Tutorial: An Introduction to Avaya JTAPI SDK

Contents

About this Tutorial

This tutorial provides a brief introduction to the basic steps required to develop telecommunication applications using Avaya Java Telephony API (JTAPI) SDK.

After completing this tutorial, the developer will be able to:

- Implement the steps for the initialization of a JTAPI application.
- Implement the steps for the shutdown of a JTAPI application.

Intended Audience

This tutorial is intended for Java programmers who have a working knowledge of telecommunications applications.

For a complete understanding of the Avaya JTAPI SDK, refer to *Avaya Computer Telephony 1.3 Java Telephony API (JTAPI) for Avaya MultiVantage® Programmer's Reference, October 2003, Issue 1*, found online on the Avaya DevConnect website (http://www.avaya.com/devconnect) and on the Avaya Support Center website (http://www.avaya.com/support) under *Communication Systems*. This document is the last .pdf version of the JTAPI Programmer's Reference that was delivered. The documentation is now provided as a javadoc. To access the javadoc, download it from the Avaya DevConnect website (http://www.avaya.com/devconnect). It can be located by performing a DevConnect search for "AE Services JTAPI Programmer's Reference".

For a complete understanding of the Avaya JTAPI libraries, refer to the JTAPI documentation. Refer the package summary for the package `com.avaya.jtapi.tsapi` in the JTAPI Programmer's Reference at the following page:

```
com\avaya\jtapi\tsapi\package-summary.html
```

For those new to Avaya Communication Manager, please take a course from Avaya University (http://www.avaya.com/learning) to learn more about Communication Manager and its features.

Prerequisites

Please refer to the document *Avaya MultiVantage® Application Enablement Services TSAPI, JTAPI, and CVLAN Client and SDK Installation Guide* to install the JTAPI client and the SDK for configuration instructions for the development environment to successfully run JTAPI applications. If the development environment has not been configured, then please do so before proceeding further.

Chapter 1: Overview of JTAPI

JTAPI (Java Telephony API) is a portable, object-oriented application programming interface which specifies the standard Telephony Application Programming Interface (API) for computer-telephony applications using Java.

JTAPI is a Java-based client-side interface to the TSAPI service running on AE Services server, and as such, it provides third party call control capabilities to Avaya Communication Manager. These third-party call control capabilities include:

- The ability to make, answer, log, transfer, hold, retrieve, divert, conference and drop calls.
- Control and interaction of calls in vector processing, predictive dialing and call classification, or skills-based routing.
- Providing a snapshot of a device, including information regarding the calls on the device and the parties on the call.
- Performing logical services such as Agent login and logout.

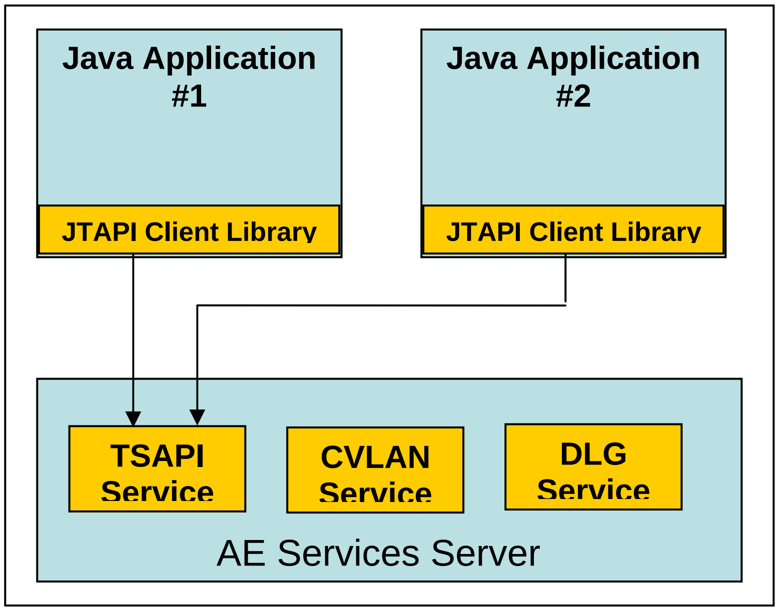

Figure 1: AE Services Server interacting with the Java applications

The diagram shown above (Figure 1) describes how JTAPI provides an interface between Java Applications or Applets and the AE Services server.

