ITX Programmer's Guide

The ITX Project
Computer Science Dept.
Cornell University

August 31, 1999

Contents

1 Introduction **5**

2 If You Want to Use ITX ... **7**

3 Data Exchange **11**

4 Signaling **15**
- 4.1 Registering with Directory Service 15
- 4.2 Dialing Out 17
- 4.3 Incoming Calls 18
- 4.4 Signaling and the Other Components 19
- 4.5 How ITX Signaling Works 19

5 Directory Services **21**

6 The Gateway **23**
- 6.1 Implementation Details 24

7 About Events, Exceptions, Properties, and Statistics **27**
- 7.1 Events 27
- 7.2 Exceptions 28
- 7.3 Connection Properties 28
- 7.4 Statistics 29

8 Examples **31**
- 8.1 Minimal ITX Program 31
- 8.2 ITX Program That Rings a Telephone 32
- 8.3 Registry Dumper 32
- 8.4 PhoneRecorder 33

8.5 Network Client 35

Chapter 1

Introduction

ITX is a set of Java packages which allow one to write telephony applications in Java. (Some sample applications are provided with the ITX distribution.) This Guide introduces the reader to the ITX Application Programming Interface (API), starting with an overview. Subsequent sections explain each component of the API in more detail.

Telephony applications move sound across the telephone network and the internet. The ITX system allows telephony connections to be established between any combination of computers and telephones.

When sound is being transmitted over the internet, it is generally transferred as UDP traffic. We use RTP (Real Time Protocol) to handle this traffic. RTP helps with synchronizing send and receive buffers, timestamping, and sequence numbering.

Sound on the telephone lines, however, is transmitted as sound samples (8-bit samples, 8000 samples per second, Pulse Code Modulation), generally multiplexed into a TDM (Time Division Multiplexing) flow and transferred at 64kbs.

Sound can also be stored as files, of course, for later replay. One file format that is often used is the `.wav` format.

Besides the data plane just described, telephony applications also involve a control plane. The control plane, or "signaling", is what sets up calls, tears down calls, makes telephones ring, and the like. DTMF (Dual-Toned Multiple Frequency) tones are also handled in the control plane. The Session Initiation Protocol (SIP) can optionally be used for the control plane.

To move sound between the PSTN (Public Switched Telephone Network) and the Internet, we use special adapter cards. A networked computer that contains one or more of these cards is called a "Gateway". The Gateway is not visible to ITX application programmers; an ITX program does not know or care whether it is talking to an ordinary computer running an ITX application or to a Gateway.

If you have your own PBX (Public Branch Exchange), you can acquire a JTAPI (Java Telephony API) package from your vendor. In this case, the gateway can send control requests to the PBX, and the PBX can be used to set up and tear down calls. In this way you can trade off call quality for communications cost, and take advantage of any additional functionality that the PBX provides.

Finally, you need a way to look up telephone numbers and email addresses. The ITX system includes a directory service based on BIND, but you could plug in any backend database you choose. The backend database contains an entry for each user and application registered with the local ITX implementation. Application programmers use the ITX Directory Service to look up addresses for users. These addresses can include telephone numbers, ip addresses of applications currently being used by the user, and other users. It is important for applications to register themselves with the database if they expect to handle incoming call invitations. Users register with the Directory Service, allowing them to be located by other users. The implementation of the backend database is transparent to the application programmer or user.

This overview has described the four main components of the ITX telephony software system:

- Data Exchange (implemented in the `cnrg.itx.datax` Java package)
- Signaling (implemented in the `cnrg.itx.signal` Java package)
- Gateway (implemented in the `cnrg.itx.gtwy` Java package)
- Directory Service (implemented in the `cnrg.itx.ds` Java package)

The rest of this Guide explains how you, programming in Java, can use the ITX API to create your own internet-telephony applications. Please refer to the JavaDoc for more detail; it can be found at `http://www.tc.cornell.edu/cnrg/telephony/JavaDocs`.

Chapter 2

If You Want to Use ITX ...

If you are a faculty, staff, or student at Cornell, use can use the demonstration ITX telephony system running in the System Lab in Upson 331. Get yourself registered in the ITX database by emailing Donna Bergmark at *bergmark@cs.cornell.edu*. If you are writing a computer-based telephony application that will be sending or receiving sound data from ITX demo computers, then your computer must also be located within the CS firewall.

