Avaya Aura® Application Enablement Services

JTAPI Programmer's Guide

02-603488
Release 7.1.1
August 2017
Issue 1.0

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.

For full support information, please see the complete document,

***Avaya Support Notices for Software Documentation*, document number 03-600758.**

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 1-800-242-2121 in the United States. For additional support telephone numbers, see the Avaya Web site: http://www.avaya.com/support.

CONTENTS

1. About this Document

1.1 Scope of this Document

This document shows you how to use the Application Enablement (AE) Services JTAPI implementation to develop, debug, and deploy telephony applications.

- Chapter 1: “About this document” details certain pre-requisites required to read this document and lists supporting reference documents.
- Chapter 2: “Avaya JTAPI Implementation” provides background information about JTAPI in general and the AE Services JTAPI implementation in particular.
- Chapter 3: “Getting Started” gets you ready to configure and program to this API, as well as walk through the JTAPI Exerciser and sample code.
- Chapter 4: “Writing a client application” and Chapter 5: “Compiling and Debugging” guide you in developing and debugging applications.
- Appendix A and Appendix B list Avaya specific deviations and enhancements to the JTAPI API respectively.
- Appendix C is a useful reference for TSAPI developers starting to use JTAPI or vice versa. It also provides a mapping of deprecated observer style events that were the norm in JTAPI 1.2 to their JTAPI 1.4 listener equivalents.
- Appendix D lists all of the values for the TSAPI error codes.
- The Glossary defines the terminology and acronyms used in this document.

1.2 Intended Audience

This document is written for application developers. A developer must know:

- Java™
- Telephony concepts
- JTAPI object model

You do not need to understand Avaya Aura® Communication Manager features and concepts, but such an understanding might be helpful.

If you are new to JTAPI, you may wish to start by reading the JTAPI overview whitepaper at the following link:

http://java.sun.com/products/jtapi/reference/whitepapers/index.html

Additionally, consider reading a portion of the JTAPI 1.4 specification, which can be found here:

http://java.sun.com/products/jtapi/

Upon downloading and unzipping the archive for the 1.4 specification, open the index.html file. After clicking the **Description** link at the top of this page, you will find several **Overview** paragraphs. Click the **JTAPI core Overview** link for a helpful overview of JTAPI concepts.

The Avaya JTAPI Javadoc, Avaya Aura® *Application Enablement Services JTAPI Programmers Reference*, can be found online on the Avaya DevConnect Web site (http://www.avaya.com/devconnect) and on the Avaya Support Web site (http://www.avaya.com/support) under “Communication Systems”.

For those new to Avaya Communication Manager, you may wish to take a course from Avaya University (http://www.avaya.com/learning) to learn more about Avaya Aura® Communication Manager and its features. It is recommended that you start with the *Avaya Aura® Communication Manager Overview* course (course ID AVA00383WEN).

1.3 Conventions used in this document

The following fonts are used in this document:

To represent...	**This font is used...**
Java class, method and field names	the `getDeviceID` method
Window names	The buttons are assigned on the **Station** form
Browser selections	Select **Member Login**
Hypertext links	Go to the http://www.avaya.com/support website.

1.4 Related documents

While planning, developing, deploying, or troubleshooting your application, you may need to reference other Avaya Aura® Application Enablement Services documents, Avaya Aura® Communication Manager documents, or JTAPI documents listed below.

1.4.1 Application Enablement Services documents

For developers, the other important source of Java API information is the Javadoc:

- *Avaya Aura® Application Enablement Services JTAPI Programmers Reference*

Here you can find details about each package, interface, class, method, and field in the API. You can also find out what parts of the JTAPI protocol have been implemented.

Other Application Enablement Services documents include:

- *Avaya Aura® Application Enablement Services Overview (02-300360)*
- *Avaya Aura® Application Enablement Services Installation and Upgrade Guide for a Software Only Offer* (02-300355)
- *Avaya Aura® Application Enablement Services Installation and Upgrade Guide for a Bundled Server* (02-300356)
- *Avaya Aura® Application Enablement Services Administration and Maintenance Guide* (02-300357)
- *Avaya Aura® Application Enablement Services OAM Help* (HTML)
- *Avaya Aura® Application Enablement Services 7.0 TSAPI for Avaya Aura® Communication Manager Programmer's Reference* (02-300544)

You can find all these documents online on the Avaya Support Center Web Site: http://support.avaya.com.

1.4.2 Avaya Aura® Communication Manager documents

Since JTAPI gives you programmable access to Avaya Aura® Communication Manager features, you may wish to reference documents about that system. The following documents from the Avaya Aura® Communication Manager documentation set provide additional information about administering Avaya Aura® Communication Manager. They are on the Avaya Support Centre Web Site (http://www.avaya.com/support).

- *Administering Avaya Aura® Communication Manager* (03-300509)
- *Avaya Aura® Communication Manager Feature Description and Implementation (555-245-205)*

1.4.3 JTAPI documents

The *Java Programmers Reference* (Javadoc) contains much of what you need to know about the JTAPI API. For JTAPI details not found in the *Javadoc* or this document, please refer to the JTAPI specification. The specification and API documents can be downloaded from the Java Community Process page for JTAPI - http://jcp.org/aboutJava/communityprocess/final/jsr043/index.html:

- *JTAPI 1.4 specification*

1.5 Providing documentation feedback

Let us know what you like or do not like about this document. Although we cannot respond personally to all your feedback, we promise we read each response we receive. Please email feedback to **document@avaya.com**

Thank you.

2. AE Services 7.x Modifications

2.1 New in AE Services 7.1.1

- To identify an incoming SIP trunk call as either voice or video via CTI, AE Services 7.1.1 has added support for Channel Type in CSTA Delivered and CSTA Established events.

  Delivered and Established event Channel Type values

  - UNKNOWN - The channel type is not specified.
  - VOICE - The channel type is voice.
  - VIDEO - The channel type is video.

2.2 Update for AE Services 7.0.1 server

- Support for the TLSv1.2 protocol on AE Services server 7.0.1

In AE Services 7.0.1, only the Transport Layer Security (TLS) 1.2 protocol is enabled by default. The lower level TLS protocols 1.0 and 1.1 are disabled by default. Note, according to the National Institute of Standards and Technology (NIST) Special Publication 800-52, TLS version 1.1 is required, at a minimum, in order to mitigate various attacks on the TLS 1.0 protocol. The use of TLS 1.2 is strongly recommended.

This change may cause older AE Services clients (version AE Services 7.0 and earlier) that are using TLS to fail to establish a secure socket connection to the AE Services 7.0.1 server. In order to achieve a more secure client/server socket connection, we encourage current client applications to use an AE Services 7.0 SDK where the TLS 1.2 protocol is supported. Note, all the latest versions of the AE Services 7.0 SDKs support TLS 1.2. If upgrading to AE Services 7.0 SDK is not a viable option, an AE Services administrator can enable the TLS 1.1 and/or TLS 1.0 protocol via the AE Services Management Console web interface. Note, all three TLS protocol versions can be active at the same time. This allows a gradual migration of current client applications to move towards a more secure TLS protocol over a period of time.

3. Avaya implementation of JTAPI

This chapter provides a high level overview of JTAPI concepts and packages

Avaya has implemented the following packages.

- JTAPI Core Package (javax.telephony)
- JTAPI Core Events Package (javax.telephony.events)
- JTAPI Call Center Package (javax.telephony.callcenter)
- JTAPI Call Center Events Package (javax.telephony.callcenter.events)
- JTAPI Call Control Package (javax.telephony.callcontrol)
- JTAPI Call Control Events Package (javax.telephony.callcontrol.events)
- JTAPI Media Package (javax.telephony.media)
- JTAPI Media Events Package (javax.telephony.media.events)
- JTAPI Private Data Package (javax.telephony.privatedata)

Note: The JTAPI Phone Package and the JTAPI Phone Events Package are not supported. Many of the features represented by these packages are available from the AE Services Device, Media and Call Control API.

Please refer to Appendix A for a full description of the behavior you can expect when you invoke Avaya's implementation of certain JTAPI methods.
Please refer to Appendix B for a description of Avaya specific enhancements to the JTAPI specification.

3.1 Understanding basic concepts of JTAPI

Before getting into package details, let's understand a few basic concepts of JTAPI. JTAPI stands for Java Telephony API. It is a standard that was created through the Java Community Process (JCP). JTAPI 1.4 is specified by Java Specification Request (JSR) 43.

The following diagram shows the JTAPI call model and the objects that compose the call model. A description of each object follows the diagram. The diagram and descriptions were largely taken directly from the JSR 43 specification. Further details can be found in that specification.

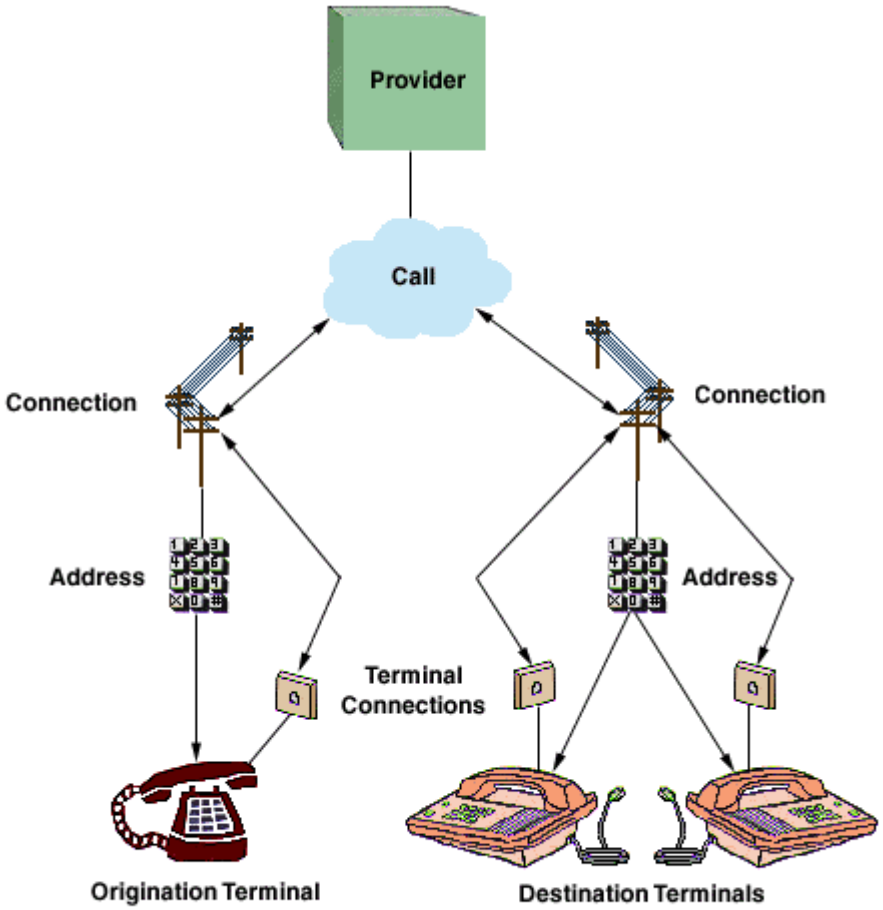


Figure 1: JTAPI Call Model

Provider Object

The Provider object is an abstraction of telephony service-provider software. In the case of Avaya, the Provider is an abstraction of a Communication Manager. A Provider hides the service-specific aspects of the telephony subsystem and enables Java applications and applets to interact with the telephony subsystem in a device-independent manner. The Provider is the core object with which a JTAPI application interacts to obtain references to the other JTAPI objects described below.

Call Object

The Call object represents a telephone call, the information flowing between the service provider and the call participants. A telephone call comprises a Call object and zero or more connections. In a two-party call scenario, a telephone call has one Call object and two connections. A conference call is three or more connections associated with one Call object.

Address Object

The Address object represents a telephone number. It is an abstraction for the logical endpoint of a telephone call. Note that this is quite distinct from a physical endpoint. In fact, one address may correspond to several physical endpoints (i.e. Terminals)

Connection Object

A Connection object models the communication link between a Call object and an Address object. This relationship is also referred to as a "logical" view, because it is concerned with the relationship between the Call and the Address (i.e. a logical endpoint). Connection objects may be in one of several states, indicating the current state of the relationship between the Call and the Address. These Connection states are summarized later.

Terminal Object

The Terminal object represents a physical device such as a telephone and its associated properties. Each Terminal object may have one or more Address Objects (telephone numbers) associated with it, as in the case of some office phones capable of managing multiple call appearances. The Terminal is also known as the "physical" endpoint of a call, because it often corresponds to a physical piece of hardware.

TerminalConnection Object

TerminalConnection objects model the relationship between a Connection and the physical endpoint of a Call, which is represented by the Terminal object. This relationship is also known as the "physical" view of the Connection (in contrast to the Connection, which models the logical view). The TerminalConnection describes the current state of relationship between the Connection and a particular Terminal. The states associated with the TerminalConnection are described later in this document.

3.2 Avaya implementation of standard JTAPI API

Most method calls are implemented faithfully as specified by the JTAPI specification with certain notable deviations. These deviations are mainly of the following types,

- Extra pre-conditions or CM / AE server settings required
- unsupported methods
- certain method post conditions

Please refer to Appendix A for more information.

3.3 Avaya extensions to JTAPI

The Avaya JTAPI implementation provides value-added extensions to the standard JTAPI specification.

These extensions fall under four main categories:

- Extensions to JTAPI exceptions (to provide CSTA/ACS error codes)
- Extensions to JTAPI provider events (to provide low level information regarding the provider state)
- Avaya Aura® Communication Manager extensions to JTAPI exposing CM features to a JTAPI application
- Private data extensions to JTAPI (to assist independent switch vendors in the creation of a private data package for their switches, or allow application programmers to use or interpret private data when they are supplied with private data in its raw form)

Most extensions are provided via interfaces in the com.avaya.jtapi.tsapi package that extend the standard JTAPI interfaces.

Please refer to Appendix B for more information.

4. Getting Started

This section describes what you need to do and what you need to know before you begin programming to this API.

4.1 Understanding the Avaya JTAPI architecture

The diagram below illustrates the architecture for the Avaya implementation of JTAPI. Application invocations on JTAPI objects result in CSTA 1 messages being exchanged with the TSAPI service on AE Services. The TSAPI service then converts between CSTA 1 and ASAI and exchanges ASAI messages with CM.

It is important to note that the JTAPI operational model is quite different than the CSTA operational model. That means that the Avaya JTAPI library is a rather sophisticated piece of software. It does not simply translate a method invocation to a single CSTA message and vice versa. Instead, it must maintain call state and often translate a method invocation into a series of CSTA messages.

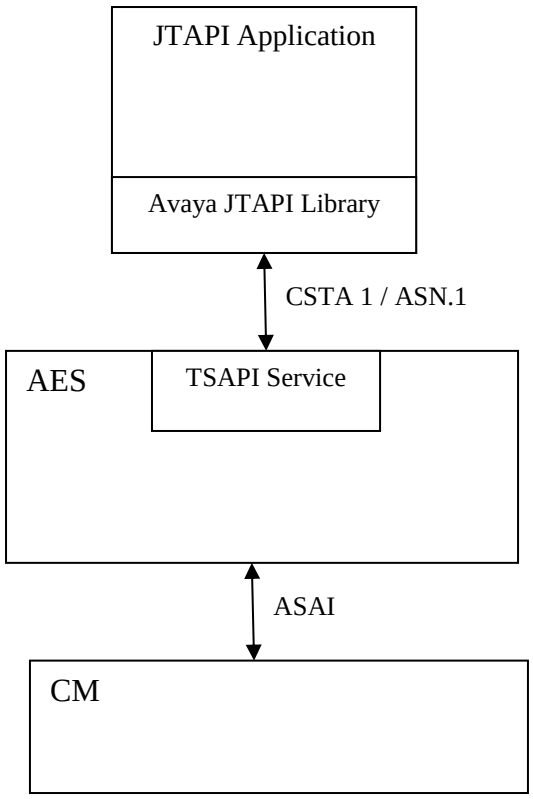


Figure 2: Avaya JTAPI architecture

4.1.1 Tlink

A Tlink (or T-Link) represents a TSAPI CTI link between the AE Services server and Communication Manager. When a TSAPI link is provisioned on the AE Services server, the TSAPI Service generates a Tlink identifier for the TSAPI link. There can only be one TSAPI CTI link (When the security for a TSAPI CTI link is adminsitered as "both" [encrypted and unencrypted], there are two Tlink identifiers associated with the TSAPI CTI link) for one AES-Avaya Aura® Communication Manager combination. However, multiple Tlinks can be created between one AE Services server and multiple Communication Managers or vice-versa. Thus, multiple TSAPI CTI links will allow a

JTAPI application to work with any number of AE servers each fronting up to 16 CM servers with unique identifiers for each of those pairs.

Each JTAPI application needs to specify one of the available Tlinks at the time of establishing a connection to the TSAPI Service on the AE Services server..Through the chosen Tlink, the TSAPI service learns which CM the application wishes to interact with on this session. (A single AES may support multiple CM systems).

The Tlink is of type String and has following format:

AVAYA#Switch_Connection#Service_Type#AE_Services_Server_Name

For example AVAYA#CMSIM#CSTA#AESSIM, where

1. "AVAYA" is a fixed constant.
2. Switch_Connection is a unique name assigned to identify a switch (i.e., Communication Manager). In general, hostname of the switch is assigned as the name of Switch Connection in the AE Services server.
3. Service_Type: refers to the CSTA service type. It can be either of the following:
    - "CSTA" – when accessing an unencrypted TSAPI Link (non-secure connection).
    - "CSTA-S" – when accessing an encrypted TSAPI Link (secure connection).

    The CSTA versus CSTA-S service types specify whether or not encryption is used between the application and the AE Services server.

4. AE_Services_Server_Name represents the host name of the AE Services server which is assigned during the AE Services installation.

4.1.1.1 Alternate TLINK

As of Release 4.1.0, AE Services introduces the Alternate Tlinks feature to provide a link failover capability for the JTAPI client. To effect this failover capability you must specify the alternate Tlinks in the JTAPI Configuration file.

Note: When multiple AE Servers are used as alternates, the username and password specified by the application in the getProvider() request should be configured identically for each AE Server.

Follow these steps to set up a list of alternate Tlinks in the TSAPI.PRO file (this file is a JTAPI configuration file. Refer section 3.2.4 for more details). You are essentially adding statements that specify a list of alternate Tlinks for the TSAPI Service.

- Locate the TSAPI.PRO file and open for editing (explained later)
- Add a list of alternate Tlink entries, using the following format.
  Alternates (TLINK) = TLINK1:TLINK2:TLINK3:TLINK4

where:

Alternates is the label for the first ordered list (you can have up to 16 lists)

(TLINK) is the name of the preferred Tlink, for example (AVAYA#CM1#CSTA#AESRV1). Be sure to enclose the preferred Tlink name in parentheses.
= The equal sign is a separator between the preferred Tlink, and the list of 1 to 4 alternate Tlinks. You must use the equal sign (=) to separate the preferred Tlink and the list of additional alternate Tlinks.
TLINK1:TLINK2:TLINK3:TLINK4 is an ordered list of Tlink names that are used as alternates if the preferred Tlink is not available. Be sure to separate each Tlink name with a colon. You can specify from 1 to 4 Tlinks for each list of alternates.

Examples

In Example 1, there are two AE Servers, AESRV1 and AESRV2, that each have a TSAPI link to the same switch, CM1. If AESRV1 is unavailable, the TSAPI client will attempt to use AESRV2 instead of AESRV1.

Example1
Alternates(AVAYA#CM1#CSTA#AESRV1)=AVAYA#CM1#CSTA#AESRV2

In Example 2, there are four AE Servers that each have a TSAPI link to the same switch, CM1.
If AESRV1 is unavailable, the TSAPI client will attempt to use AESRV2 instead of AESRV1.
If AESRV2 is also unavailable, then the TSAPI client will attempt to use AESRV3.
If AESRV3 is also unavailable, then the TSAPI client will attempt to use AESRV4.
If AESRV4 is also unavailable, then the TSAPI client will not be able to establish a connection with an AE server.

Example 2
Alternates(AVAYA#CM1#CSTA#AESRV1)=AVAYA#CM1#CSTA#AESRV2:AVAYA#CM1#CSTA#AESRV3:AVAYA#CM1#CSTA#AESRV4

4.2 Setting up the development environment

Applications can be developed in any environment supporting Sun Microsystems™ Java 2 Platform, Standard Edition (J2SE™) 1.5 or higher.

4.2.1 Downloading the Java SDK

With release 6.2 onwards, the JTAPI SDK supports OpenJDK (Open Java Development Kit), a free and open source implementation of the Java programming language. The AE Services server side Java services are based on OpenJDK 8 (aka OpenJDK 1.8) and the AE Services JTAPI SDK can use either OpenJDK 8 or Oracle JDK 8.

A release of 32-bit OpenJDK 8 for both Linux and Windows is available for download from the Avaya Support and DevConnect websites at no charge.

Note: JDK 1.4.2 or earlier versions are not supported.

4.2.2 Downloading the Application Enablement Services JTAPI SDK

Here are the hardware and software requirements for JTAPI SDK.

CPU	Any platform that supports Oracle's Java Virtual Machine (VM), Version 1.04 or later
Disk Space	20 MB
Browser	Internet Explorer (6.0 or later)
Java Development Kit (JDK)	AE Services requires Java SE SDK 8 or open JDK 8, or later.

To download the Application Enablement Services JTAPI SDK from the Avaya DevConnect Web site:

1. Go to http://www.avaya.com/devconnect.

2. Select **Member Login**.

3. Log in with your email address and password. If you do not have a login, then the link referenced above will give you the details about the DevConnect membership.

4. Click **Downloads**.

5. Click **Java Telephony API (JTAPI)**.

6. Click the arrow after **Programming Resources**, then check the **Software Development Kits** check box.

6. From the list of results, select the appropriate download –

- **Avaya Aura Application Enablement Services 7.0 JTAPI SDK and Client (Windows)**
- **Avaya Aura Application Enablement Services 7.0 JTAPI SDK and Client (Linux)**

7. Save the file to a temporary location on your computer.

8. If you are using Windows, the installer is a self-extracting zip file which, when executed, will extract the JTAPI SDK files into an appropriate folder on your file system.

a. To extract the JTAPI SDK, go to the directory that contains the JTAPI SDK file that you downloaded, and double-click on it.

b. Click OK to start installing the SDK.

c. Click Setup. The installer will extract the files to a temporary location, then display the End User License Agreement.

d. Carefully review the license agreement. When prompted, enter y to agree to the license terms.

e. When the installer prompts you to enter a directory to install the JTAPI SDK, enter a valid directory on your computer.

f. The installer will extract the JTAPI SDK files into a folder named 'jtapi-sdk' in the directory you entered above. You can safely move this folder to a different folder if needed.

9. If you are using Linux, the installer is an executable program which will, when executed, extract the JTAPI SDK files into the current directory

a. Log in to the computer as root. Go to the directory containing the JTAPI Linux SDK intstallation program that you downloaded.

b. Use the chmod command to make the JTAPI Linux SDK intstallation program executable. For example, chmod +x jtapi-sdk-7.0.0.64.bin.

c. Run the JTAPI Linux SDK intstallation program to begin the installation. For example: ./jtapi-sdk-7.0.0.64.bin.

d. Press the Enter key to display the End User License Agreement. Carefully review the license agreement. When prompted, enter y to agree to the license terms.

e. The installer will extract the JTAPI SDK files into a folder named 'jtapi-sdk' in the current directory. You can safely move this folder to a different folder if needed.

