The Java Telephony API

An Overview

January 28, 1997
Version 1.1

Introduction

The **Java Telephony API** (**JTAPI**) is a portable, object-oriented application programming interface for Java-based computer-telephony applications. JTAPI serves a broad audience, from call center application developers to web page designers. JTAPI supports both first- and third-party telephony application domains. The API is designed to make programming simple applications easy, while providing those features necessary for advanced telephony applications.

The Java Telephony API is, in fact, a set of API's. The "core" API provides the basic *call model* and rudementary telephony features, such as placing telephone calls and answering telephone calls. The core API is surrounded by standard extension API's providing functionality for specific telephony domains, such as call centers and media stream access. The JTAPI core and extension package architectures are described later in this document.

Applications written using the Java Telephony API are portable across the various computer platforms and telephone systems. Implementations of JTAPI will be available for existing computer-telephony integration platforms such as Sun Microsystem's SunXTL, Microsoft and Intel's TAPI, Novell and Lucent's TSAPI, and IBM's CallPath. Additionally, independent hardware vendors may choose to provide implementations of the Java Telephony API on top of their own proprietary hardware.

Overview Document Organization

This document is organized into the following sections:

Java Telephony API Features	Describes the features of JTAPI and the principles on which it was designed.
Supported Configurations	Summarizes the environments in which JTAPI may be used and the computer and sofware configurations for which it was designed.
Java Telephony Package Architecture	Summarizes how the Java Telephony API is organized into various Java language packages. A brief description accompanies each package along with links to more detailed documentation.
The Java Telephony Call Model	Describes how telephone calls and different objects that make up telephone calls are represented in this API.
Core Package Methods	Provides a brief summary of the key methods available in the core package which perform the most basic telephony operations, such as placing a telephone call, answering a telephone call, and dropping a telephone call.
Connection Object States	Describes the states in which the Connection object can exist. It provides a description of the allowable transitions from each state.
TerminalConnection Object States	Describes the states in which the TerminalConnection object can exist. It provides a description of the allowable transitions from each state.
Placing a Telephone Call	One of the most common features used in any telephony API is placing a telephone call. This section describes the JTAPI method invocations required to place a telephone call, while examining the state changes the call model undergoes. This analysis will begin with placing the telephone call at an originating phone. It will progress through a called phone answering and end with the termination of the call.
The Java Telephony Observer Model	Describes the JTAPI observer model. Applications use observers for asynchronous notification of changes in the state of the JTAPI call model.
Application Code Examples	Provides two real-life code examples using the Java Telephony API. One places a telephone call to a specified telephone number. There other answers incoming telephone calls to a designated Terminal.
Locating and Obtaining Providers	Describes the manner in which applications create and obtain JTAPI Provider objects.
Security	Summarizes the JTAPI security strategy.

History of the Java Telephony API

The Java Telephony API specification represents the combined efforts of design teams from Sun Microsystems, Lucent, Nortel, Novell, Intel, and IBM, operating under the direction of JavaSoft.

The Java Telephony API version 1.0 specification was release to the public on November 1, 1996.

This version of the specification, version 1.1 is being release to the public on February 1, 1997.

Java Telephony Features

The features and guiding design principles for the Java Telephony API are:

- Brings simplicity to the simple and most basic telephony applications
- Provides a scalable framework that spans desktop applications to distributed call center telephony applications
- Interfaces applications directly to service providers or acts as a Java interface to existing telephony APIs, such as SunXTL, TSAPI, and TAPI
- Based on a simple core that is augmented with standard extension packages
- Runs on a wide range of hardware configurations, wherever Java run-time can be used

Supported Configurations

JTAPI runs on a variety of system configurations, including centralized servers with direct access to telephony resources, and remote network computers which access telephony resources over a network. In the first configuration, a network computer is running the JTAPI application and is accessing telephony resources over the a network, as illustrated in Figure 1. In the second configuration, the application is running on a computer with its own telephony resources, as illustrated in Figure 2.

Network Computer (NC) Configuration

In a network configuration, the JTAPI application or Java applet runs on a remote workstation. This workstation can be a network computer with only a display, keyboard, processor, and some memory. It accesses network resources, making use of a centralized server that manages telephony resources. JTAPI communicates with this server via a remote communication mechanism, such as Java's Remote Method Invocation (RMI), JOE, or a telephony protocol. The following diagram shows this configuration.