Note: ITX installations at other sites should alter this document to include local procedures for registering users in the ITX database.

To contact ITX entities by telephone, just call up the Gateway (extension 45522 or 45524 at Cornell) and then punch in the extension number (followed by a #) of the entity you are trying to reach.

To write an application that can interact with the internet and the telephone network, you need to write a Java program. This Guide, plus our JavaDoc at `http://www.cs.cornell.edu/cnrg/telephony/JavaDocs` will tell you what you need to know. You can also download the ITX source from this location.

You can install ITX on your own system. In order to interface with "real" telephones, you will need a card that accepts a telephone line and that has drivers that can call out on these lines, get input on these lines, and the like. The Cornell demonstration system uses a Dialogic D41/EPCI card; its lines are connected to a PBX which sits in the System Lab; and this PBX is connected to the campus PBX. You can buy your own Dialogic card if you have an NT with a PCI slot.

If all you are interested in is doing computer to computer telephony using microphones and speakers, but no telephones, then just omit (don't use) the `cnrg.itx.gtwy` package.

For most telephony applications, you also need a computer equipped with a full-duplex sound card.

The ITX system is programmed in Java with JNI (and some RNI)[1] to a small amount of lower-level C code. The directory server (BIND) runs in Unix and the directory client (DSComm) can run in either NT (DSComm.dll) or in Unix (libDSComm.so). The PBX server is written in pure Java and can run on any Java platform. The Gateway manager is written in Java and JNI, but must run on an NT machine because that is where the Dialogic cards are supported. Applications that use sound cards, speakers, and microphones must run on an NT platform because at the present time, the `cnrg.itx.datax.jaudio` package uses RNI to native sound libraries. When a more mature JavaSound implementation becomes available, it would be good to switch to it.

To create an ITX applications, you need to decide what components you wish to use. If you want to support real telephones, you need to use the `cnrg.itx.gtwy`, `cnrg.itx.signal`, and `cnrg.itx.datax` packages. If in addition, you want to use a PBX, then you must also use the `cnrg.itx.gtwy.pbx` package. Use of these packages require that the directory server and gateway manager be running. The PBX server is optional; you can configure the gateway to run without a PBX.

Programs that only manipulate the directory need only use the `cnrg.itx.ds` package. You also need to have a directory server running, and you need access to the `resolv.conf` configuration file, located by default in your working directory (details later in the guide).

Directory Service is optional for computer-to-computer ITX application.

If your application uses the sound card, your PATH must include `jaudio.dll`. If your application uses directory services, your PATH must include either `DSComm.dll` or `libDSComm.so`.

Table 2.1 summarizes your choices.

[1] JNI is Java Native Interface and RNI is Microsoft's Raw Native Interface.

Table 2.1: *Depending on which components you use and which servers are running, you can build different kinds of applications.* (x) *means that the associated item is optional, but could be useful, for this application. A blank indicates that the associated item would not be used by the application. Panic Button is a toy application that runs on your desktop and makes a telephone ring.*

	Packages Used				Servers Running		
Application	datax	signal	ds	gtwy	BIND	Gateway	PBX
Full-blown ITX application	x	x	x	x	x	x	(x)
Computer-to-computer telephony	x	x	x		x		
2-way telephony using specified addresses	x	x	(x)				
Directory maintenance			x		x		
Panic Button (all signal, no data)		x	x	x	x	x	(x)
Sound transport	x						
Computer-to-computer signaling, no sound		x	(x)				

Chapter 3

Data Exchange

In telephony applications, "data" refers to sound, whether encoded as PCM samples and sent over wires, stored as PCM samples in an audio file, or traveling over the internet as UDP packets that contain the samples.

ITX's data exchange package (`cnrg.itx.datax`) is used by the programmer to record, play, and transfer sound. The basic object in this package is the AudioConnection. An AudioConnection consists of two channels, an input channel (this brings sound *to* the application) and the output channel (which carries sound *from* the application).

There is a third kind of channel, the mixer channel, which is used for conferencing applications. This channel effictively mixes a number of sources and sends the result to a number of destinations.