For a quick verification you can run this from a command line (DOS prompt / UNIX shell) – "java –jar JTAPI_SDK_PATH/lib/ecsjtapia.jar ECSJtapiVersion", where JTAPI_SDK_PATH is the absolute path to the directory where the JTAPI SDK was unzipped. Also, it is assumed that the system PATH points to the right JDK. The above command should print the Avaya JTAPI library version. For further testing, see sample TSTest in section 3.5.3

Important artifacts in this distribution include

Directory / File	Description
ant	Ant 1.7.1 distribution, used by scripts to compile the sample applications
conf/TSAPI.PRO	TSAPI.PRO file read by the sample applications
conf/log4j.properties	Log4j.properties files used by the sample applications.
javadoc	This includes javadocs for the javax/telephony, com/avaya/jtapi/tsapi and com/avaya/jtapi/tsapi/adapters packages. Please see the Accessing the client API reference documentation for more information.
lib/ecsjtapia.jar	Avaya's implementation of the JTAPI 1.4 spec.
lib/log4j-1.2.12.jar	Logging dependency used internally by ecsjtapia.jar for logging.
lib/servlet-api-2.5-6.1.1.jar	Servlet 2.5 specification required for recompiling the TSTest web application
resources/ avayaprca.jks	A Java Key Store containing the Avaya Product PKI Root CA Certificate.
sampleapps	Sample applications bundled with this release. Please see Learning from the sample code section for more details
sampleapps/bin	Contains an ant based build file (build.xml) to build and run the sample apps. As a convenience, build scripts (both MS-DOS and Linux based) to invoke this ant build file are also provided in this directory.
tools/jtapiex	The JTAPI Exerciser used to quickly test out JTAPI API's. Refer to the chapter on JTAPI Exerciser.
tools/bin	Contains scripts (both MS-DOS and Linux based) to start the JTAPI Exerciser
.classpath, .project	Eclipse specific files, used to quickly setup the file in the Eclipse environment. Please see the "Using Eclipse" section for more details
Readme.txt	Contains information regarding features new to this release, instructions for running sample applications, links to online resources and late breaking changes that could not be included in this document

4.2.3 Setting up your test environment

Before running an application you will need to have an AE Services machine set up. For instructions see the appropriate *Avaya Aura® Application Enablement Services Installation and Upgrade Guide* for the offer you have purchased (bundled server, software only or System Platform).

4.2.4 Configuring your JTAPI client library

In order for your JTAPI application to run, two main categories of configuration are required: configuration of JTAPI properties, and log4j configuration. A Sample configuration can be found in the TSAPI.PRO and log4j.properties files in the JTAPI_SDK_PATH/conf directory.

Please note that unlike previous releases, TSAPI.PRO is not a required configuration file anymore. Properties that were traditionally set in TSAPI.PRO can now be passed as system properties to your JTAPI application. However these property keys need to be prefixed by com.avaya.jtapi.tsapi

For example, the TSAPI.PRO debugLevel can also be passed as a system property named com.avaya.jtapi.tsapi.debugLevel.

Please note that in the case where a property is specified in both TSAPI.PRO as well as via its equivalent system property, the TSAPI.PRO entries will be given precedence.

4.2.4.1 Configuration properties

The JTAPI library can be configured by setting certain properties. See section "JTAPI properties" to learn more about these JTAPI properties.

The primary configuration required in order to run a JTAPI application is the AE Services IP Address or fully qualified domain name.

IPV6 Support:

With JTAPI 6.1 onwards, IPV6 support has been added. To specify an IPV6 address for the AE server, the IPV6 address should be enclosed within '[]'. For instance, if the AE server IPV6 address is 2001::10:15:192:100, then the address should be represented as [2001::10:15:!92:100]. This convention is required for all the three methods described below.

There are three ways to provide this information:

Method	Additional Information

Method	Additional Information
Using TSAPI.PRO	Create a file named TSAPI.PRO anywhere in the application's class path or in the directory from which the application is run. In case TSAPI.PRO is available at both of these locations, the file in class path is given precedence. This file should at a bare minimum contain line(s) of the format <AES_IP_ADDRESS>:<TSAPI_SERVICE_PORT> TSAPI_SERVICE_PORT is 450 by default IPV6 address example [2001::10:15:192:100]=450 IPV4 address example 192.168.1.1=450 Hostname example AES1.xyz.com=450
Using system properties	All system properties with keys pre-pended by com.avaya.jtapi.tsapi are read by the JTAPI library. If the AE server is not found in a TSAPI.PRO file, it will attempt to read it from the "com.avaya.jtapi.tsapi.servers" property. This property value must be of the format <AES_IP_ADDRESS1>:<TSAPI_SERVICE_PORT1>,<AES_IP_ADDRESS2>:<TSAPI_SERVICE_PORT2> IPV6 address example [2001::10:15:192:100]:450,[2010::11:15:192:11]:450 IPV4 address example 192:168.1.1:450,192.168.1.5:450 Hostname example AES1.xyz.com:450,AES2.xyz.com:450
Using the connect string	The string parameter to the JtapiPeer.getProvider() method can contain the AE server(s) address information. The format of the String is "<tlink>;login=<loginID>;passwd=<pw>;servers=<server entries>" Where server entries follows the format <AES_IP_ADDRESS1>:<TSAPI_SERVICE_PORT1>,<AES_IP_ADDRESS2>:<TSAPI_SERVICE_PORT2>

Method	Additional Information
	IPV6 address example [2001::10:15:192:100]:450,[2010::11:15:192:11]:450 IPV4 address example 192:168.1.1:450,192.168.1.5:450 Hostname example AES1.xyz.com:450,AES2.xyz.com:450

Please ping the AE server address/DNS name from the box where your client application runs, before configuration to verify network connectivity.

4.2.4.2 Log4j configuration

Previous releases of this JTAPI implementation relied on a custom logging implementation. As of version 5.2, log4j has been incorporated as the logging mechanism. Hence, as with all log4j based applications, logging can now be configured with a log4j.properties file.

In order to maintain backward compatibility, the implementation reads the debugLevel and other logging related properties and programmatically configures the log4j implementation with reasonable log4j equivalents.

A programmatic API call ITsapiProvider#setDebugPrinting(boolean) is also available. This method can be used to enable/disable debug printing in the implementation at runtime. Setting this parameter to 'true' will enable trace logging for the com.avaya.jtapi.tsapi logger. A value of 'false' results in logging either turned down to ERROR if error logging was already enabled when this method was called or OFF if error logging was originally disabled.

The implementation honors the following order of precedence:

1. Programmatic configuration (i.e. ITsapiProvider#setDebugPrinting(boolean))
2. TSAPI.PRO configuration
3. Log4j.properties configuration

The samples provide a log4j.properties file which contains four different commonly used configurations of log4j as examples. In all scenarios, the root appender is configured as a console appender.

Configuration	Description
A	In this case, only messages of level “error” and worse are logged to files named jtapiErrors.log.n, where n is a number from 1 to 15. Use of each file as a log destination is discontinued once the file size hits 50 MB and the next file starts getting used. Please note that this is the recommended level of logging that should be enabled by default by all client applications
B	In this case, only messages of level “error” and worse are logged to files named jtapiErrors.log. This file grows indefinitely, limited by only hard disk space availability. These messages are also duplicated on the console.
C	In this case two different files are produced, one containing trace data (all messages that the library can generate excluding audit dumps) and another containing only error data. These messages are not duplicated on the console.

D	In this case, trace data is logged to 16 rotating files 50MB each with fine grained logs enabled (Threshold set to ALL), while errors are logged to jtapiErrors.log. These messages are not duplicated on the console.

Please check the "debugging" section of this guide for more information on log4j related configuration for ecsjtapia.jar.

For more information regarding log4j please refer to the log4j home page at http://logging.apache.org/log4j

4.2.5 Running the JTAPI SDK contents

4.2.5.1 Using the scripts provided

The JTAPI SDK includes two scripts at JTAPI_SDK_PATH/sampleapps/bin and JTAPI_SDK_PATH/tools/bin to run the samples and Exerciser respectively. Both DOS and UNIX versions are available at these locations. These scripts setup the required PATH/java CLASSPATH and run the applications. (JTAPI_SDK_PATH points to the directory where the JTAPI SDK has been unzipped).

Running the samples

Navigate to the 'sampleapps/bin' directory and type "ant help". This will provide further help to run the sample you require.

For example, on Windows assuming JTAPI_SDK_PATH as D:\jtapi-sdk-7.0.0.64 …

```
D:\jtapi-sdk-7.0.0.64\sampleapps\bin>ant help
Buildfile: build.xml

help:
 [echo] buildAll  -- builds all the sample apps.
 [echo] clean     -- cleans up generated classes of all samples.
 [echo] buildAcdSampleApp  -- builds the ACD sample app.
 [echo] runAcdSampleApp  -- runs the ACD sample app.
 [echo] buildAutoAnswerSampleApp -- builds the AutoAnswer sample app.
 [echo] runAutoAnswerSampleApp -- runs the AutoAnswer sample app.
 [echo] buildCallLogSampleApp -- builds the CallLog sample app.
 [echo] runCallLogSampleApp  -- runs the CallLog sample app.
 [echo] buildRouteSampleApp -- builds the Route sample app.
 [echo] runRouteSampleApp -- runs the Route sample app.
 [echo] buildTsTestSampleApp -- builds the TsTest sample app.
 [echo] distTsTestWebApp -- creates a war for TsTest sample app
 [echo] runTsTestSampleApp -- runs the TsTest sample app.
 [echo] buildClick2CallSampleApp -- builds the Click2Call sample app
```

```
 [echo] runClick2CallSampleApp -- runs the Click2Call sample app
 [echo] help                 -- Prints this information message.



BUILD SUCCESSFUL
Total time: 0 seconds
D:\jtapi-sdk-7.0.0.64\sampleapps\bin>
```

As described above all “build*SampleApp” targets build individual samples or “buildAll” will build all samples, while “run*” targets run individual samples.

An exception is the “distTsTestWebApp” which does not run the TSTest sample, but builds it and packages it as a WAR file suitable for deployment into any J2EE container like Tomcat (http://tomcat.apache.org/) or WebLogic (http://www.oracle.com/appserver/weblogic/weblogic-suite.html).

For more information on WAR files please see http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/WebComponents3.html

Running the web sample

The web sample tstest.war can be run in any servlet container. The following instructions use Tomcat 6 as an example.

- Download Tomcat 6 from http://tomcat.apache.org/
- Unzip it to the file system and place tstest.war in directory TOMCAT_HOME/webapps where TOMCAT_HOME is the location where tomcat was unzipped.
- Start the server using the script startup.bat/startup.sh (depending on your OS) found in location TOMCAT_HOME/bin
- The servers should print an INFO level message stating that the tstest web application deployed successfully.
- Visit http://localhost:8080/tstest with a web browser.

Running the JTAPI Exerciser

Navigate to the tools/bin directory and type “ant”. This will start up a Swing GUI that can be used to quickly test out JTAPI API’s. For more information regarding the JTAPI Exerciser, please refer to the JTAPI Exerciser section.

4.2.5.2 Using the Eclipse development environment

If Eclipse is your development environment of choice, you can take advantage of the Eclipse project files that are included with the SDK.

Simply create a new project in the JTAPI SDK directory on your file system, and that project will automatically be configured with appropriate sample app source files and class-path.

The instructions below are applicable to Eclipse 3.2. Please search for “New Project Wizard” in the Eclipse help for more information on how to complete this operation.

- Select File > New > Project from the Eclipse menu.
- Select Java Project and click “Next”
- Enter a project name and select “Create project from existing source”
- In the “Directory” field, enter the directory path where the .project and .classpath files of the SDK reside,
- Ensure you are using a JDK 8 or higher and click “Finish”

Alternatively, you can use the import wizard of Eclipse to import from the file system to an existing project. Please search for “Import Wizard” in the Eclipse help for more information on how to complete this operation.

Running the samples

Please first update TSAPI.PRO and log4j.properties in the conf directory as described above.

Each sample can be run by right clicking on the sample’s main class and selecting “Run As >> Java Application” or “Debug As >> Java Application”

The main classes for each sample are as detailed below

Sample	Main Class
ACD sample	acd.ui.ACDFrame
TSTest sample	tstest.ui.TSTestFrame
Call log sample	calllog.CallLog
Route sample	route.ui.RouteFrame
Auto answer sample	autoanswer.ui.AutoAnswerFrame
Click2Call sample	click2Call.Click2Call

Running the JTAPI Exerciser in Eclipse

The instructions below to run the Exerciser as a java application, apply to Eclipse 3.2. They may vary slightly depending on your Eclipse installation.

- Please update TSAPI.PRO in the conf directory,
- Open the debug configuration dialog of Eclipse using Run > Debug … from the Eclipse menu.
- Right click on “Java Application” in the tree menu to the left and select “New”
- Ensure that the correct Java project containing the SDK is selected.
- Ensure that “Include libraries when searching for a main class” is selected
- Enter “jtapiex.Jtapiex” in the main class field and click “Debug” to start the Exerciser.

For more information on the Exerciser, please see Using the JTAPI Exerciser

4.3 JTAPI properties

Below is a list of properties that can be passed to the JTAPI implementation to allow greater control over its behavior. These properties can be set as System Properties or set in a configuration file named TSAPI.PRO. When using the TSAPI.PRO configuration file, place it in the CLASSPATH or the application’s working directory (defined by the java system property “user.dir”). If defining these

properties as System Properties, please prefix them with 'com.avaya.jtapi.tsapi.'. eg. com.avaya.jtapi.tsapi.debugLevel=5

Property Name	Description	Default value
altTraceFile	Full path of the file for JTAPI logs	console
traceFileCount	Maximum number of trace files.	10
traceFileSize	Maximum size of trace files. Size may be expressed as an integer in the range 0 – 2^63. Suffixes "KB", "MB" or "GB" can be used to indicate kilobytes, megabytes or gigabytes respectively	50MB
errorFile	Full path of the file for error logs	None
errorCount	Maximum number of error files.	10
errorFileSize	Maximum size of error files. Size may be expressed as an integer in the range 0 – 2^63. Suffixes "KB", "MB" or "GB" can be used to indicate kilobytes, megabytes or gigabytes respectively	50MB
debugLevel	The following values are accepted: 0 => No logging 1 – 5 => Information messages 6 => debug logs (Logs all CSTA/ACS messages exchanged) 7 => Logs entry / exit of API implementation invocations (methods in classes in the com.avaya.jtapi.tsapi.impl package and invocation of observer/listener callbacks) 7 (with enableAuditDumps property set to true) => Audit dumps and trace messages	0
maxWaitForSocket	Maximum time to wait (in seconds) for JTAPI to connect to TSAPI. Note: Windows platform may not support a value higher than 20. Even if the application sets a value higher than 20 seconds, on Windows it may timeout in less than 20 seconds.	20
propertyRefreshRate	The rate (in seconds) at which properties will be re-read from TSAPI.PRO / System properties. Only intervals of 10 seconds are supported. Values that are not a multiple of 10 will be rounded up.	100
callCleanupRate	The rate (in seconds) at which JTAPI will audit calls to clean up any calls that should no longer exist. Only intervals of 10 seconds are supported. Values that are not a multiple of 10 will be rounded up.	100
trustStoreLocation	The path to the trust store containing trusted certificates. See section 3.3.1	System classpath.
trustStorePassword	The password to the trust store containing trusted certificates specified by 'trustStoreLocation'.	"password"
verifyServerCertificate	It is a setting that determines whether the JTAPI client verifies the Fully Qualified Domain Name (FQDN) in the Server Certificate (for added security).	false

	• If you want the client to check the certificate for the FQDN, use this setting:verifyServerCertificate=true. • When this setting is true, the JTAPI client will validate the date range of the server's certificate and validate that the server's certificate chain is trusted • If you do not want the client to check the certificate for the FQDN, use this setting: verifyServerCertificate=false.	
tsDevicePerformanceOptimization	When true, internal TSDevice objects are not deleted from the Provider's device hash. This reduces the overhead of repeatedly sending CSTAQueryDeviceInfo requests to the switch.	false
maxThreadPoolSize	The maximum number of threads for the thread pool that is used to deliver events to the application.	20
enableAuditDump	A boolean value that indicates whether the audit thread should dump state of JTAPI objects like provider, calls, connections and agents to the log file.	false
getServicesTimeout	The maximum number of seconds to wait for AE Services to respond with the set of TLinks that it supports. In some cases, it may take longer than 10 seconds for TSAPI to timeout when attempting to resolve a Worktop IP Address to a hostname. Please try increasing this timeout if the JtapiPeer.getServices() API logs an error in the log file or it returns an empty list (assuming other parameters like the server name and port are correct).	10
callCompletionTimeout	The maximum number of seconds to wait for post-conditions to be met following a Call.connect(), LucentCallEx2.fastConnect(), CallControlCall.consult() or CallCenterCall.connectPredictive() operation. Please try increasing this timeout if any of the above API calls fails with an error “Could not meet post-conditions of <api-name>”	15

4.3.1 Specifying location of certificates

As of Release 4.1, the AE Services provides Transport Layer Security (TLS) for encrypting links between the JTAPI client and the AE Services server. If you plan to use encrypted links, you have the option of using the certificate from the AE Services license file (this is the default), or the CA certificate issued by a trusted in-house or third-party certificate authority (also referred to as your own certificate). For more information about certificates, see Chapter 1: Managing certificates from the document “Application Enablement Services TSAPI and CVLAN Client and SDK Installation Guide” (02-300543).

Note:
You do not have to add any configuration settings for certificates under the following conditions:

- You do not use encrypted links, and, hence, certificates.
- You use encrypted Tlinks with the default AE Services certificate.This certificate is signed by the Avaya Product Root Certificate Authority (CA), and a Java Key Store containing the certificate for the Avaya Product Root CA is installed with the JTAPI client as <installation-directory>\avayaprca.jks. Therefore, you do not need to configure the location of the Trusted CA File in the tsapi.pro configuration file.

Defining Certificate configuration properties - If you have installed your own certificates on the AE Server, you must define the following properties to specify where your certificates are located. For example:

trustStoreLocation=C:\\Documents and Settings\\user\\certs\\aesCerts.jks
trustStorePassword=password
verifyServerCertificate=true|false (optional)

As mentioned in Section 3.3 these properties can be defined through TSAPI.PRO or System properties.

4.4 Accessing the client API reference documentation

You will need to reference the *Java Programmer's Reference* (Javadoc) provided with this API. It is available on the Avaya support web site (http://www.avaya.com/support) as both a viewable HTML document and a downloadable zip file.

This html documentation is also included with the SDK. This documentation describes all of the interfaces and their parameters.

To access the Javadoc bundled in the SDK,

1. Go to the jtapi-sdk-7.0.x.y/javadoc directory.

2. Double click on index.html (or open this file with a browser).

The Javadoc includes descriptions of classes in the following packages:

● javax/telephony/**

This section of the Javadoc is also available if you download the JTAPI specification except for "Implementation Notes" attached to some API's. These sections describe where Avaya's implementation of the SDK

- deviates from the specification,
- adds extra functionality not described in the specification or
- doesn't support the functionality mentioned in the specification

Please refer to the "Implementation Notes" sections where ever available, while writing your application.

- com/avaya/jtapi/tsapi

This package contains non-standard Avaya specific additions to the JTAPI API, created to make Avaya Aura® Communication Manager features, TSAPI service extensions and Avaya private data extensions available to users.

- com/avaya/jtapi/tsapi/adapters

This package contains adapter equivalents for all listener interfaces in the JTAPI specification. The methods in these adapters are empty and provided as a convenience for easily creating listeners by extension, overriding only the methods of interest.

Unsupported parts of the specification like the javax.telephony.mobile package and a large majority of the javax.telephony.media package are not included in the Javadoc.

4.5 Learning from the sample code

4.5.1 ACD sample

This sample demonstrates using the JTAPI Call Center package. The ACD sample does the following operations:

- gets the ACDAddresses known to the provider and the agents logged-in at those ACD splits.
- tries to log-in two agents specified.
- Ensures the agents logged in successfully
- Tries to log out the agents
- Ensures that no agents remain logged in

4.5.2 CallLog application

This application uses JTAPI along with some of the AE Services extensions to JTAPI to access functionality specific to Communication Manager. The CallLog application has the following purposes:

- To monitor a terminal to log all incoming and outgoing calls to/from the specified device. "User To User information," if any, associated with the call is also displayed.
- To make calls and optionally send "User To User information" along with the call.
- To send DTMF through an active call.
- To disconnect an active call.

4.5.3 TSTest application (in the TSTest directory)

Use TSTest to make and hang up a call in order to test the installation of the JTAPI client software. TSTest can be executed as a web application or a stand-alone Java application.

Once the “TSTest Application” window is open, complete the fields as follows:

- In the LoginInfo->Login field, type your CT User user id.
- In the LoginInfo->Password field, type your CT User password.
- Click ‘Next’
- In the TSTestInfo->Service field, select the TLINK that corresponds to the AES-CM to be tested.
- In the TSTestInfo->Caller field, type a phone number that is administered in Avaya Communication Manager.
- In the TSTestInfo->Callee field, type another phone number that is administered in Avaya Communication Manager.
- Click Dial to make the call.

4.5.4 Route sample

The route sample demonstrates the use of the JTAPI Call Center package. It is a routing application that registers the VDN specified for routing. When a call is received by the VDN, the sample requests a route destination. When the route destination is entered, the call is routed to that destination.

It should be noted that an AES ADVANCED SWITCH license is required to run this sample application.

4.5.5 Auto answer sample

The auto answer sample demonstrates a client application which monitors a terminal and auto answers any call placed to that terminal. The call is dropped after a brief interval.

4.5.6 Click2call sample

The click2Call sample demonstrates a client application which monitors a terminal for calls. This application will also maintain a log of all these calls. Such a call log will enable an user to call back the calling party by just clicking on the log record. The status of the log record is updated accordingly. This sample application also supports LDAP configuration.

5. Writing a JTAPI application

This chapter describes how to write an application using the Application Enablement Services JTAPI SDK. It will frequently refer to the details in the Javadoc, so you may wish to have ready access to the Javadoc while reading this chapter. Read Accessing the client API reference documentation to find out how to get access to the Javadoc and where to find which kinds of information within the Javadoc.

5.1 Initializing a JTAPI application

Initializing a JTAPI application involves following the sequence of steps listed below:

- Getting the JtapiPeer object.
- Getting the services list.
- Getting the provider.

5.1.1 Getting the JtapiPeer object

The term 'peer' is Java nomenclature for "a platform-specific implementation of a Java interface or API". The JtapiPeer interface in the javax.telephony package represents a vendor's particular implementation of the Java Telephony API (in this case Avaya's).

An instance of the JtapiPeer object can be obtained using the JtapiPeerFactory class. The getJtapiPeer() method of the JtapiPeerFactory class returns a JtapiPeer object that, in turn, enables applications to obtain the Provider object.

The JtapiPeerFactory.getJtapiPeer() method returns an instance of a JtapiPeer object given a fully qualified classname of the class which implements the JtapiPeer object. If no classname is provided (i.e., if classname is null), a default class named DefaultJtapiPeer is chosen as the

classname to be loaded. If it does not exist or is not installed in the CLASSPATH as the default, a JtapiPeerUnavailableException exception is thrown.

