THE ETHERNET-TO-PHONE TELEPHONY SYSTEM

By

Susheel Daswani, Sumit Bhansali,
Siva Gaggara, Ashish Shah, Satyam Vaghani

CS344: Projects in Computer Networks
STANFORD UNIVERSITY
Fall 2000

Acknowledgements

We would like to take this opportunity to thank Prof. Nick McKeown for inviting us to participate in the course ‘Projects in Computer Networks’. Prof. McKeown, your thoughtful comments helped us organize ourselves to undertake a comparatively large-scale project.

Our ‘loud’ Wednesday afternoon meetings with Paul Hartke, the Teaching Assistant for the course, ensured that our work was evenly paced out during the quarter and there was no need of a last minute rush. Thanks, Paul.

The team is grateful to Prof. Henning Schulzrinne at Columbia University, NY, for granting us a license to use the Columbia SIP library in our development. Acknowledgements are also due to Kundan Singh, his student at Columbia University, for responding to all our SIP library related queries.

- S.D., S.B., S.G., A.S., S.V.

Table of Contents

1. Introduction

1.1 Problem Description and Overview

The Eth2Phone project was undertaken to solve a specific problem: the Gates Computer Science building here at Stanford University affords its tenants a limited range of telephony options. Specifically, offices shared by groups of graduate students are equipped with one telephone line, adding to the pre-existing tension amongst this already disgruntled group of people. In stark comparison to this dearth of telephony lines, offices are equipped with an abundance, if not an overflow, of data lines. Furthermore, these data lines are serviced by a state of the art, very high performance Ethernet network. Given these parameters, the Eth2Phone project team undertook the task of alleviating the severity of this situation by leveraging the data services individual users have for telephony purposes. The Eth2Phone team is adding a high quality telephony service to specific clients with access to a data line. Using clients supported by the Windows 2000 platform, people can dial into the Stanford University Telephony network, fundamentally turning their PCs into a telephone.

This problem is an interesting one to solve because it demands that a heterogeneous set of tools be integrated efficiently in order to provide this high quality telephony service to Gatesians. Modern software engineering demands an integration of a diverse set of tools, platforms, and applications, as emphasis has shifted from low-level issues to building larger and larger systems that provide new and once unattainable features.

Moreover, this project is interesting because it necessitates that Eth2Phone design be an extensible system. At every step, the team has sought to provide clean interfaces and components that are essentially 'plug and play', allowing future software development teams to build upon our successes.

Lastly, the opportunity for innovation in the telephony field is also present, as we could add capabilities to this telephony service that existing IP Telephony (Dialpad, Yahoo Messenger, etc.) networks have yet to provide. In particular, we are building a infrastructure that will either incorporate or be very easy to extend to support 'incoming calls': rather than just allowing PCs at Gates to make outgoing calls, allow these PCs to be called by any phone.

The field of IP Telephony, though relatively new in the scheme of things, is quite mature. As previously mentioned, various IP Telephony services exist to this day, the most notable being the DialPad.com and Net2Phone genre. These services allow PC users to add a telephony interface to their clients, avoiding long-distance fees. Moreover, various messaging protocols, such as AOL Instant Messenger and Yahoo Messenger, have incorporated Telephony features. The maturity of this field was a boon to our development cycle, as pre-existing tools addressed major issues a system of our type faces. Nevertheless, plugging all these tools together has been the real challenge.

1.2 Our Solution

With our system in place, a Windows 2000 user can make a call from his/her PC to any telephone that the Stanford telephone system allows one to connect to. The call is routed via a telephony gateway (a Linux box) that connects to the Stanford PBX (Public Branch Exchange). An extension of our present system would support incoming calls, where a user can receive a call on his/her PC. Also, PC-to-PC call capability is automatically supported by the system. We describe the system architecture in detail later.

This project has been a systems-integration project. We wanted to provide a useful system in a short period of time and did not want to reinvent the wheel. At the same time, we wanted the system to be extensible and not be limited by the choice or state of the existing components that we decided to use. The project was intended to be an open-source project that would be freely distributed with a license. We decided to build the system around SIP (Service Initiation Protocol) and related sets of protocols. (A detailed section on design choice follows). Columbia University has built an extensive system around SIP and we chose to re-use their SIP implementation. This choice fulfills multiple goals: their code is under academic license for development, the architecture is of the plug-and-play nature and is very extensible. We discuss later how we built our system around the Columbia University's SIP implementation.

2. System Architecture

A diagram showing the high-level system architecture is shown in Figure 1. Before we dive into the detailed system architecture and implementation, we will discuss briefly the design choices that we made.

2.1 Design Choices

Why SIP and not H.323? SIP (Service Initiation Protocol) and H.323 are two major suites of protocols that have been extensively used for IP Telephony systems. In our system, these protocols would be used to handle call management between the client and the gateway across an IP network. The gateway would then communicate with the telephony world. SIP comes from the "Internet World" and is the protocol promoted by IETF, along with SDP (Session Description Protocol) that goes hand-in-hand and supports description and negotiation of session parameters (media, formats etc.). H.323 is a suite of protocols standardized by ITU-T (International Telecommunication Union, formerly CCITT). There are tools that support both the protocols and systems have been built using both of them. One of the major benefits of SIP (and SDP) is that it is very lightweight, simple and flexible. On the other hand, H.323 incorporates many ideas and is therefore a very complex and heavyweight protocol. For example, H.323 specifies the protocols to be used for media transport in addition to the signaling. With SIP/SDP, one can use either RTP/RTCP or any user defined transport protocol (over, say UDP) for media transport. Another major weakness of H.323 (version 1) is the call setup delay – it takes about 6 to 7 RTT (round trip times) to set up a call. In case of SIP/SDP, this delay is just 1.5 RTT. Although H.323 has a huge installed base, more and more people prefer SIP/SDP for new implementations. There are lots of resources on the comparison of the two protocols. After going over these resources and talking to people in academia (Columbia University) and industry (Cisco Systems), we decided to go ahead with SIP/SDP.

Choices for SIP implementations: Given that we had decided to use SIP, we had two main choices: build it from scratch or use an existing SIP implementation. Various factors we had to consider included time of development, maintaining the open-source goal of the system, and extensibility and modularity of our architecture. We looked at a few choices of existing implementations: Columbia University's SIP library (libsip++, which is a part of the CINEMA system), DynamicSoft's SIP user agent, SIP Center's SIP API, Vovida's SIP implementation. We decided to choose Columbia University's SIP library.

Why Columbia University's SIP implementation? Columbia University's CINEMA (Columbia Internet Extensible Multimedia Architecture) is an extensive system for multimedia communication across the Internet. It has been principally built around SIP. We decided to license (for academic use) their SIP implementation library (libsip++) to build our system. The architecture is very modular and it has precisely the components that were necessary. Also, our system fits very nicely into theirs. For supporting incoming calls, we will use their implementation of the SIP redirect, proxy and registration server (sipd), which is also a part of the CINEMA system.

By reusing the SIP code from Columbia, we reduced our development cycle considerably. We did not have to re-invent what has already been built and tested and is available for non-commercial use. This gave us time to incorporate interesting features and build a useful system in a short period of time. The library is supported on Unix, Linux, and Windows NT machines, the systems that we were interested in building our system around. The implementation is in C/C++, which is good considering the real-time nature of the system.

Compared to the other choices of SIP implementations, Columbia SIP library fulfilled all the objectives. DynamicSoft's implementation is in Java. We considered using it on the client side, but discarded it once we decided that Java was too slow for voice capturing. Also, they had a limited license of 90 days, after which there was an "unspecified" amount of fee involved. The Vovida SIP implementation is also in Java. SIP Center has no SIP implementation yet, just an API. Thus Columbia SIP library was the natural choice.

We should point out that using an existing system and building around it was not as simple as it initially seemed. Understanding the Columbia system took a considerable amount of time, as did the design of an extensible system that interfaced to their SIP library.