The channels contain devices. Each channel has a source device and zero or more destination devices. (A common setup has a channel connected to a source and one destination.) The current ITX package contains implementations for the following devices:

Destination Devices

- NetworkDestination – used to send sound to another application or gateway across the network
- SpeakerDestination – used to play sound to the speaker attached to your computer
- JSpeakerDestination – an experimental version of speaker destination that uses JavaSound.

- StreamDestination – used to send sound samples to a Java Stream (e.g. a file)

Source Devices

- MicrophoneSource – used to get sound from the microphone attached to your computer
- JMicrophoneSource – an experimental version of microphone source that uses JavaSound.
- NetworkSource – used to receive sound from the internet (usually from a corresponding NetworkDestination)
- StreamSource – used to read in sound from a Java Stream, such as a file on disk

Example: Suppose you want to set up an audio connection that uses your computer's microphone as input, and a file as output. Of course, there are already tools for generating audio files, but here's how you can use ITX to record your own:

```
Channel c = new Channel();
c.setSource ( new MicrophoneSource(c));
FileOutputStream fos = new FileOutputStream ("mumble.raw");
c.addDestination ( new StreamDestination ( fos ) );
```

This application is inherently "half-duplex": there is only one uni-directional channel with the microphone as a source and the file as a destination. A "full-duplex" application would have two channels, each with a source and destination.

To conclude the example, the application sets up a half-duplex connection as follows:

```
AudioConnection ac = new AudioConnection ( c, null );
ac.open();
```

After a certain period of time, or after the application determines that the connection is no longer needed, it closes the connection:

```
ac.close();
```

For data transfer over the internet, forward error correction is done at the network layer so that lost samples can be recovered. Buffering is also done at the NetworkSource to remove jitter due to network delays.

The package could be extended by adding more devices. Any device that implements the Source or the Destination can be used in the data transfer.

Like most ITX components, Data Exchange can be used together with or independently of the other components.

Chapter 4

Signaling

Like other ITX components, Signaling can be used with or without the others. However, this is the one component that *every* ITX *telephony* application must explicitly reference, because without it the application cannot authenticate itself to the database. If the application is not in the database, it cannot be called by another entity, nor could it dial another entity. In short, it would not be a telephony application.

All ITX applications must be a "SignalingObserver", either by extending AbstractSignalingObserver or by implementing SignalingObserver (both of these are in `cnrg.itx.signal`).

In addition, applications that choose to do non-blocking Dial()s must also implement `cnrg.itx.signal.SignalConnectionObserver`.

The DesktopSignaling object is an application's primary control plane interface. This object is used to register the user with Directory Service, to dial other applications, and to tell the application if it has an incoming call. We will discuss each in turn.

4.1 Registering with Directory Service

To obtain a signaling component, you instantiate a DesktopSignaling object. At minimum, you must provide a SignalingObserver, which is typically just a handle to the application itself. The functions in the SignalingObserver are called by DesktopSignaling to notify the application of events, such as call invitations and hangups.

Instantiating a DesktopSignaling with no further parameters causes the

DesktopSignaling to simply pick a port on which to listen for incoming messages. The application can find out what port this is by calling getPort() on its DesktopSignaling.

Usually, however, an application will also choose to register with Directory Service so that other applications will know how to find it, and so that it can find and dial other users. In this case, the selected port will be entered into the directory. To get a usable directory object, the application must provide a legal user id and password when instantiating its DesktopSignaling:

```
import cnrg.itx.signal.*;
public class myFirstITXApp extends AbstractSignalingObserver {
       :
   DesktopSignaling signal = new DesktopSignaling (this,<id>,<pass>)
       :
}
```

Passing "this" as a handle tells our DesktopSignaling who its SignalingObserver is. The next two arguments are Strings which are authenticated against the database of currently registered ITX users. In addition, the current location (IP address and port number) of this application is automatically registered in the database as `<id>`'s current roaming location. For testing, we recommend that your directory contain a user named "guest" with password "guest".

In the present version of ITX, DesktopSignaling locates the Directory Server (the only centralized server in the ITX system) by reading a file called `resolv.conf`, by default from the current working directory. The contents of this file says where the directory server is running, e.g.

```
nameserver 128.84.223.58
nameserver 128.84.223.59

```

At Cornell, these are the ip addresses of the primary and secondary BIND servers, i.e. `sim1.cs.cornell.edu` and `sim2.cs.cornell.edu`.

