An Introduction to the Java Telephony API (JTAPI)

Marcel Graf

Marcel Graf/Zurich/IBM@IBMCH
IBM Research Division
Zurich Research Lab

March 2000

1 Java Telephony API (JTAPI)

1.1 Scope

The Java Telephony API (JTAPI) is an object-oriented application programming interface for Java-based telephony applications. Similar APIs for other platforms are the Telephony API (TAPI) for the Microsoft Windows platform and the TSAPI for the Novell Netware platform.

1.2 JTAPI basics

1.2.1 First party vs. third party call control

The purpose of JTAPI is to serve as an interface between a Java application and a telephone system. The point where this interface is located determines the degree of control an application has. In a first-party call control scenario the interface is located at a terminal. The application has the same degree of control a normal telephone user has. In a third-party call control scenario the interface is located inside the telephone system. Depending on the telephone system this internal access provides the application usually with more control capabilities than a first-party call control scenario.

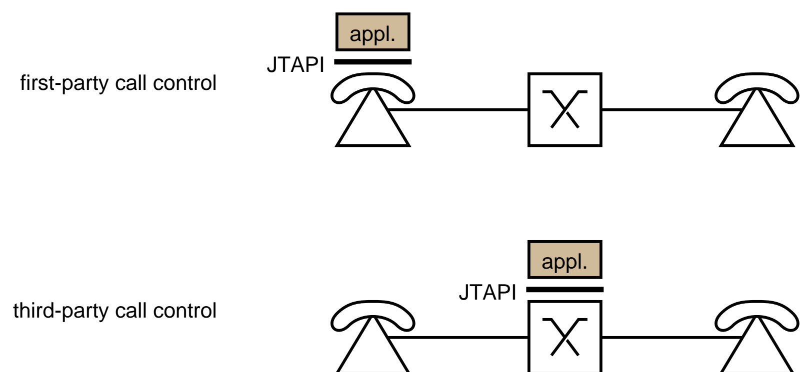


Figure 1: First-party and third-party call control in a Private Telephone Network with a PBX

A design goal of JTAPI has been to cover both scenarios. As a consequence JTAPI provides a model of the telephone system and of telephone calls that corresponds to the more general third-party view, even when JTAPI is used for first-party call

control. A third-party view of a call does not distinguish between the local end and the remote end of a call. Instead the two ends are symmetrical.

1.2.2 The JTAPI call model

1.2.2.1 Principles

Underlying the design of JTAPI is a *call model* which is a high-level, technology independent abstraction. It describes the call as a set of finite state machines (FSM) that undergo state transitions as the call evolves.

The call model is very general to cover many different call scenarios. It is able to describe for example

- A call between two parties,
- Multiple simultaneous calls on the same terminal,
- A conference between multiple parties,
- A call setup that alerts multiple terminals.

The call model describes the call as well as the call parties. Altogether it defines five base classes (in the general sense of object types). Two classes describe call parties. Their objects are persistent and independent of calls:

- A user is represented by an *Address* object. The main attribute of the Address object is the user identifier.
- A telephone terminal is represented by a *Terminal* object. The main attribute of the Terminal object is the terminal address.

The three other classes describe a call. Their object instances are not persistent, but created dynamically during a call. Each includes a finite state machine:

- A *Call* object is created for each call.
- A *Connection* object is created for each user participating in the call. It connects the user's Address object with the Call object.
- A *TerminalConnection* object is created for each terminal participating in the call. It connects the terminal's Terminal object with the Connection object.

1.2.2.2 Sample call configurations

This section presents selected call configuration examples which help to explain the call model. It starts with a basic two-party call and then extends the example with another call, another terminal and another user.

Two-party call

An example of a call with two participants is shown in Figure 2. For novices it may be surprising that in this simple case there are two Connection objects attached to the Call object, one for each participant. This representation is important for the seamless extension to the case of a conference call with three or more parties, which is shown in a different example further down. One should note that the model is completely symmetric (it does not distinguish between local and remote entities) because it provides a third-party view.

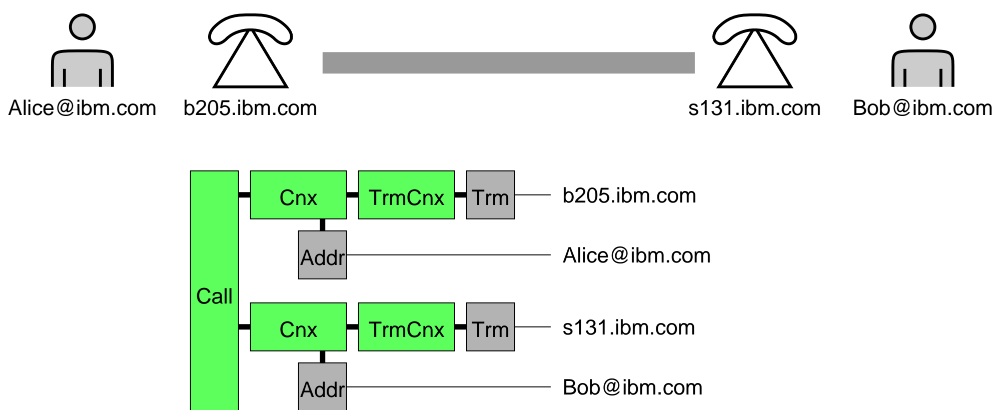


Figure 2: Call model for two-party call

Two simultaneous calls

An example of a user who has two simultaneous calls on the same terminal is shown in Figure 3. All call-related objects have doubled their number. The Address object and Terminal object of the user who has two calls exist once but are attached to two Connection and TerminalConnection objects.

