A HIGH LEVEL AUDIO COMMUNICATIONS API FOR THE UNICON LANGUAGE

BY

ZIAD A. AL-SHARIF

A thesis submitted to the Graduate School
in partial fulfillment of the requirements
for the degree
Master of Science

Subject: Computer Science

New Mexico State University
Las Cruces, New Mexico
August 2005
Copyright © 2005 by Ziad A. Al-Sharif

“ A High Level Audio Communications API For the Unicon Language ,” a thesis prepared by Ziad A. Al-Sharif in partial fulfillment of the requirements for the degree, Master of Science, has been approved and accepted by the following:

Linda Lacey
Dean of the Graduate School

Clinton L. Jeffery
Chair of the Examining Committee

Date

Committee in charge:
Dr. Clinton L. Jeffery, Chair
Dr. Roger T. Hartley
Dr. Phillip L. De Leon

DEDICATION

For
My Mother who gave me life
My Wife who shares my life

ACKNOWLEDGMENTS

I would like to thank my advisor, Clinton Jeffery, for his encouragement, interest, and patience. Personally, I would like to thank him for sharing his knowledge which has enriched my study.

ABSTRACT

A HIGH LEVEL AUDIO COMMUNICATIONS API
FOR THE UNICON LANGUAGE
BY
ZIAD A. AL-SHARIF

Master of Science
New Mexico State University
Las Cruces, New Mexico, 2005
Dr. Clinton L. Jeffery, Chair

VoIP aims to make voice communication over the internet very easy and less costly, despite difficulties such as the limitations of bandwidth and the supporting technology. VoIP has numerous applications; for example it greatly enhances the potential usefulness of Collaborative Virtual Environments (CVEs). If 3D graphics make a CVE feel like a *place*, voice communication will help that place feel real and make it more useful. This thesis presents a VoIP facility developed for Unicon, a high level language that simplifies the task of writing programs, reducing their development cost, and programming time. Unicon's VoIP interface is part of an audio communications API designed to be minimal and consistent with the rest of the language. These goals keep the VM size reasonable and reduce time a programmer spends learning how to write VoIP applications. Through a set of extended and added built-in functions, Unicon now supports peer-to-peer, one-to-many, and many-to-many VoIP sessions with no need for a server. This project uses an open source cross platform library called JVOIPLIB, a C++ library that provides adequate VoIP QoS on both Linux and MS Windows.

TABLE OF CONTENTS

LIST OF TABLES ix

LIST OF FIGURES x

1 INTRODUCTION 1

1.1 Digitizing Audio 2

1.1.1 Audio Compression 3

1.1.2 Audio File Formats 3

1.2 UNIX/Linux Digital Audio Systems 5

1.2.1 Open Sound System (OSS) 5

1.2.2 Advanced Linux Sound Architecture (ALSA) 5

1.2.3 Sound Servers 6

1.3 MS Windows Digital Audio Systems 6

1.4 Sound Libraries 6

1.4.1 Libsndfile 7

1.4.2 OpenAL 7

1.4.3 Port Audio 8

1.4.4 Java Sound API 8

1.4.5 Comparison Between Sound Libraries 9

2 VOICE OVER IP 10

2.1 VoIP Protocols 11

2.1.1 RTP 11

2.1.2 SIP 12

2.2 Jori's VoIP Libraries 12

2.2.1 JThread 12
2.2.2 JRTPLIB 13
2.2.3 JVOIPLIB 13
2.2.4 Jori's Libraries VoIP Transmission Quality 14
2.2.5 Compiling Jori's Libraries 15
3 A HIGH LEVEL VOIP INTERFACE 17
3.1 Overview of the Unicon Language 17
3.2 Unicon's RTL 17
3.3 Unicon's VoIP Design Considerations 18
3.4 Adding new VoIP Built-in Functions to Unicon 18
3.5 Built-in Functions 19
4 IMPLEMENTATION OF UNICON'S AUDIO/VOIP FACILITIES . . 20
4.1 Mixing C and C++ 20
4.1.1 Accessing C from C++ Program 21
4.1.2 Accessing C++ functions from C Program 21
4.1.3 Accessing C++ Classes from C Program 21
4.2 A C++ Language API for JVOIPLIB 22
4.2.1 The Main JVOIPLIB Classes (Data Structure) 22
4.2.2 The Supporting Unicon VM Data Structure 23
4.2.3 The C++ API Function Prototypes 24
4.3 The Impact on the Size of the Unicon VM 27
4.3.1 The Impact on the Linux Unicon VM 27
4.3.2 The Impact on the Windows Unicon VM 28

4.3.3 Disabling Components from JVOIPLIB 28
4.3.4 The Impact on the Unicon VM after Optimizing JVOIPLIB 28
4.3.5 The Impact on the Build Process 30
5 USING THE NEW UNICON VOIP FUNCTIONS 31
5.1 Unicon's VoIP Functions 31
5.1.1 open() 32
5.1.2 close() 33
5.1.3 VAttrib() 34
5.2 Unicon's Audio Functions 35
5.2.1 PlayAudio() 36
5.2.2 StopAudio() 36
6 AN APPLICATION EXAMPLE: THE NMSU'S CVE 37
6.1 Voice Implementations in the CVEs 38
6.2 Voice Modes in the NMSU's CVE 39
6.2.1 Disabled "No Voice" 40
6.2.2 Local "Current Room" 40
6.2.3 Private "Virtual Cell Phone" 40
6.2.4 Group "Future Work" 43
6.3 Voice Implementation in the NMSU's CVE 43
7 CONCLUSION 46
REFERENCES 47

LIST OF TABLES

2.1 JVOIPLIB 14

4.1 JVOIPSessionParams Variables 23

4.2 JVOIPLIB Optimized 29

LIST OF FIGURES

1.1 Sound Libraries 9

4.1 The Changes in JVOIPLIB after Disabling Components 29

4.2 The impact on the Unicon VM after adding VoIP 30

6.1 The Client Interface in the NMSU's CVE 38

6.2 The GUI interface for the Local "Room-Based" Voice Connection Mode 41

6.3 The GUI interface for the Virtual Cell Phone 42

6.4 The Voice Implementation in 3D Room-Based Environment . . . 44

6.5 The Voice Implementation in the Virtual Cell Phone 45

CHAPTER 1
INTRODUCTION

People rely on telephones for distant voice communications, even though the telephone interface has not changed much since its invention a long time ago. This consistency in the telephone interface may relate to the nature of telephone networks that makes continuous connection hard or even impossible. However, conventional telephone networks are gradually being replaced with IP networks to carry voice data which is called Voice over IP (VoIP). This project takes this new technology and advances it by adding VoIP facilities to a programming language. Direct language support makes building applications with VoIP facilities much easier and more accessible to ordinary programmers. The capabilities are demonstrated by building VoIP into a Collaborative Virtual Environment (CVE) for the computer science department at New Mexico State University.

The CVE of the computer science department at New Mexico State University is an educational CVE. It integrates a 3D virtual view of the computer science department with 2D collaborative tools for working on various software engineering and development tasks. This project is funded by the NSF. It supports distance education and aims to improve communications inside the department as well as in between the department and the CS community around the state of New Mexico. This CVE is written in Unicon. Unicon is a high level goal directed language now being extended to directly support CVEs.

The project described in this thesis uses VoIP functions provided by an open source library called JVOIPLIB [1] [2]. Since the Unicon VM is written in C, a mechanism for calling a C++ library from C code was needed. This thesis discusses the requirements and the design goals before proceeding with the implementation that starts with how to mix C and C++ and then describes the data structures and the API functions that are needed to make the VoIP simple for Unicon programmers. The Unicon runtime system uses these data structures internally in order to establish and maintain VoIP sessions. Two Unicon functions have been extended and another one has been added to the Unicon built-in functions. Those extended and added functions help Unicon programmers write programs that can use VoIP sessions with little effort.

Augmenting Unicon with audio and VoIP facilities requires C open source cross platform libraries. The first chapter of this thesis gives a background discussion about the digital audio systems before it describes our experience with audio libraries such as Libsndfile [3], OpenAL [4], and Port Audio [5]. Those libraries are written in C and can digitize, write, and read sounds. The second chapter describes an interesting VoIP library that met the requirments for this project.

Chapter three presents the language design considerations that shaped the new Unicon VoIP facilities. Chapter four outlines the implementation of Unicon's VoIP facilities on top of JVOIPLIB. It starts by discussing the issue of mixing C and C++ programs, and then describes the details of the API functions and the mechanism used to add new built-in functions into the Unicon run-time system and the impact on the Unicon VM.

Chapter five describes the syntax of the VoIP and audio functions available to Unicon programmers; this description is supported with small examples that outline the use the new facilities. Chapter six talks about NMSU's CVE, which is a real application that uses the audio and VoIP functions.

The rest of this chapter gives an overview of digital audio and audio compression techniques, then discusses audio formats on popular operating systems such as UNIX/Linux and MS Windows. The chapter concludes with a description of experiences with some of the open source cross platform audio libraries.

1.1 Digitizing Audio

In digital audio, the analog audio signals are sampled at regular intervals by an analog-to-digital converter in a process called *digitizing*. Digitizing converts the analog sound into a stream of numbers; each number represents a sample measurement of the voltage coming from the sound source. The voltage is measured many times per second. A digitized sound signal can be in two forms: linear or nonlinear encoding. Linear encoding is often referred to as Pulse Code Modulation (PCM). PCM is a direct representation to the original sound signal, which is called the waveform encoding scheme. PCM is supported by most PC soundcards and operating systems. Nonlinear encoding maps the sound signals to the stored values using a nonlinear function. u-law (pronounced mew-law) is a common nonlinear encoding scheme that provides a more compressed version of the sound data.

Digitizing can produce different sound qualities depending on two factors: *sampling rate* and *sampling size*. Sampling rate is the number of samples recorded per second. The most common sample rates used in multimedia applications are: 8000 Hz, 11025 Hz, 22050 Hz, 44100 Hz, and 48000 Hz. A sample rate of 44.1 kHz means that the sound is sampled, or measured, 44,100 times per second. Digitizing with a high sample rate produces digital sound very close to the original sound, but it requires more disk space to store and more bandwidth to transmit over the network. The sample size describes the accuracy of the voltage measurement which is usually 8 or 16 bits. For an 8-bit sample size; a number between 0 and 255 is used to describe 0%-100% voltage. For a 16-bit sample size; a number between 0 and 65,535 is used to describe 0%-100% voltage. Therefore, using a

higher sample size gives a closer approximation to the original sound which also increases the size of the digitized audio [6].

1.1.1 Audio Compression

The simplest way to process audio signals on a computer is to get the voice data in the PCM format, which allows very simple and direct conversion from analog to digital audio. However, PCM produces gigantic files; ten minutes of sound can produce 10 MB in PCM format. Audio compression can alleviate such problems, but standard data compression methods that reduce the number of bits used to represent data, do not work well on audio. So, there is a necessity for dedicated audio compression techniques in a standard that supports compatibility between audio software.

The limits of network bandwidth and hard disk space have motivated the development of numerous compressed audio formats. In general, there are two basic categories of audio compression: lossless and lossy. In lossless compression algorithms, the extracted audio from a compressed file will be identical to the original file before the compression, but lossless compression yields only moderate compression ratios.

Lossy compression algorithms offer much higher compression ratios by discarding some of the original data; data that is unnecessary and redundant such as sounds that most people can not hear. Lossy compression reduces the amount of data, not just the number of bits used to represent that data. The reproduced audio is at a lower quality than it was before compression, but in many cases the difference is difficult to perceive . DPCM and ADPCM are common lossy audio compression algorithms that are simple modifications to the PCM format [6].

DPCM stands for Differential Pulse Code Modulation. It is a direct modification to the common PCM encoding scheme. It stores only the difference between consecutive samples. DPCM predicts the value for the next sample and records the difference between this value and the value that really occurred. This difference is usually small, so it can be encoded with fewer bits than the PCM sample. ADPCM stands for Adaptive Differential Pulse Code Modulation. It is similar to DPCM except that the number of bits used to store the difference between samples is varied depending on the complexity of the signal, which results in increased compression rate with an acceptable quality loss.

1.1.2 Audio File Formats

Knowing the audio data format is very important, since it determines how to interpret a series of bytes of sampled audio data, such as samples that have been read from a sound file, or captured from a microphone. Audio data format provides information such as the encoding technique, the number of channels such

as mono or stereo, the sample rate, sample size, frame rate, frame size, and byte order such as big-endian or little-endian. At the time of digitizing or playing back, the data format of the sound must be specified.