If your `resolv.conf` file is in a different directory, you can use a variant of the DesktopSignaling constructor to say where it is:

```
// tell signaling where the directory server's config file is
DesktopSignaling signal = new DesktopSignaling
   ( this, "id", "pass", "my custom telephony application ",
    "c:\\net\\reconfig.ini");
```

Note that this variant also lets you give a name to your application, which is entered as a descriptive tag in the directory's database.

Since the DesktopSignaling registers your current location in the database, you should clean up when finishing by invoking DesktopSignaling's logout() method.

4.2 Dialing Out

Suppose your application wants to call somebody on a real telephone, or maybe it wants to contact another application (Voice Mail, for example), or just wants to call a person, contacting them wherever they may be located. This is done by your desktop signaling's Dial() method. This method takes a String object that says who or what you want to call. If it is a telephone, you might put in a string like `5551212`, which would dial a telephone number.

If your application wishes to contact a user in the system, then Dial their userid, which is also their email address.

The Dial method sends a call invitation to the other party; if the invitation is accepted, Dial sets up a SignalConnection (which is an AudioConnection as described in the previous section with control information added to it). If the invitation is refused, an appropriate Exception is thrown.

By default, a full-duplex connection is set up. One channel goes from the microphone on your computer to the internet address of the entity you called (if your application is dialing a real telephone, the internet address is that of a gateway machine that has the right equipment to place the call). The other channel comes in from the network (from whatever or whomever you called) and goes to the speakers on your computer.

If you don't like this setup for your input and output channels, then you can override them in the dial call.

Here are two examples ("signal" is the DesktopSignaling object):

```
// Example 1: default is to use microphone and speakers
SignalConnection flow1 = signal.Dial ( "user@itx.org" );
    :
// Example 2: over-ride the defaults for a half-duplex connection
Channel cOut = new Channel();
cOut.setSource(new Microphone(cOut));
cOut.addDestination(new NetworkDestination())
SignalConnection flow2 = signal.Dial ("5551212",
```

```
    null, cOut);
```

The first example will send a dial invitation to user@itx.org trying each location in the list of places where she is currently callable, until the call is picked up or the list is exhausted. Her current locations are listed in the ITX Directory, and might include a roaming location.

The second example dials a telephone number and will send sound to the telephone but will not receive data because the input channel is null.

If the application decides that it wants to hangup a call, it can call its DesktopSignaling's Hangup method:

```
// signal is our DesktopSignaling
// con is the SignalConnection we were using
signal.Hangup ( con );
```

4.3 Incoming Calls

Taking incoming calls starts with the application's SignalingObserver handler methods. These methods include:

- `onInvite(InviteSignalEvent)` - a peer application is asking you to answer ("inviting", in telephony jargon). Your application can accept, reject, or say you are busy.

- `onStartCall(SignalConnection sc)` - a peer application has confirmed your accept, and call setup is complete. The application should use the Connection inside of the SignalConnection to do data exchange, after opening it.

- `onAbortCall(AbortSignalEvent)` - the peer application has aborted the invitation made to you (perhaps after learning that you are running on a computer with no speakers; see the section on Connection Properties). The event object contains the reason for the abort.

There are a few other SignalingObserver methods that your application might wish to implement. These are

- `onHangup(HangupSignalEvent)` - the peer application has hung up. This would normally come during an established call.

- `onDTMF(DTMFSignalEvent)` - the peer application has sent you a DTMF tone. Details are in the event object.

4.4 Signaling and the Other Components

There are certain variations available to the ITX applications programmer concerning the interaction between your control layer and the directory and the data layer. At minimum, the application need never directly invoke directory or data exchange functions, because your DesktopSignaling can do this for you. For example, the application's DesktopSignaling takes responsibility for setting up the data connection between two peer applications if the Dial() invocation doesn't say otherwise.

Once a Dial() has been performed, the application's DesktopSignaling communicates with the DesktopSignaling instance in the peer application. They notify each other that they are still present by sending "keep-alive" messages.

4.5 How ITX Signaling Works

ITX signaling protocol is based on the SIP call initiation protocol. Users are addressed by email-like addresses. Applications have names that do not include the @-part.

The first step in a 3-way session setup is when the caller sends an INVITE signal to the callee. The invitation contains enough information to describe the caller and what media types and formats will be accepted in the call.