Call setup with two alerting terminals

An example of a two-party call with two alerting terminals is shown in Figure 4. It motivates the separation of Terminal-Connection objects from Connection objects. In the example Bob has multiline appearance, that means that when Bob is called, several terminals are ringing. The multiline appearance is represented by the two TerminalConnection objects that attach to the Connection object of Bob, one for each terminal. When one of the terminals answers the call the other terminal is disconnected (in terms of the call model the TerminalConnection object goes into a disabled state).

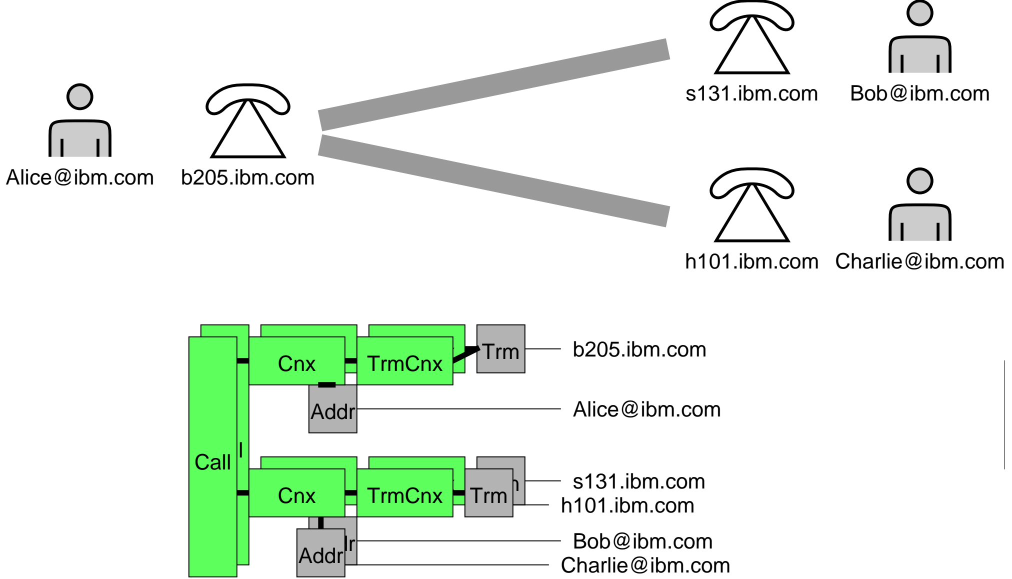


Figure 3: Call model for two simultaneous calls

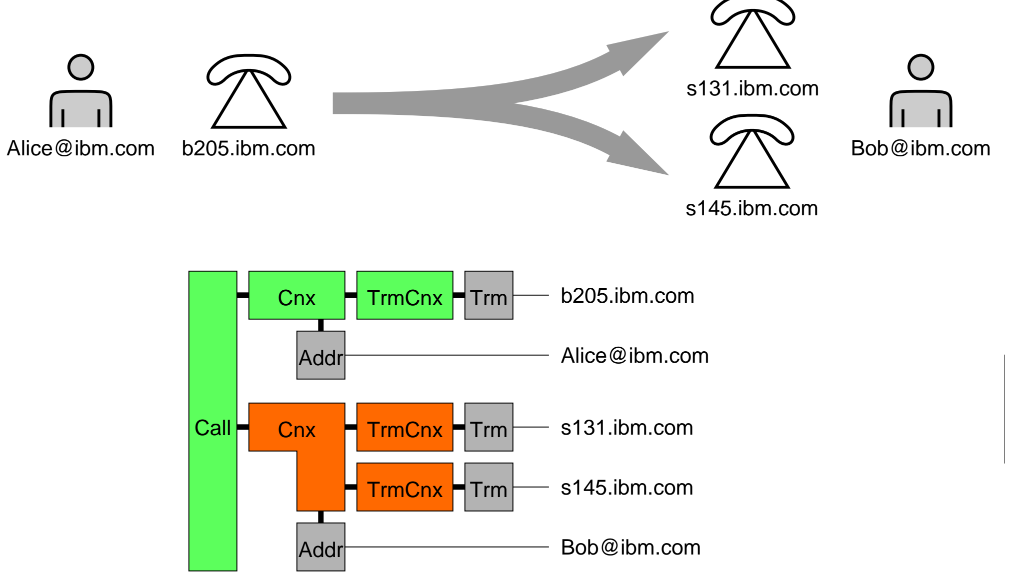


Figure 4: Call model for two alerting terminals

Three-party call

An example of a conference call with three participants is shown in Figure 5. It motivates the separation of Connection objects from Call objects. The call model turns out to be a straightforward extension of the basic call with two participants. The model simply adds a third leg with Connection, Address, TerminalConnection and Terminal objects for the third participant.