The Avaya JTAPI client library is composed of a set of Java *packages.* Each package provides a specific piece of functionality for a certain aspect of computer-telephony applications. JTAPI client library communicates with the Telephony Services API (TSAPI) service running on Avaya AE Services server. TSAPI provides a whole range of call control capabilities such as controlling specific calls or stations, routing of incoming calls and receiving notification of events.

The AE Services server is a platform that enables the development of modular building blocks for advanced communication services.

Chapter 2: Setting Up the JTAPI Programming Environment

The JTAPI programming environment is the software installation and configuration that is needed to compile and run a JTAPI application. Before setting the JTAPI programming environment, make sure that the Java Development Kit (version 1.4.2 or higher) and Java Runtime Environment are installed on the system(s) where the JTAPI application is to be developed and run.

Setting up and configuring the JTAPI programming environment consists of the following initial steps:

1. Installing the JTAPI client.
2. Installing the JTAPI SDK.
3. Changing `classpath` and the configuration files.

2.1 JTAPI Client

The JTAPI client includes libraries that provide an interface to the AE Services server. The server interacts with Avaya Communication Manager to provide call processing services.

For JTAPI applications to run in an AE Services/Communication Manager environment, the JTAPI client must be installed on the machine with the application. The following components are available after the JTAPI client is installed:

1. **`ECSJtapia.jar`**: This is a Java archive file which contains the Avaya JTAPI implementation (executables). This file must be imported into the Java application.
2. **`TSAPI.PRO`**: This is a configuration file and is used by the JTAPI application to get the IP address and port for the AE Services server. Other information, such as library configuration, may be present in this file
3. **`ReadMe`**: This file contains updated information about the JTAPI client.
4. **`JTAPI Test:`** JTAPI Test is a sample application which can be used to test the installation of the JTAPI client software.

2.2 JTAPI SDK

The AE Services Java Telephony API (JTAPI) SDK is available to developers to help them create applications that access the AE Services' TSAPI service using a Java API. The JTAPI client is a prerequisite for installing and using the JTAPI SDK – developers must install the JTAPI client first. The JTAPI SDK components that are installed by default are as follows:

1. **`Sample Code`**: This directory contains coded examples for developing various applications using Avaya JTAPI.
2. **`Avaya JTAPI Exerciser`**: This is a Java application tool that allows the user to create JTAPI objects and invoke their methods interactively. The JTAPI Exerciser is meant to be used to help JTAPI application developers discover how to use the API to access information and invoke requests.
3. **`ReadMe`** file: This file contains updated information about the Avaya JTAPI SDK.

2.3 Setting up `CLASSPATH`

Developers should ensure that `CLASSPATH` correctly points to the directory where the files **`TSAPI.PRO`** and **`Ecsjtapia.jar`** reside. When the JTAPI client is installed, `classpath` is set to the directory `<Installation Path>/AE Services/JTAPI Client/JTAPI`. Any changes to the location of these files must be made to the `CLASSPATH` environment variable. This will ensure that the JTAPI client library is located correctly and that the library finds the **`TSAPI.PRO`** file which contains the AE Services' IP address and port number for accessing TSAPI services.

2.4 Changes to the file `TSAPI.PRO`

Once the JTAPI client is installed, developers also need to change the **`TSAPI.PRO`** file to use the correct AE Services' IP address and port number. The following parameters need to be changed to accomplish this:

1. Telephony Servers: This is a mandatory parameter and should be set to the IP Address of the AE Services server.

Optional settings in the **`TSAPI.PRO`** file include the following variables useful to developers:

1. `debugLevel`: This is an optional parameter and should be used only during application development. This parameter provides debugging information for the JTAPI applications.
2. `altTraceFile`: This parameter needs to be set when `debugLevel` is selected. The logs generated will be written into this file.