Following is the syntax of the getJtapiPeer() method:

public static JtapiPeer getJtapiPeer(java.lang.String jtapiPeerName)

The code snippet below shows the procedure to obtain JtapiPeer object:

```
/*
* Get the JtapiPeer object using JtapiPeerFactory class
*/
try
{
peer = JtapiPeerFactory.getJtapiPeer(null);
}
catch (Exception excp)
{
System.out.println("Exception during getting JtapiPeer: " + excp);
}
```

The exception thrown is JtapiPeerUnavailableException which indicates that the JtapiPeer can not be located using the CLASSPATH that is available.

In most cases, an application can provide a null parameter and accept the default JtapiPeer implementation. The catch block is required because the API throws a checked exception, but no exception should be thrown because a null parameter was passed.

5.1.2 Getting the services list

Once the application has successfully accessed a JtapiPeer object, the application typically gets a listing of the services that are supported by the system(s) implementing the JtapiPeer object. The services supported are the links from the AE Services server(s) to one or more Communication Managers that are provisioned and active. These links are also referred to as CTI-links. The application uses the getServices() method to acquire the list.

Following is the syntax of the getServices() method:

```
public java.lang.String[] getServices()
```

After getting a JtapiPeer object, the application needs to retrieve a list of the supported CTI-links provisioned on the AE Services server. The getServices() method returns an array of the services that this implementation of JtapiPeer supports. This method returns null if no services are supported. The getServices() method obtains the IP addresses of the AE Services server(s) from:

-the TSAPI.Pro file.

-the com.avaya.tsapi.servers system property.

to get all the Tlinks (See Tlink) configured on each of the AE Services servers.

Please see the JTAPI properties section for more information regarding these options.

If no services are returned from the getServices() method request, check the AE Service server's IP address in the TSAPI.PRO file or the value of com.avaya.tsapi.servers system property, and the provisioning and state of the Tlinks on the AE Service server.

Following is a sample Java code snippet for retrieving the services supported by the JtapiPeer implementation:

```
Try
{
/*
* Get service method services supported by the JtapiPeer implementation
*/
String[] services;
services = peer.getServices();
if(services == null)
{
System.out.println("Unable to obtain the services list from JTAPI peer.\n”);
System.exit(0);
}
String myService = serv[0];
}
catch (Exception ex)
{
System.out.println("Exception during getting services: " + ex);
}
```

Once getServices() has returned a list of the services available on the AE Services server(s) listed in the TSAPI.PRO file or specified in the com.avaya.tsapi.servers system property value, the application can use any one of those services to create a Provider object.

5.1.3 Getting the provider

The next step for the application is to acquire a Provider instance from the JTAPI middleware. A Provider represents the telephony software-entity (i.e. AE Services) that interfaces with a telephony subsystem such as Communication Manager. Please refer to section 4.1 for more details.

The getProvider() method of the JtapiPeer object returns an instance of a Provider object.

Following is the syntax of the getProvider() method:

public Provider getProvider(java.lang.String providerString)

If the argument providerString is null, the getProvider() method returns a default Provider as determined by the JtapiPeer object.

The method takes a single string as an argument. This string contains a <service name>, a login=xxxx; and a passwd=yyyy; along with other optional parameters separated by semi-colons. The <service name> is the name of the service that the application wishes to utilize (typically one of the services returned by the getServices() API). The login=xxxx; is the account that the application will use for authentication and authorization purposes. The passwd=yyyy; provides the password for the login that is provided. An example of the argument to the getProvider() method is as follows:
AVAYA#CMPROD#CSTA#AESPROD;login=appaccount;passwd=Passw0rd;
As per the syntax given above, <service name> is mandatory and each optional argument pair that follows is separated by a semicolon. The keys for these arguments are Avaya implementation specific, except for two standard-defined keys, described below:

1. login: provides the login user name to the Provider.

2. passwd: provides a password to the Provider.

JTAPI also allows programmatic specification of AE Services server IP addresses and/or hostnames. It supports this by providing an additional optional keyword argument “servers=X” or “servers=X:P” permitted to be included in the semi-colon separated list of keyword arguments required for the JtapiPeer.getProvider() API call. More than one server can be provided by comma separating the entries e.g. “servers=X:P,Y:P1

```
myprovider = peer.getProvider(myService + ";login=" + login + ";passwd="
+ password + ";servers=135.8.1.2:450,[2010::11:15:192:11]:450");  //
":450" optional
```

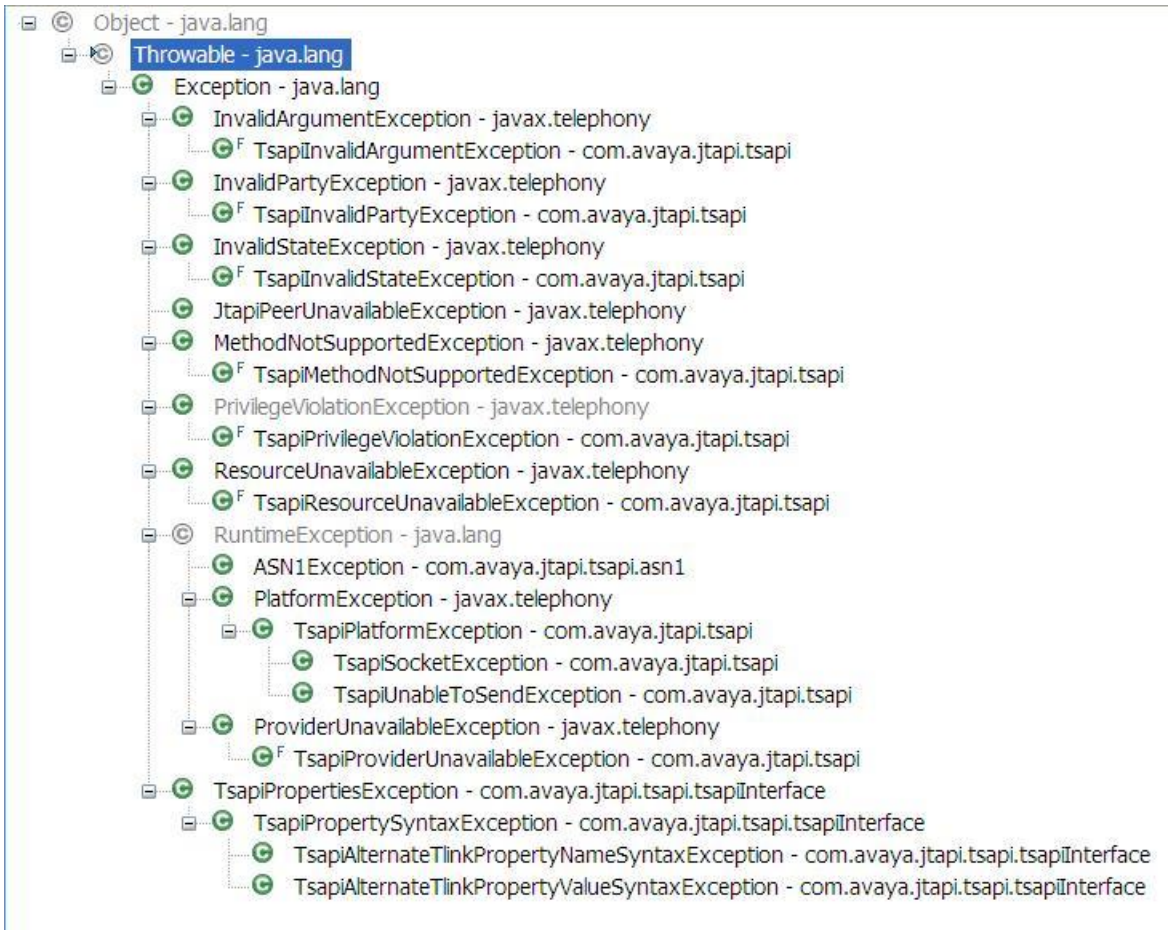
Following is the sample Java code snippet for getting the provider:

```
/** Create a provider with AE Services server CTI-link, user name and
password.
*
* @param String serv – AE services server cti-link selected by the
application.
* @param String login – user name for authentication purposes
* @paramString password – password for authentication purposes
* @throws Exception
*/
try
{
myprovider = peer.getProvider(myService + ";login=" + login + ";passwd=" +
password);
System.out.println(serv + ";login=" + login + ";passwd=" + password);
}
catch (Exception ex)
{
System.out.println("Exception during getting services: " + ex);
}
```

5.2 Catching Exceptions

Each service request may generate throw an exception; therefore the JTAPI application must be prepared to catch exceptions with a try/catch block around service requests.

The following diagram indicates the hierarchy of expected telephony related exceptions that may be thrown by this implementation.



Exceptions of the pattern Tsapi*Exception that extend the standard JTAPI specification's exceptions have an implementation specific error type and error code (e.g. TsapiInvalidStateException, TsapiInvalidArgumentException).

Error types include

- ACS (for ACS related exceptions)
- CSTA (for CSTA related exceptions)
- JTAPI (In case of failure to meet method specific pre-conditions)
- Internal (for internal implementation specific exceptions)

When the errorType is ACS or CSTA, the errorCode will contain the Tsapi ACS or CSTA error code which is documented in Appendix A of the Avaya Aura® Application Enablement Services TSAPI for Avaya Aura® Communication Manager Programmer's reference.

Most exceptions are checked exceptions and will be documented in the method signature, forcing client code to handle it. Please refer to the javadoc of these checked exceptions for more information.

The table below lists the runtime exceptions that this implementation may throw. These are unchecked, because a client cannot conceivably take corrective action on the fly in these conditions.

Exception	Location	Potential causes and solutions
ASN1Exception	Multiple	Indicates that an IO exception occurred while encoding/decoding ASN data types
TsapiPropertiesException	JtapiPeer.getServices() JtapiPeer.getProvider()	Indicates that an error was encountered while processing the value of Alternates (TLINK) JTAPI property specified via configuration file (TSAPI.PRO) or via system properties. Please ensure that the alternate Tlink entries are syntactically and semantically valid.
TsapiSocketException	Multiple	Informs applications that a socket IO error has occurred between the JTAPI client and the AE server
TsapiUnableToSendException	Multiple	Informs applications that the provider is in OUT_OF_SERVICE state and is unable to process requests
ProviderUnavailableException	JtapiPeer.getProvider()	Informs an application that a provider cannot be created for the given provider string
TsapiPlatformException	Multiple	API's that throw this exception are documented to do so via their javadoc throws clause. Please refer to the API's javadoc before using it. In addition this exception may be thrown in the following cases • If a confirmation event is not received from TSAPI for any CSTA request issued • If an ACS/CSTA error occurs • If the creation of the concrete implementations of Call, Terminal, Address or Trunk fails

It is recommended that the application catch and log all possible exceptions since this will be an important source of information for debugging the application.

5.3 Change from "observer" to "listener" paradigm

With 5.2 release onwards, the Avaya JTAPI implementation has deprecated the various observer interfaces available for handling JTAPI events and introduced support for listener interfaces. It is recommended that the future applications use listeners since the observer interfaces have been deprecated. The implementation still supports observer interfaces for backward compatibility

Listener events have the same content as Observer events. The principal motivation for the change is to keep up with the changes in the Java SDKs and communities; in this case, the shift away from the Observer pattern and toward the Listener pattern.
Such a change simplifies the client code, which does not have to implement conditional statements to cycle through events at runtime; the appropriate listener callback (which contains client logic) is automatically called by the JTAPI library.

JTAPI 1.4 Listeners are different from JTAPI 1.2 Observers in the following ways:

- EVENTS MOVED UP A PACKAGE – JTAPI is organized into 6 basic packages:
  - core (javax.telephony),
  - callcontrol (javax.telephony.callcontrol),
  - callcenter (javax.telephony.callcenter),
  - phone (javax.telephony.phone),
  - mobile (javax.telephony.mobile) and
  - private (javax.telephony.private).

  In JTAPI 1.2, events are defined for each basic package; the events for each are defined in a sub-package called X.events (e.g. javax.telephony.events for the core package events). In JTAPI 1.4, the interfaces which define the events are moved "up" out of the "events" sub-packages and into the basic package. This was done to help avoid confusing the events. So while in JTAPI 1.2, the interface javax.telephony.events.Ev represents the superclass of all Observer events, in JTAPI 1.4 the interface javax.telephony.Event represents the superclass of all Listener events.

- EXPANDED NAMES – In JTAPI 1.2, events all inherit from the root interface "Ev" (javax.telephony.events.Ev); all Observer events end in "Ev"; generally the first component of the event name is an abbreviation (e.g. Provider observer events start with "Prov"). In JTAPI 1.4, Listener events all inherit from the root interface "Event" (javax.telephony.Event); all Listener events end in "Event"; generally the first component of the event name is a full name (e.g. Provider listener events start with "Provider").

- FEWER EVENT INTERFACES – the JTAPI mobile community objected to the many interfaces defined in the JTAPI 1.2 specification. In order to meet their memory constrained needs, the number of interfaces was reduced in the listener hierarchy. In the new scheme, a group of events were all represented by a single interface; each "old" event was then represented solely by the event's ID.

- A METHOD FOR EACH EVENT – the Listener pattern calls for there to be a method named to match each kind of event, and for that event only to be delivered to that method. Because the JTAPI middleware provider provides classes called adapters, or classes which provide a default "do-nothing" implementation for a Listener interface, this makes it easy for application developers to write very simple Listener objects.

- META EVENTS GIVE WARNING ABOUT CALLS – Meta events were added to JTAPI 1.4. JTAPI already had the concept of a meta event code (Ev.getMetaCode) and new meta event flag (Ev.isNewMetaEvent). The former gave a hint as to the larger process that led to these small grained events; the latter gave an indication as to which sequences of JTAPI events were, together, generated because of a single outside stimulus (like the receipt of a CSTATransferredEvent). This one item (sending actual MetaEvents) represents the only new real content for JTAPI 1.4 Listener support, and it is a modification of something JTAPI already provides (the meta code and new event flag data). In 1.2, the application would have to invoke the 'isNewMetaEvent' flag to identify a batch of JTAPI events that corresponded to a higher-level operation. In 1.4 instead, meta events are defined in pairs, eg. CallListener.singleCallMetaProgressStarted() and CallListener.singleCallMetaProgressEnded(). For more information, please refer to Javadoc for javax.telephony.MetaEvent

5.4 Requesting notification of events

Each individual change to a JTAPI object is reported by an event sent to the appropriate Listener (or Observer). These changes could be as a result of JTAPI receiving an unsolicited event from AE Services (e.g. CSTADeliveredEvent, CSTAHeldEvent), or receiving a synchronous/asynchronous

response to requested action (e.g. Transfer a call, Snapshot a call to get latest status). A JTAPI application can choose to be notified of events by implementing and adding listeners.

To listen for certain types of events, an application must:
1. Implement a listener.
2. Add the listener.

Note:

1. Avaya recommends that the JTAPI application use a different listener implementation instance for each JTAPI object. In other words, applications should not re-use the same listener instance for any JTAPI object (Call/Terminal/Address). If applications do use the same listener instance on all devices, they might still get all the events, but the CSTACause (obtained using getCSTACause() method of TsapiCallEvent/CallEventImpl) might not represent the value which the application expects..
2. Once the application receives an event, release the event thread immediately and continue with event processing on a different thread. If this recommendation is not adhered to, it could drastically reduce the performance of the JTAPI application. This is because the JTAPI application will not be notified of other events until the previous event's callback method is complete.

Each event triggers a specific callback method in a listener. Listener implementations provide a way for your application to respond to each event. For every listener, Avaya also provides an adapter class which provides a default implementation for each callback method which ignores the received event.
An application developer can extend this abstract adapter with his or her own concrete implementation and override only the callbacks he/she is interested in. Extending this adapter (instead of directly implementing the listener) allows the application developer to define only the callbacks he/she is interested in and delegate the rest to the default implementation in the adapter class. This simplifies the client application's listener code.

Provided below is a list of Listeners that are supported. Also included is the name of the adapter class (for each Listener), provided by Avaya.

JTAPI Listener interface	**Avaya JTAPI's Adapter class**
Package javax.telephony	*Package com.avaya.jtapi.tsapi.adapters*
AddressListener	AddressListenerAdapter
CallListener	CallListenerAdapter
ConnectionListener	ConnectionListenerAdapter
ProviderListener	ProviderListenerAdapter
TerminalConnectionListener	TerminalConnectionListenerAdapter
TerminalListener	TerminalListenerAdapter
Package javax.telephony.callcenter	
ACDAddressListener	ACDAddressListenerAdapter
AgentTerminalAddressListener	AgentTerminalAddressListenerAdapter
Package javax.telephony.callcontrol	
CallControlAddressListener	CallControlAddressListenerAdapter
CallControlCallListener	CallControlCallListenerAdapter
CallControlConnectionListener	CallControlConnectionListenerAdapter
CallControlTerminalConnectionListener	CallControlTerminalConnectionListenerAdapter
CallControlTerminalListener	CallControlTerminalListenerAdapter
Package javax.telephony.privatedata	

JTAPI Listener interface	Avaya JTAPI's Adapter class
PrivateDataAddressListener	PrivateDataAddressListenerAdapter
PrivateDataCallListener	PrivateDataCallListenerAdapter
PrivateDataProviderListener	PrivateDataProviderListenerAdapter
PrivateDataTerminalListener	PrivateDataTerminalListenerAdapter

The list of specific listener registration methods that are supported is given in the table below. Note that Avaya's implementation of JTAPI does not support anything in the Phone package or the Mobile package; the same applies to Observers, Listeners and events in the Media package.

ACDAddress.addListener()
Address.addCallListener()
AgentTerminal.addListener()
CallCenterCall.addListener()
CallControlAddress.addListener()
CallControlCall.addListener()
CallControlConnection.addListener()
CallControlTerminalConnection.addListener()
CallControlTerminal.addListener()
Address.addListener()
Call.addListener()
Provider.addListener()
Connection.addListener()
TerminalConnection.addListener()
Terminal.addListener()
Terminal.addCallListener()

5.5 Call Control – Basic Telephony operations

This section covers a few basic telephony operations that can be performed in JTAPI. These operations can be used to create larger applications covering complex scenarios.

5.5.1 Detecting an incoming call

The destination Connection state changes to CallControlConnection.ALERTING when the destination Address is being notified of an incoming call. This change is signaled to the application by invoking the CallControlConnectionListener implementation's connectionAlerting() method.

The following code snippet shows the implementation of the call detection process.

```
//implementation of javax.telephony.callcontrol.CallControlConnectionListener
public void connectionAlerting(CallControlConnectionEvent event) {

             Call call = event.getCall();

             String callingDeviceID = null;

             String calledDeviceID = null;

             if(event.getCallingAddress() != null)
             {
                callingDeviceID = event.getCallingAddress().getName();
             }

             if(event.getCalledAddress() != null)
             {
               calledDeviceID = event.getCalledAddress().getName();
             }

              System.out.println("Incoming call from " +
                                callingDeviceID+ " to " + calledDeviceID);
      }
```

5.5.2 Answering a call

5.5.2.1 Triggering Answer from the Application

An incoming call can be answered by using the answer() method of the TerminalConnection object. The TerminalConnection object can be retrieved by using the getTerminalConnection() method of the Connection object. The getConnections() method of the Call object can be used to get the array of Connection objects associated with a Call.

When a call is answered, the Connection state changes from CallControlConnection.ALERTING to CallControlConnection.ESTABLISHED

A sample code snippet for answering the call at a particular TerminalConnection is shown below.

```
    Call mycall;
    Address myStationAddress; // Address for the station extension
    Terminal myStationTerminal; // Terminal for the station extension

    public void answerCall() throws Exception
    {
        Connection localConn = null;
        TerminalConnection[] terminalConns = null;

        // Get all the connections related to this call object
        Connection connection[] = this.mycall.getConnections();
        if( connection == null )
        {
            /* If connection array is null, there are no connections
            associated with the call, this can happen if Call is no
            longer ACTIVE. This can happen if there is a race condition
            with a disconnect.*/
            System.out.println("There are no connections associated with "+
            "the call");
            return;
        }

        for( int conn_index = 0; conn_index < connection.length; conn_index++)
```

```
        {
            // get the connection object
            localConn = connection[ conn_index ];
            /* find the Address for the station extension from where
            the call needs to be answered*/
            if(localConn.getAddress().equals(myStationAddress)){

                //get the terminal connections for the connection
                terminalConns = localConn.getTerminalConnections();
                if( terminalConns == null ){
                    System.out.println("No valid TerminalConnection found.");
                    return;
                }

                for( int term_conn_index = 0; term_conn_index <
                terminalConns.length; term_conn_index++ ){

                    TerminalConnection termConn =
                        terminalConns[term_conn_index ];
                    /* find the Terminal for station extension from where
                    the call needs to be answered*/
                    if(termConn.getTerminal().equals(myStationTerminal)){

                        try{

                            // Answer the call at the specified
                            // terminal connection.
                            if(termConn.getState()==
                                TerminalConnection.RINGING){
                                termConn.answer();
                            }
                        }
                        catch(Exception e){
                            System.out.println("Exception occurred " +
                                    "during Answer Call: " + e.getMessage());
                            return;
                        }
                        return;

                      }// End of if
                  }// End of for
              }// End of if
          }// End of for
    }
```

5.5.2.2 Events Received When a Call is Answered

When a call is answered, either manually or programmatically, the state of the destination Connection changes from CallControlConnection.ALERTING to CallControlConnection.ESTABLISHED. If the application has registered a CallControlConnectionListener for either the Terminal or Address of the originating or destination station extension, the listener implementation's connectionEstablished() method will be invoked. The event will contain information for the destination Connection.

The sample code snippet shown below demonstrates how to handle the CallCtlConnEstablishedEv event.

```
//implementation of javax.telephony.callcontrol.CallControlConnectionListener
    public void connectionEstablished(CallControlConnectionEvent event) {

        Connection conn = event.getConnection();

        System.out.println("Connection for Address " +
                conn.getAddress().getName() + " is in " +
        "Established state.");

        // Add code to handle connection established event here
    }
```

5.5.3 Disconnecting a call

5.5.3.1 Triggering a disconnect from the application

During an active call, the Connection is in one of the following states:

- Connection.INPROGRESS
- Connection.ALERTING
- Connection.CONNECTED

The state of the connection changes to the Connection.DISCONNECTED state after the connection is disconnected, say when a user drops from an active call.

In order to disconnect a Connection programmatically, the application needs to call the disconnect() method of the Connection object.

The following code snippet shows how to use the disconnect() method to disconnect a call.

```
        // The reference to the Provider object is obtained during
        //application initialization
        Provider myProvider;
        String myAddressName = "40061";
        Connection localConn = null;
        Address address = myProvider.getAddress (myAddressName);
        Connection connection[] = address.getConnections();
        /* When there is more than one call at the address, then the
         * following code selects the first connection in the
         * CONNECTED state.
         */
        for (int connectionIndex = 0; connectionIndex <
connection.length;
        connectionIndex++){
            Connection conn = connection[connectionIndex];
            int state = conn.getState();
            if (state == Connection.CONNECTED){
                localConn = conn;
                break;
            }
        }
        /*cannot locate a connection in Connection.CONNECTED state with
        the specified Address in it.*/
        if(localConn == null)
            return;

        try{
            localConn.disconnect();
```

```
        }
        catch ( Exception e ){
            System.out.println( "Exception occurred during disconnecting
" +
                    "the Connection: " + e.getMessage() );
        }
```

5.5.3.2 Events received when a Connection is disconnected

If the application has registered a CallControlConnectionListener for either the Terminal or Address of the originating or destination station extension, the listener implementation's connectionDisconnected() method will be invoked. The event will contain information for the destination Connection. After receiving this event, the associated Connection moves to the Connection.DISCONNECTED state.

