



JTAPI Concepts

Introduction

This chapter introduces the major concepts with which you need to be familiar before creating JTAPI applications for Cisco IP Telephony Solutions. The following concepts are discussed:

- [What's new in JTAPI for Cisco CallManager Release 3.1., page 1-2](#)
- [JTAPI Call Model Overview, page 1-2](#)
- [Transfer and Conference Extensions, page 1-9](#)
- [Media Termination Extensions, page 1-16](#)
- [Redundancy, page 1-25](#)

What's new in JTAPI for Cisco CallManager Release 3.1.

This is a brief overview of features and changes to JTAPI for Cisco CallManager release 3.1.

- [Redundancy](#) describes a new component called CTIManager. The CTIManager allows a JTAPI application to control devices on another Cisco CallManager in a cluster. It supports multiple Cisco CallManagers and CTI managers during failover and recovery, and supports automatic device recovery during a failover.
- [Directory Change Notification](#) describes asynchronous directory change notification.
- [Transfer and Conference Enhancement](#) describes enhancements to Transferring & Conferencing.
- [Call Forward Setting](#) describes Cisco JTAPI's implementation for supporting Call Forwarding.
- [CiscoJtapiExceptions](#) describes CiscoJtapiException handling modifications.
- [Redirect](#) describes Cisco JTAPI's Redirect request
- [Alarm Services](#) describes Cisco JTAPI's support for Alarm Services
- [Application Control of JTAPI Parameters](#) describes the parameters within jtapi.ini.
- [Dynamic Trace Enabling Using Jtprefs](#) describes dynamic enabling of traces from the Jtprefs application

JTAPI Call Model Overview

Introduction

An abstract telephony model, Java Telephony application Programming Interface (JTAPI) is capable of uniformly representing the characteristics of a wide variety of telecommunication systems. Because it is defined without direct reference to any particular telephony hardware or software, JTAPI is well suited to the task of controlling or observing nearly any telephone system. For instance, a computer program that makes telephone calls using an implementation of JTAPI for modems might work without modification using the Cisco JTAPI implementation.

As powerful as an abstraction may be in theory, in practice, programmers often need to know the details of how the JTAPI model represents the underlying components of a particular telephony system. A modem is a very simple telephony device with limited features. In contrast, the Cisco IP Telephony Solutions product family offers its users a comprehensive list of features and configuration capabilities. Programmers can best leverage the rich features of Cisco IP Telephony Solutions systems by understanding how it fits into the JTAPI model.

The goal of this section is to describe how the Cisco IP Telephony Solutions product family operates with JTAPI. After completing this section, readers will be able to correlate JTAPI call model objects with their underlying Cisco IP Telephony Solutions features, and vice-versa.

JtapiPeer and Provider

The main point of contact between applications and JTAPI implementations is the Provider object, created through the implementation of the JtapiPeer object. The Provider object is a repository that contains the entire collection of call model objects, Addresses, Terminals, and Calls, controllable at any time by an application.

The JtapiPeer.getServices(), which returns server names, is administered by the JTPREFS applications. The Provider entails two basic processes: initialization and shutdown.

Initialization

The Provider initialization process does not block while devices are being initialized. The JtapiPeer.getProvider() method returns a Provider object immediately after login is completed. However, the Provider is out of service until it has finished initializing all terminals and addresses in its domain.



Note Before continuing, applications must add a Provider observer and await a notification that the Provider is in service.

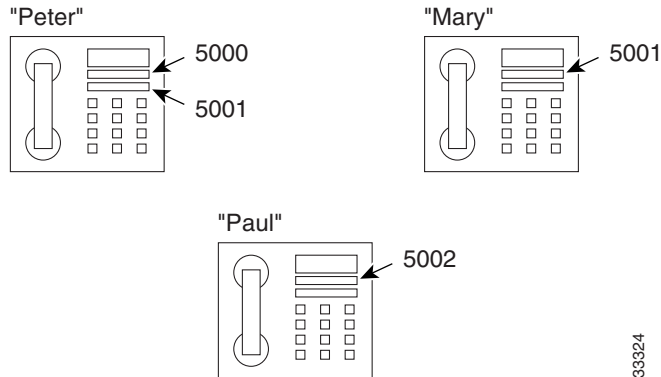
Shutdown

When an application calls provider.shutdown(), JTAPI loses communications permanently with the Cisco CallManager, and a ProviderShutdown event is sent to the application. The application can assume that the Provider will not come up again and the application must handle a complete shutdown.

Addresses and Terminals

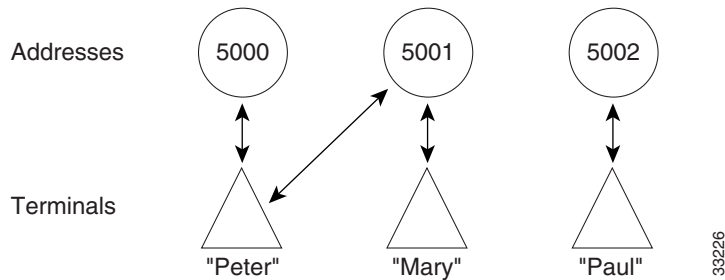
The Cisco IP Telephony Solutions architecture has three fundamental types of endpoints: telephone sets, virtual devices (media termination points and route points), and gateways. Of these endpoints, only telephones and media termination points are exposed through the Cisco JTAPI implementation.

Cisco CallManager allows users to configure telephones to have one or more lines, dialable numbers, which may be shared among multiple telephones simultaneously or configured for exclusive use by only one telephone at a time. Each line on a telephone is capable of terminating two calls simultaneously, one of which must be on hold. This is similar to the operation of the “call waiting” feature on home telephones. [Figure 1-1 “Telephone Diagram”](#) shows two configurations, Peter and Mary are sharing one telephone line, 5001, while Paul is configured with his own telephone line, 5002.

Figure 1-1 Telephone Diagram

Address and Terminal Relationship

All types of Cisco CallManager endpoints are identified by a unique name. A telephone terminal is identified by its hardware Media Access Control (MAC) address (e.g. "SEP0010EB1014"), whereas the system administrator may assign any name to a media termination point, so long as its name is distinct. For each endpoint controlled by a Provider, the Cisco JTAPI implementation uses its administered name to construct a corresponding Terminal object. Terminal objects in turn have one or more Address objects, each of which corresponds to a line on the endpoint. [Figure 1-2 "Address and Terminal Relationship"](#) shows a graphical representation of the relationship between Addresses and Terminals.

Figure 1-2 Address and Terminal Relationship

If two or more endpoints share a line (directory number), then the corresponding Address object will be related to more than one Terminal object.



Note Cisco JTAPI does not support shared lines.

CiscoCallID

The Cisco CallID object represents a unique object associated with each connection in Cisco JTAPI. Applications may use the object itself or the integer representation of the object.

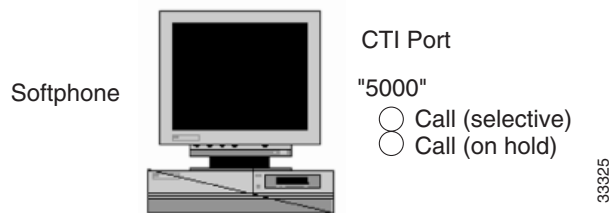
CiscoConnectionID

The CiscoConnectionID object represents a unique object associated with each connection in Cisco JTAPI. Applications may use the object itself or the integer representation of the object.

Media Termination

In Cisco JTAPI, software-based media termination is accomplished using Computer Telephony Integration (CTI) Ports. They have one or more lines (dialable numbers) that can be used to originate or receive calls. They however need a controlling application to provide the source and sink of the media. An application registers its interest in the media termination port with the Cisco CallManager. The Cisco CallManager then delivers all the events related to this virtual device to the application. In Cisco JTAPI, CTI Ports are referred to as CiscoMediaTerminals. [Figure 1-3](#) shows the CTI Port configuration. For details about administering and configuring a CTI Port, refer to the Cisco CallManager Administration web pages.

Figure 1-3 CTI Port Diagram

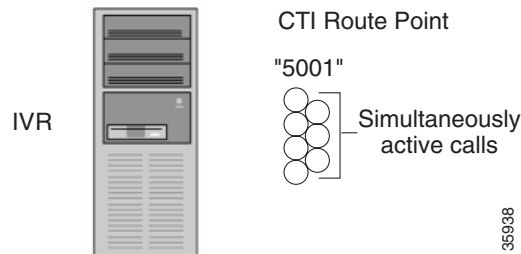


To implement a softphone application (where the PC acts as the telephone set, for example) the JTAPI application would manage a CTI Port.

Routing

Routing or call queuing is accomplished by configuring a CTI Route Point. CTI Route Points can handle an arbitrary number of active calls. Cisco JTAPI applications register their interest in this virtual device with the Cisco CallManager, and have to perform all the event handling for this device. [Figure 1-4](#) shows the CTI Route Point configuration. CTI Route Points may only have one line (or dialable number). For details on administering and configuring a CTI Route Point, refer to the Cisco CallManager Administration CCMAAdmin—Device web pages.

Figure 1-4 CTI Route Points



In order to implement a voice response application, for example, the JTAPI application would manage a CTI Route Point.