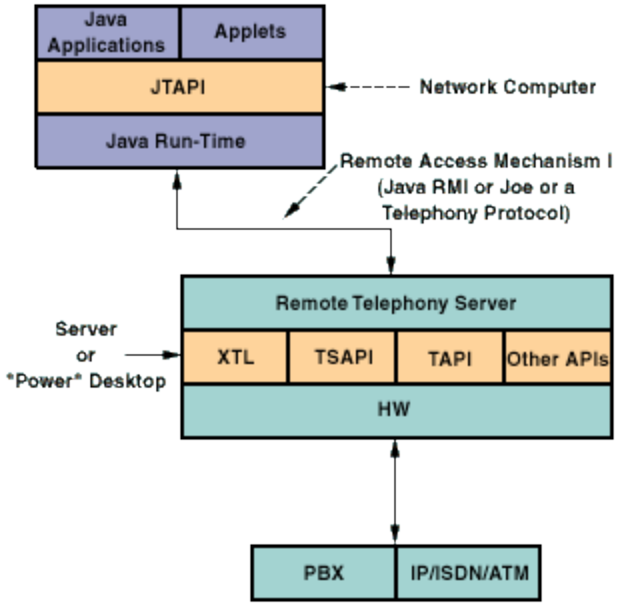

Figure 1: Network Configuration

Desktop Computer Configuration

In a desktop configuration, the JTAPI application or Java applet runs on the same workstation that houses the telephony resources. The following diagram shows the desktion configuration.

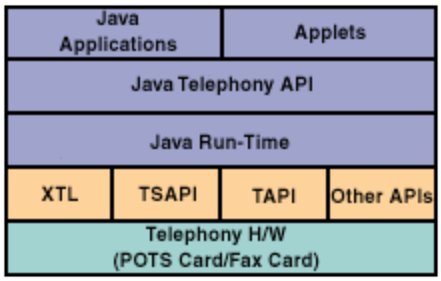

Figure 2: Desktop Configuration

Java Telephony Package Architecture

The Java Telephony API is composed of a set of Java language *packages*. Each package provides a specific piece of functionality for a certain aspect of computer-telephony applications. Implementations of telephony servers choose the packages they support, depending upon the capabilities of their

underlying platform and hardware. Applications may query for the packages supported by the implementation they are currently using. Additionally, applications may concern themsevles with only the supported packages it needs to accomplish its tasks. The diagram below depicts the architecture of the JTAPI packages.

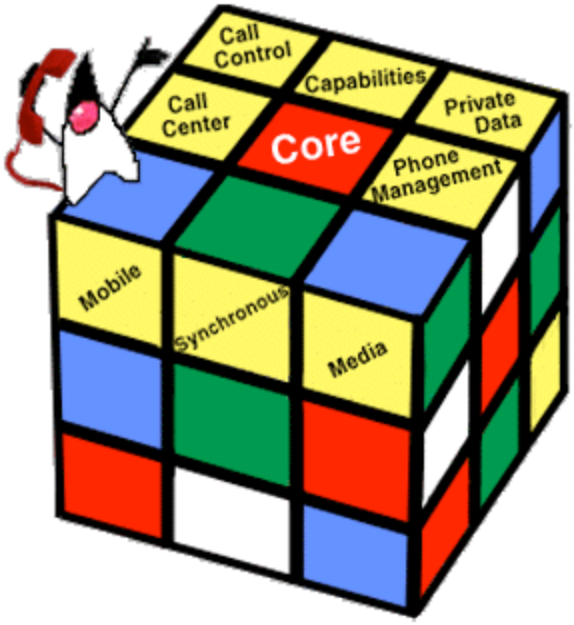

Figure 3: Core/Extension Package Relationship

At the center of the Java Telephony API is the "core" package. The core package provides the basic framework to model telephone calls and rudementary telephony features. These features include placing a telephone call, answering a telephone call, and dropping a telephone call. Simple telephony applications will only need to use the core to accomplish their tasks, and do not need to concern themselves with the details of other packages. For example, the core package permits applet designers to add telephone capablities to a Web page with ease.

Layered around the JTAPI core package are a number of "standard extension" packages. These extension packages each bring additional telephony functionality to the API. Currently, the following extension packages exist for this API: call control, call center, media, phone, private data, and capabilities packages. Each package is summarized below in terms of the features it brings to JTAPI, and is linked to a separate overview document and specifications.

The JTAPI package architecture is a two-way street for both implementations and applications. In other words, telephony server implementations choose which extension packages (in addition to the core package) they implement, based upon the capabilities of the underlying hardware. Also, applications choose which extension packages (in addition to the core package) they need to use to accomplish the desired tasks of the application. Applications may query the implementation for the extension packages it supports, and the application developer does not need to concern himself/herself

with the details of those packages not needed for the application.

Java Telephony Standard Extension Packages

Each JTAPI extension package has its own specification which augments the core API, and in most cases has its own separate overview document describing it. The chart below lists each extension package available, with a link to the individual overview document, if it exists.