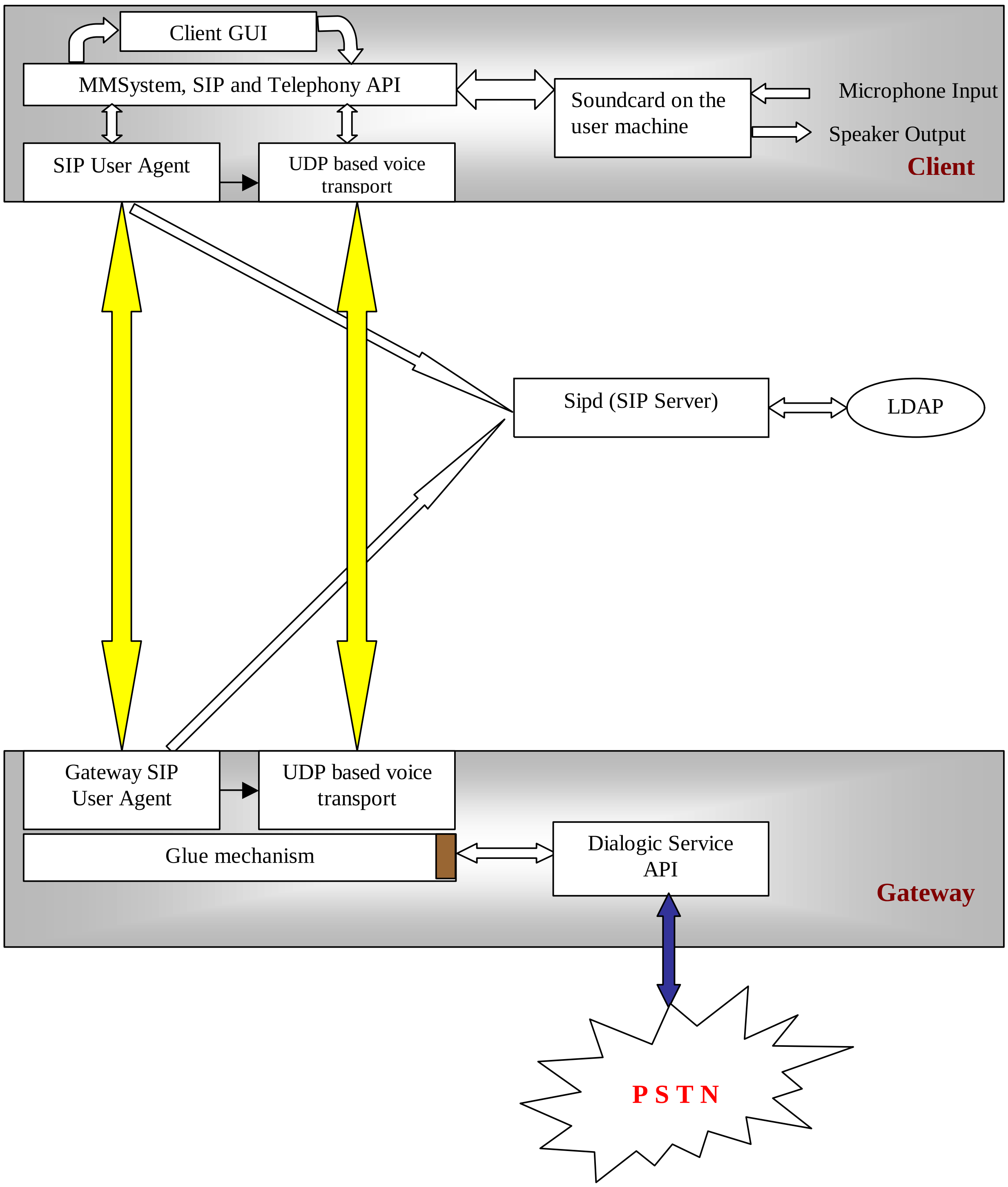


Figure 1. The Eth2Phone System Architecture

Voice Transport:

Currently we plan to use UDP as the transport protocol. Since the assumption is that the network communication would be over a fast (10MBPS –1GBPS) and reliable (department & campus LANs) network, the chance of packet drops and reordering would be minimal. However, if required, moving to protocols like RTP/RTCP over UDP would be very simple. RTP/RTCP take care of reliably transferring data across the network, including taking care of reordering of packets. Between the extremes of RTP/RTCP (a sophisticated protocol) and UDP (no protocol), we also propose a middle-ground simple protocol over UDP that takes care of packet reordering using sequence numbers but has no other overhead from RTP/RTCP. The header of this protocol would look like:

V	Seq

✍------------------------------------32 bits--------------------------------✍

V = 2 bit version
Seq = 30 bit sequence number

2.2 Client-Side Architecture

In light of our desire to build a component based, ‘plug and play’ system, the client side architecture has clear division between three elements: user interface, sound capture and playback, and call management and voice transport. Two Application Programming Interfaces (API) separate these components.

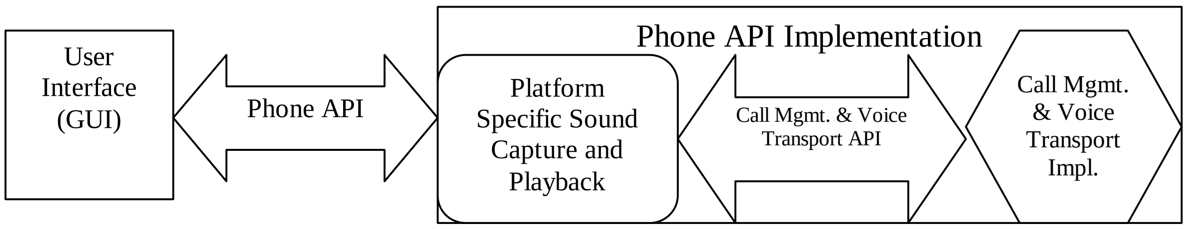

User Interface (GUI)
Phone API
Phone API Implementation
Platform Specific Sound Capture and Playback
Call Mgmt. & Voice Transport API
Call Mgmt. & Voice Transport Impl.

Figure 2. Client-Side Architecture

User Interface (UI)

Our UI will consist of an abstraction of a Telephone. The important thing to point out here is that the User Interface is not architecturally important to our system –anyone can build any sort of UI once given access to the Phone API.

Phone API

This API provides a black box that resembles operations one can perform on a phone. *All APIs displayed are not exactly the APIs exposed by our system; rather, they are abstractions that serve the purposes of this discussion*.

```
Phone API {
      call(PhoneNumber number);
      hangup();
      pickup();  // for an incoming call
}
```

The API is simple; it provides a neat abstraction for different application developers to deal with.

Phone API Design

The design of the conceptually clear Phone API consists of two main components, separated by another API. We felt this was necessary, as our research showed that sound system access varies among different platforms. Contrary to the platform specific nature of sound capture and playback, call management and voice transport could be *more* easily ported across machines. Nevertheless, strategies for implementing call management and voice transport can easily vary. This fact further testified to the need to separate these two components, as new strategies for call management and voice transport implementation should not affect the implementation of sound capture and playback.

Sound Capture and Playback

This component provided some controversy for our project. At first, it was hoped that a platform independent solution could be found (i.e. "one client fits all"). The Java Sound API is platform unspecific and easy to use. Unfortunately, it suffers from performance issues that were deemed too negative to justify its use. In the future, we hope that new implementations of this API will provide better results; this would more easily extend use of Eth2Phone across a heterogeneous user base.

With that said, an efficient, platform specific, solution was found. Using the Multimedia Sound System API (mmsystem.h) under Windows 2000 (Service Pack 1) yields excellent results. It is hard to say at this point how small the overall voice delay in our system will be, but it is clear that any delay will not originate from the client application.

A more complete discussion of the Sound Capture and Playback piece, including experimental data, can be found in the Client Side Implementation section.

Call Management and Voice Transport API

This API hides a lot of the details regarding call setup, teardown, and voice transport during calls. It can be viewed as an extension of the Phone API with a more details about voice exposed.

```
TelephonyEnabler API {
        registerCallBack(TelephonyPeer);
        call(PhoneNumber number);
        hangup();
        pickup();  // for an incoming call
        codec();   // for alternate voice representations
        sendVoice();
}

TelephonyPeer API {
        sendVoice();
        notifyIncoming();
}
```

The Call Management and Voice Transport component is a TelephonyEnabler, allowing a user to initiate calls, hang up calls, pickup incoming calls, and send voice data. The sound component is a TelephonyPeer that registers itself with the TelephonyEnabler; this provides hooks for duplex voice conversations and incoming calls. The codec() method provides functionality that allows users to convert from a non-supported sound representation to the one used by the Eth2Phone network.

Call Management and Voice Transport

This component provides a lower level abstraction of a Telephone to the sound component, as can be seen in the TelephonyEnabler API. Our specific implementation uses a pre-existing tool (SIP) from Columbia University for call management. The voice transport feature is implemented with a simple, open source, UDP-based protocol tailored to run on the Gates high performance Ethernet.

2.3 Gateway Architecture

The gateway consists of two major components: the SIP User Agent (sipua) and the Telephony Card hardware/software (In our case the Dialogic API). The sipua interacts with the client sipua across the network using SIP/SDP. This is built around the Columbia University's SIP implementation [Columbia SIP Library]. The other component is the Dialogic API that talks to the Dialogic card that connects to the Stanford telephone system on the gateway. For incoming calls, a directory service (e.g. LDAP) that resides on a SIP server (sipd of Columbia University) can be easily integrated to our system.

The gateway supports multiple simultaneous calls, bounded by the number of telephone channels available to the dialogic card. The multithreaded implementation of the gateway is described in the implementation section.

The SIP User Agent (SIPUA)

Outgoing Calls: When a PC user initiates a call, the client side sipua initiates a "sip call" with the gateway side sipua. The request contains the phone number to be called. Through the SIP library, the gateway code receives this request and makes a call setup request to the Dialogic API, which checks if there are any telephone channels available to make a call. If there is one, a telephone call is initiated with the remote telephone user. If the user picks up, the Dialogic API returns success and in turn, the SIP call between the client and the gateway sipua's is set up. At this time, a voice transport channel is also set up between the two sipua's and voice transport is activated between the gateway sipua and the dialogic card. Note that the SIP call between the two sipua's is set up only after the dialogic card sets up a telephone call to the remote user. We chose to do the call setup synchronously rather than asynchronously (i.e. complete the SIP call before the telephone call is setup) because the former is more logical as well as straightforward to implement: we can use the SIP signaling functions to describe call setup success and failures on the PSTN side (as in the former case) rather than do it otherwise (as in the latter case).