CiscoObjectContainer

The CiscoObjectContainer interface allows applications to associate an application defined object to objects that implement this interface. In Cisco JTAPI, the following interfaces extend the CiscoObjectContainer interface: CiscoJTAPIPeer, CiscoProvider, CiscoCall, CiscoAddress, CiscoTerminal, CiscoConnection, CiscoTerminalConnection, CiscoConnectionID and CiscoCallID.

Cisco JTAPI Overview

Introduction

The following sections outline the deviations, if any, from the JTAPI v 1.2 specification or the specifics of the Cisco JTAPI implementation. Cisco JTAPI also provides extensions to the JTAPI v 1.2 specification. These extensions offer additional functionality to the Cisco implementation that is not defined in the specification. The classes and interfaces for the extensions are packaged in the com.cisco.jtapi.extensions package, and are explained in detail in Chapter 2, “Cisco JTAPI Extensions.”

Obtaining a CiscoProvider

The following information must be passed in the JtapiPeer.getProvider() method for applications to obtain a CiscoProvider.:

- Hostname or IP address for the Cisco CallManager server
- Login of user administered in the directory.
- Password of user specified.
- (Optional) Application information— this parameter may be a string of any length.

Applications should include enough descriptive information so that if the appinfo were logged in an alarm, administrators would know which application caused the alarm. Applications should not include hostname or IP address where they reside, nor the time at which they were spawned. This information is either redundant with what JTAPI already logs or is potentially confusing when trying to track the progress of an application that is reconnecting on occasion. Also, no "=" or ";" characters are allowed in the appinfo string, as they are used to delimit the getProvider () string. When the appinfo is not specified, you can use a generic and quasi-unique name (JTAPI[XXXX]@hostname, where XXXX is a random four digit number) instead.

These parameters are passed in key value pairs concatenated in a string as follows:

```
JtapiPeer.getProvider( "CTIManagerHostname;login=user;passwd=userpassword;appinfo=Cisco Softphone" )
```

ProviderObserver

Initialization

In Cisco JTAPI, the `getProvider(..)` method returns as soon as the TCP link, the initial handshake with the Cisco CallManager, and device list enumeration are complete. The provider is now in the `OUT_OF_SERVICE` state. Cisco JTAPI applications must wait for the provider to go to the `IN_SERVICE` state before the controlled device list is valid. A `ProvInServiceEv` event is delivered to an object implementing the `ProviderObserver` interface. A `ProvOutOfServiceEv` event is also delivered upon an orderly Cisco CallManager shutdown.



Note It is not sufficient to implement only the `CiscoProviderObserver`, the observer must also be added to the provider via `provider.addObserver(..)`.

`Provider.getTerminals()`

This method returns an array of terminals created for the devices administered in the user's control list in the directory. Refer to the *Cisco CallManager Administration Guide* to administer the user's control list.

`Provider.getAddresses()`

This method returns an array of addresses created from the lines assigned to the devices administered in the user's control list in the directory.

Changes to the User's Control List in the Directory

If a device is added to the user's control list after the JTAPI application starts, a `CiscoTermCreatedEv`, and the respective `CiscoAddrCreatedEv`, are generated and sent to observers implementing the `CiscoProviderObserver`. In addition, applications can monitor the current registration state of the controlled devices, and dynamically track the availability of these devices. The events for an in-service Address or Terminal are delivered to observers implementing the `CiscoAddressObserver` and the `CiscoTerminalObserver`.



Note It is not sufficient to implement only the observers, the observers must also be added by `address.addObserver(..)` and, similarly, for the terminal by the `terminal.addObserver(..)` method.

Unobserved Addresses and Terminals

Cisco JTAPI learns about calls only when there is a `CallObserver` attached to the terminals/addresses of the Provider. This means that methods such as `Provider.getCalls()` or `Address.getConnections()` will return null, even when there are calls at the address, unless there is a `CallObserver` attached to the address. It is also required to add a `CallObserver` to the Address or Terminal originating a call via the `Call.connect(..)` method.

**Note**

Before invoking the `call.connect()` method, a `CallObserver` must be added to the address or terminal originating the call. Otherwise, the method returns an exception.

Threaded Callbacks

The Cisco JTAPI implementation is designed to allow applications to invoke blocking JTAPI methods such as `Call.connect()` and `TerminalConnection.answer()` from within their observer callbacks. This means that applications are not subject to the restrictions imposed by the JTAPI 1.2 specification, which cautions applications against using JTAPI methods from within observer callbacks. Applications, however, can selectively disable the queuing logic of the Cisco JTAPI implementation by implementing the `CiscoSynchronousObserver` interface on their observer objects.

Many applications will not be adversely affected by this asynchronous behavior. Applications that would benefit from a coherent call model during observer callbacks, however, can selectively disable the queuing logic of the Cisco JTAPI implementation. By implementing the `CiscoSynchronousObserver` interface on its observer objects, an application declares that it wishes events to be delivered synchronously to its observers. Events delivered to synchronous observers will match the states of the call model objects queried from within the observer callback.

**Note**

Objects that implement the `CiscoSynchronousObserver` interface are strictly forbidden from invoking blocking JTAPI methods from within their event callbacks. The consequences of doing so are unpredictable, and may include deadlocking the JTAPI implementation. However, they may safely use the accessor methods of any JTAPI object, for instance `Call.getConnections()` or `Connection.getState()`.

Querying Dynamic Objects

Beware of querying dynamic objects such as call objects. By the time you get an event, the object (e.g., call) may be in a different state than the state indicated. For example, by the time you get a `CiscoTransferStartEV`, the transferred call, which is the call that is being transferred, may have removed all of its internal connections.

`callChangeEvent()`

When the `callChangedEvent()` method is called, any references contained in the event itself, are guaranteed to be valid. For example, if the event contains a `getConnection()` method, then the application can call this method and get a valid connection reference. Likewise, a `getCallingAddress()` method will be guaranteed to return a valid `Address` object.

Connections

Connections retain their references to calls and addresses forever. So, a connection reference obtained from a call event can always be used to obtain the connection call (`getCall()`) and address (`getAddress()`).

TerminalConnections

TerminalConnections retain their references to terminals and connections forever. So, a terminal connection reference obtained from a call event can always be used to obtain the terminal connection terminal (`getTerminal()`) and connection (`getConnection()`).

CiscoConsultCall

For the `CiscoConsultCall` interface, a reference to a consulting terminal connection is retained forever. For example, when processing a `CiscoConsultCallActive` event, `getConsultingTerminalConnection()` is guaranteed to return a valid terminal connection reference. Further, the terminal connection is guaranteed to provide access to the consulting connection and thus the consulting call.

CiscoTransferStartEv

For the `CiscoTransferStartEv`, the references to the transferred call, transfer controller and final call in the event will be valid when `callChangedEvent()` is called. However, the connections on these calls may or may not still be returned by `getConnections()`.

Transfer and Conference Extensions

Introduction

Transfer and Conference events are difficult to understand in JTAPI. The reason is that when the participants are moved from one call to the other, JTAPI represents this by deleting the parties from one call and adding them to the other call. It may be confusing for an application to receive an indication that a party has dropped from the call when in reality it is in the process of being moved. The Cisco JTAPI implementation defines some extra events that make it easier for applications to deal with these functions.

Transfer

The transfer feature moves the participants of one call, the transferred call, to another call, the final call. Moving participants in a call results in their associated Connections transitioning to the DISCONNECTED state in the transferred call and new Connections for these participants being created in the final call. Similarly, any associated TerminalConnections transition into the DROPPED state in the transferred call and are created in the final call. Cisco extensions have been defined to mark the start and the end of the events that are related to transfer.

One way to correlate the newly created connection objects with the old connection objects is to use the `CiscoConnection.getConnectionID()` method to obtain the `CiscoConnectionID` for the old and new connections. Matching connections will have identical `CiscoConnectionID` objects when compared using the `CiscoConnectionID.equals()` method.

CiscoTransferStartEv

This event indicates that the transfer operation has started, and the events that follow are related to this operation. Specifically, Connections and TerminalConnections are both removed and added as a result of the transfer.

Applications may obtain the two calls involved in transfer—transferred call and final call, and the transfer controller information from this event. If the transfer controller is not being observed by the JTAPI application, the transfer controller information is not available in this event.

CiscoTransferEndEv

This event indicates that the transfer operation has ended. After this event is received, the application can assume that all involved parties have been transferred, and all Connections and TerminalConnections have been moved to the final call.

Transfer Scenarios

In the following scenarios, A, B, and C are three parties involved in the transfer.

Blind Transfer, Where B is the Transfer Controller

In blind transfer, a call that has been answered already is redirected without issuing separate consult and transfer commands.