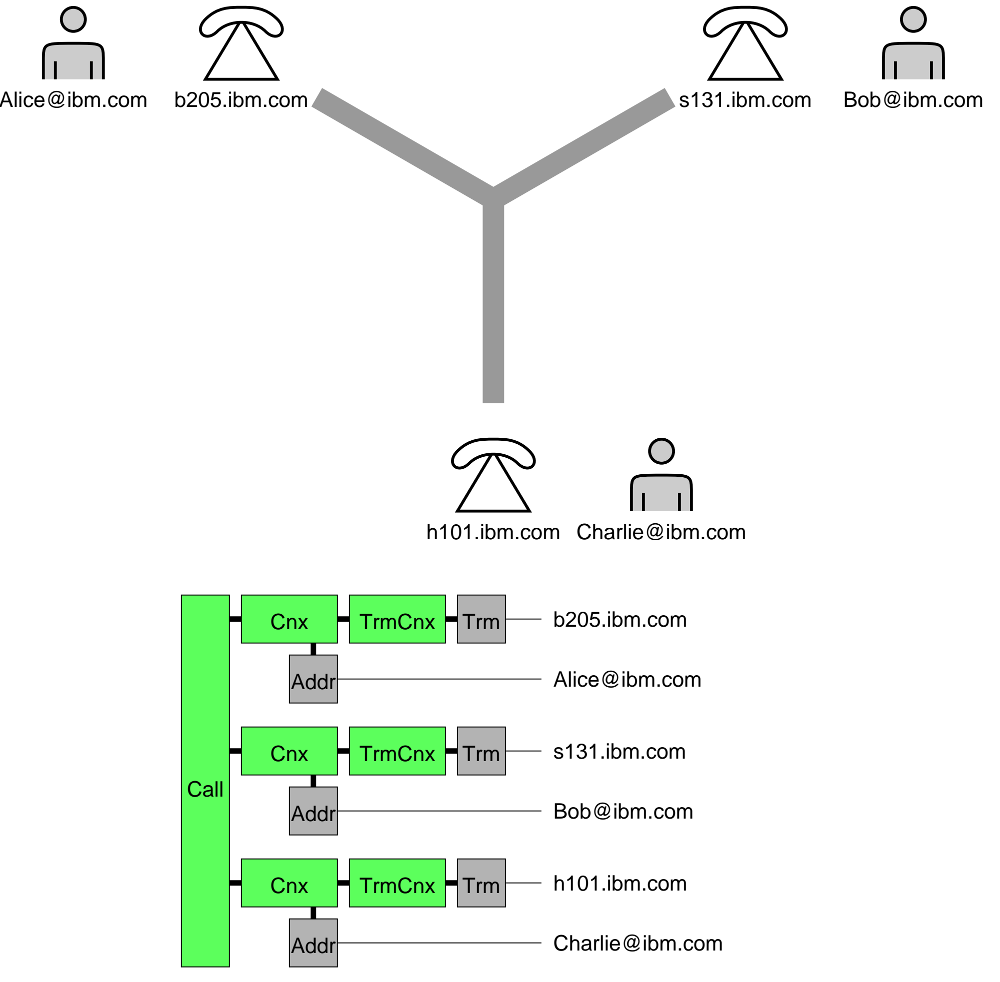


Figure 5: Call model for three-party call

1.2.2.3 Finite state machines

This section describes the call-related objects and their finite state machines in more detail.

Call object

A Call object is created for each call. The state of the Call object depends on the number of Connection objects.

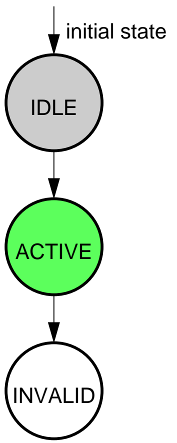


Figure 6: Finite state machine: Call object

State		Description
IDLE	Call	This is the initial state for all Calls. In this state, the Call has zero Connections.
ACTIVE	Call	A Call with some current ongoing activity is in this state. Calls with one or more associated Connections must be in this state.
INVALID	Call	This is the final state for all Calls. Call objects which lose all of their Connections objects (via a transition of the Connection object into the Connection.DISCONNECTED state) moves into this state. Calls in this state have zero Connections and these Call objects may not be used for any future action.

Connection object

A Connection object is created for each user participating in the call. It connects the user's Address object with the Call object.

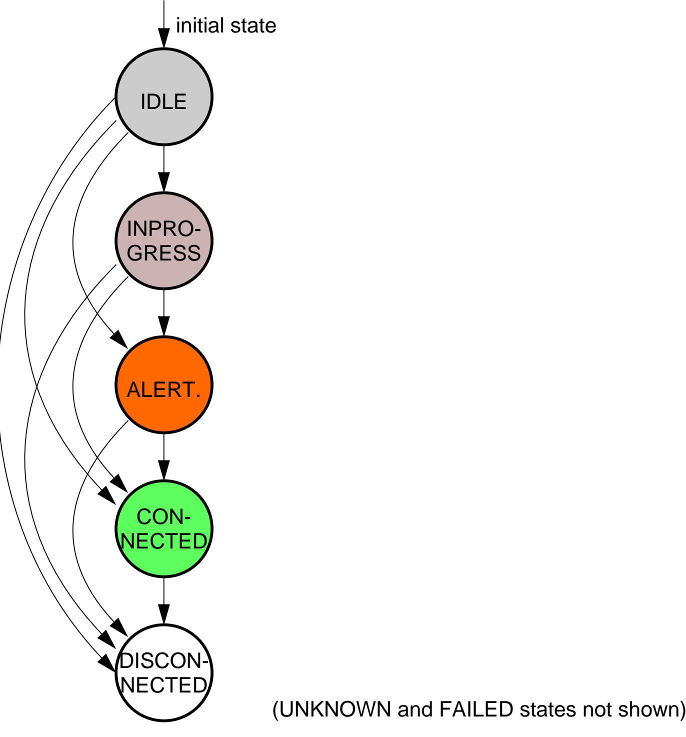


Figure 7: Finite state machine: Connection object

IDLE | Cnx

This state is the initial state for all new Connections. Connections which are in the Connection.IDLE state are not actively part of a telephone call, yet their references to the Call and Address objects are valid.

