Design of a Voice over IP System that Circumvents NAT

Jem Berkes

Timothy Czyrnyj

Justin Olivier

Dominic Schaub

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Engineering, at the Faculty of Engineering, University of Manitoba.

March 2004

Faculty Supervisors:

Dr. Robert Mcleod, Advisor

Dr. Joe LoVetri, Course Coordinator

Abstract

Voice-over-IP (VoIP) provides convenient, low-cost telephone calls over the Internet and has the potential to supercede analogue telephone systems. Current VoIP systems cannot deliver low latency communications between two firewall or Network Address Translation (NAT) protected hosts, which has impeded its widespread adoption among residential Internet users. This report describes the design and implementation of a functional peer-to-peer VoIP system targeted for home PC users on residential Internet connections. A suitable VoIP protocol was developed through network testing and experimentation. This protocol was built into software using modules implementing connection establishment, audio compression, encryption, packet queuing and audio processing. A novel connection establishment scheme that provides direct communication between hosts irrespective of firewalls or NAT was employed to eliminate the need for a centralized architecture, attaining the lower latency inherent to peer-to-peer architectures. The decentralized system scales well since each conversation occupies its own connection independent of any other communications. An external hardware device, capable of interfacing with both the computer and the analogue telephone lines, provides a bridge between our VoIP software and conventional telephone lines. The graphical user interface designed for home PC users allows people to place telephone calls over the Internet and optionally dial real telephone numbers via the hardware interface. The resulting complete system, with the help of user directory facilities, succeeds in providing the full VoIP usability as originally intended.

Acknowledgements

Our group would like to thank our advisor, Dr. R. D. McLeod, for his advice and guidance and for his inspiring Telecommunications course. We would also like to thank NSERC and the Faculty of Engineering for funding the summer 2003 research that evolved into this project. We thank Karim Abel-Hadi and Jenny Chuang and Michael Trachtenberg for their assistance in testing our software. Finally, we thank Karim Abel-Hadi again for immense help in proofreading our report.

Contributions

The background research and the preliminary protocol design for this project was a group effort. Each person dedicated time to learning and understanding the background knowledge needed to undertake each task. Although each member was assigned one segment of the project, each member received insight and help from other members from time to time.

Jem Berkes

- Hardware Design
- Hardware Implementation
- UDP Packet Loss and Packet Error Tests (Wired)
- Network Packet Latency Tests (Wired)
- Dynamic Buffering
- Feedback Cancellation

Timothy Czyrnyj

- UDP No-Checksum Interface
- UDP Packet Loss and Packet Error Tests (Wireless)
- Network Packet Latency Tests (Wireless)
- Network Jitter Testing
- The Software Controller and Hardware Interface
- Windows Audio Interface

Justin Olivier

- Existing VoIP research
- Codec Research
- Noise Reduction (IIR Filtering)
- Silence Detections
- Automatic Gain Control
- Preliminary Windows Audio Testing

Dominic Schaub

- Protocol Design
- Network Jitter Solutions
- Encryption
- Graphical User Interface
- Integrated Directory Service
- Controller Module
- Feedback Cancellation

Table of Contents

List of Figures

List of Tables

Acronyms

ADPCM	Adaptive Differential Pulse Code Modulation
ADSL	Asymmetric Digital Subscriber Line
AGC	Automatic Gain Control
API	Application Programming Interface
BER	Bit Error Rate
CELP	Code Excited Linear Prediction
CODEC	Compressor/Decompressor
DES	Data Encryption Standard
DLL	Dynamic Link Library
DMZ	Demilitarized Zone
DS	Directory Service
DSP	Digital Signal Processor
DSS	Directory Service Server
DTMF	Dual-Tone Multi-Frequency
FIR	Finite Impulse Response
GSM	Global System for Mobile Communication
GUI	Graphical User Interface
ICMP	Internet Control Message Protocol
IDEA	International Data Encryption Standard
IP	Internet Protocol
ISP	Internet Service Provider
ITU	International Telecommunication Union
LED	Light Emitting Diode
LPC	Linear Predictive Coding
MD5	Message Digest 5
MTU	Maximum Transmission Unit
MVC	Moving Volume Control
NAT	Network Address Translation
OS	Operating System
PCB	Printed Circuit Board
POTS	Plain Old Telephone Service
PSTN	Public Switched Telephone Network
RSA	Rivest, Shamir, & Adleman
RTT	Round-Trip Time (latency)
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TPS	Third Party Server
UDP	User Datagram Protocol
uPnP	Universal Plug and Play
VoIP	Voice-over-IP
WAV	Windows Wave

1. Introduction

1.1 What is VoIP?

Voice-over-IP (VoIP) refers to technologies that allow telephone-like voice communications carried within Internet Protocol (IP) data packets, rather than dedicated voice signal lines. Audio is digitized, compressed, and then filled into multiple IP packets. These data packets travel through a packet-switched network such as the Internet and arrive at their destination where they are decompressed using a compatible Codec (audio coder/decoder) and converted back to analogue audio.

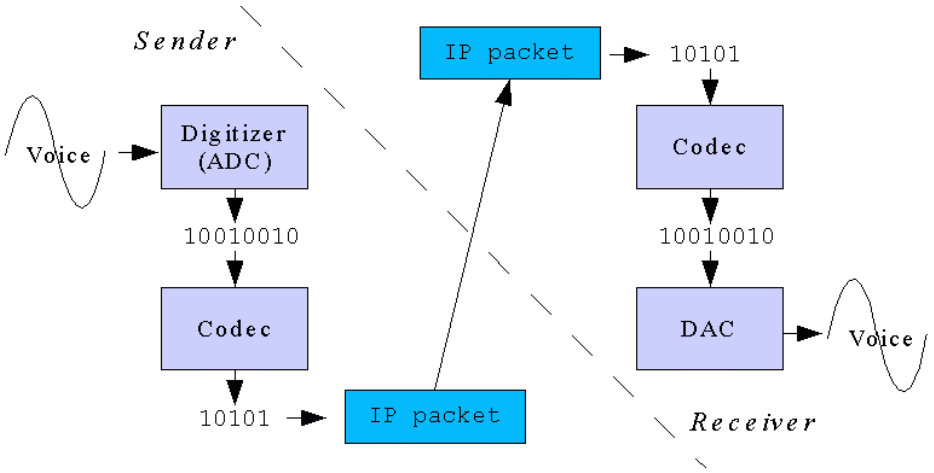


Figure 1-1: Overview of Voice over IP

Effectively, VoIP allows people to conduct normal telephone conversations without using telephone lines. While there are many different implementations of this technology, the fundamental motivations behind VoIP are the same:

- **Low cost.** An international or overseas VoIP call may cost a mere fraction of an equivalent telephone call using long-distance carrier. The costs are reduced to those required to sustain network connectivity at each endpoint, while the data travels freely over global IP networks alongside all the other Internet traffic.
- **Convenience.** While telephone or cellular services require subscription and often have physical limitations with respect to location, VoIP allows service anywhere there is Internet access.

- **Flexibility.** Despite the advances in telephone technology, the user is quite limited by the equipment and features offered. VoIP, on the other hand, allows a myriad of service types and features with the only requirement being adherence to a defined protocol or standard.

1.2 Purpose

For our group project, we designed and implemented a functional peer-to-peer VoIP system targeted for home PC users on residential Internet connections. Our project consists of three main tasks: protocol design, software design, and hardware design. The completed system should allow two people to engage in a voice conversation over the Internet.

1.2.1 Protocol design

We designed a protocol that allows audio to be sent and received over the Internet, with support for various features such as encryption. The protocol takes into consideration the types of errors we may encounter in packet communications and also ensures timely delivery of data through a variety of network conditions. In order to design a suitable protocol for our target environment, we conducted various tests to evaluate "typical network conditions".

1.2.2 Software design

Our software implements the protocol as designed, providing concrete facilities for packet transmission, coding, and audio handling. The overall software design consists of many smaller modules that handle everything from network communications to cryptographic algorithms. As part of our software design we created a Graphical User Interface (GUI) that is suitable for home PC users. The software is also capable of interfacing with our external hardware device in order to connect to the telephone line.

1.2.3 Hardware design

We designed and built a hardware device capable of connecting the home user's PC to the analogue telephone lines, or Plain Old Telephone Service (POTS). This provides a means for our software to dial external telephone numbers and therefore connect to the Public Switched Telephone Network (PSTN). The hardware bridges the gap between Internet-based VoIP and conventional telephones.

1.3 Goals

Throughout the design process we focused on the following goals:

1. **Good audio quality and low latency.** Audio should be clear and must be transported relatively quickly from end to end without long delays.
2. **Network condition adaptability**. Our system should adapt to a variety of network conditions (bandwidth, packet loss, and packet latency). This means that we should be able to use the resulting software on both high-speed reliable networks and lower speed networks with packet loss.
3. **Security (privacy).** Our system should provide some degree of data protection to guard against eavesdropping of communication by an unauthorized party. This implies data encryption.
4. **Firewall/NAT compatibility.** Our system should be easy to use even when the user(s) are behind firewalls or Network Address Translation (NAT). Such configurations typically pose a major challenge to decentralized software such as ours.
5. **Decentralization/peer-to-peer nature**. Our system should send the voice data directly from one peer to the other, and *not* require a central server that relays all data.

1.4 Unique aspects of our system

While many commercial VoIP solutions exist, they are mostly targeted towards large organizations that wish to reduce long-distance costs. There are currently few VoIP solutions designed for home (or small-scale) users because such systems are not practical for technical reasons. We believe that we have overcome these limitations and have designed a usable VoIP system for home users.

What makes our project unique is the decentralized peer-to-peer design. Our system sends voice data directly from one IP host to another, and therefore does not require a central server. A central server would automatically demand double the bandwidth and introduce twice the latency (audio delay). Our peer-to-peer solution can also scale very well, since each voice conversation over VoIP is independent of all other conversations. In contrast, a system that uses a central server loads the central host with more data for every conversation, up to a finite capacity at which the infrastructure is exhausted. Our lightweight solution has no such limitations.

In order to accomplish this peer-to-peer design, we required a way to bypass the ever-present problem of firewalls and NAT routers on home Internet connections. NAT is typically a fundamental barrier to any peer-to-peer system because it makes a private host unreachable from the public Internet. We used software developed in 2003 at the University of Manitoba in order to overcome this barrier [2].

1.5 Real-world Impact

We hope that the result of our project is the first VoIP solution oriented towards home Internet users that scales well and is free from configuration difficulties imposed by firewalls and NAT. This system would not require a high-bandwidth central server, and therefore would not require any central support services.

Our system allows people to make telephone-like calls over any distance, whether local or international, without involving a long-distance carrier. With the addition of the hardware component, our system even lets an Internet user place calls to regular telephone numbers. Someone using our VoIP software can make free international telephone calls and reach people on normal telephones without requiring any expensive infrastructure, as is normally required.

In creating this system, we further hope to demonstrate the feasibility and potential of peer-to-peer solutions. Approaches similar to ours can be applied successfully in many other problems.

2. Research and Network Testing

2.1 Existing Voice over IP Systems

Voice over IP systems have existed for quite some time, but have only recently become mainstream as audio quality has increased and can now rival that of POTS. This is equally due to newly developed compression algorithms and increased bandwidth availability.

Certain properties are vital to the quality and speed of any system. The properties are bandwidth (compression, decompression, etc.), routing, inherent Internet attributes (NAT, firewalls) and security. Our software system uses each of these things.

Previously implemented systems surmounted many key issues central to today's Internet (ie: NAT, bandwidth) in an ad-hoc manner. There are also methods that are common to all VoIP systems because of their proven reliability. Here are just a few of the systems out there.

2.1.1 Speak Freely

Speak Freely, created by John Walker in 1995–1996, was designed as a freely available chat application and an alternative to conventional long distance communication. It uses the Global System for Mobile Communication (GSM) and Code Excited Linear Prediction (CELP) Codecs for voice compression and Data Encryption Standard (DES), Blowfish and International Data Encryption Standard (IDEA) ciphers for encryption. It is written in C and available on many different platforms. Speak Freely can operate using different protocols, and communication with different applications is therefore possible. Its development was discontinued in August 1st, 2003 due to outdated technology and NAT/firewall issues.

2.1.2 TeamSpeak

TeamSpeak is a freely available VoIP system designed to provide interactive audio for online games, and minimizes latency by using Codecs such as CELP and GSM, which have high compression ratios. TeamSpeak uses servers in a centralized configuration to overcome the obstacles imposed by NAT.

2.1.3 Skype

Skype is a decentralized VoIP system developed by Kazaa that employs a pre-existing peer-to-peer network to route conversations. Skype circumvents NAT by using non-firewalled hosts as 3rd parties. It is free to users and provides good sound quality and encryption.

2.1.4 HawkVoiceDI

HawkVoiceDI, created by Hawk Software, is a voice network Application Programming Interface (API) that is used in online games. It was designed as an open source alternative to proprietary APIs. HawkVoiceDI is programmer-friendly and offers a choice of eight Codecs for compression. It uses Message Digest 5 (MD5) and Blowfish for stream encryption.

2.1.5 ITU Standards

The International Telecommunications Union (ITU) defines international protocols and standards that govern many widely used communication networks. Time constraints prevented this project from employing the H.323 standard for packet-based multimedia communications.

2.1.6 Strengths of Existing Systems

Current technology can produce high quality voice transfer rivaling that of analogue telephony. This is partly due to improved compression rates and the advancing technology associated with higher bandwidth, which improve audio quality by decreasing latency. For example, Linear Predictive Coding 10's (LPC-10) compression ratio of 26 to 1 is vastly superior to Mu-law's 2:1 ratio.

Information security has also evolved. Implementations such as Blowfish theoretically offer strong encryption, although awareness and implementation has been inadequate.

2.1.7 Weaknesses

Two NAT users are usually unable to connect to each other in a decentralized manner. The conventional technique for circumventing NAT uses a non-firewalled 3rd party (sometimes called a "super node") to connect the two firewalled users. Doing so, however, incurs additional latency and Internet traffic.

Although these constraints are unavoidable, a good solution will need to take into account this balance find a well-designed answer to VoIP. With VoIP, the specific shortcomings are with respect to NAT incompatibility, and bandwidth versus audio quality trade-offs.

2.2 Audio Codec Research

VoIP systems use compression to minimize bandwidth and latency. The compression algorithm (or Codec) used determines the compression ratio, and frame size.

2.2.1 μ -Law

μ-Law is perhaps the most established voice communications standard, and is used internationally to encode audio telecommunications. μ-Law's speed derives from its using a simple logarithmic look-up table that encodes only the 13 most significant bits of a 16-bit sample, yielding a 2:1 compression ratio. μ -Law's European counterpart is called A-Law.

2.2.2 Adaptive Differential Pulse Code Modulation (ADPCM)

Adaptive Differential Pulse Code Modulation is one of the most widely used Codecs. It has been heavily integrated into ITU's standards and is available on almost any platform that can handle voice communications. ADPCM uses a table look-up system and encodes only the differences between consecutive samples. Its compression ratio is 4 to 1.

2.2.3 Global System for Mobile Communication (GSM) 6.10

Global System for Mobile Communication was specifically developed for wireless digital voice telephony and can be found on most platforms. GSM employs LPC and therefore shares many of its characteristics such as a high compression ratio. GSM has a 10 to 1 compression ratio.

2.2.4 LPC

Linear Predictive Coding (LPC) is among the most effective compressions currently available for audio communications. However, it is computationally intensive and the compression quality degrades at high frequencies. LPC predicts the likelihood of a waveform's progression and measures differences between the predicted and actual waveform. Its

compression ratio is 12 to 1 and a modification by the US Military called LPC-10 has a compression ratio of 26 to 1.

2.2.5 Codec Comparison

Codecs are evaluated by comparing the optimal tradeoff between compression speed and ratio.

	M-Law	*ADPCM*	*GSM 6.10*	*LPC*
Data Rate	*64Kbps*	*32Kbps*	*13.2Kbps*	*10.7KBps*
Compression Factor (ratio)	*2:1*	*4:1*	*10:1*	*12:1*
Relative Compression Rating (speed)	*1*	*2*	*3*	*4*
Overall Rating	*3*	*1*	*2*	*4*

Table 2-1: Codec comparison of speed and performance. Ratings (1-best to 4-worst) based on 8kHz mono 16-bit samples

ADPCM was used for this VoIP system because it offers good compression while requiring little computational time. This algorithm will be discussed in further detail later in this report. GSM 6.10 is also attractive due to its high compression ratio. However, GSM and LPC are computationally expensive and would be better suited for dedicated audio devices.

2.3 Network packet latency

Packet latency is a measure of how long it takes a packet to get from one point to another over a network. In designing our protocol, we had to carefully consider packet latency because this defines a *lower limit* on the expected audio delays. Since one of our primary goals is to achieve low audio latency, we had to investigate actual network conditions in order to design our software to achieve the minimum latency possible.

While the relatively static path of routers between two IP hosts defines the minimum path latency, the latency for any particular packet can vary substantially due to congestion at routers or traffic prioritization. Our packet latency testing attempted to gain some insight into typical network conditions that our VoIP software would have to deal with. We wanted to:

- **Examine the relationship between packet size and latency.** Theory predicts that latency varies linearly with packet size due to the constant bitrate of transmit/receive operations. This immediately suggests that our protocol should use small packets since these are delivered more quickly than larger packets (at the detriment of bandwidth efficiency). However, since actual Internet router behaviour is much less predictable, we wanted to see whether the linear relationship held or whether there was in fact some "ideal packet sizes" that consistently resulted in lower latencies.

- **Examine how latency can change over time (long-term trends)**. Over a period of hours or even minutes, latency due to router congestion can change quite dramatically. Much of this has to do with peak-hour usage and other factors that are far beyond our control. Nevertheless, observing these changes can help us introduce adequate provisions into the protocol.

- **Examine latency ranges**. We wish to determine the overall ranges in latency for a link.

While it is common for network utilities such as 'ping' to use Internet Control Message Protocol (ICMP) packets for latency measurements, we needed a way to test UDP packet latency since our software uses UDP and not ICMP. This is particularly important because routers can treat UDP packets very differently from other protocols. For this purpose we wrote several versions of a C program called **udpstat.c** that can gather statistics on UDP packets. The software does automated tests according to programmed parameters, and writes raw output files which we can then graph or otherwise analyze.

2.3.1 Internet-wide latency tests

The **udpstat.c** tool was written to test UDP packet latency. This software generates UDP packets and sends them to a destination host that is configured as a *UDP mirror*. The mirror host "bounces back" the UDP packets to the sender, allowing the sender to analyze the resulting round-trip time (RTT). The contents of the packet are not modified in the process. This configuration allowed us to conduct numerous automated tests, even overnight (unattended).

A standard Linux 2.4 host was configured as a UDP mirror, using standard features provided by the 'netfilter' packet filter and 'iptables' firewall module in Linux. We configured a spare Linux host at the University of Manitoba to act as the UDP mirror for the duration of our tests. The host on which we ran the udpstat software was connected to the Internet via a residential Asymmetric Digital Subscriber Line (ADSL) connection also in Winnipeg.