- A calls B on call Call1
- B answers and blind transfers the call to C

In this scenario, a consult call (Call2) is placed internally between B and C. To do this type of transfer the following JTAPI method is necessary:

- Call1.transfer (C)

Table 1-1 lists the core events that observers for A and B receive between the CiscoTransferStartEv and the CiscoTransferEndEv:

Table 1-1 Core Events for Observers for A and B

Meta Event Cause	Call	Event	Fields
META_UNKNOWN	Call1	CiscoTransferStartEv	transferredCall=Call2 finalCall=Call1 transferController=TermConnB
META_CALL_TRANSFER RING	Call1	TermConnDroppedEv B CallCtlTermConnDroppedEv B ConnDisconnectedEv B CallCtlConnDisconnectedEv B	
META_CALL_TRANSFER RING	Call1	ConnCreatedEv C ConnAlertingEv C CallCtlConnAlertingEv C TermConnCreatedEv C TermConnRingingEv C CallCtlTermConnRingingEvImpl C	

Table 1-1 Core Events for Observers for A and B

Meta Event Cause	Call	Event	Fields
META_CALL_TRANSFER RING	Call2	TermConnDroppedEv B CallCtlTermConnDroppedEv B ConnDisconnectedEv B CallCtlConnDisconnectedEv B	
META_CALL_TRANSFER RING	Call2	TermConnDroppedEv C CallCtlTermConnDroppedEv C ConnDisconnectedEv C CallCtlConnDisconnectedEv CCallInvalidEv	
META_UNKNOWN	Call2	CallObservationEndedEv	
META_UNKNOWN	Call1	CiscoTransferEndEv	transferredCall=Call2 finalCall=Call1 transferController=TermConnB

Consult Transfer, Where B is the Transfer Controller

In a consult transfer, applications can redirect calls to a different address and the transferrer can “consult” with the transfer destination before redirecting.

- A calls B on call Call1
- B answers and consults to C on call Call2
- B transfers call Call2 to call Call1

To do this type of transfer the following JTAPI methods are necessary:

- Call2.setTransferEnable(true) (This is optional because transfer is enabled in the call object by default.)
- Call2.consult(TermConnB, C)
- Call1.transfer(Call2)



Note During consult transfer, Call1.transfer(Call2) will transfer the call, but not Call2.transfer(Call1).

Table 1-2 lists the core events that observers of A and B receive between the CiscoTransferStartEv and the CiscoTransferEndEv:

Table 1-2 Core Events for Observers of A and B

Meta Event Cause	Call	Event	Fields
META_UNKNOWN	Call1	CiscoTransferStartEv	transferredCall=Call2 finalCall=Call1 transferController=TermConnB
META_CALL_TRANSFER RING	Call1	TermConnDroppedEv B CallCtlTermConnDroppedEv B ConnDisconnectedEv B CallCtlConnDisconnectedEv B	

Table 1-2 Core Events for Observers of A and B (continued)

Meta Event Cause	Call	Event	Fields
META_CALL_TRANSFER RING	Call1	ConnCreatedEv C ConnConnectedEv C CallCtlConnEstablishedEv C TermConnCreatedEv C TermConnActiveEv C CallCtlTermConnTalkingEv C	
META_CALL_TRANSFER RING	Call2	TermConnDroppedEv B CallCtlTermConnDroppedEv B ConnDisconnectedEv B CallCtlConnDisconnectedEv B	
META_CALL_TRANSFER RING	Call2	TermConnDroppedEv C CallCtlTermConnDroppedEv C ConnDisconnectedEv C CallCtlConnDisconnectedEv C CallInvalidEv C	
META_UNKNOWN	Call2	CallObservationEndedEv	
META_UNKNOWN	Call1	CiscoTransferEndEv	transferredCall=Call2 finalCall=Call1 transferController=TermConnB

Arbitrary Transfer, where A is the Transfer Controller

In an arbitrary transfer, one call can be transferred to another call, irrespective of how either call was created. Unlike consult transfer, there is no need to first create one of the calls using the consult method.

- A calls B on call Call1
- A puts Call1 on hold
- A calls C on call Call2
- A transfers Call1 to Call2

To do this type of transfer the following JTAPI methods are necessary:

- Call2.transfer(Call1) to transfer call Call1 to final call Call2, or
- Call1.transfer(Call2) to transfer call Call2 to final call Call1

Assuming Call1.transfer(Call2) was called, Table 1-3 lists the core events observers on A and C receive between CiscoTransferStartEv and CiscoTransferEndEv:

Table 1-3 Core Events for Observers of A and C

Meta Event Cause	Call	Event	Fields
META_UNKNOWN	Call1	CiscoTransferStartEv	transferredCall=Call2 finalCall=Call1 transferController=TermConnB
META_CALL_TRANSFER RING	Call1	TermConnDroppedEv B CallCtlTermConnDroppedEv B ConnDisconnectedEv B CallCtlConnDisconnectedEv B	

Table 1-3 Core Events for Observers of A and C (continued)

Meta Event Cause	Call	Event	Fields
META_CALL_TRANSFER RING	Call1	ConnCreatedEv C ConnConnectedEv C CallCtlConnEstablishedEv C TermConnCreatedEv C TermConnActiveEv C CallCtlTermConnTalkingEv C	
META_CALL_TRANSFER RING	Call2	TermConnDroppedEv B CallCtlTermConnDroppedEv B ConnDisconnectedEv B CallCtlConnDisconnectedEv B	
META_CALL_TRANSFER RING	Call2	TermConnDroppedEv C CallCtlTermConnDroppedEv C ConnDisconnectedEv C CallCtlConnDisconnectedEv C CallInvalidEv C	

Conference

When conferencing two calls together, JTAPI specifies that all the parties from one call be moved to the other call. The call whose parties are moved away and which becomes invalid subsequently is called the “merged” or “consult” call hereafter. The call to which the merged parties move is called the “final” call hereafter. When parties move from the merged call to the final call, the application receives events indicating that all parties have dropped from the merged call, and events indicating that the parties have reappeared on the final call.

One way to correlate the newly created connection objects with the old connection objects is to use the `CiscoConnection.getConnectionID()` method to obtain `CiscoConnectionID` objects for all old connections and all new connections. Matching connections will have identical `CiscoConnectionID` objects when compared using the `CiscoConnectionID.equals()` method.

Conference support exists for:

- `javax.telephony.callcontrol.CallControlCall.conference(Call)`,
- `javax.telephony.callcontrol.CallControlCall.getConferenceController()`
- `javax.telephony.callcontrol.CallControlCall.getConferenceEnable()`,
- `javax.telephony.callcontrol.CallControlCall.setConferenceController(TerminalConnection)`,
- `javax.telephony.callcontrol.CallControlCall.setConferenceEnable(boolean)`,

Cisco Extensions

Cisco JTAPI implementation provides two extra events that signal the Start and End of Conference: `CiscoConferenceStartEv` and `CiscoConferenceEndEv`. These events are sent when Conference is initiated and when it is completed. They give handles to the final call, the merged conference (consult call and the two controlling `TerminalConnections` (in HELD and TALKING state).

CiscoConferenceStartEv

This event is sent when `call1.conference(call2)` is invoked or if the Conference button is pressed for the second time on an IP Phone. The `ConferenceStartEv` signifies the start of the merging process. This event is followed by a sequence of merging events reflected by the Conference process in Cisco CallManager described in the previous section.

CiscoConferenceEndEv

This event is sent at the end of the merge process after a `ConferenceStartEv` is sent. It signifies the completion of the merge of the Consult (or Merged) call into the Final Conference Call. The Merged call is in `INVALID` state and an `ObservationEndedEv` is sent for the call observer.

CiscoCall.setConferenceEnable()

The Cisco JTAPI implementation uses the `CiscoCall.setConferenceEnable()` and the `CiscoCall.setTransferEnable()` methods to control whether the consult call will be initiated via the conference or the transfer feature. If none of the features are enabled explicitly, transfer will be used by default.

Conference Scenarios

The following scenarios describe the two typical types of Conference that can be invoked.

Consult Conference with B as the Conference Controller

The following sequence of steps typically describe this scenario:

- A calls B (Call 1)
- B answers
- B Consults C (Call 2)


```
setConferenceEnable()
call2.consult(tc, C)
```
- C answers
- B Completes Conference


```
Call1.conference(Call2)
```



Note You must invoke the `conference()` method on the original call to complete a conference after a consultation. Invoking `conference` in the consult call object throws an exception.

Arbitrary Conference with B as the Conference Controller

The following sequence of steps typically describe this scenario:

- A calls B (Call 1)
- B answers
- B places the call on hold
- B calls C (Call 2)

- C answers
 - B Completes Conference
- Call1.conference(Call2) or
Call2.conference(Call1)

Conference Events

Table 1-4 provides the sequence of Core, Call control and Cisco Extension events when Call1.Conference(Call2) is called:

Table 1-4 Sequence of Events

Meta Event Cause	Call	Event	Fields
META_UNKNOWN	Call1	CiscoConferenceStartEv	consultCall = Call2 finalCall = Call1 conferenceController=TermConnB
META_CALL_MERGING	Call1	CallCtlTermConnTalkingEv B	
META_CALL_MERGING	Call1	ConnCreatedEv C ConnConnectedEv C CallCtlConnEstablishedEv C TermConnCreatedEv C TermConnActiveEv C CallCtlTermConnTalkingEv C	
META_CALL_MERGING	Call2	TermConnDroppedEv B CallCtlTermConnDroppedEv B ConnDisconnectedEv B CallCtlConnDisconnectedEv B	
META_CALL_MERGING	Call2	TermConnDroppedEv C CallCtlTermConnDroppedEv C ConnDisconnectedEv C CallCtlConnDisconnectedEv C CallInvalidEv	consultCall=Call2 finalCall=Call1 conferenceController=TermConnB
META_UNKNOWN	Call2	CallObservationEndedEv	
META_UNKNOWN	Call1	CiscoConferenceEndEv	

Transfer and Conference Enhancement

All parties involved in the call transfer in are sent CiscoTransferStartEv and CiscoTransferEndEv. All parties involved in the call conference are sent CiscoConferenceStartEv and CiscoConferenceEndEv. A call transfer still generates two events—the dropping of a connection to the first call and the creation of a connection to the second call. This order has been changed in Cisco CallManager Release 3.1. Connections are first created in the final call and then dropped in the consult call.