Call Control Package	The ***java.telephony.callcontrol*** package extends the core package by providing more advanced call-control features such as placing calls on hold, transferring telephone calls, and conferencing telephone calls. This package also provides a more detailed state model of telephone calls.
Call Center Package	The ***java.telephony.callcenter*** package provides applications the ability to perform advanced features necessary for managing large call centers. Example of these advanced features include: Routing, Automated Call Distribution (ACD), Predictive Calling, and associating application data with telephony objects.
Media Package	The ***java.telephony.media*** Package provides applications access to the media streams associated with a telephone call. They are able to read and write data from these media streams. DTMF (touch-tone) and non-DTMF tone detection and generations is also provided in the java.telephony.media package
Phone Package	The ***java.telephony.phone*** package permits applications to control the physical features of telephone hardware phone sets. Implementations may describe Terminals as collections of components, where each of these component-types have interfaces in this package.
Capabilities Package	The ***java.telephony.capabilities*** package allows applications to query whether certain actions may be performed. Capabilities takes two forms: *static* capabilities indicate whether an implementation supports a feature; *dynamic* capabilities indicate whether a certain action is allowable given the current state of the call model.
Private Data Package	The ***java.telephony.privatedata*** package enables applications to communicate data directly with the underlying hardware switch. This data may be used to instruct the switch to perform a switch-specific action. Additionally applications may use this package to "piggy-back" a piece of data with a Java Telephony API object.

The Java Telephony Call Model

The JTAPI *call model* consists of a half-dozen Java object. These objects are defined using Java interfaces in the core package. Each call model object represents either a physical or conceptual entity in the telephone world. The primary purpose of these call model objects is to describe telephone calls and the endpoints involved in a telephone call. These call model objects are related to one another in specific ways, which is summarized below and described in more detail in the core package

specification.

The following diagram shows the JTAPI call model and the objects which compose the call model. A description of each object follow the diagram.

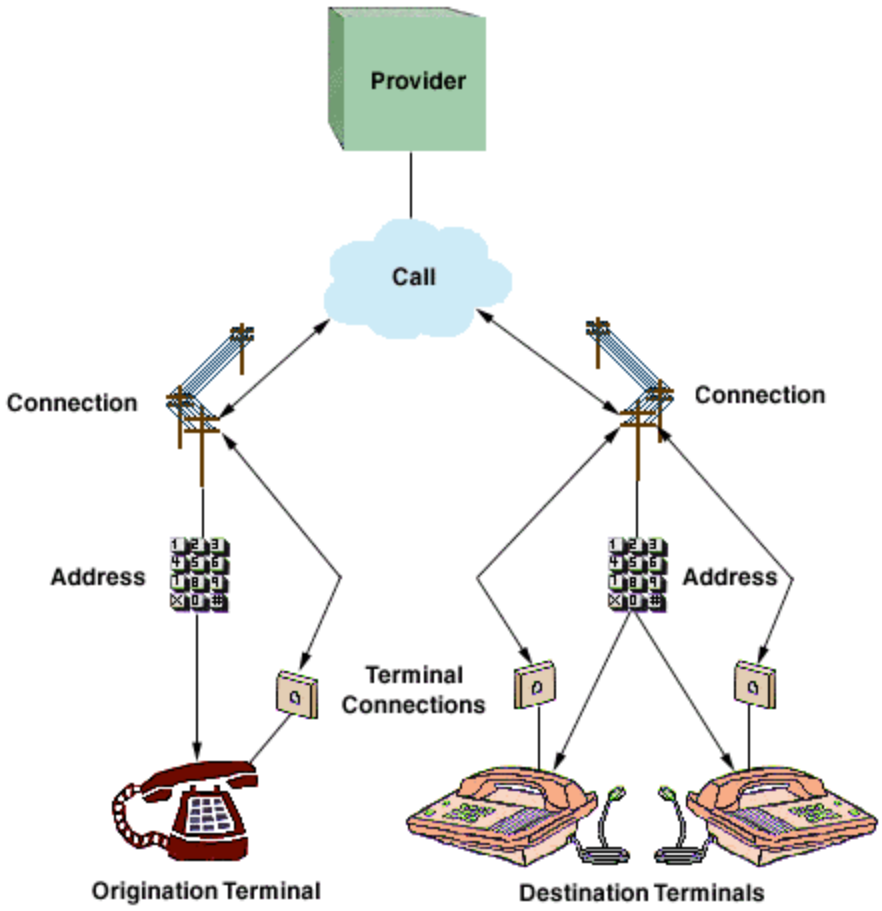

Figure 4: JTAPI Call Model

Provider Object

The Provider object is an abstraction of telephony service provider software. The provider might manage a PBX connected to a server, a telephony/fax card in a desktop machine, or a computer networking technology, such as IP. 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.

