assigned. For example: 30day
JOBNUMBER	The job number assigned by the dialer server. For example: 2
TYPEOFJOB	The job type. One of the following types: OUTBOUNDJOB, INBOUNDJOB, MANAGEDJOB, BLENDJOB. Managed job is a special outbound job used for Managed Dialing where an agent previews the record of an outbound call before the dialer places the call.
REASON	Provides a reason why the agent is on the named job. One of the following reasons: Acquired or JoinJob.

AGENTPREVIEW

The agent is previewing a record on a managed call.

Keyword	**Description**
TYPEOFCALL	The call type. For this event it is always OUTBOUNDCALL.
CALLID	Uniquely identifies the call. For example 000366095090001
DEVICEID	The logical device identification number. For example: 254
EQUIPNUM	The physical equipment number. For example 25
DNIS	The phone number to be dialed. For example 4255588694
USERDATA	The customer's key data associated with this call. For example: IDENT: ACCTNUM 1234567890123456. This information comes from the job's calling list.

AGENTONCALL

The agent is connected to a call.

Keyword	Description
TYPEOFCALL	Identifies the call as OUTBOUNDCALL or INBOUNDCALL.
CALLID	Uniquely identifies the call. For example: 000366095090001
DEVICEID	The logical device identification number. For example: 254
EQUIPNUM	The physical equipment number. For example: 25
DNIS	The phone number dialed. For example: 7185555555 or 4401777727211
USERDATA	The customer's key data associated with this call. For example: IDENT: ACCTNUM 1234567890123456. This information comes from the job's calling list.

AGENTOFFCALL

The agent or the far end party has released the call.

Keyword	Description
none	There are no data segment items associated with this message.

AGENTLOGOFF

The agent has logged off.

Keyword	Description
EXITCODE	Provides an event code that identifies the reason an agent logged off: Possible values are Normal or Abort.

JobEventNotify

The jobEventNotify method definition is located in the EventClient module of the EventClient.idl.

The SDK uses this method to notify the client application of a job event. Each job event contains a common event header structure, plus each has its own unique data segment information.

This section identifies the following job event header and data structures found in EventTypes.idl:

- IDL declaration
- Job event header
- Job event types

IDL declaration

```
oneway void job EventNotify(
in ESType:: eventTypeKind eventTypeKind,
in ESType:: EventDataSegment jobEventData);
```

Job event header

The eventTypeKind describes the basic information about job events. All job event types share this eventTypeKind structure.

type	variable	Definition
JobEventType	eventType	Identifies the type of job event. For details, see Job event types .
long	timeStamp	A number that, when converted, identifies the date and time that the call event occurred. For example 954951746 converts to 04/05/2000 09:22:26. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970. This is given in time local to where the particular dialer server is sited.
short	dialerID	Identifies the Avaya Proactive Contact in which the call event occurred. For example: 123.
long	jobNumber	The job number assigned by the Avaya Proactive Contact for example: 2.

Job event types

Each job event may have a different number of data elements in the data segment structure `jobEventData`.

The jobEventData data segment is a sequence of name-value pairs, one for each data element, per the following definitions from EventTypes.idl:

```
struct EventData
    {
        EventDataType   keyword;
        Common::LocText value;            // The client-defined values
                                          //   for the Event data types
    };
typedef sequence<EventData> EventDataSeq;
```

For descriptions of the job events and their associated data elements, see:

- JOBSTARTED
- JOBEVENTS
- JOBDYNAMICCHANGE
- JOBENDED

JOBSTARTED

Basic job parameters used when the Avaya Proactive Contact starts a job.

Keyword	Description
JOBNAME	The name of the job started. For example: 30day
TYPEOFJOB	The job type started. For example: OUTBOUNDJOB, INBOUNDJOB MANAGEDJOB, BLENDJOB, CRUISE_JOB. Cruise job is a special outbound job where the dialer automatically monitors and adjust the dialing pace throughout the outbound job.

Message content example:
`event = JOBSTARTED`
`timeStamp = 1112981212(10:26:52)`
`dialerID = 16`
`jobNumber = 29`
`JOBNAME = perf_out1`
`TYPEOFJOB = CRUISE_JOB`	

RECORDSELECT	The number of records selected for this job. For example: 10527 (Outbound only)
RECALLSELECT	The number of recalls selected for this job. For example: 35 (Outbound only)

AVAILOUTBLINES	The total number of outbound lines available for this job. For example: 240 (Outbound only)
AVAILINBLINES	The total number of inbound lines available for this job. For example: 120 (Inbound only)

JOBEVENTS

Various events that occur during a job

Keyword	**Description**
TERMINATED	The job ended abruptly. The value will be '1'.
SHUTDOWN	The job was ended manually. The value will be '1'.
LINKED	The job was linked to another job. For example: LINKED outb2
E_RECORDCNT	The number of records in the job.
E_RECALLCNT	The number of recall times in records in the job.
E_OUTLINES	The number of lines for outbound for the job.
E_INLINES	The number of lines for inbound for the job.

Message content example:
`event = JOBEVENTS`
`timeStamp = 1112981214(10:26:54)`
`dialerID = 16`
`jobNumber = 29`
`PARAM = E_RECORDCNT`
`VALUE = 62794`
`PARAM = E_RECALLCNT`
`VALUE = 0`
`PARAM = E_OUTLINES`
`VALUE = 192`
`PARAM = E_INLINES`
`VALUE = 0`	

E_TIMESTAMP	Timestamp as an integer in seconds since 1970.
E_LINENUM	Line number in the configuration code table, starting at 1.
E_RAC_DEF	Contents of a configuration code table line.

Message content example:

```
event = JOBEVENTS
timeStamp = 1112981214(10:26:54)
dialerID = 16
jobNumber = 29
     PARAM = E_TIMESTAMP
     VALUE = 1112916615
     PARAM = E_LINENUM
     VALUE = 1
     PARAM = E_RAC_DEF
     VALUE = 0:NOTCALLED:SYS:YES:NO:NO:NO:NO:YES:NOTCALLED:Record not yet
called
```

E_SCRIPT	Data script (.dat) file name.
E_IDCNTRL	Expert Calling Ratio.
E_JLABEL	Job description.
E_LIST	Outbound calling list name.
E_SELECT	Record selection file name.
E_STRATEGY	Call Strategy file name.
E_JOBTYPE	Indicates whether the job is an inbound, outbound, blend, managed, or cruise control job. Values are INB, OUT, BLND, MNGD, and CRSE.

Message content example:

```
event = JOBEVENTS
timeStamp = 1112981234(10:27:14)
dialerID = 16
jobNumber = 29
     PARAM = E_SCRIPT
     VALUE = alljobs
     PARAM = E_IDCNTRL
     VALUE = W80
     PARAM = E_JLABEL
     VALUE = Outbound Job
     PARAM = E_LIST
     VALUE = list1
     PARAM = E_SELECT
     VALUE = test
     PARAM = E_STRATEGY
     VALUE = phone1
     PARAM = E_JOBTYPE
     VALUE = CRSE
```

E_JOIN	Indicates that an agent joined the job. The actual value is the type of agent, one of the following job types: INBOUND, OUTBOUND, MANAGED, BLEND, or PTP.
E_AGENTNAME	The name of the agent that joined the job.

Message content example:
`event = JOBEVENTS`
`timeStamp = 1112981253(10:27:33)`
`dialerID = 16`
`jobNumber = 29`
`PARAM = E_JOIN`
`VALUE = OUTBOUND`
`PARAM = E_AGENTNAME`
`VALUE = op9test`	

E_AGENTNAME	The agent name associated with this event. For example: jdoe
E_IDLE	The time duration during which the agent was not working on calls or customer records. This applies to the agent named with the AGENTNAME parameter, which appears after the IDLE parameter,
E_IDLETYPE	The type of idle activity. For example: I=inbound, O=outbound This applies to the agent named with the AGENTNAME parameter, which appears after the IDLE parameter,

Message content example:
`event = JOBEVENTS`
`timeStamp = 1112829018(16:10:18)`
`dialerID = 11`
`jobNumber = 13`
`PARAM = E_IDLE`
`VALUE = 6`
`PARAM = E_AGENTNAME`
`VALUE = bp121tes`
`PARAM = E_IDLETYPE`
`VALUE = O`	

RECORDSELECT	The number of records selected for this job. For example: 10527 (Outbound only)
RECALLSELECT	The number of recalls selected for this job. For example: 35 (Outbound only)
AVAILOUTBLINES	The total number of outbound lines available for this job. For example: 240 (Outbound only)

AVAILINBLINES	The total number of inbound lines available for this job. For example: 120 (Inbound only)
E_HIT_RATES	The number of calls connected for that type compared to the total number of call attempts for this job. Examples: First 32/37, Busy 0/3, Noanswer 0/0, Callback 0/0, Misc 0/0 First 19/46, Busy 1/4, Noanswer 0/0, Callback 0/0, Misc 0/0

Message content example:

```
event = JOBEVENTS
timeStamp = 1112829027(16:10:27)
dialerID = 11
jobNumber = 13
     PARAM = E_HIT_RATES
     VALUE = First 32/47, Busy 0/3, Noanswer 0/0, Callback 0/0, Misc 0/0
```

E_CURR_HIT_RATE	The hit rate in a percentage during the last five minutes for this job. Examples: Hit Rate[phone1] = 39enclinet

Message content example:

```
event = JOBEVENTS
timeStamp = 1112829035(16:10:35)
dialerID = 11
jobNumber = 13
     PARAM = E_CUR_HIT_RATE
     VALUE = Hit Rate[phone1] = 39
```

E_LINE_USAGE	The number of assigned lines used during the peak demand. Examples: Lines Assigned = 528, Peak Demand = 38 Peak Demand is the most recent largest number of lines in use. Line Assigned is the current umber of lines assigned to this job.
E_CURR_DIALING	The current Expert Calling Ratio method in use for this job. Examples: IDM = W85, Dial Ahead = 155, Q Rate = 82, Ops = O P

Message content example:

```
event = JOBEVENTS
timeStamp = 1112829035(16:10:35)
dialerID = 11
jobNumber = 13
     PARAM = E_LINE_USAGE
     VALUE = Lines Assigned = 528, Peak Demand = 38
     PARAM = E_CURR_DIALING
     VALUE = IDM = W85, Dial Ahead = 155, Q Rate = 82, Ops = 0  P
```

This set of values is used in the job events for a cruise control job. Five data elements are reported every 2 minutes.