Either side can terminate a call. If the PC user decides to terminate the call, a SIP "BYE" message is sent via the client side sipua to the gateway sipua. The gateway terminates the call to the remote telephone user and also closes the SIP call. If the remote telephone user hangs up the phone, the Dialogic API returns this condition and a "BYE" message is sent to the client sipua to terminate the SIP call.

Incoming Calls: Incoming calls are handled asynchronously rather than synchronously between the PSTN and the IP network (using SIP). An incoming call is detected by the dialogic card, which alerts the gateway of this condition. The gateway completes the telephone call by accepting the call and asks the remote user to enter an extension number. A telephone number dialed by a caller from the PSTN is used to reach the gateway. From there on, an extension number is used to identify a specific user. We could also provide an extensive directory service that uses names to identify a user. Once the gateway has the extension number, it contacts the SIP server (sipd) to lookup a user's current location (IP address). We assume that the users would keep the SIP server up-to-date about their location. After having found the IP address, the gateway initiates a SIP call to the client sipua on the user's machine. If the user accepts the call, the SIP call is successfully set up and voice transport is initiated between the two sipua's as well as between the gateway sipua and dialogic. While the call set up is in progress, the telephone user is played a recorded message (e.g. "Please wait, your call is being processed"). The call termination would work in a similar way as the outgoing calls.

Transcoding of Media Formats: SDP (Session Description Protocol) can negotiate media formats that must be used between the gateway sipua and the client sipua. If a client does not support the default media format, the gateway can transcode the media format to the one supported by the client. Currently, we assume that the gateway would be powerful enough to do this transcoding on the fly without introducing huge amount of delay. In future, we might consider putting this functionality to a different machine and use, for example, RPC to get transcoding done. There is an API built into the Gateway to select appropriate transcoders as well as add more transcoders.

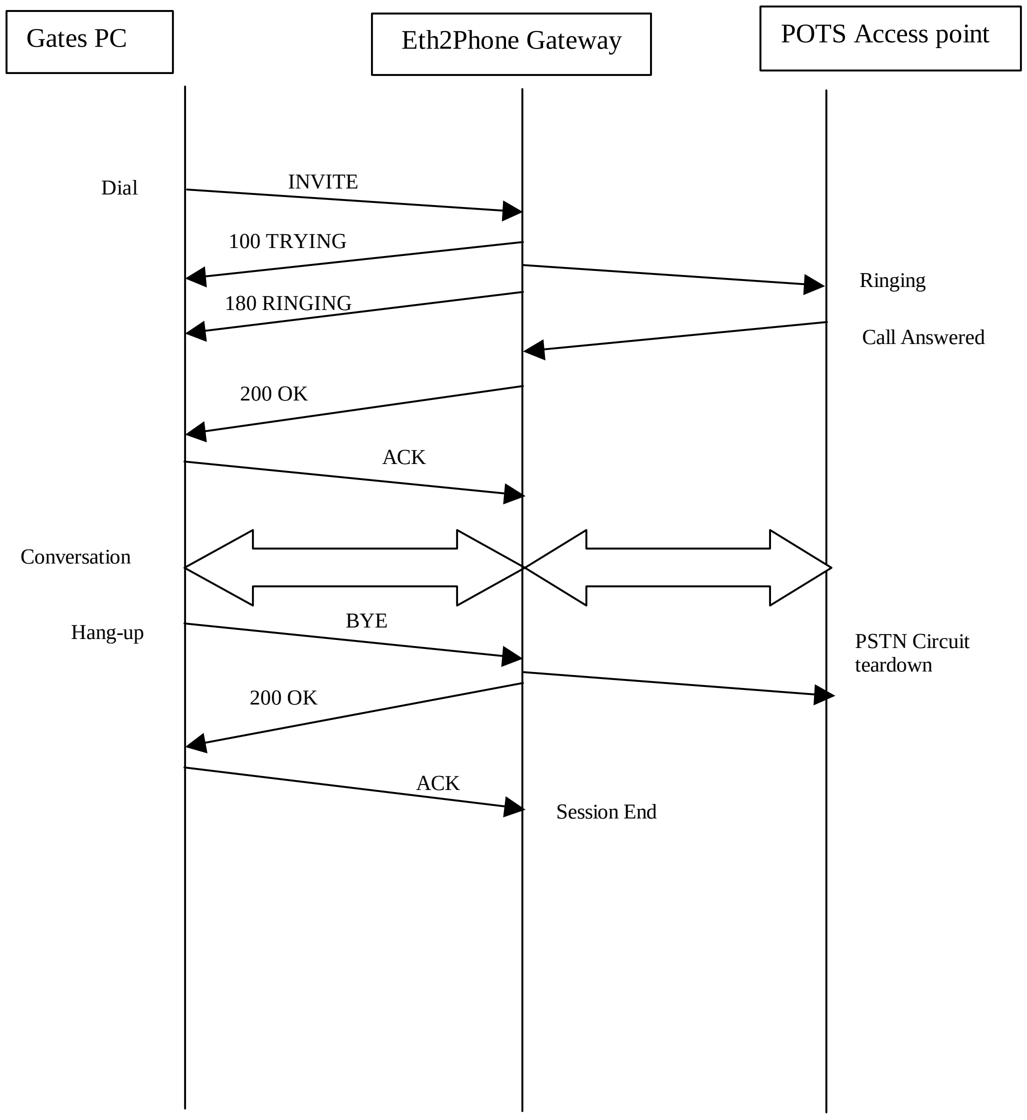


Figure 3: SIP Call - Illustrated

Please note: The transmission delays (slopes) do not represent actual transmission delays on the eth2phone target network. This diagram is for SIP illustration purposes only. It is not suitable for use as a benchmark. Also, the transmission sequence illustrates the duration of an ideal call – it does not take into account the intricacies of UDP packet loss and re-transmissions. These details are not relevant to the topic being discussed in this diagram.

Dialogic :

D/21H and D/41H cards:

The gateway connects to the PSTN (Public Telephone Switched Network) through a card that serves as the interface between the analog phone line and the gateway. We are using the Dialogic D/21H high-performance 2-port voice processing board, as the interface between the phone network and the computer. Though only two simultaneous calls can be supported with the 2-port D/21H card, it is possible to support 32 simultaneous calls by installing 16 D/21H cards in a single host. In fact, it is possible to support up to 64 simultaneous calls by installing 16 D/41H, which have four ports per card.

Voice encoding and sampling rates:

D/21H card provides flexible voice coding at dynamically selectable data rates. Four data rates are supported: 24Kbps, 32Kbps, 48Kbps, and 64Kbps. Two voice coding schemes are supported:
ADPCM (Adaptive Differential Pulse Code Modulation) and PCM (Pulse Code Modulation). 6 KHz and 8 KHz sampling rates at 4-bit and 8-bit per sample encoding are supported. Several flavors of PCM are supported. These include linear PCM, ?-law PCM, and A-law PCM. ADPCM is the default encoding for recording and playback; however, with its 2:1 compression, ADPCM is not as good as linear PCM as far as voice quality goes and so, we decided to support linear PCM as the default. We record in wave format to send to the client and play back in vox format to the person on the phone. Vox is a special dialogic format that is suited for playing digital voice on the phone line. It is important to use the same sampling rates and encoding schemes in both recording and playback. The gateway must transcode the voice data from the client to what is supported on dialogic. For example, if the client transmits voice data using GSM encoding which is unsupported on dialogic, the codec in the gateway must convert the voice data from GSM to one of the formats supported by dialogic.

Hardware architecture:

The card communicates with the host through the PC XT/AT bus. There are two on-board processors: control processor and the Motorola DSP processor. The control processor that serves as the central control point (brain) of the card connects to the XT/AT bus through the IRQ line. Multiple cards if used will all share the same IRQ line. The DSP processor connects to the control processor through a control bus on the board. Both the DSP processor and the control processor have their own RAM. The DSP performs the following key functions:

- Expansion of stored, compressed audio data for playback.
- Compression of audio data captured during recording.
- Generation of DTMF tones.

The DSP processor connects to the CODEC (coder/decoder) that in turn connects to the line interface, which is connected to the PSTN network. The CODEC filters, samples, and digitizes the incoming analog signal and also converts the outgoing digital signal into analog form. When the card is initialized (turned on), the dialogic SpringWare firmware is downloaded from the host to the on-board data RAM and DSP RAM to control all board operations. Before running any application, it is important to turn on the card by running a particular command (described in the implementation).

How the dialogic card works?

The D/21H card can place outgoing calls and receive incoming calls.

