IMPLEMENTATION OF VOICE OVER INTERNET PROTOCOL USING H.323

Thesis submitted in partial fulfillment of the requirements for the award of the Degree of

MASTER OF ENGINEERING IN COMPUTER SCIENCE ENGINEERING

OF BHARATHIAR UNIVERSITY

By

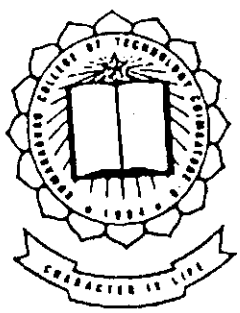
A.SENTHIL NATHAN

(Reg.No.0037K0011)

P-652

Under the Guidance of

Mrs. L. S.JAYASHREE M.E



DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

KUMARAGURU COLLEGE OF TECHNOLOGY

(Affiliated to Bharathiar University)

COIMBATORE – 641 006

2000 - 2001

CERTIFICATE

Department of Computer Science and Engineering

Certified that this is a bonafide report

of

the thesis work done by

A.SENTHIL NATHAN

(Reg.No.0037K0011)

at

KUMARAGURU COLLEGE OF TECHNOLOGY

COIMBATORE – 641 006

During the year – 2000 – 2001

18/12/01

Guide
Mrs.L.S.JAYASHREE
Computer Science Engineering Department
K.C.T., Coimbatore.

Head of the Department
Prof. S.THANGASAMY

Place : Coimbatore

Date : 20-12-2001

Submitted for Viva – Voce examination held at

Kumaraguru College of technology on 20-12-2001

20/12/01

Internal examiner

External Examiner

CERTIFICATE

This is to certify that this thesis work entitled **"IMPLEMENTATION OF VOICE OVER INTERNET PROTOCOL USING H.323"** being submitted by **A.SENTHILNATHAN**,(Reg.No.0037K0011) for the award of degree of **MASTER OF ENGINEERING IN COMPUTER SCIENCE**, is a bonafide work carried under my guidance. The results embodied in this thesis have not been submitted to any other university or institute for the award of any degree or diploma.

16/12/04.

L.S.JAYASHREE M.E
Senior Lecturer
Department of Computer Science Engineering
Kumaraguru College of Technology
Coimbatore.

ACKNOWLEDGEMENT

It is the duty of the author to express his deep sense of gratitude to his **Parents** and **Sister** whose support and encouragement made to do this course **M.E., (COMPUTER SCIENCE&ENGINEERING)** in this prestigious institution.

The author wishes to take this opportunity to offer a respectful note of thanks to **Dr. K.K.PADMANABAN, Ph.D.,** Principal of the college, for the excellent facilities made available to accomplish this project work.

The author would like to deem it a privilege to record his sincere thanks to **Prof.S.THANGASAMY, Ph.D.,** Head of the Department of Computer Science & Engineering for his valuable suggestions and motivations during the entire period of this course.

The author would like to express his heartfelt gratitude to his guide **Mrs. L.S.JAYASHREE M.E,** for her valuable guidance, suggestions, and consistent encouragement, which lead to the successful completion of the project.

The author is very much indebted to his class advisor, **MR.R.BASKAR B.E, M.S** for his motivations during the entire period of the project work.

Last but not the least, the author thanks all the teaching and non teaching staff members of the college and his friends for their kind cooperation throughout this project work.

A. SENTHIL NATHAN.

SYNOPSIS

Internet telephony is the transport of telephone calls over the Internet (IP based networks).It is the use of Internet Technology to replace a long distance or international provider of traditional telephone service.

Support for voice communications using the Internet Protocol (IP), which is usually just called "**Voice over IP**" or VoIP, has become especially attractive given the low-cost, flat-rate pricing of the public Internet. VoIP can be defined as the ability to make telephone calls and to send facsimiles over IP-based data networks rather than the PSTN circuits at a much superior cost/benefit with a minimal Quality of service. Since data traffic is growing much faster than telephone traffic, there has been considerable interest in transporting voice over data networks. The main task taken here is to set a frame work to make communication possible using computer as an interface. To make this communication possible the voice Signal has to be digitized, get compressed in order to meet the bandwidth demands, made into IP packets and passed over IP based networks. The Packets upon reaching the destination gets decompressed to give-up the original Voice signal. Linear Predictive coding (LPC) compression is used to compress/decompress the voice signals at a low bit rate channel. The project includes an open standard protocol suite, H.323 which is an International Telecommunications Union (ITU) standard that provides specification for computers, equipment, and services for multimedia communication over networks. Support for H.323 standard conferencing is enabled by telephony application programming interfaces (TAPI).The communication takes place between PCs via IP networks.

Contents

Page no

1. INTRODUCTION

1.1 Introduction to Voice over Internet Protocol

Voice over Internet protocol–or in its broader meaning VoIP –is one of the fastest growing segments of the computer and telecommunication industry. VoIP has succeeded in many areas fundamental to the way we communicate

- By integrating existing company investments, such as computers, telephone systems data networks.
- Delivering advanced and productive communication features like document sharing White boarding and video
- Bringing massive management and administrative potential
- The advantages of flexible, productive and effective IP networks.

Combine the telephone and the internet and you've got the beginning of the biggest productivity revolution in decades. The telephone has not changed in 30 years …with the arrival of the VoIP we can sure it will.

1.1.1 Benefits:

In short term communication across the internet offers business a clear and tangible competitive edge in many areas:

- Bringing immediate cost reduction
- Improved profitability and productivity
- Unprecedented levels of manageability.

Whether by reducing international voice and fax accounts with a gateway solution or enhancing inter branch office communication with the internet phone or speeding up the whole process of business with the real time collaborative application

1.1.2 Advances in voice technology:

Over a period of two years, the industry has solved many of the sound quality problems to produce a clear and crisp voice signal often comparable to that of a regular call. The technology has evolved in the following main areas:

- Successfully bridging the gap between data networks (internet, intranet) the traditional circuit switch network(PSTN)with the arrival of voice and fax gateway.
- Development of powerful compression/decompression algorithms.
- Dedicating all processing to an industry voice processing board significantly Improving voice quality.
- Ensemble architecture, carrier –Grade and service provider platforms with industry Standard protocols.

VoIP can mean a number of different things. It can mean the use of Internet Technology to replace a long distance or international provider of traditional telephone service, or an enhanced form of human to human communication based on the computer as the user interface, rather than the telephone.

The purpose of organizing telephony applications into classes is that it provides a frame work around which to speculate on the broader implications of VoIP. The different classes of VoIP have very different justifications, and very different implications for the relevant industrial sectors involved, as well as policy makers and users. Some telephony applications are focused on a short term cost savings strategy, which may not have strong long term market viability.

However, a possible long-term outcome of the VoIP evolution is that people use computers rather telephones to communicate. This outcome, were it to happen, could trigger a major restricting of the telephone industry, in which separate firms provide the low-level physical connectivity and Internet service, and the higher

Level telephone service itself. The final speculative form of telephony described above is not practical today, because the necessary supporting features internet are not in place. It is our hypothesis that VoIP will evolve as a series of incremental steps.

Early variants of VoIP will be identified that can be deployed without first requiring as much enhancement of the Internet. These offerings will serve as experiments to prove the market, evaluate demand, explore the desirability of features, and motivate the fuller deployment of Internet service.

We begin by proposing a first basis for classification, which is how much interoperations required between the Internet and the existing telephone system or PSTN. We then describe in greater detail what is implied by the long term vision of telephony alluded to the above. By listing the critical features and requirements by which we can organize the different telephony applications.

We will show that there are applications with different mixes of these features and requirements, which in turn suggests a possible evolutionary path for the future. How much PSTN? The important question. The most significant distinction between the various Internet telephony applications is the question of how much PSTN and how much computer-based telephony is in the scheme. Here we identify three important classes of telephony applications.

Class I: proposals with the goal of using Internet to provide POTS telephony between existing telephone end-user equipment. Applications of this class require technology for interconnection between the PSTN and Internet networks, but do not require access to computer-based end-nodes, and can often operate across dedicated regions of the Internet.

Class II: proposals that require interoperation between the existing telephone and Internet networks, and provide communication between users with either computers or existing telephone sets as end nodes. This class requires both the interoperation between Internet and PSTN, as well as the use of computer based networks.

Class III: proposals that use Internet –attached computers to provide some form of human communication across the PSTN-switched Internet. This class, the pure form of Internet –based communication, does not involve any aspect of PSTN

Interaction or internetworking with telephone end-nodes.

1.2.1 Conventional telephony

When you make a telephone call, the telephone exchange

Establishes an exclusive connection to the number you dial. While you're having the conversation, anybody else trying to dial either party will get an 'engaged' tone. That's essentially what a circuit-switched Network does. It establishes an exclusive and continuous physical connection between two parties. Circuit-switched technology itself has evolved quite a bit. It started with cord board switches where an operator manually connected two parties through a cord. From there, it moved on to SxS systems, also called the Strowger exchanges, named after its inventor. These were electromechanical in nature. After this came the crossbar exchanges, which are still being used in many Countries.Crossbar used electromagnetic principles. Today, electronic telephone exchanges are fairly common, which are more compact and powerful, and convert voice into digital signals for transmission.

Digitizing sound is an interesting process. When we make a telephone call, our voice is first converted into analog electrical signals. This signal is then encoded into digital format using a technique called PCM (Pulse Code Modulation). This technique takes samples of the analog signals at a rate of 8,000 samples per second. Each sample therefore represents 125 microseconds of a voice stream, and is eight bits, or one byte long. This signal is then carried over high-speed digital lines and again decoded into an analog electrical signal at the receiving end. The analog signal is finally converted into the original sound.

Speaking of sound, any conversation consists of two components-sound and silence. When the digital sound signals are transmitted over a circuit switched network, both components have to be sent. Not only that, but the order of transmitting signals also has to be retained, else quality of transmission suffers. That's why all equipment in a circuit-switched network must be highly synchronized using expensive TDM (Time Division Multiplexing) equipment. Since sound and silence are both transmitted, a lot of

bandwidth gets wasted in circuit-switched networks. In fact, one voice conversation requires a 64 kbps channel, which is quite a lot of bandwidth.

Description	PSTN	Internet
Designed for	Voice only	Packetized data, voice & video
Bandwidth Assignment	64Kbps (dedicated)	Full-line bandwidth over a period of time
Delivery	Guranteed	Not guaranteed
Delay	5-40ms (distance-dependent)	Not predictable (usually more than PSTN)
Cost for the Service	Per-minute charges: long distance; Monthly flat rate: local access	Montly flat rate for access
Voice Quality	Toll quality	Depends on customer equipment
Connection Type	Telephone, PBX, switches with frame relay and ATM backbone	Modem, T1/E1, Gateway, Switches, ISDN, bridges, Routers, Backbone
Quality of Service	Real-time delivery	Not real-time delivery
Network Management	Homogeneous and interoperable at network and user level	Various styles with interoperability established at network layer only
Network Characteristics (Hardware)	Switching systems for assigned bandwidth	Routers & bridges for layer 3 and 2 switching
Network Characteristics (Software)	Homogeneous	Various interoperable software systems
Access Points	Telephones, PBX, PABX, ISDN, switches, high-speed trunks	Modem, ISDN, T1/E1 Gateway, high-speed DSL/cable modems

1.2.2 How VoIP will make your calls cheaper

The telephone network and the Internet are two different networks with minimal interconnects. Both use different technologies too. But that could soon become a thing of the past. It is now possible to send voice over the Internet, using the same technologies that your PC uses. When this gets going, the telephone would be replaced by a PC-based device, and voice calls would cost a fraction of what they do today the telephone is the most common of today's communication facilities.

It is the one that has been around the longest. It is also the enabler of the Internet. That is, you use telephone lines (dial-up, ISDN, etc.) to access the Internet. It would therefore only be natural for us to assume that the familiar telephone would be around for all time to come.

Nothing, however, could be further from the truth. A silent revolution that is taking place on telephone networks the world over could make the telephone of today useless and extinct.

That revolution is Voice over IP, VoIP for short. Today you have to maintain two separate networks for voice and data. Also, the charges for voice transmission are way higher than those for data transmission. Once the two combine over IP you will be able to make voice calls virtually for free and telephone companies will need to maintain only one network. In short, the telephone of today will be replaced by an IP-based voice device.

First there were the telephone networks owned by different telephone companies. These networks spanned the globe and enabled you to call up friends and relatives across the world. Then came computer networks that evolved into the Internet. One of the features of this computer network was that you did not have to be always connected, but could connect to it when required. And what better way was there to connect than to use the already present telephone line? Thus there was a considerable degree of interaction between the voice and data networks. But the two networks remained essentially separate, evolving along their separate paths. Also, while computer-to-computer communication was far cheaper, voice communication continued to remain expensive.

One of the main reasons for these is the basic difference in the technologies used for voice and data communication. Voice networks use circuit switching while data networks use Packet switching. What this means is that for a voice conversation to happen, one complete and continuous circuit has to be held open between the two parties during the full duration of the conversation, while for data transmission such a continuous connect is unnecessary. In data transmission, the data is split up into independent packets that can take alternate routes to the intended destination, where they are reassembled. In voice communication, circuit switching is used because it is able to handle data in real time. Packet switching, on the other hand, can lead to delays, which, though small, can lead to considerable degradation in the quality of the conversation.

So, the two networks went their separate ways till the Real time Transport Protocol (RTP) was developed. Running on top of UDP (User Datagram Protocol, that runs on IP networks and is primarily used to broadcast messages on the network), RTP provides support for streaming audio and video over computer networks.

Unlike what its name seems to suggest, RTP does not ensure real time delivery of data. Instead it provides mechanisms for time stamping the packets and methods for synchronizing data streams with time properties. Thus it became technically possible to send voice over the Net. But the question of quality remained. Voice traveling over congested data networks is as good as no voice at all. To avoid this, a certain minimal quality of service has to be ensured. This can be done by having more bandwidth all around and by providing for primacy for voice in the various routers on the Net. That is, if a voice packet arrives at a router, then it has to be routed in preference to a data packet. This requires routers to be voice aware. If both of these are done (and they are in the process of being done) then there is no reason why voice and data cannot be transmitted on the same network. In fact that is also happening. Many telecom networks, the world over, including at least two in India, are now IP based. Then why is your telephone not getting replaced by your PC? In fact it can be replaced, but for regulations decreeing otherwise .But such limitations can be enforced only for so long. The time is coming when VoIP will completely replace the current circuit switched telephony.

How will the system work then? Would you still require a separate Internet account and a separate telephone line?

To find an answer, consider the case of electricity. When you build a new building, you apply for an electric connection. The connections are classified on the load you are going to draw from the system into domestic, low tension, high tension, and so on. You get one electricity point at your building that you can extend to all parts of the building with multiple outlets. The utility is not concerned with how many plug points you create or whether you connect electric irons, TV sets or dish washers to the outlets, as long as you do not draw more than your rated load. Also, electricity is always available (or at least it is supposed to be!). You do not have to buy some special equipment and dial up every time you want to use some equipment that uses electricity.

Finally you do not pay one rate for electric irons and another for TV sets. Nor do you pay by the hour, or by how far away you are from the power house. You pay according to your rated load and the power you use up.

1.3.1 Importance of VoIP

This new addition to the ITU Internet Reports series looks at the topic. Voice over Internet Protocol (VoIP) is rapidly reaching the top of the agenda for the telecommunications industry worldwide. The key issue that has gained the attention of policy-makers, regulators, and industry alike is that the Internet, and other IP-based networks, are increasingly being used as alternatives to circuit-switched telephone networks. The many different `flavors' of VoIP provide, to varying degrees, alternative means of originating, transmitting, and terminating voice and data transmissions that would otherwise be carried by the public switched telephone network (PSTN). In many countries it is now possible, using a standard telephone, to call almost any other telephone in the world by means of VoIP. By 2004, this could account for up to 40 per cent of all international traffic. Because these calls are mainly carried outside of the PSTN, they are also outside the regulatory and financial structures that have grown up around

1.3.2 Future of Voice-over-Internet Protocol (VoIP)

Several factors will influence future developments in VoIP products and services. Currently, the most promising areas for VoIP are corporate intranets and commercial extranets. Their IP based infrastructures enable operators to control who can and cannot use the network.

Another influential element in the ongoing Internet-telephony evolution is the VoIP gateway. As these gateways evolve from PC based platforms to robust embedded systems, each will be able to handle hundreds of simultaneous calls. Consequently, corporations will deploy large numbers of them in an effort to reduce the expenses associated with high-volume voice, fax, and videoconferencing traffic. The economics of placing all traffic data, voice, and video over an IP based network will pull companies in this direction, simply because IP will act as a unifying agent, regardless of the underlying architecture (i.e., leased lines, frame relay, or ATM) of an organization's network.

Commercial extranets, based on conservatively engineered IP networks, will deliver VoIP and facsimile over Internet protocol (FAXoIP) services to the general public. By guaranteeing specific parameters, such as packet delay, packet jitter, and service interoperability, these extranets will ensure reliable network support for such applications.

VoIP products and services transported via the public Internet will be niche markets that can tolerate the varying performance levels of that transport medium. Telecommunications carriers most likely will rely on the public Internet to provide telephone service between/among geographic locations that today are high-tariff areas. It is unlikely that the public Internet's performance characteristics will improve sufficiently within the next two years to stimulate significant growth in VoIP for that medium.

However, the public Internet will be able to handle voice and video services quite reliably within the next three to five years, once two critical changes take place:

- an increase by several orders of magnitude in backbone bandwidth and access speeds, stemming from the deployment of IP/ATM/synchronous optical network (SONET) and ISDN, cable modems, and x digital subscriber line (xDSL) technologies, respectively
- the tiering of the public Internet, in which users will be required to pay for the specific service levels they require

On the other hand, FAXoIP products and services via the public Internet will become economically viable more quickly than voice and video, primarily because the technical roadblocks are less challenging. Within two years, corporations will take their fax traffic off the PSTN and move it quickly to the public Internet and corporate Intranet, first through FAXoIP gateways and then via IP capable fax machines. Throughout the remainder of this decade, videoconferencing (H.323) with data collaboration (T.120) will become the normal method of corporate communications, as network performance and interoperability increase and business organizations appreciate the economics of telecommuting.

2. LITERATURE SURVEY

2. LITERATURE SURVEY

- **PcQuest magazine:** *how voice goes over IP,* April 2001

The article gives the technical details about the working of voice over IP and how it is different from the working principles of PSTN circuits. It also provides the advantage of moving towards the new technology and the need to move towards the new technology.

- **Openh323.org**

The OpenH323 project aims to create a full featured, interoperable, Open Source implementation of the ITU H.323 teleconferencing protocol that can be used by personal developers and commercial users without charge

- **Cisco.com:** ***Voice Quality in Converging Telephony and IP Networks***

The site gives the convergence of voice and data packet units and the challenges delay echo that are needed to pass when moving towards voice over IP. It suggests the importance of VoIP and procedural strategies on developing the VoIP products.

- **NETWORK COMPUTING MAGAZINE**: *BUILDING VOICE OVER IP*

Philip carden. May 2000

H.323 vs. SIP

Given that two standards currently compete for the dominance of IP telephony signaling, how do you decide which is more appropriate? The good news is that the two protocol suites appear to be converging — picking up good ideas from one another. In particular, the third and latest incarnation of H.323 (H.323 v3) has addressed some key performance issues (call setup delay and stateless processing to support UDP), which were initially key SIP advantages.