E_CC	Cruise control flag for the job. A value of 1 indicates that this is a cruise control job, and 0 indicates it is not a cruise control job.
E_DSL	Desired service level
E_CONNTOLE	Job connection tolerance
E_SERVCALLS	Job serviced calls within tolerance
E_OFFEREDCALLS	Job total calls that demand agent connection
E_CC_PARAMS	Job running parameters; this appears only with cruise control jobs.

Message content example for a cruise control job:

```
event = JOBEVENTS
timeStamp = 1112981455(10:30:55)
dialerID = 16
jobNumber = 29
     PARAM = E_CC
     VALUE = 1
     PARAM = E_DSL
     VALUE = 0.900000
     PARAM = E_CONNTOLE
     VALUE = 1
     PARAM = E_SERVCALLS
     VALUE = 69
     PARAM = E_OFFEREDCALLS
     VALUE = 84
     PARAM = E_CC_PARAMS
     VALUE =
Hitrate=0.310000,Qrlimit=0.004433,Noagent_r=0.009138,QrRatio=0.651420
```

Message content example for a non-cruise control job:

```
event = JOBEVENTS
timeStamp = 1112829035(16:10:35)
dialerID = 11
jobNumber = 13
     PARAM = E_CC
     VALUE = 0
     PARAM = E_DSL
     VALUE = 0.990000
     PARAM = E_CONNTOLE
     VALUE = 1
     PARAM = E_SERVCALLS
     VALUE = 10785
     PARAM = E_OFFEREDCALLS
     VALUE = 11663
```

E_LEAVE	Indicates that the agent left the job.
E_AGENTNAME	The name of the agent that left the job.

Message content example:

```
event = JOBEVENTS
timeStamp = 1112981595(10:33:15)
dialerID = 16
jobNumber = 29
     PARAM = E_LEAVE
     VALUE = 342
     PARAM = E_AGENTNAME
     VALUE = op38test
```

JOBDYNAMICCHANGE

Job parameters that are modified during a job.

Keyword (parameter)	**Description (value)**
CALLSTGY	A number indicates which aspect of the call strategy has been changed. 1=Alternate initial phone setting was changed, 2=Recall setting was changed, 3=Progress setting was changed.
ORDERZONE	This is a 0/1 toggle which asks if you want to order your calls by time zone. If value is '1', calls will be made by time zone (east to west). If value is '0', calls will be placed 'round-robin' style.
VIEWCNTRL	Preview time limit in seconds that an agent can preview a record before the Proactive Contact dials the record. 0=infinite preview time. n=n second preview time.
DIALCNTRL	1=YES; Allows agents to cancel the call before Proactive Contact dials the previewed record. 0=NO; Disallows agents to cancel a call before Proactive Contact dials the previewed record for managed calls only.
IDCNTRL	This is the Expert Calling ratio. The range is between 0% - 100%. 0% dial only when an agent is available for a call and puts no calls in a queue. 100% = full capacity of Proactive Contact.

Message content example:

```
event = JOBDYNAMICCHANGE
timeStamp = 1112981422(10:30:22)
dialerID = 16
jobNumber = 29
     PARAM = E_PARAM
     VALUE = IDCNTRL
     PARAM = E_VALUE
     VALUE = W70
```

SERVETIME	The amount of time the Avaya Proactive Contact allows as a cushion between transferring an agent from an inbound job to an outbound job.
INBQUEFACTOR	The maximum number of calls that will be allowed into the inbound wait queue before Proactive Contact assigns a Blend agent to take inbound calls.
RETURNTIME	The amount of time a Blend agent can be idle before returning to outbound calling.
LINKJOB	This is the name of the job that is linked to the current job.
MIN_HITRATE	A parameter that tells the Avaya Proactive Contact the lowest number of call attempts it will place before achieving a connection. The system uses the minimum hit rate to adjust the number of lines a job is using, therefore adjusting the speed of dialing.
ADDLINETYPE	States the name of the line pool that was added to the job.
DELETELINETYPE	States the name of the line pool that was removed from the job.

JOBENDED

Information provided when a job ends.

Keyword	Description
EXITCODE	Provides an exit code that identifies the reason the job ended Values 0=Normal or 1=Aborted
TOTALJOBTIME	The cumulated time a job is active with at least one agent.

Real-time statistics

Real-time statistics are organized into four categories: system, job, agent, and phone line.

This section describes the data structures that the SDK uses to deliver statistical information to client applications.

This section contains the following methods:

- systemStatNotify
- JobStatNotify
- AgentStatNotify
- LineStatNotify

systemStatNotify

The systemStatNotify method definition is located in the EventClient module of the EventClient.idl.

The SDK uses this method to deliver Avaya Proactive Contact general system configuration statistics to client applications. The SystemStatData structure provides basic configured information about Avaya Proactive Contact.

This section identifies the following system statistics data structures found in the EventTypes.idl:

- IDL declaration
- Contents of struct SystemStatData
- Contents of struct LinePoolData

IDL declaration

```
oneway void systemStatNotify(
in ESType:: SystemStatData systemStatData);
```

Contents of struct SystemStatData

Type	Variable	Definition
long	timeStamp	A number that, when converted, identifies the current date and time for Avaya Proactive Contact. For example 954951746 converts to 04/05/2000 09:22:26. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970. This is given in time local to where the particular dialer server is sited.
short	dialerID	Unique Identifier of the dialer server.
long	systemLines	The number of ports listed in master.cfg for the Avaya Proactive Contact and Avaya Proactive Contact with Avaya Proactive Contact Gateway PG230 systems. The number of lines that can be used for Avaya Proactive Contact with AES systems.
long	systemAgents	The maximum number of agents allowed on the dialer server.
long	systemJobs	The maximum number of jobs allowed on the dialer server.
long	systemUpdateTime	The update interval, in seconds, at which Avaya Proactive Contact returns data to the client application. Note that the Avaya Proactive Contact load will affect the update frequency.
LocText	timeZone	The time where the dialer is located.
LinePoolSeq	linePoolList	List of line counts by line pool. A sequence of struct LinePoolData. See Contents of struct LinePoolData

Contents of struct LinePoolData

Type	Variable	Definition
LocText	linePool	Line Pool Name
octet	lineType	I=Inbound or O=Outbound for Avaya Proactive Contact and Avaya Proactive Contact with PDG230 systems. O=Outbound for Avaya Proactive Contact with AES systems.
short	lineCount	Total number of lines in this Line Pool.

JobStatNotify

The jobStatNotify method definition is located in the EventClient module of the EventClient.idl.

The SDK uses this method to deliver job statistics to client applications.

This section identifies the following job statistic data structures found in EventTypes.idl:

- IDL declaration
- Contents of struct JobStatData
- Contents of struct JobData
- Contents of struct StaticJobData
- Contents of struct DynamicJobData
- Contents of struct AgentCountData

IDL declaration

```
oneway void jobStatNotify
(in EType::JobStatData jobData);
```

Contents of struct JobStatData

Type	Variable	Definition
long	timeStamp	A number that, when converted, identifies the current date and time for Avaya Proactive Contact. For example 954951746 converts to 04/05/2000 09:22:26. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970. This is given in time local to where the particular dialer server is sited.
short	dialerID	Unique identifier of the dialer server.
JobDataSeq	jobDataList	List of JobData structs.

Contents of struct JobData

Type	Variable	Definition
StaticJobData	staticJob	**Static information for an active job. See Contents of struct StaticJobData.** **Contents of struct StaticJobData**
DynamicJobData	dynamicJob	Dynamic statical information for an active job. See Contents of struct DynamicJobData.

Contents of struct StaticJobData

StaticJobData provides information about an active job that does not change during the job.

Type	Variable	Definition
LocText	jobName	The job name. For example: 30day.
LocText	callingList	Calling list name.
LocText	recordSelectionFile	The name of the job's record selection file. For example: 30day. Blank for inbound jobs.
LocText	phoneStrategyFile	The name of the job's phone strategy file. For example: 30day. Blank for inbound jobs.
long	jobNumber	The job number assigned by the Avaya Proactive Contact for example: 2.
long	totalRecordsToCall	The total number of records selected for calling. For example: 15,802. Zero for inbound jobs.
long	jobStartTimeStamp	The date and time that the job started running on the Avaya Proactive Contact. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970.
long	jobEndTimeStamp	The time at which the job ended. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970.
short	jobSlot	The shared memory slot on the Avaya Proactive Contact that is assigned to the job. For example: 2.
short	linesAssigned	The number of telephone lines assigned to each job that is monitored on the Avaya Proactive Contact. For example: 120.
octet	jobType	I, B, O, M, or C, designating respectively an inbound, blend, outbound, managed, or cruise control job

Contents of struct DynamicJobData

DynamicJobData provides statistical information about an active job that changes during the job.

type	variable	Definition
boolean	cruiseControl	Indicates whether or not the job is a cruise control job.
float	desiredServiceLevel	Desired service level.
long	connectTolerance	Call connection tolerance.
long	servicedCalls	Number of calls serviced within Tolerance.
long	offeredCalls	Number of calls that demand an agent connection.
long	runningHitRate	The overall hit rate for the job since the job started, which is calculated by measuring the percentage of call completions measured against call attempts. For example: 40.
long	currentHitRate	The hit rate for the job during the last 5 to 10 minutes, which is calculated by measuring the percentage of call completions against call attempts; Avaya Proactive Contact uses this number to make adjustments to the dialing model. For example: 42.
long	recordsCalled	The total number of calls made or handled for the job includes both inbound and outbound calls. For example: 6059.
long	recordsAvailable	The number of eligible records not yet called for the job. For example: 39230. Blank for inbound jobs.
long	recordsRecalled	The number of recalls remaining for the job. For example: 218. Blank for inbound jobs.
long	inbConnects	The total number of inbound connections since the job started. For example: 859.

long	outbConnects	The total number of outbound connections since the job started. For example: 1784.
long	inbTotalQueCalls	The total number of inbound calls placed in the wait queue since the job started. For example: 300.
long	inbOutQueCalls	Total number of calls removed from the inbound queue. For example: 280.
long	inbAverageQueTime	The average amount of time that inbound calls were in the wait queue. For example: 10 (seconds).
long	inbTotalQueTime	The total queue time for inbound calls. For example: 360 (seconds).
long	outbTotalQueCalls	The number of outbound calls placed in the wait queue since the job started. For example: 300.
long	outbOutQueCalls	Total calls removed from the outbound queue. For example: 275.
long	outbAverageQueTime	The average amount of time that outbound calls were in the wait queue. For example: 4 (seconds).
long	outbTotalQueTime	The total queue time for outbound calls. For example: 240 (seconds).
long	jobCallsAnswered	JOB total IN/OUT connects.
long	jobCallsInWait	JOB total IN/OUT calls in WAITQ.
long	jobCallsWorked	JOB total IN/OUT calls worked.
long	jobIdleCount	JOB total idle.
long	jobIdleTime	JOB total idle time in seconds.
long	jobTalkTime	JOB total talk time in seconds.
long	jobUpdateTime	JOB total update time in seconds.

