AVAYA

avaya.com

Performing Basic Telephony Operations using Avaya JTAPI SDK

An Avaya DevConnect Tutorial

Table of Contents

About this Tutorial

This tutorial describes the basic telephony services provided by the Avaya Java Telephony API (JTAPI) SDK and shows how to perform various telephony-related operations such as originating, detecting, answering, and disconnecting a call using the SDK.

The tutorial is divided into the following chapters:

- **Chapter 1: Introduction to JTAPI Object Model**

  This chapter provides an overview of the JTAPI Call and Connection objects along with typical call flow scenarios.

- **Chapter 2: Telephony Operations**

  This chapter describes how to perform basic telephony operations using JTAPI. It also describes in detail the call control events received by the application, during various call operations, from the Application Enablement (AE) Services server.

After completing this tutorial, the developer will have a detailed understanding of the fundamental third party call control capabilities such as originating, detecting, answering, and disconnecting a call using Avaya JTAPI SDK. More information on the events and objects described in this tutorial may be found in *Avaya Java Telephony API (JTAPI) javadoc* (reference [2]), available on the Avaya DevConnect portal (**http://www.avaya.com/devconnect**).

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 the Avaya DevConnect portal (**http://www.avaya.com/devconnect**) and do a DevConnect search for 'JTAPI'.

Prerequisites

To perform the operations covered in this tutorial, the application should first be initialized. The developer must be aware of the application initialization and event monitoring procedures to properly use the JTAPI SDK. The developer may refer to the document *Tutorial for Application Initialization using Avaya JTAPI SDK* (reference [1]) to get details about these two procedures.

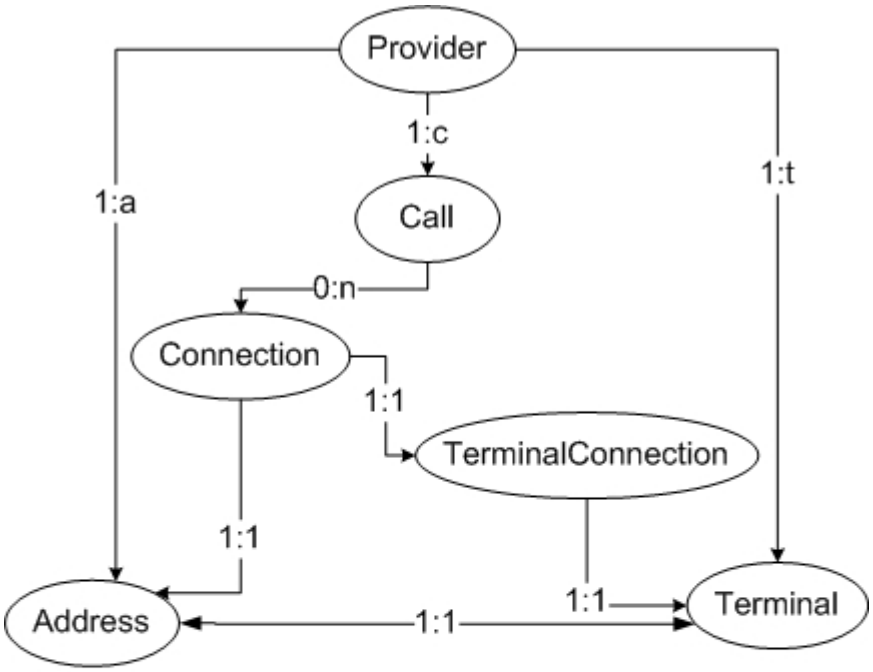
Section 1: Introduction to JTAPI Object Model

A typical telecommunications application needs to perform basic telephony operations such as originating calls, answering calls, terminating calls, etc. and take appropriate actions for the various state changes in the call. The way to accomplish this with Java Telephony API is by using third party call control, wherein an application issues higher level instructions. For example, instead of using very atomic operations such as signaling off-hook, pressing the digits etc, a JTAPI application would utilize the Call.connect() method to place a telephone call from an originating Address to a destination telephone number.

When an application performs a call control operation, the Call status changes and new Connections may be added to or removed from the Call. The application is notified about these changes using various events. To understand how to use the APIs to perform the basic operations, it is essential to know how a Call Model transitions from one state to another and what events are received by the application during this process. The Call Model used in JTAPI contains six primary objects:

- **Provider** – the telephony software entity that interfaces with a telephony subsystem.
- **Call** – the dynamic "collection of logical and physical entities" that bring two or more endpoints together.
- **Address** – a logical end-point - a "phone number".
- **Connection** – the dynamic relationship between a Call and an Address.
- **Terminal** – a physical end-point - a "phone set".
- **TerminalConnection** – the dynamic relationship between a Connection and a Terminal.

Figure 1 below shows Call Model objects and their relationships.