5.6 Getting DNIS, ANI information for a call

When the application receives listener events, it can invoke 'getCallingAddress()' to obtain ANI and 'getCalledAddress()' to obtain the DNIS associated with the call. To extract the original call information, the application should invoke the method getOriginalCallinfo whose return type is an OriginalCallInfo object. On the OriginalCallInfo object invoke the 'getCallingDevice()' to obtain ANI and 'getCalledDevice()' to obtain the DNIS

5.7 Cleanup

It is important to free resources when they are no longer needed. For example, if the application is not interested in receiving any more events, it should remove the listener from the device.Please note that if the listener object is not removed, then java would not be able to garbage collect it.The application should also shutdown the provider, when it does not need it any more. This will allow the socket connection to AE Services Server to be released and thus free up valuable system resources.

5.8 Security Considerations

This section covers the security measures that the JTAPI library takes in order to protect the application.

5.8.1 Authorization Measures

AE Services optionally enforces an authorization policy as specified in the Security Database (SDB) to ensure that only authorized users can monitor and control a given device.
The SDB allows an administrator to give a user restricted access by allowing control of a specific device or list of devices. An administrator can also allow a user to monitor/control any device by granting them "Unrestricted Access". However in the latter case, any provider API invoked to obtain a list of addresses will fail. For eg. CallCenterProvider.getACDAddresses() will fail. For any such API, the application should use a CTI user having restricted access.

The administrator can also disable the SDB entirely, which turns off all authorization enforcement and allows any user to monitor or control any device.

Please see the AE Services Administration and Maintenance Guide, 02-300357, for more information about SDB administration.

(A CTI user can be administered for "Unrestricted Access" via the "Edit CTI User" Web page AE Services OAM)

5.8.2 Transport Security

JTAPI can establish either a secure (SSL) or a non-secure (non-SSL) TSAPI session connected to the given Telephony server. The type of session returned is based on the protocol in the Tlink name. For protocol "CSTA", a non-secure session is created. For protocol "CSTA-S", a secure session is created.

For example:

This link will result in a non-secure session - AVAYA#CMSIM#CSTA#AESSIM
This link will result in a secure session with AES - AVAYA#CMSIM#CSTA-S#AESSIM

To establish a secure session, the JTAPI client library will need the keystore properties to be configured (see JTAPI.PRO properties). Such configuration is only required if the default AE Services certificate is not used by the AE Service Server.

5.9 Heartbeats

JTAPI has a heartbeat monitoring mechanism, whereby messages sent at regular intervals, by the TSAPI service, are received by it. This provides a way for the JTAPI middleware to verify that the AES is operational even if there is no other activity from the application. JTAPI application can adjust the heartbeat interval using the ITsapiProvider.setHeartbeatInterval() method.

If the JTAPI library misses two consecutive heartbeat messages, it shuts down the provider. If the application has ProviderListener registered, then it will receive notification about this event.

In addition to this heartbeat mechanism (which monitors the link between JTAPI library and AES), JTAPI also monitors the link between AES and the Avaya Aura® Communication Manager (CM). If that link goes down, JTAPI receives a notification from AES and the provider state is set to OUT_OF_SERVICE. This feature is available from release 5.2 onwards.

5.10 JTAPI Applets

This section explains the setup required for running JTAPI Applets in a browser. It describes the steps for making the applet classes available in Internet Explorer (6.0 or later). In this configuration, the clients that will access the web server do not need to install the JTAPI software.

1. Copy the avayaprca.jks, the ecsjtapia.jar and the tsapi.pro files to the directory on the Web server that will host the web page (all files should be in the same directory).
2. Edit the tsapi.pro file to include the TCP/IP addresses or host names of the AE Servers that will be used. The default port number is 450.
3. “trustProxy = true” – the Java plug-in must have this system property value. To confirm this, say in Internet Explorer, go to Tools->Sun Java Console, and then in that window type the letter “s” – it will say “Dump system and deployment properties”. Under the system properties, confirm “trustProxy = true”.

6. Compiling and debugging

6.1 Installing Java

The java core packages need to be installed separately. Please download and install the Java installation appropriate for your operation system from the

Oracle Java website (http://www.oracle.com/technetwork/java/javase/downloads/index.html) or

Open JDK from http://download.java.net/openjdk/jdk8/

6.2 Compiling and running

Compiling a test program can be done via the operating system's command line interface or via a development environment like Eclipse.

In order to compile your application in any case, you need to ensure that

- You are using a JDK 8 or open JDK 8 or higher version of Java
- The ecsjtapia.jar file from the SDK is in your CLASSPATH.

6.2.1 Compiling using the command line interface (CLI)

Installing Java will install the tools "java" and "javac" in your system PATH automatically on Windows. On Linux, soft symbolic links like /usr/bin/java and /usr/bin/javac to your installation may be required. Please read the installation instructions posted on the Sun website.

6.2.1.1 Checking the Java version

Please check the version used by typing "java –version". As mentioned above, a version greater than 1.5 should be used

6.2.1.2 Compiling a simple application

The Windows compile command may look something like this

```
D:\jtapi-sdk-7.0.0.64>javac -classpath lib\ecsjtapia.jar *.java
```

This will compile all java files in the current directory

The Linux equivalent would be below

```
[jdoe@ linux-box jtapi-sdk-7.0.0.64]# javac -classpath lib/ecsjtapia.jar
*.java
```

6.2.1.3 Compiling a complex application

Ant is the recommended build tool for any moderately complex application.

This tool needs the system variable JAVA_HOME set to point to the java installation you wish to use. Please ensure that this variable points to a Java Development Kit (JDK) and not a Java Runtime environment (JRE)

On Windows a sample command to set this variable will be

```
D:\jtapi-sdk-7.0.0.64>set JAVA_HOME= C:\Program Files\Java\jdk1.8.0_01
```

The Linux bash equivalent would be below. Please use the appropriate syntax of your shell

```
[jdoe@ linux-box jtapi-sdk-7.0.0.64]# export JAVA_HOME=/usr/java/
jdk1.8.0_01
```

Please refer to

- The build script (ant.bat or ant.sh depending on your OS variant) for an example of how to use Ant from the command line. The above scripts set the path to use the ant 1.7.1 distribution bundled with the SDK. This distribution is the same as the distribution 1.7.1 available at http://ant.apache.org/ except that it uses extra ant contrib. tasks defined at JTAPI_SDK_PATH/ant/lib/ant-contrib-1.0b1.jar.
- The build file build.xml at JTAPI_SDK_PATH/sampleapps/bin for an example of how to compile an application using ant.

6.2.1.4 Running a simple application

The Windows run command is similar to the compile command except that the “java” executable needs to be used instead of “javac”.

Log4j needs to be in the CLASSPATH. By default, the log4j jar in JTAPI_SDK_PATH/lib is included in the manifest of ecsjtapia.jar. If the log4j jar is not located in the same directory as ecsjtapia.jar, it needs to be passed in the CLASSPATH.

A class with a valid main() entry point needs to be specified.

```
 D:\jtapi-sdk-7.0.0.64>java -classpath lib\ecsjtapia.jar;log4j-
1.2.12.jar;.;conf TestApplication

D:\jtapi-sdk-7.0.0.64>
```

Assuming that a class TestApplication.java exists in the current directory and has an entry point of the signature “public static void main(String[] args)”, the above command will run that class.

Please note the contents of the CLASSPATH used. It contains

- The Avaya implementation library (ecsjtapia.jar)
- The Log4j library, (on which ecsjtapia.jar depends for logging)
- The current directory (assuming that TestApplication.java has been compiled to this directory, i.e. the current directory was the build directory for compiling TestApplication.java)

- The conf directory (so that TSAPI.PRO and log4j.properties are included in the CLASSPATH)

The linux equivalent is presented below

```
[jdoe@ linux-box jtapi-sdk-7.0.0.64]# java -classpath
lib/ecsjtapia.jar:log4j-1.2.12.jar:.:conf TestApplication
```

6.2.1.5 Running a complex application

Similar to compilation, Ant is the recommended tool for running any moderately complex application. Please see the ant compilation setup section for information regarding setting up ant. Please refer to the build file build.xml at JTAPI_SDK_PATH/sampleapps/bin for an example of how to run an application using Ant.

6.2.1.6 Deploying a simple application

Java applications are traditionally deployed as jars. Your application can be bundled as a jar with the class-path attribute including ecsjtapia.jar and a main-class attribute pointing to your main class. Please remember to bundle the log4j jar and respective configuration files i.e. TSAPI.PRO and log4j.properties along with your application.

Assuming TestApplication.class as the main class, an example Manifest.mf file would be

```
Class-Path: ecsjtapia.jar  log4j-1.2.12.jar

Main-Class: TestApplication
```

The jar can then be created using the "jar" tool included with the JDK. A sample command on Windows would be

```
D:\jtapi-sdk-7.0.0.64> jar -cfm testapp.jar Manifest.mf *.class
log4j.properties TSAPI.PRO
```

This command will create a jar testapp.jar using the Manifest.mf specified and containing all class files as well as the configuration files log4.properties and TSAPI.PRO in the current directory.

Please ensure that ecsjtapia.jar, testapp.jar and log4j-1.2.12.jar are in the same directory when run. An indicative run command would be

```
D:\jtapi-sdk-7.0.0.64> java -jar testapp.jar
```

On Linux, you may need to soft link the jar tool before usage. The jar creation and run commands are similar to the Windows commands above.

6.2.1.7 Deploying a complex application

Binaries are best created as part of the build script using Ant in this case. Please refer to the Jar task of Ant at http://ant.apache.org/manual/CoreTasks/jar.html for information on how to create jars

6.2.2 Compiling using Eclipse

This can be done by setting up a java project in Eclipse. Please search for "New Java Project Wizard" in Eclipse for instructions on how to do so.

The following instructions apply to Eclipse 3.2

- Select File > New > Project > Java Project from the Eclipse menu and click “Next”.
- Enter a name for the project and ensure that a JVM > 7 is used for this project.
- Go to the libraries tab and click “Add External Jars”. Browse to the directory. JTAPI_SDK_PATH/lib and select ecsjtapia.jar and log4j-1.2.12.jar. Click “Finish”
- Copy a properly configured TSAPI.PRO and log4j.properties from JTAPI_SDK_PATH /conf to a source folder in your Eclipse project.
- Create and your client application in this project. Please see the section “Writing a client application” for more information.

Eclipse provides wizards to create jars out of project artifacts. Please search the Eclipse documentation for “Creating a new Jar file” for more information on this procedure.

6.3 Debugging

6.3.1 Client-side debugging

Sample code can be debugged by opening the code in an IDE and stepping through it with a debugger. However, debugging the JTAPI implementation involves configuring logging at an appropriate level, executing the scenario and examining the generated logs

Following is a walkthrough on how to configure logging for the Avaya JTAPI implementation.

JTAPI 5.2 supports log4j based logging. JTAPI application developers can now control logging in 2 ways as described below.

6.3.1.1 Using a log4j.properties file

JTAPI application developers can also control logging by specifying a log4j properties file to control JTAPI logging. JTAPI uses com.avaya.jtapi.tsapi as the internal logger. JTAPI application developers are advised not to use this logger but use com.avaya.jtapi as the logger to control JTAPI logging. Developers are encouraged to use this method to control JTAPI logging. Please refer to “Configuring the JTAPI client library” for log4j properties files examples.

6.3.1.2 Using TSAPI.PRO

JTAPI logging can be controlled by setting the properties in TSAPI.PRO. The log4j level can be controlled by setting the debugLevel property in TSAPI.PRO. The values for the property debugLevel must be between 0-7.

The “altTraceFile”, “traceFileCount” and “traceFileSize” keys control trace logging.

Similarly, the “errorFile”, “errorFileCount” and “errorFileSize” control error logging.

Please refer to the section on Configuring the JTAPI client library for more information regarding both debugLevel and these trace and error logging attributes.

The exact behavior of JTAPI logging will depend on the properties that have been set in the TSAPI.PRO file. Since the trace and error parameters mentioned above are not mandatory, please keep in mind the following points while setting properties:

- If debugLevel property is set to a value greater than 0 and no other trace or error logging properties are present then JTAPI will create a console appender and log all details to the console.
- If debugLevel is set to a value greater than 0 and an errorFile is mentioned then JTAPI will log at a level corresponding to debugLevel and also turn on error logging to the file that is mentioned as value for errorLogging.
- If debugLevel is set to zero and errorFile is set then log4j level will be set to ERROR and error messages will be sent to the error file. Setting errorFile property will override debugLevel property in this case.

6.3.1.3 Using System properties

JTAPI logging can also be controlled through system properties. All the properties mentioned in the section 5.3.1.2 for enabling logging can be passed as system properties to your JTAPI application. However these property keys need to be prefixed by com.avaya.jtapi.tsapi.

6.3.1.4 Controlling logging programmatically

It is also possible to programmatically enable logging via the new ITsapiProvider.setDebugPrinting(boolean) method.

If no trace appender exists, calling this method with a parameter value of true shall create a console appender with a log level of TRACE. If a trace appender does exist calling this method with a parameter value of false will change the log level to NONE while a parameter value of true will change the log level to TRACE.. Error logging shall not be impacted by this method invocation.

6.3.2 Server-side debugging

Server side logs are available at /opt/mvap/logs. The configuration file is /opt/mvap/conf/tracemask. A common configuration of tracemask would include

```
TSAPI=0x1
```

Please see the Avaya MultiVantage™ Administration and Maintenance Guide guide to learn more about the server's logs.

6.3.3 Improving performance

Many different factors may potentially affect the performance of your system. A JTAPI system has four main parts that may be affected:

- The AE Services server
- Communication Manager
- The network
- The JTAPI client application

An excessive load on any of these may slow down the overall system. Please check the following.

On the AE Services server:

- Ensure that your AE Services server machine meets the minimum requirements specified in the appropriate Avaya Aura® Application Enablement Services Installation and Upgrade Guide for the offer you have purchased (bundled server or software only).
- Avoid running JTAPI or any other application on the AE Services server machine.
- Check that the AE Services server's Linux operating system resources are tuned correctly for your application needs. The server software makes no assumptions concerning your application needs and therefore does not tune the kernel for you. See the Linux documentation found at http://www.redhat.com/docs/manuals/linux/RHL-9-Manual.
- Update the Linux kernel with the latest updates available.

On the Communication Manager:

Ensure that Avaya Aura® Communication Manager is properly configured for your network and business needs. Misconfigured Avaya Aura® Communication Manager elements can lead to performance issues.

On the network:
Ensure that your network traffic is properly balanced. One way to do this professionally is to ask Avaya to perform a network assessment. There is also a VoIP Readiness Guide available from the Avaya Support Centre (http://www.avaya.com/support). For more information about improving the performance of your network, see the “Network Quality and Management” section of Administration for Network Connectivity for Avaya Avaya Aura® Communication Manager (555-233-504).

On the client
If your application has large memory requirements, consider increasing the memory available to the JVM (using the –Xmx attribute for a Sun JVM).
If many threads are required consider decreasing the default thread stack size (using the –Xss flag on Sun JVMs). Sun's JDK 1.4 allocates about 256K of address space per thread while JDK 1.5, seems to allocate about 1M of address space per thread.

6.3.4 Getting support

Development support is only available through Avaya's DevConnect Program at this time. As an Innovator/Premier/Strategic level member of the DevConnect Program, technical support questions can be answered through the DevConnect Portal at www.avaya.com/devconnect.
As a Registered member of the program, support is not available. If you require support as a Registered member, you can apply for a higher level of membership that offers support and testing opportunities through the DevConnect Portal. Membership at the Innovator/Premier/Strategic level is not open to all companies. Avaya determines membership status above the Registered level.

7. Using the JTAPI Exerciser

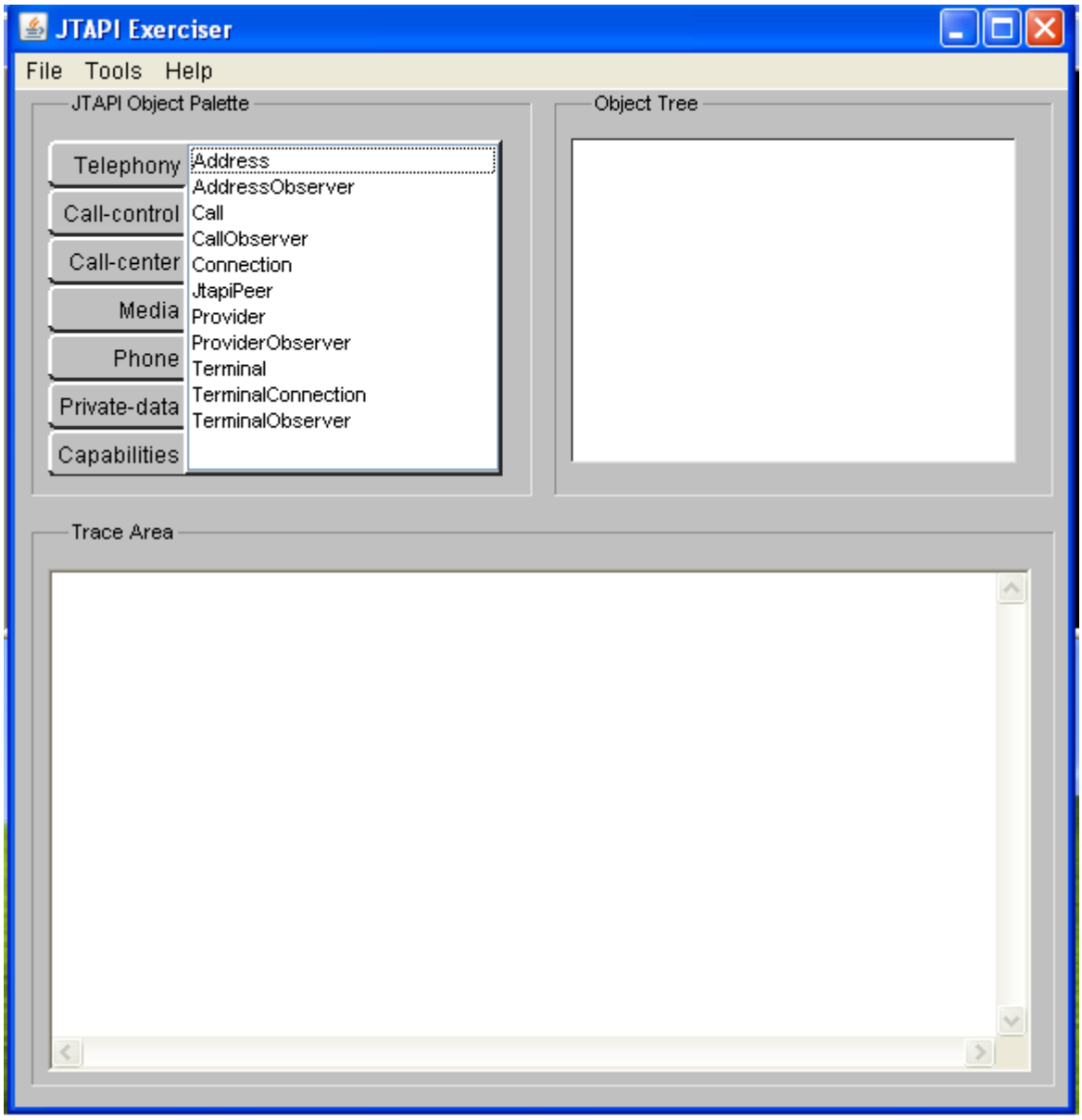
The JTAPI SDK ships with an "Exerciser" tool that is very useful for developers just learning about the API and its capabilities. The Exerciser allows trial of all the capabilities of Avaya's implementation of JTAPI without having to write any code.

This section explains the most common steps that would be performed each time the Exerciser is used. In general, you would do the following

- Create a Provider instance.
- Create one or more address objects that you are interested in monitoring.
- Add a call Observer on the address object to get events informing about the calls on this address.
- Make a call to the above address and check the events.

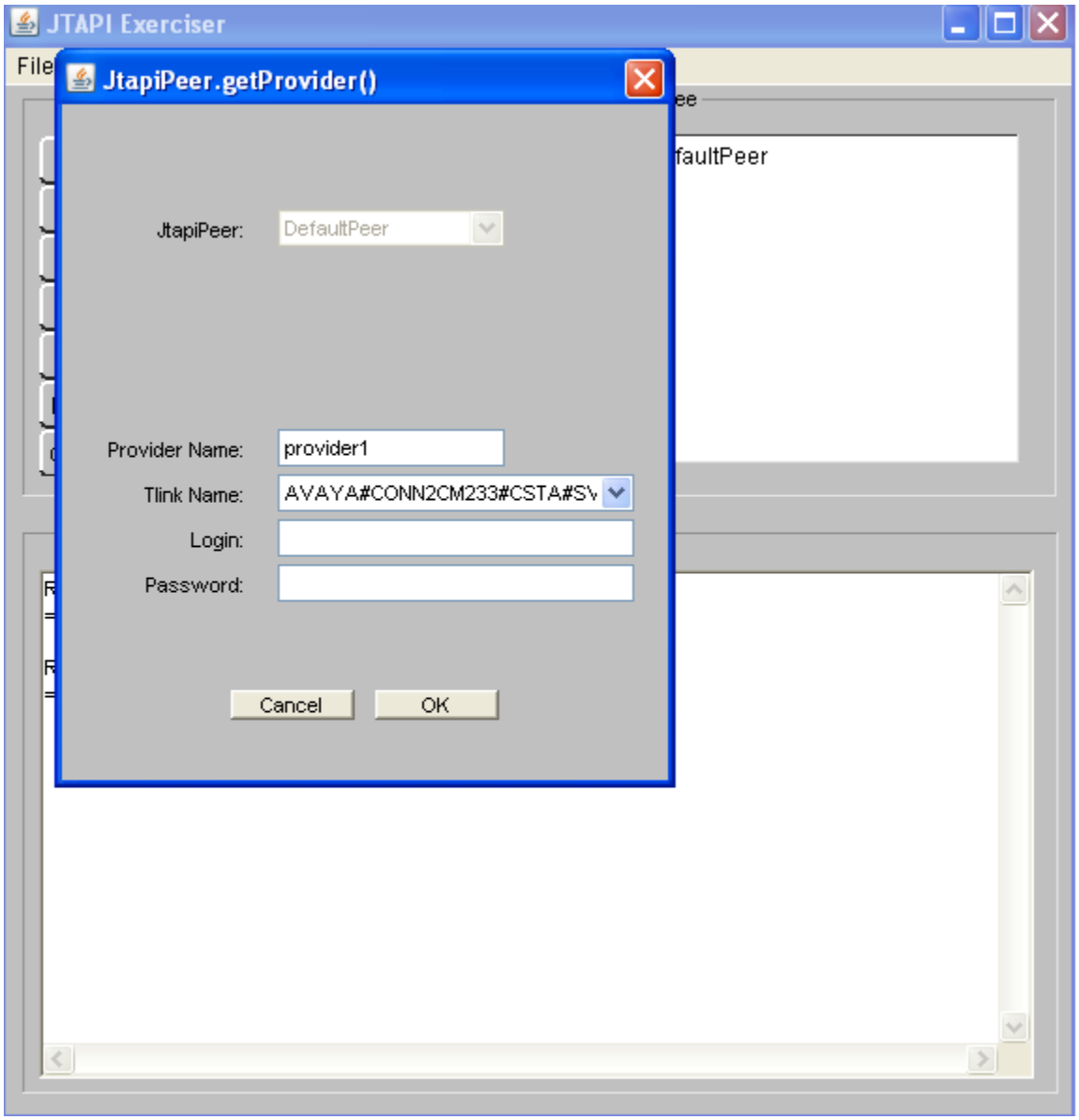
In order to run the Exerciser, you must first ensure that the TSAPI.PRO file in the jtapi-sdk/conf directory is properly configured to point to the AE server. For further details, see "Configuring your JTAPI client library". The next step would be to go to JTAPI_SDK_PATH/tools/bin and run "ant.sh runJtapiExerciser" or "ant.bat runJtapiExerciser" depending on the platform.