long	jobWaitQueueTime	JOB total waitq time in seconds.
long	jobWorkTime	JOB total work time in seconds.
long	mgdPreviewTime	Total preview time for managed calls.
octet	jobStatus	Identifies the job's current status. Values include: 0=active 1=job finished setup phase 2=job in shutdown phase 3=no more calls to ops
ExtJobDataSeq	extJobDataList	Sequence of extended job statistical data groups. One of several possible structures with further information. The possibilities are: ExtPtpJobData ExtInbJobData ExtOutJobData ExtAuxJobData
CompCodeSeq	compCodeList	Array of all possible outbound completion codes. The values in the array are the counts for each completion code. For example, if you had twenty instances of code 19 (RECALL), the values would be: `outbCallCompletionCode[19] = 20`
AgentCountSeq	agentCountList	Composite of Agent Count information. This is a sequence of AgentCountData structs. See Contents of struct AgentCountData.
LocText	expertCalling	The current rate at which Avaya Proactive Contact predicts when to make the next call. Avaya Proactive Contact uses one of the following settings to adjust the calling pace: **Calls in the wait queue** achieves a balance between agents waiting for a call and clients placed in the wait queue. **Agent work time** monitors the time agents take to complete calls and update records. **Agent update time** monitors the time agents take to update records. Expert Calling is not used for inbound calls.

Contents of struct AgentCountData

The data in AgentCountData states how many agents of a given login type are present. A sequence of these is provided in the dynamic job data.

Type	Variable	Definition
short	count	Current agent count
octet	type	Agent login type {I, O, B, M, P, Z} 73=I for Inbound 79=O for Outbound 66=B for Blend 77=M for Managed 80=P for Person-to-Person 90=Z for type not selected yet

AgentStatNotify

The agentStatNotify method definition is located in the EventClient module of the EventClient.idl. The SDK uses this method to deliver agent statistics to client applications. This section identifies the agent statistic data structures found in the ESCommonTypes.idl.

Agent information exists from the time an agent logs on to the Avaya Proactive Contact until he or she logs off. The system stores agent data on a job by job basis. To retrieve agent statistical data for a job, look up the number for each job associated with each agent data structure, then the corresponding job status for the job number.

This section contains the following topics:

IDL declaration

```
oneway void agentStatNotify (
in ESCommon:: AgentStatData agentStatData);
```

Contents of struct AgentStatData

Type	Variable	Definition

long	timeStamp	A number that, when converted, identifies the current date and time for Avaya Proactive Contact. For example 954951746 converts to 04/05/2000 09:22:26. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970. This is given in time local to where the particular dialer server is sited.
short	dialerID	Unique identifier of the dialer server
AgentDataSeq	agentDataList	List of AgentData structs, which is comprised of agent session data and agent dynamic data.

Contents of struct AgentSessionData

AgentSessionData provides statistical information for an agent's session:.

Type	Variable	Definition
LocText	agentName	The agent login user name. For example: jdoe
LocText	wkStn	The agent's workstation identifier. This is a tty port or a pseudo tty port (the data port). For example: 4
LocText	headsetID	The agent's headset ID, a numerical value.
octet	currentType	The agent's current type: 79 = O outbound 73 = I inbound 90 = Z not on a call.
octet	currentState	An agent's current state: 84 = T for talk 85 = U update 73 = I for idle 76 = L transferring to another job 79 = O off job 82 = R released to inbound 88 = X logged off system 90 = Z log off requested
long	currentJobNumber	The number of the job that the agent is currently working on. For example 57.

long	loginTimeStamp	The time at which agent logged in to the dialer system. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970. This is given in time local to where the particular dialer server is sited.
long	offlineTimeStamp	The time an agent logged off last job: ● Set to sysLoginTimeStamp on log in ● Set to time agent reached jobname prompt after selecting hid The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970. This is given in time local to where the particular dialer server is sited.
long	acqFmAcdTimeStamp	The time an agent was acquired from inbound calling to handle outbound calls. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970. This is given in time local to where the particular dialer server is sited.
long	relToAcdTimeStamp	The time an agent was released from outbound calling to handle inbound calls. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970. This is given in time local to where the particular dialer server is sited.
long	elapsedJobTime	Total of all time on jobs, in seconds, including the current job, if any.
long	elapsedSessionTime	Total of all time logged in, in seconds, including current session.

Contents of struct AgentDynDataPerJob

AgentDynDataPerJob provides statistical information for an agent by job.

Type	Variable	Definition
LocText	jobName	The job name. For example: 30day.

LocText	unitID	The unit identifier for the job. For example: field "state" = WA or “state” = Washington. For multi unti selection feature: If agent selects more than one unit, unitID field contains “Allid”. If agent selects one unit, unitID fileld contains selected unit.
octet	agentType	Type of agent: 73 = I for Inbound, 79 = O for Outbound, 66 = B for Blend, 77 = M for Managed, 80 = P for Person to Person, 65 = A for ACD
long	jobNumber	The number of the job that the agent was logged in to. For example: 85. If the value of the job number is -1 or 50, the data within the structure is invalid. While the agent is no longer logged in to this job, it may still be active.
long	lastLogonTimeStamp	A number that, when converted, identifies the date and time when the agent logged out of the Avaya Proactive Contact job. For example, 954951746 converts to 04/05/2000 09:22:26. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970. This is given in time local to where the particular dialer server is sited.
long	lastLogoffTimeStamp	A number that, when converted, identifies the date and time that the last time an agent logged out of the job. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970. This is given in time local to where the particular dialer server is sited.
long	lastStatusChgTimeStamp	A number that when converted identifies the date and time of the agent's last status change. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970. This is given in time local to where the particular dialer server is sited.

long	mgdPreviewTime	Total preview time for managed calls.
long	totalCallsWorked	Number of calls worked, a total for all inbound and outbound calls.
long	totalIdleCount	The number of times an agent is idle between calls, a total for all inbound and outbound calls.
long	totalIdleTime	The time the agent has been idle, a total for all inbound and outbound calls.
long	totalTalkTime	Talk time, a total for all inbound and outbound calls.
long	totalUpdateTime	Time updating customer records, a total for all inbound and outbound calls.
long	totalWorkTime	Work time, a total for all inbound and outbound calls
ExtAgentDataSeq	extAgentDataList	This will be one of several possible extended agent data structs. The possibilities are: ExtPtpAgentData ExtInbAgentData ExtOutAgentData ExtAuxAgentData
CompCodeSeq	compCodeList	Array of all possible outbound completion codes. The values in the array are the counts for each completion code. For example, if you had twenty instances of code 19 (RECALL), the values would be: `outbCallCompletionCode[19] = 20`.

LineStatNotify

The lineStatNotify method definition is located in the EventClient module of the EventClient.idl. LineStatData checks the status of telephone lines on Avaya Proactive Contact. The SDK uses this method to deliver phone line statistics to client applications.

This section identifies the following line statistic data structures found in the ESCommonTypes.idl:

- IDL declaration
- Contents of struct LineStatData

- Contents of struct PoolData
- Contents of struct LineData

IDL declaration

```
oneway void LineStatNotify
     in ESCommon:: LineStatData lineStatData);
```

Contents of struct LineStatData

Type	Variable	Definition
long	timeStamp	The current System time. This is a number that when converted identifies the date and time of the agent's last status change. The system returns the number stored as a 32-bit long that contains the number of seconds since January 1, 1970. This is given in time local to where the particular dialer server is sited.
short	dialerID	Unique identifier of the dialer server.
LineSeq	lineList	Composite of line use info by Jobnumber and Line Pool name.

Contents of struct PoolData

Type	Variable	Definition
LocText	linePool	Line pool name
short	linesInUse	Number of lines in use

Contents of struct LineData

Type	Variable	Definition
long	jobNumber	Job number
PoolSeq	poolList	List of lines in use by pool

Using the Client IDL

Client applications implement the EventClient.idl, which is used by the Avaya Proactive Contact server application. The EventClient.idl defines the client interface. It also provides the real-time event report interface. The Avaya Proactive Contact SDK uses this interface to send the client application callback data such as real-time events and real-time statistics defined in the ESCommonTypes.idl. This section describes the EventClient.idl.

This section contains the following topics:

CallEventNotify

The callEventNotify method is located in the EventStatIF interface of the EventClient module in the EventClient.idl. The SDK uses this method to notify a client application of call events.

IDL declaration

```
oneway void callEventNotify
     in ESCommon:: eventTypeKind eventTypeKind,
     in ESCommon:: EventDataSegment callEventData);
```

Parameter	Description
eventTypeKind	Describes the basic information about the call. For example, call ID and time stamp.
callEventData	A data segment that carries the specific data for an individual event.

Exceptions

See Avaya Proactive Contact SDK exceptions for a description of standard CORBA and TAO exceptions.

Examples

See an example in Java sample application.

AgentEventNotify

The agentEventNotify method is located in the EventStatIF interface of the EventClient module in the EventClient.idl. The SDK uses this method to notify the client application of all agent events.

IDL declaration

```
oneway void agentEventNotify
     in ESCommon:: eventTypeKind eventTypeKind,
     in ESCommon:: EventDataSegment agentEventData);
```

Parameters	Description
eventTypeKind	Describes the basic information about agents, such as agent name.
agentEventData	A data segment that delivers specific data for an agent event, such as ExitCode.

Exceptions

See Avaya Proactive Contact SDK exceptions for a description of standard CORBA and TAO exceptions.

Examples

See an example in Java sample application.

JobEventNotify

The jobEventNotify method is located in the EventStatIF interface of the EventClient module in the EventClient.idl. The SDK uses this method to notify the client application of job events.

IDL declaration

```
oneway void jobEventNotify (
     in ESCommon:: eventTypeKind eventTypeKind,
     in ESCommon:: EventDataSegment jobEventData);
```

Parameters	Description
eventTypeKind	Describes the basic information about jobs, such as job name.

jobEventData	A data segment that describes specific information for individual job events.

Exceptions

See Avaya Proactive Contact SDK exceptions for a description of standard CORBA and TAO exceptions.