Note

Not All parties involved in the transfer of calls were delivered these events in Cisco CallManager Release 3.0

**Note**

Applications should not rely on the order of events between `CiscoTransferStartEv` and `CiscoTransferEndEv` or between `CiscoConferenceStartEv` and `CiscoConferenceEndEv` for transferring and conferencing when porting applications from Cisco CallManager Release 3.0 to 3.1.

Consult Without Media

Applications can inform Cisco CallManager that a consultation call for a transfer is being placed without establishing the media path. It is not necessary to establish the media path for the intermediate call, if the consultation call is being placed to determine whether an agent is available before the actual transfer. The `consultWithoutMedia` method as defined in the `CiscoConsultCall` interface creates a consultation call without establishing the media path.

**Note**

The consultation call may only be transferred. It can not be conferenced.

Media Termination Extensions

Introduction

Media termination is a feature that allows applications to transmit and capture the bearer of a call, for example, audio or video. This is sometimes referred to as “rendering and recording” or “sourcing and sinking” media. It is distinct from call control because media termination concerns the data that flows between endpoints in a call, not the details of setting up or tearing down calls. For example, an automatic call distributor (ACD) uses call control to route calls among available agents, but does not terminate media. An interactive voice response (IVR) application, on the other hand, uses call control to answer and disconnect calls and uses media termination to play sound files to callers.

Although there are no telephony applications that are solely interested in media termination, this feature is always used in combination with call control. JTAPI 1.2 is primarily a call control specification, and offers very limited support for applications that require media termination. Since the Cisco IP Telephony Solutions platform supports media termination to a much greater degree than JTAPI standard, the Cisco JTAPI implementation extends JTAPI to add full support for this feature.

Cisco CallManager Media Endpoint Model

Endpoints are the entities within the Cisco IP Telephony Solutions platform that terminate media, such as IP telephones and gateways. A call from one endpoint to another results in media flowing between the two endpoints. All endpoints in the Cisco IP Telephony Solutions platform transmit voice data using real-time protocol (RTP). The Cisco IP Telephony Solutions telephones and gateways, for example, have built-in RTP stacks. Applications may also act as endpoints in an Cisco IP Telephony Solutions system, that is, they may terminate media. Since all Cisco IP Telephony Solutions endpoints use RTP, applications must also be able to transmit and receive RTP packets.

Payload and Parameter Negotiation

In addition to bearer data, payload, each RTP packet contains a header that helps endpoints to determine how to reassemble and decode a sequence of such packets into a media stream. What RTP does not provide, however, is a means for endpoints to negotiate which payload type to use for a particular stream, for example, audio data encoded using the G.711 standard. Furthermore, RTP does not offer a means of negotiating unique payload type parameters such as the sampling rate of the encoded data or the number of samples that are to be transferred in each RTP packet. Instead, RTP is usually used in conjunction with another protocol such as H.323, which specifies its own method for endpoints to negotiate these parameters. All such negotiation is performed prior to transmitting RTP packets between endpoints.

Cisco CallManager, not the endpoints, is responsible for selecting the payload and encoding parameters for RTP streams. The five steps involved in a typical bidirectional audio telephone call are as follows:

- Initialization
- Payload Selection
- Receive Channel Allocation
- Starting Transmission and Reception
- Stopping Transmission and Reception

Initialization

Upon startup, each endpoint informs Cisco CallManager of its media capabilities, i.e., G.711, G.723, G.729a, etc. Startup for an IP telephone, for example, occurs when the telephone is first turned on, or after it recontacts Cisco CallManager after losing its former connection. The endpoint cannot express a preference for one payload type versus another, but it can specify certain parameters for each payload type, such as, packet size.

The capability list registered by the endpoint is exclusive and immutable. If the endpoint specifies that it can support both G.711 and G.723, it is implicitly declaring that it cannot support G.729a. Moreover, the endpoint must disconnect from Cisco CallManager and reinitialize in order to change the list of capabilities that it supports.

JTAPI applications perform this step by registering a `CiscoMediaTerminal` with Cisco CallManager. The `CiscoMediaTerminal.register()` method allows applications to supply an array of media capability objects for registration with Cisco CallManager. This step informs Cisco CallManager that the application will act as the endpoint for all calls to or from a particular directory number, as determined by the device configuration in the Cisco CallManager configuration.

Payload Selection

When a bidirectional media stream is about to be created between two endpoints, for instance, when a call is answered at an endpoint, Cisco CallManager selects an appropriate payload type (codec) for the media stream. Cisco CallManager compares the media capabilities of both endpoints involved in the call, and selects the appropriate common payload type and payload parameters to use. Payload selection is based on endpoint capabilities and location, although there may be other criteria added to this selection logic in the future. The important thing to understand is that endpoints are not dynamically involved in selecting payload types on a call-by-call basis.

Receive Channel Allocation

If Cisco CallManager is able to find a common payload type for the RTP stream between the two endpoints, it requests that each endpoint create a logical “receive channel,” that is, a unique IP address and port at which the endpoint will receive RTP data for the call. Each endpoint returns an IP address and port to Cisco CallManager in response to this request.

Currently, only IP telephones and gateways perform this step. Cisco CallManager requires JTAPI applications to specify a fixed IP address and port during initialization. Therefore, JTAPI applications cannot terminate more than one media stream simultaneously for the same endpoint. Applications that wish to terminate multiple media streams must register multiple endpoints simultaneously.

If the endpoint does not respond to the open receive channel request quickly enough, Cisco CallManager disconnects the call. Because JTAPI applications always supply an IP address when registering CiscoMediaTerminals, calls to application-controlled endpoints will not be disconnected for this reason. However, if Cisco CallManager cannot find a common payload type between the two endpoints involved in the call, Cisco CallManager disconnects the call.

Starting Transmission and Reception

After Cisco CallManager receives channel information for both parties, it informs each endpoint of the codec parameters that it selected for the RTP stream and the destination address for the other endpoint. This information is conveyed in two messages to each endpoint: a start transmission message and a start reception message.

JTAPI applications receive the CiscoRTPOutputStartedEv and CiscoRTPInputStartedEv events that contain all of the codec parameters necessary for sending and receiving RTP data.

Stopping Transmission and Reception

When the RTP stream must be interrupted because of a feature such as hold or disconnect, Cisco CallManager requests that each endpoint stop its transmission and reception of RTP data. Just as when the media flow is started, the stop transmission and stop reception messages are sent separately.

JTAPI applications receive the CiscoRTPOutputStoppedEv and CiscoRTPInputStoppedEv.

CiscoMediaTerminal

In JTAPI, the terminal object represents the logical endpoint for a call and is presumed to be able to receive and transmit data (digitally-encoded voice samples, for example). Thus, Cisco IP Phones are represented by terminals in JTAPI. Even though gateways terminate media, they are not represented by terminals, however. The CiscoMediaTerminals in particular represent a special kind of endpoint for which applications take responsibility for media termination.

There are four steps associated with using CiscoMediaTerminals:

- Provisioning
- Registration
- Adding Observers
- Accepting Calls

Provisioning

CiscoMediaTerminals are analogous to physical terminals and must be provisioned accordingly in Cisco CallManager, even though they do not represent actual hardware IP telephones or gateways. Just as IP telephones must be added to Cisco CallManager database using the Device Wizard, CiscoMediaTerminals are added the same way so that Cisco CallManager can associate the application's endpoint with a directory number and other call control properties such as call forwarding. There is no device type called “CiscoMediaTerminal” in the DeviceWizard. Instead, Cisco CallManager has one or more device types that support application registration—each of these types are be exposed as a CiscoMediaTerminal through JTAPI. Currently, only the device type “CTI port” is represented as a CiscoMediaTerminal in JTAPI.

The steps for provisioning a CTI port for use as an application-controlled endpoint are as follows:

1. Within the Cisco CallManager configuration web pages, add a “CTI port” device from the Device-Phone web page using the Device Wizard. The CTI port's device name will be the name of the corresponding CiscoMediaTerminal in JTAPI.
2. Add the new CTI port device, using the User-Global Directory web page, to the list of devices controlled by the application using the User web page.

Registration

After a media termination device has been properly provisioned in Cisco CallManager, the application may obtain a reference to the corresponding CiscoMediaTerminal object by using either the `Provider.getTerminal()` method or `CiscoProvider.getMediaTerminal()` method. The difference between the two methods is that the `CiscoProvider.getMediaTerminal()` method only returns CiscoMediaTerminals, whereas `Provider.getTerminal()` will return any terminal object associated with the provider, including those representing physical IP telephones.