In Avaya's JTAPI implementation, there is a 1:1 relationship between Address and Terminal.

The relationship between the Call and a logical endpoint (Address) is called a Connection; the relationship between the Connection and a physical endpoint (Terminal) is called a TerminalConnection.

The relationship between Calls and Connections is zero-to-many. IDLE Calls have no Connections; ACTIVE Calls have at least one Connection.

Each Connection is tied to a single Address.

In Avaya's JTAPI implementation, the relationship between Connection and TerminalConnection is 1:1.

Each TerminalConnection is tied to a single Terminal.

Figure 1: Avaya's Implementation of Call Model objects' relationships

The following subsections provide an overview of the JTAPI Call and Connection objects along with the typical call flow scenarios.

1.1 Overview of the JTAPI Call and Connection Objects

This section provides the basic information about the JTAPI Call and Connection objects.

1.1.1 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. The getConnections() method of the Call object can be used to get an array of Connection objects associated with the Call. Each Connection models the relationship between a Call and an Address, where an Address identifies a particular party or a set of parties in a Call. Applications can create an instance of a Call object using the Provider.createCall() method, which returns a Call object that has zero Connections and is in the IDLE state.

A Call object has the following states:

1. Call.IDLE

2. Call.ACTIVE

3. Call.INVALID

Table 1 below provides a detailed description for each of these states:

Call State	Description
Call.IDLE	Calls are created in the IDLE state, with no Connections. This is the initial state for all Calls.
Call.ACTIVE	When a Call gets its first Connection, it becomes an ACTIVE Call, and it remains an ACTIVE Call until it has no Connections in the CONNECTED state.
Call.INVALID	Call objects which lose all of their Connections objects (via a transition of the Connection object into the Connection.DISCONNECTED state) move into INVALID state. Calls in this state may not be used for any future action.

Table 1: Call States and their descriptions

1.1.2 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 which can be accessed via the Connection.getState() method. Applications can use the Connection.getCall() and Connection.getAddress() methods to obtain the Call and the Address associated with a particular Connection,

respectively. A Connection object represents the relationship between the Call and the Address, whereas a TerminalConnection object represents the relationship between the Connection and the Terminal. The relationship between a Connection and a Terminal represents the physical view of the Call, i.e. the Terminal at which the telephone call appears. The getTerminalConnections() method of the Connection object can be used to get an array of TerminalConnection objects associated with the Connection. In Avaya's JTAPI implementation, there is a 1:1 relationship between Connection and TerminalConnection, so the getTerminalConnections() method of the Connection object will always return an array having only 1 TerminalConnection object.

Table 2 below describes the various states of the Connection object:

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 yet part of an active telephone call.
Connection.DISCONNECTED	This state implies that the Connection is no longer part of the telephone Call, although the Connection's references to Call and Address still remain valid.
Connection.INPROGRESS	This state implies that the Connection representing the destination end of the telephone call is in the process of contacting the destination side.
Connection.ALERTING	This state implies that the corresponding Address is being notified of an incoming call.
Connection.CONNECTED	This state implies that a Connection has reached its final active state in the telephone call.
Connection.UNKNOWN	This state implies that the AE services JTAPI service 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 representing the corresponding endpoint of the call has failed.

Table 2: Connection States

1.2 Call Model State Transitions, Requests and Events

A Call object models a telephone call. A basic two-party telephone call is represented by a Call having two Connection objects – a Connection for the originating Address and a Connection for the destination Address. Before a telephone call is established between two parties or endpoints, Connection objects representing both the originating and the destination Address go through various state transitions. The JTAPI event system notifies applications when changes in various JTAPI objects occur. Each individual change in an object is represented

by an event sent to the appropriate Observer. Because several changes may happen to an object at once, events are delivered as a batch. A batch of events represents a series of events and changes to the call model which happened exactly at the same time. For this reason, events are delivered to observers as arrays.

The Call Control Package provides more detailed information about the call model using an extended set of states on the Connection and TerminalConnection objects. **Figure 2** below shows a timeline diagram of the state transitions for Connections and TerminalConnections resulting from a Call.connect() invocation. This diagram presents the outcome of placing a telephone call in terms of the Call Control states. Only Call Control states are shown to keep the discussion simple. Please refer to *Avaya Java Telephony API (JTAPI) javadoc* (reference [2]) for the complete list of Connection and TerminalConnection states. In the diagram below, time is the vertical axis where time increases going down the page. The solid vertical lines denote various objects: Call, Terminal, and Address. The solid horizontal lines between the Call and the Address represent a Connection. The dotted horizontal lines between the Address and the Terminal represent TerminalConnections. More than one change may occur to the call model during a single discrete step; the application sees these changes as happening all at once.