Examples

See an example in Java sample application.

SystemStatNotify

The systemStatNotify method is located in the EventStatIF interface of the EventClient module in the EventClient.idl. The SDK uses this method to deliver statistics about the Avaya Proactive Contact.

IDL declaration

```
oneway void systemStatNotify (
     in ESCommon:: SystemStatData systemStatData);
```

Parameter	Description
systemStatData	The SDK provides basic system information in this structure, including system time, number of system lines in use, number of agents on the system, number of running jobs, and the time the system updates the data.

Exceptions

See Avaya Proactive Contact SDK exceptions for a description of standard CORBA and TAO exceptions.

Examples

See an example in Java sample application.

JobStatNotify

The jobStatNotify method is located in the EventStatIF interface of the EventClient module in the EventClient.idl. The SDK uses this method to deliver statistics about active jobs.

IDL declaration

```
oneway void jobStatNotify (
     in ESType:: JobStatData jobStatData);
```

Parameter	Description

jobStatData	The SDK provides basic job information in this structure, including the record selection and phone strategy used for calling, the job number, and the start time.

Exceptions

See Avaya Proactive Contact SDK exceptions for a description of standard CORBA and TAO exceptions.

Examples

See an example in Java sample application.

AgentStatNotify

The agentStatNotify method is located in the EventStatIF interface of the EventClient module in the EventClient.idl. The SDK uses this method to deliver statistics about agents.

Agent information exists from the time an agent logs on to the Avaya Proactive Contact until he or she logs off. The system stores agent data by job. To retrieve agent statistical data for a job, look up the number for each job associated with each agent data structure, then look up the corresponding job status for the job number.

IDL declaration

```
oneway void agentStatNotify (
     in ESType:: AgentStatData agentStatData);
```

Parameters	Description
agentStatData	The SDK provides basic agent information in this structure, including the agent name, login type; current state and job number; time spent updating calls, talking to customers, and being idle.

Exceptions

See Avaya Proactive Contact SDK exceptions for a description of standard CORBA and TAO exceptions.

Examples

See an example in Java sample application .

LineStatNotify

The lineStatNotify method is located in the EventStatIF interface of the EventClient module in the EventClient.idl. The SDK uses this method to deliver statistics about lines.

IDL declaration

```
oneway void lineStatNotify (
     in ESType:: LineStatData lineStatData);
```

Parameters	Description
lineStatData	The SDK provides basic telephone information in this structure, including jobs assigned to a line and whether the line is ready, unassigned, or in use.

Exceptions

See Avaya Proactive Contact SDK exceptions for a description of standard CORBA and TAO exceptions.

Examples

See an example in Java sample application.

Using the Server IDL

The Avaya Proactive Contact server application implements the EventServer.idl. The EventService.idl defines the server interface. It provides the interfaces that client applications use to retrieve event and statistical reports defined in ESCommonTypes.idl.

This section describes the EventServer.idl.

This section contains the following topics:

Logon

This login method is located in the DialerEventServerIF interface of the EventServer module in the EventServer.idl. This method attempts to authenticate the client application name and password on the local Avaya Proactive Contact.

When the Avaya Proactive Contact SDK receives a logon request from the client, it creates an object that implements the SDK interface as defined by the IDL and returns a reference to that object to the client application.

The Avaya Proactive Contact provides a configured default password. In the client application, verify that this password meets the customer's requirement.

IDL declaration

```
EventServiceIF logon (
     in ESCommon:: LocText clientName,
     in ESCommon:: LocText passwd)
     raises (ESError);
```

Parameters	Description
clientName	The valid login name for the client application
passwd	The encrypted password for the client application

Exceptions

```
AUTHORIZATIONFAILED INTERNALERROR PASSWDEXPIRED INCOMPATIBLEVERSION
NOTREADY TOOMANYCLIENTS
```

See Avaya Proactive Contact SDK exceptions for descriptions of these Event Service exceptions.

Logoff

This logout method is located in the DialerEventServerIF interface of the EventServer module in the EventServer.idl. The logoff method cleans up lingering data structures and pointers associated with the client and discards the service object.

When the Event Service receives a logoff request from a client with the object, it reclaims the object and performs cleanup work.

IDL declaration

```
void logoff
     in EventServiceIF eventService)
     raises (ESError);
```

Parameters	Description
eventService	Returns the service object created when the logon method was successful.

Exceptions

```
INVALIDOBJECT NOTLOGON
NOTREADY INTERNALERROR
```

See Avaya Proactive Contact SDK exceptions for descriptions of these Event Service exceptions.

SetPasswd

The setPasswd method is located in the DialerEventServerIF interface of the EventServer module in the EventServer.idl. This method changes a user's password.

The Avaya Proactive Contact provides a configured default password. In the client application, verify that this password meets the customer's requirement.

IDL declaration

```
void setPasswd (
     in ESCommon :: LocText clientName,
     in ESCommon :: LocText curPasswd,
     in ESCommon :: LocText newPasswd)
     raises (ESError);
```

Parameters	Description
clientName	The valid login name for the client.

curPasswd	The user's current password.
newPasswd	The user's new password.

Exceptions

```
AUTHORIZATIONFAILED INVALIDPASSWD SETPASSWDFAILED INTERNALERROR NOTREADY
```

See Avaya Proactive Contact SDK exceptions for descriptions of these Event Service exceptions.

registerEventStat

The registerEventStat method is located in the EventServiceIF interface of the EventServer module in the EventServer.idl. The client application uses this method to request real-time events and real-time statistics for calls, agents, or jobs. It can also specify subset events or request system statistics. This application includes a callback object reference to the registerEventStat method to tell the server where to deliver the events.

To avoid duplicate events, include unRegisterEventStat with the correct registration ID in the client application before reissuing another event with registerEventStat.

IDL declaration

```
void
 registerEventStat (
     in EventClient:EventStatIF eventStatObj,
     in Common;;KeywordValueSeq requestList,
     out long registrationID)
     raises (ESError);
```

Parameters	Description
eventStatObj	The callback object.
requestList	The list of real-time events or statistics the client is registering to receive. You can use any combination of call event, agent event, job event, or system statistic methods in your request list. Each tag/value pair is enclosed in parentheses with the tag and value separated by a comma. The following example shows a tag/value pair in a request list: (CallEvent,) If you include statistics in your list, you must include the OnDemand tag. If you do not include this tag, you will receive statistics at the default update frequency of 6 seconds. The Event Service returns a registration identification number for you to use with the GetStatistics method.
registrationID	The registration ID that is returned to the client.

Exceptions

```
BADUPDATEFREQUENCY INTERNALERROR TOOMANYREGISTER DUPLICATEREQUEST
INVALIDOBJECT UNKNOWNEVENT
INCOMPATIBLEVERSION NOTREADY
```

See Avaya Proactive Contact SDK exceptions for descriptions of these Event Service exceptions.

Examples

The following request list instructs the Event Service to deliver all call events, all job events, and agent logon and logoff events to the client:

```
(CallEvent,)(JobEvent,)(AgentLogon,)(AgentLogoff,)
```

The following request list instructs the Event Service to deliver CallIntitiated, CallEnded, AgentLogon, AgentLogoff, JobStarted, and JobEnded events to the client as they happen. It also instructs the Event Service to deliver all system statistics to the client when the GetStatistics method is used:

```
(CallInitiated, ) (CallEnded, ) (AgentLogon, ) (AgentLogoff,) (JobStarted, )
(JobEnded, ) (SystemStats, )(OnDemand, )
```

All values in the request list are null.

unRegisterEventStat

The unregisterEventStat method is located in the EventServiceIF interface of the EventServer module in the EventServer.idl. The client application uses this method to request that the server stop sending events and statistics. The method also removes the client application from the list of registered clients.

> To avoid duplicate events, include unRegisterEventStat with the correct registration ID in the client application before reissuing another event with RegisterEventStat.

IDL declaration

```
void unRegisterEventStat (
     in long registrationID)
     raises (ESError);
```

Parameter	Description
registrationID	The registration ID that identifies the client

Exceptions

```
INTERNALERROR INVALIDREGID
NOTREGISTERED NOTREADY
```

See Avaya Proactive Contact SDK exceptions for descriptions of these Event Service exceptions.

getStatistics

The getStatistics method is located in the EventServiceIF interface of the EventServer module in the EventServer.idl. The client application uses this method to request the server return a "snapshot" of the real-time statistics. The client must be registered for one or more statistic notification types before using the getStatistics method.

IDL declaration

```
void getStatistics (
     in long registrationID)
     raises (ESError);
```

Parameter	Description
registrationID	The registration ID that identified the client.

Exceptions

See Avaya Proactive Contact SDK exceptions for descriptions of these Event Service exceptions.

addInactive

The addInactive method is located in the EventServiceIF interface of the Eventserver module in the EventServer.idl. Client application requests the server to send inactive Agent and Job statistics in the next update, one shot. The client must register before invoking this method.

Appendix A: Avaya Proactive Contact SDK exceptions

The following table defines the Error IDs for the Avaya Proactive Contact SDK interface.

Exception	Returned by	Description
AUTHORIZATIONFAILED	logon setPasswd	The client application does not have the proper access privileges.
BADUPDATEFREQUENY	registerEventStat	The update frequency number entered is outside the valid range.
DUPLICATEREQUEST	registerEventStat	The client application already sent this request.
INCOMPATIBLEVERSION	logon registerEventStat	The client application is using an incompatible version.
INTERNALERROR	logon logoff setPasswd registerEventStat unRegisterEventStat getStatistics	There was an internal error on the system. Reissue the method.
INVALIDOBJECT	logoff registerEventStat	The object reference used is invalid. Check the object reference and try again.
INVALIDPASSWORD	setPasswd	The password entered is invalid.
INVALIDREGID	unregisterEventStat getStatistics	The registration identifier is invalid.
NOTIMPLEMENTED	This exception is reserved for future use.	
NOTLOGON	logoff	The client application is not logged in to the system.

NOTREADY	logon logoff setPasswd registerEventStat unRegisterEventStat getStatistics	The system is busy or down. If the system is busy, reissue the method in a few minutes. If the system is down, restart the system and then reissue the method.
NOTREGISTERED	unRegisterEventStat getStatistics	The client application has not registered to receive events or statistics.
PASSWDEXPIRED	logon	The password for the client application expired.
SETPASSWDFAILED	setPasswd	There was an internal system error and the password was not set.
TOOMANY REGISTER	registerEventStat	The number registered exceeds the limit set by the client application.
TOOMANYCLIENTS	logon	The maximum number of client connections has been exceeded.
UNKNOWNEVENT	registerEventStat	The client application sent an invalid tag name in the request list. Check your request list for errors and try again.