A sample **`TSAPI.PRO`** file (found in `<Installation Path>/AE Services/JTAPI Client/JTAPI`) is shown below:

```
# tsapi.pro
#
# This file must be located in one of the directories found in CLASSPATH
#
# This is a list of the servers offering Telephony Services via TCP/IP.
# Either domain name or IP address may be used; default port number is 450
# The form is: host_name=port_number   For example:
#
# tserver.mydomain.com=450
# 127.0.0.1=450
#
# (Remove the '#' when creating actual server entries.)
[Telephony Servers]
192.168.17.128=450
debugLevel=7
altTraceFile=tsapi_trace.txt
```

Chapter 3: Initializing the Application

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

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

The `JtapiPeer` interface in the `javax.telephony` package represents an Avaya custom implementation of the JTAPI. An instance of the `JtapiPeer` object is obtained using the `JtapiPeerFactory` class.

The term 'peer' is Java nomenclature for "a platform-specific implementation of a Java interface or API". 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.

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 uses the **`TSAPI.PRO`** file to get the IP addresses of the AE Services server(s) and then gets all the CTI-links configured on each of the AE Services servers configured in the **`TSAPI.PRO`** file.

If no services are returned from the `getServices()` method request, check the AE Services server's IP address in the **`TSAPI.PRO`** file and the provisioning and state of the CTI-links on the AE Services 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, the application can use any one of those services to create a `Provider` object.

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.

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
  * @param String 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);
}
```

Chapter 4: JTAPI Objects

This chapter explains the various JTAPI objects that are commonly used in a JTAPI application and represent various telephony objects (JTAPI objects are similar to Java objects which can be instantiated and invoked to implement various methods.) The various JTAPI objects are `Provider`, `Terminal`, `Address`, `Call` and `Connection`. Detailed description of each of these objects is presented in the following sections.

4.1 The `Provider` Object

A `Provider` represents the telephony software-entity (i.e. AE Services) that interfaces with a telephony subsystem (i.e. Communication Manager).

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

```
CTI-Link-CM1;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`.

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.

4.1.1 States of the `Provider` Object

The `Provider` may be in one of the following states:

- `Provider.IN_SERVICE`
- `Provider.OUT_OF_SERVICE`
- `Provider.SHUTDOWN`

The provider object should be in the `Provider.IN_SERVICE` state before calling any method.

The `Provider` state represents whether any action on that `Provider` is valid or not. Table 1 below describes each state:

Provider State	**Description**	**Constant Value**
`Provider.IN_SERVICE`	This state indicates that the `Provider` is currently active and available for use.	16
`Provider.OUT_OF_SERVICE`	This state indicates that a `Provider` is temporarily unavailable for use. Several methods in the JTAPI are invalid when the `Provider` is in this state. `Providers` may come back in service at any time; however, the application cannot take a direct action to cause this change.	17
`Provider.SHUTDOWN`	This state indicates that a `Provider` is no longer available for use.	18

Table 1: Provider states and their descriptions

The `getState()` method of the `Provider` object returns the current state of `Provider`. The value returned by the method is constant value for one of `Provider.IN_SERVICE`, `Provider.OUT_OF_SERVICE`, or `Provider.SHUTDOWN`. The following is the syntax for the `getState()` method.

```
public int getState()
```

Figure 2 below shows the allowable state transitions for the `Provider` object as defined in Avaya's implementation of the package.

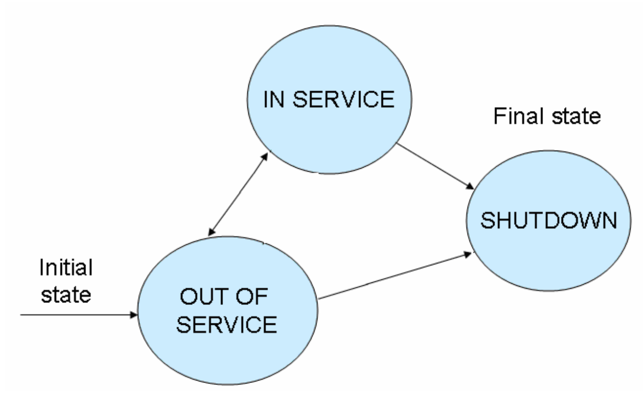

Figure 2: State diagram showing allowable `Provider` state transitions

The `Provider.shutdown()` method requests the `Provider` to shut itself down and perform all necessary cleanup. The syntax for this request is given below:

```
public void shutdown()
```