Use the `CiscoMediaTerminal.register()` method to notify Cisco CallManager of their intent to terminate RTP streams of certain payload types. The `CiscoMediaTerminal.register()` method takes an IP address, a port number, and an array of `CiscoMediaCapability` objects that indicate the types of codecs supported by the application as well as codec-specific parameters.

The IP address and port indicate the address where the application desires to receive media streams. The sample code below demonstrates how to register a CiscoMediaTerminal and bind it to a local address, port number 1234:

```
CiscoMediaTerminal registerTerminal ( Provider provider, String terminalName ) {
    final int PORT_NUMBER = 1234;
    try {
        CiscoMediaTerminal terminal = provider.getTerminal ( terminalName );
        CiscoMediaCapability [] caps = new CiscoMediaCapability [1];
        caps[0] = CiscoMediaCapability.G711_64K_30_MILLISECONDS;
        terminal.register ( InetAddress.getLocalHost (), PORT_NUMBER, caps );
    }
    catch ( Exception e ) {
        return null;
    }
}
```

In order for this sample code to work, the specified provider must be `IN_SERVICE`. Further, note that this code uses the constant `CiscoMediaCapability.G711_64K_30_MILLISECONDS`. This is actually a static reference to a `CiscoG711MediaCapability` object that specifies a 30 millisecond maximum RTP packet size. This and other common media formats are predefined in the `CiscoMediaCapability` class.

To specify a media payload that is not listed in the `CiscoMediaCapability` class, there are two options. If the desired payload type is a simple variation of one of the existing subclasses of `CiscoMediaCapability`, then constructing a new instance of the subclass is all that is needed. For instance, if an application is willing to support G.711 payloads with a 60-millisecond maximum RTP packet size, it can construct the `CiscoG711MediaCapability` object directly, specifying 60 milliseconds in the constructor.

On the other hand, if there is no existing subclass of `CiscoMediaCapability` that matches the desired payload type, construct an instance of the `CiscoMediaCapability` class directly. Note that the only other parameter that may be specified when constructing a `CiscoMediaCapability` is the maximum packet size, for example, 30-milliseconds. The following code illustrates registering a custom payload capability:

```
CiscoMediaTerminal registerTerminal ( Provider provider, String terminalName ) {
    final int PORT_NUMBER = 1234;
    try {
        CiscoMediaTerminal terminal = provider.getTerminal ( terminalName );
        CiscoMediaCapability [] caps = new CiscoMediaCapability [1];
        caps[0] = new CiscoMediaCapability (
            RTPPayload.G728,
            30 // maximum packet size, in milliseconds
        );
        terminal.register ( InetAddress.getLocalHost (), PORT_NUMBER, caps );
    }
    catch ( Exception e ) {
        return null;
    }
}
```

The payload type parameter used for constructing the `CiscoMediaCapability` object corresponds to the payload field in the RTP header. The `RTPPayload` interface defines a number of well-known payload types for this purpose.

Adding Observers

In order to receive events that indicate where and when to transmit and receive RTP data, place a `CiscoTerminalObserver` on the `CiscoMediaTerminal`. Note that the `CiscoTerminalObserver` extends the standard JTAPI `TerminalObserver` interface without defining any new methods; it is a marker interface that signals the application's interest in receiving RTP events.



Note

This is a `TerminalObserver`, not a `CallObserver`, and therefore it must be added using the `Terminal.addObserver()` method, not the `Terminal.addCallObserver()` method.

Additionally, add a `CallControlCallObserver` to the `Address` object associated with the `CiscoMediaTerminal`. This guarantees that the application will be notified when calls are offered to the `CiscoMediaTerminal`. Unlike regular IP telephones, which automatically accept any offered call, `CiscoMediaTerminals` are responsible for accepting, disconnecting (rejecting), or redirecting any call that is offered to it. Since the `CallCtlConnOfferedEv` is only presented to `CallControlCallObservers` placed on `Address` objects, not `Terminal` objects, it is important that the application places its `CallControlCallObserver` in the correct place.



Note

Be sure to implement the `CallControlCallObserver` interface, not just the `CallObserver` interface; the `CallCtlConnOfferedEv` will not be delivered to observers that implement only the core `CallObserver` interface.

Accepting Calls

When an inbound call arrives at the CiscoMediaTerminal's address, it must be accepted using the `CallControlConnection.accept()` method before a terminal connection is created. This is not necessary for outbound calls—the connection will be in the `CallControlConnection.ESTABLISHED` state as soon as the call has progressed beyond digit recognition. Once the connection has been accepted, answer the ringing terminal connection to start media flow. Assuming that Cisco CallManager is able to match the capabilities that were registered with the capabilities of the calling endpoint, Cisco CallManager sends the Media Flow events so that the application can begin transmitting and receiving RTP data.

Receiving and Responding to Media Flow Events

Whenever a media stream must be created between two endpoints, Cisco CallManager issues start transmission and start reception events to both endpoints. In JTAPI, the `CiscoRTPOutputStartedEv` and `CiscoRTPInputStartedEv` events represent the start transmission and start reception events. The `CiscoRTPOutputStartedEv.getRTPOutputProperties()` method returns a `CiscoRTPOutputProperties` object, from which the application can determine the destination address of its peer endpoint in the call, as well as the other RTP properties for the stream such as payload type and packet size. Similarly, the `CiscoRTPInputStartedEv.getRTPInputProperties()` method returns a `CiscoRTPInputProperties` object which informs the application of the RTP characteristics for the inbound stream.

At any time while media is flowing, the current `CiscoRTPOutputProperties` and `CiscoRTPInputProperties` are also available from the `CiscoMediaTerminal.getRTPOutputProperties()` and `CiscoMediaTerminal.getRTPInputProperties()` methods as well. These methods throw an exception if the `CiscoMediaTerminal` is not currently supposed to transmit or receive media.

When Cisco CallManager wants the application to stop sending or receiving media as the result of a call disconnecting or being put on hold, for example, it sends the `CiscoRTPOutputStoppedEv` and `CiscoRTPInputStoppedEv` events. These events mean that the current RTP media stream that exists between the two endpoints should be torn down.

Inbound Call Media Flow Event Diagram

[Table 1-5](#) illustrates the dialogue between Cisco CallManager and a JTAPI application when a call is presented to an application-controlled endpoint. The events in the left column are JTAPI events sent to the `CallObserver` of the application, and the requests in the left column represent methods invoked by the application.

Table 1-5 Inbound Media Flow Event

JTAPI Event	Direction	Application Request
CallActiveEv		
ConnCreatedEv	---->	
ConnProceedingEv		
CallCtlConnOfferingEv		
	<----	CallControlConnection.accept ()
CallCtlConnAlertingEv		
TermConnCreatedEv	---->	
TermConnRingingEv		

Table 1-5 Inbound Media Flow Event (continued)

JTAPI Event	Direction	Application Request
	<---	TerminalConnection.answer ()
ConnConnectedEv CallCtlConnEstablishedEv TermConnTalkingEv CiscoRTPOutputStartedEv CiscoRTPInputStartedEv		
	<---	CallControlConnection.disconnect ()
CiscoRTPOutputStoppedEv CiscoRTPInputStoppedEv TermConnDroppedEv CallCtlConnDisconnectedEv	---->	

**Note**

Only JTAPI events for the local connection are shown here, that is, for the application endpoint. The actual JTAPI metaevent stream will contain events that describe the state of the calling party.

Cisco IP Telephony Solutions RTP Implementation

The Cisco IP Telephony Solutions architecture puts a premium on performance, and thus some of the features of RTP and its often-associated real-time control protocol (RTCP) are not implemented by Cisco IP Telephony Solutions phones and gateways. To ensure its compatibility, applications must consider the following points:

- As RTCP is not supported, Cisco IP Telephony Solutions endpoints will not send RTCP messages, and they will ignore any such messages sent to them.
- Cisco IP Telephony Solutions endpoints do not currently make use of the synchronization source (SSRC) field in the RTP header, but may in the future. Applications must not multiplex RTP streams using the SSRC field, or phones and gateways may not correctly decode and present the media.

Redirect

JTAPI 1.2 specifies that one of the pre-conditions of the `CallControlConnection.redirect()` method is for the state of the connection to be in either the `CallControlConnection.OFFERING` or the `CallControlConnection.ALERTING` state. Cisco JTAPI also allows a connection in the `CallControlConnection.ESTABLISHED` state to be redirected.