Quality of Service and Management	Call setup delay, packet loss recovery, lack of resource reservation capability	Fault tolerance (H.323 supports redundant gatekeepers and endpoints), Admission Control (SIP relies on other protocols for bandwidth mgmt, call mgmt and bandwidth control), Policy Control (H.323 has limited DiffServ support vs. none for SIP)	Loop detection (SIP's algorithm using "via header" somewhat superior to H.323's PathValue approach)
Scalability and Flexibility	Stateless processing, UDP Support, Inter-server communications for endpoint location	Location of endpoints in other administrative domains (SIP does not define a method, but suggests use of DNS)	Complexity (SIP is less complex), Extensibility (SIP's use of hierarchical feature names and error codes which can be IANA registered is more flexible than H.323's vendor-specific single extension field "NonStandardParam"), Ease of customization (SIP less complex, and offers text-based protocol encoding)
Interoperability		PSTN Signaling Interoperability (SIP Internet Draft only, H.323 uses Q.931-like messages, which are SS7 compatible, though only a subset of ISUP messages), Inter-vendor interoperability (H.323 more mature, greater interoperability track	

		record, IMTC iNOW! profile to assist implementation)	

The following table summarizes the compression options available.

Compression	Bytes per second	Kilobits per second	Need fast CPU?	Sound Fidelity
No compression	8000	80000	No	Best
Simple	4000	40000	No	Poor
ADPCM	4000	40000	No	Good
Simple + ADPCM	2000	20000	No	Lousy
GSM	1650	16500	Yes	Good
Simple + GSM	825	8250	Yes	Lousy
LPC	650	6500	Yes	Depends
LPC-10	346	3460	Extremely	Okay

3. PROPOSED LINE OF ATTACK

3. PROPOSED LINE OF ATTACK

The methodology of developing a VoIP product will include following functional modules.

Connection establishment module

-winsock control

Compression/decompression module

-linear predictive coding

Communication module

-Telephony application programming Interface (TAPI)

-H.323 protocol suite

3.1 Connection establishment module using winsock control

Initial connection has to be established to make the communication across nodes in a network. For sending real audio UDP connection is used.

Applications that send and receive datagrams are generally connectionless. The identification of each end point in the conversation is established each time data is sent.

UDP/IP based socket is used for sending and receiving live audio and video. UDP connections reduce delay as because it does not provide any acknowledgements to sender upon receiving the data. It also provides fault tolerance features as it could maintain the quality of service if it receives lesser number of packets.

3.2 Compression/Decompression Module

In order to meet the bandwidth demands, the voice signals need to be compressed to travel over the IP based networks. The MTU (Maximum Transfer Unit) may vary in as because the packet crosses many networks and least MTU may lead to poor quality and adds additional delay.

Linear predictive coding (LPC) is one of the most powerful speech analysis techniques, and one of the most useful methods for encoding good quality speech at a low bit rate. It provides extremely accurate estimates of speech parameters,

and is relatively efficient for computation. The basic solution is a difference equation which expresses each sample as a linear combination of previous sample. Such a equation is called a linear predictor, which is why this is called Linear predictor coding. The linear predictive coding (LPC) model is based on a mathematical approximation of the vocal tract represented by this tube of a varying diameter. At a particular time, *t*, the speech sample *s(t)* is represented as a linear sum of the *p* previous samples. The most important aspect of LPC is the linear predictive filter which allows the value of the next sample to be determined by a linear combination of previous samples. Under normal circumstances, speech is sampled at 8000 samples/second with 8 bits used to represent each sample. This provides a rate of 64000 bits/second. Linear predictive coding reduces this to 2400 bits/second. At this reduced rate the speech has a distinctive synthetic sound and there is a noticeable loss of quality. However, the speech is still audible and it can still be easily understood. Since there is information loss in linear predictive coding, it is a lossy form of compression.

The LPC filter equation is generally given as,

$$y[n] = a[1]y[n-1] + a[2]y[n-1] + a[3]y[n-3] + e[n]$$

3.3 Communication module

- **TELEPHONY APPLICATION PROGRAMMING INTERFACE (TAPI)**

TAPI 3.0 integrates multimedia stream control with legacy telephony. Additionally, it is an evolution of the TAPI 2.1 API to the COM model, allowing TAPI applications to be written in any language, such as r Microsoft Visual C++.

Besides supporting classic telephony providers, TAPI 3.0 supports standard H.323 conferencing and IP multicast conferencing. and it supports quality-of-service (QoS) features to improve conference quality and network manageability. The project is incorporated with many of TAPI functions.

- **H.323 protocol suite**

H.323 is a comprehensive International Telecommunications Union (ITU) standard for multimedia communications (voice, video, and data) over connectionless networks that do not provide a guaranteed quality of service, such as IP-based networks and the Internet. It provides for call control, multimedia management, and bandwidth management for point-to-point and multipoint conferences. H.323 mandates support for standard audio and video codecs and supports data sharing through the T.120 standard. Furthermore, the H.323 standard is network-, platform-, and application-independent, allowing any H.323-compliant terminal to interoperate with any other. The coding segment consists many protocol specific functions needed for the transmission.

4. DETAILS OF PROPOSED METHODOLOGY

4. DETAILS OF PROPOSED METHODOLOGY

The VoIP is implemented by three different modules which are listed below; with each module have its own set of goals and the output data of each module is fed as the input to the next module

- Compression/decompression module
- Connection establishment module
- Communication module and multithreaded module

4.1 Compression Decompression Module

The Internet on the other hand, is a digital packet –switched network. this means that your computer has to translate your voice into digital signals, then wrap those signals Into little packages so they travel on the digital highway. sound is not continuously traveling back and forth from computer to computer ,as it does from telephone to telephone.

When we use the internet to transmit voice, little sound package has to travel through a maze of different computers, router, and servers before it reaches the final destination. In order to make the conversation sound like telephone conversation as possible, the compression algorithm taken here reduces delay.

4.1.1 Linear Predictive Coding (LPC)

4.1.2 Introduction

Linear Predictive Coding (LPC) is one of the most powerful speech analysis techniques, and one of the most useful methods for encoding good quality speech at a low bit rate. It provides extremely accurate estimates of speech parameters, and is relatively efficient for computation.

4.1.3 Basic Principles

LPC starts with the assumption that the speech signal is produced by a buzzer at the end of a tube. The glottis (the space between the vocal cords) produces the buzz, which is characterized by its intensity (loudness) and frequency (pitch). The vocal tract (the throat and mouth) forms the tube, which is characterized by its resonances, which are called *formants*.

LPC analyzes the speech signal by estimating the formants, removing their effects from the speech signal, and estimating the intensity and frequency of the remaining buzz. The process of removing the formants is called *inverse filtering*, and the remaining signal is called the *residue*.

The numbers which describe the formants and the residue can be stored or transmitted somewhere else. LPC synthesizes the speech signal by reversing the process: use the residue to create a source signal, use the formants to create a filter (which represents the tube), and run the source through the filter, resulting in speech.

Because speech signals vary with time, this process is done on short chunks of the speech signal, which are called *frames*. Usually 30 to 50 frames per second give intelligible speech with good compression.

4.1.4 Estimating the Formants

The basic problem of the LPC system is to determine the formants from the speech signal. The basic solution is a difference equation, which expresses each sample of the signal as a linear combination of previous samples. Such an equation is called a *linear predictor*, which is why this is called Linear Predictive Coding.

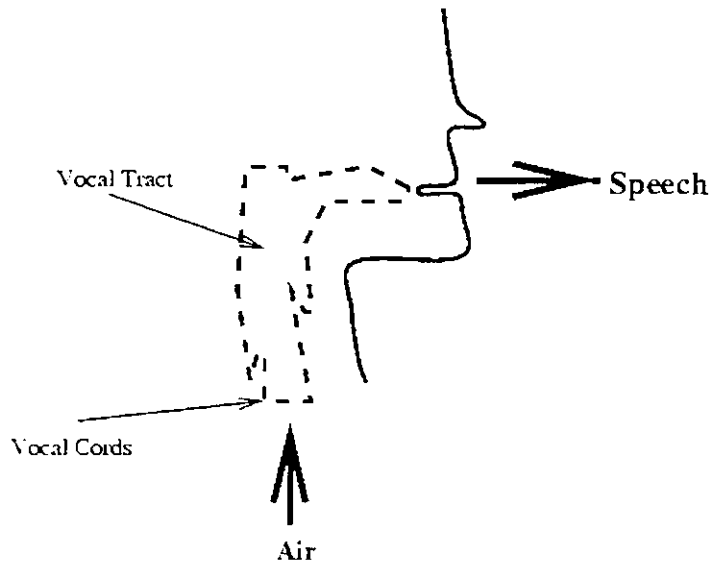
The coefficients of the difference equation (the *prediction coefficients*) characterize the formants, so the LPC system needs to estimate these coefficients. The estimate is done by minimizing the mean-square error between the predicted signal and the actual signal.

This is a straightforward problem, in principle. In practice, it involves (1) the computation of a matrix of coefficient values, and (2) the solution of a set of linear

equations. Several methods (autocorrelation, covariance, recursive lattice formulation) may be used to assure convergence to a unique solution with efficient computation

4.1.5 LPC Modeling

A. Physical Model:

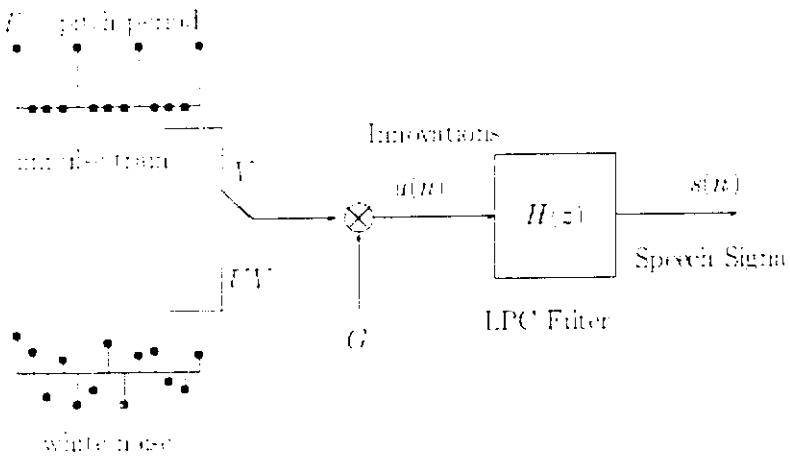


When you speak:

- Air is pushed from your lung through your vocal tract and out of your mouth comes speech.
- For certain *voiced* sound, your vocal cords vibrate (open and close). The rate at which the vocal cords vibrate determines the *pitch* of your voice. Women and young children tend to have high pitch (fast vibration) while adult males tend to have low pitch (slow vibration).
- For certain *fricatives and plosive (or unvoiced)* sound, your vocal cords do not vibrate but remain constantly opened.
- The shape of your vocal tract determines the sound that you make.

- As you speak, your vocal tract changes its shape producing different sound.
- The shape of the vocal tract changes relatively slowly (on the scale of 10 msec to 100 msec).
- The amount of air coming from your lung determines the loudness of your voice.

B. Mathematical Model:



- The above model is often called the LPC Model.
- The model says that the digital speech signal is the output of a digital filter (called the LPC filter) whose input is either a train of impulses or a white noise sequence.
- The *relationship* between the physical and the mathematical models:

$$\text{Vocal Tract} \Longrightarrow H(z) \text{ (LPC Filter)}$$

$$\text{Air} \Longrightarrow u(n) \text{ (Innovations)}$$

$$\text{Vocal Cord Vibration} \Longrightarrow V \text{ (voiced)}$$

$$\text{Vocal Cord Vibration Period} \Longrightarrow T \text{ (pitch period)}$$

Fricatives and Plosives $= UV$ (unvoiced)

Air Volume $= G$ (gain)

4.1.6 LPC Analysis/Encoding

Voiced/Unvoiced Determination

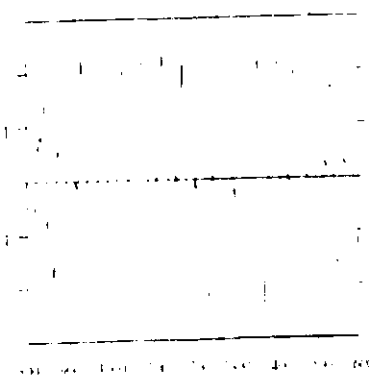
STEP 1: If the amplitude levels are large then the segment is classified as voiced and if they are small then the segment is considered unvoiced.

• This determination requires a preconceived notation about the range of amplitude values and energy levels associated with the two types of sound.

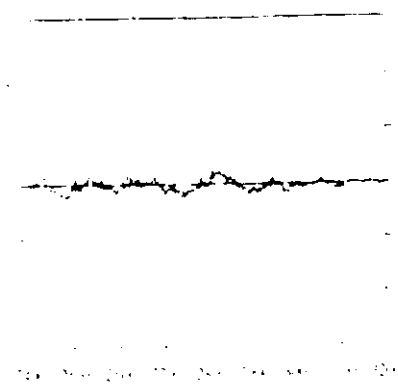
PITCH PERIOD ESTIMATION

STEP 2: This step takes advantage of the fact that:

1) Voiced speech segments have large amplitudes



2) Unvoiced speech segments have high frequencies



3) The average values of both types of speech samples is close to zero.

• These three facts lead to the conclusion that the unvoiced speech waveform must cross the x-axis more often then the waveform of voiced speech. We can

$$\mathbf{S} = (s(0), s(1), \ldots, s(159))$$

Thus the 160 values of $\mathbf{S}$ is compactly represented by the 13 values of $\mathbf{A}$.

- There's almost no perceptual difference in $\mathbf{S}$ if:
    - **For Voiced Sounds (V):** the impulse train is shifted (insensitive to phase change).
    - **For Unvoiced Sounds (UV):** a different white noise sequence is used.

- **LPC Synthesis:** Given $\mathbf{A}$, generate $\mathbf{S}$ (this is done using standard filtering techniques).
- **LPC Analysis:** Given $\mathbf{S}$, find the best $\mathbf{A}$ (this is described in the next section).

LPC Analysis

- Consider one frame of speech signal:

$$\mathbf{S} = (s(0), s(1), \ldots, s(159))$$

- The signal $s(n)$ is related to the innovation $u(n)$ through the linear difference equation:

$$s(n) + \sum_{i=1}^{10} a_i s(n-i) = u(n)$$

- The ten LPC parameters $(a_1, a_2, \ldots, a_{10})$ are chosen to minimize the energy of the innovation:

$$f = \sum_{n=0}^{159} u^2(n)$$

- Using standard calculus, we take the derivative of f with respect to a_i and set it to zero:

$$\begin{aligned} df/da_1 &= 0 \\ df/da_2 &= 0 \\ &\cdots \\ df/da_{10} &= 0 \end{aligned}$$

- We now have 10 linear equations with 10 unknowns:

$$\begin{bmatrix} R(0) & R(1) & R(2) & R(3) & R(4) & R(5) & R(6) & R(7) & R(8) & R(9) \\ R(1) & R(0) & R(1) & R(2) & R(3) & R(4) & R(5) & R(6) & R(7) & R(8) \\ R(2) & R(1) & R(0) & R(1) & R(2) & R(3) & R(4) & R(5) & R(6) & R(7) \\ R(3) & R(2) & R(1) & R(0) & R(1) & R(2) & R(3) & R(4) & R(5) & R(6) \\ R(4) & R(3) & R(2) & R(1) & R(0) & R(1) & R(2) & R(3) & R(4) & R(5) \\ R(5) & R(4) & R(3) & R(2) & R(1) & R(0) & R(1) & R(2) & R(3) & R(4) \\ R(6) & R(5) & R(4) & R(3) & R(2) & R(1) & R(0) & R(1) & R(2) & R(3) \\ R(7) & R(6) & R(5) & R(4) & R(3) & R(2) & R(1) & R(0) & R(1) & R(2) \\ R(8) & R(7) & R(6) & R(5) & R(4) & R(3) & R(2) & R(1) & R(0) & R(1) \\ R(9) & R(8) & R(7) & R(6) & R(5) & R(4) & R(3) & R(2) & R(1) & R(0) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \\ a_9 \\ a_{10} \end{bmatrix} = \begin{bmatrix} - \\ - \\ - \\ - \\ - \\ - \\ - \\ - \\ - \\ - \end{bmatrix}$$

where

$$\begin{aligned} R(k) &= \sum_{n=0}^{159-k} s(n)s(n+k) \\ &= \text{autocorrelation of } s(n) \end{aligned}$$

- The above matrix equation could be solved using:
  - The Gaussian elimination method.
  - Any matrix inversion method (MATLAB).
  - The Levinson-Durbin recursion (described below).

- **Levinson-Durbin Recursion:**

$$\begin{aligned} E^{(0)} &= R(0) \\ k_i &= [R(i) - \textstyle\sum_{j=1}^{i-1} \alpha_j^{(i-1)} R(i-j)]/E^{(i-1)} \qquad i = 1, 2, \ldots, 10 \\ \alpha_i^{(i)} &= k_i \\ \alpha_j^{(i)} &= \alpha_j^{(i-1)} - k_i \alpha_{i-j}^{(i-1)} \qquad j = 1, 2, \ldots, i-1 \\ E^{(i)} &= (1 - k_i^2) E^{(i-1)} \end{aligned}$$

Solve the above for $i = 1, 2, \ldots, 10$, and then set

$$a_i = -\alpha_i^{(10)}$$

- To get the other three parameters: $(V/UV, G, T)$, we solve for the innovation:

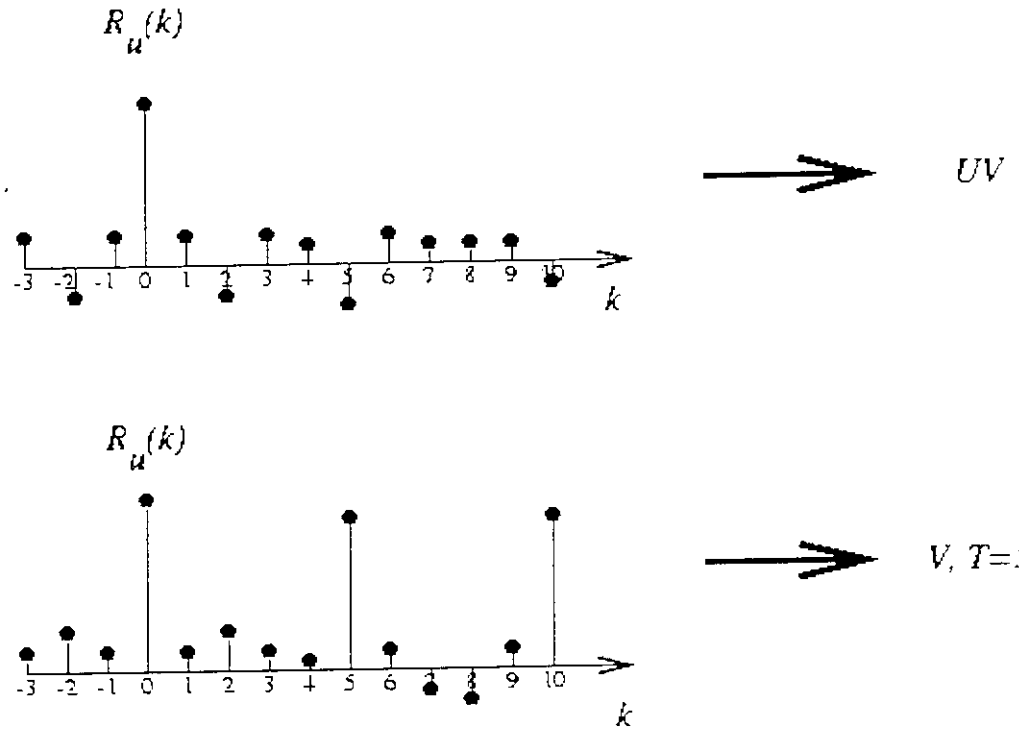
$$u(n) = s(n) + \sum_{i=1}^{10} a_i s(n-i)$$

- Then calculate the autocorrelation of $u(n)$:

$$R_u(k) = \sum_{n=0}^{159-k} u(n)u(n+k)$$

- Then make a decision based on the autocorrelation:

H.323 and Related Recommendations	
Recommendation	**Brief Description**
H.323	Document called "Visual telephone systems and equipment for local area networks which provide a non-guaranteed quality of service" (November, 1996)
H.225	Call control messages including signaling, registration, and admissions, and for the packetization and synchronization of media streams including both point-to-point and multipoint calls
H.245	Messages for opening and closing channels for media streams, and other commands, requests, and indications
H.261	Video codec for audio visual services at multiples of 64 Kbps
H.263	Specifies a codec for video over the PSTN
G.711	Audio codec for 3.1 Kbps bandwidth over 48,56, and 64 K bps channels (normal telephony)
G.722	Audio codec for 7 Kbps bandwidth over 48,56, and 64 Kbps channels
G.728	Audio codec for 3.1 Kbps bandwidth over 16 Kbps channels
G.723, G.723.1	Audio codec for 3.1 Kbps bandwidth over 5.3 and 6.3 Kbps channels (G.723.1 has been selected by the VoIP Forum for use with VoIP)
G.729, G.729a	Audio codec for 3.1 Kbps bandwidth over 8 Kbps channels (adopted by the Frame Relay Forum for voice over Frame Relay)
T.120	Data and conference control



Problem: During the quantization of the filter coefficients for transmission there is an opportunity for errors in the filter coefficients that can lead to instability in the vocal tract filter and create an inaccurate output signal.