Outgoing calls:
The card can dial out an ASCII string of digits by generating the DTMF tones corresponding to the digits.
The phone is set off-hook before dialing out the string of digits. If there is some problem in setting up the call and call analysis feature has been enabled, the reason for failing to finish the call-setup is returned to the

application. There are several reasons for call-setup failure. The called party could be on the phone already, in which case a busy tone would be detected. The dialed number given is not a valid number. Nobody picks up the phone on the other hand, in which case there is a timeout after a certain number of rings. Appropriate error codes are returned in each of the above cases to signal to the application the cause
of failure.

Once a call is setup, it is possible to play voice data from the host buffers to the callee and record voice data to the host buffers from the callee. The firmware buffer, whose size is programmable from 128 to 512 bytes, captures the voice from the callee. The data is then transferred to the driver buffer before being recorded to the application buffer (host buffer). The size of the driver buffers is also configurable to any value between 256 bytes and 16KB. The voice data from the caller follows the same path in the opposite direction (from the host buffer to the driver buffer to the firmware buffer). Since the D/21H card is half-duplex, it cannot play and record sound data on the same channel simultaneously. So, play and record operations need to be alternated in the same thread. Even with alternating play and record, the quality is very bad, because the same buffers are apparently used for both recording and playback. The call is terminated when either party hangs up. The card then performs any necessary cleanup to prepare for the next call.

Incoming calls: The card can receive incoming calls by waiting on the channel for a ring. Once the call has been received, voice recording and voice-playback is the same as the recording and playback in the outgoing calls case. The call is terminated when either party hangs up. When the call is terminated, the dialogic card performs any necessary cleanup to prepare for the next call.

Why use a hardware-independent API?

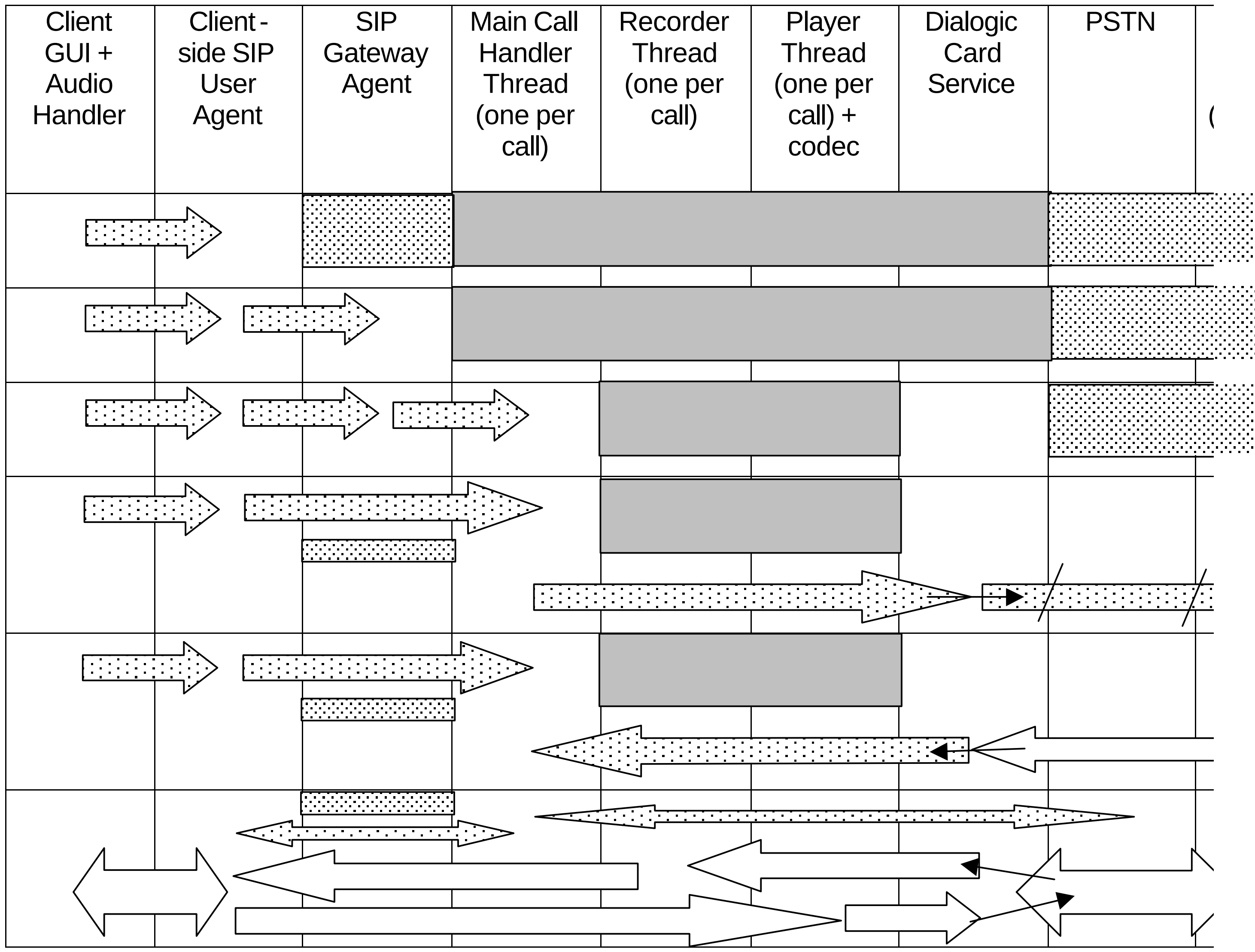
We have created a hardware-independent API to allow the gateway SIP user agent to seamlessly integrate with the telephony card that interfaces to the PSTN network. A hardware-independent API makes the system more usable by removing any dependence on a particular card. A module that satisfies the defined API can be written for a particular telephony card and plugged into the system. The hardware-independent API thus makes our system a plug-and-play system that is extremely flexible and highly usable. The API makes no reference to any of the hardware-specific features, and the hardware-specific module does all hardware-related initialization and control. The API increases the readability of the main gateway program and provides a useful layer of abstraction by hiding away the hardware details from the SIP user agent.

The implementor of the hardware module also needs to provide the header file that will be included by the main program. Since dialogic does not run in its own process and communication between the card and the main program occurs through callbacks, a call-information structure defined in the hardware-specifc header file needs to be passed with each function call made by the main program to the card. However, the main program does not ever need to look inside the structure or do anything else with it. The structure is merely to help the card keep track of which call is being referred to by the main program. For all intensive purposes, the structure serves as the call identifier.

3. Implementation

3.1 Overall Status

On the client side, every piece has been completed. The sound capture & playback component and the SIP component have been successfully integrated: Full duplex PC-to-PC calls are a reality. On the gateway side, we have completed the testing of signaling (call setup and teardown) between a client sipua and a remote telephone. The client and the gateway have been integrated; a PC can, via the gateway, call a phone. So, outgoing calls from the client to the phone are now supported. We have implemented the incoming call handling functionality on the gateway. At the time of writing this report, the incoming call implementation was being tested.


Client GUI + Audio Handler
Client - side SIP User Agent
SIP Gateway Agent
Main Call Handler Thread (one per call)
Recorder Thread (one per call)
Player Thread (one per call) + codec
Dialogic Card Service
PSTN

Legend:

▒	Components not present (instantiated) at a particular point of time
░	At a given instance, components which are instantiated but are passive/listening for activity
⇨	Direction of flow of control information (session initiation, media setup, signaling)
⋮	Control signal pending/ waiting to be serviced
⇒	Direction of flow of voice data

Please Note: This diagram illustrates the capability of the architecture. There is a provision for all the illustrated components in our current implementation but some components (for e.g. dual threads for voice transport on the gateway are not used in the light of the hardware used).

Figure 5. Thread interaction on a synchronized timeline

1	A user dials a number to set up a call. The eth2phone gateway is listening for new connections. No other threads except the listener threads exist in the system. The telephone network is on standby with respect to the eth2phone system. The client side GUI requests the client side SIP user agent to contact the gateway and set-up the call. The GUI can request call setup services through predetermined access points in the API.
2	The client side SIP user agent contacts the gateway UDP listener with a SIP INVITE. Meanwhile, the request from the GUI is considered to be pending because the reply to that request depends on the result of the gateway response to the client side SIP user agent. In effect, the client side SIP user agent has forwarded the call request to the gateway and blocked the client side GUI until response is received.
3	A call handler thread is spawned on the gateway to try to complete the client request. The call handler, in turn, checks with the dialogic card whether phone lines are available to serve the client.
4	Now that a separate thread is servicing the client request, the gateway listeners switch back to listening' state. Provided a channel is available to make the call on behalf of the client, the call handler thread routes a call through the dialogic layer into the PSTN. Please note that the column marked Dialogic Card Services' presents itself as a black box to the eth2phone system through a standardized API. For practical purposes, the gateway calls can be considered to be tunneled through the Dialogic card.
5	If the Dialogic service is able to setup a call, it'll signal the call thread to finish the call setup on the Ethernet side.
6	The client request is serviced and acknowledged. The player and recorder threads handle voice data throughout the duration of the call. The two threads communicate with Dialogic on one side and with the client voice transport handler on the other side. The main call thread still maintains a control channel with the client. This channel will be used to signal the end of call later. The main thread also maintains a control channel with Dialogic so that it can be notified if the call ends from the PSTN side.