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 phone 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 logcal 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 corresponds to a physical piece of hardware.

- **TerminalConnection Object**

  TerminalConnection objects model the relationship between a call 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 call (in contrast to the Connection, which models the logical view). The TerminalConnection describes the current state of relationship between the Call and a particular Terminal. The states associated with the TerminalConnection are described later in this document.

Core Package Methods

The core package defines three methods to support its primary features: placing a telephone call,

answering a telephone call, and dropping a telephone call. These methods are **Call.connect()**, **TerminalConnection.answer()**, and **Connection.disconnect()**, respectively.

- **Call.connect()**

  Once an application has an idle call object (obtained via **Provider.createCall()**), it may place a telephone call using the **Call.connect()** method. The application must specify the originating Terminal (physical endpoint) and the originating Address (logical endpoint) on that Terminal (in the case that a Terminal has multiple telephone numbers on it). It also provides the destination telephone number string. Two Connection objects are returned from the **Call.connect()** method, representing the originating and destination ends of the telephone call.

- **TerminalConnection.answer()**

  Applications monitor with observers (discussed later) on Terminal for when incoming calls are presented. An incoming telephone call to a Terminal is indicated by a TerminalConnection to that Terminal in the RINGING state (see TerminalConnection states below). At that time, applications may invoke the **TerminalConnection.answer()** to answer that incoming telephone call.

- **Connection.disconnect()**

  The **Connection.disconnect()** method is used to remove an Address from the telephone call. The Connection object represents the relationship of that Address to the telephone call. Applications typically invoke this method when the Connection is in the CONNECTED state, resulting in the Connection moving to the DISCONNECTED state. In the core package, application may only remove entire Addresses from the Call, and all of the Terminals associated with that Address which are part of the call are removed as well. The call control extension package provides the ability for application to remove individual Terminals only from the Call.

Connection Object States

A Connection object is always in a *state* which reflects the current relationship between a Call and an Address. The state in which a Connection exists is not only important to the application for information purposes, it is always an indication of which methods and actions can be invoked on the Connection object.

The state changes which Connection objects undergo are governed by rules shown below in a state transition diagram. This diagram guarantees to application developers the possible states in which the Connection object can transition given some current state. These state transition rules are invaluable to application developers. The diagram below shows the possible state transitions for the Connection object. Following this diagram is a brief summary of the meaning of each state.

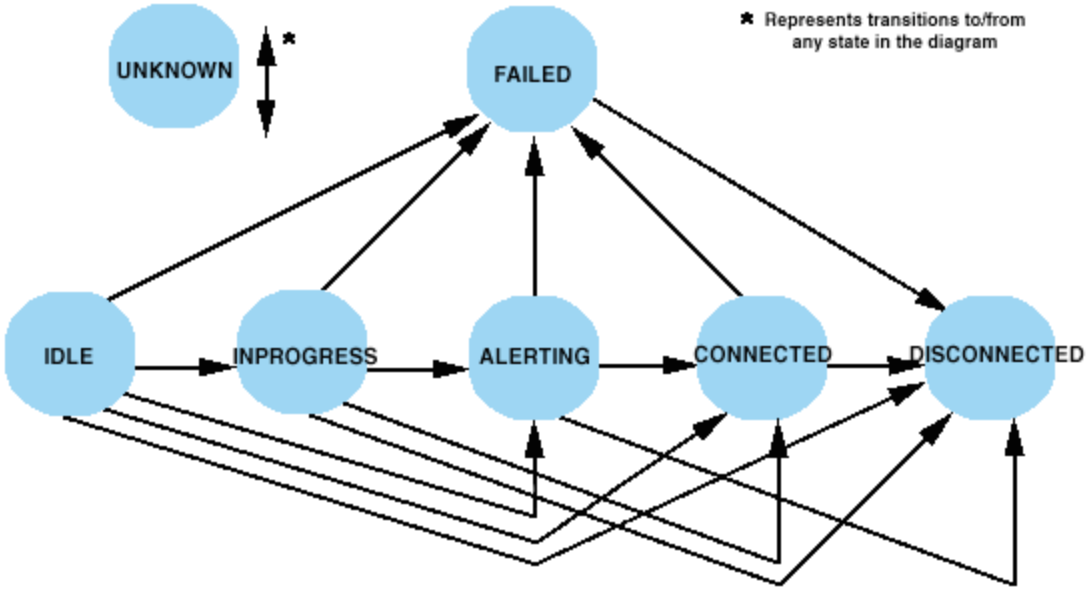

Figure 5: Connection State Transitions

- **IDLE state**

  The IDLE state is the initial state for all new Connection objects. Connections typically transition quickly out of the IDLE state into another state. A Connection in the IDLE state indicates that the party has just joined the telephone call in some form. No Core methods are valid on Connections in the IDLE state.