• *Solution*: During the L-D algorithm which computes the filter coefficients {ai} a set of coefficients, {ki}, alled reflection coefficients are also generated.

These coefficients can be used to rebuild the set of filter coefficients {ai} and can guarantee a stable filter if their magnitude is strictly less than one.

• The encoder sends a single bit to tell if the current segment is voiced or unvoiced.

• For voiced segments, the pitch period is quantized using a log-companded quantizer to one of 60 possible values.

• If the segment contains voiced speech than a 10^{th} order filter is used. This means that 11 values are needed: 10 reflection coefficients and the gain.

• If the segment contains unvoiced speech than a 4^{th} order filter is used. This means that 5 values are needed: 4 reflection coefficients and the gain.

• The reflection coefficients are denote kn where 1 <= n <= 10 for voiced speech filters and 1 <= n <= 4 for unvoiced filters.

• The reflection coefficients, kn , cause the vocal tract filter to be especially sensitive to errors if they have a magnitude close to one.

• The first few reflection coefficients, k1 and k2, are the most likely coefficients to have magnitudes ~0

• LPC-10 uses nonuniform quantization for k1 and k2.

• First, each coefficient is used to generate a new coefficient, gi of the form:

$$g_i = \frac{1 + k_i}{1 - k_i}$$

• These new coefficients, g1 and g2, are then quantized using a 5-bit uniform quantizer.

• All other reflection coefficients are quantized using uniform quantizers

• k3 and k4 are quantized using 5-bit uniform quantization

• For voiced segments k5 up to k8 are quantized using a 4-bit uniform quantizer

• For voiced segments k9 uses a 3-bit quantizer

• For voiced segments k10 uses a 2-bit uniform quantizer

• For unvoiced segments the bits used to represent the reflection coefficients, k5 through k10, in voiced segment are used for error protection.

• Once the reflection coefficients for a filter have been quantized, the only thing left to quantize is the gain.

• The gain, G, is calculated using the root mean squared (rms) value of the current segment.

• The gain is quantized using a 5-bit log-companded quantizer.

Transmitting Parameters

1 bit voiced/unvoiced

6 bits pitch period (60 values)

10 bits k1 and k2 (5 each)

10 bits k3 and k4 (5 each)

16 bits k5, k6, k7, k8 (4 each)

3 bits k9

2 bits k10

5 bits gain G

1 bit synchronization

54 bits TOTAL BITS PER FRAME

Verification of LPC Bit Rate

Sample rate = 8000 samples/second

Samples per segment = 180 samples/segment

Segment rate = Sample Rate/Samples per Segment

= (8000 samples/second)/(180 samples/second)

= 44.444444.... segments/second

Segment size = 54 bits/segment

Bit rate = Segment size * Segment rate

= (54 bits/segment) * (44.44 segments/second)

= 2400 bits/second

4.1.7 LPC Synthesis/Decoding

Generating the Excitation Signal

- *STEP 1*: Determine if speech segment is voiced or unvoiced from voiced/unvoiced bit
- Voiced speech segments use a locally stored pulse consists of 40 samples
- Unvoiced speech segments use white noise produced by a pseudorandom number generator
- *STEP 2*: Determine pitch period for voiced speech Segments
- The pitch period for voiced segments is used to determine whether the 40 sample pulse used as the excitation signal needs to be truncated or extended.
- If the pulse needs to be extended it is padded with zeros since the definition of a pulse is "...an isolated disturbance, that travels through an otherwise undisturbed medium"

Building the Filter

- The decoder builds a filter to generate the speech when given the excitation signal as input.
- 10 reflection coefficients are transmitted for voiced segment filters and 4 reflection coefficients are used for unvoiced segments.
- These reflection coefficients are used to generate the vocal tract parameters
- The vocal tract parameters are used to create the filter.

Generating the Output Signal

• The excitement signal is passed through the filter to produce a synthesized speech signal that accurately represents the original speech.

• Each segment is decoded individually and the sequence of reproduced sound segments is joined together to represent the entire input speech signal

4.2 Connection Establishment Module

4.2.1 Socket Programming Using Winsock

Winsock is a network application programming interface(API) for Microsoft windows -a well defined set of data structures and function calls implemented as a dynamic link library (DLL).

A group of over network providers have agree upon winsock as an open standard, Which gives the application programmer a single interface to network functionality? It clearly defines which duties are assigned to the network vendors software and which ones are left to the application.

Winsock is a translator of sorts, In your application, you make function calls requesting generic network services; Winsock translates these generic requests into protocol specific requests and perform the necessary tasks. Residing between your application and the network implementation, winsock shields you from the details of low level network protocols

4.2.2 Sockets Paradigm

The group responsible for creating winsock made some good decisions early in the process –the most important of which was to follow the sockets model. First introduced in the Berkeley UNIX software distribution in the early 1980's.The concept of sockets was originally designed as a method of interprocess communication. Soon, however, it grew into encompass network communications as

well. The Berkeley sockets module conceptualize network communications as taking place between two endpoints or sockets .Analogies have been drawn between Plugging an application into a network and plugging a handset into the telephone system, or an appliance into an electrical system.

A socket defines one endpoint in the communication between two processes. Before communication can start, two sockets must exist: Sending socket and receiving socket.

4.2.3 The Client Server Model

Rather than trying to start the two network application simultaneously, one application is theoretically always available (the server) and another requests services as needed (the client)

The server creates a socket, names it so that it can be identified and found by a client, and then listens for service requests. A client application creates a socket, finds a Server socket by a name or address and then plugs to initiate a conversation. Once a conversation is initiated, data can be sent. Multiple clients can connect to the server .A client can sent a compressed data to another client and receive decompressed data from that client. The communication takes place between clients through the services given by the client.

The specification of conversation is unique to each set of client and server applications. Both applications must know what messages and data to expect from each other, and both must follow some mutual rules about when to send and when to expect to receive data .To communicate successfully, both the client and server must speak the same language –they must both use the same protocols.

Remember, the concept of a socket is purely an abstraction used in the winsock API.client and server applications need not both be written with winsock communication Winsock is a source code interface a way for programmers to envision network

connections. It's not protocol or network type. A program written with winsock can communicate with different types of systems, as long as they use the same protocols.

Two fundamental types of client-server application pairs exist:

-Connection oriented applications
-Connectionless applications

Whether an application is connection oriented or connectionless is usually determined by the protocols used by the application to communicate and the amount of data to be exchanged.

4.2.4 Connectionless Applications

Applications that send and receive data grams are generally connectionless. The identification of each endpoint in the conversation is established each time data is sent. This scheme is only recommended for applications that transmit small amounts of Information.

In the Internet protocol suite, this usually corresponds to applications that use the User Datagram Protocol (UDP).

The server application starts first. It creates a Socket using the socket () function, names the socket with the bind (), and then waits for requests from clients. Note, servers of this type don't generally use the listen () and accept () functions; they wait for exactly like the rec() function used in connection-oriented servers, but it is also tells the server, from where the data came. The server can then use this information to communicate back to the client, and with sendto() function. It does not call connect () to establish a connection. it sends data directly to the server with the sendto() function. The identification of each application is reestablished every time data is sent or received. When the applications are finished communicating, they close their sockets with the close socket() function.

4.2.5 Creating a Socket

Before we can connect to a server or accept connections from clients,. we have to create a socket our endpoint in the communication circuit

A socket is associated with three elements: an address family, a socket type, and a protocol. you create a socket by calling the socket() function.

Socket (int af,int type,int protocol);

Parameter	Description
Af	address family. for a standards version 1.,1 Winsock, this value is always AF_INET (address internet).In other winsock Implementations. It could be AF_IPX, AFOSI, AFAPPLETALK or some of value.
Type	socket type.
Protocol	transport protocol.

Create() returns true if it succeeds or false if it fails.

4.2.6 Socket Type and Protocol

The sound parameter of the socket () function denotes a socket type. Here, we will examine the most common type:

SOCK_STREAM: which supports byte stream, connection oriented, reliable communication, uses TCP for the Internet address family (AF_INET)

SOCK_DGRAM: which supports datagram, connectionless,"unreliable" Communication. Uses UDP for the Internet address family (AF_INET)

The third parameter to socket() specifies a protocol. The combination of socket type and address family implies a protocol, the winsock specification only
Requires a single protocol foe each socket type using a given address family. For this reason, the protocol parameter can be set using one of the predefined type(IPPROTO_TCP,IPPROTO_UDP.ETC)or simply set to 0 to let
Winsock choose a default. for example , if you create a socket specifying AF_INET and SOCK_DGRAM winsock will assume you want to use UDP.

UDP socket is used for sending and receiving live audio and video as because it reduces delay in which UDP connections does not provide acknowledgements. Also it provides fault tolerance for example if a socket receives a set of eight number of packets instead of ten even then it will not yield to poor quality.
In order to make your conversation sound as much like a telephone conversation as possible, the compression program itself must minimize the delay involved in compressing and decompressing the sound.

. 4.3. Communication Module

4.3.1 H.323 Standard

H.323 is an International Telecommunications Union (ITU) standard that provides specification for computers, equipment, and services for multimedia communication over networks that do not provide a guaranteed quality of service. H.323 computers and equipment can carry real-time video, audio, and data, or any combination of these elements. This standard is based on the Internet Engineering Task Force (IETF) Real-Time Protocol (RTP) and Real-Time Control Protocol (RTCP), with additional protocols for call signaling, and data and audiovisual communications. Users can connect with other people over the Internet and use varying products that support H.323, just as people using different makes and models of telephones can communicate over Public Switched Telephone Network (PSTN) lines. H.323 defines how

audio and video information is formatted and packaged for transmission over the network. Standard audio and video codecs encode and decode input/output from audio and video sources for communication between nodes. A codec (coder/decoder) converts audio or video signals between analog and digital forms. Also, H.323 specifies T.120 services for data communications and conferencing within and next to an H.323 session. Most importantly, this T.120 support means that data handling can occur either in conjunction with H.323 audio and video, or separately.

4.3.2 Benefits

H.323 products and services offer the following benefits to users:

Products and services developed by multiple manufacturers under the H.323 standard can interoperate without platform limitations. H.323 conferencing clients, bridges, servers, and gateways support this interoperability.

H.323 provides multiple audio and video codecs that format data according to the requirements of various networks, using different bit rates, delays, and quality options. Users can choose the codecs that best support their computer and network selections.

The addition of T.120 data conferencing support to the H.323 specification means that products developed under H.323 can offer a full range of multimedia functions, with both data and audiovisual conferencing support.

4.3.3 H.323 Interoperability and Testing

The interoperability of H.323 products is measured on the following three levels:

LEVEL I: Test cases verify that NetMeeting can establish a conference over Transmission Control Protocol/Internet Protocol (TCP/IP) connections with the appropriate data flow and sequencing. Testing identifies whether third-party products interoperate based on the H.323 specifications for the H.245 and Q.931 protocols.

Call signaling and control tests attempt to negotiate these capabilities in the following ways:

Verifying that a third-party product can accept a NetMeeting call using the same default codecs, or that NetMeeting can negotiate a suitable set of codecs.

Determining whether the products can open channels and pass data after the call is established. Verifying that all control sequencing runs correctly.

Typically, call control interoperability testing fails when a call is out of sequence or the call is not accomplished in the allotted amount of time.

LEVEL II Audio and video streaming

Test cases verify that NetMeeting and third-party products can manage the streaming of audio and video packets over User Datagram Protocol (UDP) connections. The streaming mechanisms for H.323-compliant products are Real-Time Protocol (RTP) and Real-Time Control Protocol (RTCP). Interoperability problems might occur within RTP and RTCP; for example, the packet header might be incorrect or a bit could be missing.

LEVEL III Audio and video codec compatibility

Test cases determine whether a third-party product provides compatible codecs for formatting and transmitting the audio and video input/output. NetMeeting runs best using G.723 and H.263, but can negotiate other codecs, such as H.261 and G.711, as necessary. Testers verify that the third-party product can exchange real-time audio and video successfully with NetMeeting. Typically, codec problems evolve from subtle differences in the algorithms used by NetMeeting and the third-party product

4.3.4 H.323 Architecture

The following illustration shows the H.323 architecture. This Architecture defines a set of specific functions for framing and call control, audio and video codecs, and T.120 data communications. The illustration also shows interfaces for the network, and audio and video equipment interfaces.

H.323 terminal architecture, shown in the illustration, is the most common implementation of the H.323 specification. This same architecture can also be implemented for an H.323 Multipoint Control Unit (MCU), gateway, and gatekeeper.

4.3.5 Framing and Call Control

The following standards make up the System Control Unit, which Provides call control and framing capabilities:

- **H.225.0**

This standard defines a layer that formats the transmitted video, audio, data, and control streams for output to the network, and retrieves the corresponding streams from the network. As part of audio and video transmissions, H.225.0 uses the packet format specified by Internet Engineering Task Force (IETF), RTP, and RTCP specifications for the following tasks:

- **Logical framing**

Defines how the protocol frames (packages) the audio and video data into bits (packets) for transport over a selected communications channel.

- **Sequence numbering**

Determines the order of data packets transported over a communications channel.

- **Error detection**

After initiating a call, one or more RTP or RTCP connections are established. Multiple streams allow H.225.0 to send and receive different media types simultaneously, each with their own frame sequence numbers and quality of service options.With RTP and RTCP support, the receiving node synchronizes the received packets in the proper order, so the user hears or sees the information correctly.

The H.225.0 standard also includes registration, admission, and status (RAS) control, which is used to communicate with the gatekeeper. A RAS signaling channel makes the connections between the gatekeeper and H.323 components available.The gatekeeper controls H.323 terminal, gateway, and MCU access to the local area network (LAN) by granting or denying permission to H.323 connections.

- **Q.931**

This protocol defines how each H.323 layer interacts with peer layers, so that participants can interoperate with agreed upon formats. The Q.931 protocol resides within H.225.0.As part of H.323 call control, Q.931 is a link layer protocol for establishing connections and framing data. Q.931 provides a method for defining logical channels inside of a larger channel. Q.931 messages contain a protocol discriminator that identifies each unique message with a call reference value and a message type. The H.225.0 layer then specifies how these Q.931 messages are received and processed.

- **H.245**

This standard provides the call control mechanism that allows H.323-compatible terminals to connect to each other. H.245 provides a standard means for establishing audio and video connections — the series of commands and requests that must be followed for one component to connect and communicate with another. This standard specifies the signaling, flow control, and channeling for messages, requests, and commands.

The built-in framework of H.245 enables codec selection and capability negotiation within H.323. Bit rate, frame rate, picture format, and algorithm choices are some of the elements negotiated by H.24

- **T.120 Data Communications**

H.323 makes a provision for using T.120 as the mechanism for packaging and sending data. T.120 can use the H.225.0 layer to send and receive data packets or simply create an association with the H.323 session and use its own transport capabilities to transmit data directly to the network. Data from conferencing programs, such as file transfer and program sharing, use T.120 support to operate in conjunction with H.323 connections. Also, H.323-compatible products interoperate with data conferencing products developed under the T.120 specification.

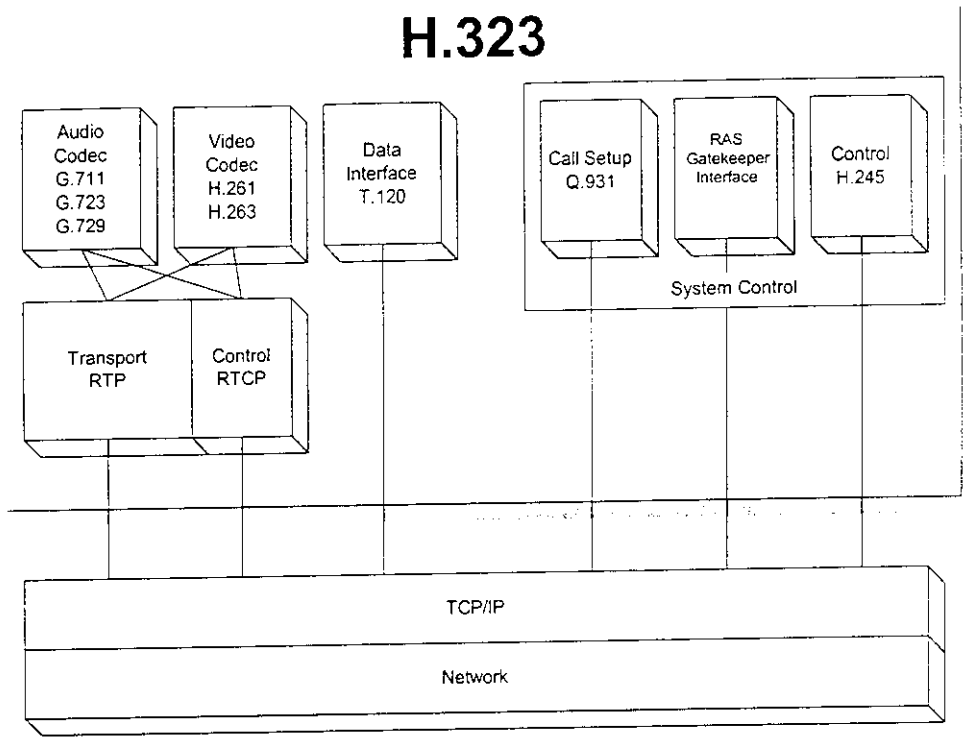


Figure 1. H.323 architecture

4.3.6 Inside TAPI 3.0

TAPI 3.0 integrates multimedia stream control with legacy telephony. Additionally, it is an evolution of the TAPI 2.1 API to the COM model, allowing TAPI applications to be written in any language, such as C/C++ or Microsoft® Visual Basic®.