Once the Exerciser is launched, the system will display



Your first step will be to acquire an instance of Provider object. You can do this by first acquiring a JtapiPeer, as your application would do, or you can take a short cut and double click directly on the “Provider” label. If you take the latter option, the DefaultJtapiPeer will be used.

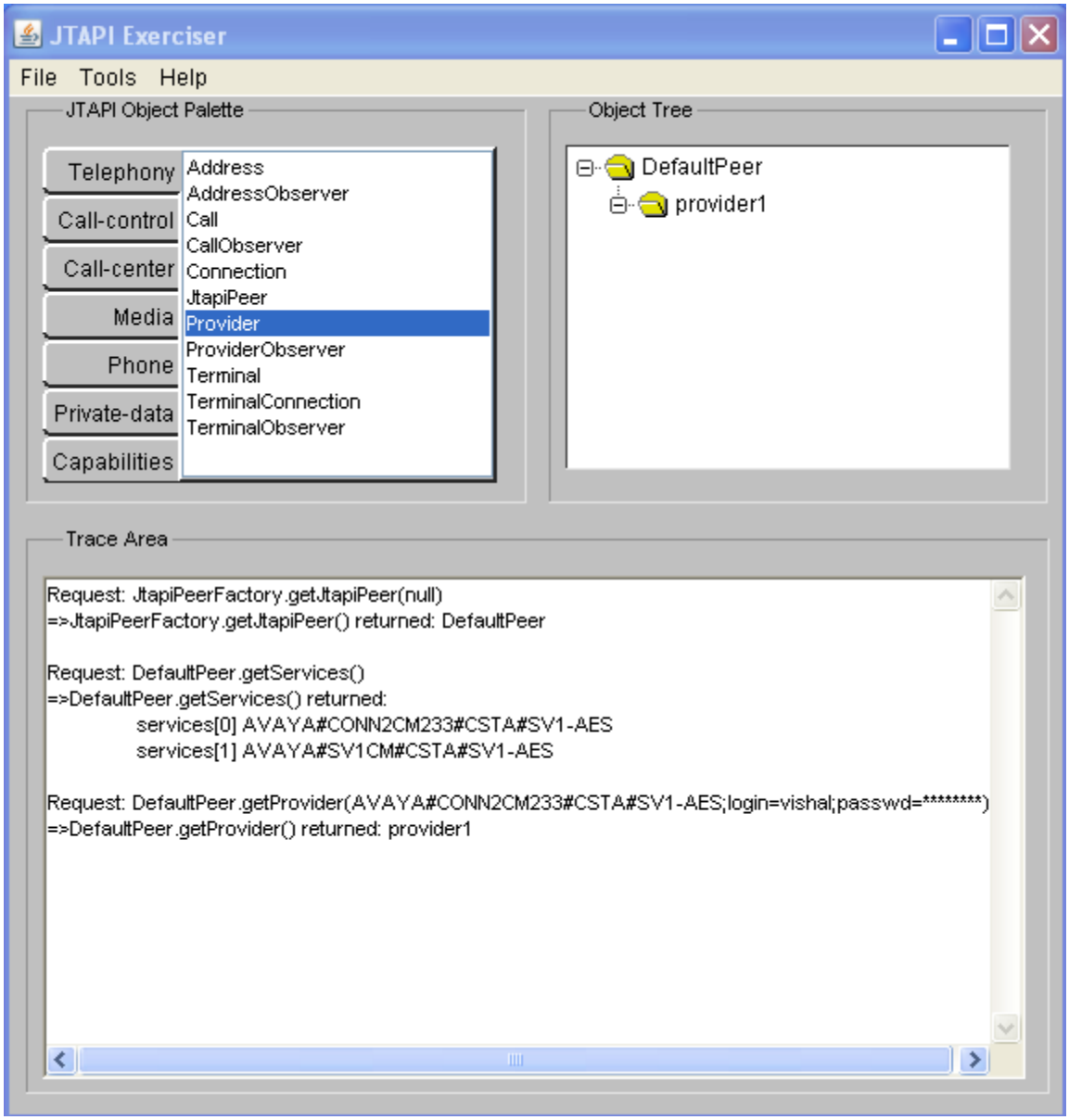
After clicking on provider label directly, the system will display



You will have to enter the following information to be able to create a provider :

- Select the appropriate Tlink from the combo box labeled as “Tlink Name” (see Tlink)
- Enter username of an user in the text box labeled as Login.
- Enter password of the same user in the text box labeled as Password.

Click the “OK” button after entering the above information. Once this is done, check the object tree on the right panel of the Exerciser and it should show a node labeled as “provider1” under a node labeled as DefaultPeer. Please refer to the figure below.

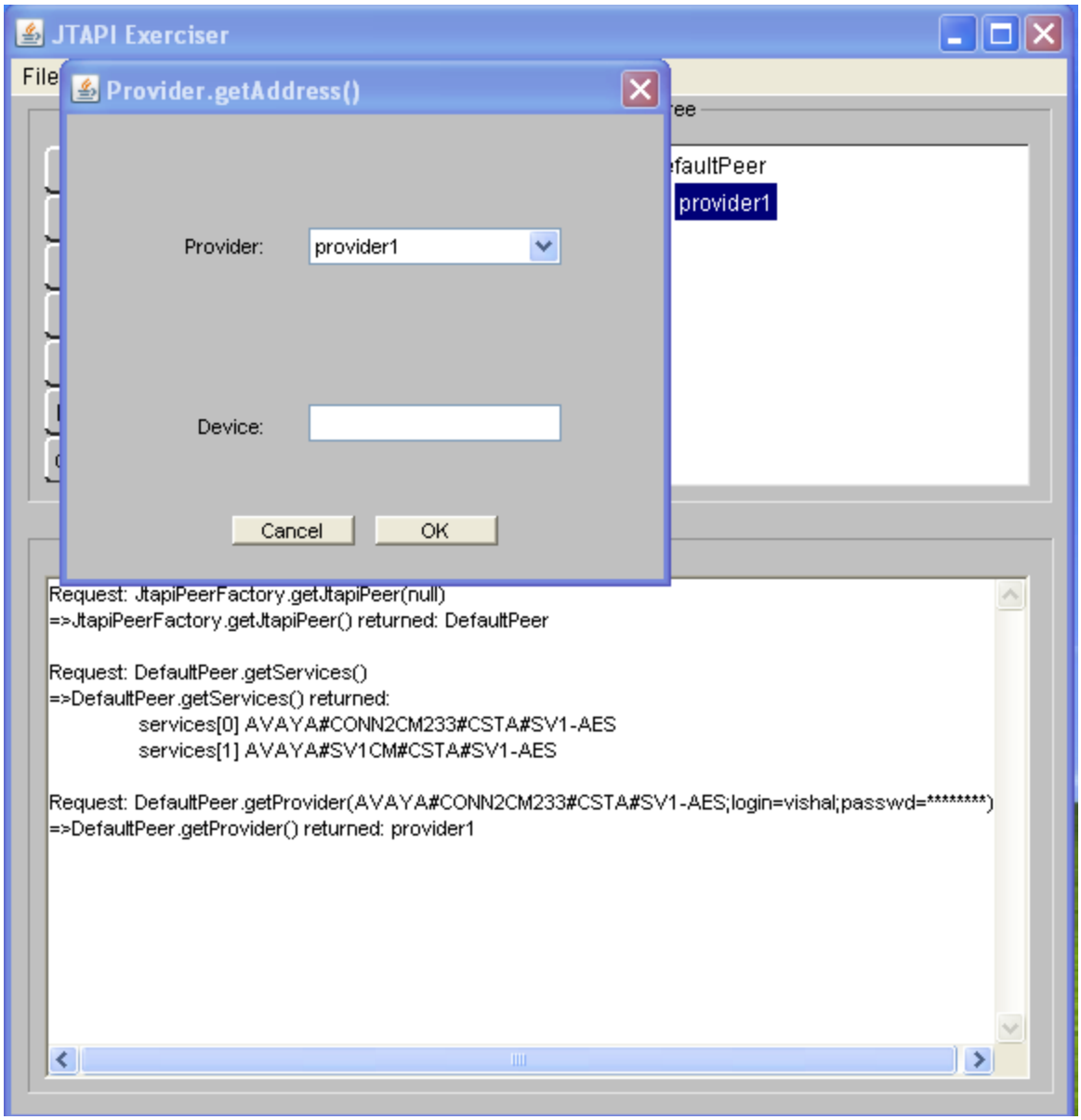


Now that you have a valid provider instance, the next thing is to create an address object.

Let us assume that we want to create an address object for a extension 4701.

Double-click on the “provider1” node in the object tree to launch the provider window and then from this window, you can select the getAddress() function from the ‘Methods’ menu. Or you can take a shortcut by clicking on the address label in the JTAPI Object Palette to enter the extension directly.

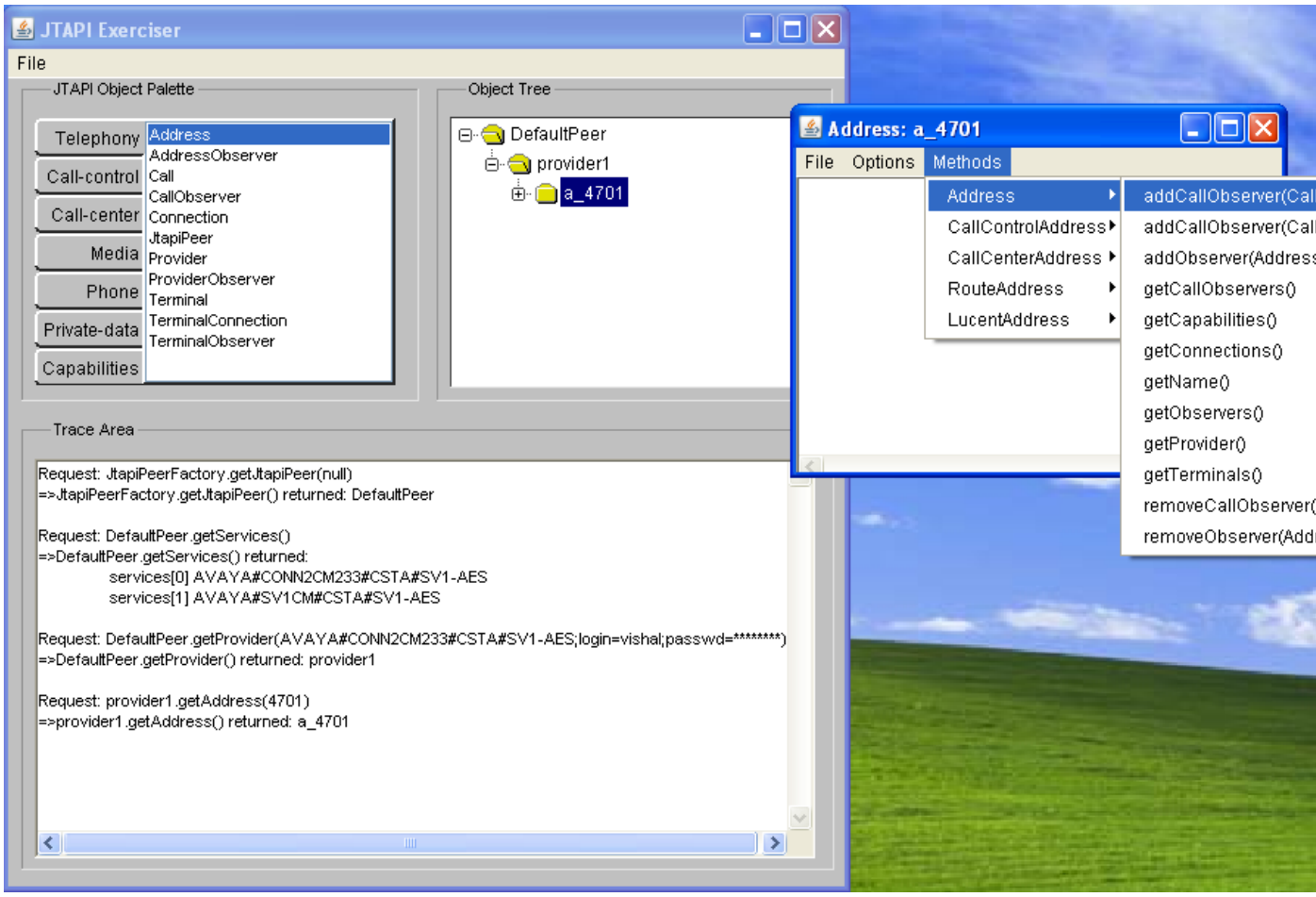
After double-clicking on the address label in the JTAPI Object Palette, the system shows the window as below.



Enter the extension 4701 in the textbox labeled as “Device” and then click “Ok” button. You should now see a node labeled “a_4701” in the JTAPI object tree on the right panel of the Exerciser.

Now we will add a CallObserver on the address object to be able to get information about the calls to this address.

To do this, double-click on the “a_4701” node in the Jtapi object tree. This will launch a address window labeled as “Address:a_4701”. In this address window,click on the menu item labeled as “Methods “. Click on the first Item “Address” and then click the “addCallObserver()” method. Please refer to the figure below.



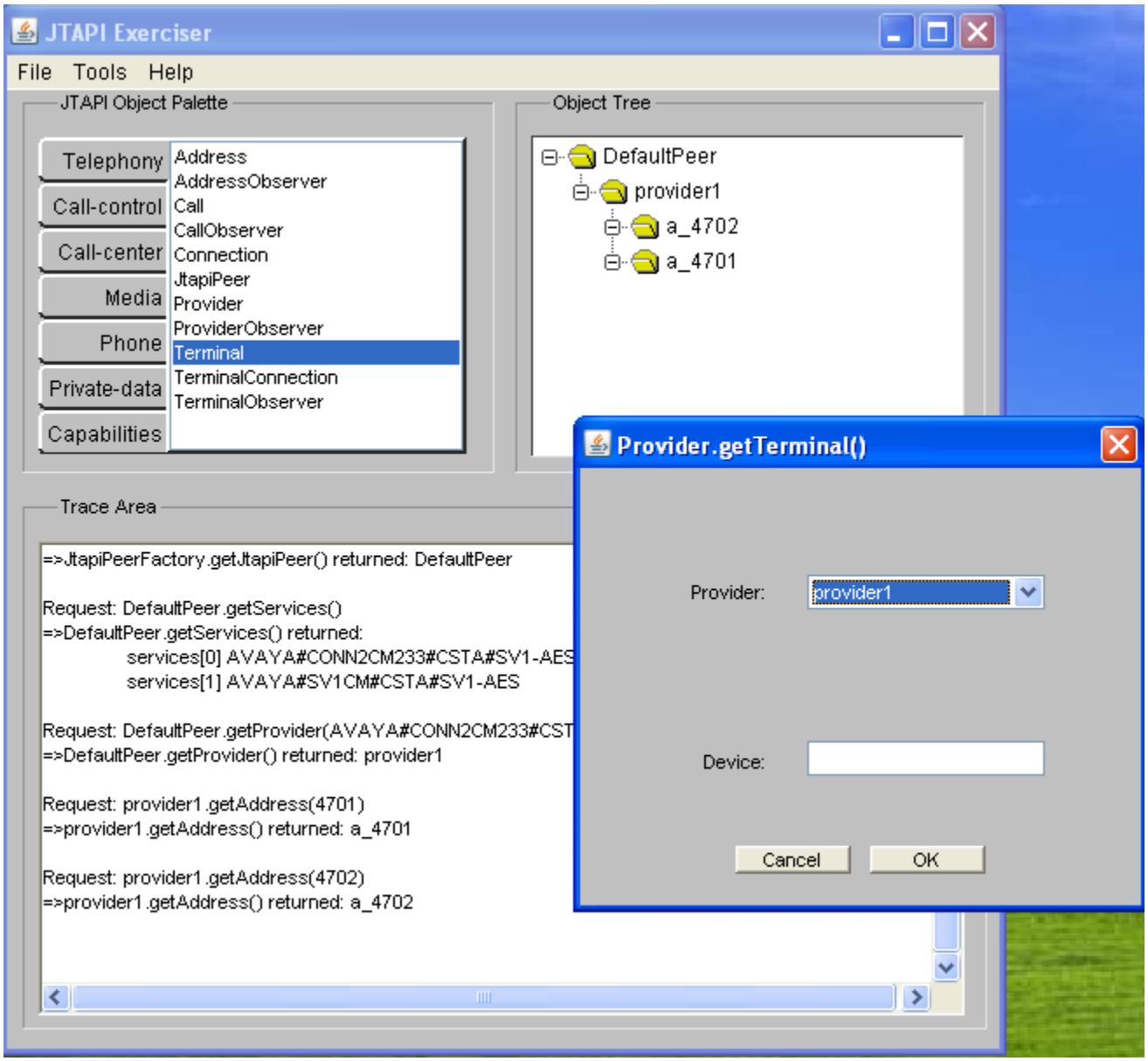
Having done this, you now have a CallObserver added to this address. Any events about the calls to this address will be logged on the address window labeled “Addess:a_4701”.

We now move onto the next step of placing the call to the above address 4701 from an extension, say 4702.

To do this, you need to create an Address and Terminal object for the extension 4702. The process for creating the address object for 4702 is similar to the one followed for creating address object for 4701.

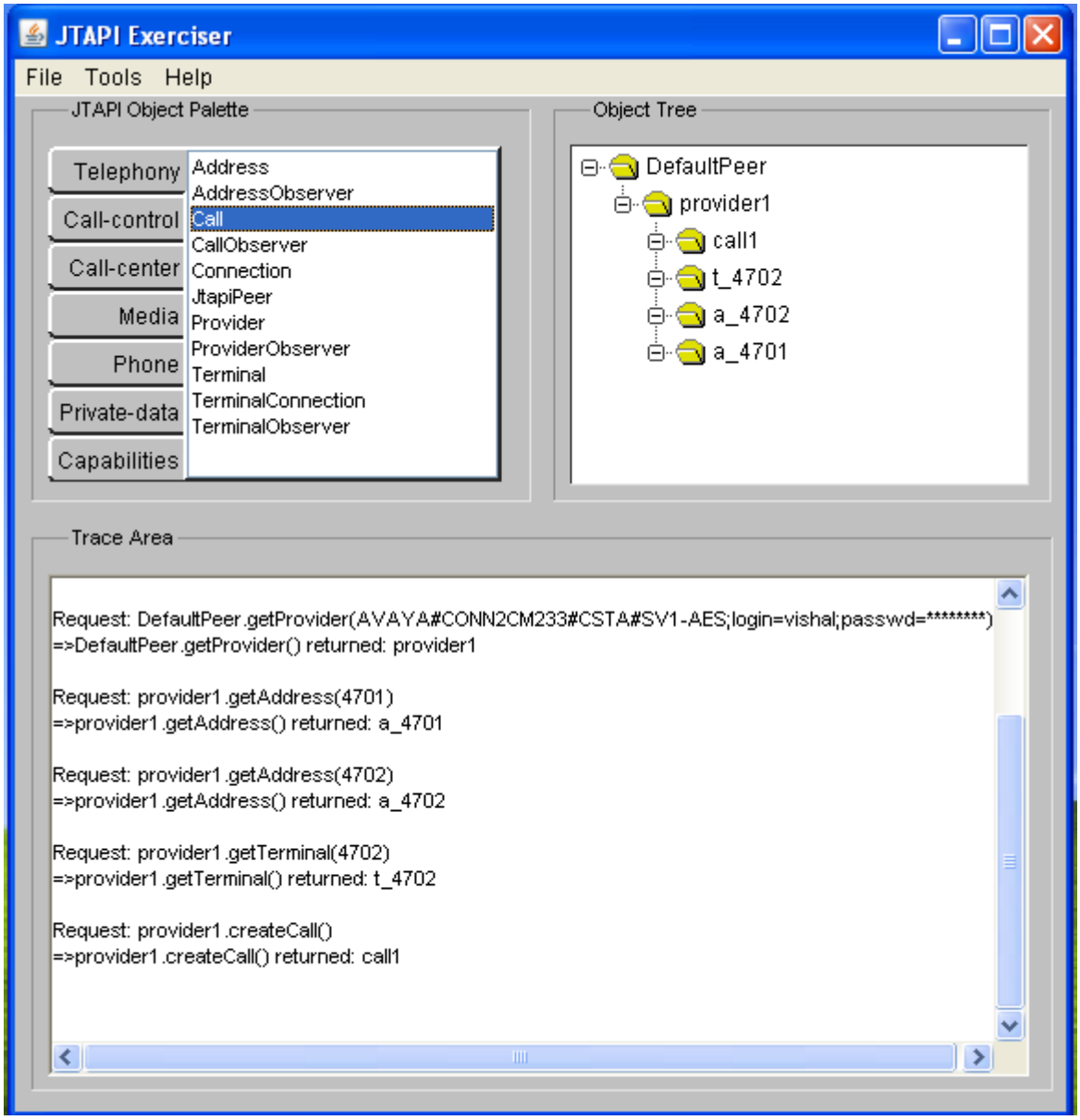
To create a terminal object for 4702, double-click on the label “Terminal” in the JTAPI Object Palette.

This would launch a window as shown below.