Figure 6: Thread Synchronization explained

3.2 Client Implementation

Client Installation

To use the Eth2Phone client, perform the following steps:

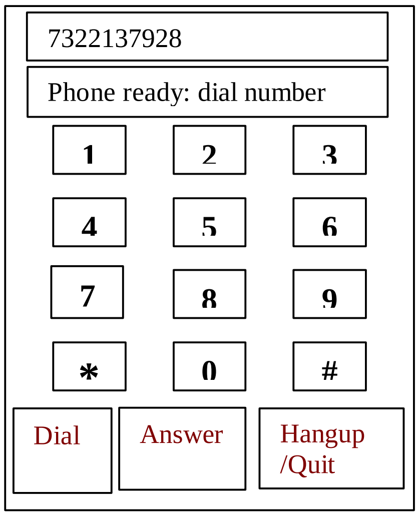
1. Install Windows 2000 on your T1 enabled PC.
2. Go to www.microsoft.com and download Windows 2000 Service Pack 1. Install it.
3. Get the Eth2PhoneClient.ZIP file and use WINZIP to unpack it to any directory you wish. Copy the DLLs provided in the zip file to your WINDOWS/SYSTEM32 directory.
4. Run Eth2PhoneGUI from the directory you unpacked the ZIP file in.

To develop client applications and/or modify the SIP or sound code, follow steps 1 and 2 from above. Then:

1. Unpack the Eth2PhoneClientCODE.ZIP file into the directory of your choice. Be sure to read README.TXT.
2. Install Microsoft Visual C++ on your system (6.0 or higher).
3. To modify the sip portion of the client, open libsipclient/libsipclient.dsw. Any new library file produced by this compilation should be copied to the Eth2PhoneGUI/LIBS directory. Then, build a new executable of the client; open Eth2PhoneGUI/Eth2PhoneGUI.dsw.
4. To modify the GUI, open Eth2PhoneGUI/Eth2PhoneGUI.dsw. This code links with the libraries in the LIB directory.

User Interface

The layout of the Graphical User Interface is very simple and intuitive.



This tool will allow convenient access to the Eth2Phone network.

Sound Capture and Playback Implementation

Java Sound API: Even though the Java Sound API was not used in our implementation, it is instructive to talk about our experiences with it. The Java Sound API allows application developers to write sound applications that are portable and hardware unspecific. The three major Operating Systems –Windows, Solaris, and Linux –all have a Java Sound API implementation. Given this fact, it is obvious that we thought first of using this package to implement the Sound Capture and Playback portion of our client; supporting a heterogeneous client base would be trivial. Nevertheless, the client needed to capture and playback any sound data without noticeable delay.

To test the efficiency of the Java Sound API, we built an experimental application called SpeakPlay. The SpeakPlay application used the Java Sound API to capture one's voice from the PC's microphone input and immediately send this data to the speaker output. The implementation of this application was 'no frills'; any delay experienced would be inherent in the Java Sound API Package.

The results of this test showed that Java Sound API, while extremely easy to understand and use, is not the best solution to problems that demand real time sound sensitivity. The simple SpeakPlay program would exhibit delays from anywhere between 500ms and 1.5 seconds (i.e. when you spoke into the microphone, the playback of your voice noticeably lagged). Voice quality and continuity were good, but the latency issue effectively overruled using the API for our client implementation.

MMSystem Sound API: The implementation of the Sound Capture and Playback piece is highly dependent on the Windows MMSystem API. This interface allows two important operations to be carried out: WaveInWrite(DataBuffer b) and WaveOutWrite(DataBuffer b). The WaveInWrite function writes the DataBuffer buffer with WAV data from the system's microphone. The WaveOutWrite function writes (plays) the DataBuffer WAV buffer to the system's speakers.

The basic operations of this component consist of filling an array of buffers with microphone WAV data (voice capture via WaveInWrite) and then funneling this data to the telephone session peer via the TelephonyEnabler interface (via sendVoice() method). Moreover, playback is accomplished via registration of a local object (a TelephonyPeer) as a callback to TelephonyEnabler. The TelephonyEnabler implementation then calls the TelephonyPeer sendVoice() method to affect playback of the telephone session's peer input data. In turn, WaveOutWrite is called with this data.

Direct access to the sound system via the MMSystem API is a mixed bag. The API is not the most easy to use, and as has been previously noted, it is hardly portable. This portability isn't only a problem for Linux and Unix platforms; older versions of Windows do not seem to provide the same results we have witnessed on the Windows 2000 (Service Pack 1) system. Given that, performance on this restricted platform is excellent.

The SpeakPlay application that was implemented using the MMSystem API exhibited the precise behavior that was necessary. As you speak, the speakers more or less instantaneously playback your voice. The key to discovering this good performance has to do with n-buffering, a variation of the common double buffering strategy. It is common when accessing low-level devices to double buffer; to provide a buffer to the device for writing its data to, while processing a buffer that was already filled by the device. We provide more than two buffers to the sound device; 128 buffers results in great performance. Moreover, the size of the buffers provided affects voice quality and latency. A larger buffer size would result in higher voice quality while increasing the playback latency, and vice versa. The magic buffer size we discovered is 160. 128 buffers of length 160 bytes resulted in the low latency telephony client.

In the future, we hope to support more platforms, either via use of an efficient, portable sound API or adding support on a platform-by-platform basis.

Call Management and Voice Transport Implementation

The implementation of call management is dependent upon the libsip++ API provided by the Columbia group. The underlying API implements SIP protocol for call management.

The handle provided by the API is the class SIPCall with virtual methods for application specific implementation. So our implementation extends the above class and overwrites the virtual methods according to our needs. Some of the important virtual methods of the SIPCall class are

`OnNewIncomingCall:` The library calls this when the remote user attempts to call the local user.
`OnHangup:` The library calls this when remote user hangs up the ongoing call.
`OnCallEstablished:` The library calls this when the remote user accepts the requested call and call is established.
`OnCallRejected:` The library would calls this when the remote user rejects the requested call.

The above methods are passive methods and are initiated by the remote user. There is a set of active methods provided by the API for actions initiated by the local user. Some of the important methods are

`Initiate`: This sends an invite to the specified location.
`Accept:` This accepts the incoming call.
`Reject:` This rejects the incoming call.
`Reinvite:` This sends duplicate invite to the same location. This is helpful in dialing extra digits like PAC.

The classes provided by the API that deals with the session description and media formats are SessionDescription and MediaInfo respectively.

The implementation of Voice Transport has been done using a simple UDP-based protocol. The voice transport provides an API for the call management. The important methods provided by this API are

`RegisterCallBack:` This registers a call back object required for dealing with the received voice data.
`StartListenerThread:` This starts a thread that listens for the incoming voice data at the specified port.
`SendVoice:` This method sends the voice data to the specified location.

Platforms on which the code is compiled and tested: Solaris, Linux and Windows NT.

In the future, we plan to expand the API provided by the call management to include features like multiple simultaneous calls, call waiting, answering machine etc. We are also planning on providing a Java API using JNI in order to integrate with already implemented Java sound clients.

The following figure shows the interaction between call management, voice transport, and sound component.