Besides supporting classic telephony providers, TAPI 3.0 supports standard H.323 conferencing and IP multicast conferencing. TAPI 3.0 uses the Windows® 2000 Active Directory service to simplify deployment within an organization, and it supports quality-of-service (QoS) features to improve conference quality and network manageability.

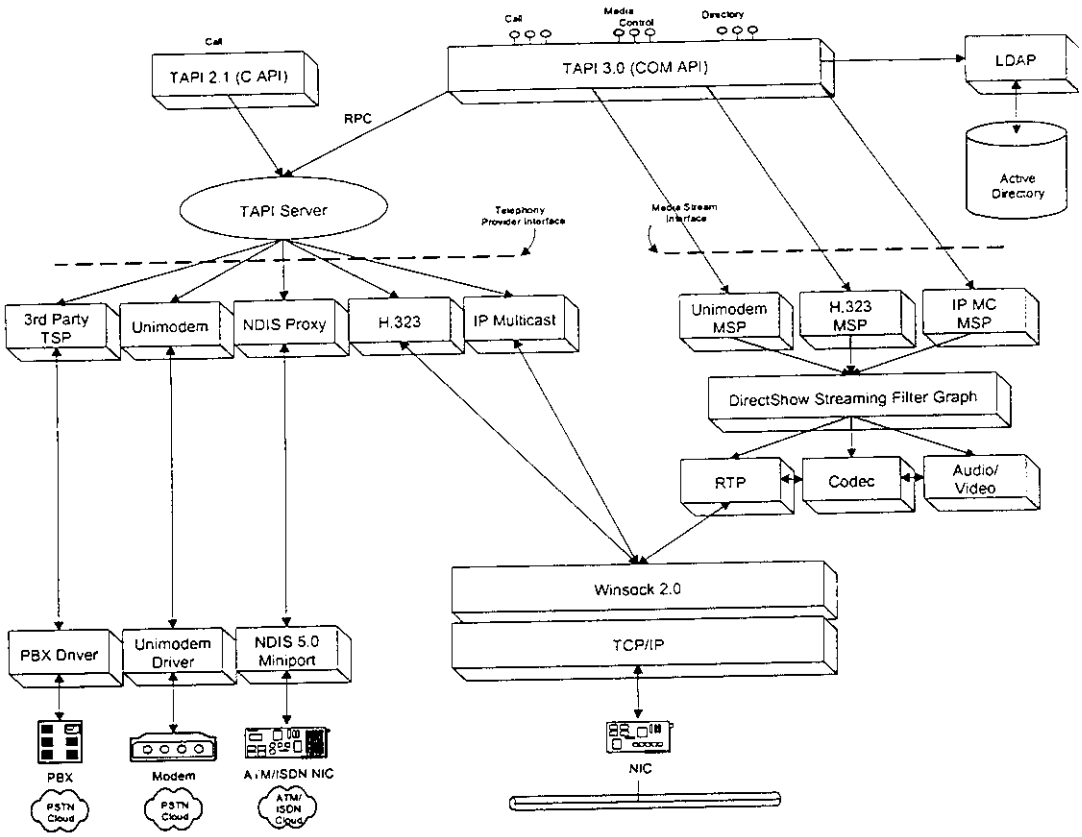


Figure 2. TAPI architecture

There are four major components to TAPI 3.0:

- TAPI 3.0 COM API
- TAPI Server
- Telephony Service Providers
- Media Stream Providers

In contrast to TAPI 2.1, the TAPI 3.0 API is implemented as a suite of COM objects. Moving TAPI to the COM model allows component upgrades of TAPI features. It also allows developers to write TAPI-enabled applications in any language.

The TAPI Server process (TAPISRV.EXE) abstracts the TSPI (TAPI Service Provider Interface) from TAPI 3.0 and TAPI 2.1, allowing TAPI 2.1 Telephony Service Providers to be used with TAPI 3.0, maintaining the internal state of TAPI.

Telephony Service Providers (TSPs) are responsible for resolving the protocol-independent call model of TAPI into protocol-specific call-control mechanisms. TAPI 3.0 provides backward compatibility with TAPI 2.1 TSPs. Two IP telephony service providers (and their associated MSPs) ship by default with TAPI 3.0: the H.323 TSP and the IP Multicast Conferencing TSP, which are discussed below.

TAPI 3.0 provides a uniform way to access the media streams in a call.. TAPI Media Stream Providers (MSPs) implement DirectShow interfaces for a particular TSP and are required for any telephony service that makes use of DirectShow streaming. Generic streams are handled by the application.

The figure 2. Shows the relationship between the TAPI functions and H.323 protocol. The lower layer shows the details inter-relation with winsock and H.323

Protocol stack.

5. RESULTS OBTAINED

5. RESULTS OBTAINED

The technology is simulated by using a client/server model in an LAN network.

- **Name resolution**

Before a client application can initiate communication with as server it must first know the server's name or address. The Domain mane system (DNS) will return an IP Address when given a valid host name.

- **Creating a socket**

Winsock defines many types of sockets and the one used here is SOCK_DGRAM which supports datagram, connectionless "unreliable" communication.

5.1 Client's Role: Connecting To the Server

Each client connects to server and binds its address to a well known port. UDP Socket is created to send the real-time audio. As multiple clients can connect to a server each client may request another client and start the communication. A client upon accepting the request can start to communicate.

The figure shows the arrangement of two clients connecting to the server

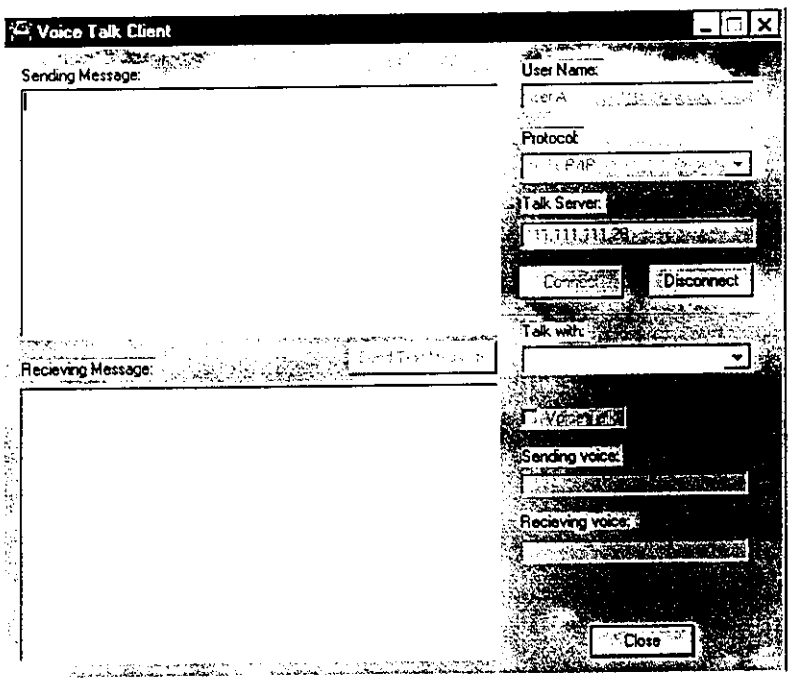


Figure.3. User A connects to the server

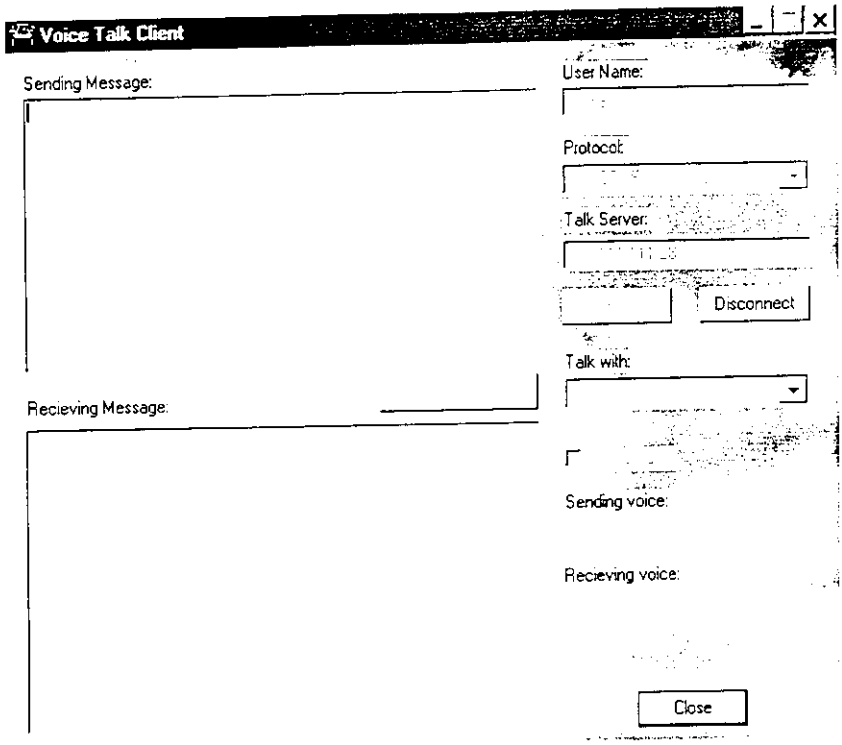


Figure 4.user B connecting to the server

Threads are associated to the every client attaching to the server and the server takes care of all its associated client threads. Once if a client disconnects from the server then the corresponding thread is deleted. Clients connect to the server by knowing the IP address of the server.

5.2 Server' Role: Listening For Incoming Connection

The server listens for incoming connections and accepts connections and provides services to the client. Server creates socket a binds to defined port. The identification of each application is reestablished every time data is sent or received.

The following figure gives the status of server, giving the information the number of clients attached to it. It is in listen mode and accepts the incoming connections.

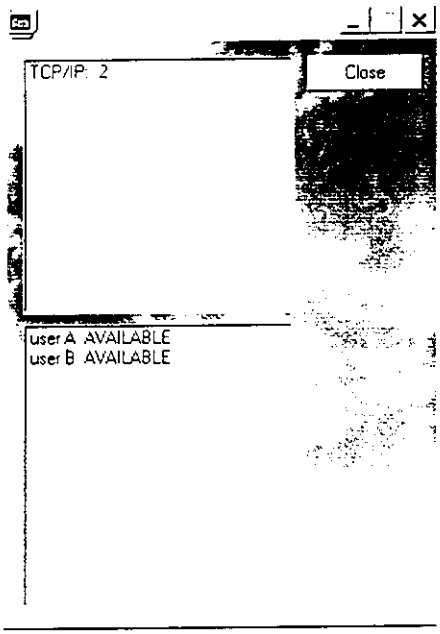


Figure 5 Server accepting incoming connections

5.3 Sending and Receiving Data

Once the connection is established, voice communication starts between the clients. Every voice signal is compressed by Linear Predictive Coding (LPC) algorithm and sent it to the destination. At the receiving end it is decompressed and produces the original sound.

The following figure shows the communication phase between two clients in which one client sends the data and another one receives the data.

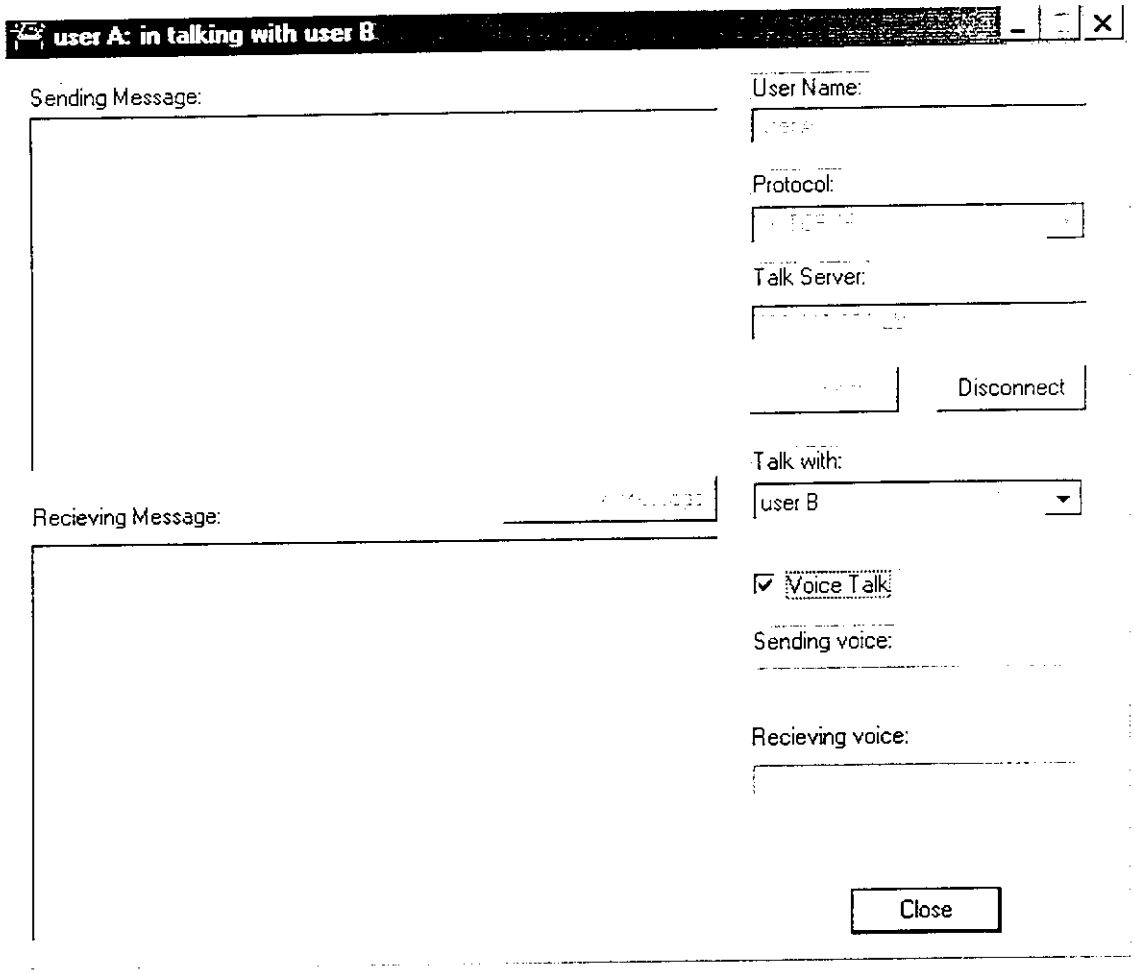


Figure.6. client A sending mesage

The specification of conversation is unique to each set of client and server applications. Both applications must know what messages and data to expect from each other, and both must follow some mutual rules about when to send and when to expect to receive data .To communicate successfully, both the client and server must speak the same language –they must both use the same protocols.

Both the clients receive messages from each other and the output is directed to the wave device. The server maintains the status of the communication system and once if the server connection is dropped, the entire communication will be aborted.

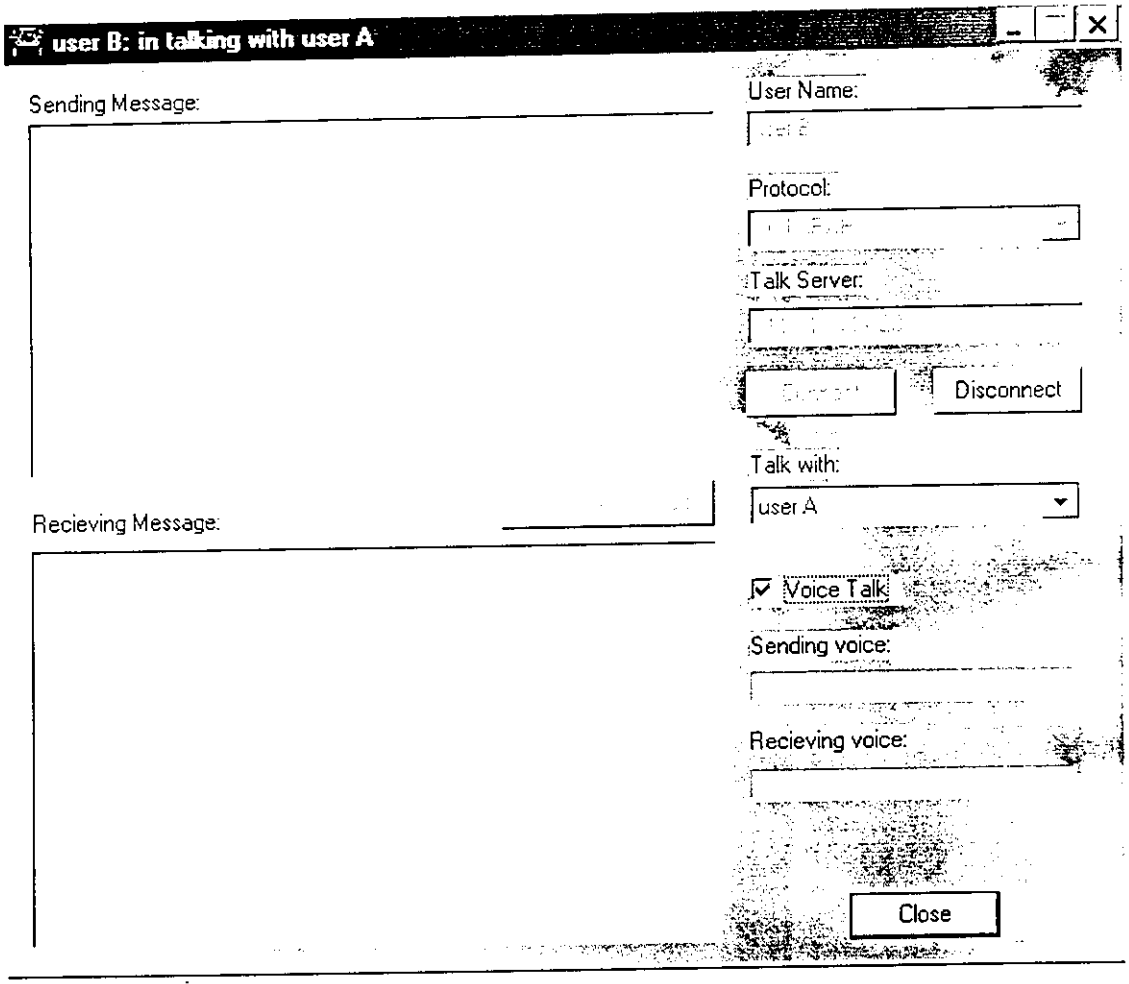


figure 7.client B sending messages

6. CONCLUSION AND FUTURE OUTLOOK

6. Conclusion

Data traffic has traditionally been forced to fit onto the voice network (using modems, for example). The Internet has created an opportunity to reverse this integration strategy - voice and facsimile can now be carried over IP networks, with the integration of video and other multimedia applications close behind. The Internet and its underlying TCP/IP protocol suite have become the driving force for new technologies, with the unique challenges of real-time voice being the latest in a series of developments. Telephony over the Internet cannot make compromises in voice quality, reliability, scalability, and manageability. It must also interwork seamlessly with telephone systems all over the world. Just about today's entire network devices will need to be voice-enabled (and eventually multimedia-enabled).