Digital audio files usually consist of two main parts: a header, and data. The header is used to store information about the audio data such as the sampling size, sampling rate, compression type, etc. The audio file type refers to the structure of the audio data within that file. However, it is common for the same audio format to be used in more than one file type. For example, the PCM format is used in both WAV and AIFF files. The following list describes some of the common file types and their formats:

- *AIFF* stands for Audio Interchange File Format. It was developed originally for use on Macintosh computers. It uses the PCM audio format. Its common file extension is .aif or .aiff.

- *AU* or u-law is the Sun audio file format. It can be played on a wide number of platforms, such as Mac, and UNIX systems; especially Sun and NeXT. u-law (pronounced "mew-law") is a technique similar to ADPCM. Its file extension is .au.

- *WAV* is a large, uncompressed PCM based, high-quality, standard audio file. It was developed by Microsoft for use on Intel-based computers. Professional digital audio recording and editing systems use the WAV files as their standard. Its file usually has the extension .wav.

- *MIDI* stands for Musical Instrument Digital Interface. It was designed so that electronic instruments of all kinds could exchange musical information. Unlike the other sound formats, MIDI does not capture and store actual sounds. Instead, it translates the sounds into a set of steps that can be recreated by any sound synthesis device that understands MIDI. Because it does not contain actual sounds, the file size of MIDI files is extremely small.

- *MPEG* stands for Moving Picture Expert Group. It is a family of standards for compressed audio and video. MP3 is an acronym for MPEG layer 3. MP3 compression focuses on removing the redundant and irrelevant parts of a sound signal by predicting sound signals that the human ear can not recognize. It creates very small files suitable for streaming or downloading over the Internet. The audio file size can be reduced by a factor of 12, without much damage to the sound quality of the original file. Because of the small size and good quality of MP3 files, they became a popular way to store music files on both computers and portable devices and a good choice to stream audio over the internet. Its common audio file extension is .mp3. In PCM encoding, the frame rate is equal to the sample rate. In MP3, each

frame encapsulates a whole series of samples which makes the frame rate slower than the sample rate. The sample rate and sample size refer to the PCM data that the encoded sound is converted into before being output to the sound card. Similarly MP3 files are created by digitizing into PCM technique and then converting PCM into MP3 format.

- *Ogg Vorbis* is a weird name for a fully open audio compression format that produces audio quality and size very competitive to the famous MP3 formats. Its idea came after the copyright holders of the MP3 codec closed the source of their product and began demanding licensing fees. Like MP3, Ogg Vorbis is a lossy compression scheme. Its file extension is .ogg [7].

- *RM* is audio file format for streaming audio that is used by RealNetworks. The purpose of these files is to do live broadcasting and other real-time internet audio applications. Streaming means that the user does not have to download everything in the file before they can listen to it, it can be played and downloaded at the same time. The Real Audio player is the main player for this kind of audio. Its common file extension is .rm, .ra, or .ram

1.2 UNIX/Linux Digital Audio Systems

Most popular operating systems support sound digitizing and playback. In UNIX/Linux, there are two different standard sound systems: the Open Sound System (OSS), and the Advanced Linux Sound Architecture (ALSA). In the next sections we will take a look at some of their features.

1.2.1 Open Sound System (OSS)

Open Sound System (OSS) is a unified digital audio architecture for UNIX. It is a set of sound device drivers that provide a uniform API across all the major UNIX architectures. With OSS, applications on UNIX can provide the same audio capabilities as those found on popular PC operating systems. OSS/Linux is a commercial implementation of the Linux sound drivers that are packaged with the Linux kernel 2.4. In the Linux kernel 2.6, OSS drivers have been replaced with ALSA. However, OSS is available not only for Linux but also for BSD OSes and other UNIX platforms such as Solaris[8].

1.2.2 Advanced Linux Sound Architecture (ALSA)

Advanced Linux Sound Architecture (ALSA) is the new Linux sound hardware abstraction layer that replaces OSS. It is free and open source. ALSA drivers

are included in the Linux kernel 2.6. ALSA contains a user space library to help with developing new programs and an API compatibility layer for the many programs still using the older OSS drivers[8].

1.2.3 Sound Servers

Sound servers are an additional layer of software between the user and the hardware that allows applications to play multiple sounds at the same time on a single sound card without any need for a sound card that natively supports that. So, applications can share the sound hardware, because sound servers support multiple channels even if the kernel sound device supports only one. Sound servers multiplex and stream this output to the */dev/dsp* device. Moreover, some Linux sound servers such as ESD and aRTs are built on a client-server model that enables sound to be played remotely and transparently on a network.

ESD stands for Enlightenment Sound Daemon. It was originally developed for Enlightenment and is now part of the GNOME Project. ESoundD supports full duplex and network transparency, and it is especially suited for sound effects and long unsynchronized music[8].

aRTs stands for analog RealTime synthesizer. It is the KDE's sound server. Various commentaries suggest that aRTs has a better sound quality than ESD due to better sound processing routines but higher latency due to their complexity. aRTs supports full duplex even though it has been reported to be a bit buggy in this area. It also supports network transparency and works on BSD operating systems[8].

1.3 MS Windows Digital Audio Systems

Microsoft Windows supports many API components for audio and Music such as DirectSound, DirectSound3D, and DirectMusic. Recently, those API's were unified under Microsoft's DirectX. DirectSound is limited to the WAV file format which was originally developed by Microsoft. One of the great advantages of DirectSound is its support for live voice. DirectMusic is an audio programming API designed to support musicians. It supports multiple audio file formats such as WAV and MP3 [9].

1.4 Sound Libraries

This section examines C /C++ open source cross platform audio libraries that can digitize, write, and read sounds. Three different audio libraries are examined: Libsndfile, OpenAL, and Port Audio. These libraries have a lot in common; they focus on the main sound functions like digitizing and playing back.

The main difference between them is in their portability, and the sound formats that they can read and write. The libraries are evaluated in order to identify the simplest, and the most comprehensive one, that has usable functions with a reasonable size, for use in Unicon.

1.4.1 Libsndfile

Libsndfile is an open source C audio library. It was written for Linux systems but it can be compiled on any UNIX-like system including Macintosh as well as on MS Windows using the Microsoft Visual C/C++ compiler (MSVC). The strong point in this library is the different kinds of sound formats that it can read and write. It can currently read and write 8, 16, 24 and 32-bit PCM files as well as 32 and 64-bit floating point WAV files and a number of compressed formats through one standard interface . However, Libsndfile does not have any support for MP3 or similar formats. Libsndfile is also designed to be compiled and to run correctly on little-endian processors, such as Intel and DEC/Compaq Alpha, as well as big-endian processors such as Motorola 68k, Power PC, MIPS and Sparc.

Libsndfile is able to convert sound data from one format into another. For example, in a sound playback program, the library caller typically wants the sound data in 16-bit short integers to dump into a sound card even though the data in the file may be 32-bit floating point numbers such as the Microsoft's WAVE_FORMAT_IEEE_FLOAT format. Another example would be someone doing speech recognition research who has recorded some speech as a 16-bit WAV file but wants to process it as double precision floating point numbers.

Libsndfile is supported and updated; the last stable version is libsndfile-1.0.11. The library and much other important information are in the library website [3].

1.4.2 OpenAL

The Open Audio Library (OpenAL) is a software interface to audio hardware. It is an open source C audio library. OpenAL designed to provide a cross-platform open source solution for programming 2D and 3D audio. It has been released under the GNU Lesser General Public License (LGPL). It supports Microsoft Windows, Mac OS, Solaris, and on Linux it supports both Open Sound System (OSS) and the Advanced Linux Sound Architecture (ALSA).

OpenAL focuses on rendering audio into an output buffer. Many of the design considerations for OpenAL have been driven from the similar considerations for the visual effects in OpenGL. OpenGL programmers have an advantage in using OpenAL due to the similarities between OpenGL and OpenAL APIs. The integration between OpenAL and OpenGL makes OpenAL the best choice for

real-time sound rendering that may associate with graphic applications such as the sound effects.

OpenAL APIs focus only on PCM audio (Pulse Code Modulation). Programmers must rely on other mechanisms to obtain audio (e.g. voice) input or generate music. OpenAL can support MP3 with assistance of other libraries such as the Simple DirectMedia Layer (SDL-1.2) [10] and SDL MPEG Player Library (SMPEG) [11]. OpenAL supports playing the traditional WAV format. OpenAL is supported, documented, and there are many projects using it [4].

1.4.3 Port Audio

Port Audio is a simple, portable, and open source audio library. Using Port Audio is very easy and with small effort programmers can write audio programs in C that can be compiled and run in different platforms like Windows, Mac, and UNIX. Port Audio runs on Open Sound System (OSS); support for Advanced Linux Sound Architecture (ALSA) is going to be released in the next version. PortAudio was designed to promote the exchange of audio synthesis software between developers on different platforms. PortAudio has no support for reading or writing formatted audio files and they recommend using libsndfile for formatted audio files I/O and conversions between different audio formats. It was recently selected as the audio component of a larger PortMusic project that includes MIDI and sound file support.

The most interesting thing in PortAudio is its easiness since it provides very simple API functions for audio tasks like recording and playing back. It is well documented, allowing programmers to understand the low level details for every function. Building a simple audio program using Port Audio requires a smaller number of C lines compared with other libraries like OpenAL or even Libsndfile. The last stable version is V19. The library, its status, and much more information are in the library website [5].

1.4.4 Java Sound API

For comparison purposes, it is worth mentioning the Java Sound API. Java Sound API is a low-level API for controlling the input and output of sound media, including both audio and Musical Instrument Digital Interface (MIDI). It is the lowest level of sound support on the Java platform. This API is supported by an efficient sound engine which guarantees high-quality audio mixing and MIDI synthesis capabilities.

The Java Sound API supports mechanisms for installing, accessing, and manipulating system resources such as audio mixers, MIDI synthesizers, file readers and writers, and sound format converters. The API supports audio file formats such as AIFF, AU and WAV with 8-bit and 16-bit audio data, in mono and stereo,

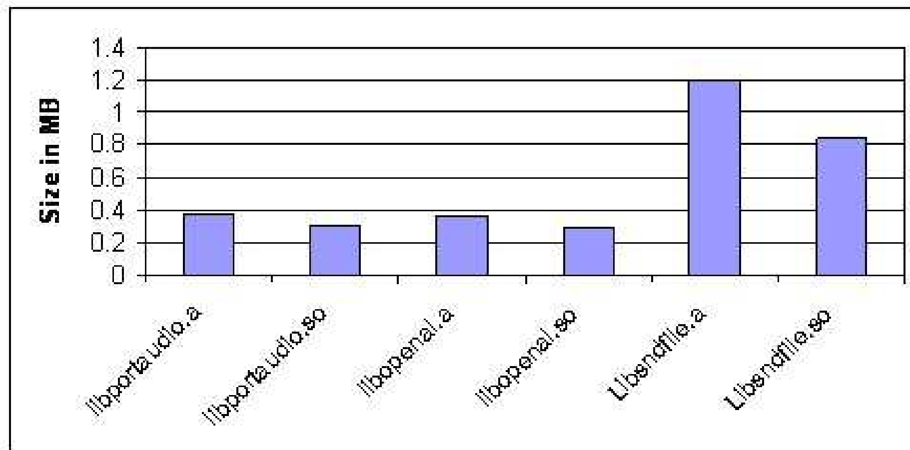


Figure 1.1: Sound Libraries

with sample rates from 8 kHz to 48 kHz. The default implementation of the Java Sound API does not support compressed formats such as MP3.

There are some other Java platform APIs that have sound related elements such as the Java Media Frame Work (JMF) which is a unified architecture, messaging protocol, and programming interface for capturing and playing back time based media. Other Java APIs with sound aspects include the Java telephony API and the Java Speech API. The Java telephony API is used for the integration of computers with telephones, allowing for both first-party and third-party calling. The Java Speech API is used for speech recognition and speech synthesis capabilities[12].

1.4.5 Comparison Between Sound Libraries

The previous three libraries all met our requirements that they are open source and cross platform to at least UNIX/Linux and MS Windows. Port Audio provides low level primitive functionalities for reading and writing audio. Libsndfile is useful for playing many different sound file formats and converting between them. OpenAL is a great library useful for game environments especially with OpenGL; it also benefits from the presence of other libraries such as Ogg Vorbis and SMPEG to play .ogg and .mp3 respectively. OpenAL is the best of the three libraries for general purpose audio functions as well as 3D audio and sound effects within a 3D environment. OpenAL was the best choice for Unicon because of its usefulness, generality and its reasonable size. The size of the adapted library is important since it will affect the size of the Unicon VM. Figure 1.1 shows the sizes of the different Sound libraries compiled with gcc-3.3.4.