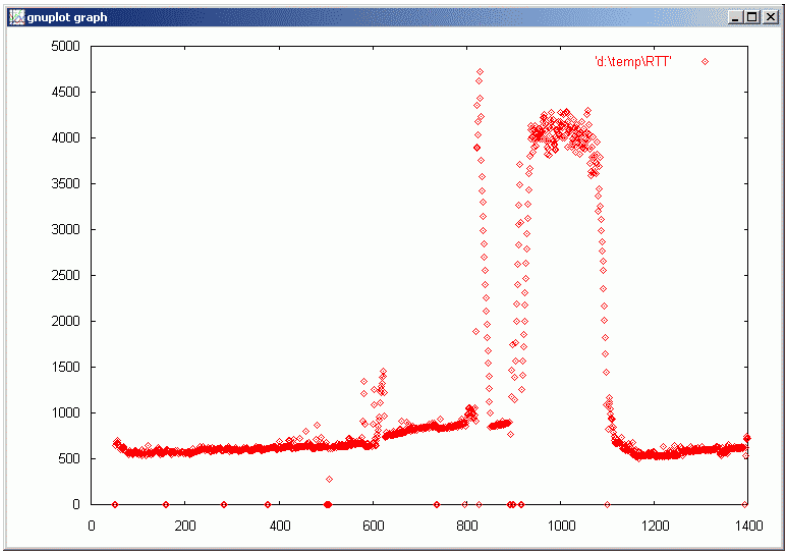
A *traceroute* between the two hosts involved in the test showed 13 router hops, indicating a rather long distance between the ADSL and University of Manitoba host which are both physically located in Winnipeg. (In fact, the packets travel from Winnipeg to Toronto, and then to Calgary before returning back to Winnipeg). This provides a suitable test environment because a large number of router hops will lead to greater variations in packet latency, since each individual router has an unpredictable behaviour.

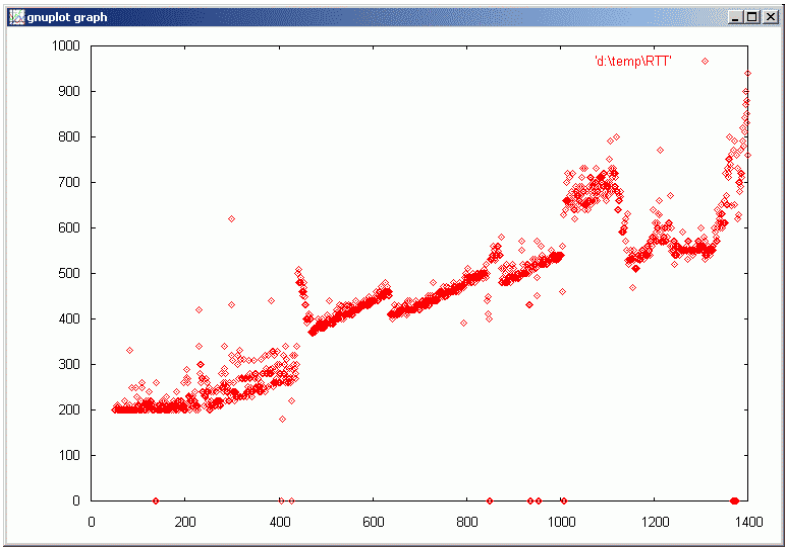
We wrote a number of different versions of udpstat.c and conducted several tests. We discuss these tests and the results in the following sections.

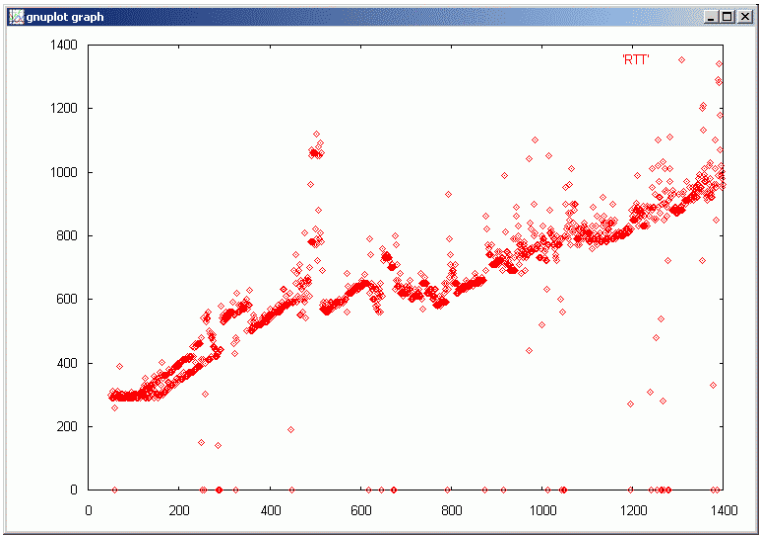
2.3.1.1 Tests with udpstat v1.2

This version of the software generates UDP packets with sizes that linearly sweep over a range (typically 50 to 1400 bytes, in 1 byte increments in either the up or down direction). Groups of 10 or 20 identical packets are constructed and sent to the UDP mirror, which sends back the packets to the originating host. The software measures the delay between sending the packets and receiving the returned versions (total RTT for the batch). Some packets may get lost and never return, invoking a timeout and resulting in higher total RTT for the batch.

Four trials were performed with this version of the software. The following graphs were constructed from udpstat's output, showing batch RTTs (in milliseconds) on the y-axis and batch packet sizes on the x-axis. These tests were conducted at different times of day over multiple days.


gnuplot graph
'd:\temp\RTT'
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
200
400
600
800
1000
1200
1400


gnuplot graph
'd:\temp\RTT'
1000
900
800
700
600
500
400
300
200
100
0
0
200
400
600
800
1000
1200
1400


gnuplot graph
'RTT'
1400
1200
1000
800
600
400
200
0
0
200
400
600
800
1000
1200
1400

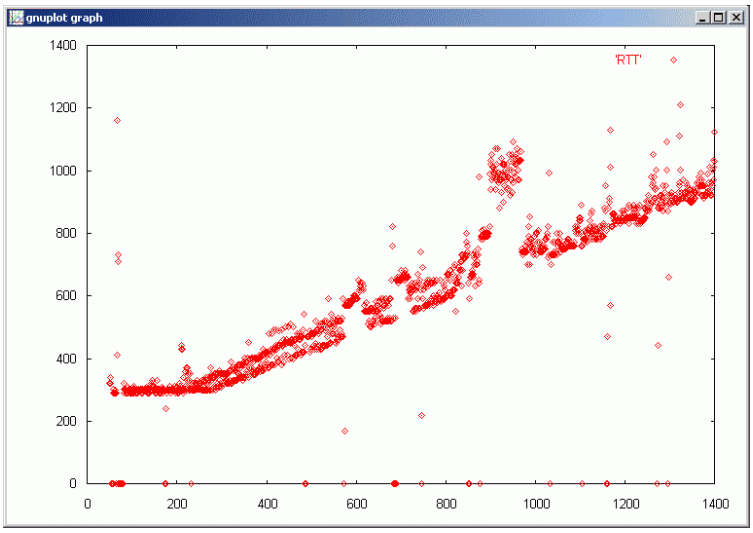


Figure 2-1: Total RTT vs. Packet Size in sequence

These results cannot easily illustrate the relationship between packet size and latency because the packet sizes increase or decrease in order, and line burst-errors cause dominant artifacts (the next series of tests correct for this by randomizing the order of packet sizes sent).

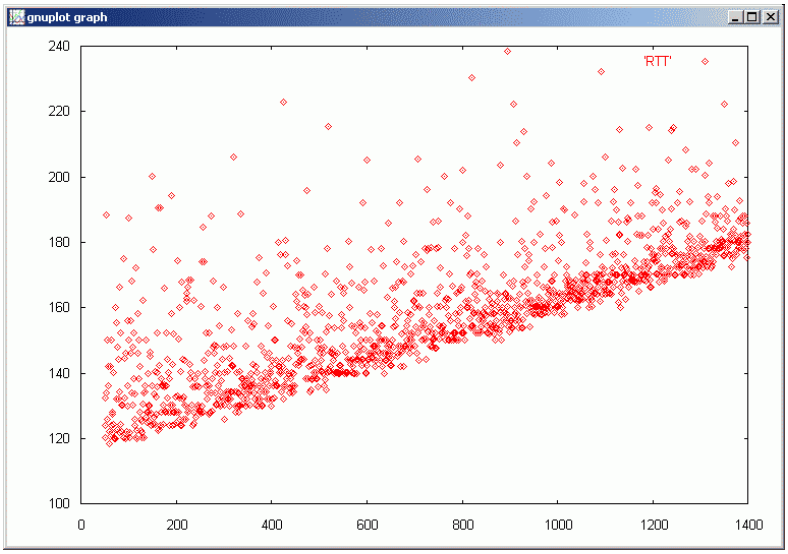
However, the graphical results indicate some of the characteristics of the path between the ADSL host and the University of Manitoba. Generally, larger packets have greater total RTTs, as expected. However, there are periods in which the total RTT increases substantially. For instance, the first graph shows that the RTT is relatively uniform until the packet sizes reach about 900 bytes. From this time until when the packet sizes have reached about 1100 bytes, the total RTT increases by a factor of four. The RTT fluctuations are fundamentally a function of time, and not necessarily packet size.

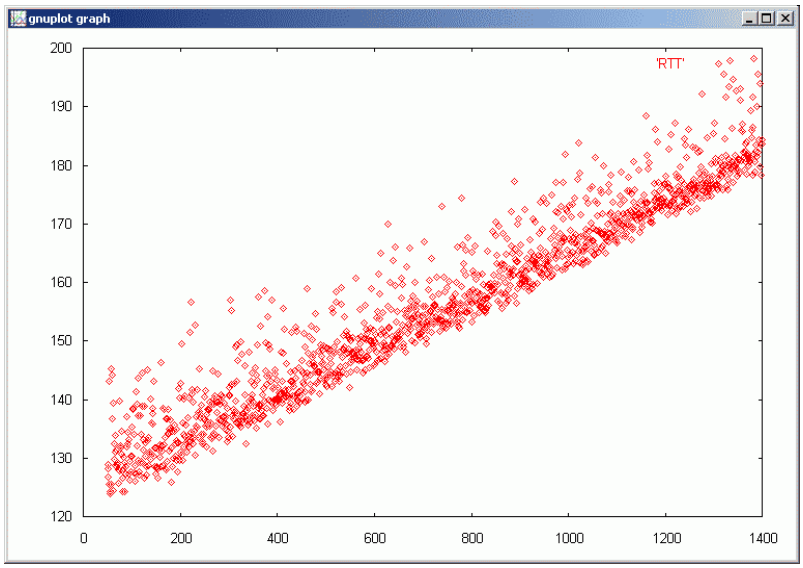
The first graph exhibits the most extreme variations in latency. The single packet RTT varied from approximately (500 / 10) = 50ms up to as much as 400ms during the period of (assumed) router congestion.

2.3.1.2 Tests with udpstat v1.3 through v1.5

While the earlier software incremented packet sizes after each batch, this software version randomizes the order of transmitted packet sizes, eliminating the effect of the router's current congestion state upon the overall RTT for a specific packet size class. There are still multiple trials of each packet size sent out, but these trials are dispersed over a wide range in

time. The RTT shown on the graph now refers to the average RTT for a certain size of packet, with time-dependency removed.





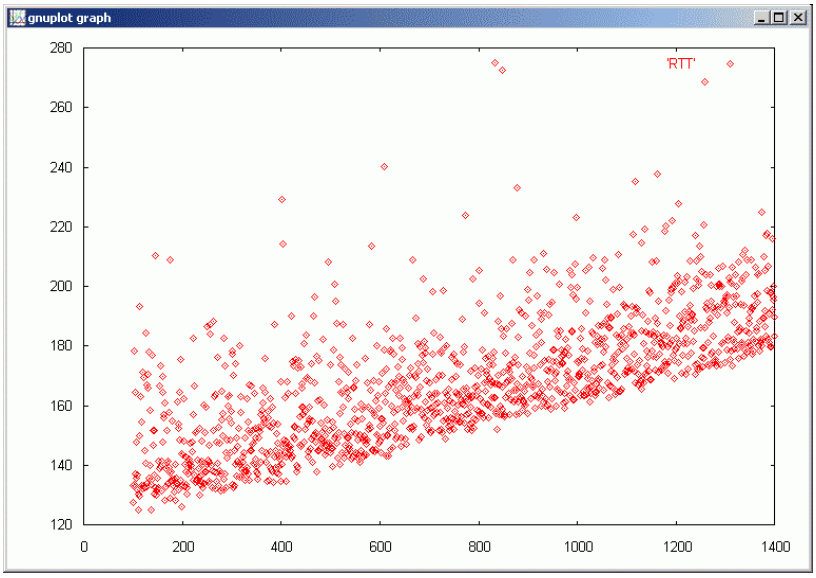


Figure 2-2: Average RTT vs. Packet Size in random sequence

Because of the large number of trials, these tests spanned several hours. For instance, the test with 100 trials of each packet size took nearly 8 hours to complete. These results now clearly demonstrate the relationship between packet size and round trip time. RTT does indeed vary linearly with packet size, with a minimum RTT strictly defined by the packet size (as predicted by theory). However, the cloud of points above the baseline does indicate substantial variation in RTT.

These Internet-wide tests helped answer our original questions with respect to packet latency. Primarily, latency does indeed vary linearly with packet size. Also, periods of congestion or other router trouble can change latency drastically for extended periods (even quadrupling the latency!). Finally, for this particular pair of Internet hosts we can say that the average RTT varies from a minimum of 120ms for the smallest packets up to 180ms for the largest packets, but actual RTTs can be up to 50ms greater than this minimum value.

2.3.2 Wireless Tests

In addition to the wired tests shown above, tests were run to see if a wireless connection would yield significantly different results. These tests were run using the same udpstat test application mentioned above and a similarly configured Linux NAT firewall. The Linux firewall was on the same network as the wireless access point, as depicted in Fig. 2-3. An 802.11b access point and wireless network card were used in these tests.

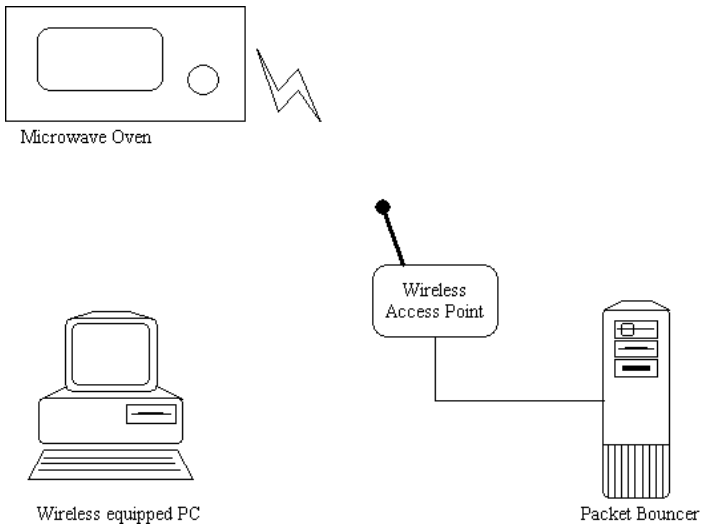


Figure 2-3: The configuration of machines for the wireless tests

The wireless tests were less rigorous than the wired ones. Their purpose was merely to see any substantial negative effects contributed by the wireless network. The figures below show the total round trip times of the wireless test. We employed the timer functionality of the Operating System (OS), which has a granularity of around 10ms; as such any round trip times within about 20ms cannot expressly be relied upon.

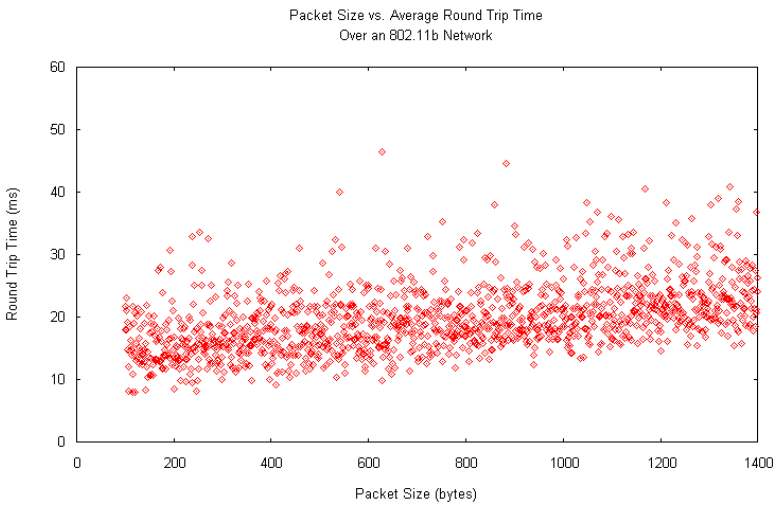


Figure 2-4: Round trip times over a 802.11b wireless network

In addition to this, it has been documented that microwave ovens can cause interference to 802.11 networks [7]. We took a cursory look to see how this may affect RTTs. The graph below shows the results of a test with a microwave running near by.

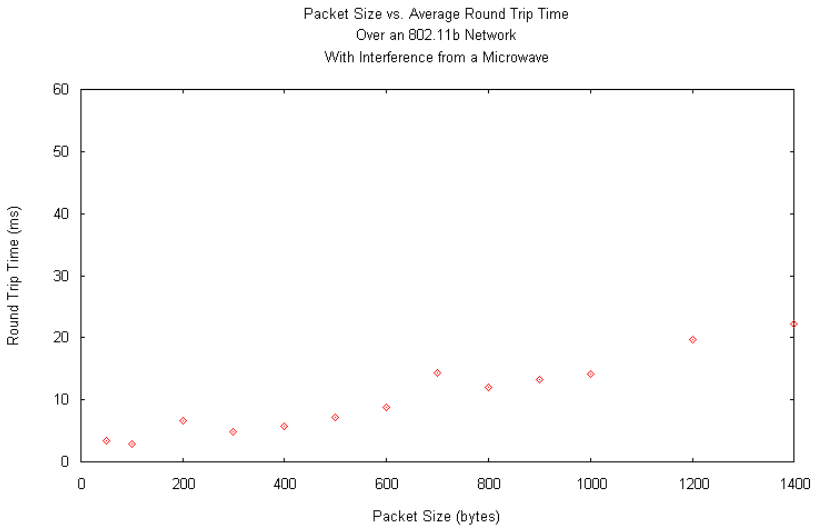


Figure 2-5: Round trips times over a 802.11b wireless network in microwave interference.

2.4 Packet Loss and Bit Error Rate

Various forms of packet errors are also typically seen on both small and large networks, and especially on the Internet over long distances. Packet loss measures the fraction of packets lost (discarded due to congestion or errors encountered along IP router hops) while in transit between the sender and receiver. Bit Error Rate (BER) refers to the rate at which individual bits within packets arrive in error (flipped) at the destination.

While these errors seriously affect data integrity, the problem is compounded by two factors specific to UDP packets. Primarily, UDP is an unreliable protocol that does not guarantee message delivery. This means that in our software, we will have no direct way of knowing that a packet did not arrive at its destination. The second factor is that many routers today prioritize Transmission Control Protocol (TCP) traffic over UDP, and may even explicitly throttle back UDP packet delivery rates depending on site policy. (Note: At the time of these experiments, Academic Computing & Networking at the University of Manitoba informed us that their routers do in fact use "traffic shaping" to limit the total UDP traffic entering or leaving campus.)

The BER also affects speed considerations in protocol design. UDP supports a checksum field (similar to TCP's checksum) as defined per RFC 768 or STD 6 [28]. Network stacks in OSs drop UDP packets with invalid checksums, thereby safeguarding the data payload. However, for many voice communications we can afford to have a limited amount of data in error, implying a

higher data transfer rate may be attainable by leaving the UDP checksum field blank, which has the effect of disabling the checksum feature as per the protocol specification. Once disabled, we may receive corrupted packets that would have otherwise been discarded by the OS.

We decided upon several goals in the packet loss and BER experiments that followed. We wanted to learn the following in order to direct our VoIP protocol design:

- **What typical UDP packet loss rates can we expect on a network?** If high packet loss rates are common, then we may have to design our protocol to recover from data loss.
- **Does packet loss depend on packet size?** If it does, we should send packets of a size that minimizes losses.
- **By disabling UDP checksums, can we receive packets with errors in them?** If we can, then we can recover data that might otherwise have been discarded.