The VoIP simulated here works well with the class I level i.e. making communication with PC to PC through IP networks with the minimal delay. The voice signals from the source end is made digitized .The digitized signal is compressed by using Linear Predictive Coding (LPC) algorithm and packetized to travel over IP networks . The IP packet travels over the IP network and reaches the destination. At the destination end, the packet gets decompressed, produces the original voice signal.

Future Extension

Future extensions will include innovative new solution including conference bridging, voice/data synchronization, real time and message-based services and voice response systems. The product can be tested with large Wide Area Networks (WAN) and in Internet and its performance can be measured. The Quality of Service (QoS) support Can also be included when packets has to travel across many IP networks. The VoIP software can also be tested with the new protocol stack, named Session Initialization protocol (SIP), which is an standard recommended by the Internet Engineering Task Force (IETF).the resultant product can be compared with this H.323 protocol suite and the performance between them can be measured.

7. REFERENCES

7. REFERENCES

Books and Journals

1. Marcus Goncalvus, *Voice over IP Networks*, McGraw-Hill, 1999

2. Douglas O'Shaughnessy, *Speech communications,* second edition, IEEE press

3. Daniele Rizzetoto, Claudio Catania, IEEE transactions on Internet computing

   (vol.3,no.3) –*A voice over IP service architecture for Integrated Communication*

   May/June 1993

4. Philip carden, Network Computing Magazine, *Building VoIP Networks*

   May 2000

Web sites

5. http:// www.cisco.com, July 30 2001,

   -*Convergence of voice, video, data architecture*- white paper

6. http:/ /www.itu.int, July 30, 2001.

   -Information about VoIP standards

7. http://www.ietf.org

   Request for comments (RFC) 1889, RTP: A Transport Protocol for Real-Time Applications , Jan 1996.

APPENDIX

Public Switched Telephone Network (PSTN)—General term referring to the variety of telephone networks and services in place worldwide.

Quality of Service (QoS)—Measure of performance for a transmission system that reflects its transmission quality and service availability.

MTU—The largest amount of data that can be transferred across a given physical network.

Real-Time Transport Protocol (RTP)—The standard protocol for streaming applications developed within the IETF.

Resource Reservation Protocol (RSVP)—A protocol that supports the reservation of resources across an IP network. Applications running on IP end systems can use RSVP to indicate to other nodes the nature (bandwidth, jitter, maximum burst, and so on) of the packet streams they wish to receive.

RTP Control Proctocol (RTCP)—A protocol providing support for applications with real-time properties, including timing reconstruction, loss detection, security, and content identification. RTCP provides support for real-time conferencing for large groups within an Internet, including source identification and support for gateways (like audio and video bridges) and multicast-to-unicast translators.

Voice Activity Detection (VAD)—Saves bandwidth by transmitting voice cells only when voice activity is detected.

Pulse Code Modulation (PCM)—Transmission of analog information in digital form through sampling and encoding the samples with a fixed number of bits.

Time-Division Multiplexing (TDM)—Technique in which information from multiple channels can be allocated bandwidth on a single wire-based on preassigned time slots. Bandwidth is allocated to each channel regardless of whether the station has data to transmit.

Dial Tone Multi-Frequency (DTMF)—The set of standardized, superimposed tones used in telephony signaling - as generated by a touch tone pad.

Sample code

```
// stdafx.cpp : source file that includes just the standard includes
//      talkclient.pch will be the pre-compiled header
//      stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"

// client.cpp :.
//

#include "stdafx.h"
#include "talkclient.h"
#include "talkclientDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CTalkclientApp

BEGIN_MESSAGE_MAP(CTalkclientApp, CWinApp)
        //{{AFX_MSG_MAP(CTalkclientApp)
                // NOTE - the ClassWizard will add and remove mapping macros here.
                //    DO NOT EDIT what you see in these blocks of generated code!
        //}}AFX_MSG
        ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CTalkclientApp construction

CTalkclientApp::CTalkclientApp()
{
        // TODO: add construction code here,
        // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CTalkclientApp object

CTalkclientApp theApp;
```

```
//////////////////////////////////////////////////////////////////////
// CTalkclientApp initialization

BOOL CTalkclientApp::InitInstance()
{
	AfxEnableControlContainer();

	// Standard initialization
	// If you are not using these features and wish to reduce the size
	//  of your final executable, you should remove from the following
	//  the specific initialization routines you do not need.

#ifdef _AFXDLL
	Enable3dControls();			// Call this when using MFC in a shared
DLL
#else
	Enable3dControlsStatic();	// Call this when linking to MFC statically
#endif

	CTalkclientDlg dlg;
	m_pMainWnd = &dlg;
	int nResponse = dlg.DoModal();
	if (nResponse == IDOK)
	{
		// TODO: Place code here to handle when the dialog is
		//  dismissed with OK
	}
	else if (nResponse == IDCANCEL)
	{
		// TODO: Place code here to handle when the dialog is
		//  dismissed with Cancel
	}

	// Since the dialog has been closed, return FALSE so that we exit the
	//  application, rather than start the application's message pump.
	return FALSE;
}



#include "stdafx.h"
#include "talkclient.h"
#include "talkclientDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
```

therefore count this number to refine our determination. In LPC the average magnitude difference function (AMDF) is used to determine the pitch period.

• *STEP 3*: An additional factor that influences this classification is the surrounding segments. • The classification of these neighboring segments is taken into consideration because it is undesirable to have an unvoiced frame in the middle of a group of voiced frames or vice versa.

- The LPC filter is given by:

$$H(z) = \frac{1}{1 - a_1 z^{-1} + a_2 z^{-2} + \ldots + a_{10} z^{-10}}$$

which is equivalent to saying that the input-output relationship of the filter is given by the linear difference equation:

$$s(n) + \sum_{i=1}^{10} a_i s(n-i) = u(n)$$

- The LPC model can be represented in vector form as:

$$\mathbf{A} = (a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, G, V/UV, T)$$

- $\mathbf{A}$ changes every 20 msec or so. At a sampling rate of 8000 samples/sec, 20 msec is equivalent to 160 samples.
- The digital speech signal is divided into *frames* of size 20 msec. There are 50 frames/second.
- The model says that

$$\mathbf{A} = (a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, G, V/UV, T)$$