The `redirect()` method has the following overloaded form in the `CiscoConnection` interface. It allows applications to specify the behavior desired when there is a failure while redirecting a call, specifying the calling search space or resetting the original called field. Applications choose the desired behavior, by passing one of the following INT parameters in the overloaded `redirect` method from the `CiscoConnection` interface:

- Redirect drop on failure—When a call is directed to a busy or an invalid destination, Cisco CallManager can either drop the call if the redirect fails or leave the call at the redirect controller. The JTAPI application then has a chance to take corrective action, such as redirecting the call to another destination. The option for the redirect mode parameter are as follows:
 - CiscoConnection.REDIRECT_DROP_ON_FAILURE
 - CiscoConnection.REDIRECT_NORMAL
- Calling Address search space—Redirect uses the calling search space parameter to indicate which callingSearchSpace is used. Applications can either use the calling party's search space or the redirect controller's search space. The parameter options for this are as follows:
 - CiscoConnection.CALLINGADDRESS_SEARCH_SPACE
 - CiscoConnection.ADDRESS_SEARCH_SPACE
- Resetting original called—The called address option parameter is used to reset the original called fields. The options for this are as follows:
 - CiscoConnection.CALLED_ADDRESS_UNCHANGED
 - CiscoConnection.CALLED_ADDRESS_SET_TO_REDIRECT_DESTINATION. This option affects the fields when the call arrives at the redirect destination.

For more information, refer to the `com.cisco.jtapi.extensions.CiscoConnection` documentation.

Routing

Introduction

Routing in JTAPI requires the configuration of a CTI Route Point on the Cisco CallManager. Multiple calls can be queued to this Route Point, but only a single line can be configured on a CTI Route Point device. JTAPI implementation of adjunct Routing as described in the call center package includes the following:

- Registering route callbacks on Route Addresses
- Creating appropriate handlers in response to the various routing events (`routeSelect`, `routeEnd`)



Note CTI Route Points are devices with the ability to process any number of incoming calls simultaneously on the same line. Calls may be routed using the methods in the `javax.telephony.callcenter` package, or calls may be accepted, redirected or disconnected using the methods in the `javax.telephony.callcontrol` package. Each CTI Route Point may be configured with only one line in the Cisco CallManager. For details on how to configure and administer the CTI Route Point, refer to the *Cisco CallManager Administration CCMAdmin*.

Cisco Route Session Implementation

When a call comes into the `RouteAddress`, the implementation starts a `Route Session` thread and sends the application a `RouteEvent`. This thread in turn starts a timer thread to time the application's response to a `RouteEvent` with either a `routeSelect()` or an `endRoute()`. If the application responds with a `routeSelect (String[] selectedRoutes)`, JTAPI verifies that all preconditions are satisfied and then attempts to route the call to the first destination specified in the array. If the destination is a valid and

available number, the call is routed and the application gets a `RouteUsedEvent` followed by a `RouteEndEvent`. Otherwise, if there is an error in routing (which may be caused by an invalid/busy/unavailable destination), the application gets a `ReRouteEvent`. JTAPI starts the Timer Thread again before it sends the re-Route Event. As Cisco CallManager does not support re-Routing, if the routing was unsuccessful, the caller will either hear a busy tone or the call will be dropped. The application can clean up all failure instances and/or send JTAPI an `endRoute` to clean up the `RouteSession`. If the application does not respond with an `endRoute()`, the JTAPI timer once again expires and JTAPI cleans up the Route Session by sending the application a `RouteEndEvent()`.

If the routing timer expires before the application returns with a `selectRoute()` or an `endRoute()` method the Cisco CallManager applies same treatment as when a call is made to an unregistered phone (that is play fast busy). If `ForwardNoAnswer` is configured on the Route Point, the call is immediately forwarded to that number upon the expiration of the timer.

If the application is unable to respond with a valid address to route the call to, the application may choose to call `endRoute` with an error. The JTAPI specification defines three errors in the `RouteSession` interface. These are: `ERROR_RESOURCE_BUSY`, `ERROR_RESOURCE_OUT_OF_SERVICE` and `ERROR_UNKNOWN`. If an `endRoute` is invoked on the `RouteSession`, the implementation currently accepts() the call at the `RouteAddress` so the caller may begin to hear ringback. Thereafter if forwarding is configured for the Route Point, the call will be forwarded when the Forwarding Timer expires.

Select Route Timer

This timer is configurable via the `JTAPI.ini` configuration file which has a key called `RouteSelectTimeout=5000`. The unit is milliseconds. The default value for this timer is set to 5 seconds, however, depending on the needs of the application, this timer can be extended/decreased to improve Route Session cleanup efficiency. This timer should not be unreasonably large. Each Route Session is a thread and represents a call to the Route Point and we want these Route Sessions to be cleaned up. Should an application expect significant delays between receiving the Route Event and responding with a `routeSelect/endRoute` event, the application would want to appropriately extend this timer.

Forwarding Timer

The timer for Forward on No Answer is currently system wide (i.e., it applies to all devices on Cisco CallManager) and can be configured via the Cisco CallManager Service Parameters configuration. The default value for this timer is set to 12 seconds. In future releases a separate timer for CTI Route Points will be included so that Forwarding for the Route Point takes into effect immediately after the call is accepted by JTAPI (when the application calls an `endRoute` or if the routing timer expires).

RouteSession Extension

`CiscoRouteSession` is a Cisco Extension to the JTAPI specification. Most importantly this extension exposes the underlying Call object to the Applications. `CiscoRouteSession.getCall()` returns `CiscoCall` and this call exposes other Call Model Objects such as the associated Addresses, Connections etc. The extension also defines additional errors for the application. There is a plan to replace these with an overloaded `endRoute(error, action)` to specify the call disposition. These include error and action codes—for example the application might want to end the Route Session and trigger the default Forward on No Answer to kick in, in which case the implementation will accept() the call.

Caller Options Summary

In the absence of a callback or in the scenario where a routeEvent has not been responded to with a RouteSession.routeSelect() or endRoute(), the caller hears nothing until:

- The application can disconnect() or reject() the connection on the Route Point and thereby the caller hears a busy tone.
- The application can accept the call and then the Forward No Answer if configured will kick-in.
- The application can drop the call. The caller holds the receiver with no clue of what happened.

With a callback, if the application chooses to call an endRoute(), then after endRoute() returns, the caller hears a ringback until:

- client calls a disconnect() which would drop the call.
- The client redirects() the call.
- The forward on no answer timer configured via the scm.ini will kick in and would forward the call unless the above two have already kicked in.
- If no forwarding is configured for the Route Point, the caller continues to hear a ringback unless the first two options kick in.

Fault Tolerance When Using Route Points

One way for an application that uses route points to deal with fault tolerance is to have two JTAPI applications connected to two different Cisco CallManagers, each registering a different RouteAddress. For example, Application1 manages RouteAddress1 using CallManager1. Application2 manages RouteAddress2 using CallManager2. In the Cisco CallManager Administration, the ForwardNoAnswer configuration for these CTI Route Points must be administered so they point to each other. In this example, RouteAddress1 would have FNA=RouteAddress2 and RouteAddress2 would have FNA=RouteAddress1. If CallManager1 goes down, calls are forwarded to RouteAddress2 so that Application2 takes over. Furthermore, both applications could be configured to reconnect to their proper Cisco CallManager server when they receive a ProviderShutdown event.

Redundancy

Cluster Abstraction

The CTIManager is a virtual representation of all the Cisco CallManagers in a cluster. Cisco JTAPI applications communicate with the CTIManager instead of with a specific Cisco CallManagers. The CTIManager also maintains connection between Cisco CallManagers in a cluster. This allows a provider to represent any devices in the cluster under the CTIManager. [Figure 1-5](#) illustrates “Single box configuration with JTAPI, Cisco CallManager and CTIManager in one box”. [Figure 1-6](#) illustrates “Redundant Cisco CallManager and CTIManagers with JTAPI deployed as a separate client”

For more details about the cluster administration and device pool settings, refer to the Cisco CallManager help pages.

Figure 1-5 Single box configuration with JTAPI, Cisco CallManager and CTIManager in one box

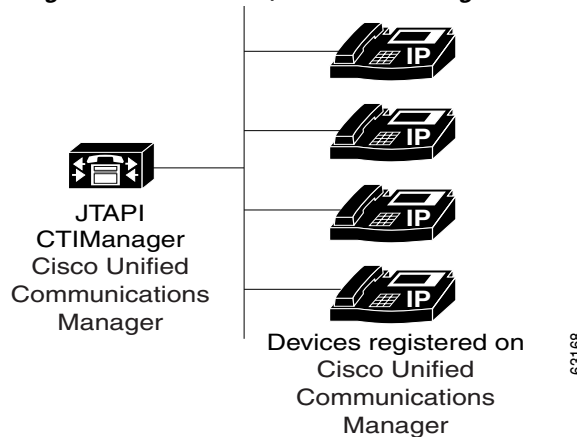
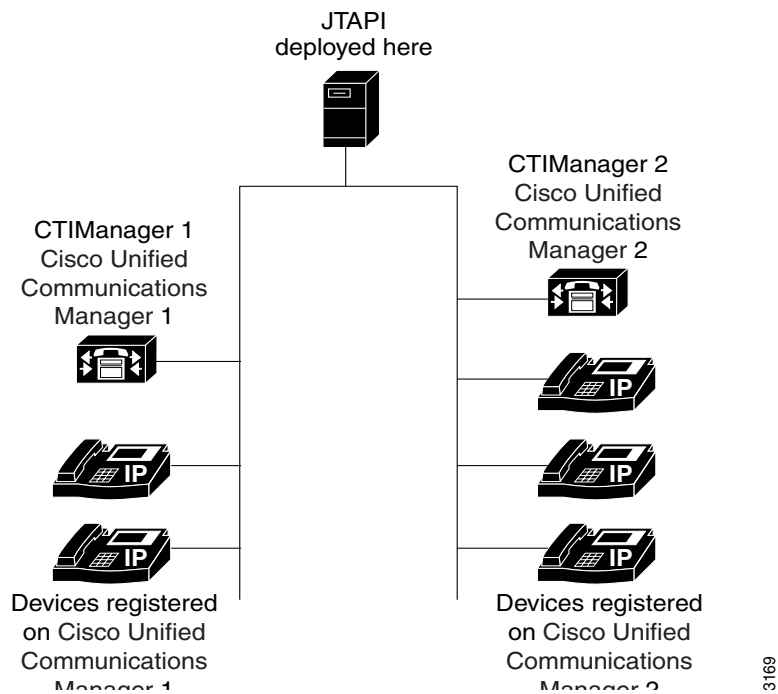


Figure 1-6 Redundant Cisco CallManager and CTIManagers with JTAPI deployed as a separate client



Note

In previous releases of Cisco CallManager, applications running on Cisco JTAPI could only control or monitor devices registered under a single Cisco CallManager. If a Cisco CallManager server went down, the connection between the Cisco CallManager server and JTAPI would terminate and the Provider would shutdown. CTIManager service was only introduced with Cisco CallManager Release 3.1.