2.4.1 Internet-wide Tests

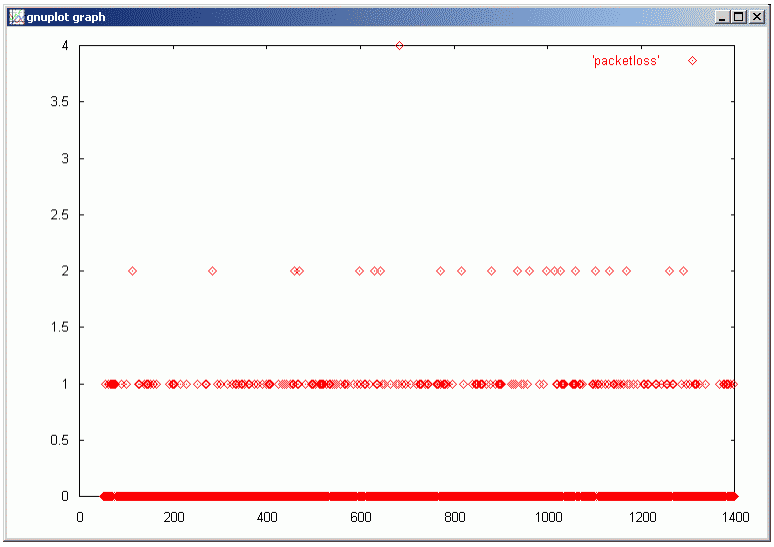
The **udpstat.c** software carried out UDP tests over the Internet. The testing procedure was discussed in detail earlier in §2.3.1, but this time the packet error information obtained from the output logs was used to compile information on packet errors. Note that the **udp_nocs.c** software used by udpstat.c can generate UDP packets with disabled checksums. This allows us to test for packets arriving with bit errors, which would otherwise have been dropped by the OS.

The following data was compiled from earlier tests done between a residential ADSL host and the University of Manitoba. Early versions of the software measured byte errors as opposed to bit errors. The only errors observed occurred during one trial using the earlier software. Our software did not test for packets arriving out of order. The results from multiple trials are summarized in Table 2-2.

Trial/log file	Packet Loss (%)	Bytes in error	Byte error rate
Trial 3/Log.3	711 lost / 13510 sent = 5.3%	0	0
Trial 4/Log.4	270 lost / 13510 sent = 2.0%	0	0
Trial 5/Log.5	1252 lost / 27020 sent = 4.6%	0	0
Trial 6/Log.6	1105 lost / 27020 sent = 4.1%	21 bytes	Based on mean packet size of 725 bytes, error rate for trial = 10^{-6}
Trial 7/Log.7	252 lost / 6755 sent = 3.7%	0	0
Trial 8/Log.8	3228 lost / 135100 sent = 2.4%	0	0
Trial 9/Log.9	836 lost / 65050 sent = 1.3%	0	0
TOTAL	7654 lost / 287965 sent = 2.7%	0	21 errors / 208 * 10^{6} bytes sent = 10^{-7}

Table 2-2: Packet errors in Internet-wide tests

From udpstat v1.3 onwards, packet sizes were randomized, removing time dependence so we can now see overall results grouped by packet size buckets. The following graphs show the number of packets lost as a function of packet size.



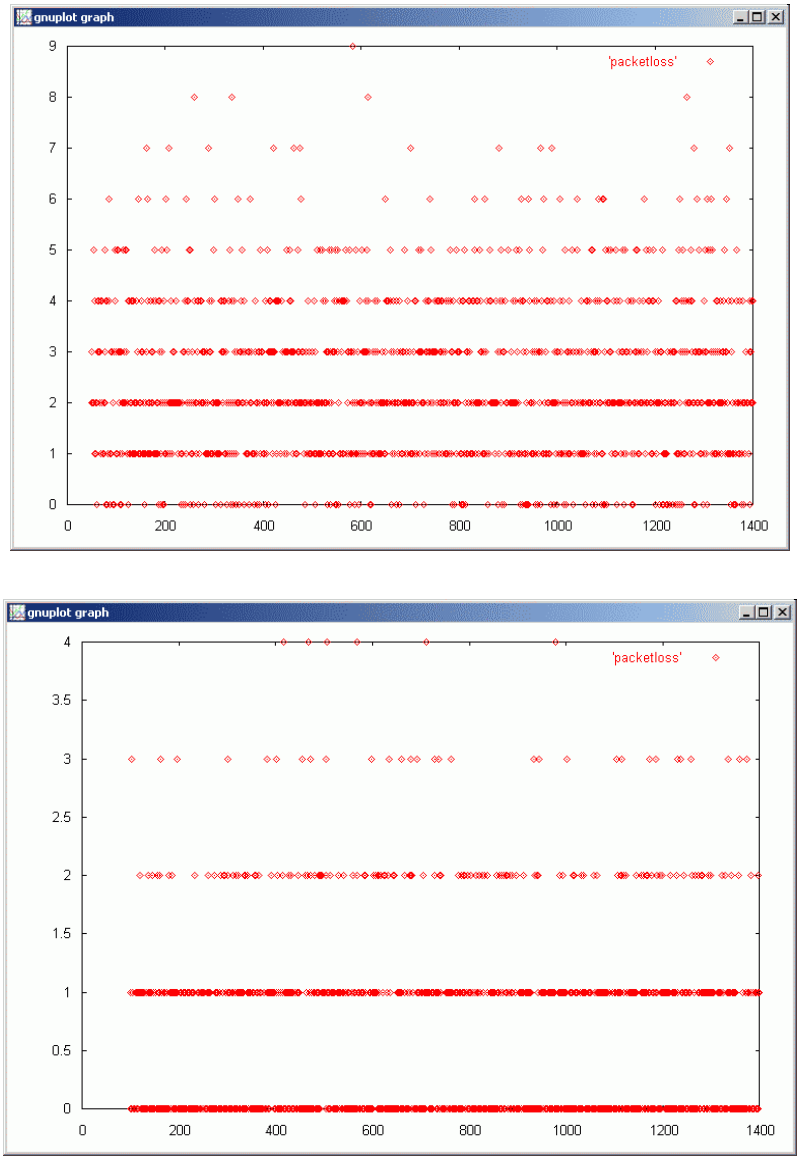


Figure 2-6: Packet Loss vs. Packet Size

These experimental results helped us arrive at a number of conclusions with respect to packet errors experienced over a typical Internet connection between two hosts. First, the packet loss of approximately 3% (and as much as 5%) means that our protocol should be able to deal with considerable losses. Second, packet loss does not appear to depend on packet size since there is a relatively uniform packet loss irrespective of the size of the packet being sent.

Finally, the results of these Internet experiments showed us that individual bit or byte errors inside packets, even when UDP checksums are disabled, are extremely rare. In fact, the only errors observed were for a single cluster of packets during one trial, and no other bit errors

were seen in the rest of the remaining several hundred megabytes of data. This is significant for our protocol design because it implies that there is no advantage to sending UDP packets with checksums disabled, since damaged packets are extremely rare. Packet loss is therefore the predominant form of packet error.

2.4.2 Packet Loss, Bit Errors and Wireless Networks

To further examine common residential Internet configurations, tests were run over the wireless IEEE 802.11b wireless network shown in Fig. 2-4 in §2.3.2. These tests used the udpstat software to test network behaviour. The results in Fig. 2-7 show that there were no packets received with erroneous data. These tests were run again with the microwave on.

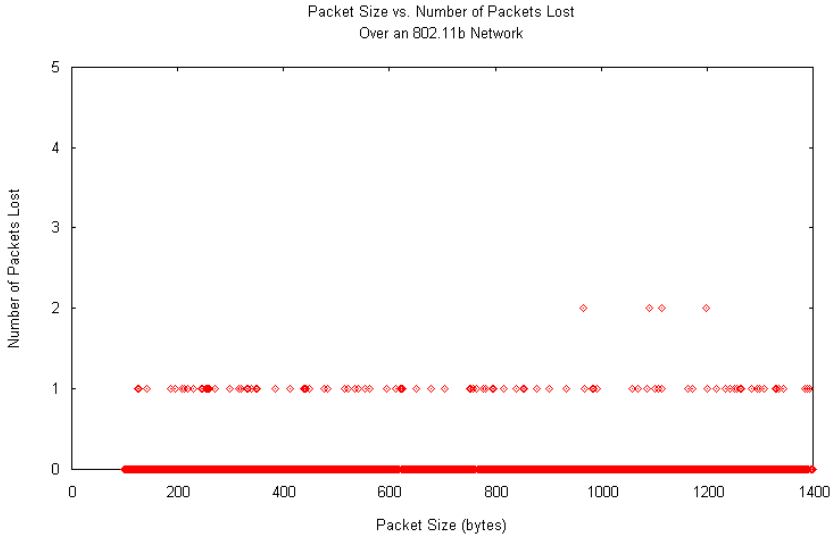


Figure 2-7: Packet loss on a wireless network

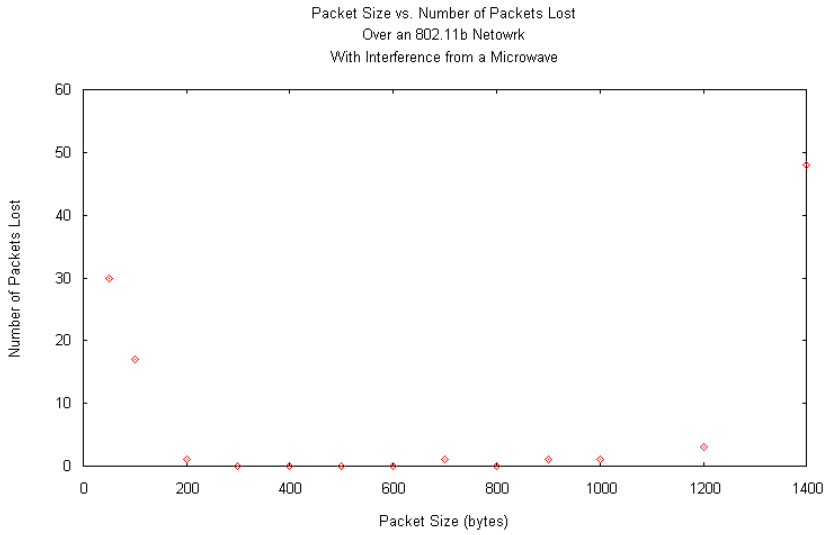


Figure 2-8: Packet loss on a wireless network with a microwave running near by

2.4.3 Checksum-less UDP Packets

To test for bit errors, the checksum on the UDP packets had to be disabled. Microsoft Windows does not provide this ability on a per application level. To avoid disabling UDP checksums for every program, special software was developed. This software used Windows raw sockets to construct packets with arbitrary IP and UDP headers.

According to the UDP specifications, setting the checksum field to 0 would cause the receiving end of a UDP packet to ignore this integrity check. The checksum-less UDP software has an interface similar to Windows Sockets. There is a send function with a similar syntax, and it accepts a checksum-less socket and data to be sent. This data is pre-pended with the IP and UDP headers that are constructed by the software. There is an initial "connect"-like function that acquires the data for the headers, and a close function that shuts down the socket.

A packet sniffer was used to examine UDP packets during our tests to ensure that their checksums were indeed set to 0.

2.5 Network Jitter

Consecutively transmitted packets rarely traverse the Internet in equal times. The variance in time required to traverse the Internet is known as *jitter*, a phenomenon that becomes especially pronounced in congested networks [4]. Jitter has a significant bearing on how much buffering is necessary to receive packets at a constant rate.

We wrote another test application to observe the jitter of packets sent out at a constant rate. This application consists of a sending program and a receiving program. The sending program runs on Windows and uses the audio function callbacks to send data out, thereby ensuring that it sends at the same rate as our application. The receiving program runs on another host and measures the time between the packets arrivals. The next figure shows the results of one such test.

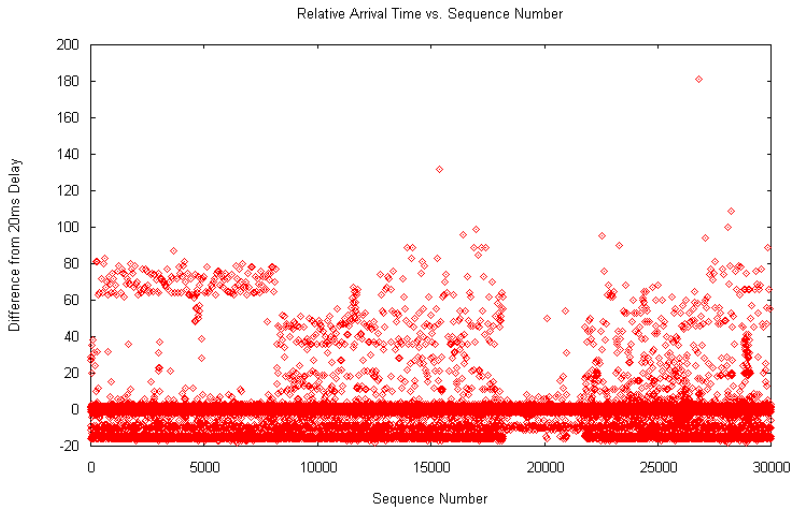


Figure 2-9: Time between packets

Examining the peaks above shows that a long delay for one packet is followed by many short delays for the next few packets.

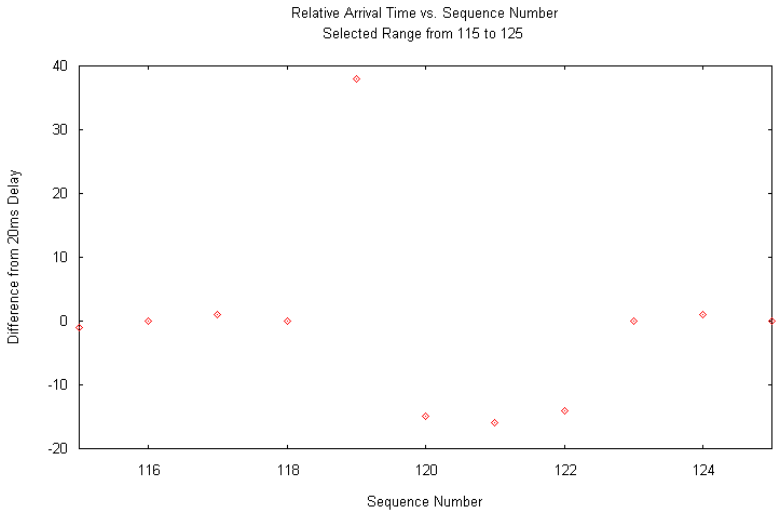


Figure 2-10: Substantial delay and subsequent short inter-packet times

3. Overview of Protocol Design

Our protocol forms one of the cornerstones of our project. In comparison to TCP, it is comprised of very little, and its minimalist nature has been vital to this project's success. The designed protocol resides between the network and application level and might be thought as being a transport layer. While technically this distinction belongs to the UDP layer, for our purposes UDP is only used in order to avoid the inherent difficulty of integrating a transport layer with the OS. Thus in our project, the UDP layer is only a thin wrapper over IP. The UDP layer and our protocol, which resides directly above it, together comprise the transport layer.

3.1.1 Issues affecting Protocol Design

We embarked upon protocol design after considering the unique demands of voice. These issues are (in descending order of importance):

- Lowest possible latency
- Uninterrupted audio stream
- Bandwidth usage
- Ease of connection establishment

High latency is a significant problem plaguing many VoIP systems and is an impediment to VoIP's widespread adoption. High latency results in users "talking over", or interrupting each other, and significantly deteriorates the quality of dialogue, even if the audio streams are uninterrupted and of good quality. The primary factors affecting latency are:

- Data route
- Error correction scheme
- Buffer sizes (both sender and receiver, network and audio)
- Packet size

Our protocol design addresses all but the first of these, which is tackled by the connection establishment scheme and NAT circumvention scheme.

Audio stream quality is also important, but this issue recedes in importance if interruptions are occasional and not particularly bothersome. As will be further explained,

compromises in audio quality can be exchanged for reductions in latency through laxer error correction.

The ever-widespread presence of high speed Internet led us to dismiss bandwidth usage as a significant concern. Finally, we opted to omit connection establishment as a consideration when designing our protocol; instead, the Directory Service (DS) will be employed to this end. The reader is referred to §4.6 and Appendix B for information concerning connection establishment.

3.1.2 Incorporation of Research into Protocol Design

Internet tests played a pivotal role in our final protocol design. At the outset we had planned on using Hamming codes to correct errors and regenerate lost packets. Subsequent testing, however, indicated that i) bit errors occur *very* infrequently, and ii) packets are almost always lost in burst errors (where sequential packets are lost), and that single packets are rarely lost. The reader is referred to §2.3 and §2.4 for detailed test results.

3.1.3 Latency Minimization

3.1.3.1 Error Correction and Data Recovery scheme

Hamming codes, either for single packets or multiple packets (where additional parity packets would be sent with regular packets), are ill suited to correct burst errors and are therefore not incorporated as an error correction scheme. In fact, no error correction was included at the transport level because correcting burst errors (the only error we consider) requires adding buffering and a mechanism to retransmit lost packets.

Retransmission of lost packets introduces latency directly related to RTT (an error message must be sent to the sender, which in turn must send a response), which over connections that are characterized by high RTTs, such as satellite or transoceanic links, becomes unacceptable. High transit times over these mediums cannot be shortened; they are innately restricted by the speed of light. What directly follows is that any improvements in latency *must* be made at the software level.

By treating dropped packets as lost data, latency can be minimized at the expense of audio quality, that is, by allowing the conversation to occasionally cut out when packets are dropped. As our tests indicate that burst errors are infrequent, we predicted that audio

degradation caused by burst errors is minimally perceptible. This was later confirmed in voice tests.

Allowing data to be lost in transit imposes certain constraints on information that is transmitted. Data in each packet must be independent of data in previous packets, given that earlier data may have been lost. This issue resurfaces in §4.2 and §4.3, where this constraint sets the criteria for selecting the Codecs and encryption scheme.

3.1.3.2 Buffer Sizes

The protocol does not call for specific buffer sizes; when data is available from the sound card, our program immediately sends it out in the subsequently described fashion. However, sound is buffered at the receiving end to correct for jitter. This is elaborated upon in §3.3. In general, buffer sizes were kept to a minimum.

3.1.3.3 Packet Size

Packet size is considered for completeness. By and large, most Internet applications transmit packets sized according to the network's Maximum Transmission Unit (MTU). This usually yields the greatest throughput and efficiency by maximizing the ratio of data to packet headers. However, concerns of efficiency and throughput are superseded by our desire to minimize latency. Any data we send over a network is automatically buffered by at least the size of the packet, so packets should be kept as small as possible.

The Codecs used also affect packet size. The standard frame size (used by most Codecs, such as GSM) is 20ms of audio data. Being irreducible, we refer to this frame length as an *atom* of sound data. Sending less than an entire atom at once realizes no benefits, as an atom must be reassembled before it can be decoded. Thus our packets transmit one atom at a time, with the size of the packet dictated by the Codec used.

3.2 Error Correction Choices

Packet loss and bit errors are inevitable in a VoIP system using UDP. After careful consideration, we found that error correction is unimportant and can be excluded altogether. The reasons for this are:

1. **Error Occurrence Vs. Payload Size:** Our UDP tests indicate that packet loss usually occurs in small bursts and rarely as single packet loss. Bit errors were even more rare. For data payload sizes of 20ms, our defined minimum, losing between one and ten consecutive packets (200ms) does not significantly erode the conversation's intelligibility. We therefore avoid correcting errors because there exists no potential for substantial gains in audio quality.

2. **Encryption Considerations:** The encryption our system uses is a block cipher. Were we to encounter single bit errors, only the affected packet would be garbled. Since we use a block, not stream cipher, any packet loss will not affect the decoding or security of any subsequent packets.

3. **Totally Unreliable Networks:** Adding error correction to compensate for unreliable networks is not suitable for this project. If a user's network is already error prone, the software cannot correct for this.