CHAPTER 2
VOICE OVER IP

VoIP stands for Voice over Internet Protocol. It is a mechanism to make a voice call similar to a regular phone call except that it uses an internet connection where the voice is transmitted as data packets. The voice is converted from analog into digital form and later while receiving the digital voice it gets converted back into the original analog form. A real-time protocol (RTP) helps make sure that packets are delivered in the right order with the necessary speed. This mechanism allows calls to be from PC-to-PC, PC-to-phone, phone-to-PC, or even phone-to-phone over the computer network with the right equipment. Most of the time PC-to-PC voice calls do not cost anything other than the internet connection which gives VoIP an advantage over the other forms phone calls. More over, PC-to-phone or phone-to-PC providers give a lower cost rate than regular phone calls. PC-to-PC call requirements are a high speed internet connection such as DSL or a local area network (LAN) and at least two computers with soundcards, speakers, and a microphone, managed by the right software. If those requirements are available, a call can be initiated with any person in the world. It also can be initiated with many people at the same time.

VoIP is a rapidly growing segment of the telecommunications market. For illustration purposes, this chapter describes representative applications related to collaborative environments that are using VoIP.

- The first one is *3Com IP conferencing module*; it is one of the commercial products that support advanced voice, video, and data collaboration such as co-editing documents in real time on a shared desktop, instant messaging and file sharing [13]. The basic features of the program costs about $50,000 [14].

- The second one is *ICQ* or *I Seek You*. It is a freeware instant messenger that allows people to get in touch with each other. It includes voice, message board, data conferencing, and internet games. It also supports various popular internet applications such as file sharing, multiple-user mode for conferences, PC-to-PC talk and e-mail messages. The ICQ Phone allows free phone calls to other ICQ members. The instant messaging enables sending messages that immediately pop up on an online contact's screen. If the user is away from his personal computer he can still chat with friends, even if the ICQ client is not installed, by using the web based ICQ2Go that works from any computer [15].

- The third one is *Arsenal*, it is a client-server real time collaboration and conferencing project built in Java with plug-in architecture for easy ex-

tensibility. The goal of the Arsenal project is to provide a safe real time collaborative tool. Arsenal is written using client-server architecture; it is not peer-to-peer. It has a multi-threaded networking and connection server that can allow clients and servers to communicate in a safe manner using 2 non-secure protocols: socket (TCP/IP) and http. It supports group collaboration and communication, instant messaging, files sharing, group chatting, persistent sessions, shared web browser, whiteboard, and VoIP via the Session Initiation Protocol (SIP). It is an open source cross platform project released under the LGPL [16].

2.1 VoIP Protocols

Transmitting packets containing voice data requires that data arrive in the same order as it was transmitted. IP networks provide service through the TCP or the UDP protocols. TCP seems to be a good protocol for transmitting voice data; it provides reliable ordered delivery of data. TCP sets up the connection, exchanges the data, and then closes down the connection, similar to traditional phone calls. However, TCP retransmits any corrupted or lost packets which add extra delay to the communication. It is better to have occasional lost or corrupted packets than to have a large amount of delay, especially when the size of the packets is small. In contrast to data communication, an occasional error will not seriously disturb voice communication as long as it does not affect a large portion of the voice data. The other choice is UDP which has the advantage of speed and simplicity. However, UDP does not ensure ordered delivery of data which is required during voice communication [2].

So, neither TCP nor UDP protocols are ideal for transmitting voice data over an IP network. There is a necessity for an alternative protocol dedicated for voice data. Two different protocols dedicated to VoIP are RTP and SIP.

2.1.1 RTP

Real-time Transport Protocol (RTP) was developed by the Audio/Video transport working group of the Internet Engineering Task Force (IETF). The first version of RTP was completed in January 1996 [17]. It provides an end-to-end network transport function that is suitable for applications transmitting real-time data such as audio, or video, over multicast or unicast network services. It can be used for media-on-demand as well as interactive services such as internet telephony. RTP does not address resource reservation and does not guarantee quality-of-service for real-time services. The transmitted data is augmented with an RTP control protocol (RTCP). RTCP provides a minimal control and iden-

tification functionality. RTP and RTCP are designed to be independent of the underlying transport and network layers[18] [19].

2.1.2 SIP

Session Initiation Protocol (SIP) is an application-layer control protocol that can be used to establish, maintain, and terminate calls between two or more endpoints. It was published by the IETF as RFC 2543 in March 1999. It is currently the leading signaling protocol for Voice over IP since it was designed to address the functions of signaling and session management within a packet telephony network. SIP supports five different functionalities of multimedia communication: user location, user capabilities, user availability, call setup and call handling. It was designed as a client-server protocol architecture using a text based message formats that is similar to HTTP[18].

2.2 Jori's VoIP Libraries

JVOIPLIB is a comprehensive open source cross platform C++ VoIP library.The Unicon VM is written in C. It would be easier to use a library written in C if one met our project requirements. However, C and C++ are link-compatible languages, making JVOIPLIB a reasonable library for the Unicon VM. JVOIPLIB consists of three sub-libraries JVOIPLIB, JThread, and JRTPLIB; each one is necessary for a specific purpose.

The goal of the project was to support the Unicon run-time system with audio and VoIP facilities. Building the needed libraries from scratch was not a viable option. The project started with a search for an open source cross platform C audio library to augment the Unicon run-time system with sound facilities and VoIP.

Jori's libraries are three C++ open source cross platform object oriented libraries supporting VoIP with an adequate quality of service. They are written originally by Jori Liesenborg. JVOIPLIB is the VoIP library that uses the RTP protocol from JRTPLIB library. Both of the libraries, JVOIPLIB and JRTPLIB, are supported by a thread library called JThread. Jori's libraries were selected for their quality of service and reasonable compatibility with the Unicon run-time system.

2.2.1 JThread

JThread is not a Java Thread. It is Jori Liesenborg's Thread library that provides a class-based abstraction of threads and mutexes. JThread makes the use of threads easy on different platforms like Windows and UNIX/Linux from one

interface. On a UNIX-like platform, the pthread library is the underlying thread mechanism. On Microsoft Windows, the native Win32 threads are used. JThread classes are simple wrappers around existing thread implementations. Using these wrapper classes applications can use threads, without having to worry about which platform the program will run on. JThread is a stand alone library that is used by JRTPLIB and JVOIPLIB. The last stable version is JThread 1.1.1 [2].

2.2.2 JRTPLIB

JRTPLIB is a Real-time Transport Protocol (RTP) library based on RFC 3550. JRTPLIB uses UDP over both IPv4 and IPv6. RTP makes it easy to send and receive real time packets over the IP network. The JRTPLIB library makes it possible for the user to send and receive voice data with good quality of service. The user only needs to provide the library with the data packets to be sent, and the library will give the user access to incoming data [2].

JRTPLIB is an essential component of JVOIPLIB and it can be used in other RTP-based applications. It is still being refined by Jori and others who are interested. The last stable version is JRTPLIB 3.1.0 which was released in October 2004.

2.2.3 JVOIPLIB

JVOIPLIB stands for Jori's VoIP Library. It uses the previous two libraries, JRTPLIB and JThread. It works on both MS Windows and UNIX/Linux. The library makes the creation and managing of VoIP sessions easy by providing a class-based interface with operations such as create a session, destroy a session and set session attributes. In order to reduce the size of the transmitted sound, JVOIPLIB provides many compression techniques such as Raw PCM, Mu-law, DPCM, GSM, and LPC; some of these compression methods require a special range of sampling rates. The main job of this library is to digitize the input sound according to attributes such as sampling-rate, sampling-encoding, and sampling-interval, put them in packets, compress these packets using one of the compression techniques, and then deliver them to the RTP protocol which sends them to the destinations. At the destination side, it combines the received packets, decompresses them, and plays them back. The RTP protocol puts the received packets in the correct order, which helps the playback to be fairly good [2]. JVOIPLIB has the following features:

- C++ Object Oriented VoIP session creation and destruction.
- Highly configurable sessions: sampling rate, sample interval, compression type, etc. These features can also be changed during a session.

Item	Size
jvoiplib.lib *(win32- MSVC)*	2.82 MB
libjvoip.a *(win32- gcc2.95-mingw32)*	10.2 MB
libjvoip.a *(Linux gcc-3.3.4)*	2.9 MB
libjvoip.so.1.3.0 *(Linux gcc-3.3.4)*	2.3 MB

Table 2.1: JVOIPLIB

- Openness and extensibility.
- Supports for 3D effects.
- Soundcard input and output
- Different compression techniques such as DPCM compression, u-law encoding, GSM (06.10) 13 kbps compression, LPC 5.4 kbps compression.
- Using an RTP (Real-time Transport Protocol) for transmitting the sound data.
- It has its own elementary voice mixer

Unfortunately, JVOIPLIB is large enough to be a problem; its sizes on both Linux and MS Windows are shown in Table 2.1.

2.2.4 Jori's Libraries VoIP Transmission Quality

In real phone conversations, usually only one person speaks at a time. This presents an opportunity to save bandwidth since silent packets do not need to be sent. However, in order to avoid sending packets that have silence in them, a mechanism to distinguish between packets is needed. One way to do that is to calculate the strength of the voice signal contained in a voice packet. Packets that do not have sufficient strength could be considered to be silent packets and dropped. This technique is used by Jori and proved to be simple and effective; it checks the packets and uses a threshold value to decide whether the voice packet is silent or not. This technique could have a side effect: if there are no voice packets at the destination, it has no sound at all, not even a noise in the background. This could make the person at the destination think that they lost the connection.

The quality of service is affected by many factors such as the sampling rate, the sampling size and the bandwidth. JVOIPLIB proves that sampling rate of 8000 Hz and 8-bit sample size are sufficient to provide a telephone quality communication if a bandwidth of 64 kbps is available with a delay no more that

200 ms. For a LAN network, there is no problem achieving this rate, but for dial-up connection this rate is not possible. However, even the LAN may get heavily loaded by other traffic. For a WAN, achieving a telephone quality connection depends on the connection speed and how much load it carries. The RTP does not guarantee voice packet delivery to the destination, which encourages making the voice packets small enough that lost or corrupted packets will not disturb the conversation[2].

2.2.5 Compiling Jori's Libraries

Jori's libraries are easy to compile and to use on both GNU/Linux and MS Windows. In doing this project, we gained experience with them that may be useful to other developers.

On Linux, JVOIPLIB uses the OSS System. It opens the */dev/dsp* audio device. According to the OSS specifications, this device can not be opened twice. So, before using JVOIPLIB, it must ensure that no other application is using this device. In Redhat Linux 9, aRTs in KDE is using this device for background sounds; the aRTs must be disabled before using JVOIPLIB. However, GNOME does not use this device. In Linux systems that have ALSA on them, ALSA provides OSS emulation; under ALSA, there is no problem using the audio device. Any program using JVOIPLIB must be linked to libjvoip.a, libjthread.a, and libjrtp.a besides the Linux thread library *-lpthread*.

On MS Windows, JVOIPLIB was tested by Jori on Windows 2000. For the project it was tested it on Windows XP Home and Professional Editions. Jori built the libraries using Microsoft Visual Studio 6.0; the libraries are jvoiplib.lib, jthreadlib.lib, and jrtplib.lib. It is necessary to have the Microsoft Visual C version 6 updated with the Service Pack 6 in order to compile and build those three libraries from the source code. Without this service pack, compilation errors will show up. This service pack update was not necessary in the previous versions of the library. Two additional Windows libraries, ws2_32.lib and winmm.lib, are needed in order to use JVOIPLIB on Windows.

The Unicon source code on MS Windows can be compiled using both MSVC and mingw32 or Cygwin Windows's distributions of gcc. For this project, we ported Jori's libraries to be compiled using gcc-2.95-mingw32 which is the main compiler used to build Unicon on Windows.