- **INPROGRESS state**

  The INPROGRESS state indicates that a telephone call is currently being placed to this destination endpoint.

- **ALERTING state**

  The ALERTING state indicates that the destination party of a telephone call is being alerted to an incoming telephone call.

- **CONNECTED state**

The CONNECTED state indicates that a party is actively part of a telephone call. A Connection in the CONNECTED state implies that the associated party is talking to the other parties on the call.

- **DISCONNECT state**

  The DISCONNECTED state indicates that a party is no longer a part of a telephone call. No methods are valid for Connections in the DISCONNECTED state.

- **FAILED state**

  The FAILED state indicates that a telephone call placed to the endpoint has failed. For example, if an application uses Call.connect() to place a telephone call to a party who is busy, the Connection associated with the called party transitions into the FAILED state.

- **UNKNOWN state**

  The UNKNOWN state indicates that the Provider cannot determine the state of the Connection at the present time. A Connection may transition in and out of the UNKNOWN state at any time. The effects of the invocation of any method on a Connection in this state is unpredictable.

TerminalConnection Object States

The TerminalConnection object represents the relationship between a Terminal and a Call. As mentioned previously, these objects represent a physical view of the Call, describing which physical Terminal endpoints are part of the telephone call. Similar to Connection objects, TerminalConnection objects have their own set of states and state transition diagram. This state transition diagram, with a brief description of each state follows.

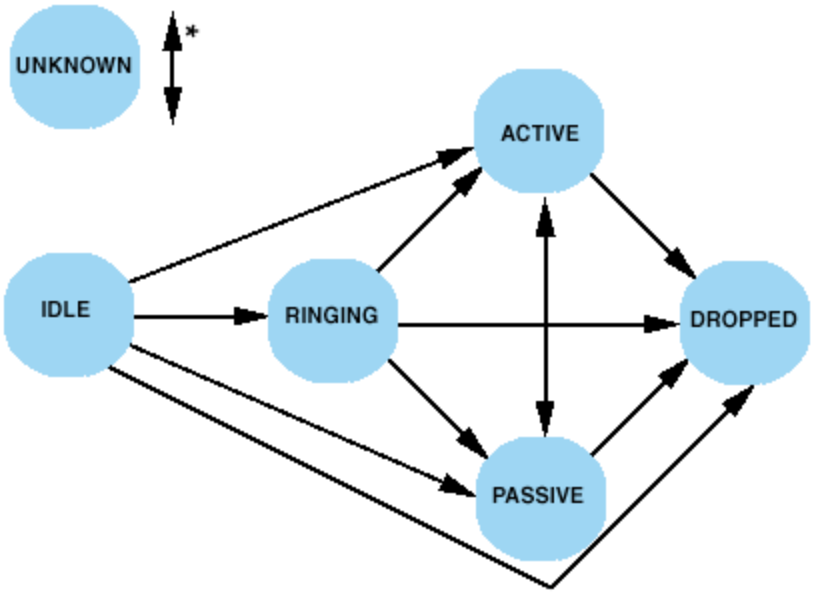

Figure 6: TerminalConnection state transitions

- **IDLE state**

  The IDLE state is the initial state for all TerminalConnection objects. It has the same connotation for the Connection object's IDLE state.

- **ACTIVE state**

  The ACTIVE state indicates a Terminal is actively part of a telephone call. This often implies that the Terminal handset is off-hook.

- **RINGING state**

  The RINGING state indicates that a Terminal is signaling to a user that an incoming telephone call is present at the Terminal.

- **DROPPED state**

  The DROPPED state indicates that a Terminal was once part of a telephone call, but has since dropped off of that telephone call. The DROPPED state is the final state for all TerminalConnections.

- **PASSIVE state**

The PASSIVE state indicates a Terminal is part of a telephone call, but not actively so. A TerminalConnection in the PASSIVE state indicates that a resource on the Terminal is being used by this telephone call. Packages which provide advanced features permit Terminals to join calls from the PASSIVE state.

- **UNKNOWN state**

  The UNKNOWN state indicates that the Provider is unable to determine the current state of a TerminalConnection. It has a similar connotation to that of the Connection object's UNKNOWN state.

Placing a Telephone Call

The past several sections have outlines the JTAPI call model, the key methods in the core package, and the Connection and TerminalConnection states. This section ties all of this information together, presenting a common scenario found in most telephony applications. This section describes the state changes the entire call model typically undergoes when an application places a simple telephone call. Readers will hopefully come away with a coherent understanding of the call model changes for this simple example.