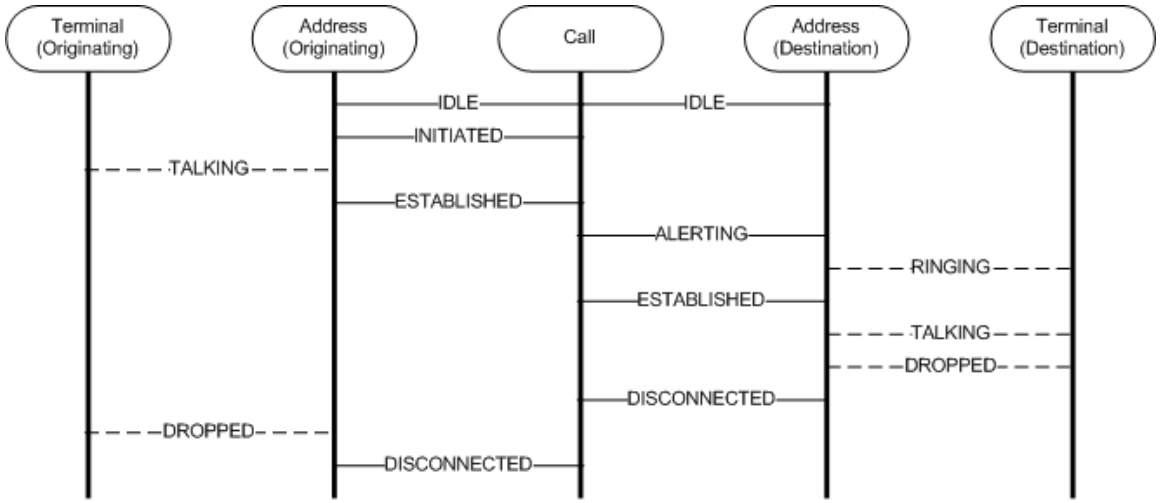


Figure 2: Timeline diagram of the Connections and TerminalConnections state transitions

The following subsections describe the state transition for each Connection (originating and destination) and the corresponding events generated. Only events extended from the CallCtlConnEv interface are covered to keep the discussion simple. CallCtlConnEv is the base interface for all call control package Connection-related events. JTAPI applications can receive these events by registering a Call Observer for an originating Address/Terminal or a destination Address/Terminal. Please refer to *Avaya Java Telephony API (JTAPI) javadoc* (reference [2]) for the complete list of events. There can be state transitions for the call model objects other than the ones described in this section.

1.2.1 State Changes and Events for the Originating Connection

Figure 3 below shows the Call Control state changes for the originating connection and the corresponding events generated when the state transition occurs.

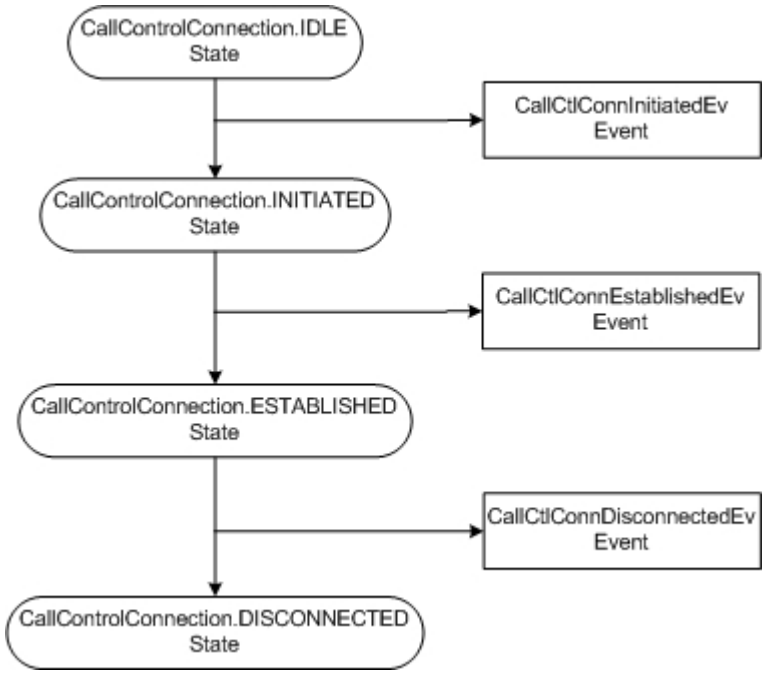


Figure 3: Call Control state changes and events generated for the originating Connection

The following steps outline the state transitions and the corresponding events generated for the originating Connection as shown in **Figure 3** above:

1. When a Connection is created, the Connection is in the CallControlConnection.IDLE state.

2. The originating Connection first moves to the CallControlConnection.INITATED state and the CallCtlConnInitiatedEv event is generated by the JTAPI event system. This state indicates the originating end of a telephone call has begun the process of placing a telephone call, but the dialing of the destination telephone address has not yet begun. Typically, a telephone associated with the originating Address has gone “off-hook”.