The JRTPLIB library uses Windows Sockets. Any Windows console application using this library must call two functions, one to start up the socket subsystem before using the library, and the other is to shut it down before ending the program. If the program is already using windows sockets, these calls must be coordinated among all uses of sockets. The two functions are as follows:

```c
/* this is just for starting up windows sockets */
int StartupWinSocket(void)
{
    WSADATA wsa_data;
    if (WSAStartup(MAKEWORD(1,1),&wsa_data)!=0) return 0;
    return 1;
}

/* this is just to clean up the windows sockets */
int CleanupWinSocket(void)
{
    if (WSACleanup()==SOCKET_ERROR) return 0;
    return 1;
}
```

CHAPTER 3
A HIGH LEVEL VOIP INTERFACE

VoIP is one of the new revolutions in the communication area. It provides the phone calls with less cost and better management such as the caller ID, call waiting,and call forwarding. Such features made many phone companies to switch to VoIP. Since VoIP operates on a computer network, it is reasonable for computer software and applications to be supported by VoIP; especially collaborative software applications. However, programming a VoIP application in a common language such as C/C++ or even Java is time consuming and requires a lot of low-level details that prevent programmers from building such applications unless they have the back ground knowledge and the time.

All of this was a motivation behind formalizing a high level VoIP Interface supported by a high level application programming language that helps the programmers build collaborative applications supported with VoIP facilities in less effort and less time.

3.1 Overview of the Unicon Language

Unicon is the unified extended dialect of Icon. It is an open source, cross platform, high level, goal directed, object oriented, general purpose application language. It descends from Icon and the SNOBOL family of languages that emphasize the ease of programming. Unicon's direct ancestor is Icon. Icon relates to Unicon as C relates to C++. Icon is an imperative, procedural language with a flavor of logic programming supported by the idea of failure or success, and backtracking. Its emphasis is on string manipulation structured by a goal-directed expression-evaluation mechanism. Icon programmers do not have to worry about pointers and the garbage collector manages space as necessary[20] [21].

Unicon modernizes the Icon language through numerous features and facilities that are necessary for modern applications such as objects, networking, accessing file systems, execution monitoring and visualization, and an ODBC database interface to SQL database servers. This modernization promotes the Icon style of goal-directed very high level language. Unicon programs have a high level of portability; programmers can take a program written for UNIX platform and run it on MS Windows with almost no modifications.

3.2 Unicon's RTL

RTL stands for Run-Time Implementation Language. It was designed to support the run-time system for the Icon language, the ancestor of Unicon. It is

used to define operators, built-in functions, and keywords for Icon and Unicon. The design of RTL was motivated by the needs of the optimizing compiler for Icon, Iconc. RTL builds a database of information about the Icon operations that is used by the optimizing compiler; since Iconc needs information about the run-time routines in order to generate efficient code for Icon programs.

RTL consists of two sub-languages, first an interface and a type specification language, and second, a slightly extended version of C. The RTL interface defines what an operation will look like in the Icon language, specifies the type checking and conversion needed for arguments to the operation, and presents the over all structure of the operation's implementation. The extended C language includes operations for manipulating, constructing, and returning Icon values. It is imbedded within certain constructs of the interface language and provides the low-level of the implementation [22].

3.3 Unicon's VoIP Design Considerations

The Unicon language style is inherited from its ancestor Icon, a goal-directed very high level language. Minimizing the interface with a high level of abstraction reduces time a programmer spends learning it. JVOIPLIB provides a C++ class based VoIP interface with low level details that Unicon programmers do not care about. A set of C++ API functions hide the C++ class based interface and allow access from the Unicon runtime system to the C++ library. Keeping the VM as small as possible and reducing the complexity of the run-time system were the main considerations of the VoIP interface design. Reducing the number of built-in functions simplifies programming. Two Unicon functions, open() and close(), have been extended to support VoIP session opening and closing. Another function, VAttrib(), has been added that is similar to an existing Unicon graphics function WAttrib().

3.4 Adding new VoIP Built-in Functions to Unicon

No programming language is totally complete; sometimes there is a need to do things in a language that are not in the language itself. However, those things can be done in another language. This generates two questions; (1) does this language have any support for using programs written in other languages? and (2) does this language have the flexibility for new improvements or upgrades. Unicon supports a mechanism for accessing C functions from a Unicon program [23], and also has the flexibility to upgrade its VM with new functionalities through its supported Run-Time Implementation Language and its open source code. The difference between those two mechanisms is not so much in the functionality as in usability. Calling functions written in C from a Unicon program is not as easy for

the beginner programmer as using a very high level built-in function supported by the Unicon VM. Building it into the language potentially allows greater runtime system integration and higher semantic level than the C calling interface. Plus it minimizes and simplifies the addition, and produces more portable distributions and executables.

3.5 Built-in Functions

Adding new VoIP functions to the Unicon VM and its run time system requires programming using the RTL language and a set of efficient C API functions and structures on top of the VoIP libraries. The RTL code determines the format of the built-in functions as seen by the Unicon programmers, including the functions' names, their parameters and their return types. In order to make those new functions consistent with the Unicon language design, the low level details are hidden within the language.

The VoIP facilities extend two existing Unicon functions open() and close(). Now open() mode "v" will open a VoIP session and close() checks if the passed parameter is a VoIP session and if it is, wraps that session up. This way of extending functions and overloading them with new jobs instead of adding new dedicated functions makes the language easier to work with; since there are fewer functions to remember.

The project also added a new function dedicated for VoIP and with ability to be extended in the future for new additions in the field of audio. This function is VAttrib(); it can be used to set up the VoIP attributes such as adding new host-names into the session, dropping some host-names out of the session, and querying about those who are in the session.

CHAPTER 4
IMPLEMENTATION OF UNICON'S AUDIO/VOIP FACILITIES

Augmenting the Unicon run-time system and its virtual machine with new audio and VoIP built-in functions requires an understanding of the Unicon source code structures and the tools used to build Uniocn such as the Run-Time Language (RTL). Care was taken for the added functionality to be consistent with the syntax and the semantics of the rest of the Unicon language: Unicon's semantics is expression based; all code in Unicon is evaluated in terms of expressions that may fail or succeed.

The Unicon VM and runtime system source code is written in C, while JVOIPLIB is a C++ library. To use it, API functions must be accessible from the Unicon C source code. A mechanism is needed to access C++ from C. This chapter starts by explaining how to mix C and C++ programs and then goes into the details of the data structures and the API functions that have been built on top of the JVOIPLIB C++ library to make them accessible from the Unicon source code. The chapter then proceeds explaining those new built-in functions and their impact on the Unicon Virtual Machine.

4.1 Mixing C and C++

The C++ language supports techniques for mixing code that can be compiled using C and C++ compilers. However, different degrees of success can be achieved during porting such mixed code. Success depends on the platforms and the C/C++ compilers themselves. The first requirement for such a mixed-language application is that the C and C++ compilers must have the same way of defining the basic types such as int, float, char or even pointers, etc. The second requirement is that the runtime libraries and the header files for the C compiler must be compatible with the C++ compiler. If the C++ compiler provides its own versions of the C headers, the versions of those headers must be compatible with the C compiler [24] [25].

If a program is primarily a C program but makes use of C++ libraries, it must link in the C++ runtime support libraries provided with the C++ compiler. The standard C++ runtime library libstdc++.so.x.x or libstdc++.a is linked automatically if you are compiling using g++. However, when compiling using gcc, it will not link automatically; they need to be link in explicitly. The libraries that must be linked in depend on the options that are used during the compilation of the C++ program and on the C++ features that are used [24] [25].

4.1.1 Accessing C from C++ Program

C functions in general do not require much modification in order to be used in C++ programs. C and C++ are link-compatible languages. This thesis does not go into the details of accessing C code from C++ since our problem was the less common opposite case.

4.1.2 Accessing C++ functions from C Program

To use a C++ function within C program, a program can take advantage of the link-compatibility between C and C++. In order to be C compatable, a C++ function mustbe declared as *extern "C"*. Adding *extern "C"* at the beginning of a C++ function is called a C linkage [25].

```
extern "C" int print_out(int i, double d)
{
        std::cout<<"i="<<i<<"\n"<<"d="<<d;
}
```

C linkage can be done only if the function's parameters and the function return type are compatible with the C language. For example, if the function declaration has a reference to an IOstream class as a parameter, there is no portable way to explain the parameter type to the C compiler, since the C language does not know any thing about C++ features such as templates and classes [24] [26].

4.1.3 Accessing C++ Classes from C Program

You can not declare a class inside your C code, but C and C++ are compatible in their pointer types. The best way to use C++ class instances from a C program is to use pointers to classes. This is very similar to the way that anonymous structures such as FILE in C standard I/O are used. Using the C linkage for C++ functions that can access any C++ property including classes, we can call these functions from C program [24] [26].

Example: Let us say that we have the following class

```
class Bar{
        public:
                Bar() { i=2; j=3 }
                int foo(int k){ return (k*i*j); }
                // ...
        private:
                int i, j;
};
```

In order to access this class from a C code we need the wrapper function with the C linkage. This function could look like

```
extern "C" int foo_bar(Bar * m, int i)
{
        std::cout<< m->foo(i);
        return 0;
}
```

4.2 A C++ Language API for JVOIPLIB

Instead of converting the JVOIPLIB C++ code into C, which is time consuming and may be very difficult, it is possible to access the C++ code using the C linkage obtained by *extern "C"*. In order to make this possible, first it is necessary to determine the exact C++ functions and data structures needed to be accessed by the C program in order to use the C++ library. Second, write wrapper functions for the C++ functions and structures which are needed to be exposed to the C code in order to make the C++ library usable in a C program. Finally, it is valuable to build an interface library from those wrapper functions on top of the original C++ library. Since C and C++ are link compatible, we can link this interface library and the original C++ library into the C code in the Unicon VM *iconx* after adding the necessary built-in functions using the Unicon Run-Time Implementation Language.

This use of the C linkage and the wrapper functions allows a program to use a C++ library inside a C program. And that is what we did in order to make the C++ in JVOIPLIB usable inside the Unicon VM.

4.2.1 The Main JVOIPLIB Classes (Data Structure)

JVOIPLIB 1.3.0 has four main classes affecting the VoIP session. Those classes must be accessed in order to make a VoIP session possible. The VoIP session components are JVOIPSession, JVOIPSessionParams class,and JVOIPRTPTransmissionParams. There is one more class required on Linux OS; this class is JVOIPSoundcardParams.

- *JVOIPSession*: encapsulates the essential data and methods that are responsible to create and destroy a VoIP session.
- *JVOIPSessionParams*: encapsulates the essential session parameter data and methods needed to setup the VoIP session parameters. The JVOIPSessionParams class has variables; some of those variables affect the quality

Variables	Default Value	Possible Values
inputsamprate	8000	{4000, 8000, 11025, 22050, 44100}
outputsamprate	8000	{4000, 8000, 11025, 22050, 44100}
sampinterval	20	20
inputtype	SoundcardInput	{ NoInput, UserDefinedInput, SoundcardInput }
outputtype	SoundcardOutput	{ NoOutput, UserDefinedOutput, SoundcardOutput }
loctype	NoLocalisation	{ NoLocalisation, UserDefinedLocalisation, SimpleLocalisation, HRTFLocalisation }
comptype	NoCompression	{ NoCompression, UserDefinedCompression, ULawEncoding, SilenceSuppression, DPCM, GSM, LPC }
mixertype	NormalMixer	{ UserDefinedMixer, NormalMixer }
transtype	RTP	{ UserDefinedTransmission, RTP }
receivetype	AcceptAll	{ AcceptAll, AcceptSome, IgnoreSome }
inputsampenc	EightBit	{ EightBit, SixteenBit }
outputsampenc	EightBit	{ EightBit, SixteenBit }

Table 4.1: JVOIPSessionParams Variables

of service. Those values are set directly using its constructor and can be changed later during the session. Changing these values before or during the session will affect the quality of service. The quality of service also depends on the bandwidth and the traffic in the network. Table 4.1 shows those session parameters, their default values, and the possible values that the session can accept.

- *JVOIPRTPTransmissionParams*: It encapsulates the essential Real Time Protocol (RTP) parameter settings and provides the interface between the JVOIPLIB and the JRTPLIB.
- *JVOIPSoundcardParams*: it has some of the Soundcard input and output settings that are useful only on Linux OS.