Applications invoke the `Provider.shutdown()` method when they no longer intend to use the `Provider`, usually, right before they exit. This method is intended to allow the `Provider` to perform any necessary cleanup that would not be handled when the Java objects are garbage collected. The `Provider.shutdown()` method causes the `Provider` to move into the `Provider.SHUTDOWN` state where it will stay indefinitely. If the `Provider` is already in the `Provider.SHUTDOWN` state, this method does nothing.

4.2 The `Terminal` Object

A `Terminal` represents a physical hardware endpoint connected to the telephony domain. An example of a `Terminal` is a telephone set (e.g., an analog station, a DCP station, an IP station or a softphone). A `Terminal` does not have to take the form of this limited and traditional example. For example, computer workstations and hand-held devices are modeled as a `Terminal` if they act as physical endpoints in a telephony network.

The `Terminal` object is represented by an interface `Terminal` under the `javax.telephony` package.

A `Terminal` object has a string name which is unique for all `Terminal` objects. The `Terminal` does not attempt to interpret this string in any way. This name is first assigned when the `Terminal` is created and does not change throughout the lifetime of the object. CSTA allows an implementation to choose its own naming scheme for `Terminals`. In Avaya's JTAPI implementation, `Terminals` are named by a "primary" telephone number address at that `Terminal`. The method `Terminal.getName()` returns the name of the `Terminal` object.

`Terminal` objects may be classified into two categories: local and remote. Local `Terminal` objects are those terminals which are part of the `Provider`'s local domain. These `Terminal` objects are created by the implementation of the `Provider` object when it is first instantiated. All of the `Provider`'s local terminals are reported via the `Provider.getTerminals()` method. The terminal objects reported correspond to the set of devices present in the AE Services security database (SDB) that are accessible using the credentials provided in the `JtapiPeer.getProvider()` request. Remote `Terminal` objects are those outside the `Provider`'s domain which the `Provider` learns about during its lifetime through various events (e.g. an incoming call from a currently unknown address). Remote `Terminal` objects are not reported via the `Provider.getTerminals()` method. Note that applications never explicitly create new `Terminal` objects.

The syntax of the `getTerminal()` method is as given below:

```
public Terminal getTerminal(java.lang.String name)
```

The method returns an instance of the `Terminal` class which corresponds to the given name. If no `Terminal` is available for the given name within the `Provider`'s domain, this method throws the `InvalidArgumentException` exception. This indicates that the name provided does not correspond to the name of any `Terminal` which is accessible by the `Provider`.

The following code snippet describes the procedure to use the `getTerminal()` method:

```
try
{
	String extension;
	extension = “40001”;
	Terminal term = Provider.getTerminal(extension);
}

catch (Exception e)
{
	System.out.println("Exception during getTerminal: " + extension + e);
}
```

4.3 The Address Object

An `Address` object represents what is commonly known as a "telephone number".

The `Address` is an interface under the `javax.telephony` package.

An `Address` object represents a logical endpoint of a telephone call like a "telephone number". Terminals and Address objects are essentially one to one in Avaya’s JTAPI implementation.

`Address` objects may be classified into two categories: local and remote. Local `Address` objects are those addresses which are part of the `Provider`'s local domain. These `Address` objects are created by the implementation of the `Provider` object when it is first instantiated. All of the `Provider`'s local addresses are reported via the `Provider.getAddresses()` method. The addresses that are reported are the ones that are accessible using the credentials provided in the `JtapiPeer.getProvider()` request. Remote `Address` objects are those outside of the `Provider`'s domain which the `Provider` learns about during its lifetime through various events. Remote `Addresses` are not reported via the `Provider.getAddresses()` method. Note that applications never explicitly create new `Address` objects.