Though error correction is currently unfit for our project, we researched possible error correction schemes that might be suited for future VoIP endeavors, notably Hamming Codes and Convolutional Codes, both of which are currently used in communication algorithms.

3.3 Jitter Correction with Buffering

Large jitter is highly undesirable, as it demands commensurately greater buffering to maintain a stream of uninterrupted data, which in turn adds proportionally more delay that degrades conversation. Unfortunately jitter is unavoidable.

Jitter varies by network and time of day, making a simple catchall solution unattractive. Such a solution would either require a sufficiently large receiver buffer to smooth out the greatest expected jitter, or alternatively discard incoming packets arriving late. The former adds latency in transmitted speech, while the latter has the potential to seriously degrade speech quality. Dropping late packets is unacceptable because the lost data is irrecoverable (Hamming codes

would not work, because missing packets would need to be implicitly included in earlier packets, requiring sender-side buffering that again adds latency).

In order to minimize latency for a given network, we instead opted for receiver-side queuing that adapts to the degree of network jitter. This means that the queue is shorter under favorable network conditions and becomes commensurately longer in the presence of high jitter. Indeed, the protocol often reduced the network buffer to zero-length in subsequent testing, resulting in exceptionally natural conversation.

Our protocol realizes this functionality by keeping a target queue size. The queue size is permitted to deviate within a certain range about this target without the protocol intervening. Silence is removed or added when the queue becomes too large or too small. This operation is only carried out during periods of silence, ensuring that noticeable artefacts are not imparted onto the conversation. If the queue length remains above a certain threshold (defined in relation to the target queue length) for a certain length of time, the protocol will decrement the target length. Conversely, the target length is incremented when the program experiences a hard under-run and the queue length becomes zero. The protocol begins a session with a zero-length queue, which invokes a series of queue under-runs that cease when the queue reaches a steady-state length that is tailored to the network. Through this mechanism, optimized jitter buffering is established from the outset of a conversation.

4. Software Development

4.1 Introduction to modular design

The software is the central component of our VoIP system. It allows users with the required hardware (microphone, speakers, and soundcard), software (Windows 2000 and later) and network connectivity to engage in conversation over the Internet.

Like every large software program, the VoIP application was developed in modules. This is a standard method of developing a large system that allows individual sections to be developed, debugged and tested before they are aggregated into the final application. This approach is absolutely crucial in successfully developing a fully functional system.

Writing software in modules also afforded the luxury of using different programming languages for each task. This group used C/C++ when performance was critical and Visual Basic where rapid graphical development was desirable.

Decomposition of the larger program naturally fell along the lines of functionality. The six groups of functional modules are: audio, network, encryption, NAT circumvention, hardware and the GUI. The reader is directed to Appendix B for a detailed reference, but it is important to stress that there is not a one to one mapping of these modules to libraries.

In fact, the six functional modules are located in four resulting libraries, which are named: **tpc.dll**, **dsc.dll**, **pots.dll**, and **controller.dll**. The first two are NAT circumvention libraries, and the third is the hardware interface library, which receives further attention in §5.5. **Controller.dll** ties together the audio, network and encryption modules, and provides an abstracted "front-end" for the GUI. The Dynamic Link Libraries (DLL) files were written in C/C++, while the final GUI executable was written in Visual Basic. Figure 4.1 relates the modules and libraries to each other.

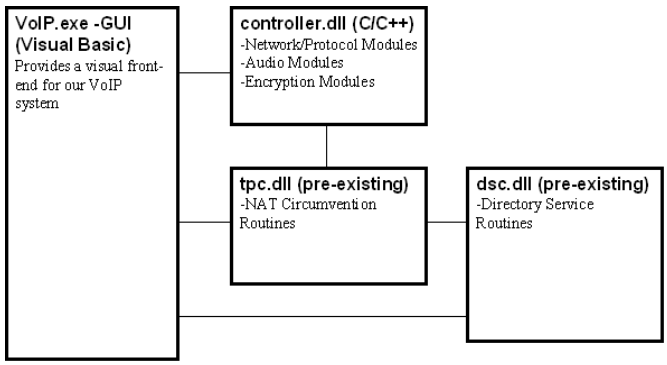


Figure 4-1: Software block diagram

4.2 Audio Modules

4.2.1 Windows Audio Interface

Our VoIP application interfaces with the computer's audio device. As the target system for this application runs a Microsoft Windows OS, our program interfaces with the Windows Multimedia functions. The primary advantage in wrapping the interface to the multimedia functions is the modularity that this affords. A modular interface allows us to change the functioning of the audio interface without needing to alter the rest of the application.

4.2.1.1 Audio Interface Functional Overview

The audio interface is essentially a simple queue. It takes in 20ms sound atoms and writes them to the output buffer. Before the sound data is written to the output device, the buffer must be filled to a desired level, at which point the audio interface begins sending sound atoms to the audio device.

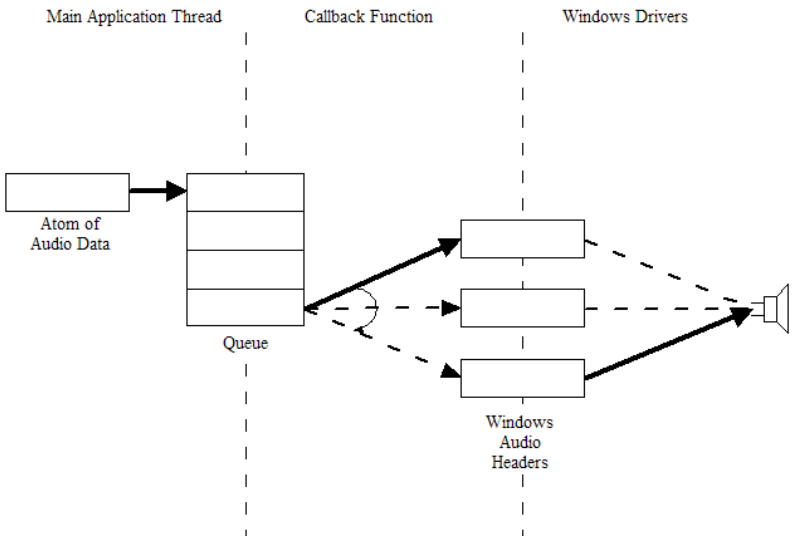


Figure 4-2: Data flow through the Audio Interface

The figure above shows how data flows through the audio interface. Audio data first enters our application over the network. It is then unpackaged and passed to the audio interface, which places the sound atom into a queue. The sound atom is then removed from the queue and sent to the sound device through the Windows audio functions.

4.2.1.2 The Function of the Queue

The additional queue between the application and the Windows audio device results in an easily manageable audio buffer size.

The queue's size is set by a function call. Knowing the size of the queue alerts us to potential synchronization problems, such as where clock skew between hosts causes the queue to grow or shrink. The queue is resized by adding or removing atoms of silence.

Conference calling imposes additional constraints to the VoIP system. Employing a single queue requires that incoming sound atoms be combined before they are written to the queue. Jitter renders such an approach impractical. Supplying each connection its own queue solves this problem by permitting each individual queue to fluctuate according to conditions specific to that connection.

4.2.1.3 Facilities for Dealing with Buffer Under Runs

The audio interface provides facilities for dealing with buffer under runs. A buffer under run causes the audio interface to call a user-defined callback function. This function's return

code determines whether the audio interface will stop or continue. The callback function can continue playback by either adding an audio atom, or arbitrarily increasing the queue size. The software might opt to insert silence if data is unavailable or it could alternatively suspend audio playback. The latter choice requires that the audio interface be restarted once the buffer is refilled.

4.2.1.4 Delays Introduced by Buffering

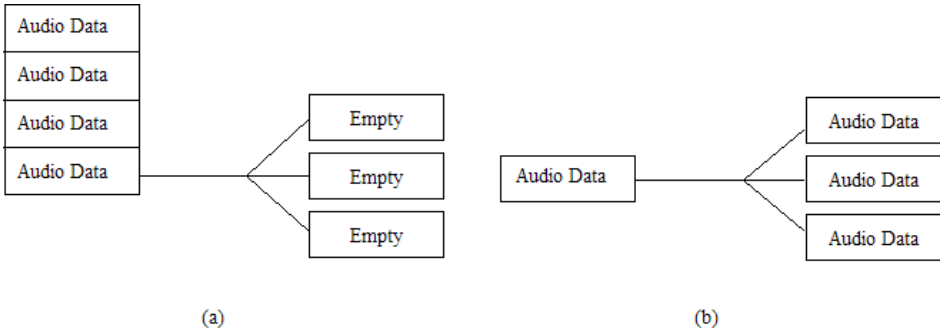


Figure 4-3: Position of audio atoms when starting the Audio Interface

Figure 4-3 shows how playback on the audio interface begins. The buffer is first filled to the desired level. When the process is started, audio atoms are removed from the buffer and passed to the Windows audio functions. This reduces the buffer size by the number of Windows audio headers being used by the Audio Interface (Fig 4.3b). New atoms that are added to the buffer must wait for the currently playing audio atoms and those remaining in the queue to finish playing. As the queue for filling the Windows audio headers has a minimum size of 3, this introduces a minimum delay of

$$(n-1) * 20 \text{ ms}$$

$$(3-1) * 20 \text{ ms} = 40 \text{ ms}$$

(Where n is the number of Windows Audio Headers) In the (n-1) term, one is subtracted from n as the first buffer begins playing immediately when the queue is emptied into the Windows Audio functions. Unfortunately, the use of the queue separate from the Windows Audio Headers prevents the 40 ms of buffering from combating jitter, as they must always have data in them even if they won't begin playing for another 40 ms.

4.2.2 Codecs

Our system employs the ADPCM Codec due to its compression ratio of 4:1 and its superior speech quality. The algorithm employs a table for step size, an index table and a predicted sample. The predicted sample variable is usually initialised to zero for both compression and decompression purposes. The step size varies according to the rate at which the signal fluctuates and so that the difference of the signal is more accurately calculated.

4.2.2.1 Compression Algorithm

1. The predicted sample is down- sampled from 16 to 12 bits. The difference is then found between the original sample and the predicted sample.

2. This difference is quantized to a signed 4-bit value, which becomes the new sample. The new sample is the compressed information that will be transmitted.

3. This difference is now added to the previously predicted sample, yielding the new predicted sample to be used for the next original sample.

4. The step size is then adjusted using the new sample, which is done by moving the index of the step size table.

4.2.2.2 Decompression

1. To find the original sample we use the step size to find the linear difference.

2. Due to truncation errors during quantization, we add ½ to the original sample during decompression.

3. That linear difference is added to a predicted sample to find the new sample (really the original sample).

4. The step size is then adjusted using the original sample, which is done by moving the index of the step size table.

This algorithm faithfully recreates the original waveform. Our tests, where we encoded and decoded Windows Wave (WAV) files confirmed this, and found that ADPCM was more than adequate for voice transfer because it accommodates both high and low frequencies.

Although we used ADPCM, our software was designed to be able to incorporate any Codec by simply changing a function call.

4.2.3 Digital Filtering

When recording raw audio data, it is simplest to filter noise as soon as possible. The ideal filter would exhibit a passband between 300 and 3400Hz, which corresponds to the frequency range that carries speech's intelligibility.

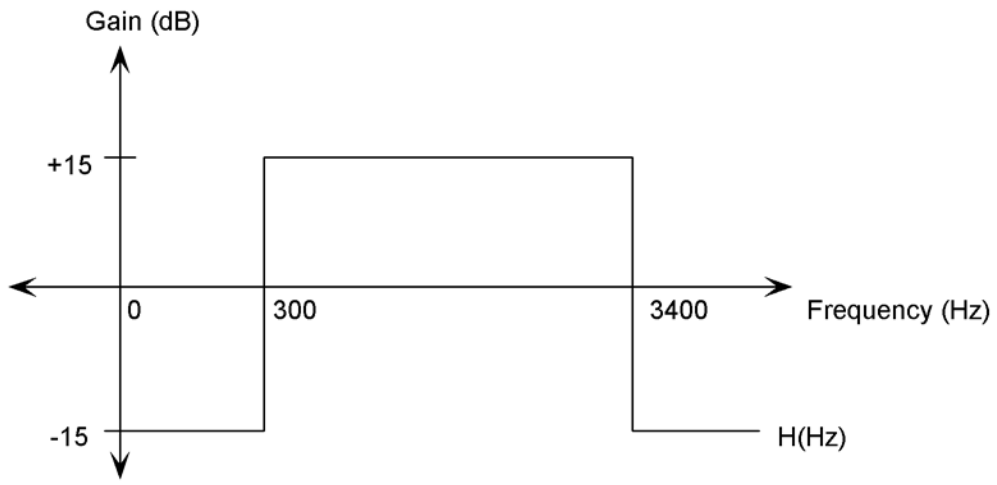


Figure 4-4: Ideal Transfer function for digital filter.

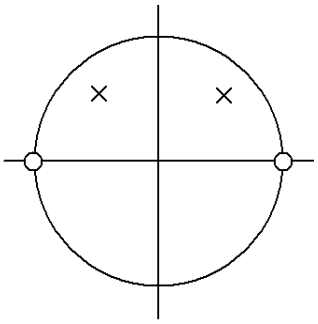
Such a digital filter is mathematically impossible to implement, but a digital filtering software module was designed to closely resemble this and to eliminate frequencies that do not contribute to speech's intelligibility.

Hardware is assumed to remove aliased signals, so the software only encounters data between 0 and 4000 Hz. Placing two zeros at the extremes of the frequency range eliminates the undesirable frequencies. Two poles are also placed at the outer frequencies of the pass band to retain the desirable frequencies. This gives a bell shaped filter, which is not ideal but is adequate for current purposes. Additional poles and zeros would increase the band-shaped look.

In the frequency domain (using Z - transforms)

$$H(z) = ((z - b2) * (z - b3)) / ((z - a2) * (z - a3))$$

Bringing the denominator over to the left hand side and producing a polynomial out of the roots we get:



In the time domain

$$a1 * y[t - 2] + a0 * y[t - 1] + y[t] = x[t] + b0 * x[t - 1] + b1 * x[t - 2]$$

Figure 4-5: Transfer function for a digital IIR filter and pole/zero plot

As the above figure shows, an Infinite Impulse Response (IIR) filter in the time domain was used to digitally filter the raw signal. An IIR is preferable to a Finite Impulse Response (FIR) filter because it requires fewer delay elements, thereby reducing latency.

Each buffer is individually filtered, but the last two samples of each buffer carry over to the next calculation (see the difference equation in Fig. 4-5). If a packet is dropped or the program has just begun, the starting samples are padded with zeros to prevent the filter from relying on erroneous data.

Implementing the filter requires a set of coefficients to be inputted into the difference equation. The IIR filter derives its coefficients from the centre frequency, the band gain and the slope of the band walls. The default values for these parameters are 1850 Hz, 10 dB and a 0.05 slope. This creates a bell shaped filter with a smoother band gain.

4.2.4 Silence Detection

Audio data on a computer consists of digitally raw input that must be sampled at the Nyquist rate in order to reconstruct the original wave.

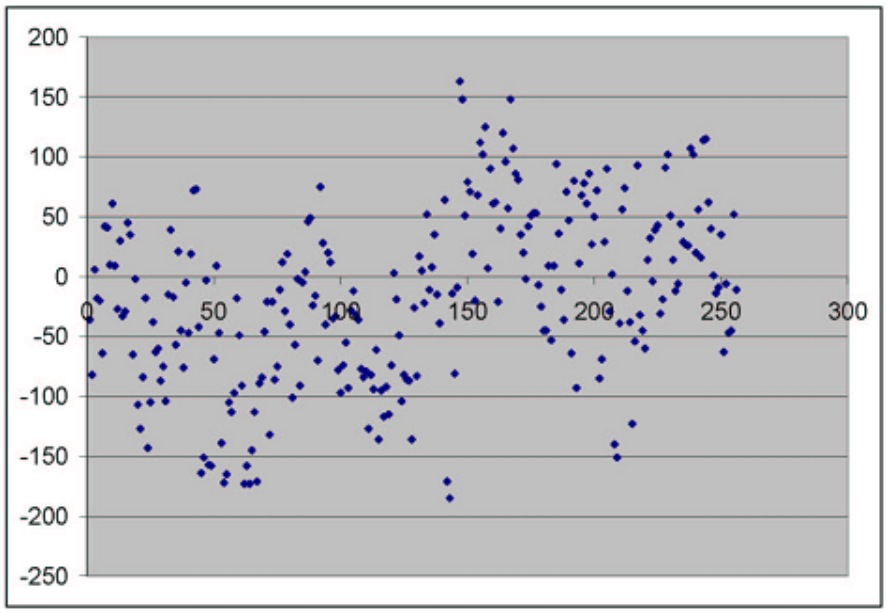


Figure 4-6: 256 samples of raw data input. This recording is of silence (no speech)

The above plot displays a small number of time domain samples of noise through a microphone and suggests the difficulty of filtering the desired signal from noise in the time domain. Larger sample sizes that include both silence and speech would be even more difficult to filter, especially because VoIP systems are constrained by bandwidth and processing time. These constraints can be overcome by reducing the volume of information to be processed and transmitted.

To this end, this system employs silence detection to reduce both data processing and transmission. Because frequency-based silence detection is computationally expensive and therefore incurs additional latency, this system instead calculates the energy of the signal in the buffer. The average energy in each buffer can be found by:

$$\text{Energy} = \text{Ln} \mid (\Sigma \text{ Buffer Samples}) / \text{Number of samples} \mid$$

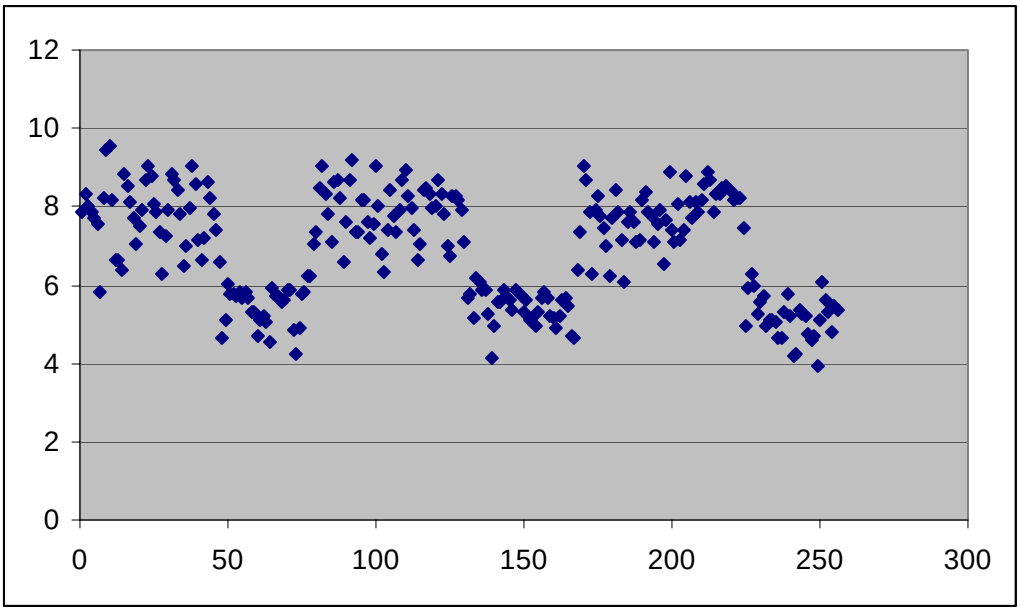


Figure 4-7: Averages of data buffers of raw data input. This recording is of silence and speech

The above recording shows speech (between 6 - 10 on the y-axis) and silence (between 4 and 6). Silence detection now only requires determining whether an averaged volume is above or below a threshold value.