is equivalent to

```
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/*************************************************************************
***
 *                                                                       *
 * CAboutDlg dialog used for App About                                   *
 *                                                                       *

*************************************************************************
**/
class CAboutDlg : public CDialog
{
public:
        CAboutDlg();

// Dialog Data
        //{{AFX_DATA(CAboutDlg)
        enum { IDD = IDD_ABOUTBOX };
        //}}AFX_DATA

        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CAboutDlg)
        protected:
        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
        //}}AFX_VIRTUAL

// Implementation
protected:
        //{{AFX_MSG(CAboutDlg)
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
        //{{AFX_DATA_INIT(CAboutDlg)
        //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(CAboutDlg)
        //}}AFX_DATA_MAP
```

```
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
        //{{AFX_MSG_MAP(CAboutDlg)
                // No message handlers
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/*****************************************************************************
***
 *                                                *
 *                                                        *
 * CTalkclientDlg dialog                          *
 *                                                *

*****************************************************************************
**/
CTalkclientDlg::CTalkclientDlg(CWnd* pParent /*=NULL*/)
        : CDialog(CTalkclientDlg::IDD, pParent)
{
        //{{AFX_DATA_INIT(CTalkclientDlg)
        //}}AFX_DATA_INIT
        // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
        m_strUserName = _T("");
        m_strServer = _T("");
        m_bVoiceTalk = FALSE;
        m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
   m_bConnect = FALSE;
   m_bTalking = FALSE;
        memset(m_lpPeerName, 0, 16);

        m_bVoiceMsg = FALSE;
        m_uiBufLen = VT_MAX;
        m_iWaveFormat = VT_WAVE1M08;
}

void CTalkclientDlg::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(CTalkclientDlg)
        DDX_Control(pDX, IDC_BTNCONNECT, m_btnConnect);
        DDX_Control(pDX, IDC_BTNSEND, m_btnSendTxtMsg);
        DDX_Control(pDX, IDC_CHKVOICE, m_chkVoice);
        DDX_Control(pDX, IDC_EDITNAME, m_editUserName);
        DDX_Control(pDX, IDC_EDITSERVER, m_editServer);
        DDX_Control(pDX, IDC_EDITRECV, m_TextRecv);
        DDX_Control(pDX, IDC_EDITSEND, m_TextSend);
```

```
        DDX_Control(pDX, IDC_COMBPROT, m_ComBoProts);
        DDX_Control(pDX, IDC_COMBCHATTER, m_ComBoTalker);
        DDX_Control(pDX, IDC_SENDMETER, m_progctrlSend);
        DDX_Control(pDX, IDC_RECVMETER, m_progctrlRecv);
        //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CTalkclientDlg, CDialog)
        //{{AFX_MSG_MAP(CTalkclientDlg)
        ON_WM_SYSCOMMAND()
        ON_WM_PAINT()
        ON_WM_QUERYDRAGICON()
    ON_MESSAGE(VTM_DATAREADY, OnVTDataReady)
    ON_MESSAGE(VTM_READYFORWRITE, OnVTReadyForWrite)
    ON_MESSAGE(MM_WIM_DATA, OnRecordEvent)
    ON_MESSAGE(MM_WOM_DONE, OnPlayEvent)
        ON_EN_CHANGE(IDC_EDITNAME, OnChangeEditname)
        ON_EN_CHANGE(IDC_EDITSERVER, OnChangeEditserver)
        ON_CBN_SELCHANGE(IDC_COMBPROT, OnSelchangeCombprot)
        ON_CBN_SELCHANGE(IDC_COMBCHATTER, OnSelchangeCombchatter)
        ON_BN_CLICKED(IDC_BTNCONNECT, OnBtnconnect)
        ON_BN_CLICKED(IDC_BTNDISCONNECT, OnBtndisconnect)
        ON_BN_CLICKED(IDC_BTNSEND, OnBtnsend)
        ON_BN_CLICKED(IDC_CHKVOICE, OnChkvoice)
        ON_WM_CLOSE()
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
//
// CTalkclientDlg message handlers
//

/////////////////////////////////////////////////////////////////////////////
// Initialize the dialogbox
//
/////////////////////////////////////////////////////////////////////////////
BOOL CTalkclientDlg::OnInitDialog()
{
        CDialog::OnInitDialog();
```

```
	// Add "About..." menu item to system menu.

	// IDM_ABOUTBOX must be in the system command range.
	ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
	ASSERT(IDM_ABOUTBOX < 0xF000);

	CMenu* pSysMenu = GetSystemMenu(FALSE);
	if (pSysMenu != NULL)
	{
		CString strAboutMenu;
		strAboutMenu.LoadString(IDS_ABOUTBOX);
		if (!strAboutMenu.IsEmpty())
		{
			pSysMenu->AppendMenu(MF_SEPARATOR);
			pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
strAboutMenu);
		}
	}

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here

	// Initialize the edit boxes and combo-boxes
  m_ComBoTalker.ResetContent();
	m_ComBoProts.ResetContent();
	m_editUserName.LimitText(15);
	m_progctrlSend.SetRange(0, 1);
	m_progctrlRecv.SetRange(0, 1);
	m_progctrlSend.SetPos(0);
	m_progctrlRecv.SetPos(0);


	// Check version data and load winsock dll
  WORD	wVersionRequested;
  WSADATA	wsaData;
  char	outtext[128];
  int	iLoop;

  wVersionRequested = MAKEWORD(1,1);
  if (WSAStartup(wVersionRequested, &wsaData) != 0)
  {
```

```
    return FALSE;
}

        // Check compatible network protocols
        if(!InitProtocols())
                return FALSE;

        // Add compatible protocol to protocol combo-box
        for(iLoop = 0; iLoop < m_iNumWorkProts; iLoop++)
{
                wsprintf(outtext, "%2d: %s", iLoop,
                        m_lpProtBuf[m_iWorkProts[iLoop]].lpProtocol);
                m_ComBoProts.InsertString(-1, outtext);
        }

        // Set the controls status
        DoConnection(FALSE);

        // Initialize the wave I/O device
        m_bVoiceMsg = FALSE;
    m_vtPlay.SetHwnd(GetSafeHwnd());
    m_vtPlay.SetVTWaveFormat(m_iWaveFormat);
    m_vtPlay.AllocMemory();

        m_vtRecord.SetHwnd(GetSafeHwnd());
    m_vtRecord.SetVTWaveFormat(m_iWaveFormat);
        m_vtRecord.AllocMemory();

        // return TRUE  unless you set the focus to a control
        return TRUE;
}

/*****************************************************************************
***
 * Dialog box closed                                                         *

*****************************************************************************
**/
void CTalkclientDlg::OnClose()
{
        // TODO: Add your message handler code here and/or call default

        // Clean the combo-boxes
    m_ComBoTalker.ResetContent();
        m_ComBoProts.ResetContent();
```

```
        // Close and clean the socket
    closesocket(m_MySock.m_hSock);
    WSACleanup();


        // Close the wave I/O device and
        // Clean the wave I/O buffers
    if(m_bVoiceMsg)
      m_vtPlay.StopPlay();
        if(m_bVoiceTalk)
      m_vtRecord.StopRecord();

        m_vtPlay.CleanMemory();
    m_vtRecord.CleanMemory();

        CDialog::OnClose();
}


/*************************************************************************
***
 * System menu commands                                                  *
*************************************************************************
**/
void CTalkclientDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
        if ((nID & 0xFFF0) == IDM_ABOUTBOX)
        {
                CAboutDlg dlgAbout;
                dlgAbout.DoModal();
        }
        else
        {
                CDialog::OnSysCommand(nID, lParam);
        }
}


/*************************************************************************
***
 * Paint function                                                        *
 * If you add a minimize button to your dialog, you will need the code  *
 * below to draw the icon.  For MFC applications using the document/view *
 * model, this is automatically done for you by the framework.           *
```

```
*************************************************************************
**/
void CTalkclientDlg::OnPaint()
{
	if (IsIconic())
	{
		CPaintDC dc(this); // device context for painting

		SendMessage(WM_ICONERASEBKGND, (WPARAM)
dc.GetSafeHdc(), 0);

		// Center icon in client rectangle
		int cxIcon = GetSystemMetrics(SM_CXICON);
		int cyIcon = GetSystemMetrics(SM_CYICON);
		CRect rect;
		GetClientRect(&rect);
		int x = (rect.Width() - cxIcon + 1) / 2;
		int y = (rect.Height() - cyIcon + 1) / 2;

		// Draw the icon
		dc.DrawIcon(x, y, m_hIcon);
	}
	else
	{
		CPaintDC dc(this);
		CRect rc;
		GetClientRect(&rc);

		CDC dcMem;
		CBitmap bmp;
		bmp.LoadBitmap(IDB_BGBMP);

		BITMAP bm;
		bmp.GetObject(sizeof(BITMAP),&bm);

		dcMem.CreateCompatibleDC(&dc);
		dcMem.SelectObject(&bmp);


	dc.StretchBlt(0,0,rc.right,rc.bottom,&dcMem,0,0,bm.bmWidth,bm.bmHeight,SR
CCOPY);

		CDialog::OnPaint();
	}
}
```

```
/*********************************************************************
***
 * The system calls this to obtain the cursor to display while the user   *
 * drags the minimized window.                                            *
*********************************************************************
**/
HCURSOR CTalkclientDlg::OnQueryDragIcon()
{
        return (HCURSOR) m_hIcon;
}


/*********************************************************************
***
 * Check compatible network protocols                                     *
*********************************************************************
**/
BOOL CTalkclientDlg::InitProtocols(void)
{
   int     numStrucs;
   int     index;
   unsigned long  iProtBufSize = 0;

        // Call EnumProtocols with NULL buffer in order
        // to determine size of buffer required
        EnumProtocols(NULL, NULL, &iProtBufSize);

        // Allocate buffer to protocol information structure
   if((m_lpProtBuf = (LPPROTOCOL_INFO)VirtualAlloc(NULL,
                        iProtBufSize,
                        MEM_COMMIT,
                        PAGE_READWRITE)) == NULL)
   {
     // ERROR -- abort
                AfxMessageBox("Failed to allocate memory for m_lpProtBuf!");
     return FALSE;
   }

   // Calling EnumProtocols with large enough buffer
   if((numStrucs = EnumProtocols(NULL,
                                m_lpProtBuf,
                                                &iProtBufSize)) ==

 SOCKET_ERROR)
```

```
{
  // Error -- abort
            AfxMessageBox("Failed to numerate protocol!");
  return FALSE;
}

      // Add every connection oriented protocol to protocol combo box
      m_iNumWorkProts = 0;
 for(index = 0; index < numStrucs; index++)
 {
   if((m_lpProtBuf[index].dwServiceFlags & XP_CONNECTIONLESS) == 0)
   {
                  // Keep track of workable protocols
     m_iWorkProts[m_iNumWorkProts++] = index;
   }
 }

      // OK
      return TRUE;
}

/*************************************************************************
***
 * Intialize the different protocol sockets                              *

*************************************************************************
**/
BOOL CTalkclientDlg::InitSockets(int iProtIndex)
{
  int numStrucs;
  GUID guidNW = SVCID_NETWARE(VT_NETID);   // Generate GUID for Novel
prot
  GUID guidDNS = SVCID_TCP(VT_DNSID);      // Generate GUID for DNS
      char strServName[128];
  char outtext[128];


      // Check the protocol type
  switch(m_lpProtBuf[iProtIndex].iAddressFamily)
  {
    case AF_IPX:
                // SAP/Bindery Name Space
                m_dwCSABufsize = sizeof(m_CSABuf);
      numStrucs = GetAddressByName(0,
                        &guidNW,
```

```
                    "VOICE_TALK_SERVER_IPX",
                    NULL,
                    RES_FIND_MULTIPLE,
                    NULL,
                    m_CSABuf,
                    &m_dwCSABufsize,
                    NULL,
                    NULL);

  if((numStrucs == SOCKET_ERROR) || (numStrucs == 0))
  {
    // Error -- try another protocol.  We've got lots!
    AfxMessageBox("Server not found.  Try a different protocol");
    return FALSE;
  }
  break;
case AF_INET:
  //  DNS Name Space
  // Static Name Space requires us to specify the host name
            m_dwCSABufsize = sizeof(m_CSABuf);
  strcpy(strServName, m_strServer);
  numStrucs = GetAddressByName(0,
                    &guidDNS,
                    strServName,
                    NULL,
                    RES_FIND_MULTIPLE,
                    NULL,
                    m_CSABuf,
                    &m_dwCSABufsize,
                    NULL,
                    NULL);

   if((numStrucs == SOCKET_ERROR) || (numStrucs == 0))
   {
     // Error -- try another protocol.  We've got lots!
     AfxMessageBox("Server not found.  Try a different protocol");
     return FALSE;
   }
   break;

 case AF_NETBIOS:
                  // NetBIOS name space???

   // no netbios name space provider so fill in lpCSABuf ourselves
   numStrucs = 1;
   m_CSABuf[0].iSocketType = m_lpProtBuf[iProtIndex].iSocketType;
```

```
        m_CSABuf[0].iProtocol = m_lpProtBuf[iProtIndex].iProtocol;

        SET_NETBIOS_SOCKADDR(&m_NBAddr,
                        NETBIOS_GROUP_NAME,
                        "VOICE_TALK_SERVER_NETBIOS",
                        0);
              m_CSABuf[0].RemoteAddr.lpSockaddr = (LPSOCKADDR)&m_NBAddr;
                m_CSABuf[0].RemoteAddr.iSockaddrLength =
sizeof(m_NBAddr);
        break;

      default:
                // We don't support anything else
        AfxMessageBox("Name Space not supported over this protocol family. \
                                Try a different protocol");
        return FALSE;
    }

        // Set SOCK data
        m_MySock.m_iSockType = m_lpProtBuf[iProtIndex].iSocketType;
        m_MySock.m_iProtocol = m_lpProtBuf[iProtIndex].iProtocol;

        // Call socket() using triple provided by EnumProtocols()
    if((m_MySock.m_hSock = socket(m_lpProtBuf[iProtIndex].iAddressFamily,
                m_lpProtBuf[iProtIndex].iSocketType,
                m_lpProtBuf[iProtIndex].iProtocol))
                                                == INVALID_SOCKET)

    {
      // ERROR
      wsprintf(outtext, "Failure: socket() returned %u", WSAGetLastError());
              AfxMessageBox(outtext);
      return FALSE;
    }

        // Call connect() using info from GetAddressByName()
    if(connect(m_MySock.m_hSock,
          m_CSABuf[0].RemoteAddr.lpSockaddr,
          m_CSABuf[0].RemoteAddr.iSockaddrLength)
                          == SOCKET_ERROR)

    {
      // ERROR
      wsprintf(outtext, "Failure: connect() returned %u", WSAGetLastError());
              AfxMessageBox(outtext);
      return FALSE;
    }
```

```
        // Specify message to signal accepted socket
    if(WSAAsyncSelect(m_MySock.m_hSock,
                        GetSafeHwnd(),
                        VTM_READYFORWRITE,
                                FD_WRITE) == SOCKET_ERROR)

    {
      wsprintf(outtext, "Failure: WSAAsyncSelect() returned %u", WSAGetLastError());
                AfxMessageBox(outtext);
      return FALSE;
    }

          return TRUE;
}


/*************************************************************************
***
 * Set dialogbox status in connection/disconnection                    *

*************************************************************************
**/
void CTalkclientDlg::DoConnection(BOOL bTrue)
{
        m_bConnect = bTrue;

        // Connection
        if(bTrue)
        {
                // Reset the talker list
       m_ComBoTalker.ResetContent();
                m_ComBoTalker.InsertString(0, "No one to talk");

                // Reset socket working status
                if(WSAAsyncSelect(m_MySock.m_hSock,
                                GetSafeHwnd(),
                                        VTM_DATAREADY,
                                        FD_READ | FD_CLOSE) ==
 SOCKET_ERROR)
                {
         // ERROR  do something
                        AfxMessageBox("Failure: WSAAsyncSelect() to get online
 users!");
                        return;
                }
        }
        else // disconnection
```

```
    {
            // Close socket
    closesocket(m_MySock.m_hSock);

            // Reset the edit boxes, combo-boxes, etc.
    m_ComBoTalker.ResetContent();
            m_ComBoTalker.InsertString(0, "No one to talk");
            m_chkVoice.SetCheck(0);
            m_bVoiceMsg = FALSE;
        m_progctrlSend.SetPos(0);
        m_progctrlRecv.SetPos(0);
      }

      // Enable/Disable various buttons, edit boxes
      m_btnConnect.EnableWindow(!bTrue);
      m_editUserName.EnableWindow(!bTrue);
      m_editServer.EnableWindow(!bTrue);
      m_ComBoProts.EnableWindow(!bTrue);

      // If disconnected
      if(!bTrue)
      {
            // Reset flags and controls
            EnableTalking(FALSE);

            // Close wave I/O devices
            if(m_bVoiceMsg)
      m_vtPlay.StopPlay();

        if(m_bVoiceTalk)
      m_vtRecord.StopRecord();
       }
}

/*************************************************************************
***
 * Set communication status flags and controls                          *

*************************************************************************
**/
void CTalkclientDlg::EnableTalking(BOOL bTrue)
{
  char sShowText[128];
  int  iIndex;
```

```
        // Set communication flag
        m_bTalking = bTrue;

        // Set graphic controls status
        m_btnSendTxtMsg.EnableWindow(bTrue);
        m_chkVoice.EnableWindow(bTrue);
        m_progctrlSend.EnableWindow(bTrue);
        m_progctrlRecv.EnableWindow(bTrue);

        // Set talker list and dialog title bar
        if(bTrue)
        {
      wsprintf(sShowText, "%s: in talking with %s", m_MySock.m_sUserName, \
                              m_lpPeerName);

          iIndex = m_ComBoTalker.FindString(0, m_lpPeerName);
                m_ComBoTalker.SetCurSel(iIndex);
        }
        else
        {
                if(lstrlen(m_MySock.m_sUserName) > 0)
                   wsprintf(sShowText, "%s: free", m_MySock.m_sUserName);
                else
        wsprintf(sShowText, "Voice Talk Client");
        }

        SetWindowText(sShowText);
}


/*************************************************************************
***
 * Parse the content in the receiving message                          *

*************************************************************************
**/
void CTalkclientDlg::ParseMsgData(void)
{
        CString         strRecv;

        // Check message type
        switch(m_vtRecvMsgBuf.m_ucMsgType)
        {