3. The originating Connection then moves to the CallControlConnection.ESTABLISHED state and the CallCtlConnEstablishedEv event is generated by the JTAPI event system. This state indicates the originating end of a telephone call has completed the process of dialing the destination telephone address. The originating end of the telephone call is now talking and the connection to the destination telephone is in progress.

4. When the telephone call is dropped by going "on-hook" or by programmatically calling Connection. disconnect(), the originating Connection moves to the CallControlConnection.DISCONNECTED state and the CallCtlConnDisconnectedEv event is generated by the JTAPI event system.

Note: Please note that CallCtlConnInitiatedEv will be received by the application only when the Call Observer is associated with the originating Address or Terminal.

1.1.2 State Changes and Events for the Destination Connection

Figure 4 below shows the Call Control state changes for the destination connection and the corresponding events generated when the state transition occurs.

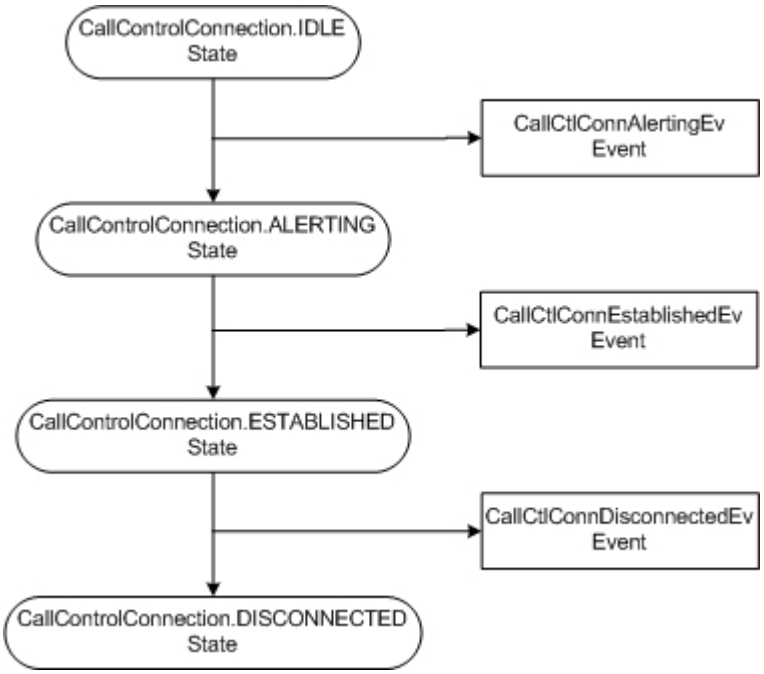


Figure 4: Call Control state changes and events generated for the destination Connection

The following steps outline the state transitions and the corresponding events generated for the destination Connection as shown in **Figure 4** above:

1. When a Connection is created, the Connection is in the CallControlConnection.IDLE state.

2. The destination Connection first moves to the CallControlConnection.ALERTING state and the CallCtlConnAlertingEv event is generated by the JTAPI event system. This state indicates the destination Address is being notified of an incoming call.

3. When the telephone call is answered either manually by going "off-hook" or programmatically by calling the TerminalConnection.answer() method, the destination Connection moves to the CallControlConnection. ESTABLISHED state and the CallCtlConnEstablishedEv event is generated by the JTAPI event system.

4. When the telephone call is dropped by going "on-hook" or by programmatically calling Connection. disconnect(), the destination Connection moves to the CallControlConnection.DISCONNECTED state and the CallCtlConnDisconnectedEv event is generated by the JTAPI event system.

Section 2: Telephony Operations

This chapter describes in detail how to perform basic telephony operations using the call control services provided in the Avaya JTAPI SDK. The call control services can be used to make, answer and terminate calls. The following subsections detail the procedures involved in carrying out these operations.

Note: All the code snippets in this chapter assume that the application initialization has already been performed and that a Call Observer is registered for the Address or Terminal object representing the device. When a Call Observer is registered, all subsequent Call and Connection related events are reported in the callback method callChangedEvent(). For understanding the basic steps of application initialization and registering Observers, please refer to the document *Tutorial for Application Initialization using Avaya JTAPI SDK* (reference [1]).

2.1 Event Information

In JTAPI, the call model state is represented by the internal state of call model objects. During a call's lifecycle, call model objects go through various state transitions. There are events generated for call model objects' state transitions and JTAPI applications can receive these events by registering Call Observers for an Address or Terminal. This section covers the general information that is included in the events. Any other information that is specific to a particular event will be covered in the corresponding section describing that event.