INPROGRESS | Cnx

This state implies that the Connection, which represents the destination end of a telephone call, is in the process of contacting the destination side. Under certain circumstances, the Connection may not progress beyond this state. Extension packages elaborate further on this state in various situations.

ALERTING | Cnx

This state implies that the Address is being notified of an incoming call.

CONNECTED Cnx

This state implies that a Connection and its Address is actively part of a telephone call. In common terms, two people talking to one another are represented by two Connections in the Connection.CONNECTED state.

DISCONNECTED Cnx

This state implies a Connection is no longer part of the telephone call, although its references to Call and Address still remain valid. A Connection in this state is interpreted as once previously belonging to this telephone call.

FAILED

This state indicates that a Connection to that end of the call has failed for some reason. One reason why a Connection would be in the Connection.FAILED state is because the party was busy.

TerminalConnection object

A TerminalConnection object is created for each telephone terminal participating in a Connection. It connects the terminal's Terminal object with the Connection object.

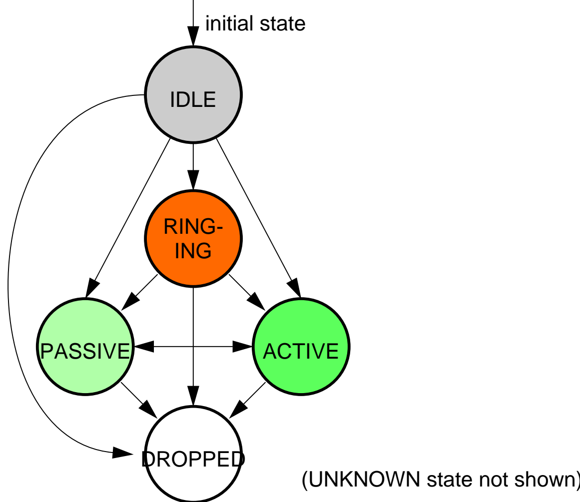


Figure 8: Finite state machine: TerminalConnection object

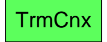
IDLE TrmCnx

This state is the initial state for all TerminalConnections. TerminalConnection objects do not stay in this state for long. They typically transition into another state quickly.

RINGING TrmCnx

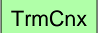
This state indicates the a Terminal is ringing, indicating that the Terminal has an incoming Call.

ACTIVE



This state indicates that a Terminal is actively part of a telephone call. This usually implies that the party speaking on that Terminal is part of the telephone call.

PASSIVE



This state indicates that a Terminal is part of a telephone call but not in an active fashion. This may imply that a resource of the Terminal is being used and may limit actions on the Terminal.

DROPPED

This state indicates that a particular Terminal has permanently left the telephone call.

1.2.2.4 Preconditions and postconditions

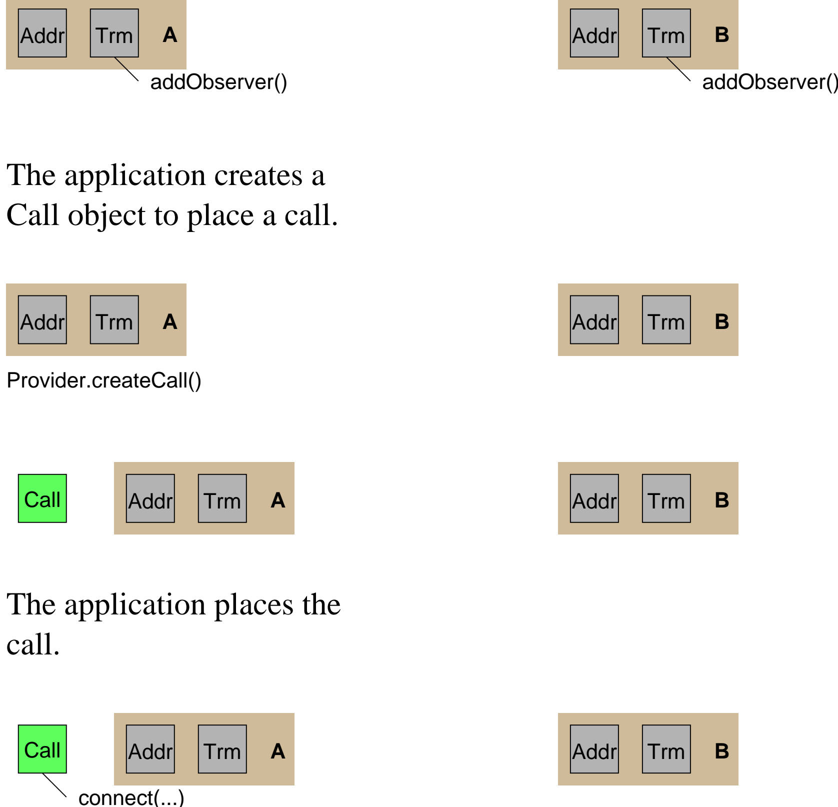
When looking at the call state at the level of an individual Call, Connection or TerminalConnection object one will notice that the state machines define many possible state transitions. When combining these state machines in a call and looking at the overall call state resulting from them it may appear that the engendered space is too large, i.e. it contains meaningless call states. To discard these meaningless states JTAPI specifies restrictions at a higher level in the form of pre- and postconditions for method calls on objects of the call model.