The callee then decides whether or not to ACCEPT. If so, then the packet that goes back to the caller contains the callee's part of a data connection (which media properties are supported, etc.). At that point, both peers know all the properties of the data channel. Finally, the caller completes the packet with its properties and sends a CONFIRM back to the callee.

Once the 3-way handshake is complete, both sides open their data connection and the call proceeds. If at any time, the 3-way handshake times out, the call is torn down, and the application is informed through the onAbortCall() event handler. Likewise, if the callee sends an ACCEPT packet back to the caller but never gets an ACK, it times out and tears down the half-way built connections.

Chapter 5

Directory Services

A directory service client is usually encapsulated within a DesktopSignaling object, and is therefore bound with the ITX application. This client communicates with the directory server, to look up names and locations, to authenticate this application or application user with the server, to add and delete entries, and so on. The actual directory server used by the ITX project is BIND; by modifying the DSComm class, one can interface with other databases.

The directory service client is part of the DesktopSignaling object. An application can get its DirectoryService object from its DesktopSignaling as follows:

```
DesktopSignaling signal;  // our DesktopSignaling
DirectoryService ds;      // our directory client
     :
ds = signal.getDirectory();
```

Now the application can use the directory through any DirectoryService public methods. It should be pointed out that there are levels of privilege associated with the directory. Applications and users have the lowest level of access, and can manipulate only those database entries associated with their username. The access level for a user or application is determined when the entry is first added to the database by the database administrator.

One simple ITX application might be a program that allows people to update the database to specify new telephone numbers where she or he can be reached. This sample application would register on the user's behalf (collecting the userid and password via a dialogue box, say), and then call addCustomLocation and deleteCustomLocation as the user wishes.

One convenient method is "dumpAllUsers" which lists all the users currently registered with ITX. The list is returned as a vector of UserProperty objects.

Given a userid, you can look up their ITX extension number using the "getExtension" method:

```
// ds is my DirectoryService.  How can I contact bob@cornell.edu?
Digits extension = ds.getExtension(new UserID("bob@cornell.edu"));
```

(Digits is an object in the `cnrg.itx.ds` package.)

The application might want to look up its own information, registered in the database by your DesktopSignaling component when you constructed it. This information includes your extension or userid or whatever. In fact, if your application is invoking *any* method that requires a UserID object, one way of coming up with one is as follows:

```
// signal is my DesktopSignaling - who did I register as?
UserID me = signal.getDirectory().getID();
```

See the DirectoryService JavaDoc page for other methods you can use. Some objects in the `cnrg.itx.ds` package that programmers should be familiar with are UserID, Location, Digits, and UserProperty.

A Location contains the information about a participant in the ITX system. For example, `SignalEvent.InvitePacket.getSenderLoc()` returns the Location of the person who started a call. From the Location you can get its type (whether it is a telephone, voice mail, etc.), the UserID containing the name of the user or application making the call, whether or not that Location can be directly dialed, an so on.

The UserID object encapsulates the name of a user. `Location.getUID()` returns the caller's name; `DesktopSignaling.getDirectory().getID()` returns the name that this application was logged in with.

Digits is the object that represents extensions, and was shown in one of the examples above. UserProperty contains information about a single entry in the database.

Chapter 6

The Gateway

ITX telephony applications that wish to include real telephones located in the telephone network use the `cnrg.itx.gtwy` package to relay sound data between the internet and the telephone network.

The way the gateway package is integrated with ITX is as follows: when launched, the Gateway Manager runs as a peer application with its own signaling, data, and directory clients, just like any other ITX application. The difference is that when it gets an invitation from another application, that invitation is asking for a real telephone number. The Gateway's implementation of the Dial method is to place a call from one of its telephone lines to some other telephone.

For example, suppose my application wants to call a telephone number. When Directory Service is asked to look up that number, it realizes it is a telephone because (1) it is not in the database, and (2) it is all digits. In this case, it returns to my application the list of all the locations (IP and port) of running gateways.

Similarly, using a real telephone, one might dial the gateway, and then punch in the extension of an application or user on the network that one wants to call. The gateway will then set up a datapath between the line you are calling on and a socket that leads to the indicated application, which could be some user's desktop telephone application.