Following is the syntax of the `getAddresses()` method:

```
public Address[] getAddresses() //throws the ResourceUnavailableException exception.
```

The method returns an array of addresses associated with the `Provider` and within the `Provider`'s domain. This list is static (i.e. it does not change for the given instance) after the `Provider` is first created. If no `Address` objects are associated with this `Provider`, then this method returns `null`. A `ResourceUnavailableException` is thrown by this method to indicate that the number of addresses present in the `Provider` is too large to return as a static array.

The following code snippet describes the procedure to use the `getAddresses()` method:

```
try
{
	Address [] addr = Provider.getAddresses();
}

catch (Exception ex)
{
	System.out.println("Exception during getAddresses: "+ ex);
}
```

4.4 The `Call` Object

A `Call` object models a telephone call. A `Call` can have zero or more `Connections` i.e., a two-party call has two `Connections` and a conference call has three or more `Connections`. Each `Connection` models the relationship between a `Call` and an `Address`, where an `Address` identifies a particular party or a set of parties on a `Call`. Applications create instances of a `Call` object with the `Provider.createCall()` method, which returns a `Call` object that has zero `Connections` and is in the `Call.IDLE` state. The `Call` maintains a reference to its `Provider` for the life of that `Call` object. This `Provider` object instance remains persistent throughout the lifetime of the `Call` object. The `Provider` associated with a `Call` is obtained via the `Call.getProvider()` method.

4.4.1 States of the `Call` Object

A `Call` has a state which is obtained via the `Call.getState()` method. This state describes the current progress of a telephone call and where the call is in its life cycle. The `Call` state may be one of the following three values:

1. `Call.IDLE`
2. `Call.ACTIVE`
3. `Call.INVALID`

Table 2 below provides a detailed description of each of these states:

Call State	**Description**
`Call.IDLE`	This is the initial state for all `Calls`. In this state, the `Call` has zero `Connections`, that is `Call.getConnections()`method returns `null`.
`Call.ACTIVE`	A `Call` with a current ongoing activity is in this state. `Calls` with one or more associated `Connections` must be in this state. If a `Call` is in this state, the `Call.getConnections()` method returns an array with one or more entries.
`Call.INVALID`	This is the final state for all `Calls`. `Call` objects which lose all of their `Connections` objects move into this state. Calls in this state have zero `Connections` and these `Call` objects may not be used for any future action. In this state, the `Call.getConnections()`returns `null`.

Table 2: Call States and their descriptions

The possible `Call` state transitions are given in the state diagram below (Figure 3):

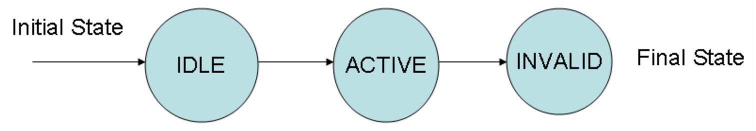

Figure 3: `Call` state transitions

4.5 The `Connection` Object

A `Connection` represents a link (i.e. an association) between a `Call` object and an `Address` object.

A `Connection` object exists if the `Address` is a part of the telephone call. Each `Connection` has a state (accessed via the `Connection.getState()` method) which describes the particular stage of the relationship between the `Call` and the `Address`. Applications can use the `Connection.getCall()` and `Connection.getAddress()` methods to obtain the `Call` and `Address` associated with a particular `Connection`, respectively.

4.5.1 States of the `Connection` Object

The list of connection states defined in the `Connection` interface is described in Table 3 below:

Connection State	**Description**
`Connection.IDLE`	This state is the initial state for all new connections. Connections which are in the `Connection.IDLE` state are not actively part of a telephone call.
`Connection.DISCONNECTED`	This state implies the connection is no longer part of the telephone call, although the connection's references to `Call` and `Address` still remain valid. A `Connection` in this state is interpreted as once previously belonging to this telephone call.
`Connection.INPROGRESS`	This state implies that the `Connection`, which represents the destination (e.g. called) end of a telephone call, is in the process of contacting the destination side.
`Connection.ALERTING`	This state implies that the `Address` is being notified of an incoming call.
`Connection.CONNECTED`	This state implies that a `Connection` and its `Address` are actively part of a telephone call.
`Connection.UNKNOWN`	This state implies that the implementation is unable to determine the current state of the `Connection`. Connections may move in and out of the `Connection.UNKNOWN` state at any time.
`Connection.FAILED`	This state indicates that a `Connection` to that end of the call (`Address` and `Terminal`) has failed.

