Building Blocks for IP Telephony

D. Bergmark and S. Keshav*

Cornell University, Dept. of Computer Science

February 4, 2000

To appear in IEEE Communications Magazine, April 2000

1 Introduction

The global telephony infrastructure is rapidly evolving from circuit-switching to IP-based packet-switching. Voice over IP (VoIP) is becoming common and is the intended base technology for corporate giants such as AT&T and Qwest. The marketing research firm, IDC, states "IP telephony is poised to become the fastest-growing, network services market of the decade" [1]. Reference [2] reviews the motivations behind the move toward Internet telephony.

The internet already provides sufficient service quality for one-way voice transmission (such as for recording voice messages, listening to messages, etc.). For these purposes, VoIP surpasses the audio quality of the PSTN. It is likely that in the near future the Internet will also provide sufficient service quality for two-way voice transmission. Nevertheless, there is a large investment in the PSTN infrastructure which will not go away overnight. However, this infrastructure, with its complex interfaces, is increasingly difficult to manage and control. In order to smoothly transition from the current PSTN infrastructure, while simultaneously leveraging it to create new services, we believe that there is a need to rapidly prototype *multimodal applications*, that is, applications that span the telephone network and the Internet [3].

At Cornell, the ITX project has created an open-source, portable, programmable platform for developing multimodal applications. The ITX platform allows these applications to span the telephone and IP networks, or run only on one of these networks. Our main design philosophy was to create a small set of core components that could be composed to create non-trivial multimodal applications. We have not only used this to create some example applications, but have also made the code widely available, in source form, for others to use (see Appendix A for details). In this paper, we describe the design of the ITX platform, present a simple application built using this platform, and present preliminary performance measurements.

*This work was supported by NSF grant ANI-9615811. The web page for this project is at `http://www.cs.cornell.edu/cnrg/telephony`.

2 Related Work

In this section, we briefly review technologies exploited in the ITX system and related telephony projects.

2.1 VoIP

How to transmit voice over a packet network was developed decades ago. Influenced by the existing telephone system, voice is typically sampled in 8 bits per sample, 8000 samples per second, and transmitted in Pulse Code Modulation (PCM) format. Modern PC's routinely come equipped with speakers, microphone, sound card, and a sound library. All of this VoIP technology is readily exploitable by the ITX software system.

2.2 SIP/H323

Session Initiation Protocol (SIP) is rapidly gaining support as the signaling protocol for internet telephony. It specifies user location (i.e. directory lookup), handshake to set up a session, and how the invitee responds to an INVITE [4, 5]. ITX's native signaling is a variant of SIP, and a program switch can be set to do SIP protocol exactly, though this has not been tested with an actual SIP server. H.323 is an older session standard, which is in common use today but, is very complex. ITX does not, therefore, currently implement H.323.

2.3 RTP

Realtime Transport Protocol [6] allows for transport of sound data, usually over UDP. RTP headers describe codec and sample properties. There are also timestamps to assist in synchronization. ITX's data layer uses RTP, so it is interoperable with other RTP applications.

2.4 TOPS

Telephony over packet networks (TOPS) [7] is a component-based system with some similarities to ITX. Like ITX, it keeps in a directory the list of locations at which a user could be reached, along with preference profiles. Like ITX, each location in the list can be tried in turn until the callee picks up. One basic difference is that TOPS uses its directory to store terminal capabilities, whereas in ITX, each end of the data channel automatically negotiates the set of properties to be used for the call. Another difference is that in TOPS, the signaling protocol is exposed to the application; the application decides which packets to send and when. In ITX, application logic is separated from signaling logic. The interface between the application and signaling is an API to initiate activities (e.g. `Dial()` and `Hangup()`) and event handlers (e.g. `onHangup()`) that can optionally be over-ridden to provide custom handling of asynchronous events.

2.5 Current Work in Telephony APIs

Most existing Internet telephony applications are monolithic and therefore do not allow application builders to reuse components to build new systems. For example, existing applications such as Microsoft NetShow integrate a signaling protocol into the application, and it is not possible for other application writers to reuse this signaling library for other Internet telephony applications. In contrast, our approach is to provide building blocks with clean and well-specified interfaces. This allows other application developers to build on our work.