Gateway code (unlike control, data, and directory) is not bound into other ITX applications; it runs as a server and as a peer application. The name of the server is Gateway. Only one Gateway per Dialogic card is permitted, because a set of telephone lines can only be controlled by a single manager.

A gateway may or may not be associated with a PBX. If your group has control over a PBX, or part of a PBX (as we do in the ITX project), then you can have the gateway use the PBX to place and receive calls. In our case, the gateway passes all invitations to a telephone on to the PBX Server (PBXSignalingServer), which then dials the telephone and reports back whether the telephone line was picked up or not. In addition, it hangs up the call when asked to do so, and reports back to the gateway if the remote telephone hangs up.

The gateway also allows you to use a real telephone to send sound to an application running on another computer. The steps involved are:

1. Pick up the telephone and call a number on some gateway (in Cornell's ITX demo, one calls 4-5522 or 4-5524 to get a line connected to `vada.cs.cornell.edu`).

2. The gateway answers the call, playing a recorded message to the caller. In the case of the CUTel demo, the message asks you to punch in an extension number, follwed by a pound sign.

3. The gateway then makes a Dial to the user at that extension. As a result, the gateway's DesktopSignaling component sends the invite on to the current Location of that user (presumably an application running on a computer somewhere).

4. When the user or application accepts, the gateway opens up a data channel from the oncoming call to the network, and the call commences.

All sound data between the Gateway and another application passes over the network, in UDP packets, using RTP. Accordingly any data sources or destinations that involve a telephone will use a NetworkSource or NetworkDestination device.

6.1 Implementation Details

None of the information in this section is needed by the ITX programmer, but it may be of interest anyway.

The gateway server, Gateway, is coded in Java with JNI calls to the low-level Dialogic system library, written in C. The Dialogic library handles such things as initializing the telephone lines and playing sound bytes to the telephone.

The Gateway instantiates a SignalInterface object, which in turn instantiates a standard ITX DesktopSignaling object (which in turn registers the gateway with directory service). The SignalInterface is the part of the gateway that implements the SignalingObserver interface.

The SignalInterface object also optionally communicates with a PBX server. The PBX runs entirely under the control of a gateway and does not directly communicate with any other application.

The PBX server is coded entirely in Java, handles no data, and uses JTAPI (Java Telephony API) to control the PBX. Using JTAPI, the PBX can set up outgoing telephone calls to anywhere. (Incoming telephone calls go through the PBX switch directly to one of the lines on the gateway, without being handled by the PBX server.) The JTAPI code in ITX could be extended greatly to exercise other features of the PBX.

Users of the `cnrg.itx` package will have to implement their own Gateway if they use a different telephony card. However, the DesktopSignaling component, programmed entirely in Java, should work as is. The PBX server should also work as is, because your PBX vendor should be able to supply you with a JTAPI implementation for their machine.

The PBX server is optional; when you start up Gateway, you can say whether or not you want to use the PBX server to place outgoing calls. The Dialogic card can be used directly to place outgoing calls.

Chapter 7

About Events, Exceptions, Properties, and Statistics

ITX uses the Java event model. Events are passed to handlers, so that the handlers can take appropriate action. Exceptions are thrown in case of unexpected events. Properties are roughly media capabilities, and are part of every connection. Statistics can be generated for all data transfers.

7.1 Events

Each ITX application is a SignalingObserver. Hence, events are used to convey information to the application's event handlers. For example, your onInvite handler is passed an InviteSignalEvent, which contains all the information about who is asking for you to pick up a call.

Most event objects have methods defined on them for extracting bits of information. For example, the AbortSignalEvent has the following methods:

- `getAbortReason()` - returns the String containing explaining why the proffered invite was aborted.

- `getConnection()` - returns the data Connection that had been setup between you and the party that is aborting the invite.

If your application extends the AbstractSignalingObserver, all of the default event handlers might be acceptable. (None of these handlers do anything at all.) In this case, you do not need to put any of the handlers into your application. But if you want to do something special on onInvite, etc., then you must override the corresponding handler.

All of the SignalEvents (AbortSignalEvent, DTMFSignalEvent, HangupSignalEvent, and InviteSignalEvent) contain the InvitePacket that was used to initiate the call. For example, `HangupSignalEvent.getInvitePacket()` can be used to retrieve the InvitePacket, and all of the information inside. There are *many* public methods that can be called on an InvitePacket.