Although this usually gives a good indication of when there is silence, buffers initially marked as silent may in fact convey useful information. For example, people briefly pause between words and take longer pauses between sentences in order to make their communications coherent. Moreover, the level of energy that corresponds to silence varies according to the environmental backdrop.

4.2.4.1 Compensation for Pauses in Speech

A simple algorithm was devised to distinguish between silence and information-conveying pauses. An audio stream is categorized as silence (and therefore discarded) when five consecutive buffers contain energy below the threshold value.

Essentially, the two states are silence and not silence. The system is initialised in the "not silence" state and enters the "silence" state when five consecutive energies are detected below the threshold. The counter resets when an energy level occurs that it above the minimum threshold. Similarly, a transition from silence to non-silence occurs when two consecutive energies above the threshold are recorded. OpenH.323 employs a similar method [8].

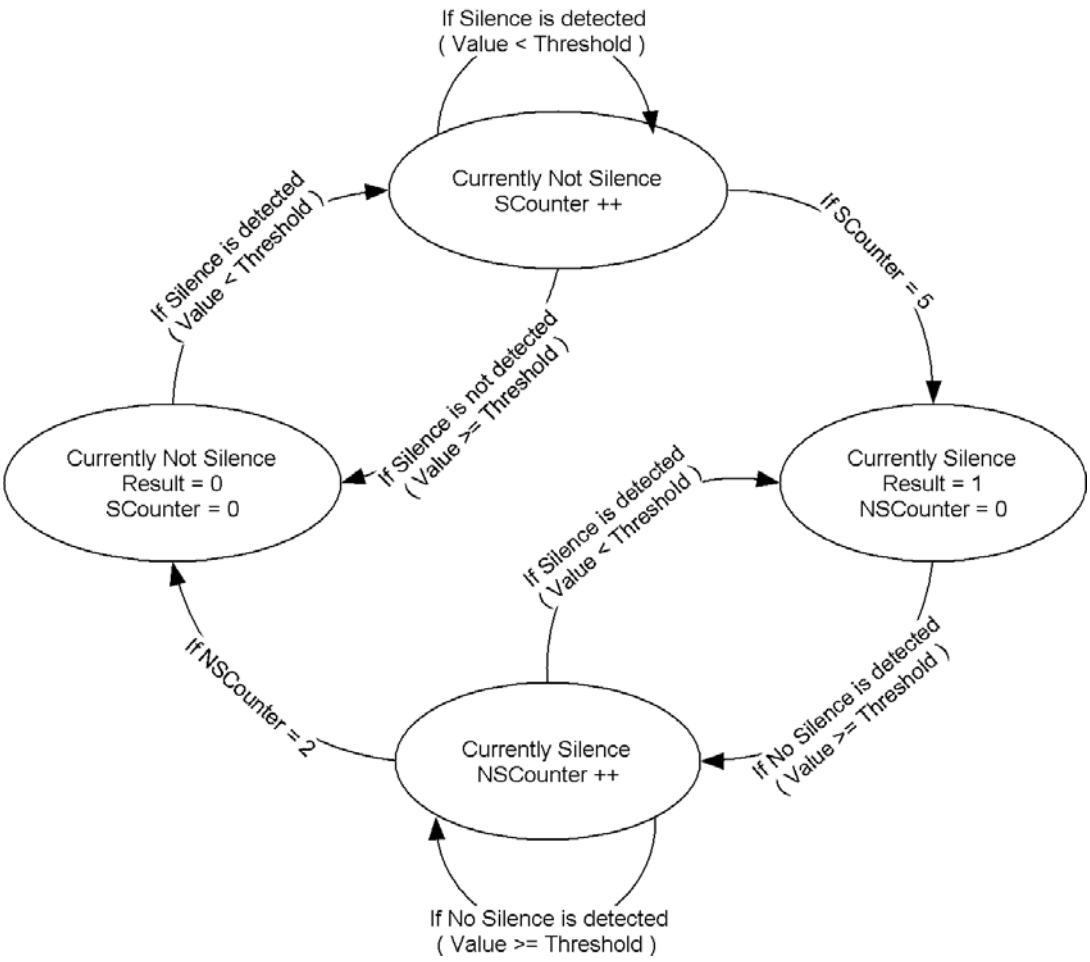


Figure 4-8: FSM of extended detection of silence and non-silence for speech breaks.

4.2.4.2 Threshold Adjustment

Energy levels vary according to speaker, equipment, and background. Assigning a fixed threshold will therefore result in the system either not classifying enough data as silence, or, conversely, registering too much data as silence. This system therefore instead employs dynamic thresholding, which sets cut-off energy levels that fluctuate in accordance with the extreme energy levels.

A buffer's first two energy levels are used to provide a reference maximum and minimum to estimate an appropriate threshold. Testing showed that a threshold of 40% between these energy levels proved satisfactory. Once this is done, the following algorithm is performed for every packet:

1. The average energy of the current packet is determined

2. The current energy level is compared to the silence threshold energy level

    a. If it is below this threshold:

        Reference Minimum = 0.95 * (Reference Minimum) + 0.05*(Current Energy Level)

    b. If it is above this threshold:

        Reference Maximum = 0.95 * (Reference Maximum) + 0.05*(Current Energy Level)

3. The threshold is adjusted to fall between 40% of the reference maxima and minima

This dynamic threshold algorithm compensates for any sudden and prolonged increase in background noise by continuously increasing threshold values until the thresholded values and the background noise are at the same level.

This detector was also employed in the system to reduce data exchange rates and reduce audio feedback from the hardware crossover device.

4.2.5 Volume Control / Automatic Gain Control

Operating system functions that request audio data from a device return this data in an encoded signed digital format. Volume adjustment is required to adjust these data feeds—which differ with sound card and microphone combinations—to acceptable volume levels. This software does so by employing both static volume control and automatic gain control.

4.2.5.1 Static Volume Control

In Static Volume Control, the user assigns a specific value multiplier that adjusts every digital sample by a constant. The output can be found by:

New Sample = Multiplier * Old Sample

The new sample is found by multiplying as opposed to dividing variables in order to prevent division by zero. Negative numbers are avoided because the sampled data is signed. It is

necessary to ensure that the calculated value does not exceed a sample's upper bound. This is done with the following check (n = the number of bits per sample).

$$\text{If New Sample} > 2^{(n-1)} - 1 \text{ then}$$

$$\text{New Sample} = 2^{(n-1)} - 1$$

$$\text{Else if New Sample} < 2^{(n-1)} \text{ then}$$

$$\text{New Sample} = 2^{(n-1)} - 1$$

Clipping results when the data does not fall between the maximum and minimum allowable values and is undesirable because it distorts the sampled data.

Static Volume Control is easily implemented, but provides unsatisfactory volume control due to the unavoidable clipping that results. It is also impractical because it requires constant manual input to adjust to the surrounding environment. For example, someone assigning a volume multiplier to listen to someone whispering would experience the effects of clipping upon resuming normal conversation.

Dynamically controlling the volume is therefore preferable. This method is called Automatic Gain Control (AGC).

4.2.5.2 Automatic Gain Control

AGC relies on continual feedback from raw sampled data in order to adjust the volume multiplier. There exist many ways of doing this, but this system incorporates only two methods.

4.2.5.3 Dynamic Volume Control

Dynamic Volume Control smoothens the transitions between loud and soft audio, thereby adapting to abrupt changes in volume and resulting in a continuous stream of clip-free sound. The algorithm used is listed below. (n = the number of previously recorded sound buffers)

1. The average volume of n previous sound buffers is found
2. The average volume of current sound buffer is found
3. The scaling factor is determined by

Scaling Factor = Previous Average / Current Average

4. The scaling factor is ensured to not be out of bounds

5. The current buffer is adjusted with the new scaling factor

6. The oldest buffer is rotated out and the newest scaled buffer is rotated in.

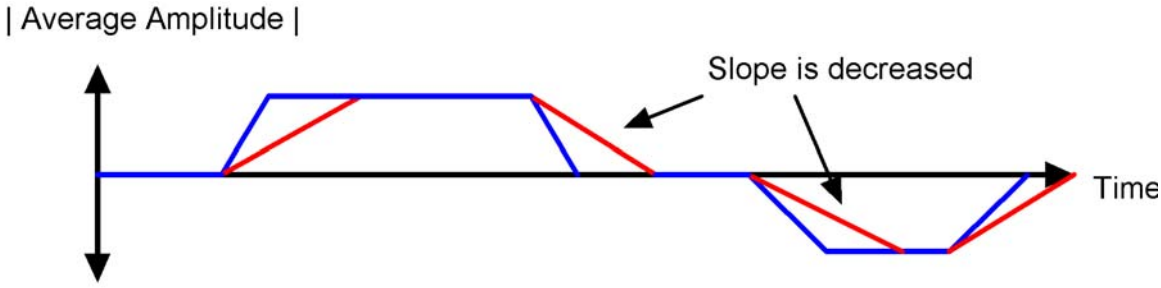


Figure 4-9: The effect of Dynamic Volume Control on average volume (adjustment in red)

Despite its advantages, DVC is not ideal. Moving Volume Control (MVC) improves upon DVC by adjusting all volumes to be at or near a specified value.

4.2.5.4 Moving Volume Control (MVC)

Moving Volume Control is superior to AGC because it adjusts the buffer volume to an exact value instead of multiplying it by a constant factor. The scaling factor used in MVC is bounded by maximum and minimum values. The output of the MVC algorithm is determined by:

1. The average volume of current sound buffer is calculated

2. The scaling factor is determined:

Scaling Factor = Pre-assigned Volume / Current Average

3. The scaling factor is verified to not be out of bounds

4. The current buffer is adjusted using the new scaling factor

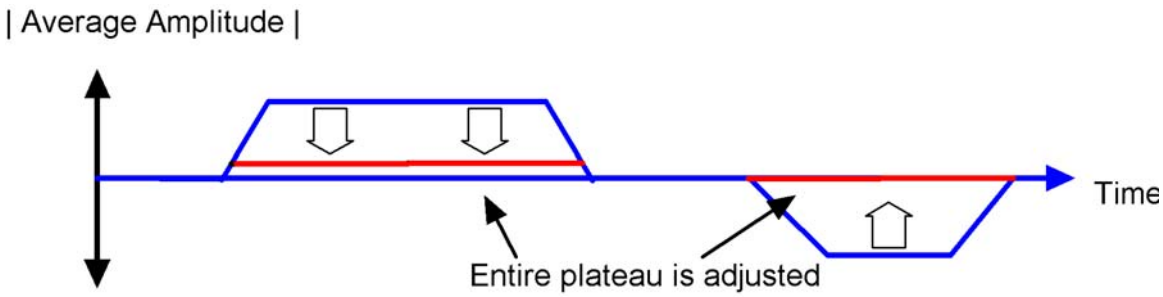


Figure 4-10: The effect of Moving Volume Control on average volume (adjustment in red)

This is a simple method for adjusting volume, but requires experimentation to determine an appropriate value for the "volume bar". Despite its simplicity, MVC provides the best performance because a whisper and a shout are adjusted to similar values, thereby ensuring clear communication irrespective of the voice, background, soundcard and microphone being used.

4.2.6 Dynamic Buffer Queue

When considering multiple design alternatives in the protocol, an abstract method was required for adding units of data into a buffer from one thread while removing units of data at a constant rate from another thread. As UDP packets arrive over the network, their audio contents would be extracted and added to the buffer. Data would arrive at variable intervals, due to packet loss and router congestion. However, audio data must be removed from the buffer at a constant rate in order to play the sound on the PC sound card.

This need motivated a "leaky-bucket" design, implemented as a thread-safe dynamic buffer queue.

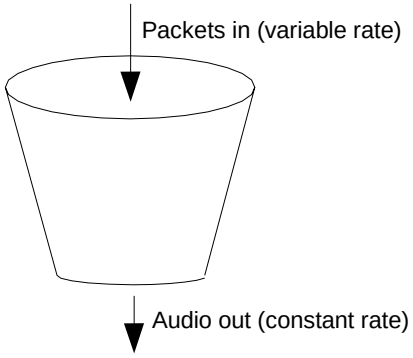


Figure 4-11: The Leaky Bucket

The queue must be thread-safe because two different threads will operate on it simultaneously. The networking-side thread will be inserting data received from UDP packets, while the audio-side thread will be removing data for the sound card. Both threads might try to access the queue at the same time, leading to race conditions and software instability.

Our software solution to this problem used win32 Critical Sections that provide mutual exclusion (mutex) access to the shared buffer resource. Buffer elements within the queue can be dynamically allocated, so they can support varying sizes. This entire **queue.c** software module was written in C for high performance and low overhead. Table 4-1 explains the software facilities provided by the module, separated for portability into queue.h:

Function	***Description***	***Thread-safe?***
`int q_init()`	*Initialise queue*	*No*
`int q_cleanup()`	*Cleanup queue*	*No*
`int q_enter(int len, void* buf)`	*Enter data block into queue. A new buffer is internally allocated for protected data storage.*	*Yes*
`int q_leave(int maxlen, void* destbuf, int* stored)`	*Return next data block from queue, de-allocating the internal buffer while leaving queue.*	*Yes*
`int q_length()`	*Query the queue length*	*Yes*

Table 4-1: Facilities provided by queue.h

4.3 Encryption module

One of the aims of our software is to safeguard the privacy of its users. Being easily intercepted, Internet traffic must be encrypted to remain private. Sloppy implementation has historically been encryption's Achilles heel, nullifying the protection of otherwise perfectly valid encryption. Furthermore, security schemes applied without care are doubly dangerous as they convey a deceptive sense of protection. These serious issues shaped the approach we took developing a security layer.

4.3.1 Choice of Cipher

In an effort to explain how we chose our encryption scheme, we divide encryption schemes along two lines, symmetric and asymmetric, and then block and stream.

Symmetric key encryption utilizes the same key for encryption and decryption, a simple realization being XOR. Asymmetric encryption uses different keys for encryption and decryption, an example being modular exponentiation employed in Rivest, Shamir, & Adleman (RSA). Public key cryptography allows the public key to be freely distributed but incurs additional computational overhead over private key encryption. An elegant solution sees private keys exchanged over an asymmetrically encrypted channel; this is the basis of the Diffie-Hellman secure key exchange. We opted to employ a symmetric cipher because the Directory Service Server (DSS) provides a Secure Sockets Layer (SSL) channel through which our application is able exchange keys.

Our protocol demands that levels above the network layer be tolerant of lost data. This restricted us to using a block cipher in over a stream cipher because frames encrypted with the former are independent of previous frames, and have no bearing following frames. Thus, dropped packets do no invoke any reaction from the security layer. A stream cipher could only function if we allowed retransmissions of lost or corrupted data –an unacceptable cost to latency.

Our final choice is the Blowfish algorithm, which though fairly new has been adopted into the Linux kernel –a good indication of its strength. Being freely available and widely evaluated it achieves security unattainable by proprietary encryption schemes. We preferred Blowfish to similar freely available algorithms like triple DES because it boasts a significantly larger maximum key size of 448 bits as opposed to 56 for triple DES, the latter being on the verge of breakability. In fact, every conversation our application carries is encrypted non-negotiably with a 448-bit key. Security is not a setting that can be overlooked or incorrectly configured; it is built into the architecture of our system.

4.3.2 Secure Key Exchange

Not to be confused with a Diffie-Hellman style key exchange, ours takes place over a pre-existing SSL channel. As both hosts use the same key, this is unidirectional. Fig 4.14 illustrates the connection establishment process. The key to be used in the potential conversation is randomly generated by the host that initiates the connection (Peer 2 in Fig 4.14). The key is included in the original connection request, and if the connection is successfully established, all subsequent data exchanged between the hosts over the UDP channel is encrypted with this key.

We observe that users must trust the DSS, through which the key is exchanged. To clarify we note that peers communicate to each other through two SSL connections: one between

one peer and the DSS and the other between the DSS and the other peer. A compelling concern stems from a compromised DSS. A simple solution would employ a Diffie-Hellman style secure key exchange over either the DSS SSL channel or alternatively, over the UDP connection preceding the commencement of communication. The former may be preferred, as a Diffie-Hellman style exchange demands a reliable interchange of data, which implemented over UDP requires timeout and retransmission mechanisms.

4.3.3 Cipher Implementation

To dodge possible security loopholes, we decided to implement a version of the algorithm written by Bruce Schneier, the inventor of the cipher and a preeminent computer security expert. The algorithm code can be found in blowfish.c and is initialized with a call to InitializeBlowfish() to which we pass our key and its length. The key is generated by our makeKey() function that employs the rand() pseudorandom number generator from stdlib in Windows. Ideally we wish to have a true random number generator that generates a flat distribution of random numbers. Unfortunately, *true* random number generators are very difficult to implement on a computer.

Blowfish encrypts blocks of 8 bytes of data to blocks of 8 bytes of encrypted data, making its block length 8 bytes. The algorithm we employed treats a block as two unsigned long integers (64 bits total), and encrypts and decrypts data one block at a time. Our system handles data as character arrays, so we wrote wrapper functions, decrypt and encrypt (found in controller.c) that respectively decrypt and encrypt an entire buffer of data at a time. As caveats, data must be aligned to 64 bits, and there is no implicit means of calculating the length of encrypted data.

Encryption is the final transformation that data undergoes before it is sent across the Internet. We decided to implement it as final step as to avoid potentially misjudging what elements of communication are sensitive. The final layer of encapsulation includes a length field that enables the decrypt function to determine the size of the original data, which if not already aligned to 64 bits is padded before encryption.

4.4 Graphical User Interface

The GUI serves as the front end to our system. At the user's direction, it commands the client software, placing and answering calls.

All windows reside in a main window that is displayed when the program starts. The program can be minimized to the system tray and be activated again by clicking its icon, which shows a telephone handset. An incoming call automatically maximizes the program window and plays a ring sound file, alerting the user of the computer.

The windows internal to the main window are: contacts, settings, help/about, call session (up to twenty of these), and incoming call. The contact window lists users who can be called by double clicking their name. The settings window allows the user to change his/her username and password. The help/about window identifies the version of the software.

One call session window is displayed for every call in progress. It indicates the status of the call (connecting, connected, or disconnected), identity of the remote user, and the call's duration. During the connection establishment phase it also indicates the current progress of the firewall circumvention.

An incoming call invokes the incoming call window, allowing the user to accept or decline an incoming call. If the user selects neither after ten seconds, the call times out and the window disappears, and the other user is notified that there was no answer.

4.5 Controller

The controller ties together the disparate modules in our system into the DLL controller.dll. Fig. 4.12 relates the threads that run concurrently during the program's execution. Program execution commences with the OS loading the GUI, which immediately calls initController(). This function initializes the resources it will subsequently use and starts the Audio Capture thread that will run concurrently with the program. The complementary function, shutdownController(), is called before the GUI exits and terminates the Audio Capture thread and cleans up the resources used.

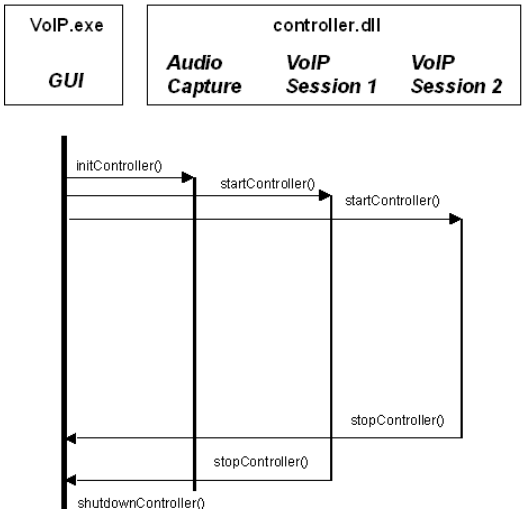


Figure 4-12: Thread Concurrency Diagram

The GUI creates a VoIP Session when the user receives and then accepts a peer's request to start a conversation. A VoIP Session is comprised of several objects that straddle the GUI, TPC, DSC, and the controller. We note that the controller is not directly linked (minus one function) to the DSC or TPC and that the GUI serves as a bridge between the controller and these components by passing the controller a newly created socket and TPC context object upon a successful connection.