The vehicle used to describe the state changes undergone by the call model is the diagram below. This diagram is a *call model timing diagram*, where changes in the various objects are depicted as times increases down the vertical axis. Such a diagram is given below describing the typical state changes after an application invokes the **Call.connect()** method.

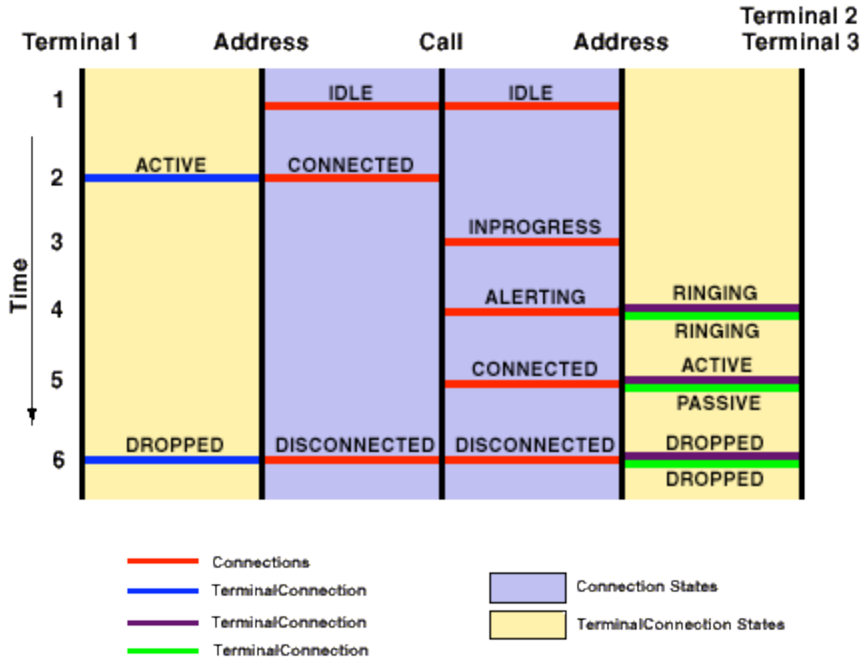

Figure 7: Call Model timing diagram

In the diagram above, discrete time steps are denoted by integers down the vertical axis. Time increases down this axis, and the integers have no relationship between real (clock) time.

This diagram, as a whole, represents a single telephone Call. In this case, the diagram represents a two-party telephone call (The **Call.connect()** method always results in a two-party call). The diagram may be broken into two halves: the left half and the right half. The left half represents the originating-end of the telephone call and the right half represents the destination-end of the telephone call.

On the left-hand (originating) side of the diagram, the two veritcal lines represent the originating Terminal and Address (which are arguments to the **Call.connect()** method) objects, as indicated on the diagram. The horizontal lines represent either Connection objects or TerminalConnection objects as marked. Note that the Connection objects are drawn in the inner- most region, whereas the TerminalConnection objects are drawn in the outer- most region.

Similarly, on the right-hand (destination) side of the diagram, the two vertical lines represent the destination Address and Terminals. In this example, there are two destination Terminals associated with the destination Address. This configuration has been depicted previously in Figure 4. Note that since there are two Terminals, there are two TerminalConnection objects on the destination side.

This diagram can be read as follows: as time increases, the Connection and TerminalConnection objects may or may not change states. The appearance of a new Connection or TerminalConnection horizontal line corresponds to a new object of that type being created during that time period.

In the example of placing a telephone call, we see that after the two Connections are created in the IDLE state, the originating transitions to the CONNECTED state, while the destination transitions to the INPROGRESS state. At that time, a TerminalConnection to the originating Terminal is created and transitions to the ACTIVE state. When the destination Connection transitions to the ALERTING state, two TerminalConnections are created in the RINGING state.

At this point, a person at one of the destination Terminals answers the call. When this happens, that TerminalConnection moves to the ACTIVE state, and the other TerminalConnection moves to the PASSIVE state. Also, the destination Connection concurrently moves to the CONNECTED state. When the telephone call ends, all Connections move to the DISCONNECTED state, and all TerminalConnections move to the DROPPED state.

As a final point, this document has used the terms "logical" and "physical" view of a telephone call. This diagram makes these concepts clear. An application can only monitor the state changes of the Connection object (i.e. the logical view). By looking at the diagram, the reader can hopefully understand that these states provide a higher-level view of the progress of the telephone call. The TerminalConnection state changes represent the physical view. By monitoring the TerminalConnection state changes, applications can find out what is happening at each physical endpoint.

The Java Telephony Observer Model

The Java Telephony API uses the Java observer/observable model to asynchronously notify the application of various changes in the JTAPI call model. These changes may include the state change of an object or the creation of an object.