7.2 Exceptions

If unexpected things happen during the course of ITX execution, then Exception objects are created and thrown. There are a wide variety of specific exceptions, all of which inherit from Java's Exception class. You may choose to field individual exceptions, or just Exception in general. In any case, the `toString()` method is defined on the Exception which gives more information about why the ITX Exception was thrown.

7.3 Connection Properties

While the ITX package provides a number of default Channels that can be used in an AudioConnection, different computers and different applications may have differing sound capabilities, modulation preferences, and codecs. For this reason, ITX has the notion of a PropertiesCollection. This object is the collection of Properties supported by a device. There are other methods for getting and setting Properties on a device.

When negotiating a call setup, a final PropertiesCollection is agreed up by the two parties. The initial contact includes a collection containing the caller's available properties on each of the endpoints of its channel. When the callee decides to accept, it sets up its devices and merges sets their properties into the appropriate channel.

Once a call has been set up, each party to the call has a SignalConnection object, which contains basically all the information relating to a call. In particular you can use the `getProperties()` method to extract the PropertiesCollection from the SignalConnection. The Properties Collection is the set of device properties available to a call.

The NetworkProperty allows you to get the IP address and port of a network device's data port. Properties also include features of the encoded data stream, such as packet size and encoding method.

7.4 Statistics

Does your application want to know how much sound data is flowing, or ineed whether any sound is flowing at all? The Stats object (in the `cnrg.itx.datax` package) does the trick.

Sources, Destinations, Channels, and Connections all implement the Statistics interface, meaning that they keep a count of bytes transmitted or whatever. The `Statistics.getStatistics()` method returns a printable Stats object, for example:

```
System.out.println ( myNetworkSource.getStatistics() );
```

With some effort, an application could also parse the object to extract, say, the number of bytes that have been sent. An application could also use the Statistics interface to generate its own statistics.

Chapter 8

Examples

In the examples below, it is assumed that your CLASSPATH has the `itx.jar` archive somewhere in it, that you have a `resolv.conf` file in your working directory, that your PATH includes the directories containing `DSComm.dll` and `jaudio.dll`, and that the Gateway and Directory Server are up and running. If the Gateway is using the PBX Server, then the PBX Server is assumed to be running as well.

8.1 Minimal ITX Program

The minimal ITX application makes and takes no calls, sends and receives no data, but the location of its signaling component is registered in the directory for the duration of the run:

```
import cnrg.itx.signal.*;
public class Minimal extends AbstractSignalingObserver {
   public static void main (String[] args) {
     Minimal me = new Minimal();
     try {
       DesktopSignaling sig = new DesktopSignaling(me,"guest","guest");
       sig.logout();
     } catch (Exception e) {
       System.out.println("Couldn't get authenticated");
     }
   }
}
```

In a real application you would store "sig" as a local variable, and have a logout method that invokes `sig.logout()`.

8.2 ITX Program That Rings a Telephone

The next most minimal ITX program runs entirely in the control plane, and does not send or receive any sound data. However, it does use signaling to register with the database, ring a telephone, and unregister with the database. Edit the preceding example to add the following code between constructing a DesktopSignaling and logging out:

```
:
// Make 2545454 ring
try {
   SignalConnection con = sig.Dial ("2545454",(Channel)null,null);
} catch ( ConnectException e ) {System.out.println("No can dial");}
:
```

Here we over-ride the default speaker and microphone data channels by specifying null input and output Channels. We ask our control layer to place a call to "2545454". The directory client will return a list of one or more locations at which a gateway is running. Our control layer sends an invite to the first one; as soon as the peer accepts, our control sends back a confirm packet and returns a SignalConnection to the application. At this point, the application can choose to open the connection (`con.open()`) and data will begain flowing. Alternatively, if the gateway is down or has no free lines, it will reject our invitation, and our signaling component will throw an exception, which we catch and process accordingly.