The Audio Capture function may be considered a thread conceptually, though more accurately it is a function that the OS calls every 20ms (one atom). This function handles the details of audio extraction in Windows an in turn calls `sendAtom()`with a buffer of sound data.

`sendAtom()` traverses a linked list of current connections/sessions, and for each one, encapsulates (as per our protocol), encrypts, and sends the current buffer.

The full functionality of the controller is realised through countless ancillary functions, which in the aim of clarity have been omitted from discussion. The interested reader can find these functions in their entirety in Appendix B.

4.6 Connection Establishment

A defining aspect to our VoIP system is its innovative connection establishment system. This system's Internet application is unique in that it can establish a direct connection between

two hosts that both reside behind separate firewalls. This has a marked effect on latency and simplifies overall system design.

4.6.1 Connectivity in the presence of NAT

NAT is system whereby several hosts on a LAN share one or more Internet routable IP addresses. The firewall is the machine that bridges the LAN to the WAN, and it is also often referred to as the gateway.

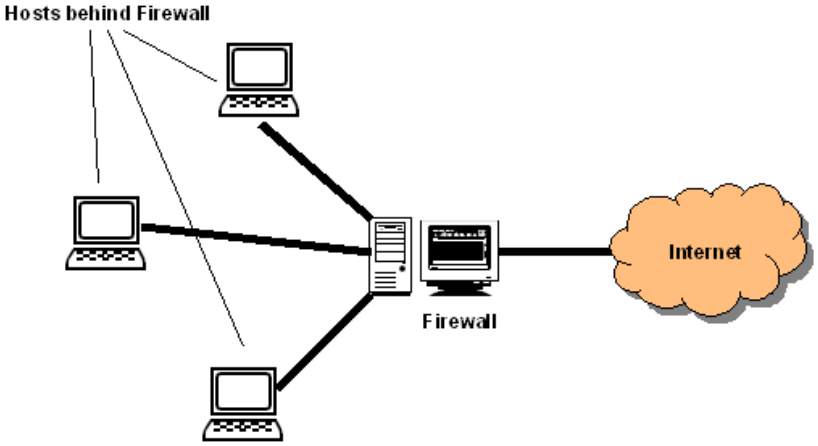


Figure 4-13: A typical NAT configuration

A host on the LAN can communicate to other hosts connected to the Internet via the gateway, if and only if the host behind the gateway initiates the connection. In other words, outgoing connections are permitted, while incoming connections are not. Ordinarily this is inconsequential because hosts usually communicate with servers that do not reside behind firewalls. In this situation, NAT is transparent to the end user.

Unfortunately, two hosts behind separate gateways cannot directly connect to each other. A popular solution to this problem is to use an additional server that both firewalled parties connect to and through which they exchange information. This is an undesirable solution because it requires additional infrastructure, consumes greater bandwidth, and incurs additional latency, as data packets must take a circuitous path from host to host and must be buffered at the server. This last drawback becomes especially problematic when the server is overloaded, and it may acutely affect VoIP systems.

The additional latency incurred by using a central server to relay VoIP data is heavily dependent on network topology and congestion and cannot be simply modeled. To obtain an

intuitive sense of how detrimental using third parties can be when using third parties, consider two neighbors in Australia who wish to have a "real-time" conversation through the Internet, but are constrained to using a connection-bridging server. If the server is based in New York, their conversation is routed through New York, sustaining several hundreds of milliseconds of additional latency. In fact, the delay in this connection would be *double* that of one between New York and Australia, as a simple ping and pong would traverse the ocean *four times*. Moreover, the two connections on which this method relies will also increase the jitter. Statistically, the resulting variance in jitter is not simply the sum of the two constituent variances. Rather a convolutional summation would be required to model jitter after a Gaussian distribution [20]).

Another potential solution to circumventing NAT involves capable users configuring their firewall to allow incoming traffic through to a Demilitarized Zone (DMZ), or the host(s) that wish to establish direct connections. While effective, this solution requires that end users be sufficiently familiar with Internet technology to carry out this operation, and such people are uncommon. A new technology named Universal Plug and Play (uPnP) seeks to, among other aims,automatically configure firewalls to allow direct connections. Despite its optimistic name, uPnP is not universal, and its adoption by Internet Service Providers (ISPs) that employ NAT on a large scale seems technically infeasible.

A better solution is clearly required. This issue is becoming particularly important as home users (for whom this project is designed) are increasingly installing hardware routers that serve the dual purpose of sharing an Internet connection and fending off malicious traffic.

4.6.2 Direct Connection Establishment in a Double NAT Environment

Our system can establish a direct UDP connection between two hosts residing behind separate firewalls, irrespective of the specific type of NAT being used, and requires no configuration on the part of the end users. The system comprises a pair of servers that this group has tentatively named a DSS, and a Third Party Server (TPS). These servers help the two firewalled hosts initiate the direct connection, but then withdraws once the connection has been established. We will briefly discuss the functions of the DSS and TPS will be briefly discussed. Interested readers are invited to consult Appendix B for further details.

The DSS provides a means for users to locate each other and subsequently negotiate a connection. Users have accounts on the DSS that are reminiscent of email accounts. For example a user, Alice, who has an account at the DSS, wonderland.ca, could be reached at

"alice@wonderland.ca". This abstracts otherwise confusing and difficult to obtain network information such as IP addresses and ports, which means, for example, that Bob does not have to discover Alice's gateway's external IP address, in order to communicate with her.

The TPS, meanwhile, carries out the second half of the connection establishment by providing carefully timed exchanges of data that is required setting up of a connection. The TPS provides the actual NAT circumvention functionality.

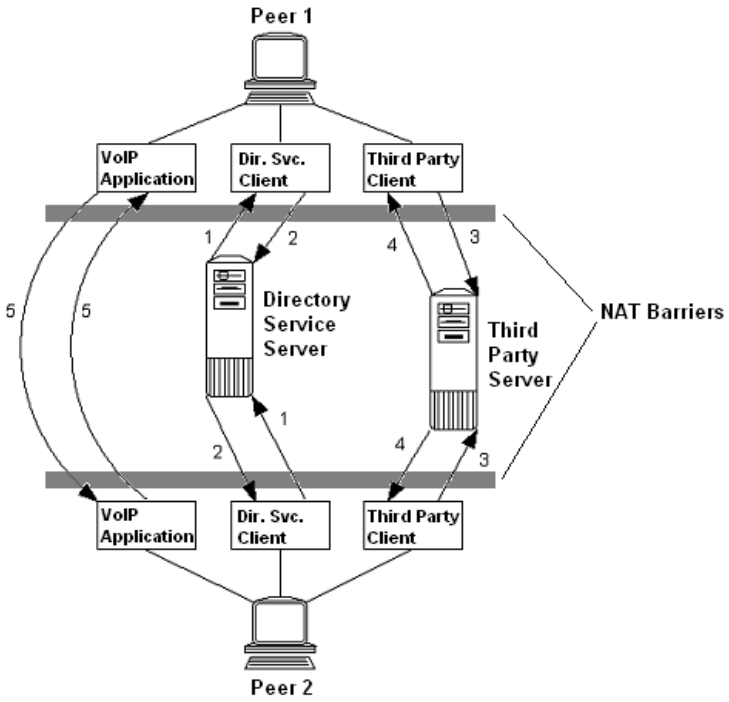


Figure 4-14: Diagram illustrating the steps in establishing a direct connection

Figure 4-14, shows the steps required to establish a connection when two users are logged into the DSS and wish to connect to the other. The process commences with Peer 2 issuing a connection request to Peer 1 through the DSS. Peer 1 accepts this request through the DSS, and the process is passed onto the mutually agreed upon TPS. Both parties then send seed ports to the TPS while the TPS waits for both hosts to become synchronized. The TPS then forwards these seed ports and additional information to the hosts, which then begin scanning each other's gateways with UDP packets bearing carefully selected destination ports that are calculated from the seed ports. A short handshake ensues once at least one packet is admitted through either gateway, after which the Third Party Software returns the newly created socket returns to our VoIP application.

Though fairly involved, this process is completed rather quickly and usually produces a socket pair in approximately two seconds. This system is also secure; all communications with the DSS occur over an SSL channel, and users are required to log in with a password, thus safeguarding against identity theft. Additionally, a unique session key is generated and securely disseminated to each host for every connection established via the SSL connection, preventing simultaneous connection attempts from interfering with each other, and preventing malicious users on a hostile network from "stealing" incoming connections. However, neither of these security mechanisms guard against man in the middle attacks and (like most security measures) these rely entirely on the integrity of DNS.

4.7 Data Encapsulation

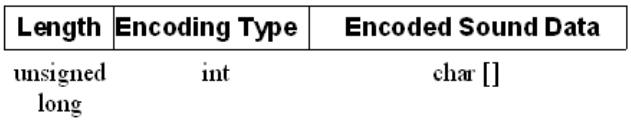


Figure 4-15: Encapsulation of encoded sound atom

Encoded sound atoms are encapsulated to include both length and encoding type. These fields instruct the decoder to use a particular algorithm and inform the decoder of the length of data to be decoded. This stateless design simplifies the protocol by eliminating the need for handshaking and Codec negotiation. Codecs can moreover be switched on the fly in response to changing network conditions.

The following shows the final encapsulation layer. An atom is encrypted and transmitted after being wrapped in this layer. The data length cannot be recovered implicitly because the packet is aligned to 8 bytes before encryption, necessitating the Length field. The Data field contains the encapsulated encoded sound atom (Fig. 4-15). The sequence number allows the program to detect packets that have been dropped and that arrive out of order (which the program discards). The Flags field allows future extensions to be easily implemented.

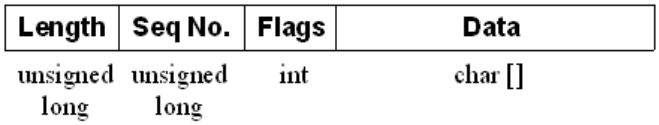


Figure 4-16: Final encapsulation of data

The minimalist encapsulation is a natural extension of the network layer design for VoIP. It has been immensely useful in improving the project's modularity and in enabling the software to be extended and modified in an elegant manner.

5. Hardware Design

5.1 Description of hardware

As described in the Introduction, the purpose of the hardware is to connect the computer to the analogue telephone lines, often referred to as Plain Old Telephone Service (POTS), in order to allow our VoIP calls to dial out and talk to people at real telephone numbers. This implies that the external hardware device has to do a number of different things:

- Receive control signals from the computer, and possibly send control signals back
- Input and output audio to and from the computer
- Provide a proper electrical interface to phone line
- Input and output audio to and from the telephone line

Circuits to accomplish each of these functions can be built individually, and this is the approach we took to the hardware design. The goal in the end was to produce a single large circuit that could do all of the above functions. Ideally, this device could be connected to the telephone line and the computer and provide a safe, reliable computer/POTS interface. The design process was very much modular in nature.

The circuits were first constructed on modular breadboard (spring-loaded boards) in an experimental fashion, adjusting as necessary to maximize audio quality while minimizing complexity and minimizing power consumption. After thorough testing on breadboard, a Printed Circuit Board (PCB) layout was constructed and finally built by hand.

5.2 Telephone line interfacing

The analogue telephone line (POTS) available in nearly all North American households is a simple audio interface consisting of two wires, with relatively standard electrical characteristics. There are potentially high voltages on the telephone line, which poses a risk to any computer we may connect. We carefully considered the following line conditions while developing an appropriate interface circuit [27]:

- Phone on-hook, DC voltage approx. 48V (no current)

- Phone off-hook, DC voltage drops to approx 5V and equipment draws approx 25mA
- As long as current flows, line is in use [3]
- Ringing adds approx. 90V AC superimposed on 48V DC. Peak voltage can exceed 138V
- 1:1 audio transformer is standard interface procedure

We started by building a circuit to handle these basic electrical conditions. Since our goals are to be able to input and output audio to and from the telephone line, we include connections to standard audio connectors. Fig. 5-1 demonstrates the basic POTS interface.

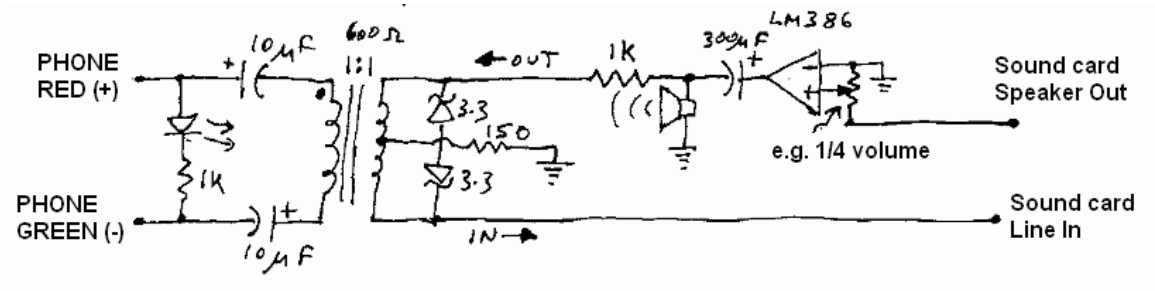


Figure 5-1: Basic POTS Interface

The left side of the circuit connects to the two wires of the household telephone line. While wiring varies from house to house, the red telephone line is usually positive with respect to the green telephone line. The LED, with limiting resistor, lights up when the telephone line is connected with the correct polarity. The polarity of the phone line is an important consideration due to the polarized electrolytic capacitors, which can become damaged if exposed to a large reverse bias. The LED serves a dual-purpose, as it also keeps the phone line active by drawing current.

The two coupling capacitors block the DC on the telephone line, such that the audio AC signal reaches the transformer. We used a 600Ω (telephone standard) 1:1 centre-tapped transformer to isolate the POTS-side of the circuit from the audio circuitry. The back-to-back low voltage zener diodes provide over-voltage protection by conducting in the case of a high AC voltage on the transformer's secondary [5]. This high voltage can occur if there is a telephone ring, which results in a 90V AC signal.

The centre tap of the transformer becomes audio-ground while the two other ends of the transformer are used for audio input and output. Included in the audio output path is an LM386 (power audio amplifier), in order to boost and isolate the source audio signal. A speaker can

optionally be connected after the LM386 in order to "monitor" the audio being output. The audio input path is a direct tie between the other end of the transformer, and the destination audio port.

This basic circuit was constructed and connected to both the telephone line and the PC sound card. It worked as intended, allowing the computer to record audio from the telephone line and simultaneously output audio to the telephone line. The audio quality was very acceptable, and the power amplifier provided adequate volume control.

Tests with a digital multi-meter confirmed safe voltage levels on the audio-side of the circuit, even when the telephone line was ringing. This completed the basic POTS interface, which will provide the essential telephone-line interfacing function for the overall hardware. The only thing missing is a physical relay to automatically control the connection to the telephone line. Otherwise, this circuit always draws current and therefore always stays connected to the phone line. The control circuit will be responsible for bringing the entire hardware device on or off-hook.

5.3 Control circuit design

While the basic POTS interface was completed, the hardware design was still lacking a way to exchange control information. We wanted the software to be able to automatically control the hardware, which means that in the very least the software has to be able to instruct the hardware to go off-hook (pick up the telephone line).

The basic POTS interface shown in Fig. 5-1 only provides an audio interface. This means that currently, the hardware only connects to the computer via the PC sound card. With the desire to keep the complexity of the computer/hardware interface to a minimum, we investigated the possibility of using special audio signals to send the hardware control commands. This would mean that the two audio cables alone would be sufficient for both audio I/O and control (in-band signaling).

5.3.1 Control option: DTMF

The most attractive possibility was using Dual-Tone Multi-Frequency (DTMF) audio signals in order to control our hardware, as these are often used in industrial designs to carry in-

band signaling information. It is easy to generate DTMF tones in the computer, but the hard part is *accurately detecting* the tones in the hardware.

We started by trying active bandpass filters in order to detect single tones, based on a popular single op-amp design [32]. By selecting component values and evaluating the frequency response, we built circuits that were tuned to specific frequencies.

$$A(s) = \frac{-s\left(\frac{\alpha}{R4\,C1}\right)}{s^2 + s\left(\frac{1}{C1} + \frac{1}{C2}\right)\left(\frac{1}{R3}\right) + \frac{1}{R3\,R4\,C1\,C2}}$$

Transfer function of single op-amp band-pass filter

The corresponding tones were generated on the computer (using CoolEdit) and output to the bandpass circuit. As an example, we built a circuit with two bandpass filters to see if our circuit could tell the difference between these two tones, 300Hz and 500Hz, and demonstrated by the following plots.

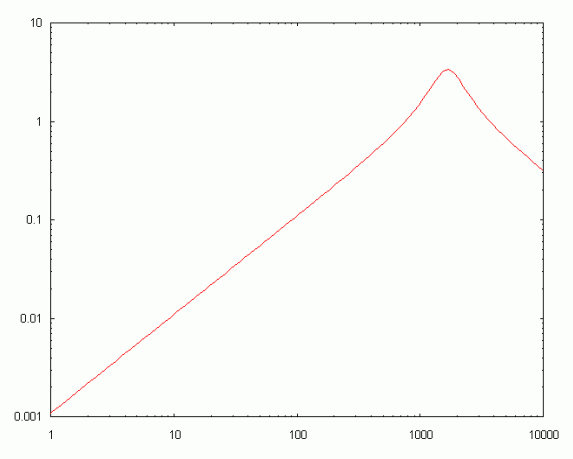


Figure 5-2: Band-pass filter passing 300Hz

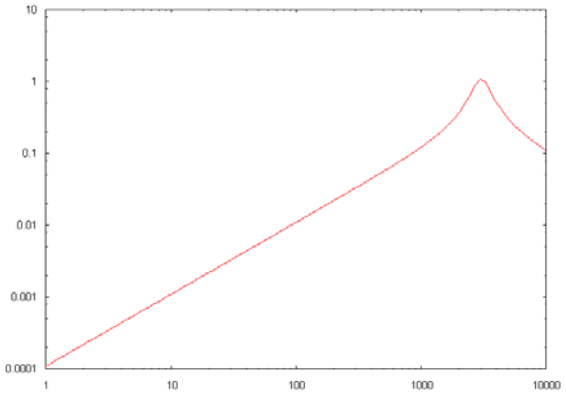


Figure 5-3: Band-pass filter passing 500Hz

Unfortunately, the actual results were quite unsatisfactory. The circuit designed to detect Tone A (300Hz) in fact detected an approx. 180Hz tone. And the circuit designed to detect Tone B (500Hz) in fact detected an approx. 400Hz tone. When additional circuitry was added on in order to "digitize" the detection (by way of comparators), the accuracy of the response further slipped. The result was that the range over which tones were detected overlapped significantly, even though the theoretical circuits were designed to have peak frequency responses several hundred Hz apart.

Because of the inaccuracy of our tone detector circuits, we decided to abandon this route based on DTMF signal detection. The number of op amps and comparators required to implement a proper DTMF detector would be quite large. However, this in-band signaling method (based on DTMF) could still be implemented, perhaps by using a specialized tone detector. For our design, however, an alternative digital interface to the computer would accomplish the same job more easily.

5.3.2 Control option: Parallel port

Since sending control information via DTMF tones appeared to be infeasible, we pursued an alternate route: the PC's parallel port. The standard parallel port provides TTL-compatible logic and provides numerous input and outputs. Software can set outputs and read inputs through multiple x86 ports (for example, 0x378 for LPT1) while an external circuit can do TTL I/O to read and write values. The interface is purely digital, meaning that individual bits accessible via the PC's ports correspond to individual signal lines on the parallel port's DB25 connector [1].