4.2.2 The Supporting Unicon VM Data Structure

According to the previous JVOIPLIB components, and in order to make the VoIP session possible to be created and maintained in Unicon, the following structure has been defined for being in the Unicon VM.

```
#ifndef WIN32              /* for linux OS */
struct VOIPSession{
        JVOIPSession                    Session;
        JVOIPSessionParams              sessparams;
        JVOIPRTPTransmissionParams      rtpparams;
        JVOIPSoundcardParams            sndinparams;
        JVOIPSoundcardParams            sndoutparams;
        int                             MaxList;
        int                             InList;
        char                            **ListenerList;
};
#else                      /* for MS Windows OS */
struct VOIPSession{
        JVOIPSession                    Session;
        JVOIPSessionParams              sessparams;
        JVOIPRTPTransmissionParams      rtpparams;
        int                             MaxList;
        int                             InList;
        char                            **ListenerList;
};
#endif

typedef struct VOIPSession    VSESSION;
typedef struct VOIPSession*   PVSESSION;
```

- *VSESSION*: This struct represents the main components required for establishing and managing the VoIP Session in the Unicon VM.
- *PVSESSION*: This data type is a pointer to a struct of type VSESSION. In Unicon, the programmer needs to deal with variables of this type in order to make a VoIP session possible.
- *JVOIPSession*: This structure has the main exposed components from JVOIPLIB and some additional fields needed by the Unicon VM; MaxList, InList, and ListenerList. Those fields are added in order to maintain a list of the listener's host-names and their related ports. Whenever there is a new listener they are added to the list and whenever a listener drops out, they are deleted from the listener's list.

4.2.3 The C++ API Function Prototypes

The API functions are C++ wrapper functions working as an interface between C and JVOIPLIB C++ code. They are focusing on creating, casting,

stopping casting, setting up, and destroying a VoIP session. In order to give Unicon programmers the ability to create, maintain, and destroy a VoIP session, the Unicon VM was extended with calls to the following C API functions, which are designed to make it easy to deal with VoIP sessions. The function prototypes are defined as follows.

```
PVSESSION CreateVSession(char Port[5],char Destinations[]);
```

This function is responsible for allocating and creating an instance object of the VSESSION struct in the heap, which contains the main voice session components. It takes the base port that the VoIP session should be established on as a first parameter, the second parameter is a string for destinations with their base ports. The destinations are separated by commas and there is a colon between each username, hostname and its base port. This function returns a pointer to a VSESSION object. The returned pointer is the main handler for the created session. This function initializes the main session components contained in the VOIPSession struct defined previously. Then it tries to establish a session on the specified port, if the Destinations parameter is not NULL, it will start adding them into the destinations list and casting to them. It returns a non-NULL value if the components and the session are established successfully. Otherwise, it returns a NULL value.

```
void SetVoiceAttrib(PVSESSION VSession,char StrAttrib[]);
```

This function is responsible for adding new destinations to the session, deleting existing destinations, and querying about those who are in the VoIP session. It takes the VoIP session handle as the first parameter, followed by the attribute assigned to some destinations host-names or host-addresses with their port number separated by colons. The attributes are:

- *cast+=Username:Hostname:BasePort, ...*: This attribute is used to add new destinations to the VoIP session.
- *cast-= Username:Hostname:BasePort, ...*: This attribute is used to drop out some existing destinations from the VoIP session.
- *cast*: This attribute is to ask about the destinations that are in the VoIP session listeners list. It will print them out on the screen.

```
int Cast(PVSESSION VSession,char Destinations[]);
```

This function is used to cast to any number of destinations. It is called from SetVoiceAttrib() when the attribute is *cast+=*. The destination can be recognized by its username, host-address and the base port that should be passed to the function as a string argument separated by colons; the different destinations are separated by commas. Before using this function, voice session components must be allocated and initialized on both the source and the destination. This function returns 1 if the casting has been done successfully, otherwise it returns a 0. Success does not mean that the destination is receiving what the source is casting; it only means that the source can cast and the destination can benefit from the received voice data only if he already has an active voice session. Returning 0 means it can not add the destination to the session for some reason.

```
int DropCast(PVSESSION VSession,char Destinations[]);
```

This function stops the casting to one or more of the destinations. It will be called from SetVoiceAttrib() when the attribute is *cast-=*, which means there is no need for it to be declared with the C linkage. DropCast takes a pointer to a struct of type VSESSION, and a bunch of host-names or host-addresses, each one attached to its base port with a colon; the users are separated by commas in the string parameter. It returns 1 if the destinations have been deleted successfully. Otherwise, it returns a 0.

```
void CloseVoiceSession(PVSESSION VSession);
```

This function is responsible for closing the voice session and returning the allocated memory for the VSESSION struct back to the heap. It takes the session handle, which is of type PVSESSION, as an argument and returns nothing.

Those VoIP API functions can be used to establish and maintain a voice session. First, the programmer needs to create and allocate the essential voice components needed for a session. This can be done using the C function.

```
PVSESSION MySession;
MySession=CreateVoiceSession("5000",NULL);
```

Variable MySession is of type PVSESSION, which is a pointer variable to a VSESSION struct. This function allocates the necessary space for the voice components and creates the real voice session object at the base port 5000. The second parameter is NULL, which means that there are no destinations we need to cast to at this time. This function returns a non-NULL value when it successfully creates and initializes the session components; otherwise it returns a NULL value. This function will initialize the session components with the predefined default settings.

At this point, the session is ready for unicasting or multicasting using one of the specified functions. In order to unicast to a specific destination and make a peer-to-peer voice connection, the Unicast function is used.

```
Status=Cast(MySession,"ziad:128.123.64.48:5000");
```

The same happens with multicasting, except that it takes a list of IP addresses instead of only one destination IP address and a list of base hosts associated with the IPs. After unicasting or multicasting, it is possible to stop casting to a specific destination using the following function:

```
Status=DropCast(MySession,"ziad:128.123.64.48:5000");
```

It is very important to close the voice session explicitly before the end of the program by calling the function:

```
CloseVoiceSession(MySession);
```

If for any reason the programmer forgot to close the VoIP session, this will keep the session open and the allocated resources reserved which could affect any reopening for another VoIP session during the life of the program. Also, on the Linux sound system OSS, the sound device */dev/dsp* can be allocated only by one program. Closing the VoIP session will release it for any future opening. If the VoIP session did not close at the end of the session the sound device will be not available.

4.3 The Impact on the Size of the Unicon VM

There is a tradeoff between adding new functions into the Unicon VM and the cost of this addition. Our cost here is in the size of the VM itself. In order for VoIP facilities to be practical as a built-in part of the language, they must impose a reasonable increase in the VM size. Jori's VoIP libraries are large enough to make an interesting question whether this addition should be part of the standard Unicon distribution or not. This section describes the size of those libraries and the increase in the VM caused by them. The resulting changes in the Unicon VM *iconx* after adding the VoIP functions on the Linux and Windows systems is in the following sections.

4.3.1 The Impact on the Linux Unicon VM

The Linux Unicon VM is *iconx*. The size of *iconx* before augmenting it with VoIP is (583 KB), after applying the strip command; the size is reduced to

(516 KB). After adding VoIP facilities, the new size of *iconx* is (1.5 MB). So, the increase in the *iconx* is about (0.9838 MB) which is about (190.5 %) of the original size of the VM. This ratio of increase, for the Unicon VM, is not acceptable.

4.3.2 The Impact on the Windows Unicon VM

On MS Windows, Unicon can be built using the native Microsoft Visual C/C++ compiler (MSVC) or using the GNU C compiler for Windows. Windows Unicon has two VMs. The first is *iconx.exe*; it is used for console applications with graphics disabled. The second is *wiconx.exe*, which is the VM with enabled graphics. The original size of those VMs, when they are built using gcc-2.95-mingw32, is (607 KB) for *iconx.exe* and (616 KB) for *wiconx.exe*.

After adding VoIP facilities, the size of *wiconx.exe* built using gcc-2.95-mingw32 is (1.34 MB). The increase in the Unicon Windows VM, *wiconx.exe*, is about (728 KB), which is about (118%) of the original size of the VM. This ratio of increase is not acceptable for the Unicon VM.

4.3.3 Disabling Components from JVOIPLIB

JVOIPLIB can be built on both UNIX/Linux and MS Windows. On Linux the compiler is g++ and the produced library is libjvoip.a, whereas on MS Windows the library can be built either using the Microsoft Windows C/C++ compiler (MSVC) where the generated library is jvoiplib.lib, or using any g++ for Windows such as gcc-2.95-mingw32 which we used to build the libjvoip.a library.

The library sizes on both MS Windows and Linux are shown in the Table 4.2. JVOIPLIB supports many features such as compression techniques and 3D sound effects. Those features can be disabled without affecting the functionality of the library. However, it affects the size of the resulting library. Those disabled features can be enabled again at build time. The disabled features are the GSM, LPC, SILENCESUPPRESSION, DPCM, ULAW, compression techniques and the LOCALISATION_SIMPLE, LOCALISATION_HRTF 3D sound effects. After disabling those components, the new sizes of JVOIPLIB library are shown in Table 4.2. Figure 4.1 shows the size differences before and after Optimizing JVOIPLIB.

4.3.4 The Impact on the Unicon VM after Optimizing JVOIPLIB

Adding VoIP threatened to increase the VM size by a factor of 2-3 times or more. This situation encouraged us to find a way to reduce this size cost of VoIP. The structure of the Jori's VoIP libraries and their components allowed to disable some components that are not necessary for us such as the compression techniques and the 3D sound effects.

Item	Size
jvoiplib.lib *(win32- MSVC)*	1.7 MB
libjvoip.a *(win32- gcc2.95-mingw32)*	6.38 MB
libjvoip.a *(Linux gcc-3.3.4)*	1.6 MB
libjvoip.so.1.3.0 *(Linux gcc-3.3.4)*	1.2 MB

Table 4.2: JVOIPLIB Optimized

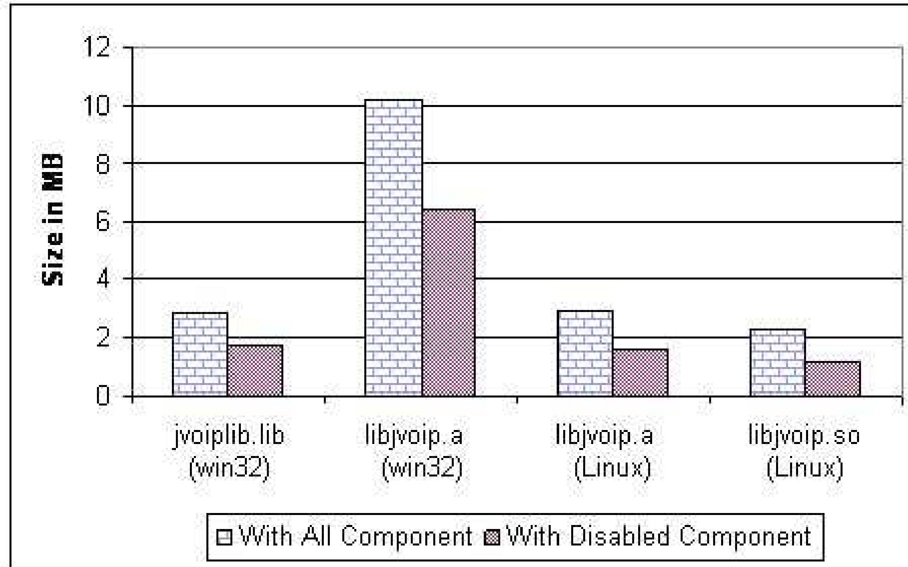


Figure 4.1: The Changes in JVOIPLIB after Disabling Components

On Linux, augmenting the Unicon VM *iconx* with VoIP facilities, after disabling some components of JVOIPLIB, increases the size of the VM with an acceptable ratio. The new size of *iconx* is (1.0 MB). The increase in the Linux VM is about (483.8MB) which is about (93.7 %) of the original size of the VM. This is a reasonable amount of increase even if it is still not acceptable to be part of the standard Unicon distribution.

On MS Windows, the size of the Unicon VM *wiconx.exe* built with gcc-2.95-mingw32 after augmenting it with VoIP facilities, is 850 KB. There is a (234 KB) increase in the VM, which is about 37.9 % of the original size of the wconx.exe. Figure 4.2 shows the Impact on the Unicon VM after optimizing JVOIPLIB.