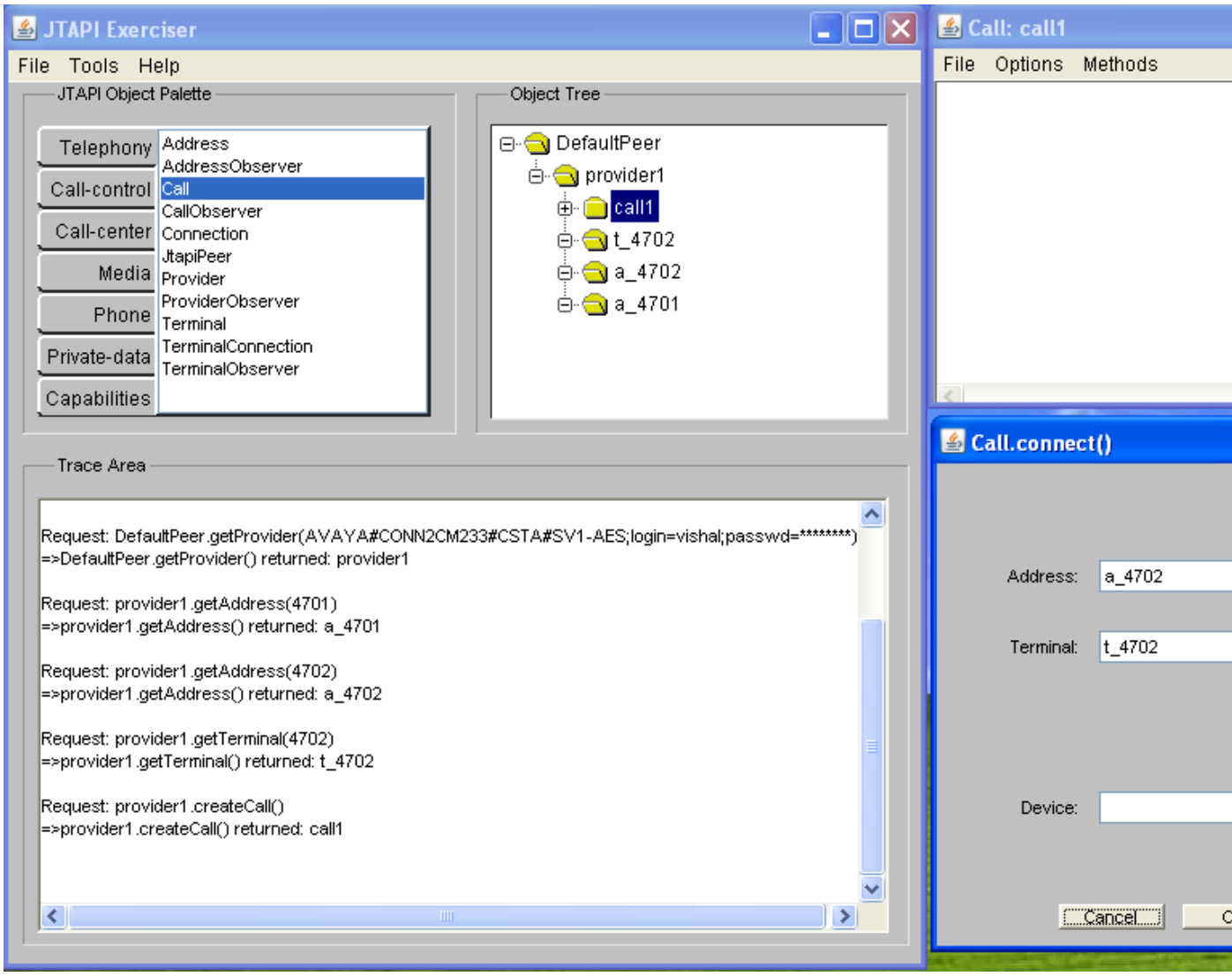


Enter the extension 4702 in the text box labeled as “Device” and click the button “Ok”. This would add a node labeled “t_4702” in the object tree.

The next step is to double-click on the label “Call” in the JTAPI Object Palette. This would launch a window labeled provider.createCall() . Click the “Ok” button on this window. Now you should see a node labeled as “call1” in the JTAPI Object tree. Please refer to the figure below.

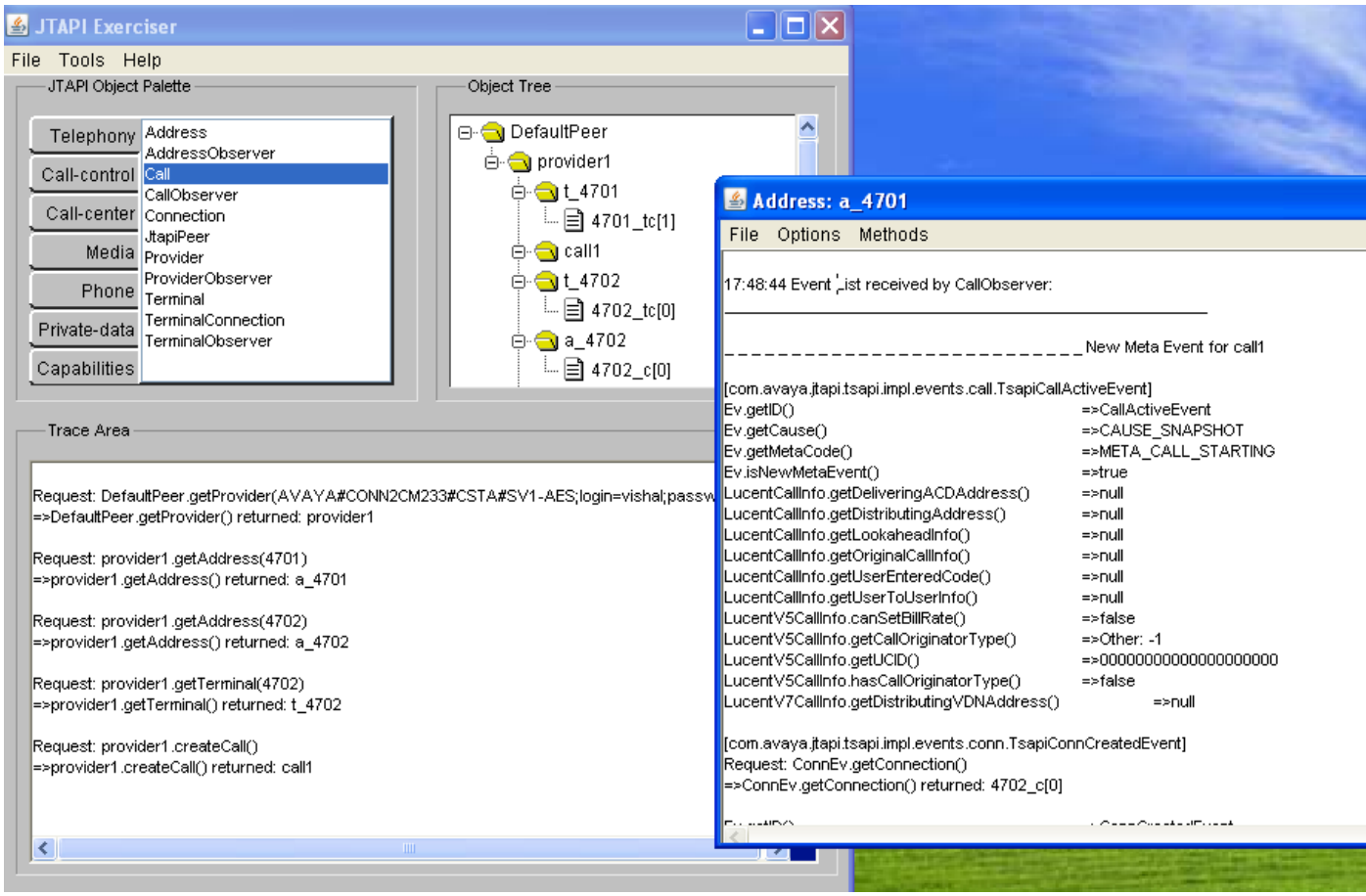


Double-click on the node labeled as “call1” in the object tree to launch a menu labeled as “Call:call1”. In this Menu click on the menu item labeled as “Methods”. From the list that appears, click on the first item labeled as “Call” and then click on the method labeled as “connect(Terminal,Address,String)”. This would open a window labeled as Call.connect() as shown below.



Enter extension 4701 in the textbox labeled as Device and then click “Ok” button.

You should now be able to see JTAPI events in the window labeled Address:a_4701 indicating a call alerting at extension 4701 as shown below. Scroll down the window to look for other JTAPI events pertaining to this call.

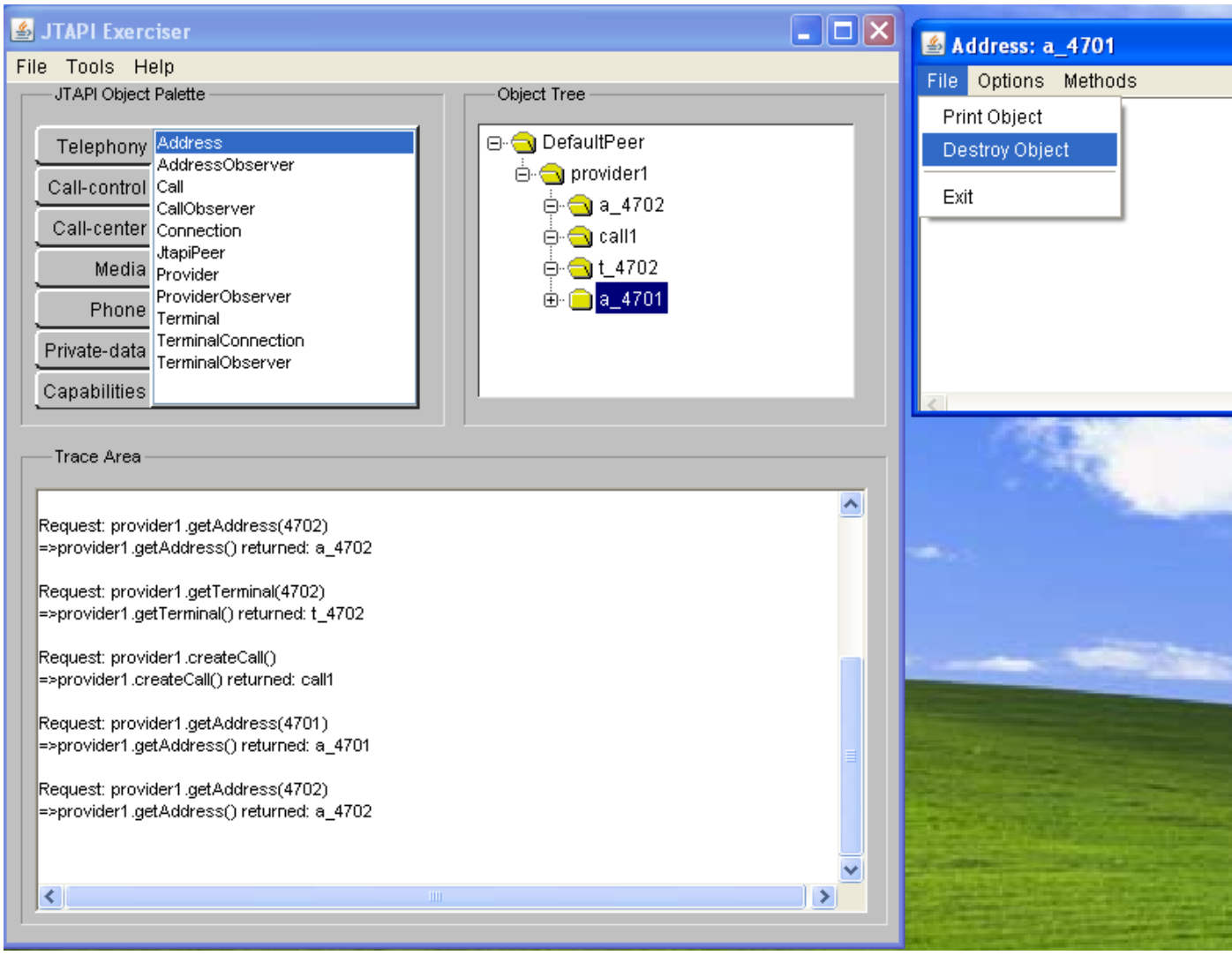


The object tree also gets updated with new nodes representing connections and terminal connections for respective address and terminal objects.

It is now possible to answer the call and do some other API operations like transfer using the various nodes that are available on the object tree.

The next step is to learn about how to clean up various objects that we created in our previous operations.

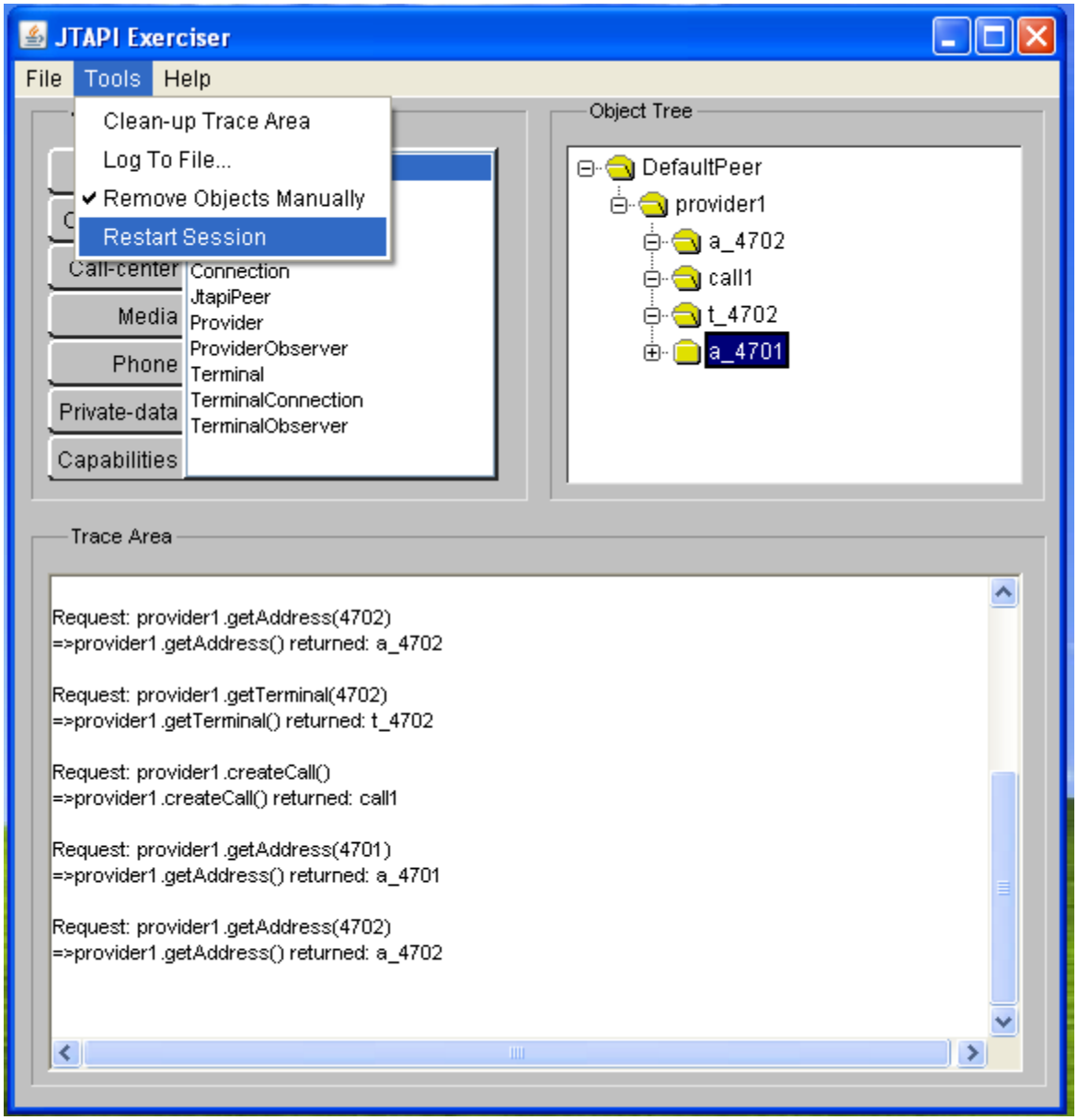
Let us assume we want to clear an object a_4701. To do this, check if you already have the window labeled Address:a_4701 opened. If not ,double-click on the node labeled as “a_4701”. This will launch a window labeled as Addess:a_4701. In this window from the menu bar, select 'File | Destroy Obj. Once this is done, you should no longer see node labeled as a_4701 in the object tree. Please refer to the figure below:



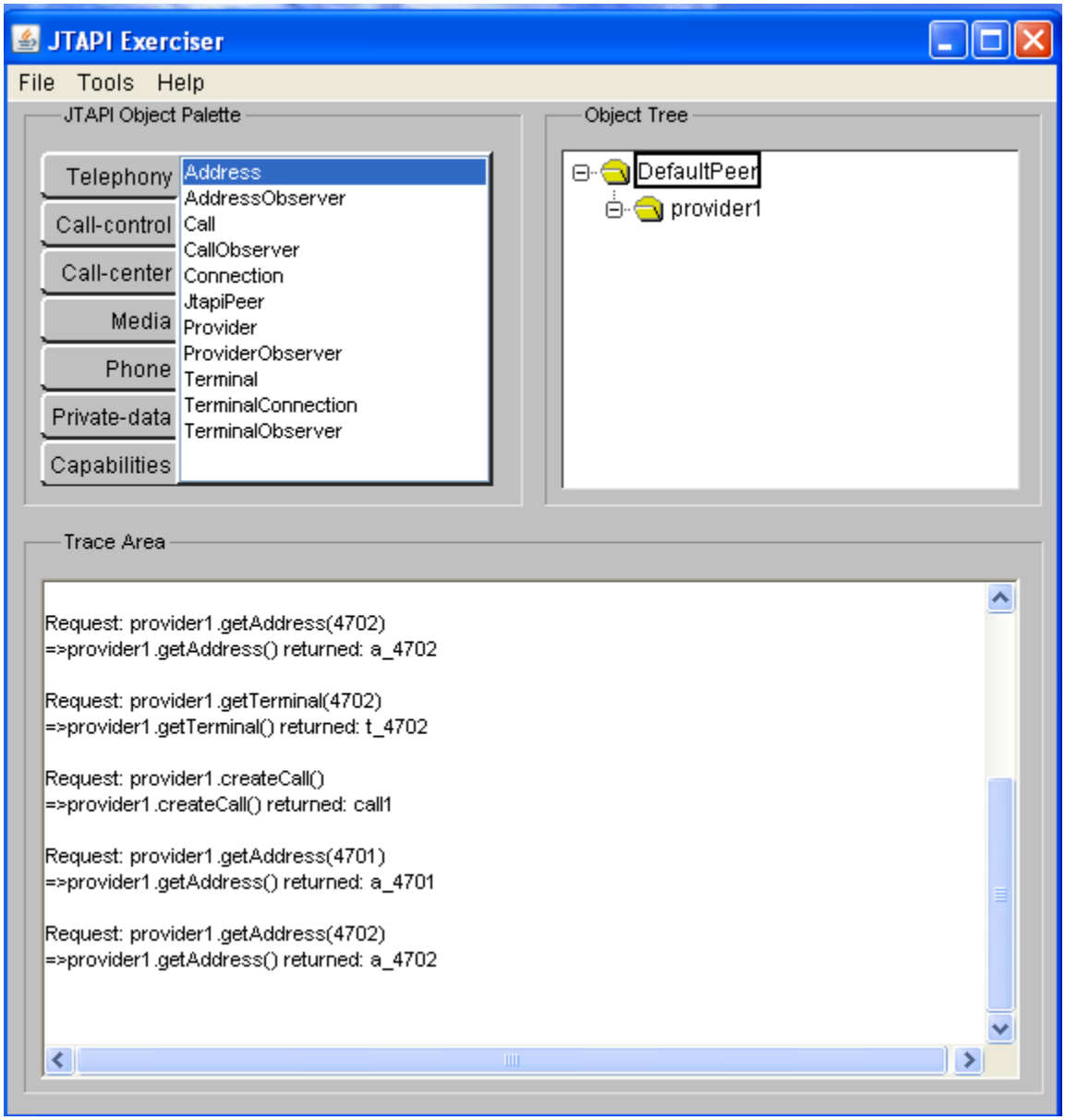
The above process should be repeated for cleaning up other objects in the object tree.

If you want to clear up all the objects in one click, do the following:

From the menu bar select Tools | Restart Session. Please refer to the figure below.



Once this action is completed, there should be no nodes other than the DefaultPeer and the provider node in the object tree. Please refer to the picture below.


JTAPI Exerciser
File
Tools
Help
JTAPI Object Palette
Telephony
Call-control
Call-center
Media
Phone
Private-data
Capabilities
Address
AddressObserver
Call
CallObserver
Connection
JtapiPeer
Provider
ProviderObserver
Terminal
TerminalConnection
TerminalObserver
Object Tree
DefaultPeer
provider1
Trace Area
Request: provider1.getAddress(4702)
=>provider1.getAddress() returned: a_4702
Request: provider1.getTerminal(4702)
=>provider1.getTerminal() returned: t_4702
Request: provider1.createCall()
=>provider1.createCall() returned: call1
Request: provider1.getAddress(4701)
=>provider1.getAddress() returned: a_4701
Request: provider1.getAddress(4702)
=>provider1.getAddress() returned: a_4702

APPENDIX A – Avaya implementation specific deviations from the JTAPI specification

Core package implementation details

The following table describes the Core Package interfaces and methods..

Interface	Method	Implementation Notes
Address	getTerminals	The implementation of this request relies on the AE Services Security Database (SDB). If the SDB is not enabled, NULL will be returned for address.getTerminals() and terminal.getAddresses(). Without the SDB, there is no listing of addresses and no information to pass.
Connection	disconnect	Must be called with Connection in the CONNECTED, INPROGRESS, ALERTING, FAILED or UNKNOWN state. If the connection is not in either of these states then a InvalidStateException is thrown.
JtapiPeer	Not applicable	Obtain a JtapiPeer object using the JtapiPeerFactory class. The TsapiPeer class represents this implementation of the JtapiPeer. To obtain TsapiPeer, invoke JtapiPeerFactory.getJtapiPeer("com.avaya.jtapi.tsapi.TsapiPeer")
JtapiPeer	getServices	Returns an array of service names that can be used to build the String needed to be passed to JtapiPeer.getProvider(). These Strings are the AE Server Tlink names.
JtapiPeer	getProvider	The providerString parameter to this method must contain an AE Services Tlink name as well as login and password for user authentication. Optionally, the AE server to connect to can also be specified in this string as the value of the parameter servers.. The format of the String is "<tlink>;login=<loginID>;passwd=<pw>;servers=<server entries>" Where server entries follows the format server1:port,server2:port,server3:port
Terminal	getAddresses	The implementation of this request relies on the AE Services Security Database (SDB). If the SDB is not enabled, NULL will be returned for address.getTerminals() and terminal.getAddresses(). Without the SDB, there is no listing of addresses and no information to pass.

Call Center package implementation details

The following table describes the Call Center Package interfaces and methods.

Interface	Method	Implementation Notes
ACDAddress	getOldestCallQueued	Method not supported.
ACDAddress	getRelativeQueueLoad	Method not supported.
ACDAddress	getQueueWaitTime	Method not supported.
ACDAddress	getACDManagerAddress	Method not supported.
ACDManagerAddress	getACDAddresses	Method not supported.
Agent	getAgentID	Returns a null string.
AgentTerminalObserver	No methods defined.	The AgentTerminalObserver only supports the AgentTermLoggedOnEv and AgentTermLoggedOffEv when the state change is produced through the JTAPI application. In order to monitor agent activity (e.g., agents logging on and off manually), an ACDAddressObserver should be added to the ACDAddress. Similarly in case of listeners, AgentTerminalListener# agentTerminalLoggedOn and AgentTerminalListener# agentTerminalLoggedOff are supported only when the JTAPI application itself is logging the agent on and off. To completely monitor agent activity, please use an ACDAddressListener
CallCenterCall	connectPredictive	The answeringEndpointType parameter is not supported. The maxRings and answeringTreatment parameters are supported. If the Call is observed and the ACDAddress or AgentTerminal is also call observed, then two unique Call objects will be created that are associated with the same real call. One of the following methods must be used to determine that there are two Call objects representing the same real call. • If the called address is unique among all calls, the Call.getCalledAddress() method can be used. • Another way is to use the UserToUserInfo Avaya Aura Software server-specific extension. The application can send a unique ID in the UserToUserInfo with the connectPredictive and this ID will be reported in call events for the

		ACDAddress or AgentTerminal. The UserToUserInfo can also be retrieved directly from the Calls. In any case, both Call objects and all Connections and TerminalConnections in both Calls are valid. Valid requests may be made of any of the objects. Currently, only Connection.CONNECTED is valid as the connectionState parameter. If Connection.ALERTING is specified, it is ignored and Connection.CONNECTED is used.
CallCenterCall	getApplicationData	Returns the application-specific data associated with the Call. This method returns null is there is no associated data.
CallCenterCall	setApplicationData	This method associates application specific data with a Call. The format of the data is application-specific. The application-specific data given in this method replaces any existing application data. If the argument given is null, the current application data (if any) is removed. In the case that a Call is transferred or conferenced, the application data from the Call from which the conference or transfer is invoked will be retained.
PrivateTermEv	getPrivateData	Method not supported.
RouteAddress	registerRouteCallback	Only one RouteCallback may be registered for an Address at a time.
RouteSession	selectRoute	Only the first route specified in the routeSelected parameter is used. The subsequent routes are ignored.
RouteSession	getRouteAddress	The RouteAddress returned by RouteSession.getRouteAddress() is the originally called device if there is no distributing device (ACD or VDN), or the distributing device if the call vectoring with VDN override feature of the PBX is enabled.
AgentTerminal	setAgents	Method not supported.