Appendix B: System exceptions

The following table defines the system Error IDs defined by the CORBA system. For example, TAO_SYSTEM_EXCEPTION(UNKNOWN); //unknown exception.

Exception	Description
BAD_CONTEXT	An error processing context object occurred.
BAD_INV_ORDER	Routine invocations are out of order.
BAD_OPERATION	The operation is invalid.
BAD_PARM	An invalid parameter was passed.
BAD_TYPECODE	TypeCode is bad.
COMM_FAILURE	A communication failure occurred.
DATA_CONVERSION	A data conversion error occurred.
FREE_MEM	Cannot free memory.
IMP_LIMIT	Implementation limit was violated.
INITIALIZE	An ORB initialization failure occurred.
INTERNAL	An ORB internal error occurred.
INTF_REPOS	An error accessing the interface repository occurred.
INV_FLAG	An invalid flag was specified.
INV_IDENT	Identifier syntax is invalid.
INV_OBJREF	Object reference is invalid.
INV_POLICY	Invalid policies are present.
INVALID_TRANSACTION	Invalid TP context passed.
MARSHAL	An error occurred when marshalling the

	parameter/result.
NO_IMPLEMENT	Operation implementation is unavailable.
NO_MEMORY	A dynamic memory allocation failure occurred.
NO_PERMISSION	Permission does not exist for the attempted operation.
NO_RESOURCES	There are insufficient resources for the request.
NO_RESPONSE	The response to the request is not yet available.
OBJ_ADAPTER	The object adapter had a failure.
OBJECT_NOT_EXIST	Object does not exist.
PERSIST_STORE	A persistent storage failure occurred.
REBIND	A rebind is needed.
TIMEOUT	Operation timed out.
TRANSACTION_MODE	An invalid transaction mode occurred.
TRANSACTION_REQUIRED	Operation needs transaction.
TRANSACTION_ROLLEDBACK	Operation was a no-op.
TRANSACTION_UNAVAILABLE	No transaction occurred.
TRANSIENT	A transient failure occurred; reissue the request at a later time.
UNKNOWN	Exception is not known.

Appendix C: Completion codes

The following table shows the completion codes for an Avaya Proactive Contact.

The values 20-34 and 51-85 are available for agent completion codes.

Code	Keyword	Type	Description	Report Header
000	NOTCALLED	System	The account has not been called.	
001	CODE1	System	Reserved for the system.	
002	ERROR	System	The system detected an invalid phone number.	
003	TIMEOUT	System	The system did not receive a dial tone.	Timed out
004	HANG_PORT	System	The line was idle after the system dialed the customer record phone number.	
005	NOTINZONE	System	The local time for the customer phone is outside calling hours.	Not within legal hours
006	MOFLASH_B	Agent	Used for native voice and data transfer. An agent transfers a call to an inbound agent without remaining on the line.	Blind transfer
007	HANG_TRANS	System	No agent is available for a supervisor transfer.	
008	TDSS_HF_B	Agent	ADAPTS API: the agent transfers a call without remaining on the call. This is known as blind hook flash transfer.	
009		System	Reserved for the system.	
010		System	Reserved for the system.	
011	BUSY	System	The system detected a busy signal.	

012	CONTTONE	System	The system detected a continuous tone, such as a fax or modem.	
013	AUTOVOICE	System	The system detected an answering machine.	
014	VOICE	System	Interim code when a person is on the line.	
015	NOANSWER	System	The call placed was not answered.	
016	RINGING	Agent	Can be user defined but is usually defined as a phone call that was still ringing but was passed to an agent.	
017	CUSTHU	Agent	Can be user defined but is usually used to define when a customer hangs up while the call is in the wait queue, and the call is still passed to an agent.	
018	TRANSFER	Agent	Can be user defined but is usually defined as a transfer release.	Transferred
019	RECALL	Agent	Can be user defined but is usually defined as a recall release.	
020-034		Agent	Customer assigned codes used by agents.	
035	CANCEL	System	Can be user defined but is usually defined as the agent cancelled the managed call.	
036	INTERCEPT	System	Special Information Tone (SIT) received that indicates an operator intercepted the call.	
037	NOCIRCUIT	System	SIT received that indicates the circuits were unavailable.	
038	DISCONN	System	SIT received that indicates the call was a disconnected number.	
039	VACANT	System	SIT received that indicates the call cannot be completed as dialed.	

040	REORDER	System	The call resulted in a fast busy tone.	
041	R_RINGING	System	Reserved.	
042	LINEFAIL	System	A failure on the phone line occurred.	
043	OP_RECALL	System	Operator set recall.	
044	DTMF_V	System	DTMF tone detected.	Voice DTMF
045	HU_INB	System	The customer hung up while in the inbound wait queue.	
046	HU_OUT	System	The customer hung up while in the outbound wait queue.	
047	HANG_INB	System	An agent was unavailable for the inbound call.	
048	HANG_OUT	System	An agent was unavailable for the outbound call.	
049	OPDIED	System	The agent session ended abnormally.	
050	R_HSONHOOK	System	The agent headset disconnected from Avaya Proactive Contact.	
051-088		Agent	Customer assigned codes used by agents.	
089	MANAGEDA	Agent	Managed Dial: Managed non-connection A.	
090	MANAGEDB	Agent	Managed Dial: Managed non-connection B.	
091	VIRTVOICE	System	Virtual Agent: Virtual message to VOICE, a person.	
092	VIRTAUTOV	System	Virtual Agent: Virtual message to AUTOVOICE, a calling machine.	

093	SOLD	Agent	Sales Verification: Sold campaign.	
094	VERIFIED	Agent	Sales Verification: Sale verified.	Verified sale
095	UNVERIFIED	Agent	Sales Verification: Sale not verified.	
096-097		System	Reserved for the system.	
098	AORECALL	Agent	Agent Owned Recall.	
099		System	Reserved for the system.	
100-200		Agent	Customer assigned	

Appendix D: Windows sample setup

This section explains how to set up the TAO. It also explains how to build the client executable on Windows for the SDK.

The toolkit contains IDLs of EventService, ORB ACE+TAO-1.6a_Windows zip, solution files, and sample code of clients for Windows.

This section contains the following topics:

NOTE: For Windows 64-bit OS, the TAO and Client application were tested using Win32 settings in Visual Studio.

Required Software:

```
Perl for windows (For Example:
http://www.activestate.com/Products/activeperl/features.plex)

WinZIP or similar tool for extracting software archives
```

ORB TAO setup on Windows

The ORB of CORBA that is tested in the toolkit for windows is ACE+TAO-1.6a_Windows.zip. The sample code can be easily adjusted in higher versions of TAO and other ORBs as long as they are CORBA 2.3 or higher compliant.

1. Copy the SDK package to the C drive. For example:

   C:\v_pdssdk\EventService\...

   C:\v_pdssdk\tao

   C:\v_pdssdk\idl

2. Ensure the file, ACE+TAO-1.6a_Windows.zip, is in the `C:\v_pdssdk\tao` folder.
3. Use WinZip again to extract files from ACE+TAO-1.6a_Windows.zip to the root C: directory (recommended location).

   After extracting, C:\ACE_wrappers should be created.
4. Copy folder openssl to C: directory (recommended location).
5. Create a file named **config.h** in **C:\ACE_wrappers\ace**. Type the following line and save the file:

```
#define ACE_DISABLE_WIN32_ERROR_WINDOWS
#define ACE_DISABLE_WIN32_INCREASE_PRIORITY
#include "config-win32.h"
```

6. Follow these steps to configure your environment variables:
    a. Right-click **My Computer**, and then select **Properties**.
    b. On the **Advanced** tab, select **Environment Variables**.
       (For Windows 7 the **Environment Variables** can be accessed as
       My computer -> Properties -> System Protection ->Advanced ->
       Environment Variables)
    c. In the **User Variables** area, click ***New***.
    d. In the **New User Variable** dialog box, type **ACE_ROOT** for **Variable Name** and type **C:\ACE_wrappers** for **Variable Value**.
    e. Click **OK**.
    f. In the **User Variables** area, click ***New***.
    g. In the **New User Variable** dialog box, type **TAO_ROOT** for **Variable Name** and type **%ACE_ROOT%\TAO** for **Variable Value**.
    h. Click **OK**.
    i. In the **User Variables** area, click **New**.
    j. In the **New User Variable** dialog box, type **SSL_ROOT** for **Variable Name** and type **C:\openssl** for **Variable Value**.
    k. In the **User Variables** area of the **Environment Variables**, click **Path**.
    l. Click **Edit.**
    m. In the **Edit User Variable** dialog box, type **%ACE_ROOT%\bin;%ACE_ROOT%\lib; %SSL_ROOT%\out32dll** at the end of the line with a semicolon in front of it in the **Variable Value** field.
    n. Click **OK**.
    o. Close Environment Variables and Properties.
7. Copy folder **Certificates** to **C:** directory (recommended location).
8. In Certificates folder, open the file corba_svc.conf and edit the path of certificate files to the current path.

```
     dynamic SSLIOP_Factory Service_Object *
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() "-
SSLAuthenticate SERVER_AND_CLIENT -SSLPrivateKey
PEM:C:\Certificates\corbaServer_key.pem -SSLCertificate
PEM:C:\Certificates\corbaServer_cert.pem -SSLCAfile
PEM:C:\Certificates\ProactiveContactCA.pem"
```

```
static Resource_Factory  "-ORBProtocolFactory
SSLIOP_Factory"
```

9. Add **ssl=1** to your MPC **%ACE_ROOT%/bin/MakeProjectCreator/config/default.features** file.

   If the default features file is not present, then create it.

10. Run MPC to add support for building the ACE_SSL library for ACETAO project using DOS command prompt for the command execution.

    ```
    cd %TAO_ROOT%

    %ACE_ROOT%\bin\mwc.pl -type vc[version] TAO_ACE.mwc
    ```

    Where [version] is "71", "8", or "9" for Visual C++ 7.1,

    8, or 9, respectively.