8.3 Registry Dumper

This application lists the contents of the current ITX database. It also illustrates Directory Service being used independently of any other ITX components.

```
import cnrg.itx.ds.*;
public class StandaloneDump {
  public static void main (String[] args) {
    try {
```

```
      DirectoryService dir = new DirectoryService ();
      dir.declareIdentity (new UserID("guest"),new Password("guest"));
      Vector x = dir.dumpAllUsers();
      for (int i=0;i<x.size();i++){
         UserProperty p = (UserProperty)x.elementAt(i);
         System.out.println(Integer.toString(p.getExtension())
         + "\t" + p.getUserID());
      }
    } catch (Exception e) {
      System.out.println("Couldn't dump because " + e);
    }
  }
}
```

8.4 PhoneRecorder

If you don't have a microphone on your computer, you can still use your telephone to record voice data. The following application will write incoming sound data into an output file:

```
import java.io.*; // for OutputStream
import cnrg.itx.signal.*;  // for DesktopSignaling
import cnrg.itx.signal.SignalEvent.*;
import cnrg.itx.datax.devices.*;  // for AudioConnection, etc.
import cnrg.itx.datax.*;          // for Channel

public class PhoneRecorder extends AbstractSignalingObserver {

   private DesktopSignaling myDS = null;
   private FileOutputStream ourFile = null;
   private NetworkSource source = null;
   private StreamDestination dest = null;

   // Default constructor gets a DesktopSignaling object
   public PhoneRecorder () {
      [ myDS = new DesktopSignaling ( this, "PhoneRecorder", ... ) ]
   }

   // Main entry point is here
```

```
public static void main ( String[] args ) {
   PhoneRecorder me = new PhoneRecorder();
}

// SignalingObserver method - invoked when someone calls this app
public void onInvite (InviteSignalEvent ise) {
   Connection c = null;            // our AudioConnection
   Channel in = new Channel();     // This will be telephone to file

   // Open a FileOutputStream, which is the file to which sound is written
   try { ourFile = new FileOutputStream ( getName() );
   } catch ( Exception e ) {
      System.out.println ("Could not open an output file because" + e);
      System.exit( 0 );
   }

   // Set up a half-duplex AudioConnection and open it
   try {
      source = new NetworkSource(in, SpeakerDestination.SAMPLE_SIZE);
      dest = new StreamDestination(ourFile);
      // our in channel is NetworkSource -> StreamDestination
      in.setSource ( source );    in.addDestination ( dest );
      // our connection has only an in channel (half-duplex)
      c = new AudioConnection ( in, null );     c.open();
   } catch ( Exception e ) {
      System.out.println ("Could not open data connection because" + e);
      System.exit( 0 );
   }

   // If we get to here, we have an open data connection
   ise.accept( c );  // set the accept flag in the invitation

   // Exit back to DesktopSignaling, which will return the
   // Accepted invitation back to the Gateway

}

// This method is invoked by DesktopSignaling when the caller hangs up.
public void onHangup(HangupSignalEvent hse) {
   try {
```

```
         ourFile.flush();
         ourFile.close();
      } catch ( Exception e ) {}
      System.out.println ( source.getStatistics() );
      System.out.println ( dest.getStatistics() );
      myDS.logout();
   }

   // +++++++++++  Private methods ++++++++++++++++=

   // Generate a file name to capture the data
   private String getName () {
      return "file1";   // Substitute your own here
   }
}
```

8.5 Network Client

The previous example illustrated a simple half-duplex application. A full duplex connection between two applications is illustrated by the following example, which is the client side. The server side is left as an exercise to the reader. This example also illustrates that it is not necessary to use ITX signaling to construct a data connection.

```
package cnrg.itx.datax.devices;

import cnrg.itx.datax.*;
import java.io.*;
import java.net.*;

/**
 * This class tests audio streaming over the network.  This sample routine
 * uses a NetworkSource
 * and SpeakerDestination to play audio originating
 * from a NetworkServer.  Note that in creating the various destinations
 * and sources, the remote source/destination sample size MUST equal the
 * local source/destination sample size.
 */
public class NetworkClient
```

```
{
  public static int PORT = 7777;

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

      // Input channel, source, and destination
      Channel inChannel = new Channel();
      NetworkSource inSource = new NetworkSource(inChannel,
                 PORT,
                    SpeakerDestination.SAMPLE_SIZE);
      SpeakerDestination inDest = new SpeakerDestination();

      // Set input channel
      inChannel.setSource(inSource);
      inChannel.addDestination(inDest);

      // Start destination
      inChannel.open();

      // Let the audio file play...
      System.out.println("Playing audio from the network...");
    }
    catch (Exception e)
    {
      System.out.println(e.getMessage());
    }
  }
}
```