- **Call:** Specifies the Call object associated with the event. This information is available in the event that extends the CallEv interface. The getCall() method of the event object can be used to get the Call object associated with the event.

- **Connection:** Specifies the Connection object associated with the event. This information is available in the event that extends the ConnEv interface. The getConnection() method of the event object can be used to get the corresponding Connection object.

- **Calling party Address:** Specifies the Address of the calling party. This information is available in the event that extends the CallCtlCallEv interface. The Address object representing the calling address can be retrieved by using the getCallingAddress() method of the event object.

- **Calling party Terminal:** Specifies the Terminal of the calling party. This information is available in the event that extends the CallCtlCallEv interface. The Terminal object representing the calling terminal can be retrieved by using the getCallingTerminal() method of the event object.

- **Called party Address:** Specifies the Address of the called party. This information is available in the event that extends the CallCtlCallEv interface. The Address object representing the called address can be retrieved by using the getCalledAddress() method of the event object.

- **ID:** Each event type has an ID. The integer value representing the ID of the event can be retrieved by using the getID() method of the event object. ID allows applications to switch on event id rather than having to use multiple "if instanceof" statements to determine the event received.

- **Meta Code:** Specifies the Meta Code associated with the event. The getMetaCode() method of the event object can be used to get the Meta Code of the event. Events are grouped together using Meta Codes to provide a higher-level description of an update to the call model. Since events represent singular changes in one particular object in the call model, it may be difficult for the application to infer a higher-level interpretation of several of these singular events. Meta Codes exist on events to assist the application in this regard. For more information on Meta Codes, please refer to *Avaya Java Telephony API (JTAPI) javadoc* (reference [2]).

- **Last Redirected Address:** The last redirected Address (extension) associated with this Call. The last redirected Address is the Address from which the current telephone call was redirected before coming to the current Address. This is common if a Call is forwarded to several Addresses before being answered. This information is available in the event that extends the CallCtlCallEv interface. The getLastRedirectedAddress() method of the event object can be used to get the last redirected Address associated with the call.

- **Cause:** Specifies the cause associated with the event. The getCause() method of the event object can be used to get the cause associated with the event. For a normal operation, the value will be set to CAUSE_NORMAL. For more information on the causes associated with the various events, please refer to *Avaya Java Telephony API (JTAPI) javadoc* (reference [2]).

Note: Please note that this common information mentioned above is not available in events of all types. The application should check for the event type, if not already known, by using "if instanceof", before using the method to get the corresponding information.

2.2 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 the CallCtlConnAlertingEv event. The application can detect an incoming call by monitoring for the CallCtlConnAlertingEv event. This event does not contain any additional information other than the event information covered in section 2.1.

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

```
public void callChangedEvent(CallEv[] eventList)
{
	String callingDeviceID; // save calling number
	String calledDeviceID; // save called number
	for ( int eventIndex = 0; eventIndex < eventList.length; eventIndex++)
	{
		CallEv e = eventList[eventIndex];
		// Check whether the event is of type CallCtlConnEv.
		if ( e instanceof CallCtlConnEv )
		{
			switch ( e.getID() )
			{
				case CallCtlConnAlertingEv.ID :
				{
					// save Call object for future reference
					Call call = e.getCall();
					CallCtlConnEv ccEv = (CallCtlConnEv)e;
					if(ccEv.getCallingAddress() != null)
					{
						callingDeviceID =
							ccEv.getCallingAddress().getName();
					}
					if(ccEv.getCalledAddress() != null)
					{
						calledDeviceID =
							ccEv.getCalledAddress().getName();
					}
					System.out.println("Incoming call from " +
					    callingDeviceID+ " to " + calledDeviceID);
					/* Handle incoming call here */
				}// End of case
			}// End of switch
		}// End of if
	}// End of for
```

Code Snippet 1

2.3 Answering a Call

2.3.1 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 as shown in **Figure 4** above.

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

```
Call mycall;                // preserved by the application upon receiving new
                            // incoming call event. See section 'Error! Reference
source not found.' to understand
                            // how to detect an incoming call.

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
}
```

Code Snippet 2

2.3.2 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 Call Observer for either the Terminal or Address of the originating or destination station extension, the application will receive a CallCtlConnEstablishedEv event for the destination Connection as shown in **Figure 4**. This event does not contain any additional information other than the event information covered in section **2.1**.