1.2.2.5 Sample call setup

This section shows the state transitions of the call model for a call setup. The scenario is a two-party call where user A calls user B. Shown are the call models of two JTAPI interfaces at the terminal of user A and the terminal of user B, respectively.

Call model at terminal A	**Signalling message**	**Call model at terminal B**

After initialization the service provider reports two objects in its address space: a Terminal representing the local terminal and an Address representing the local user. The application adds an Observer to the local Terminal.

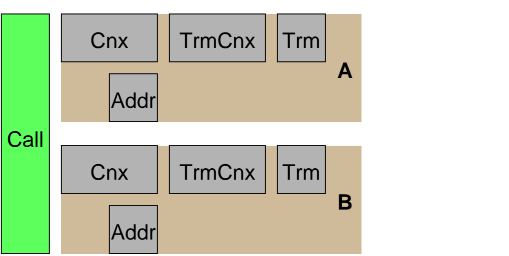


The application creates a Call object to place a call.

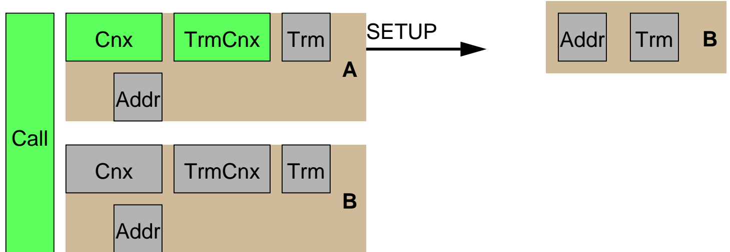
The application places the call.

Call model at terminal A | **Signalling message** | **Call model at terminal B**

The service provider creates a Call with two call legs, each with a Connection and TerminalConnection. After creation the FSMs are IDLE.

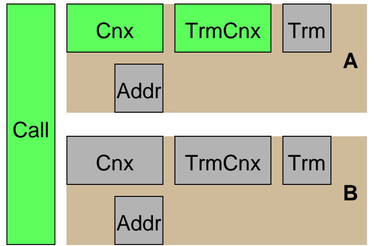


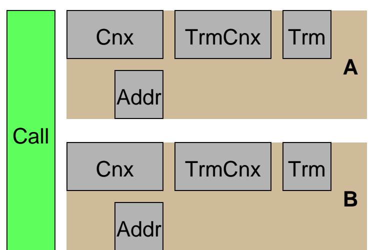
The local call leg transitions into CONNECTED/ ACTIVE. The protocol stack sends a SETUP message.



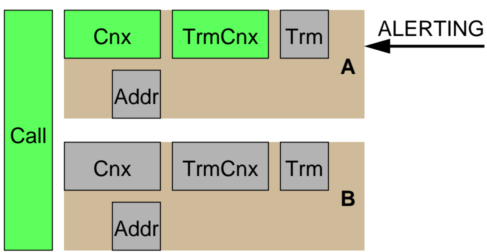
Call model at terminal A | **Signalling message** | **Call model at terminal B**

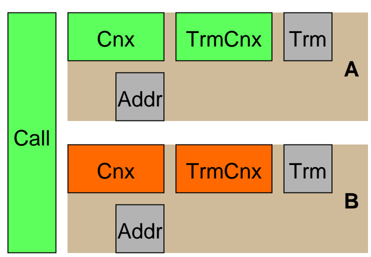
The service provider creates a Call with two call legs, each with a Connection and TerminalConnection. After creation the FSMs are IDLE.





The remote call leg transitions into CONNECTED/ ACTIVE. The local call leg transitions into ALERTING/RINGING. The protocol stack sends back an ALERTING message.

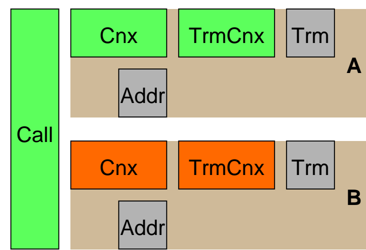


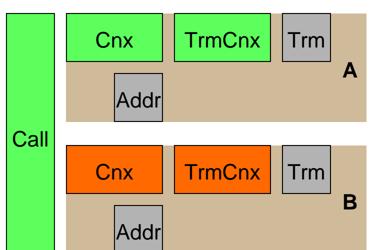


Call model at terminal A	Signalling message	Call model at terminal B

The remote call leg transitions into ALERTING/RINGING.

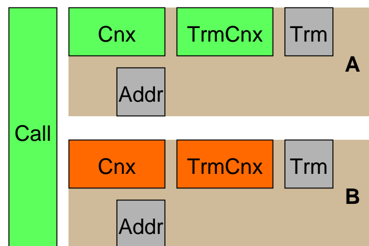
The application notifies the user that a call is coming in from A.