The *ChaiTime* system at Bellcore [8] is similar to the ITX project in that it is an architecture for generating new telephony applications. That project is motivated by many of the same reasons as our project, and like ITX uses JTAPI and SIP. However, it is more PBX-centric than ITX and is focussed on providing a platform in which third parties can plug in additional services, rather than develop applications.

Work at the Swiss Federal Institute of Technology by Gbaguidi *et al.* [9] is similar in many aspects to ours, though independently conceived. Like us, their aim is to put together a platform on which hybrid applications can be implemented. Where we put together a collection of Java packages which can be used directly to implement telephony applications in Java, they produced a collection of Java Beans which can be selected and combined to produce an application. Both approaches allow a "mix-and-match" strategy. However, we preferred to produce example applications, based on a library of Java packages, that could be modified to suit (this is the *template* approach) whereas they preferred to provide small pieces of application code that could be hooked together.

3 ITX Architecture

Our goal in designing the ITX architecture was to come up with a small set of extensible building blocks for implementing telephony applications. We wanted a small, but complete, set of components (the challenge was the tension between "small" and "complete"). In terms of completeness, the system includes data, control, gateway, and directory service components.

Another design goal was to keep the interface between components small and well defined. This allows us to make the components largely independent of each other and individually replaceable. We also wanted to reuse existing technology *and* protocols. We now describe the results of our design.

The entire ITX package is divided into four main components, distributed as Java packages:

- Signaling
- Data Exchange
- Directory Service
- Gateway

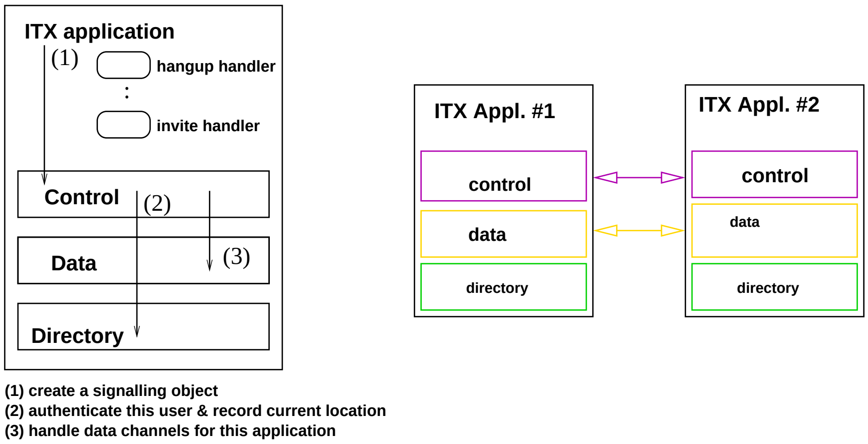


Figure 1: *The figure on the left shows a single ITX application bound with the ITX components. A few of the interfaces are shown here: handlers in the application, which are invoked by the control layer as needed; instantiation of a control object, which in turn instantiates directory service and data paths. The figure on the right shows two peer ITX applications, with communications between them. The control planes signal each other using a SIP-like TCP/IP protocol, and the data planes exchange data using RTP and UDP.*

We describe each of these components below.

3.1 Application-specific Components

Figure 1 illustrates the architecture of an ITX application bound with three layers of ITX core components. At minimum, the ITX application interacts only with the control layer. During a call, ITX applications communicate with each other using the control and data layers. The control plane uses the reliable TCP/IP protocol, while the data plane uses the more efficient RTP and UDP protocols. The directory layer communicates with the database server. Each layer is described below in more detail.

3.1.1 The Control Plane

The control plane is used to set up calls, tear down calls, make telephones ring, and the like. This plane is handled by the Signaling package.

This is the one component that *every* ITX telephony application must explicitly reference, because without it the application cannot authenticate itself to the directory. If the application is not in the directory, it cannot be called by another entity, nor can it locate other ITX applications. In particular, all ITX applications must handle the “SignalingObserver” class, either by extending *AbstractSignalingObserver* or by implementing *SignalingObserver*, which specifies various handlers for events such as a hangup or an incoming invitation (the default is to do nothing).