Redundancy in Cisco CallManager

Devices are configured into device pools, and are assigned static Cisco CallManager groups. Devices register with a particular Cisco CallManager server which handles call control signaling. When a Cisco CallManager server fails, the devices failover to the backup Cisco CallManager server in the group. When the primary Cisco CallManager comes back online and it waits until there are no active calls on the device, then re-homes back to the primary Cisco CallManager server. Cisco JTAPI informs the applications of this transition by sending a temporary out-of-service message while registering to the backup Cisco CallManager server.

Cisco CallManager Failure

When a Cisco CallManager server fails, devices associated with it re-home to the next Cisco CallManager server in the group. This is defined in the prioritized list of Cisco CallManagers in the device pool information configuration for each device. Therefore, failure of a Cisco CallManager server only results in a partial outage of devices in the cluster. Those devices are available following a successful Cisco CallManager failover and registration with a secondary Cisco CallManager.

**Note**

A device such as a Cisco IP Phone 7960 fails over to a secondary Cisco CallManager server only when there are no calls active on that device. The failure of a Cisco CallManager server during a call results only in termination of observation of that device. The media path continues to exist, but without any further call control features.

Cisco JTAPI communicates this partial outage to applications using `CiscoAddrOutOfServiceEv` and `CiscoTermOutOfServiceEv` events. When the Cisco CallManager fails over, the device must successfully register to the secondary Cisco CallManager before the device is available to the JTAPI applications. Cisco JTAPI will send the `CiscoAddrInServiceEv` and `CiscoTermInServiceEv` events.

The Provider remains in service during this time. Devices on other Cisco CallManager servers are available for call control. The events are sent on callbacks of the respective Address or Terminal observer objects. `CiscoAddrOutOfServiceEv` and `CiscoAddrInServiceEv` events are sent to an object implementing the `AddressObserver` and added to an Address using the `addressChangedEvent()` callback object method. The `CiscoTermOutOfServiceEv` and `CiscoTermInServiceEv` events are sent to an object implementing the `TerminalObserver` interface and added to a Terminal using the `terminalChangedEvent()` call back method.

If the devices are currently in a call, a `CallObservationEnded` message is sent on the `CallObserver` `callChangedEvent()` callback, followed by the `CiscoAddrOutOfServiceEv` and `CiscoTermOutOfServiceEv` messages.

**Note**

Applications must monitor for and respond to the `CiscoAddrOutOfServiceEv`, `CiscoTermOutOfServiceEv`, `CiscoAddrInServiceEv`, and `CiscoTermInServiceEv` events before the calling call control functions on the Address or Terminal. Applications that do not support this may encounter unexpected errors because the applications are not aware of the exact state of the system.

Redundancy in CTIManagers

Cisco JTAPI also offers applications transparent redundancy via the CTIManager. When the primary CTIManager fails, Cisco JTAPI automatically connects to the backup CTIManager and communicates the reconnection to applications. Instead of connecting to a single Cisco CallManager server, applications now connect to a set of CTIManagers. The CTIManager server names are supplied by the applications when invoking JTAPI.

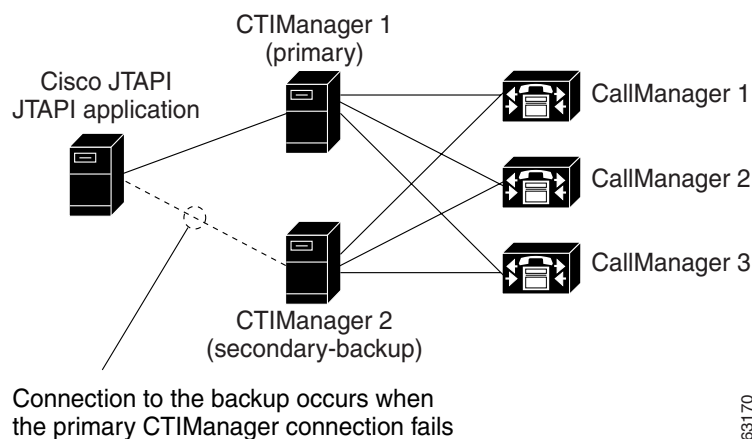
Cisco JTAPI and the CTIManager maintain bidirectional heartbeat signals to detect a loss of connectivity between them. The CTIManager detects when an application is no longer running and cleans up its allocated resources. Figure 1-7 illustrates the “Logical representation of JTAPI, CTIManager and Cisco CallManager in a cluster”



Note

Applications in Cisco CallManager Release 3.0 should stop interpreting the out-of-service state as a dead CTI connection. After Cisco JTAPI successfully connects to the primary CTIManager, it will alternately attempt to reconnect to the primary or backup CTIManager if JTAPI’s connection to the CTIManager fails.

Figure 1-7 Logical representation of JTAPI, CTIManager and Cisco CallManager in a cluster



Invoking CTIManager Redundancy

When you call a `getProvider()` method on the `CiscoJtapiPeer` during application startup, Cisco JTAPI attempts a connection to the first CTIManager in the list. If the first CTIManager is not available or if a connection is refused by the CTIManager, an exception is sent to the application and there are no further reconnection attempts. After the first successful connection, Cisco JTAPI attempts to connect to the backup or primary CTIManager alternately when a failure to the CTIManager or connection to the CTIManager is detected.

The list of redundant CTIManagers is a comma-separated list passed into the `CiscoJtapiPeer.getProvider(String providerString)` method as a `String`. The usage for the `providerString` is as follows:

- `providerString = CTIManager;login=XXX;passwd=YYY;appinfo=ZZZ` (Non-redundant feature)
- `providerString = CTIManager1,CTIManager2;login=XXX,passwd=YYY;appinfo=ZZZ` (Redundant feature)

**Note**

The appinfo parameter is optional. If no specific appinfo parameter is provided by the application, Cisco JTAPI generates one from a JTAPI instance ID and the local host name.

Additionally, different CTIManager lists may be defined in the jtapi.ini file to support the CiscoJtapiPeer.getServices() method. Cisco JTAPI accepts the following definition:

```
CTiManagers=<CTIManager1>,<CTIManager2>;<CTIManager3>
```

where

<CTIManager1>,<CTIManager2> are a redundant group.

<CTIManager3> is a non redundant group.

CTIManager Failure:

When Cisco JTAPI detects a loss of connection to a CTIManager, the application is notified of this loss in service. The following events are sent to the application on the appropriate Observers:

- A CallObservationEndedEv event is sent to all Call Observers and calls in progress are ended. The calls are physically connected, but the applications' observation of the call ends since Cisco JTAPI does not have the ability to send call state changes.
- The Provider is set in the out-of-service state and the ProvOutOfServiceEv event is delivered on any ProviderObserver callbacks present on the Provider.
- A CiscoAddrOutOfServiceEv event is sent to all Addresses that have an active AddressObserver and a CiscoTermOutOfServiceEv event is sent to all Terminals that have an active TerminalObserver.

Cisco JTAPI then attempts a connection to the next CTIManager in the list and reinstates the devices previously registered under the application control in the new CTIManager. Once the device is reinstated, the ProvInServiceEv, CiscoAddrInServiceEv and CiscoTermInServiceEv events are sent to the application via the respective Observers. All previously added call Observers are maintained. If there are any existing calls on the devices, a snapshot of the call is sent to the respective CallObservers.

**Note**

Unlike physical devices like the Cisco IP 7960 phone, Cisco JTAPI does not automatically re-home to the primary CTIManager when it comes back up after a failure. Re-homing to the primary CTIManager is only triggered by a failure of the backup CTIManager.

**Note**

On a Cisco CallManager failure, applications lose call control capability. Devices retain call control capability but the applications' ability to observe call control is lost. All call states maintained in the CTIManager are also lost. The reconnection to the backup CTIManager is akin to a fresh connection to the CTIManager, where JTAPI has to initiate queries on devices. The reconnection is transparent to applications

The failover to the backup CTIManager happens as soon as the connection to the primary CTIManager is down. The ProviderRetryInterval is an important parameter in this process

- If the backup CTIManager is not available, Cisco JTAPI attempts to reconnect to the primary CTIManager at the ProviderRetryInterval.