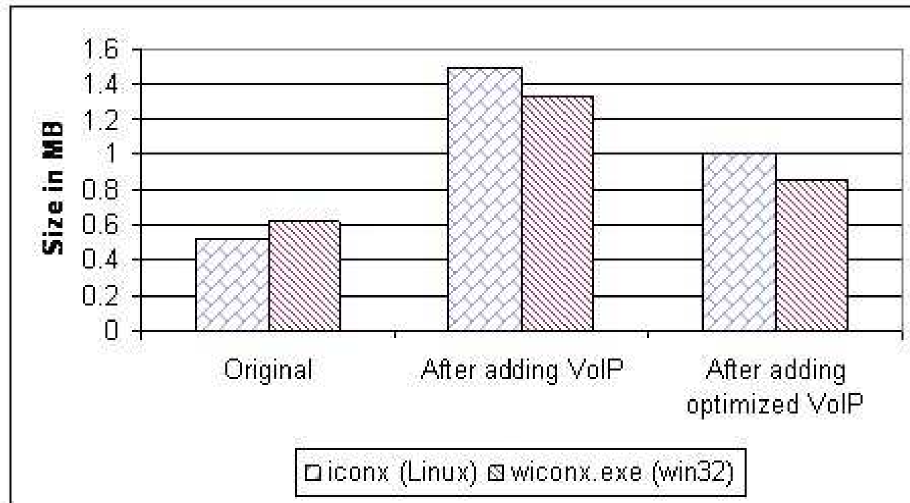


Figure 4.2: The impact on the Unicon VM after adding VoIP

4.3.5 The Impact on the Build Process

After disabling some components from JVOIPLIB, a ttained a more acceptable increase in the Unicon VM. However, since using VoIP is limited for a small number of applications. We retain flexibility to build Unicon with or without the VoIP facilities. This flexibility will save those who are not interested in VoIP some time during the build process beside the saved space in the Unicon VM.

Building the Unicon source code without VoIP on Suse Linux running on an Intel Pentium 4, CPU 2.4 GH, full cache, and 512 MB of RAM takes about six minutes. Building the Unicon source with voice enabled increases this time to nine minutes). This indicates that there is about 2 minutes building time for the VoIP library.

CHAPTER 5
USING THE NEW UNICON VOIP FUNCTIONS

Unicon is always under development. The project described in this thesis extended it in the area of sound and VoIP facilities in order to apply it in the field of Collaborative Virtual Environments (CVE). These new developments focused on playing sounds and supporting voice communications.

As in the case of text chat, in order to make a bidirectional voice connection, both of the end users must be in the voice session at the same time. So, in order to make a voice connection that enables two or more users to talk to each other in a phone-like conversation, both of them must have an open voice session and both of them must add the addresses of those who is going to talk to them into the destination list of broadcasting. However, if they want to make a one directional voice connection the destination must have at least an open voice session on some port.

In addition to the simple phone-like voice communications, Unicon now supports on-line conferencing, connecting n-user using a VoIP session through the high level built-in functions. Unicon also supports the idea of multicasting; one user can send to n-users, including servers which may extend the casting to another m-users and so on. Unicon's VoIP functions are designed to be consistent with the very high level language design. VoIP in Unicon is very simple and can be managed using only three built-in functions.

In addition to VoIP, Unicon was extended to play audio files such as WAV, MP3, and Ogg Vorbis on both MS Windows and UNIX/Linux. Audio recording and playback are an obvious important complement to VoIP facilities. The audio playback is accomplished using the built-in audio function PlayAudio(). This function has been built on top of the open source library OpenAL. PlayAudio() takes the audio filename as a string parameter and plays it in a separate thread in parallel with the main program thread. This mechanism allows the programmer to play audio and process something else at the same time. The built-in function StopAudio() can be used to kill the audio thread at any point of the program execution.

5.1 Unicon's VoIP Functions

The following are details of the three VoIP built-in functions supported with short Unicon example programs.

5.1.1 open()

The function open() with mode *V* or *v*, for voice, opens a voice session and returns a handle to that session which should be saved into a variable. It takes the port number to accept incoming voice connections on. There is an additional optional parameter that allows the programmer to specify the destinations at the opening time. This optional parameter is a string of one or more destinations attached to their ports; (i.e *Username:address:baseport*), in the case if there is more than one destination, different destinations are separated by commas in the same string. However, this function could fail if the sound device is reserved by some body else or if it could not open a socket for the RTP protocol.

The syntax of open is
Variable: = open ("base port", "V|v" [, string of one or more destinations])

Example:

```
# A Unicon program lets you connect to your self
procedure main()
    local VSession
    VSession := open("4500", "v", "yourname:localhost:4500")
    write("The voice session is opened. Press Enter to close it")
    read()
    close(VSession)
end
```

The previous program, opens a voice session and maks a connection to the localhost at the same time. The *yourname:localhost:4500* attribute lets you make a voice connection to the localhost at the same opening port which is 4500. The port usually is a number from 4000 to 6000, most of the time we use the port 5000 for VoIP. The *yourname:localhost:4500* can be followed by as many other hosts as you need separated by commas. However, the same program with the same functionality can be rewritten as follows:

```
# A Unicon program lets you connect to your self
procedure main()
    local VSession
    VSession := open("4500","v")
    write("The voice session is opened")
    VAttrib(VSession ,"cast+=yourname:localhost:4500")
    write("you are casting to yourself. Press Enter to close")
    read()
```

```
    close(VSession)
end
```

This program first opens a voice session as a listener only, then uses the VAttrib() function to add new destinations to the voice session using the *cast+=* attribute followed by the destinations list which in this case contains only the local machine. In general it can consist of as many destinations as necessary, different destinations are separated by commas.

5.1.2 close()

To close a voice session that is already opened there is a close function. close() takes the voice session handle as a parameter and returns nothing. It should close and clean every thing related to the voice session.

Example:

```
# A Unicon program lets you make a voice connection with
# ziad:128.123.64.49 at port 5000 and
# jeffery:unicon.cs.nmsu.edu at port 4500
procedure main()
    local voice, dest1, dest2
    dest1 := "ziad:128.123.64.49:5000"
    dest2 := "jeffery:unicon.cs.nmsu.edu:4500"
    voice := open("5000", "V", dest1 || "," ||dest2)
    write("the voice session is opened. Press Enter to close it")
    read()
    close(voice)
end
```

In the previous program, notice that each destination is attached to its own opening port and the destinations are separated by a comma. Also, the destination can be defined by its host-name or by its IP-address. The same program can be written as follows:

```
# A Unicon program lets you make a voice with two destinations
# ziad: 28.123.64.49 at port 5000 and
# jeffery:unicon.cs.nmsu.edu at port 4500
procedure main()
    local voice
    voice := open("5000","V","ziad:28.123.64.49:5000")
    VAttrib(voice,"cast+=jeffery:unicon.cs.nmsu.edu:4500")
```

```
    write("the voice session is opened. Press Enter to close it")
    read()
    close(voice)
end
```

Here, one of the listeners was specified when the voice session is opened, the other was added after opening the voice session, this addition done using the VAttrib() function.

5.1.3 VAttrib()

The function VAttrib() manages attributes, performing tasks such as adding and dropping destinations from the voice session and making a queries about those who are listening to the voice session at any time.

The VAttrib function takes two parameters, the first is the voice session handler which has been obtained by the previous open function and the second is a string which has an attribute that may be attached into a list of destinations especially in the case of adding and dropping. Those attributes are:

- VAttrib() can add one or more destinations to an open voice session. The attribute *cast+=destinations* adds new destinations. For example: VAttrib(voice,"cast+=ziad:localhost:5000,jeffery:unicon.cs.nmsu.edu:5000") Each destination consists of *user-name:host-address:base-port*. More than one destination can be separated by commas.
- VAttrib() can drop one or more destinations from an open voice session. The attribute *cast-= destinations* dropps one or more existing destinations. In this case, the destinations can be a string of names, addresses or both names and addresses. For example to remove both Ziad and Jeffery from the existing voice session we can use either one of the following:
  VAttrib(voice, "cast-=ziad,jeffery") or
  VAttrib(voice, "cast-=loclahost:5000,unicon.cs.nmsu.edu:5000") or
  VAttrib(voice, "cast-=ziad:localhost:5000,jeffery:unicon.cs.nmsu.edu:5000");
- The VAttrib function also provides a mechanism for getting information about the voice session listeners. VAttrib() with the *cast* attribute will return a string of the current session listeners, *castlist* will return the listeners information each listener as an entry in a list, *castnames* returns the listeners names, *castaddresses* returns the listeners addresses with their base ports. This flexibility saves the programmer the effort to process the listener information into the shape that is suitable for his own use.

The syntax of VAttrib is

VAttrib(Session handle,"cast+=dest[,dest] | cast-=dest[,dest] | cast")

Example:

```
procedure main()
    local voice, dest1, dest2, listeners
    dest1 := "ziad:zorro.cs.nmsu.edu:4500"
    dest2 := "erick:128.123.64.225:5000"
    voice := open("5000","V","jeffery:unicon.cs.nmsu.edu:5000")
    write("The voice session is opened successfully.")
    VAttrib(voice,"cast+=" || dest1 || "," || dest2)
    write("Ziad and Erick have been added to the voice session")
    write("To drop Erick out of the voice session press Enter")
    read()
    VAttrib(voice,"cast-=erick")
    # It is the same as VAttrib(voice,"cast-=" || dest2)
    write("To get the listeners as a string: Press Enter")
    read()
    listeners := VAttrib(voice, "cast")
    write("To get the listeners as a list: Press Enter")
    read()
    listeners := VAttrib(voice,"cast=list")
    every write(! Listeners)
    write("To get the listeners names as a list: Press Enter")
    read()
    listeners := VAttrib(voice,"cast=list(name)")
    every write(! Listeners)
    write("To close the voice session: JUST Press Enter")
    read()
    close(voice)
end
```

The VAttrib() function is very important since it allows the programmer to see and remember those who are listening to the current voice session especially in the case of a multicasting situation for n-users conferencing.

5.2 Unicon's Audio Functions

Unicon's Audio functions are designed to be consistent with the very high level language design. Audio in Unicon is very simple and can be managed using only two built-in functions that can play WAV, MP3, and Ogg Vorbis audio files.

Audio plays in the background, allowing users to perform other tasks while playing audio. The following are details of those audio functions supported by short Unicon programs.

5.2.1 PlayAudio()

This function is not for VoIP, it plays audio files in formats such as the traditional WAV format, MP3, and Ogg Vorbis. PlayAudio(filename) takes the string name of the audio file and returns an audio file handle. This handle can be used to stop the audio playing at any time. The most important property of this function that it does not block the execution of the program until the audio is finished or stopped, but it starts the playing in a background thread while the program execution continues normally.

Example:

```
procedure main()
    local audio
    audio := PlayAudio("sample.ogg")
    write("still playing !!!)
    read()
end
```

5.2.2 StopAudio()

The PlayAudio() function does not block the program execution. There is a need for a control function that can be used to stop the audio playing at any time. StopAudio() does this job. It takes the audio file handle that was returned from the PlayAudio() as a parameter and it returns nothing.

Example:

```
procedure main()
    local audio
    audio := PlayAudio("sample.ogg")
    write("To stop the audio playing at anytime, Press Enter")
    read()
    StopAudio(audio)
    write("Good bye !!!")
end
```

CHAPTER 6
AN APPLICATION EXAMPLE: THE NMSU'S CVE

CVE stands for Collaborative Virtual Environment. In general, Collaborative Environments (CEs) are places where people get together, work together, or even play together in the same physical place at the same time. However, CVEs are kind of currying the same meaning with one crucial change, this change is replacing the physical place with an online digital virtual space. people from different geographical parts of the world can get in touch with each other in a realistic fashion over the virtual space.

Information sharing and communication tasks are in the center of the CVEs. One of the CVE conferences defines a CVE as a "computer-based, distributed, virtual space or set of places. In such places, people can meet and interact with each others, with agents or with virtual objects. CVEs might vary in their representational richness from 3D graphical spaces, 2.5D and 2D environments, to text-based environments"[27].

The language design presented in chapter 4 and 5 of this thesis provides Unicon with audio and VoIP facilities that are useful in CVEs. Most entertainment-oriented CVEs focus on 3D graphics with a small portion of audio used for sound effects. Building a CVE with voice communication facilities elevates the CVE into a new level of usefulness and reality. voice communication is just as important as the other 3D or 2D collaborative tools in a CVE. If 3D graphics make CVEs feel like a place, voice communications within the CVE make that place feel real and more useful.

The NMSU CVE is an educational CVE that mainly aims to serve distance education students, especially those who are prevented from physically attending CS classes for various reasons such as having work obligations or living far from the NMSU main campus. NMSU's CVE provides a 3D model of the actual CS department at New Mexico State University, supported with 2D collaborative tools such as text chatting, live white-board (Electronic Board), shared Integrated Development Environment (IDE) such as shared text editor and file browsing, and VoIP facilities for peer-to-peer voice communication, broadcasting of live classes, and teleconferencing. Figure 6.1 is a screen shot of the client interface of the NMSU's CVE.