A key class in the signaling package is *DesktopSignaling*. Typical ITX applications instantiate a DesktopSignaling object, which then communicates directly with the DesktopSignaling object in a peer application. The two peers use a 3-way handshake to set up a call; for example a typical call setup involves INVITE, ACCEPT, and CONFIRM packets. As described in the Reference [4], each *DesktopSignaling* acts both as a User Agent Client (which initiates calls) and a User Agent Server (which answers a call).

One novelty of the signaling layer is that because it is bound with the ITX application, it can send "keep-alive" signals to its peer. If an application should unexpectedly die, its *DesktopSignaling* goes away, too, and its peer learns that the connection has been broken. This fate-sharing between the signaling component and the application makes failure recovery much easier than is normally the case. Usually, when the control function runs as a standalone daemon, it must somehow track the state of the associated application, and notify its peer in case the application terminates abnormally. Our design eliminates this complex process, replacing it with a straightforward HELLO protocol between signaling peers. Because signaling is written entirely in Java, and because Java is a threaded language, the pinging thread can run separately from the thread that waits for incoming calls. Thus, we have exploited the thread facilities in Java to automatically provide fate-sharing.

3.1.2 The Data Plane

While it is possible to write an application that runs only in the control plane, most telephony applications also need to handle voice data, and hence need a data plane. The data plane is handled by the Data Exchange package.

The Data Exchange package includes methods for setting up data connections between two peer ITX applications. This data connection, more specifically an audio connection, has two channels: one that brings data *to* the application, and one that carries sound data *from* the application. By "wiring" the output channel of one application to the input channel of another, and vice verse, a network-based, full-duplex audio stream can be implemented. Each channel, whether input or output, has a source device and a destination device. Among the data devices supported by ITX are microphones, speakers, network sources and destinations, and stream sources and destination (e.g. files).

Voice data is transmitted over channels in RTP packets containing a 64-byte payload of PCM samples. The packets are buffered at both ends of the channel. In order to reduce jitter due to dropped packets, Forward Error Correction (FEC) is used. In ITX, this is implemented by sending packets with a 192-byte payload, representing the current sample and the two previous samples. Thus, even if two out of every three consecutive packets are lost, the receiver's voice playout is unaffected. Of course, this comes at the cost of tripling the bandwidth, and perhaps causing the very packet loss that we seek to recover from. In a future implementation, we help to carry a compressed version of the previous samples to reduce packet size, as is done in the IVP tool from INRIA.

In order to minimize dropout due to buffer underflow, we implemented the adaptive playout scheme described in [10]. The Data Exchange component keeps track of the RTP timestamps to estimate the current end-to-end delay jitter. It automatically moves the playout point to match this jitter. (The effectiveness of this technique is highly dependent on the quality of the endsystems' clock.)

3.1.3 The Directory

The directory layer is handled by the Directory Service package. It provides a central repository for quasi-static information, such as user information, and dynamic information, such as the IP address at which a user is currently located. It allows an ITX application to dynamically locate other applications, and it allows a PSTN user to connect transparently with an Internet user. The key class in this package is *DirectoryService*. One of these objects is instantiated for every ITX application by the application's *DesktopSignaling*. Authentication happens automatically when the *DesktopSignaling* is constructed, and lookups happen automatically when the application invokes `Dial()`.

The Directory Service package also contains an object (a `.dll`) that serves as a local client to the backend database server. The ITX database contains an entry for each user and application registered with a site's ITX installation. Each user has an extension number, which if called, will connect with the user at some current location (such as a real telephone, another ITX application, or email). The only ITX application whose address needs to be known *a priori* is that of the database server. All other servers and users can be looked up in the database.

The current "roaming" location of a user is updated when the user logs into an Internet telephone. At this time, the current IP address and port number on which the Internet telephone's *DesktopSignaling* is listening is entered into the database as that user's current location. Users can now call each other by name (on the Internet) or by typing extension numbers into the ITX system, followed by a # (on the Internet or the PSTN).

The directory also contains user profiles that specify, according to time period, where they prefer to be reached. This feature was inspired by a similar feature in the TOPS system [7]. Finally, an access level is associated with each user or application. The *adm* account, for example, has the authority to add and delete users, but regular users can update only their own records. An ITX application that manages the database must authenticate itself by specifying *adm*'s password.

3.2 The Gateway

The fourth core component of ITX is the Gateway. The Gateway package consists of a Gateway Manager and a PBX Manager, both of which run as servers.