          case VT_TEXT:  // Text
                        // Stop voice communication
                        if(m_bVoiceMsg)
```

```
                {
                    m_bVoiceMsg = FALSE;
        m_vtPlay.StopPlay();
            m_progctrlRecv.SetPos(0);
                }

                // Put text message to the edit box
                strRecv = (char*)(m_vtRecvMsgBuf.m_pData);
                m_TextRecv.SetWindowText(strRecv);
                return;
    case VT_WAVE1M08:  // Voice message
    case VT_WAVE1S08:
    case VT_WAVE1M16:
    case VT_WAVE1S16:
    case VT_WAVE2M08:
    case VT_WAVE2S08:
    case VT_WAVE2M16:
    case VT_WAVE2S16:
    case VT_WAVE4M08:
    case VT_WAVE4S08:
    case VT_WAVE4M16:
    case VT_WAVE4S16:
                // Reset the indicatior
       m_progctrlRecv.SetPos(1);
                // Get wave data buffer size
                m_uiBufLen = m_vtRecvMsgBuf.m_lLength -
SIZEVTMSGHDR;
                // Copy wave data buffer
            memcpy((void*)m_pWaveData, (void*)m_vtRecvMsgBuf.m_pData,
          m_uiBufLen);
                if(!m_bVoiceMsg)
                {
                        // Set wave data to output device and play it
                  m_bVoiceMsg = TRUE;
                        m_vtPlay.SetPlayData(m_pWaveData, m_uiBufLen);
        m_vtPlay.StartPlay();
                }
                return;
        }
}

/*************************************************************************
***                                                                    *
 * Response to the VTM_DATAREADY message
```

```
**********************************************************************
**/
LONG CTalkclientDlg::OnVTDataReady(WPARAM wparam, LPARAM lparam)
{
  // If the server connection was closed
        char sPeerName[16];
        char sShowText[128];
  int  iIndex;
        int  iCount;

        // Clean the temperary buffers
        memset((void*)sPeerName, 0, 16);
        memset((void*)sShowText, 0, 128);

  // If get "server close" message
        if(LOWORD(lparam) == FD_CLOSE)
        {
                // Disconnect
                AfxMessageBox("Server Connection Dropped!");
                DoConnection(FALSE);
                return 1L;
        }

        // Clean the buffer for receiving message
  memset((void*)&m_vtRecvMsgBuf, 0, SIZEVTMSG);

  // Read the incoming message data
  if(!RecvVTMessage(&m_MySock, &m_vtRecvMsgBuf))
  {
                // Failed
    return 0L;
  }

  // Recieved the whole message!
        // check the command flag
  switch(m_vtRecvMsgBuf.m_ucCmd)
  {
                // Register a talking user name
    case VTCMD_REGNAME:
      // Add the peer name to "Talk With" combobox
            memcpy((void*)sPeerName, (void*)m_vtRecvMsgBuf.m_pData,
                                16*sizeof(char));
                        iCount = m_ComBoTalker.GetCount();
                        m_ComBoTalker.InsertString(iCount, sPeerName);
```

```
// Deregister a talking user name
            case VTCMD_DEREGNAME:
    // Remove name from list box
                    memcpy((void*)sPeerName, (void*)m_vtRecvMsgBuf.m_pData,
                            16*sizeof(char));
    iIndex = m_ComBoTalker.FindString(-1, sPeerName);
              if(iIndex != LB_ERR)
                    {
      // found the index to the item...delete it!
                        m_ComBoTalker.DeleteString(iIndex);

    }
    break;

  // Invite to join the session
              case VTCMD_REQSESSION:
                // Someone is asking us for a chat session
     lstrcpy(sShowText, (char*)m_vtRecvMsgBuf.m_pData);
     lstrcat(sShowText, " requests to talk.");
     if(AfxMessageBox(sShowText, MB_OKCANCEL) == IDOK)
     {
       // Save the name of the peer
       lstrcpy(m_lpPeerName, (char*)m_vtRecvMsgBuf.m_pData);
       // Send response
       memset(&m_vtSendMsgBuf, 0, SIZEVTMSG);
                        m_vtSendMsgBuf.m_ucIdentity = VT_IDENTITY;
                        m_vtSendMsgBuf.m_ucCmd =
VTCMD_SESSIONREQRESP;
       m_vtSendMsgBuf.m_ucMsgType = VT_TEXT;
       m_vtSendMsgBuf.m_lLength = SIZEVTMSGHDR + 2;
                        m_vtSendMsgBuf.m_pData[0] = 1;
       if(SendVTMessage(m_MySock.m_hSock, &m_vtSendMsgBuf))
                                EnableTalking(TRUE);
                        else
         AfxMessageBox("Can't send message to server");
       return 1L;
     }
     else
     {
       // Rejected the session request,
                        // send the denial message
       memset(&m_vtSendMsgBuf, 0, SIZEVTMSG);
                        m_vtSendMsgBuf.m_ucIdentity = VT_IDENTITY;
                        m_vtSendMsgBuf.m_ucCmd =
VTCMD_SESSIONREQRESP;
```

```
        m_vtSendMsgBuf.m_lLength = SIZEVTMSGHDR + 2;
                        m_vtSendMsgBuf.m_pData[0] = 0;
        if(SendVTMessage(m_MySock.m_hSock, &m_vtSendMsgBuf))
                            EnableTalking(FALSE);

    }
    break;  // should never get here

// Response to request
case VTCMD_SESSIONREQRESP:
            // Someone responded to our session request!
    if(m_MySock.m_iStatus != VTSOCKET_REQSESSION)
    {
                        // unwanted packet
      return 0L;
    }
    if(*m_vtRecvMsgBuf.m_pData == 1)
    {
      //Acceptannce message
      EnableTalking(TRUE);
    }
    else
    {
      // Denial message
      EnableTalking(FALSE);
    }
                break;

// Peer exit the session
case VTCMD_SESSIONCLOSE:
                // Set dialogbox
                EnableTalking(FALSE);
                // Set socket status
    m_MySock.m_iStatus = VTSOCKET_AVAILABLE;
    wsprintf(sShowText, "%s: available for talking", m_MySock.m_sUserName);
    SetWindowText(sShowText);
                // Stop wave I/O devices
    if(m_bVoiceMsg)
      m_vtPlay.StopPlay();

        if(m_bVoiceTalk)
      m_vtRecord.StopRecord();

    return FALSE;

// The valid message packet
        case VTCMD_MSGDATA:
```

```
                    // Parse the message and display it
                    ParseMsgData();
        break;

                default:
                    return 0L;
        }

        return 1L;
}

/*************************************************************************
***                                                                    *
 * Response to the VTM_READYFORWRITE message

*************************************************************************
**/
LONG CTalkclientDlg::OnVTReadyForWrite(WPARAM wparam, LPARAM lparam)
{
        // Clean and reset the user name
        memset((void*)m_MySock.m_sUserName, 0, 16*sizeof(char));
        strncpy(m_MySock.m_sUserName, m_strUserName,
m_strUserName.GetLength()+1);

   // Prepare the name registration packet
   memset(&m_vtSendMsgBuf, 0, SIZEVTMSG);
        m_vtSendMsgBuf.m_ucIdentity = VT_IDENTITY;
        m_vtSendMsgBuf.m_ucCmd = VTCMD_REGNAME;
   m_vtSendMsgBuf.m_ucMsgType = VT_TEXT;
   m_vtSendMsgBuf.m_lLength = SIZEVTMSGHDR +
REALLEN(m_MySock.m_sUserName);
        memcpy((void*)m_vtSendMsgBuf.m_pData, (void*)m_MySock.m_sUserName,
                REALLEN(m_MySock.m_sUserName));


   // Send name registration packet
   if(SendVTMessage(m_MySock.m_hSock, &m_vtSendMsgBuf))
        {
      DoConnection(TRUE);
        }

        return 1L;
}
```

```
/*********************************************************************
***
 * Response to the MM_WIM_DATA (completing recording buffer) message      *
*********************************************************************
**/
LONG CTalkclientDlg::OnRecordEvent(WPARAM wparam, LPARAM lparam)
{
  // Wrong status
      if(!m_bVoiceTalk)
            return 1L;

      m_progctrlSend.SetPos(1);

      // Get WAVEHAEDER data
   WAVEHDR* lpwhdr = (WAVEHDR*)lparam;

      // Reset the voice message packet
   memset(&m_vtSendMsgBuf, 0, SIZEVTMSG);
   m_vtSendMsgBuf.m_ucIdentity = VT_IDENTITY;
   m_vtSendMsgBuf.m_ucCmd = VTCMD_MSGDATA;
   m_vtSendMsgBuf.m_ucMsgType = m_iWaveFormat;
   m_vtSendMsgBuf.m_lLength = SIZEVTMSGHDR + lpwhdr->dwBufferLength;
      memcpy((void*)m_vtSendMsgBuf.m_pData, (void*)lpwhdr->lpData,
            lpwhdr->dwBufferLength);

      // Restart to record new sound
      m_vtRecord.ResetRecord();

      // Send the voice message packet
   SendVTMessage(m_MySock.m_hSock, &m_vtSendMsgBuf);

      // Reset indicator
      m_progctrlSend.SetPos(0);

      return 1L;
}

/*********************************************************************
***
 * Response to the MM_WOM_DONE (completing playing buffer) message      *
*********************************************************************
**/
LONG CTalkclientDlg::OnPlayEvent(WPARAM wparam, LPARAM lparam)
```

```
{
  // Wrong status
  if(!m_bVoiceMsg)
    return 0L;

      // Reset indicator
  m_progctrlRecv.SetPos(0);

      // Reset playing device
      m_vtPlay.ResetPlay();
  // Copy the new sound data to output device buffer
  m_vtPlay.SetPlayData(m_pWaveData, m_uiBufLen);
      // Restart play sound
      m_vtPlay.RestartPlay();

      return 1L;
}

/*****************************************************************************
***                                                                        *
 * Input new user name

*****************************************************************************
**/
void CTalkclientDlg::OnChangeEditname()
{
      // TODO: If this is a RICHEDIT control, the control will not
      // send this notification unless you override the CDialog::OnInitDialog()
      // function and call CRichEditCtrl().SetEventMask()
      // with the ENM_CHANGE flag ORed into the mask.

      // TODO: Add your control notification handler code here
      m_editUserName.GetWindowText(m_strUserName);
}

/*****************************************************************************
***                                                                        *
 * Input new server name

*****************************************************************************
**/
void CTalkclientDlg::OnChangeEditserver()
{
      // TODO: If this is a RICHEDIT control, the control will not
```

```
        // send this notification unless you override the CDialog::OnInitDialog()
        // function and call CRichEditCtrl().SetEventMask()
        // with the ENM_CHANGE flag ORed into the mask.

        // TODO: Add your control notification handler code here
        m_editServer.GetWindowText(m_strServer);

}

/*************************************************************************
***
 * Selected the network protocol                                        *
*************************************************************************
**/
void CTalkclientDlg::OnSelchangeCombprot()
{
        // TODO: Add your control notification handler code here
   int iIndex;

        iIndex = m_ComBoProts.GetCurSel();

        if(iIndex == LB_ERR)
        {
                AfxMessageBox("An error ocurred in slecting protocols!");
                return;
        }

   // Protocol selected!  Check which one...
   switch(m_lpProtBuf[m_iWorkProts[iIndex]].iAddressFamily)
   {
     case AF_IPX:
     case AF_NETBIOS:
       // For IPX and NetBIOS, we don't need the server name
                // so disable that edit control
                m_editServer.SetWindowText("");
                m_editServer.EnableWindow(FALSE);
       break;

     case AF_INET:
            // For TCP/IP we need a protocol, name, and machine name
                    // in order for data to be valid
                m_editServer.EnableWindow(TRUE);
       break;
     default: // Nothing
```

```
                m_editServer.SetWindowText("");
                m_editServer.EnableWindow(FALSE);
        break;
    }
}

/*************************************************************************
***                                                                    *
 * Selected the peer user

*************************************************************************
**/
void CTalkclientDlg::OnSelchangeCombchatter()
{
        // TODO: Add your control notification handler code here
    int   iIndex;
    int   iRev;
        char  sNewPeer[16];

        memset(sNewPeer, 0, 16);

        iIndex = m_ComBoTalker.GetCurSel();

        if(iIndex == LB_ERR)
        {
                AfxMessageBox("An error ocurred in slecting peer talker");
                return;
        }

        // Exit current session
        if(iIndex == 0)
        {
                if(m_bTalking)
                {
        // Prepare the "session close " message
                  // and send it out
        memset(&m_vtSendMsgBuf, 0, SIZEVTMSG);
                  m_vtSendMsgBuf.m_ucIdentity = VT_IDENTITY;
                  m_vtSendMsgBuf.m_ucCmd = VTCMD_SESSIONCLOSE;
        m_vtSendMsgBuf.m_ucMsgType = VT_TEXT;
        m_vtSendMsgBuf.m_lLength = SIZEVTMSGHDR;
        if(SendVTMessage(m_MySock.m_hSock, &m_vtSendMsgBuf))
                  {
                        // Reset socket state
                        if(WSAAsyncSelect(m_MySock.m_hSock,
```

```
                                        GetSafeHwnd(),
                                                        0,
                                                        0) != SOCKET_ERROR)

                        {
        EnableTalking(FALSE);
                                return;
                        }
                    }
                }
                return;
    }
else
    {
  // Get peer user name
        iRev = m_ComBoTalker.GetLBText(iIndex, sNewPeer);
        if(iRev == CB_ERR)
        {
     AfxMessageBox("An error ocurred in getting new talker's name");
                return;
        }

        // No change
        if(lstrcmp(sNewPeer, m_lpPeerName) == 0)
        {
     return;
   }

        // Store the new peer's name
        lstrcpy(m_lpPeerName, sNewPeer);

        // Prepare the "session request" message and
        // send it out
        m_MySock.m_iStatus = VTSOCKET_REQSESSION;
   memset(&m_vtSendMsgBuf, 0, SIZEVTMSG);
   m_vtSendMsgBuf.m_ucIdentity = VT_IDENTITY;
        m_vtSendMsgBuf.m_ucCmd = VTCMD_REQSESSION;
   m_vtSendMsgBuf.m_ucMsgType = VT_TEXT;
   m_vtSendMsgBuf.m_lLength = SIZEVTMSGHDR + REALLEN(sNewPeer);
        memcpy((void*)m_vtSendMsgBuf.m_pData, (void*)sNewPeer,
                REALLEN(sNewPeer));

   if(SendVTMessage(m_MySock.m_hSock, &m_vtSendMsgBuf))
   {
                // Reset the socket state
                if(WSAAsyncSelect(m_MySock.m_hSock,
                                GetSafeHwnd(),
```

```
                                                    VTM_DATAREADY,
                                                    FD_READ | FD_CLOSE) !=
SOCKET_ERROR)
                {
        EnableTalking(TRUE);
                        return;
                }
        }
    }
}

/*************************************************************************
***
 * Clicked the "Connect" button *
*************************************************************************
**/
void CTalkclientDlg::OnBtnconnect()
{
        // TODO: Add your control notification handler code here
   int iIndex;

        iIndex = m_ComBoProts.GetCurSel();

        // No protocol selected
        if(iIndex == LB_ERR)
        {
                AfxMessageBox("Did not select a protocol!");
                return;
        }

        // No user name
        if(m_strUserName.IsEmpty())
        {
                AfxMessageBox("Did not input a user name");
                return;
        }

        // No server name for TCP/IP
        if(m_lpProtBuf[m_iWorkProts[iIndex]].iAddressFamily
          == AF_INET && m_strServer.IsEmpty())
        {
                AfxMessageBox("Did not input a server name");
                return;
        }
```

```
        // Set up socket
        if(!InitSockets(m_iWorkProts[iIndex]))
        {
                //DoConnection(TRUE);
                AfxMessageBox("Can't bulid connection with server!");
        }
}

/*****************************************************************************
***                                                                        *
 * Clicked the "Disconnect" button

*****************************************************************************
**/
void CTalkclientDlg::OnBtndisconnect()
{
        // TODO: Add your control notification handler code here
        // Disconnect
        if(m_bConnect)
    DoConnection(FALSE);
}

/*****************************************************************************
***                                                                        *
 * Clicked the "Send text message" button

*****************************************************************************
**/
void CTalkclientDlg::OnBtnsend()
{
        // TODO: Add your control notification handler code here
        CString strSend;

        // Get text message
        m_TextSend.GetWindowText(strSend);

        // Prepare the text message packet
   memset(&m_vtSendMsgBuf, 0, SIZEVTMSG);
   m_vtSendMsgBuf.m_ucIdentity = VT_IDENTITY;
   m_vtSendMsgBuf.m_ucCmd = VTCMD_MSGDATA;
   m_vtSendMsgBuf.m_ucMsgType = VT_TEXT;
   m_vtSendMsgBuf.m_lLength = SIZEVTMSGHDR + strSend.GetLength()+1;
        lstrcpy((char*)m_vtSendMsgBuf.m_pData, strSend);
```

```
        // Sen the text message packet
    SendVTMessage(m_MySock.m_hSock, &m_vtSendMsgBuf);

}

/*************************************************************************
***
 * Checked/Unchecked the "Voice Talk" radio button                        *

*************************************************************************
**/
void CTalkclientDlg::OnChkvoice()
{
        // TODO: Add your control notification handler code here
        // Wrong state
        if(!m_bConnect)
                return;

        // If unchecked previously
        if(!m_bVoiceTalk)
    {
                // Check it
      m_bVoiceTalk = TRUE;
                m_chkVoice.SetCheck(1);
                // And disable "Send text message" button
                m_btnSendTxtMsg.EnableWindow(FALSE);
                // Start to reord voice message
            m_vtRecord.StartRecord();
        }
        else // If checked previously
        {
                // Uncheck it
      m_bVoiceTalk = FALSE;
                m_chkVoice.SetCheck(0);
      // And enable "Send text message" button
                m_btnSendTxtMsg.EnableWindow(TRUE);
                // stop reording voice message
                m_vtRecord.StopRecord();
        }

 }
```