    More information on MPC is available here:

    http://www.ociweb.com/products/mpc

11. Build TAO.

    a. Create a new "**openssl**" folder in **C:\openssl\include**, the hierarchy should now look like: **C:\openssl\include\openssl**

    b. **Copy all header files (.h) files from C:\openssl\include** directory to this new directory **C:\openssl\include\openssl**.

    c. Go to **C:\ACE_wrappers\TAO** and open the below solution file as per the VC++ compiler version

       %TAO_ROOT%\TAO_ACE.sln

    d. Set the OpenSSL library and include/header directory path

       - Go to **Tools->Options**
       - If **Show all Setting** checkbox is present at the bottom, select it.
       - Select **Projects and Solutions** in Options pane
       - Select **VC++ Directories**
       - Set **Platform** to Win32
       - Select **Library file** in Show Directories for:
       - Add **C:\openssl\out32dll**
       - Select **Include files** in **Show Directories for**:
       - Add **C:\openssl\include**

    e. Click the **Build** menu and select **Batch Build**

    f. Sort by **Platform** column and select all Win32 projects in **Build** column

g. Ensure that all Win32 build entries are selected. In the **Batch Build** dialog box, click **Build**. The build may take up to about 3 hours

The projects in the TAO_ACE solution build the ACE and TAO libraries, TAO IDL compiler, gperf, ORB services libraries and executables, and some common utilities.

Libraries will be installed in %ACE_ROOT%\lib. Some executables will be installed in %ACE_ROOT%\bin, others (the ORB services executables) will be installed in their source directories.

12. Perform the following steps to verify that ORB TAO is set up correctly.

    a. In **C:\ACE_wrappers\TAO\orbsvcs\Naming_service**, start the `Naming_Service` executable in a DOS window by typing:

    ```
    Naming_Service -ORBEndpoint iiop://myhostname:88888
    ```

    where 88888 can be any unused port number at the moment. The port numbers will be less than 65000.

    Note that here, myshostnsme is hostname of windows m/c.

    b. In a different DOS window, cd to **C:\ACE_wrappers\bin**.

    c. Then, start `nslist` by typing:

    ```
    tao_nslist -ORBInitRef
      NameService=iioploc://myhostname:88888/NameService
    ```

    You should see the following result:

    ```
    Naming Service:
    ```

Build client executable on Windows for Proactive Contact 5.0 SDK

The below steps for creating a sample application is based on the Visual Studio 2008 Win32 project. For other Visual studios 2003 or higher the steps should be more or less similar.

1. Before beginning, ensure the following items are complete.

   - EventService SDK has been installed at C:\v_pdssdk.
   - ORB TAO has been installed at C:\ACE_wrappers.
   - Visual Studio 2008 has been installed.
   - Environment Variables such as ACE_ROOT, TAO_ROOT and SSL_ROOT, Path have been set as described earlier in the ORB TAO setup procedure.

2. Perform the following steps to set up your Visual C++ Options.

   a. Start a **Visual C++** session.
   b. Select the menu option **Tools > Options**.

c. If **Show all Setting** checkbox is present at the bottom, select it
d. In the **Options** dialog box, select the **Projects and Solutions**
e. Expand the **Projects and Solutions** by pressing "+" button
f. Select **VC++ Directories**
g. On the **Directories** page, select the **Include Files** from the **Show directories for** combo box.
h. In the Directories add the following directories:

```
C:\ACE_wrappers
C:\ACE_wrappers\TAO
C:\ACE_wrappers\TAO\tao
C:\ACE_wrappers\TAO\orbsvcs
C:\ACE_wrappers\TAO\orbsvcs\orbsvcs
C:\openssl\include
C:\v_pdssdk\EventService\v4_0\C++\sdk
C:\v_pdssdk\EventService\v4_0\C++\sdk\Interface
```

i. In the **Show directories for:** list, select **Library Files**.
j. In the **Directories** box, type:

```
C:\ACE_wrappers\ace
C:\ACE_wrappers\TAO\tao
C:\ACE_wrappers\TAO\orbsvcs\orbsvcs
C:\openssl\out32dll
C:\ACE_wrappers\lib
```

k. In the **Show directories for:** list, select **Executable Files**
l. In the **Directories** box, add

```
C:\ACE_wrappers\bin.
C:\ACE_wrappers\lib.
```

m. Click **OK**.

3. Perform the following steps to create a solution.

a. In the Visual Studio session, select File > New > Project.
b. In the left pane select Other Project Types, then **Visual Studio Solutions** from the expanded list.
c. Select **Blank Solution** from the Visual Studio installed templates in Templates pane
d. Set the name **sdk** in Name box and location **C:\v_pdssdk\EventService\v4_0\C++** in **Location** box for your solution, and then click **OK**.
e. At the top of the solution window, the following will appear:
Solution 'sdk' (0 projects)

4. Perform the following steps to create a Project Interface. The Interface project contains the code generated by the IDL compiler.

a. In **Solution Explorer**, right-click the solution **Solution 'sdk' (0 projects)**, click **Add**, and then click **New Project**.
b. In the **New Project** dialog box, select **Visual C++.**
c. Select **Win32** from **Visual C++** expanded list**.**
d. Select **Win32 Project** from the **Visual Studio installed templates** in Templates pane
e. Enter **Interface** in **Name** box.
f. Enter the project location **C:\v_pdssdk\EventService\v4_0\C++\sdk** in **Location** box.
g. Press **OK**
h. A **Win32 Application Wizard – Interface** dialog box will open.
i. From the **Overview** page of the **Win32 Application Wizard – Interface** dialog, press **Next**.
j. From the **Application Settings** page of the **Win32 Application Wizard – Interface**, select **Static library**
k. From the **Application Settings** page of the **Win32 Application Wizard – Interface**, deselect **Precompiled header** under **Additional options**,
l. Press **Finish** to create the project.

Interface project will be created under the **sdk** solution.

5. Perform the following steps to add files to the Interface project.

a. In the solution explorer, right-click on project **Interface**.
b. Select **Add > Add Existing Item**.
c. On **Add Existing Item** dialog box, go to the path **C:\v_pdssdk\idl\ v4_idl.**
d. Select the files **Common.idl, EventClient.idl**, **EventServer.idl**, and **EventTypes.idl.**
e. Press **Add**

6. Perform these steps to specify how IDL compiling takes place.

a. In the solution explorer, double-click **Interface**.
b. Double-click the **Source Files** folder.
c. Select all idl files.
d. Right-click all selected idl files and then select **Properties**.
e. In the **Property Pages** dialog box, select **General** from expanded list of **Configuration Properties** in left pane.
f. In right pane, click on **Excluded From Build** and select **No** from the drop down box.
g. In right pane, click on **Tool** and select **Custom Build Tool** from the drop down box.
h. Now press **Apply** button.
i. **Custom Build Step** will be displayed under the **Configuration Properties**
j. Click on **Custom Build Step**
k. In right pane select **Command Line** box and type the following

```
$(ACE_ROOT)\bin\tao_idl.exe -I$(InputDir) -GI $(InputPath)
```

l. select **Description** box and type the following

```
Invoking TAO IDL compiler on $(InputPath)
```

m. In Outputs box, click on the drop down and Select <Edit>

n. A new window will be opened, type the following in the Outputs window and Press ok

```
$(InputName)C.cpp
$(InputName)C.h
$(InputName)C.i
$(InputName)S.cpp
$(InputName)S.h
$(InputName)S.i
$(InputName)S_T.cpp
$(InputName)S_T.h
$(InputName)S_T.i
```

7. Perform the following steps to run the IDL compiler on four IDL files.

   a. In the **Solution Explorer > Interface > Source Files**, right-click **Common.idl** and select Compile.
   b. Repeat the above step for the all other idls.

8. These steps describe creating the Interface library and adding source files generated by IDL compiling into the project Interface.

   a. In the Solution Explorer, right-click **Interface**.
   b. Select **Add > Add Existing Item**.
   c. On **Add Existing Item** dialog box, go to the path **C:\v_pdssdk\ EventService\v4_0\C++\sdk\Interface.**
   d. Select the following files

```
CommonC.cpp
CommonS.cpp
EventClientC.cpp
EventClientS.cpp
EventServerC.cpp
EventServerS.cpp
EventTypesC.cpp
EventTypesS.cpp
```

   e. Press **Add**
   f. Right-click on the Interface and select **Properties**
   g. **Interface Property Pages** window will be opened.

h. Select **C/C++** from the **Configuration Properties** expanded list.
i. Select **Code Generation** from the **C/C++** expanded list.
j. In right pane,
   For debug: select **Multi-threaded Debug DLL (/MDd)** in **Runtime Library** drop down.

   For release: select **Multi-threaded DLL (/MD)** in **Runtime Library** drop down.
k. Right-click on the **Interface** and select **Build.**

Wait for the build to complete.

9. Perform the following steps to create a project called **enclient**. The enclient project contains the client code.

   a. In **Solution Explorer**, right-click the solution **Solution 'sdk' (1 projects)**, click **Add**, and then click **New Project**.
   b. In the **New Project** dialog box, select **Visual C++.**
   c. Select **Win32** from **Visual C++** expanded list**.**
   d. Select **Win32 Console Application** from the **Visual Studio installed templates** in Templates pane
   e. Enter **enclient** Interface in **Name** box.
   f. Enter the project location **C:\v_pdssdk\EventService\v4_0\C++\sdk** in **Location** box.
   g. Press **OK**
   h. A **Win32 Application Wizard – enclient** dialog box will open.
   i. From the **Overview** page of the **Win32 Application Wizard – enclient** dialog, press **Next**.
   j. From the **Application Settings** page of the **Win32 Application Wizard – enclient**, deselect **Precompiled header** and select **Empty project** under **Additional options**,
   k. Press **Finish** to create the project.

   enclient project will be created under the **sdk** solution.

10. Perform these steps to add source files into the project **enclient**.

    a. In the Solution Explorer, right-click **enclient**.
    b. Select **Add > Add Existing Item**.
    c. On **Add Existing Item** dialog box, go to the path **C:\v_pdssdk\ EventService\v4_0\C++.**
    d. Select the following files

```
EventClient.cpp
EventClient.h
EventClient_i.cpp
EventClient_i.h
StringConvert.cpp
StringConvert.h
```

e. Press **Add**
f. Right-click on the enclient and select **Properties**
g. **enclient Property Pages** window will be opened.
h. Select **C/C++** from the **Configuration Properties** expanded list.
i. Select **Code Generation** from the **C/C++** expanded list.
j. In right pane,
   For debug:

   select **Multi-threaded Debug DLL (/MDd)** in **Runtime Library** drop down.