The Gateway Manager is a special-purpose ITX application. Its purpose is to move voice data between the PSTN and the Internet. The PBX Manager is a special-purpose program that interfaces between the Gateway Manager and a PBX.

In general, ITX applications do not know whether they are talking to a Gateway or to another ITX application. When launched, the Gateway Manager runs as a peer application with its own signaling, data, and directory layers, just like any other ITX application. The difference is that when it gets an invitation from another application, it is for PSTN telephone number. The Gateway Manager handles an incoming invitation by placing a call from one of its telephone lines to the requested telephone number.

Likewise, if a PSTN telephone is used to call a Gateway, the incoming phone call is picked up, and the caller is requested to enter an ITX extension number followed by a #. This results in the caller being connected to a voice application. The Gateway uses its *DirectoryService* to look up the extension, finds where that user or application is currently accepting calls, and forwards the INVITE. Upon receiving an ACCEPT, the Gateway's *DesktopSignaling* opens the datapath between the phone line and the voice application.

The Gateway Manager must run on a computer equipped with voice card(s) which reformat and move voice data between the telephone network and the Internet.

4 Building Multimode Applications

Perhaps the distinguishing feature of ITX is its building-block approach to creating multimode applications. Figure 2 illustrates a few of the many possibilities for putting together the ITX components. Figure 2(a) illustrates using only the data exchange package. This would be useful for applications that want to send sound or voice data across the network, but which do not need telephones or directory lookup or signaling services. An example might be voice-based language-translation, as is currently being conducted at AT&T Research using our software.

A more typical ITX application is shown in Figure 2(b). Here the application adds directory services and a signaling layer by importing the `cnrg.itx.ds` and `cnrg.itx.signal` packages to implement an IP-based voice application. Now the ITX application also requires that the backend database server be running.

If one wishes to incorporate the PSTN into an ITX application, one needs to add a gateway computer (Figure 2(c)) and the Gateway component. Adding the ability to use the PSTN is the only difference between Figure 2(b) and Figure 2(c).

Although the gateway can both accept incoming calls and place outgoing calls, ITX also has support for using a PBX to place calls under program control (see Figure 2(d)). At present ITX uses only the core features present in the PBX (a Lucent Definity G3), but it would be possible to extend the `cnrg.itx.gtwy.pbx` package to access more functionality.