Table 3: `Connection` States and their descriptions

4.6 Shutting Down a JTAPI Application

To shut down a JTAPI application, the `Shutdown` method of the `Provider` object should be called. A call to `Shutdown` performs all the necessary cleanups. Applications invoke this method when they no longer intend to use the `Provider` method, most often right before they exit. This method allows `Provider` to perform any necessary cleanup which would not be taken care of when the Java objects are garbage collected. This method causes `Provider` to move into the `Provider.SHUTDOWN` state, in which it stays indefinitely.

Chapter 5: Observers and Events

Once the application gets the `Provider` object, the application can start creating instances of a JTAPI object and perform third party call control applications. Applications may also need to monitor JTAPI objects for any state changes.

Observers are basically monitors that can be added to a JTAPI object. Adding an observer to a JTAPI object is equivalent to requesting a monitor start service for that object. All the changes to the monitored object are reported as events to the observer methods. Events are asynchronous in nature and the observer's methods are used to asynchronously notify the JTAPI applications of events that occur. Any application that wishes to monitor a JTAPI object should first instantiate an observer object which implements the needed interfaces and then use the `addObserver()` method on the desired object to associate the observer with the object. This will cause the application to receive future events associated with that object for the specified observer that is added. When the application is done with monitoring a JTAPI object, it should remove the observer for the object using the method `removeObserver()`. Events are reported to the observer until the observer is removed, or connectivity between the application and the AE Services server or Communication Manager is lost, or the object the observer is monitoring is no longer capable of generating events (e.g. the call ended).

Table 4 below gives brief description of the observers for common JTAPI objects discussed in Chapter 4.

JTAPI object	**Interface**	**Description**
`Provider` object	`ProviderObserver`	The `ProviderObserver` interface reports all changes, which happen to the `Provider` object such as shutdown, in service, out of service, etc.
`Terminal` object	`TerminalObserver`	The `TerminalObserver` interface reports all changes which happen to the `Terminal` object.
`Call` object	`CallObserver`	The `CallObserver` interface reports all changes to the `Call` object and all of the `Connection` and `TerminalConnection` objects which are part of the call.
`Address` object	`AddressObserver`	The `AddressObserver` interface reports all changes which happen to the `Address` object.

Table 4: Listeners for common JTAPI objects

Note: A Call observer can be added to `Address`, `Terminal` or `Connection` objects as well. Adding observers this way ensures that Call changes related to only these objects are reported. For example, if there are more than two Connections in a Call, then the Call observer added on Call object will report status changes for all the Connections to the observer, whereas a Call observer added on a specific terminal (or address) will report changes to only the Connection(s) associated with the specified Terminal (or Address).

Also note that, Connection and Call observers end when the respective object ends. That is, Connection and Call observers stop sending events for those objects which no longer exist. This means that a new Call observer needs to be added for every new Call object. However, if the Call observer is added to the `Address` or `Terminal` objects, monitors remain active across Calls and all the calls on an `Address` or `Terminal` can be monitored with a single observer.

A sample code snippet for implementing `ProviderObserver` and the corresponding methods is provided below:

```
private Provider provider;

try
{
       provider.addObserver(new providerEventHandler());
}

catch ( Exception e )
{
       System.out.println("Exception during adding provider observer: "+e);
}

public class providerEventHandler implements providerObserver
{
       // Implementations of the methods of the ProviderObserver interface.

       public void providerChangedEvent(ProvEv[] eventList)
       {
             System.out.println("List of events representing the changes to the
                                    provider object :"+ Arrays.toString(eventList) );
       }
}
```

References

[1] *Avaya Java Telephony API (JTAPI) javadoc.*

[2] *Avaya Computer Telephony 1.3 Java Telephony API (JTAPI) for Avaya MultiVantage Programmer's Reference October 2003, Issue 1* available on the Avaya DevConnect Portal (http://www.devconnect.avaya.com/)