   For release:

   select **Multi-threaded DLL (/MD)** in **Runtime Library** drop down.
k. Select **Linker** from the **Configuration Properties** expanded list.
l. Select **Input** from the **Linker** expanded list.
m. In right pane,
   Add the following libraries in the **Additional Dependencies** box:

   For debug:

```
TAO_PortableServerd.lib
TAOd.lib
ACEd.lib
TAO_CosNamingd.lib
TAO_TypeCodeFactoryd.lib
TAO_Valuetyped.lib
TAO_AnyTypeCoded.lib
TAO_BiDirGIOPd.lib
TAO_CodecFactoryd.lib
```

   For release:

```
TAO_PortableServer.lib
TAO.lib
ACE.lib
TAO_CosNaming.lib
TAO_TypeCodeFactory.lib
TAO_Valuetype.lib
TAO_AnyTypeCode.lib
TAO_BiDirGIOP.lib
TAO_CodecFactory.lib
```

n. Right-click on the **enclient** and select **Project Dependencies….**
o. On the **Project Dependencies** dialog box, select **enclient** in the Select project to modify list.
p. In the **Depends on** box, select **Interface**.
q. Click **OK**.
r. Right-click on the **enclient** and select **Build.**

11. Verify that the contents of **C:\Certificates\corba_svc.conf** are as follows:

```
dynamic SSLIOP_Factory Service_Object *
TAO_SSLIOP:_make_TAO_SSLIOP_Protocol_Factory() "-
SSLAuthenticate SERVER_AND_CLIENT -SSLPrivateKey
PEM:C:\Certificates\corbaServer_key.pem -SSLCertificate
PEM:C:\Certificates\corbaServer_cert.pem -SSLCAfile
PEM:C:\Certificates\ProactiveContactCA.pem"static
Resource_Factory "-ORBProtocolFactory SSLIOP_Factory"
```

NOTE: In case the .pem files are located in some other location, please change the path accordingly in the corba_svc.

12. Perform these steps to execute client enclient.

a. First, make sure that the Proactive Contact 5.0 is up and running on the server machine on Red Hat Linux 5.5.
b. On the client machine, in a DOS window, cd to
C:\v_pdssdk\EventServer\v4_0\C++\sdk\Release
or
C:\v_pdssdk\EventService\v4_0\C++\sdk\Debug
c. Type:

```
enclient -ORBSvcConf C:\Certificates\corba_svc.conf

–ORBInitRef NameService=corbaloc:ssliop:<dialer
Name>:23201/NameService -h <dialer Name> -F
```

Where 23201 is the port number used by Naming_Service on the Linux machine which is running Avaya Proactive Contact 5.0. It cannot be changed.

NOTE: In case enclient is executed without -h option, there may be an error message displayed as follows: Error: Bind to object failed

Appendix E: RHEL 5.5 sample setup

This appendix explains how to set up the ORB TAO and Linux GNU C++ coding environments on RHEL 5.5. It also explains how to build the client executable on RHEL 5.5 for Event Services5.0.

This section contains the following topics:

ORB TAO setup on RHEL 5.5

Overview

The toolkit contains IDLs of EventService, ACE+TAO-1.6a_Linux.tar.gz, Makefile, and sample code of clients for RHEL 5.5.

The ORB of CORBA that's tested in the toolkit for linux is ACE+TAO-1.6a_Linux.tar.gz

The sample code can be easily adjusted in higher versions of TAO and other ORBs as long as they are CORBA 2.3 or higher compliant.

The tools that are used with RHEL 5.5 include the following:

- Compiler: /usr/bin/gcc

  gcc version 4.1.2 20080704 (Red Hat 4.1.2-48)

- Openssl: openssl-0.9.8e-12.el5_4.6
- GNU make executable (3.80 or higher)

  /usr/bin/make

  GNU Make 3.81

  Copyright (C) 2002 Free Software Foundation, Inc.

  This is free software; see the source for copying conditions.

  There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Follow the instructions below to set up ORB TAO on RHEL 5.5:

1. Install the SDK CD to **`/tmp/v_pdssdk`**.
2. Type the following to change directories and move the **ACE+TAO-1.6a_Linux.tar.gz** file as well as some conf files:

   ```
   cd /tmp/v_pdssdk/tao
   mv ACE+TAO-1.6a_Linux.tar.gz /opt
   cd /tmp/v_pdssdk/Certificates
   mv corbaServer_cert.pem corbaServer_key.pem corba_svc.conf
     ProactiveContactCA.pem /opt
   ```

The directory /opt is preferred because the run time ORB TAO on the Proactive Contact 5.0 system is installed to `/opt`. Throughout this appendix, `/opt` is assumed to be the directory that contains ORB TAO.

3. Unpackage the compressed file by typing the following:

   cd /opt

   gunzip ACE+TAO-1.6a_Linux.tar.gz

   It will create **ACE+TAO-1.6a_Linux.tar**

   Unpackage the tar file by typing following command

   tar –xvf ACE+TAO-1.6a_Linux.tar

   After unpackaging, the directory `ACE_wrappers` is created in `/opt`.

4. Set up the environment variables by typing the following:

   export ACE_ROOT=/opt/ACE_wrappers

   export TAO_ROOT=/opt/ACE_wrappers/TAO

   export SHLIB_PATH=/opt/ACE_wrappers/ace:$SHLIB_PATH

   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ACE_ROOT/lib

These environments can be set in `.profile`.

5. Next, customerize certain include files for the RHEL 5.5 platform in the directory **/opt/ACE_wrappers/ace.** `Create file config.h with following entry in it.`

   # include "ace/config-linux.h "

6. Create a new files named "**platform_macros.GNU**" in **/opt/ACE_wrappers/include/makeinclude** `with follwing entries`

   debug=0 # (or debug=1)

   optimize=0 # (or optimize=1)

   ssl=1

   include $(ACE_ROOT)/include/makeinclude/platform_linux.GNU

7. In $TAO_ROOT, regenerate your build files using the $ACE_ROOT/bin/mwc.pl script.

   $ACE_ROOT/bin/mwc.pl -type gnuace TAO_ACE.mwc

8. This step (building the ORB TAO) takes up to 2 hours. Before beginning, ensure /usr/bin/make is the one that you will use (If there are multiple make executables).

   cd $TAO_ROOT/

   make all

9. Perform these steps to verify that ORB TAO was built correctly.

a. Start `Naming_Service` and use `nslist` to connect to it.

b. Ensure that gateway address is set (using command `netconfig`) correctly because `Naming_Service` is multicast by default.

c. Type:

```
cd $TAO_ROOT/orbsvcs/Naming_Service
```

d. Type

```
./Naming_Service -ORBEndpoint iiop://myhostname:88888
```

where 88888 can be any unused port number at the moment. The port numbers will be less than 65000.

Note that here, myshostnsme is hostname of Linux m/c.

```
cd $ACE_ROOT/bin
```

e. Type

```
./tao_nslist -ORBInitRef
NameService=iioploc://myhostname:88888/NameService
```

The following result should be seen:

```
Naming Service:
```

ORB TAO is now available. Do not forget to always set your environment variables `ACE_ROOT`, `TAO_ROOT`, `LD_LIBRARY_PATH`, and `SHLIB_PATH` whenever applications of TAO are compiled and started up. It is better to set them in `.profile`.

Build client executable on RHEL 5.5 for APC 5.0 Event Services

The client executable build process for Avaya Proactive Contact is explained in the following section.

1. Before building a client executable, verify that the following are installed:
   - EventService SDK is installed at /tmp/v_pdssdk
   - ORB TOA is installed at /tao/ACE_wrappers
   - ORB TAO has been built successfully

   For simplicity's sake, the build machine and the target machine explained here are the same. However, they can be different as long as ORB TAO is installed at the same location /opt/ACE_wrappers.

   NOTE: If the SDK is ftped from another machine, it is recommended that you use RHEL's plain ftp utility and not a different ftp application tool.

2. Set up your environment.

   ```
   export ACE_ROOT=/opt/ACE_wrappers
   ```

   ```
   export TAO_ROOT=/opt/ACE_wrappers/TAO
   ```

```
export SHLIB_PATH=/opt/ACE_wrappers/ace:$SHLIB_PATH
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ACE_ROOT/lib
```

It is recommended that you set up these environments in `.profile`.

3. Build the executable.

```
cd /tmp/v_pdssdk/EventService/v4_0/C++
make idl
make enclient
```

4. Ensure that the event service executable, enserver, is running on the dialer machine.

   TAO's client cannot automatically start up enserver if it is not running,

   Go to `/tmp/v_pdssdk/EventService/v4_0/C++`

   Type the following

   ./enclient -ORBSvcConf /opt/corba_svc.conf -ORBInitRef NameService=corbaloc:ssliop:<dialer name>:23201/NameService -h <dialer name> -F

   Where 23201 is the port number used by Naming_Service on the Linux machine which is running Avaya Proactive Contact 5.0. It cannot be changed.

Appendix F: Java sample application

This section contains the following topics:

Required software

The Avaya Proactive Contact SDK requires the following software:

- Java JDK: version 1.6.0_21
- .Ant: Version 1.8.1 or above
- Jacorb 2.3.1

Building the JacORB on Linux

Follow the instructions below to build the JacORB version of the Avaya Proactive Contact SDK Sample on the Linux environment:

1. Install the SDK CD to /tmp/v_pdssdk.
2. Type the following to change directories and move jacorb-2.3.1-src.zip file

```
cd /tmp/v_pdssdk/jacorb
mv jacorb-2.3.1-src.zip /opt
cd /tmp/v_pdssdk
cp -r keystore /opt
```

3. Unpackage the compressed file by typing the following:

```
cd /opt
unzip jacorb-2.3.1-src.zip
```

   After unpackaging, the directory jacorb-2.3.1 is created in /opt

4. Set up the environment variables by typing the following

```
export JAVA_HOME="Path where you have installed java"
export JacORB_HOME=/opt/jacorb-2.3.1
```

5. Copy jacorb.properties from /tmp/v_pdssdk/jacorb to $JacORB_HOME/etc

```
cd /tmp/v_pdssdk/jacorb/
cp jacorb.properties  $JacORB_HOME/etc/
```

6. orb.properties from /tmp/v_pdssdk/jacorb to $JAVA_HOME/lib dir

```
cd /tmp/v_pdssdk/jacorb/
cp orb.properties  $JAVA_HOME/lib
```

7. This step (building JacORB) takes up to 10 minutes. Go to directory JacORB-2.3.1 and type the following:

   cd jacorb-2.3.1

   ant

   You will get a BUILD SUCCESSFUL message.