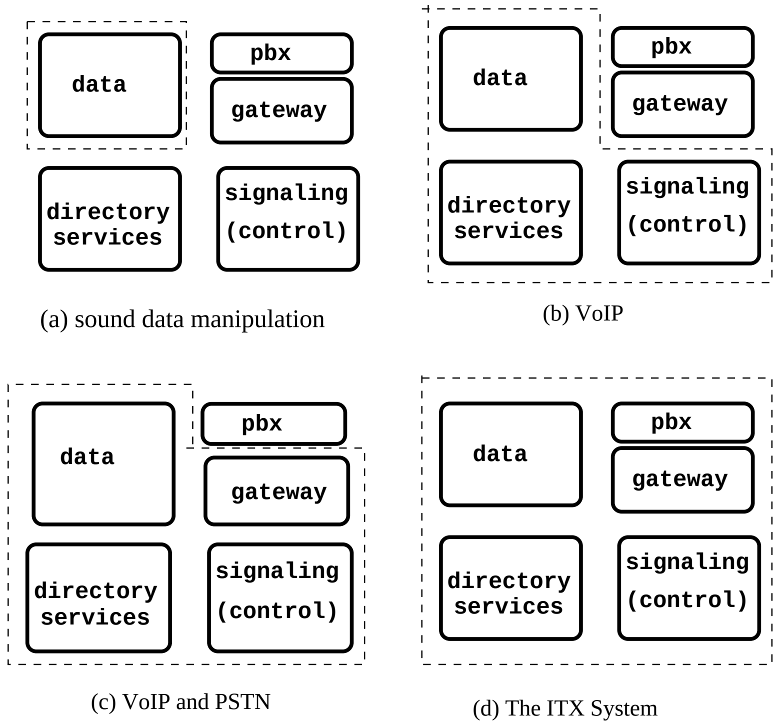


Figure 2: *Building different applications from one library*

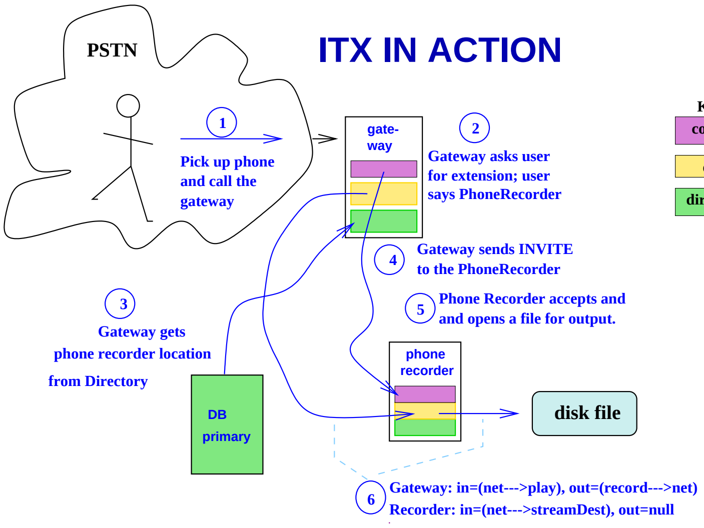


Figure 3: *The PhoneRecorder and Gateway are server applications and are already running, and registered with the directory server. This figure shows what happens when a user calls up the PhoneRecorder to leave a message.*

5 An Example

In this section, we walk through the design and implementation of a simple ITX application. Figure 3 illustrates the PhoneRecorder that uses several components of ITX. This application allows one to use a telephone to create a recorded message. It runs as a server, waits for an incoming call, then writes everything to a file until the client hangs up.

Because the PhoneRecorder must handle Invite and Hangup events, it extends *AbstractSignalingObserver*; it implements (overrides) two event handler methods, `onInvite` and `onHangup`.

The first thing the PhoneRecorder application does when launched is to get a Desktop Signaling object:

```
myDS = new DesktopSignaling ( this, "PhoneRecorder", "password");
```

The *DesktopSignaling* constructor registers PhoneRecorder as its signaling event handler, and authenticates by specifying a legal userid and password. The current running location of the PhoneRecorder is then entered into the directory (the PhoneRecorder's ITX extension number is already in the directory).

Figure 3 shows the PhoneRecorder in action. When (1) a user calls a number on the gateway and then (2) specifies the extension number of the PhoneRecorder server, the gateway asks its directory component to look up the current location(s) of the PhoneRecorder. The directory returns this to the gateway(3). (Since the PhoneRecorder has already registered itself, the location at which it is running is available from the directory.)

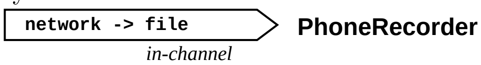
The gateway passes the INVITE on to the PhoneRecorder, along with information relating to the data channels it will be using (sampling rate, and so forth). This invitation is received by `myDS`, which has been sitting around waiting for an incoming call.

The next task is to set up the appropriate data flow, so that the user can create a file on disk just by speaking into the telephone. The gateway's channels are set up as follows:

An event containing this information is constructed by `myDS` and passed to its signaling event handler. This handler can be used to finish setting up the data connection:

```
public void onInvite ( ... ) {
   Channel in = new Channel();
   :
   in.setSource ( new NetworkSource (in, ... );
   in.addDestination ( new StreamDestination(ourFile) );
   c = new AudioConnection ( in, null );
   c.open();
   :
```

Here, assume that "ourFile" is a file that has been opened for output. We are not interested in interacting with Gateway's in-channel, but we do wish to hook up to Gateway's out-channel. The handler thus sets up this in-channel:



In ITX terms, this channel has a `NetworkSource` and a `StreamDestination`. PhoneRecorder's audio connection to the gateway uses this input channel and a null output channel. The null output channel hooks up with the Gateway's in-channel, which would normally play recordings to the telephone. No data flows this way; however, we do have a conduit for voice data running from the telephone through the gateway out to the net into the peer PhoneRecorder application and out to disk. At the end of `onInvite`, no data is flowing, but the channel is open.