Call Center Events package implementation details

All events in the Call Center Events package are only sent to the application when a state change results from an application request. If the state change occurs via some other interface (e.g. the agent pushes a button on their telephone), no event will be sent to the application.

Similarly, all callbacks in the CallCenterCallListener, ACDAddressListener and AgentTerminalListener are sent only when the state change is actively made by the JTAPI application.

Call Control package implementation details

The following table describes the Call Control Package interfaces and methods.

Interface	Method	Implementation notes
CallControlAddress	setForwarding	Avaya supports the FORWARD_UNCONDITIONALLY forwarding type only when used in combination with the ALL_CALLS filter type.
CallControlAddress	getDoNotDisturb	For this method, there is no distinction between an Address and a Terminal. CallControlAddress.getDoNotDisturb() and CallControlTerminal.getDoNotDisturb() always return equivalent values.
CallControlAddress	setDoNotDisturb	For this method, there is no distinction between an Address and a Terminal. CallControlAddress.setDoNotDisturb() and CallControlTerminal.setDoNotDisturb() behave the same.
CallControlCall	offHook	Method not supported.
CallControlCall	transfer(String address)	This method is supported with the following implementation-specific details: • the application must call setTransferController() • transfer(String) returns a connection in UNKNOWN state but followup events provide state updates
CallControlCall	consult(TerminalConnection termconn)	Method not supported.

CallControlConnection	accept	Method not supported.
CallControlConnection	reject	Method not supported.
CallControlConnection	addToAddress	Method not supported.
CallControlConnection	park	Method not supported.
CallControlTerminal	getDoNotDisturb	For this method, there is no distinction between an Address and a Terminal. CallControlAddress.getDoNotDisturb() and CallControlTerminal.getDoNotDisturb() always return equivalent values.
CallControlTerminal	setDoNotDisturb	For this method, there is no distinction between an Address and a Terminal. CallControlAddress.setDoNotDisturb() and CallControlTerminal.setDoNotDisturb() behave the same.
CallControlTerminal	pickupFromGroup (String pickupGroup, Address terminalAddress)	Method not supported.
CallControlTerminalConnection	join	Method not supported.
CallControlTerminalConnection	leave	Method not supported.
CallControlAddressListener	addressDoNotDisturb	Similar to the equivalent observer behavior described above, this callback is invoked even if DoNotDisturb was changed using CallControlTerminal.setDoNotDisturb(). For DoNotDisturb, there is no distinction between an Address and a Terminal.
CallControlConnectionListener	connectionDialing	Callback not supported
CallControlConnectionListener	connectionOffered	Callback not supported
CallControlTerminalConnectionListener	terminalConnectionInUse	Callback not supported

Call Control Events package implementation details

The following table describes the Call Control Events Package interfaces and methods..

Interface	Method	Implementation notes
CallCtlAddrDoNotDisturbEv	getDoNotDisturbState	The CallCtrlAddrDoNotDisturbEv event is sent even if DoNotDisturb was changed using CallControlTerminal.setDoNotDisturb(). For DoNotDisturb, there is no distinction between an Address and a Terminal.
CallCtlConnDialingEv	Not applicable	Interface not supported.
CallCtlConnOfferedEv	Not applicable	Interface not supported.
CallCtlTermConnInUseEv	Not applicable	Interface not supported.

Media package implementation details

This package is an optional part of the JTAPI specification. Avaya supports only DTMF related functionality in this package. MediaTerminalConnection.generateDtmf() is supported to send DTMF tones and MediaTermConnDtmfEv from the events package is supported to enable an application to be notified of the DTMF digits dialed. MediaCallObserver is supported to the extent that an observer implementing this interface is required to be used in order to be notified of a MediaTermConnDtmfEv event.

Media Events package implementation details

As mentioned above, MediaTermConnDtmfEv is the only media event that is supported.

Although the MediaTermConnDtmfEv interface has been defined as a TerminalConnection event, the TerminalConnection field will be null. The Call field will be filled in with the call to which the DTMF digits have been applied. This event is sent only when a DTMF detector is attached to the call and DTMF tones are detected. The tone detector is disconnected when the far end answers or "#" is detected. This event is used in conjunction with the Communication Manager-specific extension LucentRouteSession.selectRouteAndCollect().

In case a listener (more specifically a PrivateDataCallListener) is used, the DTMF digits applied will be passed as private data via the PrivateDataCallListener.callPrivateData callback. Invoking getPrivateData() on the parameter passed to this callback will return an instance of com.avaya.jtapi.tsapi.PrivateDtmfEvent, which will contain the dialed DTMF digits

Private Data package implementation details

The following table describes the JTAPI Private Data package interfaces and methods..

Interface	Method	Implementation notes
PrivateData	setPrivateData	For this method, the private data Object parameter must be an instance of TsapiPrivate.
PrivateData	sendPrivateData	For this method, the private data Object parameter must be an instance of TsapiPrivate.

Phone package implementation details

No class / interface in this package is supported.

Mobile package implementation details

No class / interface in this package is supported.

APPENDIX B – Avaya implementation specific enhancements to the JTAPI specification

Extensions to JTAPI Exceptions

AE Services extensions to the JTAPI exceptions provide more detailed error information than is defined in JTAPI. These extensions consist of the CSTA and ACS error codes provided by TSAPI.

For information about Computer-Supported Telecommunications Applications (CSTA) and API Control Services (ACS) error codes, refer to *Avaya MultiVantage Application Enablement Services TSAPI Programmer's Reference,* 02-300545.

ACS error codes are defined in the enum ACSUniversalFailure_t, while CSTA error codes are defined in the enum CSTAUniversalFailure_t.

The javadoc of com.avaya.jtapi.tsapi.ITsapiCSTAUniversalFailure also contains a list of CSTA failure codes.

Extension to CallCenterAddress interface

Feature Name and Description	Class or Interface	Returned/Used by Methods in Class or Interface
Call listener to monitor calls for the lifetime of the call.	Not applicable	LucentCallCenterAddress

Extensions to JTAPI Provider events

AE Services defines additional JTAPI Provider events. These events provide more detailed Provider state changes. These TSAPI Provider states map to JTAPI Provider states as follows:

TSAPI Provider State	JTAPI Provider State
ITsapiProvider.TSAPI_OUT_OF_SERVICE	Provider.OUT_OF_SERVICE
ITsapiProvider.TSAPI_INITIALIZING	Provider.OUT_OF_SERVICE
ITsapiProvider.TSAPI_IN_SERVICE	Provider.IN_SERVICE
ITsapiProvider.TSAPI_SHUTDOWN	Provider.SHUTDOWN

Avaya Aura® Communication Manager Extensions to JTAPI

This table summarizes the Avaya Aura® Communication Manager features that are available as extensions to JTAPI.

Feature Name and Description	Class or Interface	Returned/Used by Methods in Class or Interface
Advice of Charge - Reports network charges incurred by outgoing trunk calls.	LucentChargeAdviceEvent LucentChargeError LucentChargeType	LucentV5Provider
Agent Work Mode - Specifies the overriding mode of the Agent; affects the cycle of the possibly occurring Agents states. LucentV6 Agent adds support for: reason code(an application-defined reasonCode (1-9), which may be specified when the state is set to Agent.NOT_READY.) and pending Work Modes(A JTAPI Application may request to change an Agent's state to Agent.WORK_NOT_READY and Agent.NOT.READY, and to have the state change be held "pending" until all current calls that are active on the Agent's Agent Terminal are completed). LucentV7Agent adds support for expanded reason codes. (1-99),	Not applicable	LucentAgent LucentAgentStateInfo LucentV5AgentStateInfo LucentTerminal LucentV5Terminal LucentV5TerminalEx LucentV5AgentStateInfo LucentV6Agent LucentV6AgentStateInfo LucentV7Agent
Call Classifier Information - Provides information on call classifier port usage (namely available and in-use ports)	CallClassifierInfo	LucentProvider
Collect Digits - Allows a route request to wait for a specified number of digits to be collected. This feature is not supported currently. Invocation of the following	Not applicable	LucentRouteSession

Feature Name and Description	Class or Interface	Returned/Used by Methods in Class or Interface
method • selectRouteAndCollect on LucentRouteSession will throw a TsapiMethodNotSupportedException.		
Dial-Ahead Digits - Allows a route request to place digits in a dial-ahead buffer. This feature is not supported currently. Invocation of the following method • selectRouteWithDigits on LucentRouteSession will throw a TsapiMethodNotSupportedException.	Not applicable	LucentRouteSession
Direct Agent Calls - Allows calls to be made to and from specific logged-in ACD Agents Allows calls to be made to and from specific logged-in ACD Agents	Not applicable	LucentCall LucentRouteSession
Dropping Resources - Allows specific switch resources to be dropped from the call.	Not applicable	LucentConnection LucentTerminalConnection
Flexible Billing - Allows changing the billing rate for incoming 900-type calls.	Not applicable	LucentV5Call LucentBillType
Flexible Generation of DTMF Tones - Enables an application to specify tone duration and inter-tone delay duration.	Not applicable	LucentV5TerminalConnectionEx
Integrated Directory Name - Allows the Avaya Aura® Communication Manager Integrated Directory Database name to be returned	Not applicable	LucentAddress LucentTerminal
Device On Switch Allows applications to determine if an Address/Terminal object represents a	Not Applicable	LucentAddress LucentTerminal

Feature Name and Description	Class or Interface	Returned/Used by Methods in Class or Interface
station administered on the switch		
Predictive call observation - Allows the application to receive notice of all call events for the predictive dial call.	Not applicable	LucentV7ACDManagerAddress
Look-Ahead Interflow Information - Can be used by a routing server application to determine the proper destination of a call.	LookaheadInfo	LucentCallInfo OriginalCallInfo
Extended AgentTerminal connection information - Provides information regarding 1. The ACDAddress or ACDManagerAddress that was an intermediate endpoint before the call terminated at the AgentTerminal. 2. The ACDAddress that this call was delivered through to the AgentTerminal.	Not applicable	LucentCallInfo
Lucent Call Information - Provides Avaya Aura Communication Manager-specific call information on Call and CallControlCall events; information includes delivering ACD, distributing Address, originating Trunk, reason for last Call event, and other information.	LucentCallInfo	Implemented by Lucent call objects, route session objects, and CallControlCall events.LucentCallInfo (extended by LucentCall; extended by LucentV5CallInfo; extended by CallControlCall events)
LucentV5 Call Information – In addition to the Lucent call information , LucentV5 call information adds support for : Universal Call ID, Originator Type, and Flex Billing Flag.	LucentV5CallInfo	Implemented by Lucent call objects, route session objects, and CallControlCall events.LucentV5CallInfo (extended by LucentV5Call; extended by LucentV7CallInfo; extended by CallControlCall events)

Feature Name and Description	Class or Interface	Returned/Used by Methods in Class or Interface
LucentV7 Call Information – In addition to the LucentV5 call information, LucentV7 call information adds support for retrieving the current list of device history entries for this call and the distributing VDN (if defined)	LucentV7CallInfo	Implemented by Lucent call objects, route session objects, and CallControlCall events.LucentV7CallInfo (extended by LucentV7Call; extended by CallControlCall events)
Message Waiting Application Information - Indicates which types of applications have enabled message waiting	Not applicable	LucentAddress LucentAddressMsgWaitingEvent LucentCallControlAddressMsgWaitingEvent
Network Progress Information - Contains supplementary call progress information from the ISDN Progress Indicator Information Element. V5NetworkProgressInfo adds support for: trunk.	NetworkProgressInfo V5NetworkProgressInfo	LucentConnNetworkReachedEvent
Original Call Information - Contains information about the original call in conjunction with the Call.consult() service.	OriginalCallInfo	LucentCallInfo
LucentV5 Original Call Information – In addition to the Lucent Original Call information, LucentV5 Original Call Information adds support for: Universal Call ID, Originator Type, and Flex Billing Flag.	V5OriginalCallInfo	LucentV5CallInfo
LucentV7 Original Call Information - In addition to the LucentV5 Original Call information, LucentV7 Original Call information adds support for: device history.	V7OriginalCallInfo	LucentV7CallInfo
Priority Calls – Extends the equivalent standard API's to enable priority calling	Not applicable	LucentCall (connect, predictive calling and consult API's)

Feature Name and Description	Class or Interface	Returned/Used by Methods in Class or Interface
		LucentRouteSession (route selection API)
Selective Listen – Allows control of listen paths between parties on a conference call.	Not applicable	LucentV5Connection LucentV5TerminalConnection
Single Step Conference – Adds another party to a call (added party does not alert; used mainly for service observing).	Not applicable	LucentV5Call
Supervisor Assist Calls – Allows logged-in ACD Agents to place calls to a supervisor's extension.	Not applicable	LucentCall
Direct Agent Consultation Calls – Allows logged-in ACD agents to place consult calls to other agents.	Not applicable	LucentCall
Supervisor Assist Consultation Calls – Allows logged-in ACD agents to place consult calls to a supervisor.	Not applicable	LucentCall
Fast Connect – Similar to the standard Call.connect() except that this API only waits for the connection for the calling party to be created before returning. This method is useful when sending FACs(Feature Access Codes) such as TAC (Trunk Access Code) codes.	Not applicable	LucentCallEx2
Switch Date and Time Information - Returns the current date and time from Communication Manager.	Not applicable	LucentProvider

Feature Name and Description	Class or Interface	Returned/Used by Methods in Class or Interface
Trunk Group Information - Provides information on trunk group usage. Trunk associates group and member information with a connection. If a connection is associated with a trunk party, then the application can get trunk group number and trunk group member information.	TrunkGroupInfo	LucentProvider LucentV6Connection LucentTrunk ITsapiTrunk
Universal Call ID - A call identifier that is globally unique across switches and the network.	Not applicable	LucentV5CallInfo (extended by LucentV5Call)
User Entered Code - The code/digits that may have been entered by the caller through the Avaya Aura Communication Manager Call Prompting feature of the Collected Digits feature.	UserEnteredCode	LucentCallInfo OriginalCallInfo
User-to-User Information - An ISDN feature that allows end-to-end transmission of application data during call setup/teardown. UUI can be specified, and will be made available, accommodating string values up to 96 bytes.	UserToUserInfo	LucentCall LucentCallInfo LucentConnection LucentRouteSession LucentTerminalConnection LucentCallInfo
Network Call Redirection - The Adjunct Route support for Network Call Redirection capability allows an adjunct to request that an incoming trunk call be rerouted using the Network Call Redirection feature supported by the serving PSTN instead of having the call routed via a tandem trunk configuration. The LucentV7RouteSession interface extends LucentRouteSession to add the ability to use the Network Call Redirection feature of call routing on	Not applicable	LucentV7CallInfo LucentV7RouteSession

Feature Name and Description	Class or Interface	Returned/Used by Methods in Class or Interface
Avaya switches. When a Provider is bound to a Avaya Aura® Communication Manager switch, this interface may be used to access this additional capability. The route session object which implements this interface also implements the ECSCallInfo interface.		
ISDN Redirecting Number (Redirecting Number Information Element presented through DeviceHistory) - The ISDN Redirecting Number for ASAI Events. Avaya Aura® Communication Manager feature may be used by CTI applications to provide enhanced treatment of incoming ISDN calls routed over an Integrated Services Digital Network (ISDN) facility. **Device History Entry -** The V7DeviceHistoryEntry is an entry that represents a connection that was formerly on a call. This provides equivalent content to the Avaya TSAPI service implementation of CSTA3 DeviceHistory parameter (see ECMA-269 Edition 5, "12.2.13 DeviceHistory"). Note that private interfaces are defined to enable an application to use the TSAPI information (specifically the ConnectionID).	V7DeviceHistory V7OriginalCallInfo	LucentV7CallInfo LucentV7RouteSession
Query Device Name - The private Query Device Name service allows an application to query the switch to identify the Integrated Directory name assigned to an extension. When a name has been assigned to an Attendant station extension, then an application can use the getDirectoryName method of the LucentAddress interface to get the configured Integrated Directory name assigned to that attendant extension.	Not applicable	LucentAddress

Feature Name and Description	Class or Interface	Returned/Used by Methods in Class or Interface
Enhanced Get API Capabilities function - The GetAPICaps function is enhanced to return the following information. • Administered Switch Version (as administered in the system parameters customer options form on the switch) • Software Version (the same software version string that is shown when a customer logs into a SAT for a switch) • Offer Type (values to be added in future releases of TSAPI Service).Valid values include: s8300, s8400,s8500 and s8700.. • Server Type (more values to be added in future releases). Valid values include: s8300c,s8300d,icc,premio,tn8400, laptop,ibmx306, ibmx306m,dell1950, xen,hs20,hs20_8832_vm,isp2100, ibmx305,dl380g3,dl385g1, dl385g2 and unknown.,	Not applicable	LucentV7Provider
Expanded universal failure error codes - The list of universal failure codes that can be returned in CSTA UniversalFailure unsolicited events and confirmation events. This is useful, for example, for JTAPI exceptions thrown by the Avaya implementation which returns these values.	Not applicable	ITsapiCSTAUniversalFailure

Feature Name and Description	Class or Interface	Returned/Used by Methods in Class or Interface
User-to-user information used specifically for a Q.931/I.451 User-Network Call Control Message - This form of UUI can be used to send commands out to an ISDN network, and subsequently to an SS7 network if an ISDN/SS7 gateway is used. An object of this type is initialized with a byte array value (see constructor) and its value may be retrieved as a byte array (see UserToUserInfo). This information, when available, is obtained via the LucentCallInfo.getUserToUserInfo() method. An instance so acquired may be classified using 'instanceof'.	Q931UserToUserInfo	LucentCall LucentCallInfo LucentConnection LucentRouteSession LucentTerminalConnection LucentCallInfo
Connection ID - The ConnectionID is used to access the contents of a TSAPI ConnectionID as defined by Avaya's TSAPI service implementation.	Not applicable	ConnectionID
Added Cause Values – The LucentEventCause gives the list of event cause values returned in a number of contexts by the underlying Avaya TSAPI service. Note that 'EC_NONE' through 'EC_VOICE_UNIT_INITIATOR' values are taken from of the ECMA-179 'CSTA 1 Services' specification, and the subsequent cause value extensions, added specifically to expose additional capability, adopted names and values outlined in the CSTA3 service specification (ECMA-269, 'CSTA 3 Services').	Not applicable	LucentEventCause

Feature Name and Description	Class or Interface	Returned/Used by Methods in Class or Interface
Private interface to RouteUsedEvent returns an Address - This private interface to the RouteUsedEvent helps pre-Avaya JTAPI 3.1 applications which use the JTAPI 1.2 RouteUsedEvent.getRouteUsed() method to be adapted to conform to the JTAPI 1.4 specification with a one-line code change. The problem is that the JTAPI 1.4 getRouteUsed method no longer returns an Address; instead it now returns a Terminal. In many scenarios this is a problem because no Terminal may be used to represent an off-switch party, so for those 'routes' this will return 'null'. An Address may be returned for off-switch parties. To solve this problem caused by the new return value, this private interface includes a new method that returns what the JTAPI 1.2 method used to use (an Address), so that it can be used as a replacement API call.	Not applicable	LucentRouteUsedEvent
DTMF Event reporting using listeners - If a PrivateDataCallListener type listener is used, its callPrivateData() callback will be invoked with an argument of type PrivateDtmfEvent if a DTMF-tone has been detected on the telephone line.	Not applicable	PrivateDtmfEvent

Feature Name and Description	Class or Interface	Returned/Used by Methods in Class or Interface
Access to expanded range of reason codes - This interface extends the LucentV6Agent interface with features specific to TSAPI Version 7 private data.This interface may be used to access additional capabilities. This interface specifically provides access to the ability to set a broader range of reason codes for the setState() method. Specifically: it is an application-defined reasonCode (1-99) which may be specified when the state is set to Agent.NOT_READY or Agent.LOG_OUT. A zero (0) value is also allowed, meaning "no reason".	Not applicable	LucentV7Agent
Access to additional call capabilities - The LucentV7Call interface extends LucentV5Call with additional Avaya features exposed through the LucentV7CallInfo interface. This interface may be used to access additional Call capabilities.	Not applicable	LucentV7Call
Access to additional call information - The LucentV7CallInfo interface provides access to call information from methods that are implemented on the call object, the route session object, and on certain call control call events. For example, if a *CallControlCallObserver* receives a *CallCtlConnAlertingEv*, it may be cast to *LucentV7CallInfo* to use the *getDeviceHistory()* method. These methods may return null if the requested data is not available.	Not applicable	LucentV7CallInfo
Expanded queries for Avaya Aura® Communication Manager - Adds queries which give information about the underlying Avaya switching platform. Introduced with Application Enablement Services Server 3.1.	Not applicable	LucentV7Provider

Endpoint Registration and Unregistration Events

Starting with JTAPI release 6.3.1, AE Services and Avaya JTAPI now provide a way for applications to monitor a station for endpoint registration and unregistration events, and to query for the endpoints registered at a station.

Checking if Endpoint Events are Available

The Endpoint Events feature has the following prerequisites –

1. The TSAPI link on AE Services should be configured using ASAI link version 6 (or later). ASAI link version 6 is available beginning with Avaya Communication Manager Release 6.3.2.
2. The application should negotiate private data version 11 (or later) with the TSAPI service on the AE Server. The JTAPI library, starting with release 6.3.1, automatically requests private data version 11 when creating a Provider.

To check if the Provider is bound to an Avaya Communication Manager using private data version 11 or later, a new interface, `LucentV11Provider`, has been provided with three new methods. In your application, you can cast the Provider instance to `LucentV11Provider` and call one of the below methods to determine if the endpoint events feature is available.

`boolean getEndpointRegisteredEvent()`	Does the provider support endpoint registration events?
`boolean getEndpointUnregisteredEvent()`	Does the provider support endpoint unregistration events?
`boolean getQueryEndpointRegistrationInfo()`	Does the provider support querying for a list of endpoints currently registered at a given Address?