- If there is no backup CTIManager, Cisco JTAPI attempts to reconnect to the single CTIManager at the ProviderRetryInterval.
- If a primary and a backup CTIManager are indicated at application startup, Cisco JTAPI connects to the primary CTIManager. If connection to the primary CTIManager goes down, Cisco JTAPI attempts to connect immediately to the backup CTIManager. If there is a further loss of connection to the backup CTIManager, JTAPI attempts to connect alternately to the primary and backup CTIManager at ProviderRetryInterval intervals

**Note**

Applications that have their own CTI reconnect logic should modify their logic to stop interpreting a ProvOutOfServiceEv event as a dead CTI connection that will not recover (the event that truly represents this state is ProvShutdownEv). Applications may require initial connect retry logic since the Provider is not created if all the CTIManager are unavailable at initial connection time.

Heartbeats

Cisco JTAPI and the CTIManager maintain heartbeat signals to discover a failure in either the CTIManager or JTAPI. The heartbeat is bidirectional with the CTIManager server controlling the heartbeat parameters. Applications can request a desired server heartbeat interval when initializing Cisco JTAPI, but the CTIManager can override it.

Applications specify the desired heartbeat parameter using DesiredServerHeartbeatInterval in the jtapi.ini setting.

Cisco JTAPI specifies the client's desired heartbeat interval during initialization. The CTIManager specifies to Cisco JTAPI the client side heartbeat interval and the interval at which the server (CTIManager) will send heartbeats. A failure to receive heartbeat message over twice the server-specified interval results in a client-initiated teardown of the connection. To minimize heartbeat traffic, any messages from the client to the server or events from the server to the client substitutes for a heartbeat.

Device Recovery

Automatic device recovery is supported as part of Cisco JTAPI.

Device Recovery for phones

For devices such as the Cisco IP Phone 7960, the re-homing feature is part of the device firmware. On a primary Cisco CallManager failure, the phone attempts to connect to the backup Cisco CallManager when it is no longer on a call. This transition is communicated to applications in the form of out-of-service and in-service events described in the [“CTIManager Failure:” section on page 1-29](#).

For virtual devices with no firmware such as CTI Ports and CTI RoutePoints, the failover is performed by the CTIManager or by Cisco JTAPI.

CTI RoutePoints

On a Cisco CallManager server failure, the CTIManager recovers the device from the Cisco CallManager server group as defined in the device pool administration for the CTI RoutePoint. When the primary Cisco CallManager server recovers, the CTIManager attempts to recover the device on its primary Cisco CallManager. This re-homing happens when there are no more calls on the device, (similar to physical devices). On a CTIManager failure, Cisco JTAPI recovers the device on the backup CTIManager. The application is informed of the availability of a device with the CiscoAddrOutOfServiceEv and CiscoAddrInServiceEv events.

CTI Ports

CTI Ports that are registered by an application have a mechanism similar to phone devices. When the Cisco CallManager that is handling signaling for a CTIPort fails, the CTIManager recovers its services according to the device pool administration for this device. On a CTIManager failure, Cisco JTAPI re-registers the CTI Port after it has connected to the backup CTIManager. The CiscoAddrOutOfServiceEv and CiscoTermOutOfServiceEv events and the corresponding in service events are sent after recovery of the CTI Port.

Media streaming for these devices is controlled by the application and continues even when the port is out of service. It is the application's responsibility to ensure that new calls do not get presented to the device until it is ready to accept them.

Directory Change Notification

Applications are notified asynchronously of device additions or deletions from the user control list and device deletions from the Cisco CallManager database. Applications also receive notification about line changes to a device. This notification is sent to Cisco JTAPI and is propagated to applications with CiscoAddrCreatedEv, CiscoAddrRemovedEv, CiscoTermCreatedEv, and CiscoTermRemovedEv on the AddressObserver and TerminalObservers respectively.

**Note**

The device needs to be registered for CTIPorts and CTIRoutePoints to receive the line change notification.

Call Forward Setting

Cisco JTAPI supports setting the Call Forward feature according to the JTAPI Specification. Cisco's JTAPI implementation does not support all the forwarding characteristics but supports only the FORWARD_ALL attribute for the Address. Applications can invoke setForwarding, getForwarding and cancelForwarding methods on a CallControlAddress object, but the CallControlForwarding instruction can only be of type FORWARD_ALL.

CiscoJtapiExceptions

Cisco JTAPI notifies applications of CTI-generated error codes. These codes are returned when there is an exception or error in the CTIManager. The CTI returned error code is propagated to the application separately. The application can extract the error code by invoking `getErrorCode()` method on the exception object. `CiscoJtapiException` is defined in the `com.cisco.jtapi.extensions` package.



Note

In Cisco CallManager Release 3.0, the CTI error code was passed to an application as a part of the exception string and the application parse the string to extract the error code.

Alarm Services

Support for sending alarms to a service is included as part of the general serviceability framework for Cisco IP telephony applications. The alarm components are defined in the `com.cisco.services.alarm` package.

An alarm interface and framework supports the sending of alarm notifications in XML over TCP to an Alarm Service available on the network in a Cisco JTAPI applications. The alarm package features include:

- XML definition of alarms, resolved by a catalog in the alarm service.
- A bounded rollover queue to buffer alarms at the sender.
- Alarm sending on a separate thread to avoid blocking at the sending application.
- A TCP-based reconnection scheme to the alarm service.

The overall framework of the Cisco JTAPI alarm system is similar to the existing JTAPI tracing package. Applications must instantiate an `AlarmManager` for a particular facility code from which alarm objects can be created. `DefaultAlarm` and `DefaultAlarmWriter` implementation classes are included as part of the implementation.

Application Control of JTAPI Parameters

The various parameters required for configuring Cisco JTAPI are located in the `jtapi.ini` file. Cisco JTAPI looks for the presence of this file in the current Java classpath.

Cisco JTAPI supports application setting of all the parameters required by Cisco JTAPI. This allows a single point of application administration, independent of `jtapi.ini`. The `jtapi.ini` file is a starting point for default values but client applications can modify different values without having to specifically modify the `jtapi.ini` file.

This feature allows applications to get a `CiscoJtapiProperties` object and to make changes to the parameters using the accessor and mutator methods. These properties apply peer-wide to all the providers derived from a `CiscoJtapiPeer`. Different instances of client applications can impose different settings for these parameters. The `CiscoJtapiProperties` interface is defined in the `com.cisco.jtapi.extensions` package.

Jtapi.ini Parameters

Cisco JTAPI is configured using parameters in the jtapi.ini. These parameters are modified using the Jtprefs application, which is installed by the Cisco JTAPI installer.

The following parameters are pertinent to Cisco JTAPI:

```
PROTOCOL_DEBUGGING=0
UseSameDirectory=1
JTAPIIMPL_DEBUGGING=0
UseSystemDotOut=0
QueueStatsEnabled=0
PeriodicWakeupInterval=50
RouteSelectTimeout=5000
UseTraceFile=1
ProviderOpenRequestTimeout=200
Directory=
DEBUG=0
DesiredServerHeartbeatInterval=30
AlarmServicePort=1444
CTI_DEBUGGING=0
SyslogCollector=
JTAPI_DEBUGGING=0
PeriodicWakeupEnabled=0
NumTraceFiles=10
AlarmServiceHostname=
MISC_DEBUGGING=0
TracePath=D:\JtapiTraces
UseAlarmService=0
CTIIMPL_DEBUGGING=0
WARNING=0
Traces=WARNING;INFORMATIONAL;DEBUG
INFORMATIONAL=0
UseSyslog=0
JtapiPostConditionTimeout=15
JTAPINotificationPort=789
FileNameBase=CiscoJtapi
CtiRequestTimeout=30
TraceFileSize=1048576
```

```
Debugging=JTAPI_DEBUGGING;JTAPIIMPL_DEBUGGING;CTI_DEBUGGING;CTIIMPL_DEBU  
GGING;PROTOCOL_DEBUGGING;MISC_DEBUGGING
```

```
FileNameExtension=log
```

```
QueueSizeThreshold=25
```

```
ProviderRetryInterval=30
```

```
CallManagers=Server1;Server2;Server1,Server2
```

```
SyslogCollectorUDPPort=514
```

Dynamic Trace Enabling Using Jtprefs

You will generally want to turn off high level debugging traces. Typically, tracing at full debug levels imposes a load on the system and is not desirable during normal system operation. For troubleshooting purposes, an administrator can turn tracing on or off dynamically.

Cisco JTAPI supports the dynamic enabling of traces from the Jtprefs administrative application. The application communicates with JTAPI using a TCP socket and sends a signal to enable or disable the traces. For more details on dynamic tracing, refer to the Cisco CallManager Administration Guide. It is necessary for the trace file generation to be turned on prior to using dynamic tracing. If no trace files are being generated, dynamic tracing does not enable file generation. Dynamic tracing only enables or disables traces on an existing log file stream. By default, JTAPI enables file writing with all tracing levels off.