The handler exits back to `myDS`, which (5) sends an ACCEPT packet to the gateway specifying the data port to which data should be sent. Upon receiving the ACCEPT, the gateway opens its end of the audio connection and whatever the user says is copied to file (6).

What happens when the user hangs up has been omitted from Figure 3. The gateway detects the hangup, closes its end of the audio connection, and sends a HANGUP packet to the PhoneRecorder. PhoneRecorder's hangup handler is simple:

```
public void onHangup ( ... ) {
  :
  ourFile.flush();
```

```
  ourFile.close();
  :
}
```

Note that `myDS` closed the data connection when it got the HANGUP packet and before invoking `onHangup`.

With an API to initiate activities (e.g. `Hangup()`) and event handlers to react to events (e.g. `onInvite()`), ITX applications have a simple way of interacting with the underlying ITX core telephony components.

6 Performance and Scalability

We have measured performance of ITX applications running on the same 100 Mbps subnet, behind a firewall. The times given are wallclock times. In particular, the following times seem reasonable for Java:

- Latency between dialing and phone starting to ring: 2-7 seconds
- Latency for data transmission: 50–250 msecs for IP-to-IP; 250-500 msecs going through the Gateway.
- Directory lookup time: 7–16 msecs, with 16 msecs for the first call, and rapidly dropping toward 7 for subsequent calls.

The system, by design, is very scalable. This is a fully distributed system: ITX servers and applications can all run on separate computers or the same computer.

The directory service, because it is based on BIND, can be implemented on a number of servers spread across the network. We used one primary server and one backup. The directory is cleaned up periodically by detecting user and application roaming locations that are no longer active. (A roaming location could become inactive if an application crashed without unregistering its current location from the directory, or if a critical network link went down.) This means the directory will not fill up with bad data.

An ITX system can have as many Gateway managers running as there are Gateway computers. These days a single gateway machine can handle hundreds of telephone lines. Each gateway is a separate entry in the ITX directory. Once a Gateway manager comes online, if it is running with the PBX switch enabled, it determines which PBX to use. (Thus there can be many PBX machines and many Gateway machines, but once they begin running there is a one-to-one PBX per Gateway relationship.)

Signaling is distributed, since it is bound into each running ITX application. There are no central signaling servers, so no single point of failure. Individual failures are detected when a signaling component is no longer sending out heart beats, its associated application having failed. Thus the signaling component of ITX is perfectly scalable.

Data transmission is also distributed, since sound handling capability is bound into every ITX application. Sound delivery can be one-to-many, so it is possible for the sending computer to become overloaded if it has too many destinations to handle; however the ITX applications programmer could get around this by setting up a tree to deliver the sound to many applications, thus reducing the potential number of destinations per application instance, but increasing communications delay.

Thus the ITX core platform is scalable; an ITX application will be as scalable as the ITX applications programmer makes it.

7 Conclusion

The ITX software is being deployed now. We have written several applications (a workstation-based telephone and directory manager, a voice mail application, an alarm clock that rings a telephone at some pre-determined point in time, and a lecture server). It is in its first release, and we are therefore eager for others to try it out.

The system needs to be made more portable; currently is is written for the NT platform, but plans are to port it to Unix. We would like to use JavaSound when we can get a good implementation of that package.

The system is robust because it uses the Java event model; it is multi-threaded, thus making optimal use of computer and network resources; it uses "keep-alive" signals to determine the health of peer applications; the directory keeps itself clean, and Data Exchange uses Forward Error Correction to reduce jitter and adaptive buffering to minimize dropouts. The speech quality depends mostly on the pedigree and age of the computer running the application. With newer machines ($>$ 300 MHz processors and a duplex sound card), the sound quality is excellent.

Use of ITX is now limited by the imagination of application developers. We look forward to many interesting telephony services in the future.

8 Acknowledgments

The real work on this project was done by a talented group of Cornell students: James Barabas, Jason Howes, Char Shing "Wilson" Ng, Madhav Ranjan, Naveen Sastry, Pratap Singh, James Wann, and (last but not least) Jingyang Xu.

References

[1] "IDC 1999 factsheet: IP telephony," 1999. `http://www.idc.com/Factsheets99/99Services/Internet/ipt_.htm`.