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

```
public void callChangedEvent(CallEv[] eventList)
{
	for (int eventIndex = 0; eventIndex < eventList.length; eventIndex++)
	{
		CallEv e = eventList[eventIndex];
		// Check whether the event is of type CallCtlConnEv.
		if (e instanceof CallCtlConnEv)
		{
			switch (e.getID())
			{
				case CallCtlConnEstablisedEv.ID :
				{
					// Get the Connection object which entered the
					// ESTABLISHED state.
					CallCtlConnEv ccEv = (CallCtlConnEv)e;
					Connection conn = ccEv.getConnection();
					System.out.println("Connection for Address " +
						conn.getAddress().getName() + " is in " +
						"Established state.");

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

				}// End of Case
			}// End of switch
		} // End of if
	} // End of for
}
```

Code Snippet 3

2.4 Originating a Call

2.4.1 Triggering a Call from the Application

The application may need to originate calls. This can be done using the Call.connect() method.

A new call is setup using a two-step process:

1. **Create a Call object:**

   The application first needs to create a Call object for storing the Connections. This is done by using the Provider.createCall() method. The createCall() method returns a new instance of a Call object with the help of the Provider object obtained during the application initialization. The new Call gets created in the IDLE state and contains no Connections associated with it. This new Call object is then used to originate the call.

2. **Use the Connect method:**

   The application needs to initiate the call using the Call.connect() method. The Call.connect() method places a telephone call from an originating Address to a destination telephone number.

   The Call.connect() method takes the following arguments:

   - **Originating Terminal for the call:** Specifies the Terminal object representing the calling terminal. A Terminal object for a station can be retrieved by using the getTerminal() method of the Provider.
   - **Originating Address of the call:** Specifies the Address object representing the calling party. An Address object for a station can be retrieved by using the getAddress() method of the Provider.
   - **Destination:** A string whose value represents the address (phone number, possibly including/exclusively a feature access code) to which the call is to be made.

   This method returns an array of Connection objects that are created when the method is invoked with the above-mentioned arguments. One Connection object represents the originating party and the other represents the destination. These two Connections will initially be in the Connection.IDLE state. Once the Provider begins making a call, the originating Connection moves from the Connection.IDLE state into the Connection.CONNECTED (or CallControlConnection.ESTABLISHED) state. The destination Connection first moves from the Connection.IDLE state to the Connection.ALERTING (or CallControlConnection.ALERTING) state when the call is presented at the terminal, and then to the Connection.CONNECTED (or CallControlConnection.ESTABLISHED) state if and when the destination terminal answers the call.

The following code snippet shows the procedure to invoke the Call.connect() method.

Note: The following code handles InvalidPartyException. Other code can be added to handle the other exceptions. For more information on the type of exceptions that can be thrown from the Call.connect() method, please refer to *Avaya Java Telephony API (JTAPI) javadoc* (reference [2]).

```
/*
 * Places a telephone call from extension 40061 to extension 40062
 */

Address origaddr = null;
Terminal origterm = null;

Provider myprovider; // myprovider is the Provider object obtained during
                     // application initialization

origaddr = myprovider.getAddress("40061");
Terminal[] terminals = origaddr.getTerminals();

/* There is a one-to-one relationship between Address and Terminal
 * objects in Avaya's JTAPI implementation.
 * Hence only one terminal object will be in the array.
 */
origterm = terminals[0];
Call mycall = null;
mycall = myprovider.createCall();

// Place the telephone call.
if (origterm != null && origaddr != null && mycall != null)
{
       try
       {
              Connection c[] = mycall.connect(origterm, origaddr, "40062");
       }

       // Invalid Party exception
       catch (InvalidPartyException excp)
       {
              System.out.println("Originating or destination party is invalid "+ e);
       }
       //Handle other exceptions
}
```

Code Snippet 4

2.4.2 Events Received When a Call is Originated

A call can be initiated either by using a physical device or programmatically as explained in the previous section. In both cases, an application having the Call Observer registered for the originating/destination Terminal or Address will receive various events for the Connections' state transitions. Sections *'State Changes and Events for the Originating Connection'* and *'State Changes and Events for the Destination Connection'* provide more information about the events received for originating and destination Connection's state transitions respectively. These events do not contain any additional information other than the event information covered in section *2.1.*

2.5 Placing and Removing a Call from Hold

2.5.1 Triggering a Hold/Un-hold from the Application

Holding a call can be done so the holding party can handle some other call or have their audio path disconnected from the call. Hold is also a precursor to a conference or transfer operation. When the party wishes to reconnect to the call, they can unhold (a.k.a. retrieve) the call. When putting a call on hold or retrieving a held call, the application needs to use the CallControlTerminalConnection object for the corresponding Terminal for which the hold/unhold operation needs to be performed. The CallControlTerminalConnection interface extends the core TerminalConnection interface with additional features and greater detail about the TerminalConnection state. Applications may query a TerminalConnection object using the instanceof operator to see whether it supports this interface.