The Provider, Call, Terminal, and Address objects have Observers. The interfaces corresponding to these observers are ProviderObserver, CallObserver, TerminalObserver, and AddressObserver, respectively.

The ProviderObserver reports all state changes to the Provider object. For the core package, this only include when the Provider changes state from OUT_OF_SERVCE, to IN_SERVICE, to SHUTDOWN.

The Call observer reports state change information for all Connections and TerminalConnections that are part of the telephone call as well as state changes to the Call itself. These state changes are not reported on either the Address nor the Terminal observers.

At times, the application may want to monitor Address or Terminal objects for incoming telephone

calls. In these instances, the application uses the **Address.addCallObserver()** or the **Terminal.addCallObserver** methods. These methods instruct the implementation to automatically add a CallObserver to any calls that come to an Address or Terminal. These CallObservers are removed once the call leaves the Address or Terminal.

The Address and Terminal observers report any state changes in these objects. In the core there are no events for these objects. The AddressObserver and TerminalObserver interfaces still exist, however, so other packages may extend these interfaces.

Application Code Examples

This section presents two application code examples. The first places a telephone call, and the second answers an incoming telephone call to a Terminal.

Outgoing Telephone Call Example

The following code example places a telephone call using the core Call.connect() method. It, however, looks for the states provided by the Call Control package.

```
import java.telephony.*;
import java.telephony.events.*;

/*
 * The MyCallObserver class implements the CallObserver
 * interface and receives all events associated with the Call.
 */

public class MyCallObserver implements CallObserver {

  public void callChangedEvent(CallEv[] evlist) {

    for (int i = 0; i < evlist.length; i++) {

      if (evlist[i] instanceof ConnEv) {

        String name = null;
        try {
          Connection connection = evlist[i].getConnection();
          Address addr = connection.getAddress();
```

```
          name = addr.getName();
        } catch (Exception excp) {
          // Handle Exceptions
        }
        String msg = "Connection to Address: " + name + " is ");

        if (evlist[i].getID() == ConnAlertingEv.ID) {
          System.out.println(msg + "ALERTING");
        }
        else if (evlist[i].getID() == ConnInProgressEv.ID) {
          System.out.println(msg + "INPROGRESS");
        }
        else if (evlist[i].getID() == ConnConnectedEv.ID) {
          System.out.println(msg + "CONNECTED");
        }
        else if (evlist[i].getID() == ConnDisconnectedEv.ID) {
          System.out.println(msg + "DISCONNECTED");
        }
      }
    }
  }
}


/*
 * Places a telephone call from 476111 to 5551212
 */
public class SampleOutcall {

  public static final void main(String args[]) {

    /*
     * Create a provider by first obtaining the default implementation of
     * JTAPI and then the default provider of that implementation.
     */
    Provider myprovider = null;
    try {
      JtapiPeer peer = JtapiPeerFactory.getJtapiPeer(null);
      myprovider = peer.getProvider(null);
    } catch (Exception excp) {
      System.out.println("Can't get Provider: " + excp.toString());
      System.exit(0);
    }

   /*
    * We need to get the appropriate objects associated with the
    * originating side of the telephone call. We ask the Address for a list
    * of Terminals on it and arbitrarily choose one.
    */
    Address origaddr = null;
    Terminal origterm = null;
    try {
      origaddr = myprovider.getAddress("4761111");
```

```
      /* Just get some Terminal on this Address */
      Terminal[] terminals = origaddr.getTerminals();
      if (terminals == null) {
        System.out.println("No Terminals on Address.");
        System.exit(0);
      }
      origterm = terminals[0];
    } catch (Exception excp) {
      // Handle exceptions;
    }


    /*
     * Create the telephone call object and add an observer.
     */
    Call mycall = null;
    try {
      mycall = myprovider.createCall();
      mycall.addObserver(new MyCallObserver());
    } catch (Exception excp) {
      // Handle exceptions
    }

    /*
     * Place the telephone call.
     */
    try {
      Connection c[] = mycall.connect(origterm, origaddr, "5551212");
    } catch (Exception excp) {
      // Handle all Exceptions
    }
  }
}
```