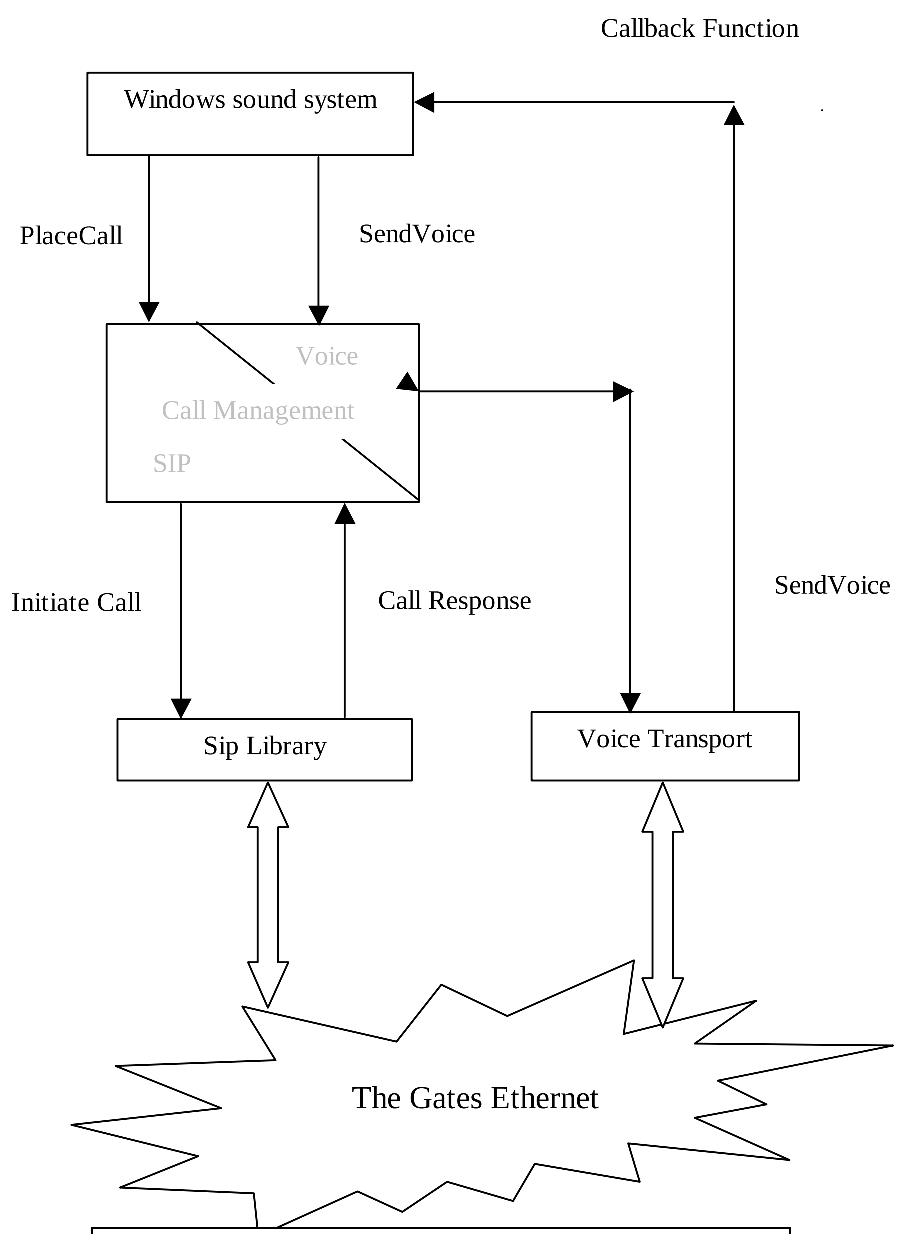


Figure 7: The Client Implementation

3.3 Gateway Implementation

SIPUA

The Columbia sip library (libsip++) is a C++ interface to a C implementation. The library supports SIP RFC 2543. The Columbia package also includes a sample SIP user agent built using the SIP library. We have built our gateway based on this user agent. Specifically, the library supports multiple simultaneous calls, but the example sipua supports only one. We have modified that to support multiple simultaneous calls. Also, there is a sample implementation of RTP (without RTCP) for voice transport. We have re-written the voice transport part to use UDP. Also, we have integrated this gateway code with the dialogic API.

Multithreaded Architecture of SIPUA: Libsip++ functions are thread-safe. In other words, it supports a multithreaded SIP service implementation. When the gateway starts, it instantiates a UDP and a TCP listener threads, which wait for incoming client requests (we use only the UDP listener thread).

OutGoingCalls: For every request from sipua client (including one for a new call setup, call teardown, acknowledgements etc.), the library spawns a new thread for processing it. In case of a new call request, a CallThread is instantiated, which stays until the call terminates (successfully or unsuccessfully). This CallThread is responsible for handling all requests for this call from the client side as well as those from the Dialogic side. Thus, there is one thread per call that stays until the call terminates.

After the CallThread is created for a new request, it requests dialogic to setup the telephone call. If the call setup succeeds, the CallThread sends acceptance (a SIP "OK" message) to the client side sipua using the library API. (Note that the library creates a new thread for handling every request and response). At this time, the CallThread starts voice transport thread(s). Since the dialogic card is half-duplex, in the current implementation, there is only one thread that handles communication from the network to the dialogic card (play) as well as the communication from dialogic to the network (record). In the ideal case if a full-duplex telephony card that supports streaming of voice, there will two independent voice transport threads: one for voice transport in each direction. Currently we have written a voice transport implementation for the walkie-talkie application (described later). We have written a skeleton implementation for the full-duplex streaming case.

While the call is active, the CallThread thread simply waits for more signaling commands and don't play a part in the call. If the PC client terminates a call, the CallThread receives a "BYE" message. It terminates the call on the PSTN side as well as the SIP call between the two sipua's and stops the voice transport thread(s). Similarly, if the PSTN user hangs up the call, dialogic card calls a callback function in the gateway, which terminates the call on the PSTN side, stops the voice transport threads and sends a BYE via the CallThread to the PC client and terminates the SIP call.

Incoming Calls: For every incoming call the Dialogic thread would create a new gateway object. The gateway object in turn would create the Call Thread which stays till the end of call. Then the Dialogic thread would call the appropriate method of the created object for completing the call from sip gateway to sip client. This method would lookup for the IP address of the given extension number and place a call to that sipua client. The rest of the communication would be similar to the out going call described above.

Note that the multithreaded architecture of the library was already present. Our contribution is to implement the multithreaded gateway. Although small in terms of amount of code, the design took lots of time and thinking. We made sure that the design was as clean and extensible as possible. Another thing we decided was not changing the sip library code. We, instead, used the existing API to build the gateway. This would aid to us to easily interface our code with other parts of the Columbia's CINEMA system and any of the future additions to the sip library.

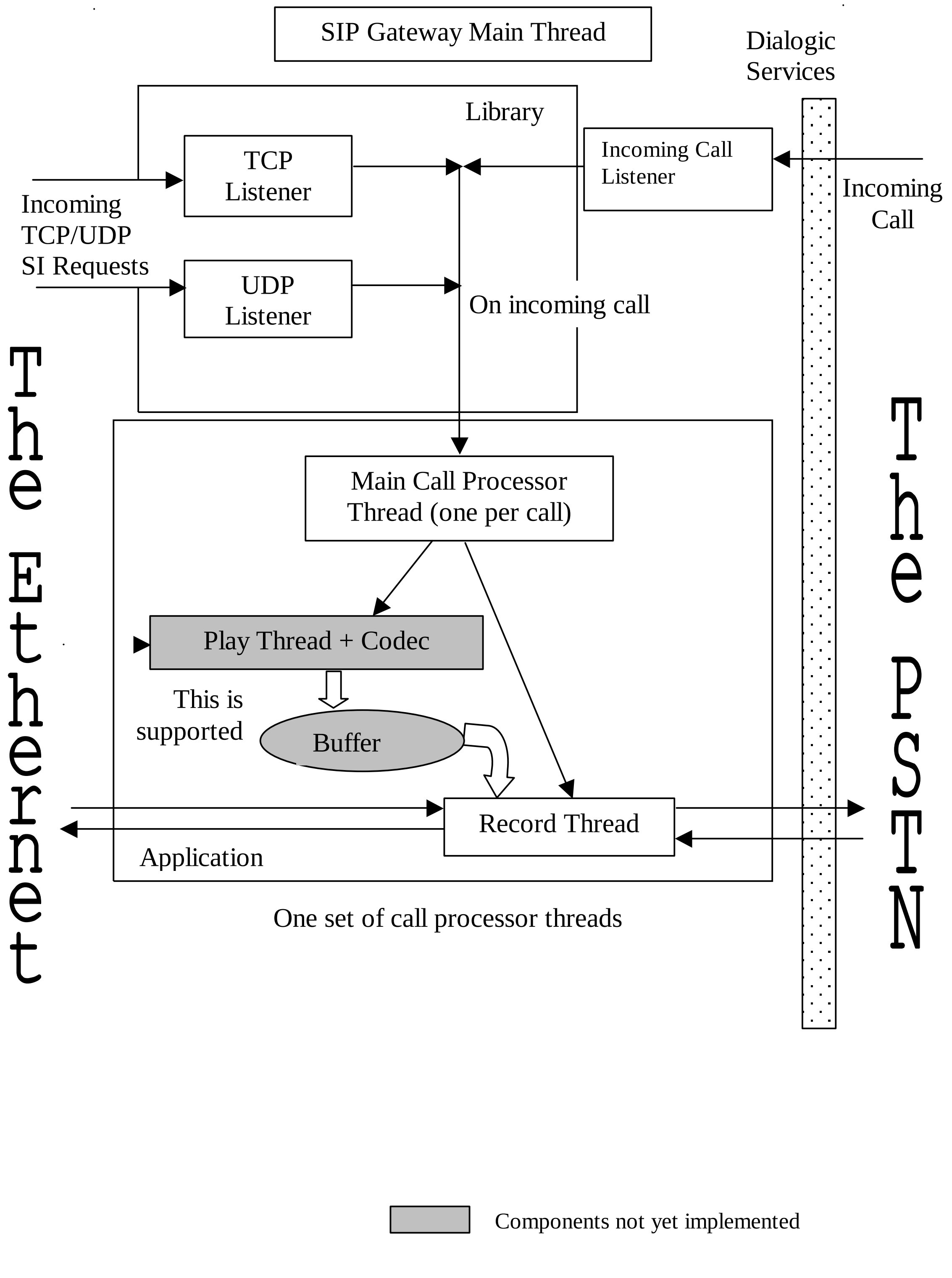


Figure 6: The Eth2Phone Gateway

Dialogic Module Implementation

The card allocates the physical channels to the calls and frees them when the calls are terminated. A global channel-array structure is used to keep track of the channel states. Since the card needs to support multiple calls at the same time, the channel-array structure needs to be thread-safe. To allow only one thread to update the state of the channels, locking mechanisms are employed. Simple mutexing is sufficient to ensure the integrity of the channel-array structure.