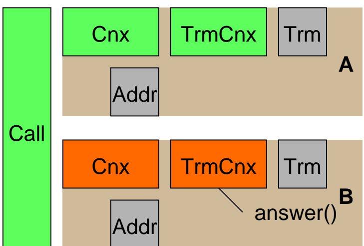




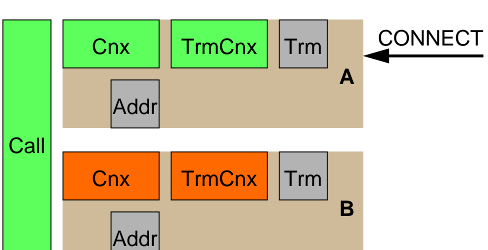
User B is alerted and the system waits until he answers the call.

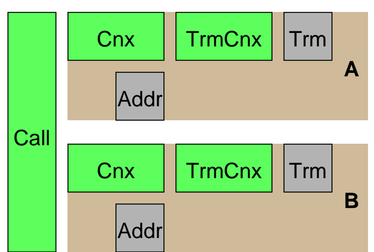
User B tells the application to answer the call.



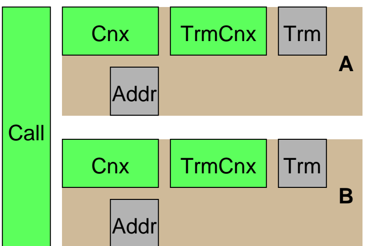


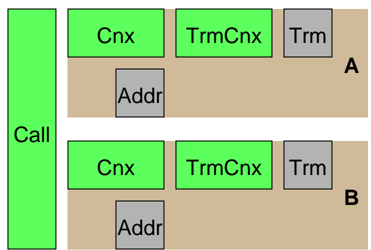
The local call leg transitions into CONNECTED/ACTIVE. The protocol stack sends a CONNECT message.





Call model at terminal A	Signalling message	Call model at terminal B
The remote call leg transitions into CONNECTED/ ACTIVE.		





The call is established.

1.2.3 Uniformity of representation vs. uniformity of control

One of the powerful principles of JTAPI is that the call model that the service provider presents to the application is independent from the location of the JTAPI interface: it can be somewhere inside the telephone system or at one of the terminals participating in the call. The example presented in the preceding section demonstrates that, neglecting the propagation of signalling messages, the call models at terminal A and B are identical. This uniformity of representation can not always be maintained, namely when signalling protocols do not carry enough information. In the PSTN for example conference calls are not visible to a terminal, even with ISDN signalling.

The uniformity of representation does not imply however a uniformity of control. The fact that the application sees uniform objects does not mean that it has the same control over them. Obviously first-party call control is not uniform. For example an application at terminal A is only able to manipulate the Connection and TerminalConnection objects that attach to terminal A's Terminal object. If an application tries to invoke a method on an object it has no control over the service provider throws an exception.

1.2.4 Several levels of detail

JTAPI makes use of inheritance to achieve a design goal summarized by the motto “simple things should be simple”. The application can choose between a basic view of entities and calls and more detailed views revealing more information about different aspects. The objects that provide the detailed view inherit from the objects of the basic view and define additional states and methods.

The basic view is provided by the *core* package, the detailed views for example by the *callcontrol* and *callcenter* packages.

1.2.5 Actors above and below the API

Most API designs are built on a master/slave relationship between the application and the service provider: The application is in control; it initiates actions and the service provider responds to these actions. This is not the case with a telephony API. It is in the nature of a telephone system that a call is manipulated by many actors: foremost the users control the call via their terminals. As we have seen an application may perform first-party or third-party call control. It is not excluded that several unrelated applications manipulate the same call. An application is just one of several actors. The service provider reports the external actions (and of course also the actions of the local application) through state changes of the call model. Regarding the control of a call JTAPI is therefore a symmetrical API, control can be exerted both from above and from below the API.

JTAPI makes use of the *Observer* [GHJV95] and *Event Object* patterns to inform the application about state changes in the call model. The application registers an object as an *Observer* with the service provider. Its purpose is to observe the call model, which is the *Subject*. Whenever the state of a finite state machine in the call model changes the service provider generates an *Event Object* that describes the state change. The service provider then calls the application’s Observer object and hands it a reference to the Event Object.

Because the appliction is not the only actor in a call a well-written application must not assume that state transitions will follow a given path; it must be always prepared to deal with unexpected state changes resulting from external actions. When designing an application for a user interface the Model/View/Controller pattern can be successfully applied: The call model corresponds to the pattern’s Model. The application has to pro-

vide the View and the Controller. The View renders the call state in a graphical representation. The Controller accepts user input and initiates actions on the call. Because the call model reports all actions, be they triggered externally or from the application, the View and Controller can be kept separate, they communicate only through the call model.

1.3 Using JTAPI for IP telephony

First-party and third-party call control in IP telephony

Just like in a traditional telephone system call control in IP telephony can be performed first-party via an interface at a terminal or third-party via an interface inside the telephone system. In a peer-to-peer architecture however the call control functions are fully implemented by the terminals such that there is no longer an 'inside'. In that case a proxy has to be introduced that intervenes in the call control protocol.