Incoming Telephone Call Example

The following code example illustrates how an application answers a Call at a particular Terminal. It shows how application accept calls when (and if) they are offered to it. This code example greatly resembles the core InCall code example.

```
import java.telephony.*;
import java.telephony.events.*;

/*
 * The MyObserver class implements the CallObserver and
 * recieves all Call-related events.
 */
```

```
public class MyObserver implements CallObserver {

  public void callChangedEvent(CallEv[] evlist) {

    for (int i = 0; i < evlist.length; i++) {

      if (evlist[i] instanceof TermConnEv) {
        TerminalConnection termconn = null;
        String name = null;

        try {
          TermConnEv tcev = (TermConnEv)evlist[i];
          TerminalConnection termconn = tcev.getTerminalConnection();
          Terminal term = termconn.getTerminal();
          String name = term.getName();
        } catch (Exception excp) {
          // Handle exceptions.
        }

        String msg = "TerminalConnection to Terminal: " + name + " is ";

        if (evlist[i].getID() == TermConnActiveEv.ID) {
          System.out.println(msg + "ACTIVE");
        }
        else if (evlist[i].getID() == TermConnRingingEv.ID) {
          System.out.println(msg + "RINGING")

          /* Answer the telephone Call */
          try {
            termconn.answer();
          } catch (Exception excp) {
            // Handle Exceptions;
          }
        }
        else if (evlist[i].getID() == TermConnDropped.ID) {
          System.out.println(msg + "DROPPED");
        }
      }
    }
  }
}

/*
 * Create a provider and monitor a particular terminal for an incoming call.
 */
public class SampleIncall {

  public static final void main(String args[]) {

    /*
     * Create a provider by first obtaining the default implementation of
     * JTAPI and then the default provider of that implementation.
```

```
     */
    Provider myprovider = null;
    try {
      JtapiPeer peer = JtapiPeerFactory.getJtapiPeer(null);
      myprovider = peer.getProvider(null);
    } catch (Exception excp) {
      System.out.println("Can't get Provider: " + excp.toString());
      System.exit(0);
    }

    /*
     * Get the terminal we wish to monitor and add a call observer to that
     * Terminal. This will place a call observer on all call which come to
     * that terminal. We are assuming that Terminals are named after some
     * primary telephone number on them.
     */
    try {
      Terminal terminal = myprovider.getTerminal("4761111");
      terminal.addCallObserver(new MyCallObserver());
    }
  }
}
```

Locating and Obtaining Providers

The Java Telehony API defines a convention by which telephony server implementations of JTAPI make their services available to applications.

The two elements that link an application to a server are:

- **JtapiPeerFactory**

  The JtapiPeerFactory class is the first point of contact for an application that needs telephony services. It has the ability to return a named JtapiPeer object or a default JtapiPeer object. It is defined as a static class.

- **JtapiPeer**

  The JtapiPeer interface is the basis for a vendor's particular implementation of the Java Telephony API. Each vendor which provides an implementation of JTAPI must implement this interface in a class that can be loaded by the JtapiPeerFactory.

  It is through a class that implements the JtapiPeer object that an application gets a Provider

object.

JtapiPeerFactory: Getting Started

The JtapiPeerFactory is a static class defined in JTAPI. Its sole public method, *getJtapiPeer()* gets the JtapiPeer implementation requested or it returns a default implementation.

getJtapiPeer() takes the name of the desired JTAPI server implementaion class as a parameter to return an object instance of that class. If no name is provided, *getJtapiPeer()* returns the default JTAPI server implementation object.

JtapiPeer: Getting a Provider Object

JtapiPeer is an interface. It is used by the JTAPI server implementors. It defines the methods that applications use to get Provider objects, to query services available on those providers, and to get the name of the JtapiPeer object instance. By creating a class that implements the JtapiPeer interface, JTAPI implementations make the following methods available to applications.

Applications use the *getProvider()* method on this interface to obtain new Provider objects. Each implementation may support one or more different "services" (e.g. for different types of underlying network substrate). A list of available services can be obtained via the *getServices()* method.

Applications may also supply optional arguments to the Provider. These arguments are appended to the string argument passed to the *getProvider()* method. The string argument has the following format:

< service name > ; arg1 = val1; arg2 = val2; ...

Where < service name > is not optional, and each optional argument pair which follows is separated by a semi-colon. The keys for these arguments is implementation specific, except for two standard-defined keys:

1. login: provides the login user name to the Provider.
2. passwd: provides a password to the Provider.

Applications use the *getName()* method to get the name of this JtapiPeer object instance. It has a *name* parameter, which is the same name used as an argument to *JtapiPeerFactory.getJtapiPeer()* method.

Security in the Java Telephony API

JTAPI peer implementations use the Java "sandbox" model for controlling access to sensitive operations. Callers of JTAPI methods are categorized as "trusted" or "untrusted", using criteria determined by the runtime system. Trusted callers are allowed full access to JTAPI functionality. Untrusted callers are limited to operations that cannot compromise the system's integrity.

JTAPI may be used to access telephony servers or implementations that provide their own security mechanisms. These mechanisms remain in place; parameters such as user name and password are provided through parameters on the JtapiPeer.getProvider() method.