The dialogic module adopts a combination of synchronous and asynchronous programming models. In the synchronous programming model: all calls are blocking i.e. they return only when the function terminates normally. In an asynchronous programming model, all calls return immediately. In an asynchronous environment, the dialogic module will be registering event handlers to handle the termination of function calls. The module needs to poll the system for events, since the termination of function calls is communicated as events. When the events are received, the event handlers are called and appropriate actions are taken. We have used synchronous play, which means that the call to play returns only when the buffer has been completely played out. On the other hand, we use asynchronous record, which means that the call to record returns immediately. The card then waits for a record-termination event. A record-termination event could be something like the buffer is completely filled up or the user pressed a number on the phone. These events are specified in a special termination-events-specification structure. An event-handler that had been registered with the card before the call to record is invoked when one of the termination events is received. The handler can recognize which channel the event was received on and also what specifc event caused it to be invoked. It can then take appropriate steps to handle the specific event.

The dialogic module registers handlers for hangup events (that is indicated by zero loop current) to handle call-termination and ring events to receive incoming calls. A number is dialed through a synchronous dial-out function call. With call-analysis enabled, the dial-out function returns the status of the call. So, information like the line is busy, there is no ringback, nobody is picking up the phone can be obtained and passed on to the client.

Outgoing calls:

When the client wants to dail out a number, the card is given an ascii string of digits to dial and set up the call. First the channel is opened and set on hook to get ready to dial out the number. If the number is succesfully dialed out, the channel is placed off-hook and the call-setup completes. If either party hangs up, all I/O on the channel is immediately stopped and the channel is placed on-hook.

Incoming calls:

The dialogic module waits for a ring event on the channel. When the ring is received, the ring-handler is invoked, which picks up the call by placing the phone off-hook. If either party hangs up, all I/O on the channel is immediately stopped and the channel is placed on-hook.

The dialogic card does all the channel-state-management: before dialing out or picking up an incoming call, the module makes sure that the channel is free. Similarly, the channel is freed when a hang-up event occurs.

The gateway application can be run from user-level; however, only an administrator with root privileges can start the dialogic card (turn it on). More information about installation and turning on the card will be included in the appendix.

The following functions are provided by the hardware-independent API.

`int initialize_hardware()`

Input : None
Output : Error code (0 if Success and –1 if Failure)
Function : The telephony card module is initialized using this function. Any configuration or setting of global variables are done by this module.

`int dial_out(char *dial_out_num, CIB *cib, int  *call_status)`

Input : String of digits, pointer to Call-Information-Block (CIB) and pointer to the status of the call. The CIB structure is used by the hardware-specific module to store all the information related to the call. Each call is associated with its own CIB structure.
Output : Error code (0 if Success and –1 if Failure)
Function : This function is used to dial out a telephone number. The result of dialing is returned in the call_status parameter. If the call is setup, appropriate structures are modified to indicate that the channel that the number was dialed out on is now in use.

`int end_call(CIB *cib)`

Input : Pointer to the Call-Information-block.
Output : Error code (0 if Success and –1 if Failure)
Function : This function is used to terminate a call. Any cleanup associated with the termination of the call needs to be done by this function.

`int play(char *buf, int buf_length, CIB *cib)`

Input : Pointer to the buffer that contains the voice data to be played out, length of the buffer, and pointer to the call information block.
Output : Error code (0 if Success and –1 if Failure)
Function : This function is used to play out the data contained in the PC host buffer on the phone line.

`int record(char *buf,  int buf_length, CIB *cib)`

Input : Pointer to the buffer that the voice data will be recorded to, length of the buffer, and pointer to the call information block.
Output : Error code (0 if Success and –1 if Failure)
Function : This function is used to record the voice data from the phone line to the PC host buffer.

4. Walkie-Talkie Implementation

There were several obstacles that we faced during our implementation of the gateway. Originally, we had planned to support PC-phone calls, where two parties would be able to speak to each other as in a normal phone conversation. However, late in the quarter, we realized that the D/21H card that we had was a half-duplex (e-mail from Dialogic folks is in the appendix) card that could not simultaneously support recording from and playing to the PSTN network. We also realized that the card that we had was incapable of playing recorded voice immediately to the client, because there seemed to be no internal buffering of the recorded data between subsequent calls to the dialogic record function, and the switching time from one record call to another was substantial enough to cause a drastic degradation in the quality of the voice received on the client. In our experiments, we also tried out a rapidly alternating play and record sequence on the same channel, but the quality was unacceptable. The experiments also confirmed our doubt that the card was half-duplex. Though we had implemented a hardware-independent API and we could have substituted the Dialogic card with a Teltone card, we decided against doing it, as we were short of time, and we believed that notwithstanding the hardware-independent API, it would take us at least two weeks to become familiar with the Teltone card and implement the hardware-specific Teltone module that would implement the required API. Also, the Teltone modem routes voice through the sound-card of the server. This essentially means that only one call can be supported at a given time. Such a setup would do injustice to the multithreaded nature of our gateway. In light of the above reasons and because we wanted to demonstrate signaling and basic PC-phone voice transport before the quarter was over, we went ahead and implemented a restricted form of walkie-talkie that had the following functionality:

The client would be able to dial out a phone number and connect to a person on a phone. The person on the phone would then speak and end his/her voice recording by pressing a button on the phone. The button-press would terminate the recording and send the recorded data to the client for immediate playback. After the voice data from the gateway would have been played to the client, the client would be able to start recording on its end and terminate the recording with a number-press. The recorded data would then be sent to the gateway, which will play out the recorded voice on the phone line immediately. The back-and-forth one-way conversation would continue until either party would decide to hang up.

The gateway would also be able to receive an incoming call from the PSTN network. The functionality in this case would be the same as in the outgoing calls case except for two details: first, the person on the phone would be required to enter four digits (to identify the callee) and second, the PC client and not the person on the phone would begin the conversation in this case. So, it is always the callee who is allowed to record first. The digits entered by the caller to identify the callee form a string, which is used to lookup a table that contains the mappings between the strings and the IP addresses of the clients.

Since our client can play linear PCM (Pulse Code Modulation), we record in linear PCM (WAV format) on the gateway. For playback on the phone line, we use the linear PCM VOX format. For both playback and record, we use 8-bit encoding at 8KHz sampling rate. The playback on both the client and the phone line is of reasonable quality.

A reason why the D/21H card might have been unsuitable for our project is because the card is primarily intended for IVR (Interactive voice response phone applications) that do not involve bi-directional conversations.
To be able to implement the complete eth2phone functionality, we would need a full-duplex card meant specifically for CT (computer-telephony). Dialogic industry-grade SCSA™-architecture-based voice processing boards with resource sharing CT Bus™ for call switching and resource-sharing applications would be ideal for our CT application. An example of such a board is the **D/120JCT-LS™**. A much better card solution would be the Dialogic DM3 IPLink that is a single-board solution for voice and fax over PSTN and IP.

5. Contributions

The team as a whole played a role in the important aspect of the design of the whole system, including the choice of the platforms (with guidelines from the Professor and the TA), the choice of the programming language(s), the choice of SIP as the protocol, Columbia SIP library as the choice of SIP library. Various members also worked on tasks like Linux and Windows installation and in general, getting things to work.

Client : Susheel and Siva
Gateway: Ashish, Satyam, and Sumit.

Susheel: Designed and implemented the voice capture and playback module on the client.
Siva : Designed and implemented a SIP user agent on the client.
Ashish : Designed and implemented multithreading of the gateway sipua.
Satyam: Designed and implemented the voice transport module on the gateway.
Sumit : Designed and implemented the dialogic services on the gateway.

6. Conclusions and Future Work

The Eth2Phone team views this project as a success. While we were not able to achieve our original goal of implementing full duplex calls between a PC and the PSTN, we feel our architecture is capable of such behavior. Given more time and some more advanced hardware, the telephony gateway would be a reality. Moreover, PC-to-PC phone calls are well supported by our clients, and the walkie-talkie application we've built to utilize our half-duplex dialogic card is a testament to the adaptability of our gateway architecture. Lastly, the completion of incoming calls to a PC is a significant accomplishment.

In the future, we'd like to make full duplex calls a reality, probably by integrating available hardware (i.e. Teltone Modem) into our system. Moreover, there are some novel application areas we'd like to explore.

All members of the Eth2Phone team thoroughly enjoyed this project. Though there were some issues that we'd rather have avoided (i.e. installation problems, half duplex nature of the dialogic card), we all learned a lot. In the area of IP telephony we are all a great deal more knowledgeable. Furthermore, we all got hands-on experience on the process of team-based large-scale software development. All in all, we had a great time with this project, and hope to steer it further in the future.

Appendix A: Dialogic Card Linux Software Installation

There are two keys steps to installing the Linux software (Release 5.0 for Red Hat Linux 6.2) for the dialogic card.