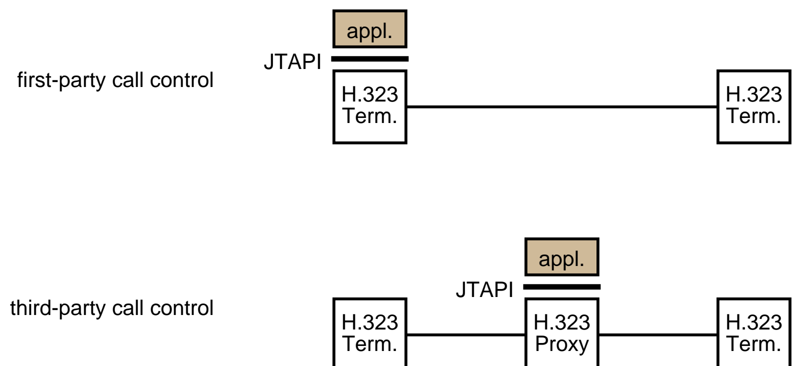


Figure 9: First-party and third-party call control in IP telephony

Addressing

Fortunately JTAPI's call model distinguishes between user identifiers and terminal addresses in the form of separate Address objects and Terminal objects. Therefore it is straightforward to map the user identifiers of an IP telephone system which have often the form of e-mail addresses to Address objects. Telephone terminals are internet hosts which have an IP address (which may also be represented by a domain name), this address maps to Terminal objects. Throughout the examples given in this document we have already used this convention.

A shortcoming of JTAPI's call model is that Address objects and Terminal objects are static. More precisely there are two kinds of Address and Terminal objects:

- There are static Addresses and Terminals that the service provider knows about already before any call exists. On a first-party call control interface these are the objects for the local terminal and the user of that terminal. These objects are said to be in the *address space of the provider*.
- Then there are those Addresses and Terminals which are not in the address space of the provider but which appear because they participate in a call. The application has no way of accessing them except through call objects, therefore the service provider is free to create them dynamically when they are needed.

JTAPI assumes that the relationship between Address objects and Terminal objects is fixed throughout the lifetime of these objects (see for example the documentation of the method Terminal.getAddresses()). In IP telephony however the relationship between user identifiers and terminal addresses is more dynamic than this.

1.3.0.1 Mapping features of an H.323 terminal to JTAPI

This section presents features of an H.323 terminal and how they are represented in JTAPI.

Multiple simultaneous calls

Because there is no physical line for a bearer connection it is easy for IP telephone terminals to support multiple simultaneous calls. An example of their representation in JTAPI through multiple call instances is shown in Figure 3.

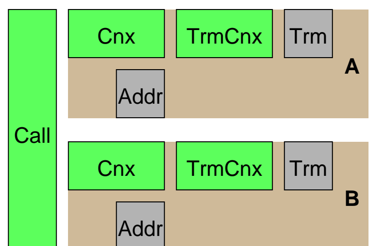
Call hold

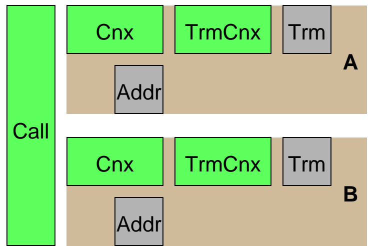
When a terminal puts a call on hold it stops processing media from the other terminals, stops sending media to the other terminals, and sends a signalling message to the other terminals that tells them to stop sending media to the terminal. An important point to note about call hold is that each party has the possibility to put a call on hold independently from all other parties. An example of a two-party call is shown below. A puts the call on hold. Independently of that B could also put the call on hold. Note how JTAPI correctly represents the party which put the call on hold.

Call hold is a feature of the call control extension package of JTAPI.

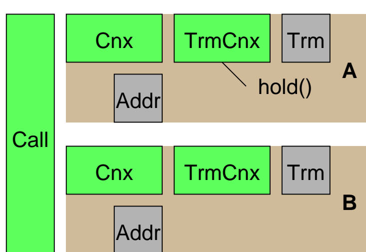
Call model at terminal A	Signalling message	Call model at terminal B

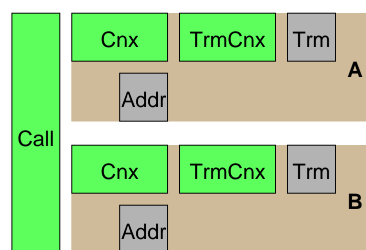
A call is established between terminals A and B.



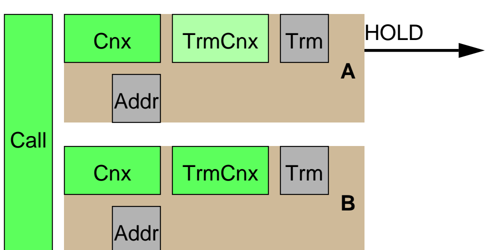


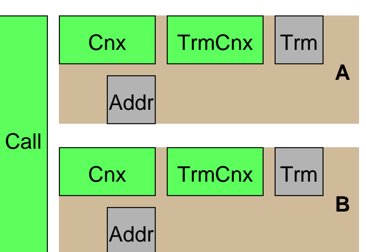
The application puts the call on hold by invoking hold() on the local Call-ControlTerminalConnection.

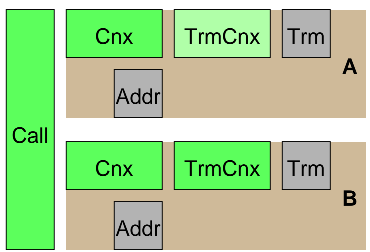
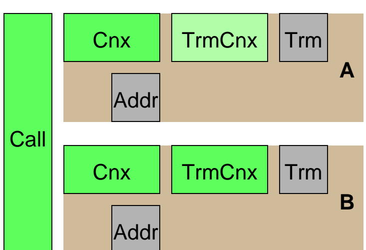




The local CallControlTerminalConnection transitions into HELD. The protocol stack sends a HOLD message.