While the parallel port promises to simplify the hardware design due to its digital nature, it creates some additional complexity on the software side. In newer OSs, access to the parallel port is strictly limited to privileged processes. As a result, there is no direct way for our software to do parallel port I/O within the aid of a kernel driver of some sort.

The WinIo software is capable of providing exactly this type of direct hardware I/O. We proceeded to create a number of simple test circuits to verify that we could read and write values to our external circuit through the parallel port, using the driver wrapper provided by WinIo. Once we confirmed that the interface was working reliably, we started to build several different functions that used the digital control interface:

1. **Relay control**. A digital signal through software should actuate the relay on demand, bringing our hardware device onto the telephone line.
2. **Power-on status**. Our software should be able to query the hardware device to see if it's powered on. This allows diagnostics in case our hardware is disconnected or powered off.
3. **POTS status**. Our software should be able to query the hardware device to see if it is currently on the telephone line. This allows diagnostics in case the phone line is not connected.

At this point we had to consider electrical safety. The computer's parallel port is rather sensitive since it is wired directly to the PC main board. High voltages on the parallel port have the potential to cause significant damage to the computer itself, so we have to be very careful about connecting the parallel port to a circuit with the high voltages inherent to POTS.

The ideal route, which we chose to pursue, was to use opto-couplers (or opto-isolators) to electrically isolate the parallel port from the rest of our circuit. One side of the opto-coupler contains a Light Emitting Diode (LED), which allows the input signal to emit light within the coupler's package. The other side of the optical isolator contains a photo transistor, which is simply switched on when light hits it within the package. There is no electrical contact within this coupler package, which makes it ideal for interfacing a sensitive device (the computer) to a circuit containing potentially high voltages. Standard opto-couplers such as the H11A3 we used can withstand as much as 7.5kV while maintaining complete electrical isolation.

Two of the parallel port lines are used for input (status) while one line is used for output (relay control). The completed digital control circuit is shown in Fig. 5-4.

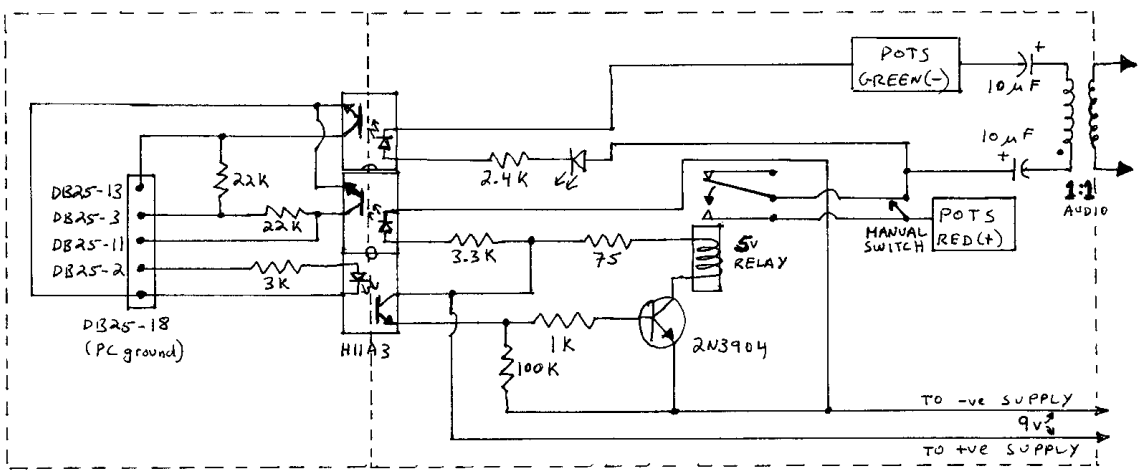


Figure 5-4: Complete control circuit

We were able to carefully configure the parallel port lines in order to do TTL input and output without requiring an external voltage source. The connector at the left side of Fig. 5-4 provides the physical connection to the computer's parallel port via a DB25 connector. Table 5-1 explains the mapping between the parallel port lines and bits within the computer's address space:

DB25 pin, name, and direction	***Mapping in x86 address space*** ***Often, BASEPORT = 0x378***	***Use in circuit***
2, D0 (output)	*Bit 0 of BASEPORT*	*Controls relay*
3, D1 (output)	*Bit 1 of BASEPORT*	*Fixed high to provide logic pull-up*
11, Busy (input)	*Bit 7 of BASEPORT+1*	*Detects power-on status*
13, SelectIn (input)	*Bit 4 of BASEPORT+1*	*Detects POTS status*
18, gnd	*N/a (electrical)*	

Table 5-1: Use of parallel port lines

Referring again to Fig. 5-4, DB25-2 (the relay control) lights the opto-coupler LED when high and causes the photo transistor on the other side of the circuit to switch on. The power NPN transistor, in a cascade configuration, then draws a large current through the relay coil causing the physical relay to switch "on" [6]. The hot side of the POTS line is gated through this relay,

allowing the relay (under software control) to bring the entire hardware device onto the telephone line.

For the two status-sensing functions, the opto-couplers are connected in reverse. DB25-11 and DB25-13, the two parallel port inputs, monitor the voltage level that is normally high due to being pulled up to the reference TTL high. When the respective photo transistors are turned on, however, they pull the inputs down to TTL low. We used a test meter to confirm that the low voltage level was well within the acceptable limits for TTL (0 to 0.8V is low, 2.5 to 5.0V is high). One photo transistor is turned on when current is drawn from the active POTS line. The second photo transistor is turned on so long as current is drawn from the circuit's power supply.

The 9V battery that powers this control circuit also controls the audio circuit, which is shown in the overall schematic in §5.4. Note that the standby current for the circuit shown in Fig. 5-4 is approximately 2mA, making it suitable for battery-powered operation over an extended period. The standby power consumption is under 20mW, but increases to 500mW when the relay is on. This is unavoidable, due to the large current required to actuate the relay.

We also wrote **pptest.c**, a simple C application that ran the control circuit through the basic functions in order to verify the expected functionality. This test application relies, of course, upon WinIo that provides low-level access to the hardware ports on the computer. The functions coded into **pptest.c** would later form the basis for the formal POTS.DLL, described in §5.5.

5.4 Final circuit

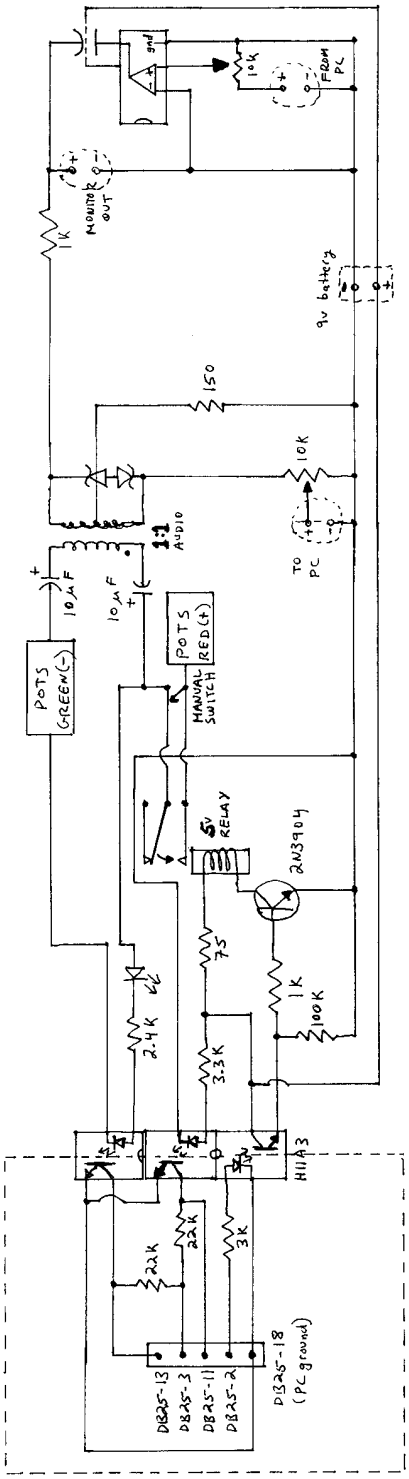


Figure 5-5: Final circuit schematic

The overall circuit was accomplished simply by combining the control circuit with an audio circuit, which is a slightly modified version of the basic POTS interface presented in §5.2. This final circuit was transferred to a single-sided PCB, closely following the above schematic. The components were soldered onto the resulting board, and the entire device was tested on multiple computers. The overall circuit worked as intended, with the only unexpected result being higher-than-expected overall power drain. This is attributable to the LM386 audio amplifier, which constantly draws approximately 13mA from the power supply.

The "From PC" connector is a mono audio connector to the sound card's "Speaker Out" plug. Because sound cards can have a wide range of volume, we included both a potentiometer and the audio amplifier in order to adjust the resulting audio level heard on the telephone line.

The "To PC" connector is a mono audio connector to the sound card's "Line In" plug. Experimentally, we found that the audio volume on the telephone line is approximately 5 times as loud as the audio levels expected at the sound card. For this reason, the potentiometer must be adjusted to decrease the amplitude of the audio signal sent to the sound card.

When both of these audio cables are connected to the sound card, and the volumes are adjusted properly, software on the computer can record from the telephone line and simultaneously play audio out to the telephone line. Along with the ability to control the relay digitally, this provides all the functionality we desired in our hardware device. The only problem that appeared immediately was due to the inherent design of the 2-wire telephone line: any audio signal we *output* is simultaneously heard as audio *input*. This means that there is unavoidable audio feedback, which is of concern to our VoIP software. We investigated numerous ways to solve this problem, which are covered later in §5.6.

5.5 Software interface, POTS.DLL

Even though the hardware device was completed, we needed to create the associated software in order to allow convenient access to our hardware from the VoIP application. We decided to write a Dynamic-Link Library (DLL) for Windows that would provide abstract access to our external device. POTS.DLL would hide all the electrical details of device I/O, and allow the VoIP software to use logical function calls to input and output control signals. Note that there is no special interface required for audio I/O, since this is accomplished via the PC's sound card.

POTS.DLL would further require the WinIo drivers in order to function, but this is not a concern so far as the VoIP application is concerned. We proceeded to write **pots.c** (a C program) which compiles into POTS.DLL. The associated header file, **pots.h**, provides a number of simple facilities to any application that links with our library, **pots.lib**. These facilities are described in Table 5-2.

Function call	***Description***
`int device_init(unsigned int)`	*Initialize the device on the specified port. The port number varies according to the PC, but is often 0x378 for the first parallel port (LPT1)*
`int device_powered_on()`	*Returns a boolean value indicating whether or not the external hardware device is powered on*
`int device_on_pots()`	*Returns a boolean value indicating whether or not the external device is on the telephone line*
`void device_set_relay(int)`	*Commands the external device to set the relay state to the boolean value indicated*
`int device_dial(const char*)`	*"Dials" a telephone number string by generating a DTMF tone sequence, sent to the sound card*

Table 5-2: Abstract facilities provided by pots.h

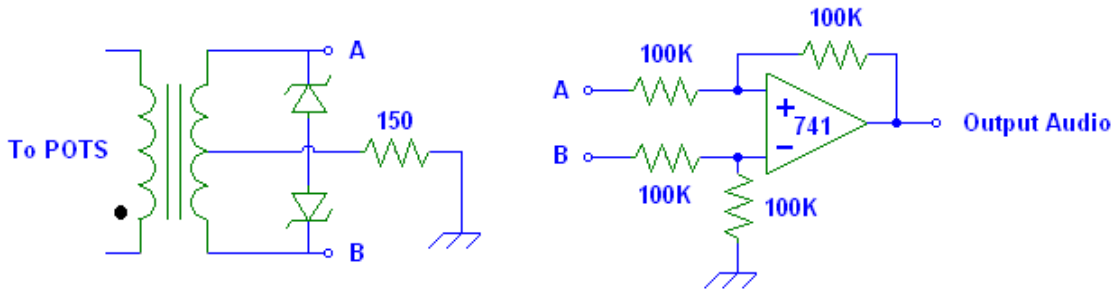
We tested POTS.DLL from **testpots.c** (a C program) that "exercises" the DLL. The test program attempts to pick up the telephone line, provides diagnostic feedback in case the device malfunctions, and ultimately dials a telephone line. The software worked as intended on all computers we tested it on.

5.6 Feedback Cancellation

The telephone system carries conversations over a single pair of twisted wires. Consequently the outgoing and incoming audio streams are mixed, and the signal we read at node B (Fig. 5-6) contains both the incoming audio from the POTS and the outgoing conversation originally injected at A. Unfortunately, this feedback becomes an echo to the caller using the our software, which as a function of round trip time that is in the order of tens of milliseconds, is very distracting.

5.6.1 Analogue Feedback Cancellation

We attempted to solve this problem by subtracting the original signal from the mixed signal using a simple operational amplifier summer circuit shown in Fig. 5-6.



(a) **(b)**

Figure 5-6: POTS interface sub circuit and summer feedback cancellation circuit

The feedback becoming even greater, this circuit failed to function as desired. We attribute this to the complex impedance of the isolation transformer, and the reflection of additional impedances from the POTS. If we neglect the impedance of the telephone line, and disregard the primary winding, we find that

$$V_B = \frac{10k\Omega * 150\Omega * V_I}{(0.5sL + 10k\Omega)(150\Omega) + (0.5sL + 1k\Omega)(150\Omega + 10k\Omega + 0.5sL)}$$

where V_B is the voltage at node B as per Fig. 5-7, V_I is the input voltage from the soundcard, and L is the impendence of secondary winding. The reader should consult Fig. 5-5 to note how the components omitted in Fig. 5-6 contribute to this equation.

Correcting this problem proved to be a veritable challenge, as the complex transfer function of the system (where we consider node A being the input, and B the output), yields a frequency dependent response.

In advancing our search for a solution, we made the following assumptions about the system:

- The system is linear, that is constituent components/signals can simply be summed. We note that transformers are not linear, but without this assumption further analysis would be prohibitively complicated.

- The POTS behaves as a current source when it is off hook.

We continued by trying to mimic this function, so that the signal being subtracted has undergone the same transformation. We were presented with several options for realising this transfer function, such as constructing a RC network. This alternative being fraught with complexity and requiring that we know L, had us favour more elegant solution where we employ a second identical transformer (Fig. 5-6 a). Fig. 5-7 illustrates this design.

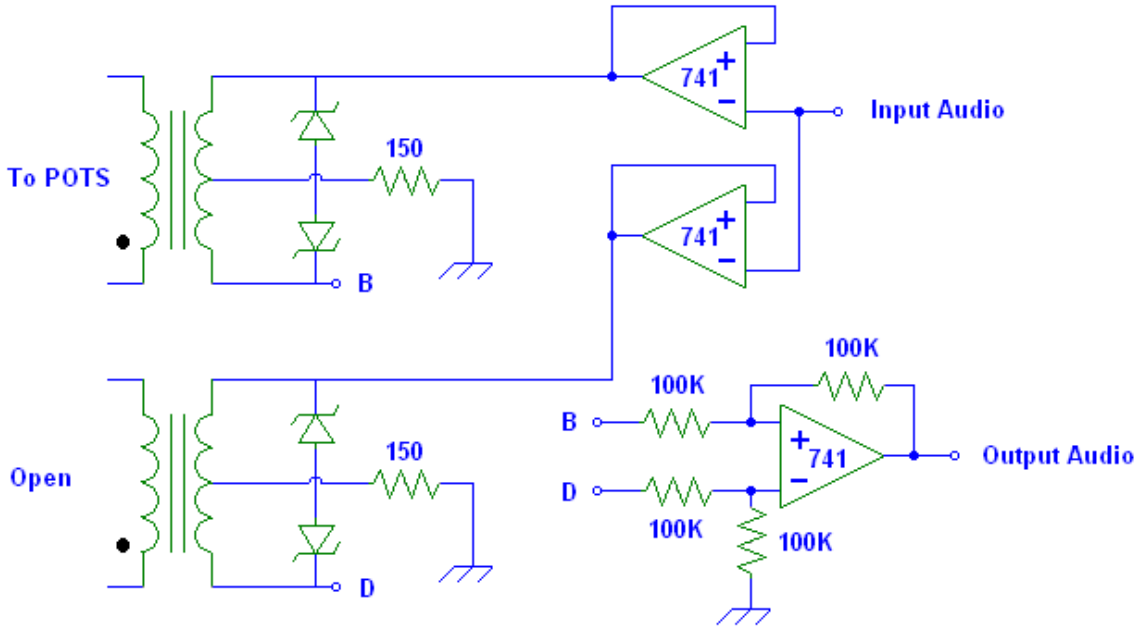


Figure 5-7: Feedback cancellation circuit

The two additional operational amplifiers provide isolation between the two transformers. We note that the output signal will not include the input signal if the impedances on the primary sides of the transformers are identical. This proved to work for when the circuit was on hook, but succumbed to difficulties as our earlier circuit. We conclude that the phone line *cannot* be modeled as a simple current source when of hook. Having exhausted simple analogue solutions, we next consider software-realised remedies.

5.6.2 Software Echo Removal

Ideally, a Digital Signal Processor (DSP) would be employed to cancel "speakerphone" echo, which is the echoing effect produced as a consequence of the microphone and speakers forming a positive feedback loop. While highly desirable, this solution is complex and proved incompatible with our project timeline. A solution based on silence detection was therefore designed instead.

This solution is implemented in the software that interfaces the hardware bridge. It works by muting the audio stream leaving the hardware-bridge computer when incoming audio is detected, thereby preventing incoming audio from being included in the outgoing stream. The silence detection function is described in §4.2.4.

5.7 Hardware Control Application

Hardware Control Application is software that connects the VoIP-POTS bridge to the VoIP network. Its purpose is to instruct the hardware to go on line, to accept connections from authorized users over the VoIP network, and to reject connections from unauthorized users.

The following figure shows how the core of the Hardware Control Application interacts with the other parts of the system.

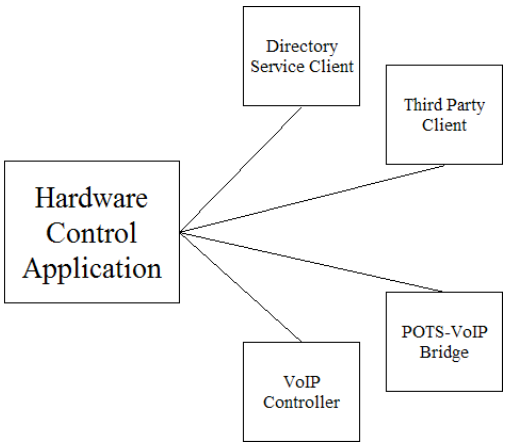


Figure 5-8: Components of the Hardware Control Application

The core of the Hardware Control Application has to interface with the surrounding modules depicted in the figure, namely the Directory Service Client library, the Third Party Client library, the Hardware Interface library and the VoIP controller library.

The purpose of the Hardware Control Application is to function as a State Machine capable of operating in multiple concurrent states. The Directory Service and Third Party Client libraries' design naturally lend themselves to a state machine because both libraries are designed around the principle of sending and receiving messages. Using messages to trigger a state change is ideal because it is just as easy to move to a "connected to peer" state as a "disconnected state".