Server

```
#include "stdafx.h"
#include "talkserver.h"
#include "talkserverDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/*****************************************************************************
***
 *                                                                           *
 * CAboutDlg dialog used for App About                                       *
 *                                                                           *

*****************************************************************************
**/
class CAboutDlg : public CDialog
{
public:
        CAboutDlg();

// Dialog Data
        //{{AFX_DATA(CAboutDlg)
        enum { IDD = IDD_ABOUTBOX };
        //}}AFX_DATA

        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CAboutDlg)
        protected:
        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
        //}}AFX_VIRTUAL

// Implementation
protected:
        //{{AFX_MSG(CAboutDlg)
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
```

```
        // Sen the text message packet
    SendVTMessage(m_MySock.m_hSock, &m_vtSendMsgBuf);

}

/*****************************************************************************
***
 * Checked/Unchecked the "Voice Talk" radio button                          *

*****************************************************************************
**/
void CTalkclientDlg::OnChkvoice()
{
        // TODO: Add your control notification handler code here
        // Wrong state
        if(!m_bConnect)
                return;

        // If unchecked previously
        if(!m_bVoiceTalk)
    {
                // Check it
      m_bVoiceTalk = TRUE;
                m_chkVoice.SetCheck(1);
                // And disable "Send text message" button
                m_btnSendTxtMsg.EnableWindow(FALSE);
                // Start to reord voice message
            m_vtRecord.StartRecord();
        }
        else // If checked previously
        {
                // Uncheck it
      m_bVoiceTalk = FALSE;
                m_chkVoice.SetCheck(0);
      // And enable "Send text message" button
                m_btnSendTxtMsg.EnableWindow(TRUE);
                // stop reording voice message
                m_vtRecord.StopRecord();
        }

  }
```

```
        //{{AFX_DATA_INIT(CAboutDlg)
        //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(CAboutDlg)
        //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
        //{{AFX_MSG_MAP(CAboutDlg)
                // No message handlers
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()


/*****************************************************************************
***
 *                                                *
 *                                                        *
 * CTalkserverDlg dialog                          *
 *

 *****************************************************************************
 **/
 CTalkserverDlg::CTalkserverDlg(CWnd* pParent /*=NULL*/)
        : CDialog(CTalkserverDlg::IDD, pParent)
 {
        //{{AFX_DATA_INIT(CTalkserverDlg)
                // NOTE: the ClassWizard will add member initialization here
        //}}AFX_DATA_INIT
        // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
        m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);

        m_iNextFree = 0;
        m_iMaxConnects = 5;
    m_iWorkProt = 0;
        m_vtMsgBuf = new VTMSG;
        memset((void*)m_vtMsgBuf, 0, sizeof(SIZEVTMSG));
  }

  CTalkserverDlg::~CTalkserverDlg()
  {
  }
```

```
void CTalkserverDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CTalkserverDlg)
	DDX_Control(pDX, IDC_CLIENT, m_ClientList);
	DDX_Control(pDX, IDC_PROTOCOL, m_ProtList);
	//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CTalkserverDlg, CDialog)
	//{{AFX_MSG_MAP(CTalkserverDlg)
	ON_WM_SYSCOMMAND()
	ON_WM_PAINT()
	ON_WM_QUERYDRAGICON()
	ON_WM_TIMER()
	ON_MESSAGE(VTM_CONNECTED, OnVTConnected)
	ON_MESSAGE(VTM_DATAREADY, OnVTDataReady)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/*****************************************************************************
***
 * Initialize the dialogbox                                                  *

*****************************************************************************
**/
BOOL CTalkserverDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Add "About..." menu item to system menu.

	// IDM_ABOUTBOX must be in the system command range.
	ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
	ASSERT(IDM_ABOUTBOX < 0xF000);

	CMenu* pSysMenu = GetSystemMenu(FALSE);
	if (pSysMenu != NULL)
	{
		CString strAboutMenu;
		strAboutMenu.LoadString(IDS_ABOUTBOX);
		if (!strAboutMenu.IsEmpty())
		{
			pSysMenu->AppendMenu(MF_SEPARATOR);
			pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
 strAboutMenu);
```

```
            }
        }

        // Set the icon for this dialog.  The framework does this automatically
        //  when the application's main window is not a dialog
        SetIcon(m_hIcon, TRUE);                     // Set big icon
        SetIcon(m_hIcon, FALSE);            // Set small icon

        // TODO: Add extra initialization here

        // Initialize the client sockets
    if(!InitClientSocks())
    {
                AfxMessageBox("Can't build connection sockets!");
                return FALSE;
        }

        // Initialize the server sockets
        if(!InitServSocks())
        {
                AfxMessageBox("Can't build server sockets!");
                return FALSE;
        }

        // Fill the compatible protocol list
        char strProt[128];
        for(int iLoop = 0; iLoop < m_iWorkProt; iLoop++)
        {
      wsprintf(strProt, "%s:  %d", m_lpServSocks[iLoop].m_lpProtocolName,
                        m_lpServSocks[iLoop].m_iConnects);
                m_ProtList.AddString(strProt);
        }

        return TRUE;  // return TRUE  unless you set the focus to a control
}


/*****************************************************************************
***
 * System menu commands                                                      *

*****************************************************************************
**/
void CTalkserverDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
        if ((nID & 0xFFF0) == IDM_ABOUTBOX)
```

```
            CDC dcMem;
            CBitmap bmp;
            bmp.LoadBitmap(IDB_BGBMP);

            BITMAP bm;
            bmp.GetObject(sizeof(BITMAP),&bm);

            dcMem.CreateCompatibleDC(&dc);
            dcMem.SelectObject(&bmp);

      dc.StretchBlt(0,0,rc.right,rc.bottom,&dcMem,0,0,bm.bmWidth,bm.bmHeight,SR
CCOPY);


            CDialog::OnPaint();
      }
}

/*************************************************************************
***
 * The system calls this to obtain the cursor to display while the user   *
 * drags the minimized window.                                            *
*************************************************************************
**/
HCURSOR CTalkserverDlg::OnQueryDragIcon()
{
      return (HCURSOR) m_hIcon;
}


/*************************************************************************
***
 * Initialize the server sockets                                          *
*************************************************************************
**/
BOOL CTalkserverDlg::InitServSocks(void)
{
   unsigned long  sizeProtBuf = 0;
   int        cNumProt;
   int        iLoop1;
      int        iLoop2;
```

```
int         iAddrLen;
WORD        wVersionRequested;
WSADATA     wsaData;
BOOL        bCastStat;

// Load winsock dll
wVersionRequested = MAKEWORD(1, 1);
if(WSAStartup(wVersionRequested, &wsaData) != 0)
{
  // ERROR
        return FALSE;
}

// Determine size of buffer needed
if(EnumProtocols(NULL, NULL, &sizeProtBuf) != SOCKET_ERROR)
{
  // ERROR
  return FALSE;
}

// Allocate buffer of appropriate size
if((m_lpProtBuf = (LPPROTOCOL_INFO)VirtualAlloc(NULL,
  sizeProtBuf, MEM_COMMIT, PAGE_READWRITE)) == NULL)
{
  // ERROR
  return FALSE;
}

// Enumerate numbers of protocols
if((cNumProt = EnumProtocols(NULL, m_lpProtBuf,
      &sizeProtBuf)) == SOCKET_ERROR)
{
  // ERROR
  return FALSE;
}

// Allocate the buffer for server sockets
if((m_lpServSocks = (LPVTSOCKET)VirtualAlloc(NULL,
        cNumProt * sizeof(VTSOCKET), MEM_COMMIT,
  PAGE_READWRITE)) == NULL)
{
  // ERROR
  return FALSE;
}

// Retrieve the worlable protocols
```

```
for(iLoop1 = 0, iLoop2 = 0; iLoop1 < cNumProt; iLoop1++)
{
  // Only want connection oriented protocols (UDP)
  if((m_lpProtBuf[iLoop1].dwServiceFlags & XP_CONNECTIONLESS)
                    == 0)
  {
                    //Set up the server sockets
                    if(BuildServer(this->GetSafeHwnd(), &m_lpServSocks[iLoop2],
                      &m_lpProtBuf[iLoop1]))
                    {
            iLoop2++;
                    }
  }
}

     // Store the numbers of workable protocol
m_iWorkProt =  iLoop2;


// Register with Name Spaces
if((m_SAPSocket = socket(AF_IPX, SOCK_DGRAM,
            NSPROTO_IPX + 4)) != INVALID_SOCKET)
{
  // Find IPX address which we are bound to
  for(iLoop1 = 0; iLoop1 < m_iWorkProt; iLoop1++)
  {
    if(m_lpServSocks[iLoop1].m_iProtocol == NSPROTO_SPXII)
    {
                // Initialize SAP data

                        // 2: SAP response(network order)
                m_vtSAPData.m_iOperation = 0x200;

                        // Network order
                        m_vtSAPData.m_iService = VT_NETID;

                        // Service name
                        strcpy(m_vtSAPData.m_sServName, "VOICE TALK");

                        // Copy address into SAP body
                memcpy((void *)m_vtSAPData.m_sNetID,
                    (void *)(m_lpServSocks[iLoop1].m_sockAddr.sa_data),
                    12);

                        // Hop count starts at zero
                m_vtSAPData.m_iHops = 0;
```

```
                    // SAP packets sent over IPX
                m_SAPSockAddr.sa_family = AF_IPX;

        // SAP destination socket address is 0452.
                Ascii2Hex("0452", (char *)&m_SAPSockAddr.sa_socket, 2);

                    // Bind to SAP socket
                if(bind(m_SAPSocket, (PSOCKADDR)&m_SAPSockAddr,
                                sizeof(SOCKADDR_IPX)) ==
SOCKET_ERROR)
                {
                  // Error binding to SAP socket
                                return FALSE;
                }
                else
                {
                  // Build destination address for SAP's
                                // IPX header (broadcast)
                  m_SAPDestSockAddr.sa_family = AF_IPX;
                  Ascii2Hex("00000000",m_SAPDestSockAddr.sa_netnum, 4);
                  Ascii2Hex("FFFFFFFFFFFF",
m_SAPDestSockAddr.sa_nodenum, 6);
                  Ascii2Hex("0452", (char *)&m_SAPDestSockAddr.sa_socket, 2);

                // Setup socket to allow broadcasts
                bCastStat = TRUE;
                iAddrLen = sizeof(bCastStat);
                setsockopt(m_SAPSocket, SOL_SOCKET, SO_BROADCAST,
                                        (char *)&bCastStat, sizeof(bCastStat));
                  sendto(m_SAPSocket, (char *)&m_vtSAPData,
sizeof(m_vtSAPData),
                                        MSG_DONTROUTE, (struct sockaddr
*)&m_SAPDestSockAddr,
                                        sizeof(m_SAPDestSockAddr));

                  // Set timer to send SAP packet every 60 seconds.
                  SetTimer(1, 60000, NULL);
                }
          break;
        }
      }
    }

        return TRUE;
  }
```

```
/*****************************************************************************
***
 * Initialize the clients sockets                                            *
*****************************************************************************
**/
BOOL CTalkserverDlg::InitClientSocks(void)
{
   // First allocate Connected Sockets Heap
   if((m_hConnectHeap = HeapCreate(0, sizeof(VTSOCKET) * 5,
                           sizeof(VTSOCKET) * 100)) == NULL)
   {
     // ERROR
              return FALSE;
   }
   if((m_lpClientSocks = (LPVTSOCKET)HeapAlloc(m_hConnectHeap, 0,
                        sizeof(VTSOCKET) * 5)) == NULL)
   {
     // ERROR
     return FALSE;
   }

        return TRUE;
}


/*****************************************************************************
***
 * Remove the deregistered client name from client list                      *
*****************************************************************************
**/
void CTalkserverDlg::DeRegisterClient(char* sClientName)
{
   int iLoop;

   // Prepare the deregister name message
   m_vtMsgBuf->m_ucIdentity = VT_IDENTITY;
   m_vtMsgBuf->m_ucMsgType = VT_TEXT;
   m_vtMsgBuf->m_lLength = REALLEN(sClientName) + SIZEVTMSGHDR;
   m_vtMsgBuf->m_ucCmd = VTCMD_DEREGNAME;
   memcpy((void*)m_vtMsgBuf->m_pData, (void*)sClientName,
                  REALLEN(sClientName));
```

```
  // Send to every connected client
        // whose status is SOCKET_AVAILABLE
  for(iLoop = 0; iLoop < m_iNextFree; iLoop++)
  {
     if(m_lpClientSocks[iLoop].m_iStatus
                        == VTSOCKET_AVAILABLE)
     {
        // Send out the deregister name message
        SendVTMessage(m_lpClientSocks[iLoop].m_hSock, m_vtMsgBuf);
     }
   }
}


/*************************************************************************
***
 * Update protocol list                                                  *

*************************************************************************
**/
void CTalkserverDlg::UpdateProtList(int index)
{
        int iReVal;
        char strProt[128];

        iReVal = m_ProtList.FindString(-1,
                    m_lpServSocks[index].m_lpProtocolName);

        // Found nothing
    if(iReVal == LB_ERR)
                return;

    m_ProtList.DeleteString(iReVal);
    wsprintf(strProt, "%s:  %d", m_lpServSocks[index].m_lpProtocolName,
                              m_lpServSocks[index].m_iConnects);
         m_ProtList.InsertString(iReVal, strProt);
}


/*************************************************************************
***
 * Update client list                                                    *

*************************************************************************
**/
```

```
void CTalkserverDlg::UpdateClientList(char* name,
                                      int status, char*
peername)
{
   int index;
   char strClient[128];

        // Find the client
        index = m_ClientList.FindString(-1, name);

        // Found it
        if(index != LB_ERR)
        {
                m_ClientList.DeleteString(index);
        }

        switch(status)
    {
                // The client disconnected
           case VTSOCKET_CLOSED:
                        return;

      // Build available client entry
                 case VTSOCKET_AVAILABLE:
                wsprintf(strClient, "%s  AVAILABLE", name);
                break;

      // Build reqsession entry
                 case VTSOCKET_REQSESSION:
                        wsprintf(strClient, "%s  SESSION SETUP  %s",
                                name, peername);
          break;

                        // Build in session entry
                 case VTSOCKET_INSESSION:
                        wsprintf(strClient, "%s  in SESSION", name);
          break;
          }

          m_ClientList.InsertString(index, strClient);
   }


   /*****************************************************************************
   ***                                                                        *
    * Timer event
```

```
**********************************************************************
**/
void CTalkserverDlg::OnTimer(UINT nIDEvent)
{
        // TODO: Add your message handler code here and/or call default
        // Send out SAP package
   sendto(m_SAPSocket, (char *)&m_vtSAPData, sizeof(m_vtSAPData),
                    MSG_DONTROUTE, (struct sockaddr *)&m_SAPDestSockAddr,
                    sizeof(m_SAPDestSockAddr));

        CDialog::OnTimer(nIDEvent);
}


/**********************************************************************
***
 * Response to the VTM_CONNECTED message                                *

**********************************************************************
**/
LONG CTalkserverDlg::OnVTConnected(WPARAM wparam, LPARAM lparam)
{
   int index, addrlen;

   // May have to increase our heap size
   if(m_iNextFree >= m_iMaxConnects)
   {
     if(HeapAlloc(m_hConnectHeap, 0, sizeof(VTSOCKET) * 5) == NULL)
     {
       // ERROR
                return 0L;
     }
     m_iMaxConnects += 5;
   }

   // Find socket that connect is refering to
   index = 0;
   while(m_lpServSocks[index].m_hSock != wparam)
                index++;

   // Lets keep track of our progress
   m_lpServSocks[index].m_iStatus = VTSOCKET_ACCEPTING;

   // If the address is bigger than a sockaddr struct.
        // use the reserved byte to cover it.
```

```
addrlen = sizeof(m_lpClientSocks[m_iNextFree].m_sockAddr)
                + sizeof(m_lpClientSocks[m_iNextFree].m_Reserved);

// Accept the connect request
if ((m_lpClientSocks[m_iNextFree].m_hSock =
            accept(m_lpServSocks[index].m_hSock,
            &m_lpClientSocks[m_iNextFree].m_sockAddr,
   &addrlen)) == INVALID_SOCKET)
{
  // ERROR
            AfxMessageBox("Can not accept the client socket");
  return 0L;
}

      // Set client socket state
      if(WSAAsyncSelect(m_lpClientSocks[m_iNextFree].m_hSock,
            this->GetSafeHwnd(), VTM_DATAREADY, FD_READ | FD_CLOSE)
            == SOCKET_ERROR)
 {
   // ERROR clean up connection
            closesocket(m_lpClientSocks[m_iNextFree].m_hSock);
            AfxMessageBox("Can not select the client socket");
   return 0L;
 }
 // Fill in socket structure for client socket
 m_lpClientSocks[m_iNextFree].m_iProtocol
            = m_lpServSocks[index].m_iProtocol;

 m_lpClientSocks[m_iNextFree].m_iSockType
            = m_lpServSocks[index].m_iSockType;

      m_lpClientSocks[m_iNextFree].m_iServSockIndex
            = index;

      m_lpClientSocks[m_iNextFree++].m_iStatus
            = VTSOCKET_CONNECTED;

      // Increment protocol connection count and display
  m_lpServSocks[index].m_iConnects ++;

      // Don't forget to set server socket status back to listening
  m_lpServSocks[index].m_iStatus = VTSOCKET_LISTENING;

      //Update the protocol list
  UpdateProtList(index);
```

```
        return 1L;
}


/************************************************************************
***
 * Response to the VTM_DATAREADY message                                *

************************************************************************
**/
LONG CTalkserverDlg::OnVTDataReady(WPARAM wparam, LPARAM lparam)
{
   int iIndex1;
        int iIndex2;
        int iIndex3;
    int byteRead;
        int iStrLen;


    // Find the appropriate socket...allow
        // for the reuse of a closed socket handle
    for(iIndex1 = 0; iIndex1 < m_iNextFree; iIndex1++)
    {
      if((m_lpClientSocks[iIndex1].m_hSock == wparam)
                        && (m_lpClientSocks[iIndex1].m_iStatus !=
 VTSOCKET_CLOSED))
         break;
    }

    if(LOWORD(lparam) == FD_CLOSE)
    {
      // Socket closed notification--cleanup!
      closesocket(m_lpClientSocks[iIndex1].m_hSock);

      // Delete name from client list display
      UpdateClientList(m_lpClientSocks[iIndex1].m_sUserName,
                                VTSOCKET_CLOSED, NULL);

      // Take name off of other clients' available list
      if(m_lpClientSocks[iIndex1].m_iStatus == VTSOCKET_AVAILABLE)
      {
        DeRegisterClient(m_lpClientSocks[iIndex1].m_sUserName);
      }

      // if this connection was in session with another peer...
                // notify peer of disconnect
```

```
// First, find peer
for(iIndex2 = 0; iIndex2 < m_iNextFree; iIndex2++)
{
  if((m_lpClientSocks[iIndex2].m_hSock ==
                    m_lpClientSocks[iIndex1].m_hPeerSock) &&
     (m_lpClientSocks[iIndex2].m_iStatus ==
                    VTSOCKET_INSESSION))
  {
    break;
  }
}
        // Did we find a peer?
if(iIndex2 < m_iNextFree)
{
  // Yes, build message information
  m_vtMsgBuf->m_ucIdentity = VT_IDENTITY;
  m_vtMsgBuf->m_ucMsgType = VT_TEXT;
  m_vtMsgBuf->m_lLength = byteRead = SIZEVTMSGHDR;
  m_vtMsgBuf->m_ucCmd = VTCMD_SESSIONCLOSE;

                // Send session close message
  SendVTMessage(m_lpClientSocks[iIndex2].m_hSock, m_vtMsgBuf);

                // Update Peer's displayed status
  m_lpClientSocks[iIndex2].m_iStatus = VTSOCKET_AVAILABLE;
                UpdateClientList(m_lpClientSocks[iIndex2].m_sUserName,
                                VTSOCKET_AVAILABLE, NULL);

                // Propagate the fact that the peer is now available for other chats
                // and give peer list of other available clients
  for(iIndex3 = 0; iIndex3 < m_iNextFree; iIndex3++)
  {
    if ((iIndex3 != iIndex2) &&
                                (m_lpClientSocks[iIndex3].m_iStatus ==
VTSOCKET_AVAILABLE))
      {
        m_vtMsgBuf->m_ucCmd = VTCMD_REGNAME;
        memcpy((void*)m_vtMsgBuf->m_pData,

(void*)m_lpClientSocks[iIndex3].m_sUserName,

REALLEN(m_lpClientSocks[iIndex3].m_sUserName)*sizeof(char));
        m_vtMsgBuf->m_lLength =
REALLEN(m_lpClientSocks[iIndex3].m_sUserName)
                                + SIZEVTMSGHDR;
```

```
            SendVTMessage(m_lpClientSocks[iIndex2].m_hSock, m_vtMsgBuf);

            memcpy((void*)m_vtMsgBuf->m_pData,

(void*)m_lpClientSocks[iIndex2].m_sUserName,

REALLEN(m_lpClientSocks[iIndex2].m_sUserName)*sizeof(char));
            m_vtMsgBuf->m_lLength =
REALLEN(m_lpClientSocks[iIndex2].m_sUserName)
                                                    + SIZEVTMSGHDR;
            SendVTMessage(m_lpClientSocks[iIndex3].m_hSock, m_vtMsgBuf);
          }
        }
      }
      // Cleanup the client sockets array
      m_lpClientSocks[iIndex1].m_iStatus = VTSOCKET_CLOSED;
      iIndex2 = m_lpClientSocks[iIndex1].m_iServSockIndex;
          // Fix protocol connection count display
          m_lpServSocks[iIndex2].m_iConnects --;

      UpdateProtList(iIndex2);

                return 1L;
    }


        // Read incoming message data
    if(!RecvVTMessage(&m_lpClientSocks[iIndex1], m_vtMsgBuf))
    {
      return 0L;
    }

    switch(m_vtMsgBuf->m_ucCmd)
    {
                // First message we should receive on a connection
      case VTCMD_REGNAME:
        if(m_lpClientSocks[iIndex1].m_iStatus != VTSOCKET_CONNECTED)
        {
          // ERROR
          AfxMessageBox("The client not connected");
          return 0L;
        }


                    // Get name and add to internal structs and display
        iStrLen = m_vtMsgBuf->m_lLength - SIZEVTMSGHDR;
```

```
    case VTCMD_MSGDATA:
      if(m_lpClientSocks[iIndex1].m_iStatus != VTSOCKET_INSESSION)
      {
        // ERROR
        return 0L;
      }

      // forward the message to peer...should be able transfer
      // message without modification
      SendVTMessage(m_lpClientSocks[iIndex1].m_hPeerSock, m_vtMsgBuf);
      return 1L;

    // Client is asking another peer for a chat
    case VTCMD_REQSESSION:
      if(m_lpClientSocks[iIndex1].m_iStatus != VTSOCKET_AVAILABLE)
      {
        // ERROR
                              AfxMessageBox("Client not available!");
        return 0L;
      }

      m_lpClientSocks[iIndex1].m_iStatus = VTSOCKET_REQSESSION;
      UpdateClientList(m_lpClientSocks[iIndex1].m_sUserName,
                                    VTSOCKET_REQSESSION,
(char*)m_vtMsgBuf->m_pData);

      // Find the socket which corresponds to the name

      for(iIndex2 = 0; iIndex2 < m_iNextFree; iIndex2++)
      {
        iStrLen = m_vtMsgBuf->m_lLength - SIZEVTMSGHDR;
                    if(memcmp((void*)m_lpClientSocks[iIndex2].m_sUserName,
                                (void*)m_vtMsgBuf->m_pData, iStrLen) == 0)
        {
          if(m_lpClientSocks[iIndex2].m_iStatus ==
                                    VTSOCKET_AVAILABLE)
          {
            // Found It!
            break;
          }
        }
      }

                    // Found nothing
      if(iIndex2 == m_iNextFree)
                    {
```

```
                    memcpy((void*)m_lpClientSocks[iIndex1].m_sUserName,
                                (void*)m_vtMsgBuf->m_pData, iStrLen+1);
        m_lpClientSocks[iIndex1].m_iStatus = VTSOCKET_AVAILABLE;

                    UpdateClientList(m_lpClientSocks[iIndex1].m_sUserName,
                                VTSOCKET_AVAILABLE, NULL);

        // Send notification to other "AVAILABLE" sockets that
                    // we have a new peer available
        for(iIndex2 = 0; iIndex2 < m_iNextFree; iIndex2++)
        {
          if ((iIndex2 != iIndex1) &&
                                (m_lpClientSocks[iIndex2].m_iStatus ==
VTSOCKET_AVAILABLE))
          {
            // message should be able to be sent just like it is
            SendVTMessage(m_lpClientSocks[iIndex2].m_hSock, m_vtMsgBuf);
          }
        }

        // Send notifications back to registering peer of
                    // all the currently available peers
        for(iIndex2 = 0; iIndex2 < m_iNextFree; iIndex2++)
        {
          if ((iIndex2 != iIndex1) &&
                                (m_lpClientSocks[iIndex2].m_iStatus ==
VTSOCKET_AVAILABLE))
          {
            // found one...build message and send it
            m_vtMsgBuf->m_ucCmd = VTCMD_REGNAME;
            m_vtMsgBuf->m_ucMsgType = VT_TEXT;
                                m_vtMsgBuf->m_lLength =
REALLEN(m_lpClientSocks[iIndex2].m_sUserName)
                                                + SIZEVTMSGHDR;
            memcpy((void*)m_vtMsgBuf->m_pData,

(void*)m_lpClientSocks[iIndex2].m_sUserName,

REALLEN(m_lpClientSocks[iIndex2].m_sUserName)*sizeof(char));
            SendVTMessage(m_lpClientSocks[iIndex1].m_hSock, m_vtMsgBuf);
          }
        }

                    return 1L;

      // For passing data between two insession peers
```

```
                    AfxMessageBox("Not find in VTCMD_REQSESSION");
                    return 0L;
                }

        // Copy requester's name into send data buffer
        m_vtMsgBuf->m_lLength =
REALLEN(m_lpClientSocks[iIndex1].m_sUserName)
                                                + SIZEVTMSGHDR;

        memcpy((void*)m_vtMsgBuf->m_pData,

(void*)m_lpClientSocks[iIndex1].m_sUserName,

REALLEN(m_lpClientSocks[iIndex1].m_sUserName)*sizeof(char));
        SendVTMessage(m_lpClientSocks[iIndex2].m_hSock, m_vtMsgBuf);

        // Update connected sockets structures
        m_lpClientSocks[iIndex2].m_iStatus = VTSOCKET_REQSESSION;
        UpdateClientList(m_lpClientSocks[iIndex2].m_sUserName,
                                VTSOCKET_REQSESSION,
m_lpClientSocks[iIndex1].m_sUserName);
        m_lpClientSocks[iIndex2].m_hPeerSock = m_lpClientSocks[iIndex1].m_hSock;
        m_lpClientSocks[iIndex1].m_hPeerSock = m_lpClientSocks[iIndex2].m_hSock;
        return 1L;

      // Response to session request
      case VTCMD_SESSIONREQRESP:
        if(m_lpClientSocks[iIndex1].m_iStatus != VTSOCKET_REQSESSION)
        {
          // ERROR
          return 0L;
        }

        // find peer entry
        for(iIndex2 = 0; iIndex2 < m_iNextFree; iIndex2++)
        {
          if ((m_lpClientSocks[iIndex2].m_hSock ==
                                m_lpClientSocks[iIndex1].m_hPeerSock)
                               && (m_lpClientSocks[iIndex2].m_iStatus ==
 VTSOCKET_REQSESSION))
          {
            // Found it!
            break;
          }
        }

        if(iIndex2 == m_iNextFree)
```

```
        {
          // ERROR
          return 0;
        }

        // forward response to requester
        SendVTMessage(m_lpClientSocks[iIndex1].m_hPeerSock, m_vtMsgBuf);

        if(m_vtMsgBuf->m_pData[0] == 1)
        {
          // Session accepted, change status of sockets
          m_lpClientSocks[iIndex1].m_iStatus = VTSOCKET_INSESSION;
          m_lpClientSocks[iIndex2].m_iStatus = VTSOCKET_INSESSION;


        UpdateClientList(m_lpClientSocks[iIndex1].m_sUserName,
                                    VTSOCKET_INSESSION,
m_lpClientSocks[iIndex2].m_sUserName);

        UpdateClientList(m_lpClientSocks[iIndex2].m_sUserName,
                                    VTSOCKET_INSESSION,
m_lpClientSocks[iIndex1].m_sUserName);


        DeRegisterClient(m_lpClientSocks[iIndex1].m_sUserName);

        DeRegisterClient(m_lpClientSocks[iIndex2].m_sUserName);
        }
        else
        {
          // Session not accepted, make sockets available
          m_lpClientSocks[iIndex1].m_iStatus = VTSOCKET_AVAILABLE;
          m_lpClientSocks[iIndex2].m_iStatus = VTSOCKET_AVAILABLE;

        UpdateClientList(m_lpClientSocks[iIndex1].m_sUserName,
                                    VTSOCKET_AVAILABLE, NULL);

        UpdateClientList(m_lpClientSocks[iIndex2].m_sUserName,
                                    VTSOCKET_AVAILABLE, NULL);
        }

        return 1L;

      // Insession client chose "End Chat" option
              case VTCMD_SESSIONCLOSE:
        if (m_lpClientSocks[iIndex1].m_iStatus != VTSOCKET_INSESSION)
```

```
    {
      // Can't close
      return 0L;
    }

    // Find Peer
    for(iIndex2 = 0; iIndex2 < m_iNextFree; iIndex2++)
    {
      if ((m_lpClientSocks[iIndex2].m_hSock ==
                                    m_lpClientSocks[iIndex1].m_hPeerSock)
                                   && (m_lpClientSocks[iIndex2].m_iStatus ==
VTSOCKET_INSESSION))
      {
        // Found it
        break;
      }
    }

    if(iIndex2 == m_iNextFree)
    {
      // ERROR
      return 0;
    }

    // forward message
    SendVTMessage(m_lpClientSocks[iIndex1].m_hPeerSock, m_vtMsgBuf);

    // Change Status
    m_lpClientSocks[iIndex1].m_iStatus = VTSOCKET_AVAILABLE;
    m_lpClientSocks[iIndex2].m_iStatus = VTSOCKET_AVAILABLE;
                UpdateClientList(m_lpClientSocks[iIndex1].m_sUserName,
                                    VTSOCKET_AVAILABLE, NULL);
                UpdateClientList(m_lpClientSocks[iIndex2].m_sUserName,
                                    VTSOCKET_AVAILABLE, NULL);

    // register names of both peers with other available clients.  Also
                // provide current available client names to both peers
    for(iIndex3 = 0; iIndex3 < m_iNextFree; iIndex3++)
    {
      if(m_lpClientSocks[iIndex3].m_iStatus == VTSOCKET_AVAILABLE)
      {
        if(iIndex3 != iIndex1)
        {
          m_vtMsgBuf->m_ucIdentity = VT_IDENTITY;
          m_vtMsgBuf->m_ucMsgType = VT_TEXT;
          m_vtMsgBuf->m_ucCmd = VTCMD_REGNAME;
```

```
                                    m_vtMsgBuf->m_lLength =
REALLEN(m_lpClientSocks[iIndex1].m_sUserName)
                                                + SIZEVTMSGHDR;
            memcpy((void*)m_vtMsgBuf->m_pData,
(void*)m_lpClientSocks[iIndex1].m_sUserName,
REALLEN(m_lpClientSocks[iIndex1].m_sUserName)*sizeof(char));

        SendVTMessage(m_lpClientSocks[iIndex3].m_hSock, m_vtMsgBuf);

            if (iIndex3 != iIndex2)
            {
                                    m_vtMsgBuf->m_lLength =
REALLEN(m_lpClientSocks[iIndex3].m_sUserName)
                                                + SIZEVTMSGHDR;
              memcpy((void*)m_vtMsgBuf->m_pData,
(void*)m_lpClientSocks[iIndex3].m_sUserName,
REALLEN(m_lpClientSocks[iIndex3].m_sUserName)*sizeof(char));
               SendVTMessage(m_lpClientSocks[iIndex1].m_hSock, m_vtMsgBuf);
            }
          }
          if(iIndex3 != iIndex2)
          {
             m_vtMsgBuf->m_ucIdentity = VT_IDENTITY;
             m_vtMsgBuf->m_ucMsgType = VT_TEXT;
             m_vtMsgBuf->m_ucCmd = VTCMD_REGNAME;
                                    m_vtMsgBuf->m_lLength =
REALLEN(m_lpClientSocks[iIndex2].m_sUserName)
                                                + SIZEVTMSGHDR;
            memcpy((void*)m_vtMsgBuf->m_pData,
(void*)m_lpClientSocks[iIndex2].m_sUserName,
REALLEN(m_lpClientSocks[iIndex2].m_sUserName)*sizeof(char));
            SendVTMessage(m_lpClientSocks[iIndex3].m_hSock, m_vtMsgBuf);

            if(iIndex3 != iIndex1)
            {
                                    m_vtMsgBuf->m_lLength =
REALLEN(m_lpClientSocks[iIndex3].m_sUserName)
```

```
                                                    + SIZEVTMSGHDR;
                memcpy((void*)m_vtMsgBuf->m_pData,

(void*)m_lpClientSocks[iIndex3].m_sUserName,

REALLEN(m_lpClientSocks[iIndex3].m_sUserName)*sizeof(char));

                  SendVTMessage(m_lpClientSocks[iIndex2].m_hSock, m_vtMsgBuf);
                }
              }
            }
        }
        return 1L;
                default:
                        AfxMessageBox("NOTHING");
        }

        return 1L;
}
```