Call model at terminal A	Signalling message	Call model at terminal B
		The remote CallControl-TerminalConnection transitions into HELD.
		

Now the call is in a state where it is put on hold by A. Note that B's ability to put the call on hold is not affected by it.

Multiple alerting terminals

Multiple alerting terminals are as easily supported by IP telephone terminals as multiple simultaneous calls. When making a call to a user who wants to be alerted via several terminals the calling terminal simply establishes several H.323 calls, one to each terminal. When one of the called terminals answers the calls to the other terminals are immediately disconnected. An example of the call model in this case is shown in Figure 4.

Multi-party conference calls

The Recommendation H.323 specifies several modes for conferences.

1.3.1 Multimedia

[Call control vs. media control. Call establishment not identical to media stream establishment.]

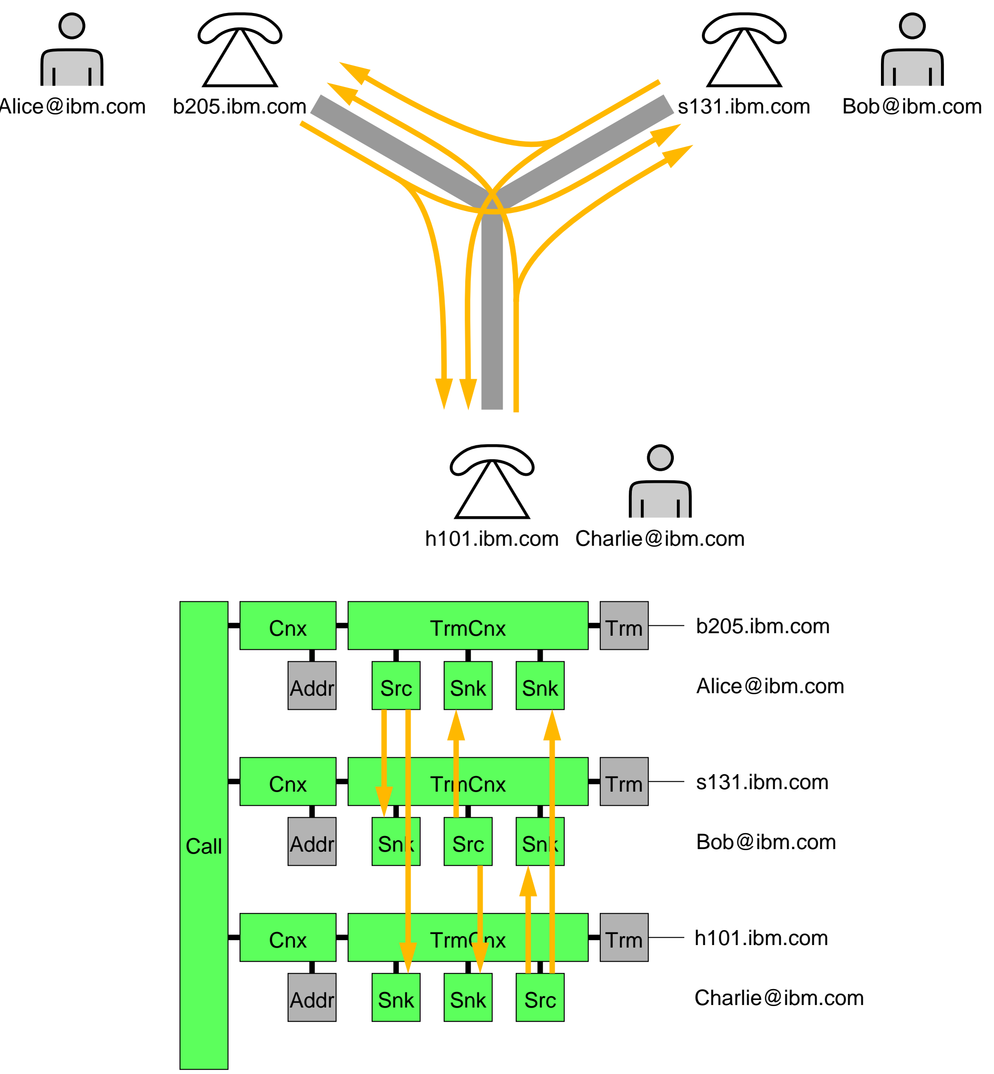


Figure 10: Call model for three-party call with media streams for one medium

MediaEndpoint

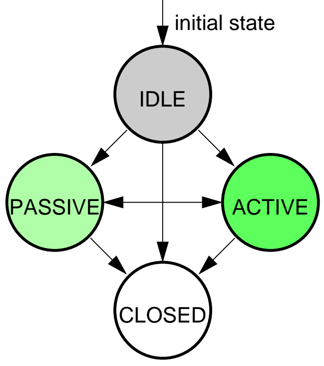


Figure 11: Finite state machine: MediaEndpoint object

IDLE	ME	This is the initial state for a newly created MediaEndpoints. There are no media resources allocated yet.
ACTIVE	ME	Media resources are allocated and media data are being transmitted.
HELD	ME	Media resources are allocated but media transmission is suspended.
CLOSED	ME	The media resources are deallocated.