Within the CVE, VoIP can be used synchronously with other tasks, because the VoIP language design we did earlier separates the voice capturing and playing back into a separate thread allowing other CVE tasks to be performed in parallel with voice.

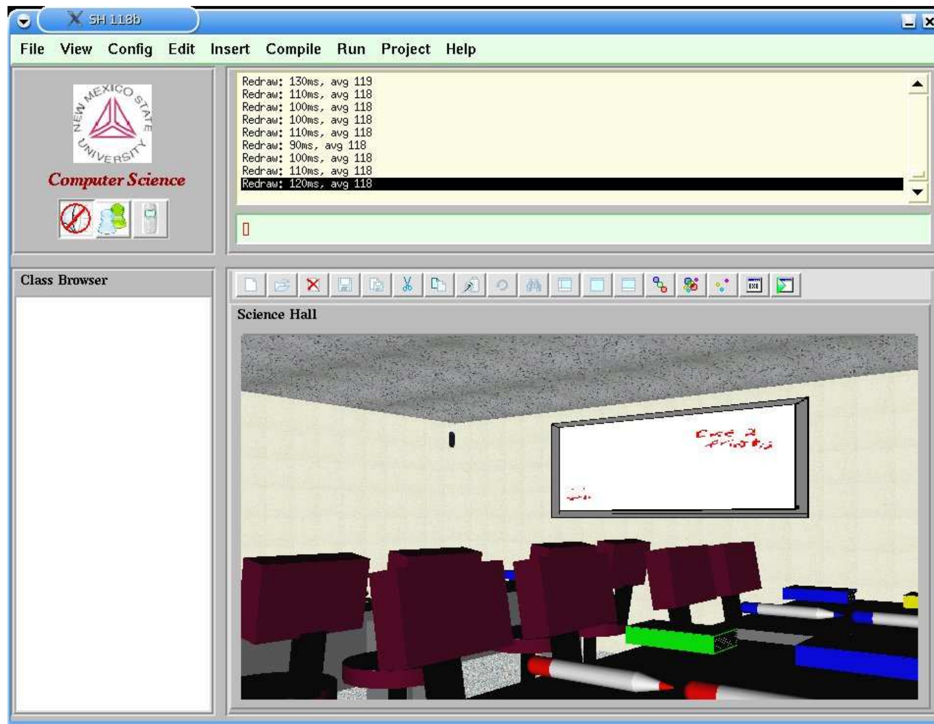


Figure 6.1: The Client Interface in the NMSU's CVE

6.1 Voice Implementations in the CVEs

The telephone is a very important means of communication. However, the telephone interface provides an unnatural communication environment that is very different from face to face communication. Despite this, it has become widely accepted by people who see it as a normal form of communication. Using IP networks for voice communication (VoIP) will dramatically change the uses and the interface of the telephone communication. One of those changes is the use of voice communication within CVEs [28].

Voice within CVEs can be implemented by many strategies depending on the purpose. One strategy can be location based connection; people in real life usually get close to each other to have a conversation. In the CVE, this can still be applicable; people who want to talk to each other can move their avatars close to each other and start talking. A distance policy can be used to start and close the voice connection such as a specific number of meters as an audible distance between the avatars.

Another location-based policy for using voice communication within CVEs could be a room-based voice connection policy instead of distance-based voice connection. The 3D modeling in the CVE consist of virtual rooms in the simu-

lated world, a design like this makes a room-based voice connection policy a very reasonable choice; avatars in the same room can have a bidirectional voice connection with each other. This is a reasonable policy only for small rooms. In real life, people in a small room can talk to each other and can hear each other clearly. However, in a big room, this policy can be unrealistic. In big rooms; people at one end of the room can not hear others at the far end unless they are using speakers, or other technologies. However, connecting a large number of people in one room may pose a network and a CPU challenge. The room-based voice connection policy in large rooms in the CVE can be realistic if the large room is divided into several virtual rooms. Anyway, the room-based voice connection can still be applicable for the NMSU's CVE, since it is an educational CVE and its rooms are class rooms, corridors, and study rooms which are small enough that people can talk to each other and hear each other clearly.

Even though the location-based voice connection in its two policies; the distance-based voice connection and the room-based voice connection, are very reasonable within the CVE, there is still a lack of security and control over those who are talking to each other since any one can get close enough to the speaker person and get a bidirectional voice connection with them automatically, this is a security and privacy breaker; especially, when there is no control over those who are getting close from the talking person or even over those who can enter a room and get a bidirectional voice connection with its members. So, a necessity for a secure or at least controlled voice connection is needed.

For sake of security and controlled voice connection, there is a new policy that is not a location or room-based voice connection policy. It is called the Virtual Cell Phone, we assume that each avatar within the CVE has its own Virtual Cell Phone that can be used to call any other CVE user with privacy and controlled voice connection. The user avatar can be in one of the two modes, the location-based voice connection or the Virtual Cell Phone voice connection, in this way those avatars who are on the Virtual Cell Phone can not be interrupted by others even if they are very close to them or reside in the same room. Moreover the Cell Phone user has a full control over those who want to talk to him supported with a flexible GUI interface that allows him to easily accept, reject, put on hold, or put online.

6.2 Voice Modes in the NMSU's CVE

Thus far 3D virtual environments have been used mainly for entertainment. The NMSU CVE's voice modes are designed to support the educational activities. Users in the CVE have their own representative avatars that can be moved around the 3D model of the first floor of the science hall. Avatars can perform several tasks and voice communication is one of them. The science hall consistst of class

rooms, corridors, and staff offices. Initially at login time, the avatar voice mode is off; no voice connection at all. In addition to this mode, the avatar can be in the Local mode, the Group mode, or the private mode. Those modes are separate and no interaction between them. However, the avatar can still be in any of those four voice modes and performing some other CVE tasks in parallel. From the voice tab in the CVE GUI interface the user can switch between the different voice modes according to his wish.

6.2.1 Disabled "No Voice"

This mode is the default mode when the user logs in to the CVE. In this mode there is no voice involvement for the user; he can not talk and can not hear anybody talking until he switchs to one of the other voice connection modes. This mode is used when the user does not have a working soundcard, speaker or microphone, or the user's internet bandwidth is not sufficient for a voice connection. In other cases, the user may be busy doing something in the CVE and does not wish to be interrupted with voice communications; so he turns the voice off.

6.2.2 Local "Current Room"

We presented earlier in this chapter some of the strategies that can be applied as voice connection policies inside the 3D virtual environment. According to NMSU's CVE design, the room-based connection would be a good choice for users with high-bandwidth connections, this mode called "Local Mode".

People generally get close to each other when they have intention to communicate. In the 3D virtual environment, users in the same virtual room can talk to each other. When the avatar in Local Mode enters a room in the CVE, it gets a bidirectional voice connection with the other N-1 avatars currently in that room and in the same voice connection mode. When the avatar leaves the room, it loses the voice connection with the avatars in that room and gets a new bidirectional voice connection with the M-1 avatars which are in this new room and are in the same voice connection mode. So, if he speaks, they hear him and vice versa.

Figure 6.2 is a screen shot of the GUI user interface where the user has selected the Local Mode voice connection. It shows the avatar current room and a list of those who are in that room at that time. This list is dynamically changed according to the avatar movements.

6.2.3 Private "Virtual Cell Phone"

Some users in the CVE may want to have a private conversation with each other. The Local Mode in the 3D Virtual Environment is location-based and it

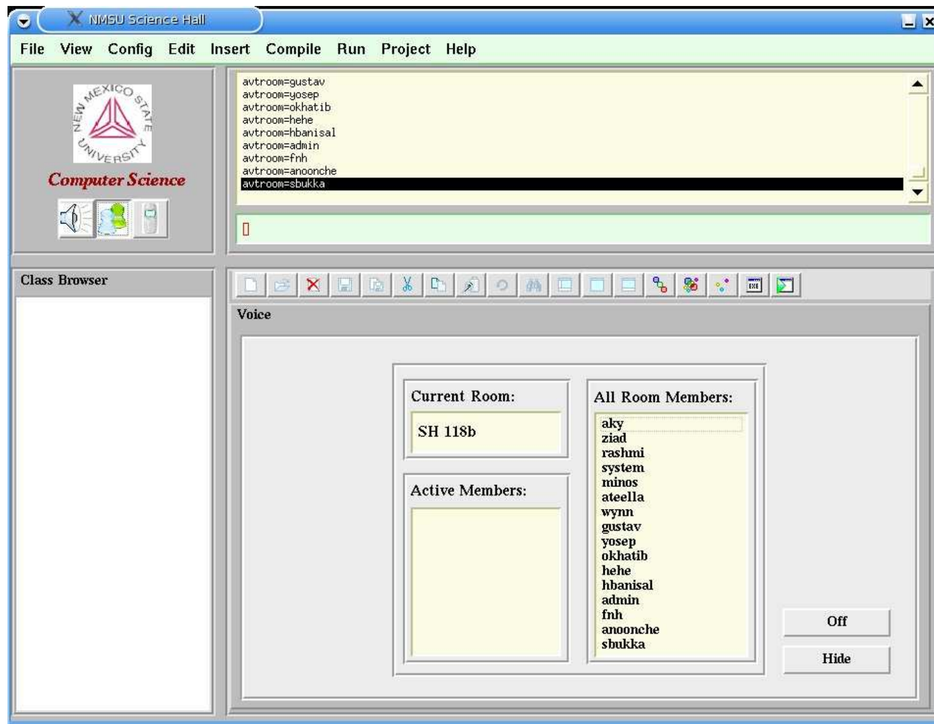


Figure 6.2: The GUI interface for the Local “Room-Based” Voice Connection Mode

is not secure; anybody who enters the room in the CVE can get an automatic bidirectional voice connection with its members. A secure or controlled voice connection mechanism is needed which is not 3D location-based, this controlled voice connection is called the Virtual Cell Phone. In real life, when you get a call on the cell phone, you can see who is calling and it is up to you to open the line and talk or ignore it, if you want to make a call you just dial the number you want to call. Our Virtual Cell Phone allows the user to have control which other users are talking to him, accept calls, put people on hold, remove the hold, and most importantly, it allows N-connections at one time. This cell phone provides the user with enough details that can help him see those users that he is connected with and currently online, those users that he is connected with and currently onhold, those who are trying to call him, and those he is trying to call.

This mode is very important because it provides the security and the control over those whom the user is going to speak with. This situation is different from the local mode where the user can speak and any user in the current room can hear him with no control.

Figure 6.3 is a screen shot of the GUI interface for the Virtual Cell Phone in the NMSU's CVE. It shows the various components that the user in the private

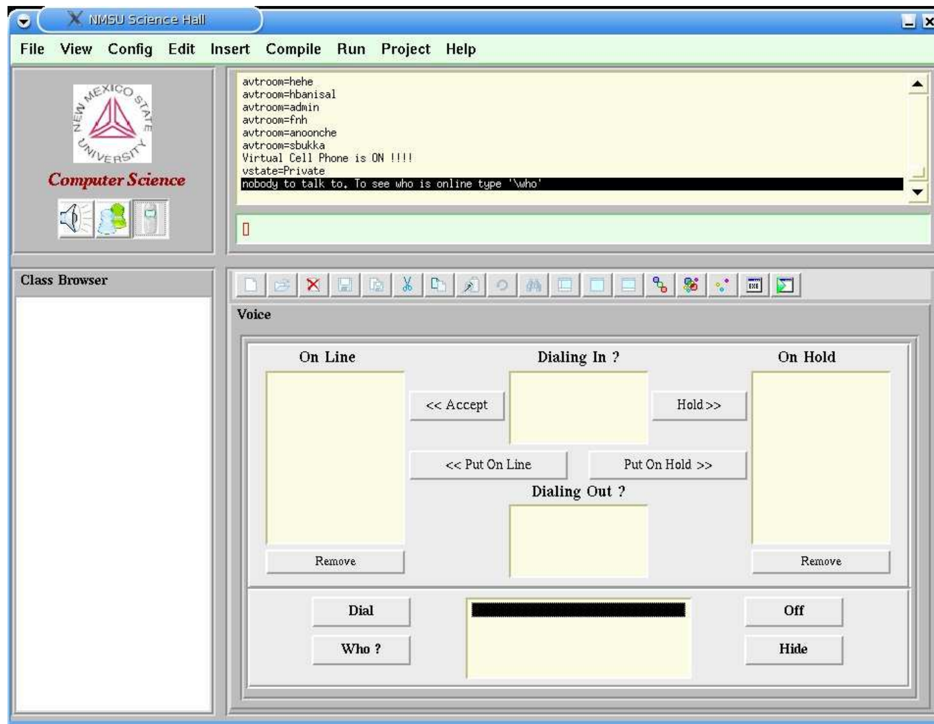


Figure 6.3: The GUI interface for the Virtual Cell Phone

mode can use. When the user switchs to the Private mode in the Voice Configuration tab, his voice state will be changed and will be ready to make any call and can be called by any other logged in user in the same mode. Pressing the who button will update the list with those users who are currently in the private mode; their names will show up in the corresponding list and the user can select any one and press the Dial button. Pressing the Dial button sends a voice request to the selected user, the called user name will show up in the Dialing-Out list and it will remain their until he gets accepted or put on hold by the callee. In the callee side, the caller name will show up in the Dialing-In list; the name will remain there until it gets selected by the callee.