Installing Linux Streams

Obtain LiS Version 2.8 from the GCOM, Inc. website at this address: *ftp://ftp.gcom.com/pub/linux/src/LiS/*
The file is in the form of a compressed file called *LiS-2.8.tgz*. Place this file in */usr/src/* and unpack it into directory */usr/src/LiS-2.8* with the command tar -xvzf LiS-2.8.tgz
Detailed instructions about downloading and installing Linux Streams can be found at:

http://www.gcom.com/LiS/Downloading.html
http://www.gcom.com/LiS/Installation.html

Choose the default answers to all questions during the installation except for the following question :
When you make STREAMS, do you want to use backward compatible constraints in the file stropts.h?
Though the default answer is 'N', choose 'Y'.

Installing the Dialogic software

You must register on support.dialogic.com before you can download the software. After registration, download the file (REL50_LINUX.tar.gz) into a temporary directory on your hard drive. Extract the rpm packages, the install script, and the documentation by unzipping and untarring the .tgz file. It is important to be root before you proceed with the installation.

Run the installscript (dlgcinstall) from the temporary directory. Choose "ALL" when asked which packages to install. You may receive some errors, but these should be ignored, because these errors are only for the packages that are not supported on the card.

You can configure the card immediately after installation (you will be prompted to proceed with the configuration setup) or at any time by running the mkcfg utility that can be found in /usr/dialogic/bin.
Before you proceed with the mkcfg utility, it is important to note the IRQ number (hardware interrupt Request number) of the card (in our case it was 9) and the base memory address of the card (in our case it was D000H). We obtained these values from Windows NT, where the card was initially installed. However, there might be ways to obtain these values in Linux too. During configuration of the card, you will be asked to enter the IRQ number and the base memory address for the card that is in your machine.

Starting and Stopping the Card

Before you run the dialogic application., you must log in as root and start (turn on) the card by running dlstart. The dlstart program downloads the dialogic system software onto the card. To stop the card, run the dlstop program. The dlstop will unload the dialogic drivers and make the card unavailable to any application. Both the
dlstop and **dlstart** programs can be found in /usr/dialogic/bin. If you get the card in some bad state, it is necessary for you to stop the card before you start the card again.

Appendix B: Messages from Dialogic Technical Support

```
From bhansali@stanford.edu Tue Nov 21 08:20:27 2000 -0800
Status: R
X-Status:
X-Keywords:
Received: from leland3.Stanford.EDU (leland3.Stanford.EDU [171.64.14.90])
        by pobox1.Stanford.EDU (8.10.1/8.10.1) with ESMTP id eALGKR321924
        for <bhansali@pobox1.stanford.edu>; Tue, 21 Nov 2000 08:20:27 -0800 (PST)
Received: from mail-dns1-nj.dialogic.com (mail-dns1-nj.dialogic.com [146.152.228.10])
        by leland3.Stanford.EDU (8.10.1/8.9.3) with ESMTP id eALGKPx24237
        for <bhansali@Stanford.EDU>; Tue, 21 Nov 2000 08:20:26 -0800 (PST)
Received: from mail4.dialogic.com ([146.152.6.40])
        by mail-dns1-nj.dialogic.com (8.9.1a+p1/8.9.1/d: dialogic.m4,v 1.3 2000/05/05 13:56:23 dmccart Exp
$) with ESMTP id QAA06961
        for <bhansali@Stanford.EDU>; Tue, 21 Nov 2000 16:24:49 GMT
Received: from exchange4nj.dialogic.com (exchange4nj.dialogic.com [146.152.3.19])
        by mail4.dialogic.com (8.9.3+Sun/8.9.3) with ESMTP id LAA12062
        for <bhansali@Stanford.EDU>; Tue, 21 Nov 2000 11:20:15 -0500 (EST)
Received: by exchange4nj.dialogic.com with Internet Mail Service (5.5.2650.21)
        id <WVMLH42L>; Tue, 21 Nov 2000 11:20:39 -0500
Message-ID: <6B0F0A1F0F44D1118BB000A024620EB508601EDE@exchange2nj.dialogic.com>
From: CustEng <CustEng@Dialogic.com>
To: "'Sumit Milap Bhansali'" <bhansali@Stanford.EDU>
Subject: DUS-66595400 - RE: D/21H on Linux
Date: Tue, 21 Nov 2000 11:20:33 -0500
MIME-Version: 1.0
X-Mailer: Internet Mail Service (5.5.2650.21)
Content-Type: text/plain;
        charset="iso-8859-1"
X-UIDL: 0d02fa0484e38711744bd52b2421df78
```

Sumit,

Sorry about that, I found out that its supported and it should be listed
under supported boards but its not in the relase notes. I will check to see
that this is corrected if not already reported.

The D/21H or other dialogic boards do not have support yet for full-duplex
operation, simultaneous play and record on the same channel is not allowed.
The reason you may be loosing data by using threads is because the use of
threading is not supported under SR 5.0 (linux), but it will be under the
next linux system release for dialogic.

Linux is not thread safe, if you have two different threads calling dialogic
functions at the same time the data might get mixed up or loss, in your case
that would be the play thread and record thread as you had mentioned. As of
now you can only use the synchronous or better yet the asynchronous
programming model without threads.

Regards,
Jeff M.

Technical Support Engineer
Dialogic, an Intel Company
mailto:custeng@dialogic.com
http://support.dialogic.com

-----Original Message-----
From: Sumit Milap Bhansali [mailto:bhansali@Stanford.EDU]
Sent: Sunday, November 19, 2000 1:08 PM
To: CustEng@Dialogic.com
Subject: Re: DUS-66595400 - RE: D/21H on Linux

Hi,
I finally got Linux to work on D/21H. D/21H and D/41H do have support
for Linux. I have a development question, though.
Can you play and record data simultaneously on a D/21H channel? I am
writing a telephony application in which I am using two threads (play that
uses dx_play() and record that uses dx_rec()), and I would like to support
two (because D/21H has two ports) bi-directional calls at the same time.
Is this possible? Do I need to use a SC-bus or CT-bus-based card?
If full-duplex operation is not possible, then I could use alternating
dx_play and dx_rec on the same channel (in the case of D/21H). But, do I
lose data between the calls to dx_rec? Is the data buffered in
the dialogic firmware buffers?
Thanks.

Sumit.

>
>
> Sumit,
>
> You are getting the WSB0010:Warning No Springware Board(s) found - check
> system configuration error amongst other because the D/21H board is not
> supported under Linux.
>
> As it is not listed in SR 5.0 Linux release notes under supported
hardware:
>
http://support.dialogic.com/documentation/unix/SR50_linux/html_files/catalog
> /1421-01.html
>
> (link may wrap around, if so type it in)
>
> Regards,
> Jeff M.
> Technical Support Engineer
> Dialogic, an Intel Company
> mailto:custeng@dialogic.com
> http://support.dialogic.com
>

References

SIP

[SIP-H.323-1] http://www.fokus.gmd.de/research/cc/glone/projects/ipt/sip.html
A concise comparison of SIP and H.323. Also features articles on why SIP is better than H.323.

[SIP-H.323-2] http://www.nuera.com/news/iptelephony_072000.cfm
A 'Telecommunications Online' article on the contention between SIP and H.323 as IP Telephony standards. The article outlines the possibility of both H.323 and SIP co-existing because of the different tastes of the Telephony industry and the Internetworking industry.

[SDP RFC 2327] http://www.cis.ohio-state.edu/htbin/rfc/rfc2327.html
The Network Working Group description of the Session Description Protocol (SDP).

[SIP RFC 2543] http://www.cis.ohio-state.edu/htbin/rfc/rfc2543.html
The Network Working Group description of the Session Initiation Protocol (SIP).

[Columbia SIP Library] http://www.cs.columbia.edu/~kns10/software/siplib/
A description of the Columbia SIP library, the architecture of various Columbia SIP products like sipd and sipua; and links to the download site.

[H.323] http://www.openh323.org/
A freeware version of the H.323 stack is available here. The aim of the site, as stated on the home page, is to offload the burden of re-inventing the H.323 stack so that developers can concentrate on writing H.323 applications.

[Internet Telephony] http://www.cs.columbia.edu/~hgs/internet/internet-telephony.html
A collection of links to Internet Telephony articles, jobs in the field, news related to IP Telephony, standards etc.

Sound

[MMSystem waveform audio] http://msdn.microsoft.com/library/psdk/multimed/wave_7jcf.htm
A guide to writing Windows sound applications using the Win32 API. The site includes a waveform audio programmer's reference and a discussion on waveform audio.

[Java Sound API] http://java.sun.com/products/java-media/sound/
The complete reference to using the Java sound API to write platform independent sound applications. A major drawback of Java sound is the delay (discussed in the body of this report).

Windows Development

[MSDN-VC++] http://msdn.microsoft.com
The online reference to writing C/C++ programs in the MS VC++ IDE. The volume is optionally available with MS-VC++ installations.

Dialogic

[Product support] http://support.dialogic.com
General information about the dialogic cards and their installation. Drivers and programs for Dialogic cards can be downloaded from here.

[Unix voice libs 1] http://support.dialogic.com/documentation/unix/voxvol1.pdf
Information on the Unix voice libraries provided by Dialogic.

[Unix voice libs 2] http://support.dialogic.com/documentation/unix/voxvol2.pdf
More information in the Unix voice libraries provided by dialogic.