Executing on RHEL 5.5 for APC 5.0 Event Services

Before beginning the execution steps, ensure you have performed the following steps:

- EventService SDK has been installed at /tmp/v_pdssdk
- /opt/jacorb-2.3.1 has been installed at /opt/jacorb-2.3.1
- JacORB-2.3.1 has been built successfully

1. Set up your environment using the following:

   - export JAVA_HOME="Path where you have installed java"
   - export JacORB_HOME=/opt/jacorb-2.3.1
   - export CLASSPATH=$CLASSPATH:$JacORB_HOME/lib/avalon-framework-4.1.5.jar:$JacORB_HOME/lib/slf4j-api-1.5.6.jar:$JacORB_HOME/lib/slf4j-jdk14-1.5.6.jar:$JacORB_HOME/lib/logkit-1.2.jar: $JacORB_HOME/lib/idl.jar:$JacORB_HOME/lib/jacorb.jar

Note: It is recommended that you set up these environments in .profile

2. Check that idl compiler of JacORB is in PATH .If it is not in the specific path then

   Set up in the environment variable using Export PATH=$PATH:$JacORB_HOME/bin

3. Build the idl's using the following command:

   cd /tmp/v_pdssdk/EventService/v4_0/Java

   Compile all the following idls:

   idl -I../../../idl/v4_idl/ -all ../../../idl/v4_idl/EventClient.idl

   idl -I../../../idl/v4_idl/ -all ../../../idl/v4_idl/EventServer.idl

   idl -I../../../idl/v4_idl/ -all ../../../idl/v4_idl/EventTypes.idl

   idl -I../../../idl/v4_idl/ -all ../../../idl/v4_idl/Common.idl

   The above commands will generate the EventClient, ESType Common and EventServer subdirectories in the source directory.

4. Compile the Common package by executing the following command in the specific directory:

/tmp/v_pdssdk/EventService/v4_0/Java:

javac -classpath . Common/*.java

5. Compile the ESType package by using the following

   /tmp/v_pdssdk/EventService/v4_0/Java :

   javac -classpath . ESType/*.java

6. Compile the Event Client package by using the following command in the following directory /tmp/v_pdssdk/EventService/v4_0/Java :

   javac -classpath . EventClient/*.java

7. Compile the EventServer package by using the following command in the specific directory /tmp/v_pdssdk/EventService/v4_0/Java :

   javac -classpath . EventServer/*.java

8. Compile the sample application code by using the following command in the specific directory

   /tmp/v_pdssdk/EventService/v4_0/Java :

   javac -classpath . *.java

   javac -classpath . TIE/*.java

Executing the Application

Follow these steps to execute the application:

1. Check environment variable $JacORB_HOME is set.
2. Check file jacorb_properties is present in $JacORB_HOME/etc.
3. Check that file orb.properties is present in $JAVA_HOME/lib
4. Go to source directory and create two files using following commands:

   /tmp/v_pdssdk/EventService/v4_0/Java/TIE

   touch ActionMenu.txt

   touch RegisterMenu.txt

5. Start the TestClient using following command:-

   java -Djacorb.home=${JacORB_HOME} -Djacorb.config.dir=${JacORB_HOME}/etc -DORBInitRef.NameService=corbaloc:ssliop:1.2@hostname:port_no_naming_service/NameService -classpath $CLASSPATH:.. TestClient hostname username password.

If you get the error ERROR : org.omg.CORBA.NO_PERMISSION: Client-side policy requires SSL/TLS, but server doesn't support it vmcid: 0x0 minor code: 0 completed: No

org.omg.CORBA.NO_PERMISSION: Client-side policy requires SSL/TLS, but server doesn't support it vmcid: 0x0 minor code: 0 completed: No

Ensure that you are using jacorb's ORB not java's native orb

Building JacORB on Windows

Follow these steps to build the JacORB on Windows platform:

1. Install JDK version 1.6.0_21
2. Download Ant (version 1.8,1 or above) and install it as described in the Ant documentation.
3. Copy the Avaya Proactive Contact SDK CD into a directory.
4. In the directory, unzip jacorb-2.3.1-src.zip to JacORB-2.3.1.
5. Set up following environment variables:
   - JAVA_HOME= <Path of Java>
   - JacORB_HOME=<Path of JacORB>
6. Copy jacorb.properties from v_pdssdk\jacorb to JacORB-2.3.1\etc
7. Edit jacorb.properties file and change the path of keystore to current location: jacorb.security.keystore= <path of keystore folder>/jacorb

   [NOTE: Replace all occurrences of back slash (\) with forward slash (/) in the keystore path. The jacorb parser does not parse (\)]
8. Copy orb.properties from v_pdssdk\jacorb to <JAVA_HOME>\jre\lib
9. Edit orb.properties file and set the path of JacORB_HOME

   jacorb.config.dir=<path of JacORB_HOME>

   [NOTE: Replace all occurrences of forward slash (\) with back slash (/) in the config path. The jacorb parser does not parse (\)]
10. To build JacORB, execute the following command in the JacORB-2.3.1 directory:

    $> ant

    You will get the BUILD SUCCESSFUL message.

You can refer to following file for JacORB build instructions: JacORB-2.3.1/doc/install.html

Executing on Windows for APC 5.0 Event Services

Before beginning, ensure you have installed the following:

- Java 1.6.0_21 is installed.
- JacORB-2.3.1 has been installed at <v_pdssdk/ JacORB-2.3.1>
- JacORB-2.3.1 has been built successfully

- EventService SDK has been installed at <v_pdssdk/EventService>

1. Set up following environment variables:
   - JAVA_HOME=<Path of Java>
   - JacORB_HOME=<Path of JacORB>
   - CLASSPATH=.;%JacORB_HOME%\lib\avalon-framework-4.1.5.jar;%JacORB_HOME%\lib\slf4j-api-1.5.6.jar;%JacORB_HOME%\lib\slf4j-jdk14-1.5.6.jar;%JacORB_HOME%\lib\logkit-1.2.jar; %JacORB_HOME%\lib\idl.jar;%JacORB_HOME%\lib\jacorb.jar

   [**NOTE**: The format of the CLASSPATH string should be exactly as mentioned above. Take care not to miss hyphens (-), semicolons (;) or periods (.)]
   - PATH=.;%JacORB_HOME%\bin
2. Compile all the idls by executing the following commands:

```
cd   <path of v_pdssdk>\EventService\v4_0\Java
idl  <path of v_pdssdk>\idl\v4_idl\EventClient.idl
idl  <path of v_pdssdk>\idl\v4_idl\EventServer.idl
idl  <path of v_pdssdk>\idl\v4_idl\EventTypes.idl
idl  <path of v_pdssdk>\idl\v4_idl\Common.idl
```

   This will generate the EventClient, ESType Common and EventServer subdirectories in the source directory (<path of v_pdssdk>\EventService\v4_0\Java).
3. Execute the following commands:.

```
javac -classpath .  Common\*.java
javac -classpath .  ESType\*.java
javac -classpath .  EventClient\*.java
javac -classpath .  EventServer\*.java
javac -classpath .  *.java
javac -classpath .  TIE\*.java
```

   This will compile the client.
4. To execute the application

```
java -Djacorb.home=%JacORB_HOME% -
Djacorb.config.dir=%JacORB_HOME% -
DORBInitRef.NameService=corbaloc:ssliop:1.2@<dialername>:<Nam
  ingServiceport>/NameService –classpath
```

```
%CLASSPATH%;.\ESType;.\Common;.\EventClient;.\EventServer;.
\TIE TestClient <dialername> <username> <password>
```

If you get the error ERROR : org.omg.CORBA.NO_PERMISSION: Client-side policy requires SSL/TLS, but server doesn't support it vmcid: 0x0 minor code: 0 completed: No

org.omg.CORBA.NO_PERMISSION: Client-side policy requires SSL/TLS, but server doesn't support it vmcid: 0x0 minor code: 0 completed: No

Ensure that you are using jacorb's ORB not java's native orb.

Appendix G: Certificate Generation, Signing and Maintenance

Currently OpenSSL CA is the CA which is evaluated . OpenSSL CA is Open Source CA distributed from http://openssl.org

OpenSSL CA is a minimal CA application. It can be used to sign certificate requests in a variety of forms and generate CRLs it also maintains a text database of issued certificates and their status.

On the shipped PC 5.0 systems there will be default certificates for all the services and internal clients. For external clients, they can get their default certificates from the SDK.

Also in the configuration file, it will provide information on whether both servers and clients have certificates, or only servers have certificates.

If you want to generate your own certificates you need to deploy the same set of certificates for server client and CA certificates on dialer as well. For deployment verify /opt/avaya/pds/openssl/ directories.

Generating certificates and Keystore using OpenSSL CA

// BBO=Black Box Operation which would not concern you. (CA creation and key certification.)

1. // BBO: create CA

   % CA.pl -newca

2. // Create openssl server keys and cert request.

   % openssl req -new -days 365 -nodes -newkey rsa:1024 -out serverreq.pem -keyout serverkey.pem -subj "/C=US/ST=MO/L=St Louis/O=OCI/CN=Hyperlink_server"

3. // BBO: CA signs server certificate

   % openssl ca -in serverreq.pem -out servercert.pem

4. // Create JacORB client key

   % keytool -genkey -keyalg RSA -alias clientalias -validity 365 -keystore client_ks -storepass clientpass -dname "C=US,ST=MO,L=St. Louis,O=OCI,CN=Hyperlink_client"

5. // Extract JacORB cert request

   % keytool -certreq -alias clientalias -keystore client_ks -storepass clientpass -file clientreq.pem

6. // BBO: CA signs client certificate

   % openssl ca -in clientreq.pem -out client_cert.pem

7. // chop out the text portion of the certificates

% openssl x509 -in demoCA/cacert.pem -out cacert.pem

% openssl x509 -in client_cert.pem -out clientcert.pem

8. // Import CA certificate into keystore

   % keytool -import -alias caalias -keystore client_ks -storepass clientpass -file cacert.pem

9. // Import signed client certificate into keystore

   % keytool -import -alias clientalias -keystore client_ks -storepass clientpass -file clientcert.pem

Appendix H: 4.2 Event SDK clients interacting with PC 5.0 Dialer

For Proactive 4.x Event SDK client, certificates need to be updated to connect to Proactive Contact 5.0 dialer. Certificates can be got from PC5.0 dialer.

1. Close any running Event SDK client application
2. Take a backup of the existing key/certificates files.
3. Copy the key/certificate files from dialer available under "/opt/avaya/pds/openssl/" to the location where your Event SDK client certificates are present. Depending on the client, C++ or Java, copy CORBA key/certificates or Java keystore file respectively.