- **Hold:** While performing a transfer or conference operation, the application needs to put the active call on hold and then originate a new call. The application can put the call on hold by using the hold() method of the corresponding CallControlTerminalConnection object. After invoking the hold() method of the CallControlTerminalConnection object, the CallControlTerminalConnection state of the TerminalConnection changes from CallControlTerminalConnection. TALKING to CallControlTerminalConnection.HELD. The CallControlTerminalConnection.HELD state indicates that a Terminal is part of a Call, but is on hold. Other Terminals which are on the same Call and associated with the same Connection may or may not also be in this state.

- **Unhold:** The application may need to unhold (i.e. retrieve) a held call. This is achieved by using the unhold() method of the corresponding CallControlTerminalConnection object. After invoking the unhold() method of the CallControlTerminalConnection object, the CallControlTerminalConnection state of the TerminalConnection changes from CallControlTerminalConnection.HELD to CallControlTerminalConnection.TALKING. The CallControlTerminalConnection.TALKING state indicates that the Terminal is part of a Call, is typically "off-hook", and the party is communicating on the telephone call.

The following steps are involved in placing/removing a call on/from hold:

Note: To place a call on hold, the call must be active, i.e., the call must be in the Call.Active state and the associated CallControlTerminalConnection must be in the CallControlTerminalConnection.TALKING state. A call can be removed from a held state only if no other CallControlTerminalConnection associated with other calls on the same terminal is in the CallControlTerminalConnection.TALKING state.

1. **Obtain the Connection list:** There are two ways for obtaining the array of Connection objects:
   - getConnections() method of Address object: This method returns array of Connection objects associated with the specified Address. An Address may be a participant in more than one Call, and thus have multiple Connections.
   - getConnections() method of Call object: This method returns array of Connection objects associated with the specified Call. A Call typically has two or more Connections.

   Code Snippet 5 uses a Call object to get the array of Connection objects and find the Connection matching address "40061".

2. **Get the Array of TerminalConnection objects:** From the Connection object, obtain the array of TerminalConnection objects by using the getTerminalConnections() method of the Connection object.

3. **Putting the call on hold/un-hold for specific TerminalConnection objects:** From the array of TerminalConnection objects, find the object that is an instance of CallControlTerminalConnection and has the Terminal with an Address that matches the Address of the Connection to be put on hold/unhold. Call the hold()/unhold() method of the CallControlTerminalConnection object to put the TerminalConnection on hold/unhold.

The following code snippet shows how to perform the hold/un-hold operations using the TerminalConnection.

```
/* Step 1: Obtain the connection list from the call object and find the
 * connection for the desired Address
 */
Call mycall; // preserved by the application upon receiving new incoming call event.
             // See section 'Error! Reference source not found.' to understand how to
detect
             // an incoming call.
Connection connection[] = mycall.getConnections();
String myAddressName = "40061";          //stores the name of the address
Connection localConn = null;

for (int connectionIndex = 0; connectionIndex < connection.length; connectionIndex++)
{

       Connection conn = connection[ connectionIndex ];
       String name = conn.getAddress().getName();

       if (name.equals(myAddressName))
       {
              localConn = conn;
              break;
       }
}

if (localConn == null)
       return;

/* Step 2:Get the TerminalConnections list for the connection obtained in step 1 */

TerminalConnection[] termConns = localConn.getTerminalConnections();

for (int  termConIndex = 0;  termConIndex < termConns.length;  termConIndex++)
{

       TerminalConnection termConn = termConns[ termConIndex ];

       // Verify that the TerminalConnection is an instance of
       // CallControlTerminalConnection
       if (termConn instanceof CallControlTerminalConnection)
       {
              String name = termConn.getTerminal().getName();

              if (name.equals(myAddressName))
              {
                   CallControlTerminalConnection ccTermConn =
                     (CallControlTerminalConnection) termConn
                     /* Step 3:Hold/unhold the selected TerminalConnection */
                     // To hold the call
                            if (ccTermConn.getCallControlState() ==
                                   CallControlTerminalConnection.TALKING)
                             ccTermConn.hold();
                     // To un-hold the call
                            if (ccTermConn.getCallControlState() ==
                                   CallControlTerminalConnection.HELD)
                             ccTermConn.unhold();
                      break;
              } // End of if
       } // End of if
} // End of for
```

Code Snippet 5

2.5.2 Events Received When a Call is Held

The following is a brief description of the events that are received when a call is put on hold or unheld:

- **CallCtlTermConnHeldEv:** When a hold operation is performed, the CallControlTerminalConnection state changes from CallControlTerminalConnection.TALKING to CallControlTerminalConnection.HELD and the application receives the CallCtlTermConnHeldEv event. The CallControlTerminalConnection state can be retrieved by calling the getCallControlState() method of the CallControlTerminalConnection object.
- **CallCtlTermConnTalkingEv:** When an unhold operation is performed on an already held connection, the CallControlTerminalConnection state changes from CallControlTerminalConnection.HELD to CallControlTerminalConnection.TALKING and the application receives the CallCtlTermConnTalkingEv event.