The callee can accept the caller by selecting his name from the Dialing-In list then followed by pressing the “Accept” button, pressing this button affects both the callee and the caller; in the callee side, it moves the user name from the Dialing-In list to the On-Line list and adds him to the active voice session, in the caller side, it moves the user name from the Dialing-Out list to the On-Line list and adds him to the active voice session. Both the caller and the callee will start a bidirectional peer-to-peer voice connection.

If the callee is not ready for a voice connection at the moment, he can decline the call by leaving the caller name in the Dialing-In list or by being nicer

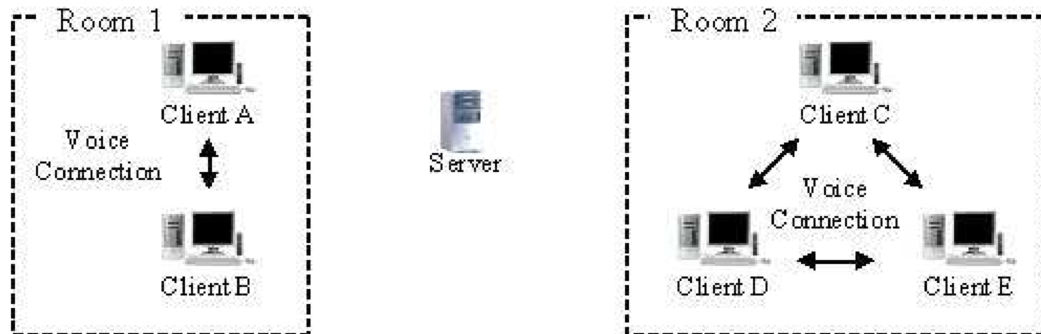
and moving him to the On-Hold list after selecting his name that followed by pressing the "Hold" button; pressing the "Hold" button affects both the callee and the caller; in the callee side, the name will be moved from the Dialing In list to the On-Hold list, in the caller side, the name will be moved from the Dialing-Out list to the On-Hold list. However, no voice connection will happen until the user name moved to the On-Line list. The user in the private virtual cell phone mode has the control and the flexibility to move any user any time from the On-Line list to the On-Hold list and vice versa.

6.2.4 Group "Future Work"

This mode is left for future work. It will serve very special situations such as class rooms, group meetings, and on line conferencing.

6.3 Voice Implementation in the NMSU's CVE

The voice connection in the 3D location-based is peer-to-peer. The server is only a session manager and organizer; its involvement is reduced to the connection and disconnection management. This mode will save the server much of unnecessary work and will increase the scalability of the system. Figure 6.4 shows the voice implementation in the 3D Room-Based voice connection. However, figure 6.5 shows the voice implementation for the Virtual Cell Phone.



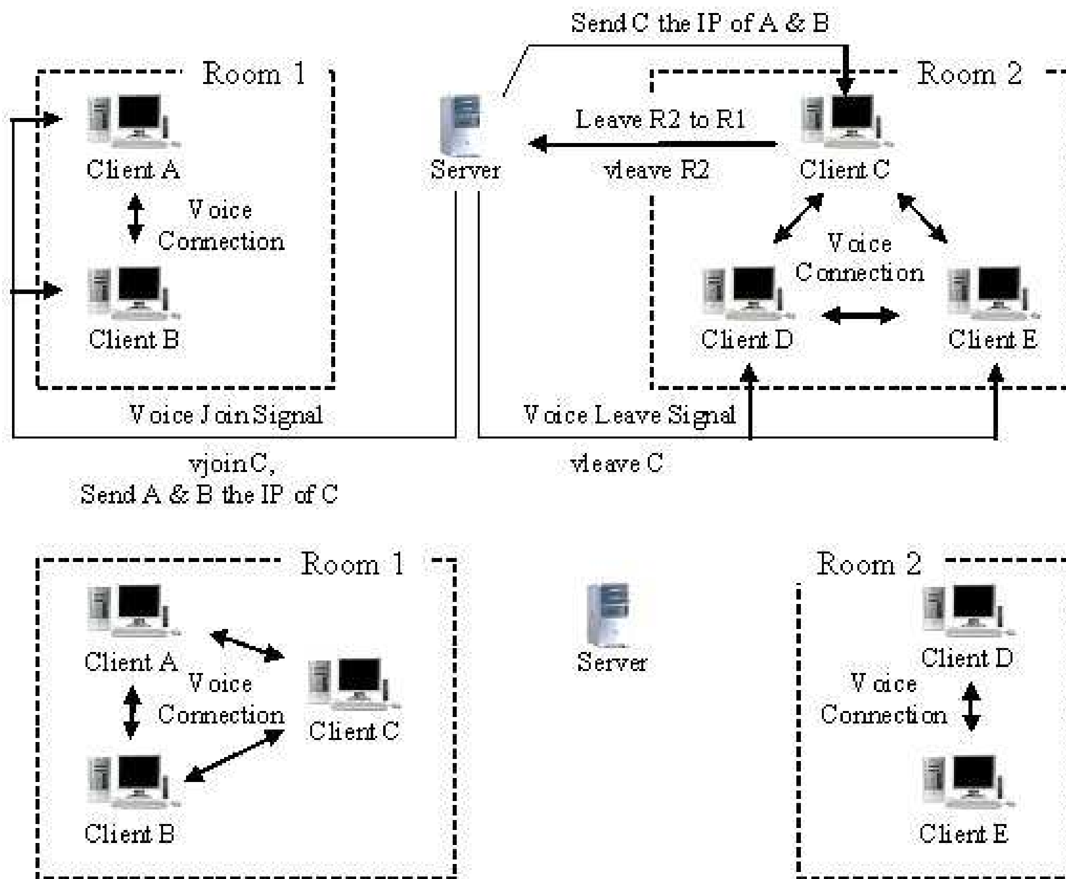
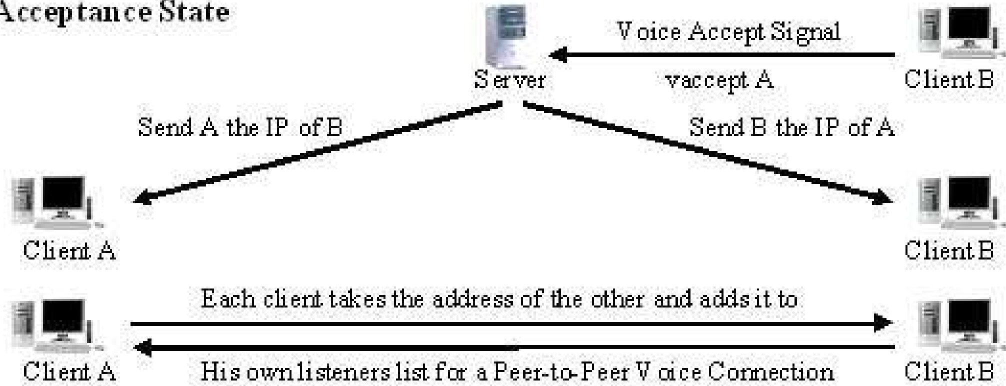


Figure 6.4: The Voice Implementation in 3D Room-Based Environment

1) Initial State

2) Acceptance State



3) Rejection State

4) Holding State

5) Unholding State

6) Ending State

Figure 6.5: The Voice Implementation in the Virtual Cell Phone

CHAPTER 7
CONCLUSION

Unicon is a very high level, goal directed, object oriented, general purpose applications language. It is been extended recently to support CVEs, and VoIP. VoIP can be established and programmed in many languages like C/C++ or Java. However, in order to produce a program in C or C++ that can use VoIP a lot of work is needed, which makes it time consuming. In Java the situation could be a little bit easier since there is a lot of work going on in Java to support APIs for the Session Initiation Protocol (SIP) which mostly used in Java VoIP. However, as far as I know, Java itself does not support any high level VoIP APIs. All this, could make Unicon the first programming language that supports the VoIP as a built-in facility.

The Unicon VoIP support is very abstracted and very high level; it aims for ease and usability, introducing just two new built-in functions in a very high level syntax. Importantly the Unicon VoIP facilities have no need for a running server, even though it can be programmed to work through a running server.

REFERENCES

[1] Jori Liesenborgs, Wim Lammotte, and Frank Van Reeth. Voice over ip with JVOIPLIB and JRTPLIB. *IEEE 26th Annual Conference on Local Computer Networks*, 2001.

[2] Jori Liesenborgs. *Voice over IP in networked virtual environments*. PhD thesis, 2000.
http://lumumba.luc.ac.be/jori/thesis/thesis.html.

[3] Libsndfile.
http://www.mega-nerd.com/libsndfile.

[4] OpenAL.
http://www.openal.org.

[5] Port Audio.
http://www.portaudio.com.

[6] Wikipedia, the free encyclopedia.
http://www.wikipedia.org.

[7] Ogg Vorbis.
http://www.xiph.org/ogg/vorbis/.

[8] The enterprise Linux resource.
http://applications.linux.com/.

[9] Russell Shilling. Videogame and entertainment industry standard sound design techniques and architectures for use in videogames, virtual environments and training systems. 2002.

[10] Simple DirectMedia Layer (SDL).
http://www.libsdl.org/.

[11] SDL MPEG Player Library (SMPEG).
http://www.lokigames.com.

[12] Java sound programmer guide.
http://java.sun.com/j2se/1.4.2/docs/.

[13] Marcus Goncalves. *Voice over IP Networks*. McGraw-Hill, 1999.

[14] 3Com IP Conferencing Module.
http://www.3com.com/products.

[15] ICQ.
http://www.icq.com/products/whatisicq.html.

[16] Arsenal project.
http://arsenalproject.org.

[17] Colin Perkins. *RTP audio and video for the Internet*. Addison-Wesley, 2003.

[18] Mark A. Miller. *Voice over IP: strategies for the Converged Network*. M&T Books, 2000.

[19] Uyless Black. *Voice over IP*. Prentice Hall, 2000.

[20] Ralph E. Griswold, Gregg M. Townsend, and Clinton L. Jeffery. *Graphics Programming in Icon*. Peer-to-Peer Communication Inc., 1998.

[21] Jeffery, Mohamed, Parlett, and Pereda. *Programming with Unicon*. September 2004.
http://unicon.sourceforge.net/book/ub.pdf.

[22] Kenneth Walker. The run-time implementation language for Icon. Technical report, Department of Computer Science, The University of Arizona, June 28 1994.
http://www.cs.arizona.edu/icon/ftp/doc/ipd261.pdf.

[23] Gregg M. Townsend and Ralph E. Griswold. Calling C functions from version 9 of Icon. Technical report, March 1996.
http://www.cs.arizona.edu/icon/docs/ipd240.htm.

[24] Stephen Clamage. Mixing C and C++ code in the same program. Technical report, July 2003.
http://developers.sun.com/prodtech/cc/compilers_index.html.

[25] Giri Mandalika. Mixed-language programming and external linkage. Technical report, March 2005.
http://developers.sun.com/prodtech/cc/compilers_index.html.

[26] Using C++ code in a C program.
http://gcc.gnu.org/ml/gcc-help/2004-07/msg00107.html.

[27] Elizabeth F. Chuchill, David N. Snowdon, and Alan J. Munro. *Collaborative Virtual Environments: Digital Places and Spaces for Interaction*. Springer-Verlag London Limited, second edition, 2002.

[28] Yasusi Kanada. Multi-context voice communication controlled by using an auditory virtual space,Central Research Laboratory, Hitachi, Japan. November 2004.
http://www.kanadas.com/voiscape/CCN2004.pdf.