Registered Endpoints Query

A new interface - `LucentV11Address` - has been added to the Address hierarchy in JTAPI. This interface defines a single new method –

- `V11RegisteredEndpointInfo[] getRegisteredEndpoints()`

This method returns a list of endpoints currently registered at the station station corresponding to the Address. For each endpoint, the following information is provided –

- `instanceID` – For H.323 endpoints registered through DMCC, the instanceID is 0-2. For H.323 endpoints not registered through DMCC and for SIP endpoints, the instanceID is always 0. To uniquely identify an endpoint, applications must use both the `endpointAddress` and `instanceID` fields.
- `endpointAddress` – For H.323 endpoints, this is the IP address of the endpoint. For SIP endpoints, this is the endpoint's Universal Resource Identifier (URI).
- `switchEndIpAddress` – The switch-end IP address serving the endpoint.
- `macAddress` – The Media Access Control (MAC) address received from the endpoint when the endpoint registered, or if the endpoint's MAC address is unknown, the value "`00:00:00:00:00:00`".

- `productID` – For H.323 endpoints, this is an identifier submitted by the endpoint during registration. Its value is one of the product IDs administered on the Avaya Communication Manager system-parameters customer-options screen. For SIP endpoints, the `productID` is "`SIP_Phone`".
- `networkRegion` – The network region (1-250) administered for the endpoint on Avaya Communication Manager.
- `mediaMode` – The media mode in use by the endpoint. The possible values are:
  - `MM_CLIENT_SERVER` – The endpoint is registered in either client media mode or server media mode.
  - `MM_TELECOMMUTER` – The endpoint is registered in Telecommuter media mode.
  - `MM_NONE` – The endpoint is registered without media control. This media mode is sometimes referred to as "Shared Control" because it allows a DMCC application to share control of an extension with another endpoint registered to that extension.
  - `MM_OTHER` – The endpoint is registered with some other media mode not listed above.
- `dependencyMode` – The dependency mode in use by the endpoint. The possible values are:
  - `DM_MAIN` – The endpoint is registered with dependency mode Main. The endpoint can originate and receive calls. Only one endpoint can be registered to the extension with dependency mode Main. Typically, this is a physical set or an IP softphone.
  - `DM_DEPENDENT` – The endpoint is registered with dependency mode Dependent. An endpoint can only register with this dependency mode if another endpoint is already registered with dependency mode Main.
  - `DM_INDEPENDENT` – The endpoint is registered with dependency mode Independent. The endpoint can originate and receive calls even if another endpoint is not registered with dependency mode Main.
  - `DM_OTHER` – The endpoint is registered with some other dependency mode not listed above.
- `unicodeScript` – For H.323 endpoints, this is a set of bit flags indicating which Unicode character sets are supported by the station. For SIP endpoints, this parameter is set to `US_NONE`. For a list of supported bit flags, refer to the Javadoc for the `LucentEndpointUnicodeScript` class.
- `stationType` – The station type administered for the extension.
- `signalingProtocol` – The signaling protocol for the endpoint. The possible values are:
  - `SP_H323` – The endpoint registered as an H.323 endpoint.
  - `SP_SIP` – The endpoint registered as a SIP endpoint.
  - `SP_NOT_SPECIFIED` – Avaya Communication Manager cannot provide the endpoint's signaling protocol.

Note: Avaya Communication Manager cannot provide some information for a SIP station that is not monitored. It is recommended that the application add a `LucentV11AddressListener` to the Address before querying it using the `LucentV11Address.getRegisteredEndpoints` () method.

Endpoint Events

A new interface - `LucentV11AddressListener` - has been added to the Listener hierarchy in JTAPI. To receive endpoint events, your AddressListener should implement this interface and provide an implementation for the following new methods –

- `void endpointRegistered(LucentEndpointRegisteredEvent event)`
- `void endpointUnregistered(LucentEndpointUnregisteredEvent event)`

The `endpointRegistered` method will be called whenever an H.323 or SIP endpoint registers to the monitored station. The information contained in the event is the same as provided by the `LucentV11Address.getRegisteredEndpoints` () method above. When the listener is added to the Address, it will immediately receive endpoint events for any endpoints already registered at that Address. Each event will contain information about a single endpoint, so multiple events will be delivered to the application if there is more than one endpoint registered.

The `endpointUnregistered` method will be called whenever an H.323 or SIP endpoint unregisters from the station. The event will contain the following information –

- `instanceID` – For H.323 endpoints registered through DMCC, the instanceID is 0-2. For H.323 endpoints not registered through DMCC and for SIP endpoints, the instanceID is always 0. To uniquely identify an endpoint, applications must use both the `endpointAddress` and `instanceID` fields.
- `endpointAddress` – For H.323 endpoints, this is the IP address of the endpoint. For SIP endpoints, this is the empty string (“”).
- `switchEndIpAddress` – The switch-end IP address serving the endpoint.
- `dependencyMode` – The dependency mode with which the endpoint had been registered.
- `stationType` – The station type administered for the extension.
- `signalingProtocol` – The signaling protocol for the endpoint. The possible values are:
  - `SP_H323` – The endpoint registered as an H.323 endpoint.
  - `SP_SIP` – The endpoint registered as a SIP endpoint.
  - `SP_NOT_SPECIFIED` – Avaya Communication Manager cannot provide the endpoint’s signaling protocol.
- `reason` – The reason that the endpoint unregistered. For a list of possible reason codes that may be reported in tis parameter, refer to the Javadoc for the `LucentEndpointUnregisteredReason` class.
- `cmreason` – The uninterpreted reason that the endpoint unregistered, as reported by Avaya Communication Manager. Because future releases of Avaya Communication Manager may include new reason codes for why an endpoint unregistered, the uninterpreted value is made available to applications. The value of the reason field forall such newly supported reason codes will be UR_OTHER.

The two code snippets below show an example of adding a LucentV11AddressListener to an Address in order to receive endpoint events, and querying an Address for currently registered endpoints.

```
// check if the provider supports endpoint events
if ( ((LucentV11Provider)provider).getEndpointEvents() ) {

    Address address = provider.getAddress(extension);

    // CustomAddressListener should implement LucentV11AddressListener
    CustomAddressListener listener = new CustomAddressListener();
    address.addAddressListener(listener);
} else {
    System.err.println("Endpoint events not supported");
}
```

```
// check if the provider supports querying for registered endpoints
if ( ((LucentV11Provider)provider).getQueryEndpointRegistrationInfo() ) {
    V11RegisteredEndpointInfo[] endpoints = null;

    Address address = provider.getAddress(extension);
    endpoints = ((LucentV11Address)address).getRegisteredEndpoints();
    if (endpoints != null && endpoints.length > 0) {
        for (V11RegisteredEndpointInfo endpoint : endpoints)
            // print or process endpoint details
    } else
        System.out.println("No registered endpoints or endpoint info not" +
                           "available for this station");
} else {
    System.err.println("Registered endpoints query not supported");
}
```

Vendor independent private data extensions to JTAPI

The private data extensions to JTAPI assist independent switch vendors in the creation of a private data package for their switches, or allow application programmers to use or interpret private data when they are supplied with private data in its raw form (i.e., without an intermediate private data package.) The following sections describe guidelines for using or interpreting private data when it is supplied in its raw form.

Initialization of Private Data

In order to use or interpret private data from a switch vendor other than Avaya, the application must specify the vendor name and the version of the private data that is to be used. The particular format of the name and version strings used is supplied by the vendor.

The specification of the vendor name and the version of the private data must be done after the application creates a JtapiPeer but before it creates the Provider. The ITsapiPeer.addVendor() method allows vendor names and versions to be specified to the application. For example, if a JtapiPeer has been created (called peer) which is an instance of ITsapiPeer, then:

((ITsapiPeer)peer).addVendor("Brand X","1-3")

indicates that the application knows how to interpret private data from vendor "Brand X" as well as versions 1, 2, and 3 of that private data. If the application supports private data produced by multiple vendors, the application may call addVendor() multiple times before receiving the Provider.

When a String containing the vendor name and version is passed to JtapiPeer.getProvider(), a particular Provider will be connected to a single vendor delivering one particular version of private data. The application determines the connected vendor and version by executing the ITsapiProvider.getVendor() and ITsapiProvider.getVendorVersion() methods.

Once a particular vendor and version is associated with a particular Provider, this association will not change for the life of the Provider. If the application wants a different Provider, the application must call ITsapiPeer.addVendor() again.

Using TsapiPrivate as a JTAPI Private Data Object

Where JTAPI specifies that a private data Object is to be passed in as an argument to a method, this implementation of JTAPI requires the Object to be an instance of TsapiPrivate. Where JTAPI specifies that a private data Object is to be returned from a method, in this implementation, the returned Object is always an instance of TsapiPrivate.

When constructing a TsapiPrivate object to be used with the sendPrivateData() methods, waitForResponse must be set so that the appropriate action is taken.

- A value of true indicates that the implementation should block sendPrivateData() until a response is received from the switch. This response will be passed back to the application as the return code from sendPrivateData(). This is equivalent to the TSAPI request cstaEscapeService().
- A value of false indicates that the implementation should return immediately (with a null) from sendPrivateData(), without waiting for a response from the switch. This is equivalent to the TSAPI request cstaSendPrivateEvent().
- When a TsapiPrivate object is passed as an argument to a setPrivateData() method, the waitForResponse flag is ignored.

APPENDIX C: TSAPI and JTAPI API level comparisons

The Avaya JTAPI implementation internally delegates to the TSAPI implementation. Hence by definition, this JTAPI implementation can support only functionality that TSAPI itself supports

The table below documents the TSAPI requests that you can expect to be initiated given a particular JTAPI API call invocation

JTAPI interface	JTAPI method	TSAPI request
Call	connect	cstaMakeCall
Connection	disconnect	cstaClearConnection
JtapiPeer	getServices	acsEnumServerNames
JtapiPeer	getProvider	acsOpenStream
JtapiPeer	getProvider	cstaSysStatStart
Provider	shutdown	acsCloseStream
Provider	shutdown	cstaSysStatStop
Provider	getState	cstaSysStatReq
TerminalConnection	answer	cstaAnswerCall
AgentTerminal	addAgent	cstaSetAgentState
Agent	setState	cstaSetAgentState
Agent	getState	cstaQueryAgentState
CallCenterCall	connectPredictive	cstaMakePredictiveCall
RouteAddress	registerRouteCallback	cstaRouteRegisterReq
RouteAddress	cancelRouteCallback	cstaRouteRegisterCancel
RouteSession	selectRoute	cstaRouteSelectInv
RouteSession	endRoute	cstaRouteEndInv
CallControlAddress	setForwarding	cstaSetForwarding
CallControlAddress	cancelForwarding	cstaSetForwarding
CallControlAddress	getForwarding	cstaQueryForwarding
CallControlAddress	getDoNotDisturb	cstaQueryDoNotDisturb
CallControlAddress	setDoNotDisturb	cstaSetDoNotDisturb
CallControlAddress	getMessageWaiting	cstaQueryMsgWaitingInd
CallControlAddress	setMessageWaiting	cstaSetMsgWaitingInd
CallControlCall	drop	cstaClearCall
CallControlCall	conference	cstaConferenceCall
CallControlCall	Transfer(call)	cstaTransferCall
CallControlCall	Transfer(string)	cstaEscapeService
CallControlCall	consult	cstaConsultationCall

CallControlConnection	redirect	cstaDeflectCall
CallControlTerminal	getDoNotDisturb	cstaQueryDoNotDisturb
CallControlTerminal	setDoNotDisturb	cstaSetDoNotDisturb
CallControlTerminal	pickup	cstaPickupCall
CallControlTerminal	pickupFromGroup	cstaGroupPickupCall
CallControlTerminalConnection	hold	cstaHoldCall
CallControlTerminalConnection	unhold	cstaRetrieveCall
PrivateData	sendPrivateData	cstaSendPrivateEvent

The following TSAPI requests are currently un-implemented by this JTAPI implementation. Therefore, there is no access to the private data for these TSAPI requests.

Service Type	**TSAPI request**
Call Control Services	cstaAlternateCall, cstaCallCompletion, cstaReconnectCall
Supplementary Services	cstaQueryLastNumber, cstaQueryDeviceInfo
Monitor Services	cstaChangeMonitorFilter, FeatureEventReport, CSTACallInfoEvent
Escape Services	cstaEscapeServiceConf, cstaEscapeService
Maintenance Services	cstaChangeSysStatFilter

The following table maps JTAPI listener callbacks to their deprecated observer events and the corresponding CSTA unsolicited TSAPI event that caused it to be invoked.

The private data related to these TSAPI events will be contained in the respective event in case of observer events and will be a part of the first parameter passed to the listener callback in case of listeners,

TSAPI event	**JTAPI Observer event**	**JTAPI listener callback**
CSTACallClearedEvent	CallInvalidEv	CallListener#callInvalid
CSTAMonitorEndedEvent	CallObservationEndedEv	CallListener#callEventTransmissionEnded
CSTADeliveredEvent	ConnAlertingEv	ConnectionListener#connectionAlerting
CSTADivertedEvent	ConnDisconnectedEv	ConnectionListener#connectionDisconnected
CSTAEstablishedEvent	ConnConnectedEv	ConnectionListener#connectionConnected
CSTAHeldEvent	CallCtlTermConnHeldEv	CallControlTerminalConnectionListener#terminalConnectionHeld
CSTARetrievedEvent	CallCtlTermConnTalkingEv	CallControlTerminalConnectionListener#terminalConnectionTalking
CSTAConnectionClearedEvent	ConnDisconnectedEv	ConnectionListener#connectionDisconnected
CSTAFailedEvent	ConnFailedEv	ConnectionListener#connectionFailed
CSTADoNotDisturbEvent	CallCtlAddrDoNotDisturbEv	CallControlAddressListener#addressDoNotDis

		turb
CSTAForwardingEvent	CallCtlAddrForwardEv	CallControlAddressListener# addressForwarded
CSTAMessageWaitingEvent (not supported)	CallCtlAddrMessageWaitingEv	CallControlAddressListener# addressMessageWaiting
CSTAServiceInitiatedEvent	CallCtlConnInitiatedEv	CallControlConnectionListener# connectionInitiated
CSTAOriginatedEvent	CallCtlConnEstablishedEv	CallControlConnectionListener# connectionEstablished
CSTANetworkReachedEvent	CallCtlConnNetworkReachedEv	CallControlConnectionListener# connectionNetworkReached
CSTAQueuedEvent	CallCtlConnQueuedEv	CallControlConnectionListener# connectionQueued
CSTALoggedOffEvent	ACDAddrLoggedOffEv AgentTermLoggedOffEv	ACDAddressListener# acdAddressLoggedOff AgentTerminalListener #agentTerminalLoggedOff
CSTALoggedOnEvent	ACDAddrLoggedOnEv AgentTermLoggedOnEv	ACDAddressListener# acdAddressLoggedOn AgentTerminalListener #agentTerminalLoggedOn
CSTANotReadyEvent (not supported)	ACDAddrNotReadyEv AgentTermNotReadyEv	ACDAddressListener# acdAddressNotReady AgentTerminalListener #agentTerminalNotReady
CSTAReadyEvent (not supported)	ACDAddrReadyEv AgentTermReadyEv	ACDAddressListener# acdAddressReady AgentTerminalListener #agentTerminalReady
CSTAWorkNotReadyEvent (not supported)	ACDAddrWorkNotReadyEv AgentTermWorkNotReadyEv	ACDAddressListener# acdAddressWorkNotReady AgentTerminalListener #agentTerminalWorkNotReady
CSTAWorkReadyEvent (not supported)	ACDAddrWorkReadyEv AgentTermWorkReadyEv	ACDAddressListener# acdAddressWorkReady AgentTerminalListener #agentTerminalWorkReady

The following table maps route related TSAPI events and their JTAPI equivalents

TSAPI Event	**JTAPI Event**
CSTARouteRequestExtEvent	RouteEvent
CSTAReRouteRequestEvent	ReRouteEvent
CSTARouteUsedExtEvent	RouteUsedEvent
CSTARouteEndEvent	RouteEndEvent
CSTARouteRegisterAbortEvent	RouteCallbackEndedEvent

Converting TSAPI based Java constructs to standard JTAPI objects

Some Avaya private interfaces expose classes that represent raw TSAPI constructs.

For example, com.avaya.jtapi.tsapi.V7DeviceHistoryEntry#getOldConnectionID() returns you a ConnectionID, which contains a TSAPI callID.

To convert this callID into a first class JTAPI object (in this case a javax.telephony.Call implementation), please cast the provider to the com.avaya.jtapi.tsapi.ITsapiProviderPrivate interface and use the getCall(int callID) API.

```
V7DeviceHistoryEntry history = ……..;  //assume code that returns a
V7DeviceHistoryEntry

Provider avayaProvider = ….;  //assume code that creates a provider
instance

/*com.avaya.jtapi.tsapi.V7DeviceHistoryEntry#getOldConnectionID()
returns you a ConnectionID, which contains a TSAPI callID.*/

ConnectionID connID = history.getOldConnectionID();

/*promote to a first class JTAPI object (in this case a
javax.telephony.Call implementation)*/

Call call = ((com.avaya.jtapi.tsapi.ITsapiProviderPrivate)
avayaProvider).getCall(connID.getCallID());
```

Similarly, the com.avaya.jtapi.tsapi.ITsapiConnIDPrivate interface exposes a ConnectionID object.

ITsapiProviderPrivate also contains methods to promote ConnectionID objects to JTAPI Connection/TerminalConnection objects (as the case may be).

ExtendedDeviceID is currently not exposed, but may be exposed in future releases. Please use the relevant methods in ITsapiProviderPrivate to promote these ExtendedDeviceID objects to JTAPI Address / Terminal implementations.

TSAPI Construct	**Java Class**	**JTAPI Object**	**Conversion Method in ITsapiProviderPrivate**
ExtendedDeviceID_t	ExtendedDeviceID	Address	getAddress()
ExtendedDeviceID_t	ExtendedDeviceID	Terminal	getTerminal()
ConnectionID_t	ConnectionID	Connection	getConnection()
ConnectionID_t	ConnectionID	TerminalConnection	getTerminalConnection()
callID (field in a ConnectionID_t)	int	Call	getCall()

APPENDIX D - TSAPI Error Code Definitions

This appendix lists all of the values for the TSAPI error codes.

There are two major classes of TSAPI error codes:

- CSTA universal Failures
- ACS Universal Failures

CSTA Universal Failures

CSTA Universal Failures are error codes returned by CSTAErrorCode:Unexpected CSTA error code. The following table lists the definitions for the CSTA error codes. Consult the TSAPI Programmer's Guide for the definition of the numeric error code.

TABLE: CSTA Error Definitions

Error	Numeric Code
genericUnspecified	0
genericOperation	1
requestIncompatibleWithObject	2
valueOutOfRange	3
objectNotKnown	4
invalidCallingDevice	5
invalidCalledDevice	6
invalidForwardingDestination	7
privilegeViolationOnSpecifiedDevice	8
privilegeViolationOnCalledDevice	9
privilegeViolationOnCallingDevice	10
invalidCstaCallIdentifier	11

invalidCstaDeviceIdentifier	12
invalidCstaConnectionIdentifier	13
invalidDestination	14
invalidFeature	15
invalidAllocationState	16
invalidCrossRefId	17
invalidObjectType	18
securityViolation	19
genericStateIncompatibility	21
invalidObjectState	22
invalidConnectionIdForActiveCall	23
noActiveCall	24
noHeldCall	25
noCallToClear	26
noConnectionToClear	27
noCallToAnswer	28
noCallToComplete	29
genericSystemResourceAvailability	31
serviceBusy	32
resourceBusy	33
resourceOutOfService	34

networkBusy	35
networkOutOfService	36
overallMonitorLimitExceeded	37
conferenceMemberLimitExceeded	38
genericSubscribedResourceAvailability	41
objectMonitorLimitExceeded	42
externalTrunkLimitExceeded	43
outstandingRequestLimitExceeded	44
genericPerformanceManagement	51
performanceLimitExceeded	52
unspecifiedSecurityError	60
sequenceNumberViolated	61
timeStampViolated	62
pacViolated	63
sealViolated	64
genericUnspecifiedRejection	70
genericOperationRejection	71
duplicateInvocationRejection	72
unrecognizedOperationRejection	73
mistypedArgumentRejection	74
resourceLimitationRejection	75

acsHandleTerminationRejection	76
serviceTerminationRejection	77
requestTimeoutRejection	78
requestsOnDeviceExceededRejection	79
unrecognizedApduRejection	80
mistypedApduRejection	81
badlyStructuredApduRejection	82
initiatorReleasingRejection	83
unrecognizedLinkedidRejection	84
linkedResponseUnexpectedRejection	85
unexpectedChildOperationRejection	86
mistypedResultRejection	87
unrecognizedErrorRejection	88
unexpectedErrorRejection	89
mistypedParameterRejection	90
nonStandard	100

ACS Universal Failures

ACS Universal Failures are error codes returned by CSTAErrorCode:Unexpected ACS error code. The following table lists the definitions for the ACS error codes. Consult the TSAPI Programmer's Guide for the definition of the numeric error code

Error	Numeric Code	Description
ACSERR_APIVERDENIED	-1	This return indicates that the API Version requested is invalid and not supported by the existing API Client Library.
ACSERR_BADPARAMETER	-2	One or more of the parameters is invalid.
ACSERR_DUPSTREAM	-3	This return indicates that an ACS Stream is already established with the requested Server.
ACSERR_NODRIVER	-4	This error return value indicates that no API Client Library Driver was found or installed on the system.
ACSERR_NOSERVER	-5	This indicates that the requested Server is not present in the network.
ACSERR_NORESOURCE	-6	This return value indicates that there are insufficient resources to open a ACS Stream.
ACSERR_UBUFSMALL	-7	The user buffer size was smaller than the size of the next available event.
ACSERR_NOMESSAGE	-8	There were no messages available to return to the application.
ACSERR_UNKNOWN	-9	The ACS Stream has encountered an unspecified error.
ACSERR_BADHDL	-10	The ACS Handle is invalid.

ACSERR_STREAM_FAILED	-11	The ACS Stream has failed due to network problems. No further operations are possible on this stream.
ACSERR_NOBUFFERS	-12	There were not enough buffers available to place an outgoing message on the send queue. No message has been sent.
ACSERR_QUEUE_FULL	-13	The send queue is full.
ACSERR_SSL_INIT_FAILED	-14	A stream could not be opened because the initialization of the OpenSSL library failed.
ACSERR_SSL_CONNECT_FAILED	-15	A stream could not be opened because the SSL connection failed.
ACSERR_SSL_FQDN_MISMATCH	-16	During the SSL handshake, the fully qualified domain name (FQDN) in the server certificate did not match the expected FQDN.

Glossary

A

AE

Used as a “shorthand” term in this documentation for Application Enablement.

AES

Stands for Advanced Encryption Scheme or Application Enablement Services

API

Application Programming Interface. A “shorthand” term in this documentation for the Java interface provided by the Application Enablement Services.

application machine

The hardware platform that the JTAPI API library and the cli are running on

C

client application

An application created using the JTAPI library

CSTA

Computer-Supported Telecommunications Applications

E

ECMA

European Computer Manufacturers Association. A European association for standardizing information and communication systems in order to reflect the international activities of the organization.

I

IPv6

Internet Protocol Version 6

J

JDK

Java Developers Kit

JRE

Java Runtime Environment

J2SE

Java 2 Platform, Standard Edition

JTAPI

Java Telephony Application Programming Interface

JVM

Java Virtual Machine. Interprets compiled Java binary code for a computer's processor so that is can perform a Java program's instructions

O

OAM

Operations, Administration and Maintenance

R

RPM

Red Hat Package Manager

S

SAT

System Access Terminal (for Communication Manager)

SDK

Software Development Kit

T

TLink

TSAPI Link: It refers to a switch connection between a specific switch and a specific AE Services Server.

TSAPI

Telephony Services API