The following code snippet demonstrates how to handle these events:

```
public void callChangedEvent(CallEv[] eventList)
{
	for (int eventIndex = 0; eventIndex < eventList.length; eventIndex++)
	{
		CallEv e = eventList[eventIndex ];
		// Check whether the event is of type TermConnEv.
		if ( e instanceof TermConnEv )
		{
			switch (e.getID())
			{
				case CallCtlTermConnTalkingEv.ID:
				{
					// The TerminalConnection state has changed to
					// CallControlTerminalConnection.TALKING state
					TermConnEv tcEv = (TermConnEv)e;
					// Get the TerminalConnection
					TerminalConnection tc = tcEv.getTerminalConnection();
					System.out.println("TerminalConnection for " +
								"Terminal " +
								tc.getTerminal().getName() +
								" is Talking");
					break;
				}//End of Case

				case CallCtlTermConnHeldEv.ID:
				{
					// The TerminalConnection state has changed to
					// CallControlTerminalConnection.HELD state
					TermConnEv tcEv = (TermConnEv)e;
					// Get the TerminalConnection
					TerminalConnection tc = tcEv.getTerminalConnection();
					System.out.println("TerminalConnection for " +
								"Terminal " +
								tc.getTerminal().getName() +
								" is put on Hold.");
					break;
				}// End of Case
			}// End of switch
		} // End of if
	} // End of For
}
```

Code Snippet 6

The CallCtlTermConnTalkingEv and CallCtlTermConnHeldEv events provide the following additional information along with the event information covered in section 2.1.

- **TerminalConnection:** Specifies the TerminalConnection associated with the event. The getTerminalConnection() method is used to get the TerminalConnection object.

- **Call Control Cause:** Specifies the call control package cause associated with the event. This information is available in the event that extends the CallCtlEv interface. The getCallControlCause() method of the event object can be used to get the call control cause associated with the event. For more information on call control causes associated with the various events, please refer to *Avaya Java Telephony API (JTAPI) javadoc* (reference [2]).

2.6 Disconnecting a Call

2.6.1 Triggering a Disconnection from the Application

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

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

There are multiple ways a connection can be moved to the Connection.DISCONNECTED state:

1. The end user can leave the call by manually going on-hook, resulting in the corresponding Connection moving to the Connection.DISCONNECTED state.

2. If auto-hold is not enabled, the end user can select some other call appearance triggering disconnection from the active call.

3. The user can use a button such as ‘release’ to trigger disconnection from the active call.

4. The application can programmatically disconnect a Connection by invoking the Connection.disconnect() method.

5. The connection may be disconnected by the far end party, triggering disconnection of the near end (application monitored) Connection when the disconnection leaves only one remaining active or held Connection.

The state of the connection changes to the Connection.DISCONNECTED state after the connection is disconnected using any of the above mentioned methods.

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; // myprovider is the Provider object obtained during
                     // application initialization
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;
        }
}

if(localConn == null) // cannot locate a connection in Connection.CONNECTED state with
                      // the specified Address in it.
       return;

try
{
       localConn.disconnect();
}

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

Code Snippet 7

2.6.2 Events Received when a Connection is disconnected

The following event is received when a connection is disconnected:

- CallCtlConnDisconnectedEv: This event is received when any of the parties in a call gets disconnected from the call. After receiving this event, the associated Connection moves to the Connection.DISCONNECTED state.

This event does not contain any additional information other than the event information covered in section 2.1.

References

[1] *Avaya DevConnect Tutorial for Application Initialization using Avaya JTAPI SDK.*

[2] *Avaya Java Telephony API (JTAPI) javadoc, version 4.2*

Both documents are available on the Avaya DevConnect Portal (**http://www.avaya.com/devconnect**).

ABOUT DEVCONNECT

The Avaya DevConnect Program provides a wide range of developer-oriented resources from Avaya, including access to APIs and SDKs, developer tools, technical support options and training materials. Registered membership is free to anyone interested in designing Avaya-compatible solutions. Enhanced Membership options offer increased levels of technical support, compliance testing, and co-marketing of innovative solutions compatible with standards-based Avaya solutions.

Please e-mail any questions or comments pertaining to this tutorial along with the full title name and filename, located in the lower right corner, directly to the Avaya DevConnect Program at **devconnect@avaya.com**.

About Avaya

Avaya is a global leader in enterprise communications systems. The company provides unified communications, contact centers, and related services directly and through its channel partners to leading businesses and organizations around the world. Enterprises of all sizes depend on Avaya for state-of-the-art communications that improve efficiency, collaboration, customer service and competitiveness. For more information please visit www.avaya.com.

AVAYA

INTELLIGENT COMMUNICATIONS

02/09 • DEV4158

avaya.com