[2] C. Polyzois, K. Purdy, P. Yang, D. Shrader, H. Sinnreich, F. Ménard, and H. Schulzrinne, "From POTS to PANS: A commentary on the evolution to internet telephony," *IEEE Network*, pp. 58–64, May/June 1999.

[3] S. Keshav, "Multiplanar applications and multimodal networks," 1999.

[4] H. Schulzrinne and J. Rosenberg, "The session initiation protocol: Providing advanced telephony services across the internet," *Bell Labs Technical Journal*, vol. 3, pp. 144–160, October–December 1998.

[5] H. Schulzrinne and J. Rosenberg, "The IETF internet telephony architecture and protocols," *IEEE Network*, pp. 18–23, May/June 1999. Online version: `http://computer.org/internet/telephony/index.htm`.

[6] IETF, "RTP: a transport protocol for real-time applications," Jan. 1996. RFC 1889.

[7] N. Anerousis, R. Gopalakrishnan, C. Kalmanek, A. Kaplan, W. Marshall, P. Mishra, P. Onufryk, K. Ramakrishnan, and C. Sreenan, "TOPS: An architecture for telephony over packet networks," *IEEE Journal on Selected Areas in Communications*, vol. 17, pp. 91–108, Jan. 1999.

[8] F. Anjum, F. Caruso, R. Jain, P. Missier, and A. Zordan, "*ChaiTime*: A system for rapid creation of portable next-generation telephony services using third-party software components," in *1999 Second IEEE Conference on Open Architectures and Network Programming (OPENARCH99)*, IEEE, 1999.

[9] C. Gbaguidi, J. Hubaux, G. Pacifici, and A. Tantawi, "An architecture for the integration of internet and telecommunication services," in *1999 Second IEEE Conference on Open Architectures and Network Programming (OPENARCH99)*, IEEE, 1999.

[10] S. Moon, J. Kurose, and D. Towsley, "Packet audio playout delay adjustment: Performance bounds and algorithms," *ACM/Springer Multimedia Systems*. To appear.

[11] P. Albitz and C. Liu, *DNS and Bind*. O'Reilly, 1998. ISC Bind site: `http://www.isc.org/products/BIND/`.

9 Appendix: Technical Specifications

ITX is distributed as a `gzip` file [1], containing the four main components discussed in the paper, as Java source. The distribution also includes two major applications: CUPS, which acts as a virtual telephone as well as a directory manager, and SPOT, a lecture server. Finally, the distribution includes two basic telephony servers: the Gateway manager and a PBX server.

The third server, which is required, is the backend database server. A database client, which invokes only four different functions on the backend, is included in the distribution, but the database server should be provided by the people installing ITX. At Cornell, the ITX project uses BIND [11] because it is very reliable, it is free, and it runs on many platforms. By extending current resource records, we can store users, extensions, and current locations in the DSN database. We note that this approach is also used by some telephony vendors.[2]

Cornell's gateway computer is equipped with a Dialogic D/41EPCI voice card which has 4 RJ11/14 jacks leading to ordinary handsets. This card takes care of turning the μ-law PCM sound data from the telephone into raw binary PCM data. It is controlled by a Dialogic library. A couple of key routines in this library are `ds_play`, which sends voice data to the telephone line, and `ds_record` which captures sound coming in over the telephone line. The library also includes functions for handling DTMF, detecting incoming calls, and hanging up.

The ITX software system is almost all written in Java. It was developed for the Microsoft Windows NT platform, using jdk 1.2 and Visual J++ (Visual Studio 6.0). It also uses JTAPI (Java Telephony API) for controlling a PBX, and some C code to control the voice board and sound devices. Our implementation handles audio data via an RNI (Raw Native Interface) to Microsoft's sound library. This interface, `jaudio.dll`, ships with the ITX system.

The entire source tree is available to the public, with the exception of gateway libraries from Dialogic and the JTAPI implementation from Lucent. If you buy a Dialogic voice card, then you should be able to use our source as is once you have installed Dialogic's SDK. If you have a PBX, you should be able to get a JTAPI implementation from your PBX vendor.

The distribution includes a Programmer's Guide.

[1] See `http://www.cs.cornell.edu/cnrg/telephony/JavaDocs/install.html`

[2] E.g. Oki Network Technologies, in their BS-1200 product (`www.okint.com`).