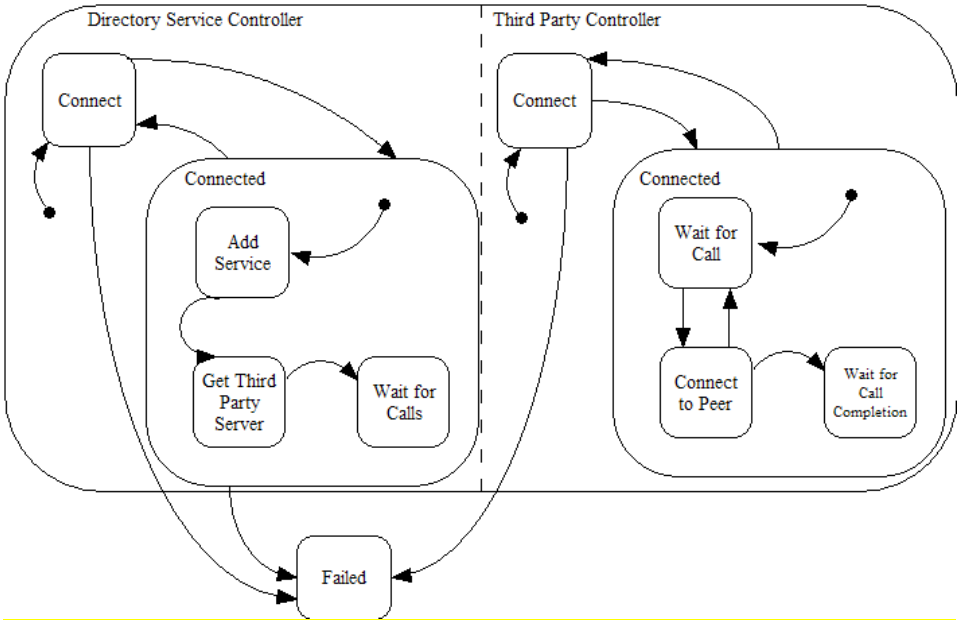


Figure 5-9: State diagram of the Hardware Control Application

5.7.1 Module Communication

The Hardware Control Application is designed to run three separate threads to control the various libraries with which it must interface. These libraries must be synchronized with each other to ensure proper operation. Successful operation of the Hardware Controller Application must initialize the modules individually and have them communicate to each other when necessary.

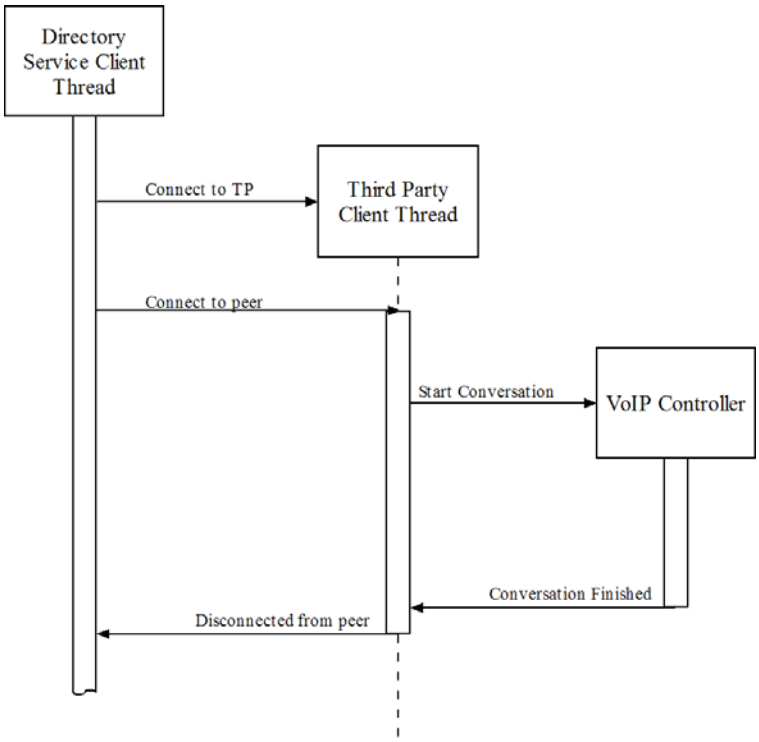


Figure 5-10: Sequence Diagram of the Hardware Control Applications components

Figure 5-10 outlines the communication between the components in the application. The Directory Service Client signals the Third Party Client once the thread has connected, which in turn performs its initialization routine and waits for a call. The Third Party Client thread is again alerted by the Directory Service Client thread when a connection request has been made. The Third Party client then alerts the VoIP controller once a successful connection is made between peers. When the VoIP conversation has finished, the VoIP controller alerts the Third Party Client thread, which alerts the Directory Service Thread.

5.7.2 Directory Service Client Library Controller

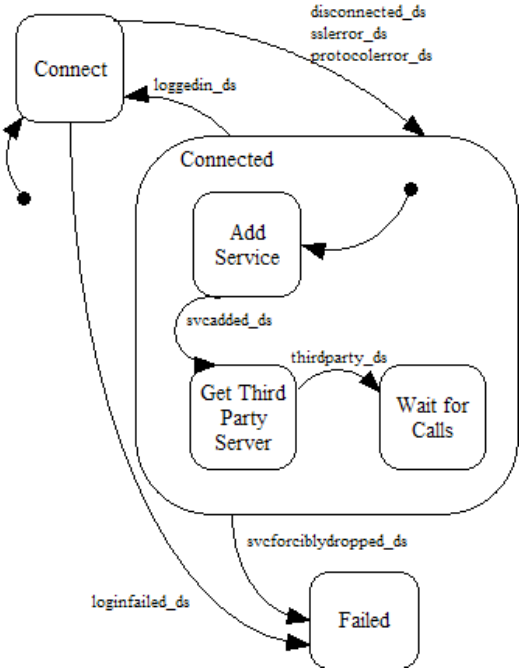


Figure 5-11: State diagram of the Directory Service Client library controller

The Directory Service Client library is controlled by the more complex of the two state machines. Successful Hardware Control Application would undergo the following state changes:

1. **Connect State** - This state attempts to connect to the Directory Service Server. It transitions to the Advertise Service State upon a successful connection

2. **Advertise Service State** - Once connected, the client tells the server which services it is running. This step ensures that the receiver is the only client accepting calls from this user. This would not be a concern for the phone interface, because ideally the account would only be used for the VoIP-POTS bridge.

3. **Obtain a Third Party State** - This state is used to obtain the address of a Third Party Server. This information is used by the Third Party Client library to connect to other peers.

4. **Wait for Call State** - Once entered, only an error will cause the FSM to leave this state. This state waits for the Directory Service to signal that a user wishes to make use of the VoIP-POTS bridge. The application checks its access list, and if the user exists, it starts

the Third Party State Machine to connect directly to the user. A shared variable, *connect*, exists to prevent multiple users from connecting to the Application at once.

Fatal, and non-fatal failures are also possible state changes. Fatal errors are errors that would not be correctable by reconnecting. The fatal errors on the state transitions are:

1. SVCLIMITEXCEEDED_DS: This message occurs when a particular user has advertised too many unique services on a Directory Service Server. This would require the user to disconnect other applications in order to allow the Hardware Control Application to connect. Requiring the user to re-run the application when they have removed other services stops the application from making numerous requests that would go unfulfilled.

2. LOGINFAILED_DS: This message occurs when an invalid username or password is submitted to the Directory Service Server. Attempting to reconnect would only result in the bad username or password to be re-sent.

3. SVCFORCIBLYDROPPED_DS: The service name associated with the VoIP system has been requested by another connection. This indicates that the user the Hardware Control Application is using has run this or the VoIP program elsewhere and wishes to take control of the service.

The non-fatal errors indicate that something has gone wrong during the connection. These are most likely not caused by a mis-configuration but due to some random error, and should reoccur. If errors persist, there is a limit on how many times the Hardware Control Application will attempt to reconnect. This is shown by the 'attempts_left = 0' transition from the "Connect" state to the "Exit" state in Fig. 5-11.

5.7.3 Third Party Client Library Controller

The Third Party Client Library is also interfaced through a state machine. Its state machine, however, is much simpler.

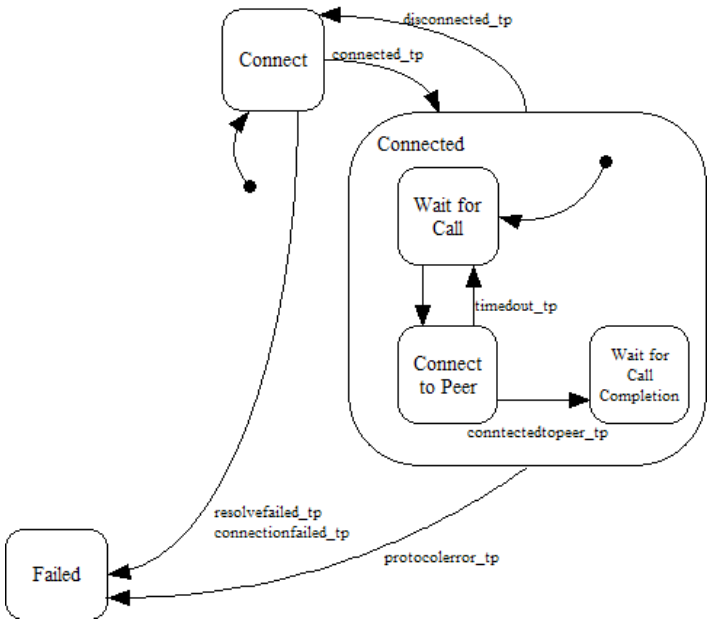


Figure 5-12: State diagram of the Third Party Client library controller

1. The Connect State: The Third Party Client attempts to connect to the Third Party Server. When this is completed, the state machine transitions to the Wait State.

2. The Wait State: The Third Party Client library controller waits here until it is signalled to connect to a peer.

3. The Connected to Peer State: When the Third Party Client has connected to the peer the VoIP Controller is started. The VoIP Controller's Interface is described in the next section.

4. The Fail State: If we are unable to connect to the Third Party, we cannot connect to other peers on the VoIP network, and as such, the program has failed. This state signals the Application to terminate.

If the Third Party Client disconnects from the Third Party Server due to an error, it would reconnect. This also requires stopping the controller, if it is running, as it uses the Third Party's state information for sending data between the peers, as shown in Fig 5-13 below. As you can see, the data is passed to the Third Party Client library to be sent to the peer. When the Third Party Client library becomes disconnected from the Third Party server, this channel becomes invalid.

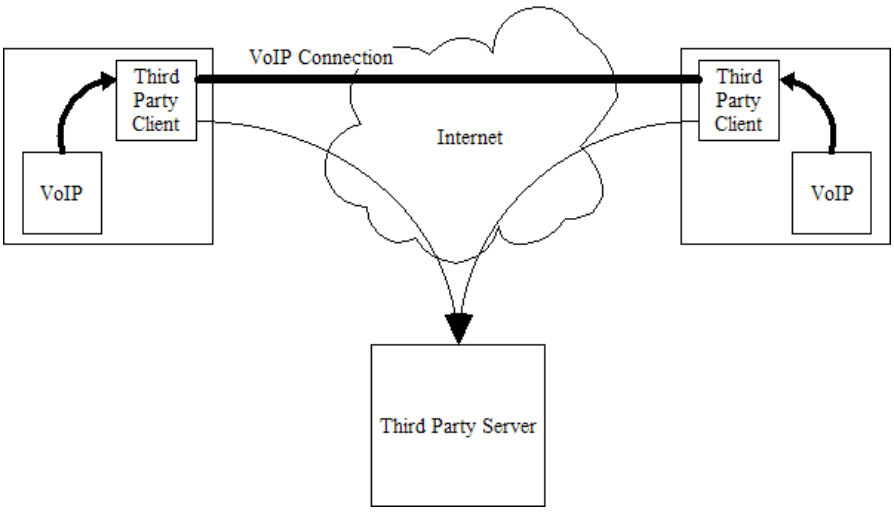


Figure 5-13: The VoIP Controller's communication channels and Third Party Client library

5.7.4 The VoIP Controller

The VoIP Controller is described in more detail in §4.5. It is well abstracted, and as such, it is started and stopped from the Third Party Client library controller, along with the VoIP-POTS Bridge Controller.

5.7.5 The VoIP-POTS Bridge Controller

The VoIP-POTS Bridge controller is described in §5.5. It is well encapsulated, and as such, it is started and stopped from the Third Party Client library controller, along with the VoIP controller.

6. Conclusion

A system has been created that functions in a decentralized peer-to-peer fashion without requiring data transfer over an additional central server. The VoIP software and accompanying GUI can accommodate many firewall and NAT configurations, resulting in its suitability for residential Internet use on home PCs.

The system employs high quality audio, using strong encryption for privacy, and minimizes latency by adjusting to network conditions. Additionally, our software can interface with analogue telephone lines via with the accompanying external hardware. The software and POTS hardware allows users of Internet-connected PCs or wireless connections to completely side-step legacy telcos when making long distance telephones call while

The design could be further enhanced to include:

1. **DSP echo cancellation.** Echo and feedback problems can be corrected in software, but it would be more efficient to use a hardware-based audio processor.

2. **Support for additional Codecs**. Codecs demand varying degrees of processing power and bandwidth. Supporting a wider range of Codecs (such as MPEG, GSM, LCP-10) would lead to a wider range of applications.

3. **Support for additional services**. The current system can be easily extended to include other services such as text, video, and file transfer.

4. **Compliance with industry standards**. Our system could enjoy widespread adoption if it complied with ITU Q.23, Q.24, K.44, K.45, and G.114 [13, 14, 15].

5. **Implementation as an embedded system**. The logical extension of our project would be to integrate the VoIP processing entirely within a telephone set.

Bibliography

1. Axelson, Jan. Parallel Port Central. 2004 <http://www.lvr.com/parport.htm>

2. Berkes, Jem, et al. “NAThack.net”. 10 Dec 2003 <http://www.nathack.net/>

3. Blechman, Fred. Simple, Low Cost Electronics Projects. Eagle Rock: Technology Publishing, 1998.

4. Chuah, Chen-Nee, and Randy H. Katz. "Network Provisioning & Resource Management for IP Telephony." 09 Mar 2004 <http://www.ece.ucdavis.edu/~chuah/paper/csd-99-1061.pdf>

5. Engdahl, Tomi. “Telephone line audio interfacing circuits”. 2001 <http://www.hut.fi/Misc/Electronics/circuits/teleinterface.html>

6. Fairchild Semiconductor Corporation. 2N3904/MMBT3904/PZT3904 NPN General Purpose Amplifier. Rev. A, 2001.

7. Flickenger, Robert. "Performance Test: 802.11b Takes a Lickin' and Keeps on Tickin' ." 09 Mar 2004. <http://www.oreillynet.com/pub/a/wireless/2001/03/29/microwave.html>

8. Gatti , Alessandro . "[openh323] Silence detection algorithm ." 25 Nov 1999. 2 Feb 2004. <http://www.openh323.org/pipermail/openh323/1999-November/002396.html>.

9. "GSM 06.10 lossy speech compression ." 1 Jul 2000. 24 Oct 2003. <http://kbs.cs.tu-berlin.de/~jutta/toast.html>.

10. "H.323 Protocols Suite ." How Stuff Works. 25 Oct 2003. <http://computer.howstuffworks.com/framed.htm?parent=ip-telephony.htm&url=http://protocols.com/pbook/h323.htm>.

11. "HawkVoice - Free Speech Compression." 13 Jan 2004. Hawk Software. 20 Oct 2003. <http://www.hawksoft.com/hawkvoice/>.

12. Hemming, Ben . "A comparison of Internet audio compression formats." 31 Dec 2003. Serious Cybernetics. 5 Nov 2003. <http://www.sericyb.com.au/audio.html#gsm>.

13. International Telecommunications Union. ITU-T G.114 One-way transmission time. Geneva, 2003.

14. International Telecommunications Union. ITU-T K.44 Resistibility tests for telecommunication equipment exposed to overvoltages and overcurrents – Basic recommendation. ITU, 2003.

15. International Telecommunications Union. ITU-T Q.24 Multifrequency Push-Button Signal Reception. ITU, 1993.

16. Isocom Components. H11A1, H11A2, H11A3, H11A4, H11A5 Optically Coupled Isolator Phototransistor Output. DB91041m-AAS/A1, July 2000.

17. Kominek, Jay. Qial. 01 Dec 2001. 23 February 2004 <http://www.miranda.org/~jkominek/qial>.

18. Kurien, Toby. "Audio Effects Algorithms." 6 Mar 1997. 19 Jan 2004. <http://users.iafrica.com/k/ku/kurient/dsp/algorithms.html>.

19. Langton, Charan . "Coding and decoding with Convolutional Codes." 31 Jul 1999. 26 Oct 2003. <http://www.complextoreal.com/convo.htm>.

20. Maxim. "An Introduction to Jitter in Communications System". Dallas Semiconductor. 6 Mar 2003. < http://www.maxim-ic.com/appnotes.cfm/appnote_number/1916/ln/en>

21. "MELP at 2.4Kbps." 9 Jan 2002. ARCON. 24 Oct 2003. <http://maya.arcon.com/ddvpc/melp.htm>.

22. Mims, Forrest M. III. Engineer's Mini Notebook: IC Op Amp Projects. 2nd ed.: Siliconcepts, 1999.

23. "MSDN Home Page". 29 Dec 2003. Microsoft Corporation. <http://msdn.microsoft.com/>

24. National Semiconductor Corporation. LM139/LM239/LM339/LM2901/LM3302 Low Power Low Offset Voltage Quad Comparators. DS005706, August 2000.

25. National Semiconductor Corporation. LM741 Operational Amplifier. DS009341, August 2000.

26. Nelson, Mark. "Data Compression Info - Speech." 31 Dec 2002. 24 Oct 2003. <http://www.datacompression.info/Speech.shtml>.

27. Popiel, Jerry, and George Bock. The New KISS Book. 2nd ed. Winnipeg: Manitoba Telecom Services, 1999.

28. Postel, Jon. "RFC 768: User Datagram Protocol". Aug 1980. Information Sciences Institute. 05 Dec 2003. <ftp://ftp.rfc-editor.org/in-notes/rfc768.txt>, August 1980.

29. Rane Corporation. "Interfacing Audio and POTS". 2002 <http://www.rane.com/note150.html>

30. "Real-Time Protocol-based Audio Engine: Audio Codec Module." 29 Dec 2003. <http://home.elka.pw.edu.pl/~mroj/homepage/works/mroj/html/audio/waveform-audio.htm>.

31. "RFC 791 - Internet Protocol." Sep 1981. Information Sciences Institute. 01 Dec 2003. <http://www.ietf.org/rfc/rfc0791.txt>.

32. Sedra, Adel S., and K. C. Smith. Microelectronic Circuits. 4th ed.: Oxford University Press, 1997.

33. Skybuck Flying. "raw socket code: c version works, delphi version does not work ?" 11 Oct 2002. Online posting. Available USENET: alt.winsock.programming. 02 Dec 2003.

34. "Skype." 31 Dec 2003. Skyper Limited. 2 Feb 2004. <http://www.skype.com/skype_p2pexplained.html>.

35. "Speech Coding." University of Southampton. 24 Oct 2003. <http://rice.ecs.soton.ac.uk/jason/speech_codecs/index.html>.

36. Steer , William. "Wave I/O in Windows." 8 Aug 2002. Techmind. 1 Jan 2004. <http://www.techmind.org/wave/>.

37. "The Team Speak Engine." 31 Dec 2000. Team Speak. 2 Feb 2004. <http://www.teamspeak.org/modules.php?op=modload&name=About&file=index>.

38. Walker, John. "Speak Freely." 15 Jan 2004. 24 Oct 2003. <http://www.fourmilab.ch/speakfree/windows/doc/compress.html>.

39. Wang, Charles. "Forward Error-Correction Coding." 28 Mar 2002. The Aerospace Corporation. 26 Oct 2003. <http://www.aero.org/publications/crosslink/winter2002/04.html>.

Appendix A. Final VoIP Software

Please see CD-ROM for the complete software application.

Appendix B. Software Modules

Please see CD-ROM for individual software modules (with full source code).

Appendix C. Raw Test Output

Please see CD-ROM for log results from network tests.

Appendix D. Software and Tools Used

CoolEdit 96
Syntrillium (no longer exists)

CoolEdit provides basic but powerful audio editing and filtering functionality. We used CoolEdit to examine and manipulate audio waveforms, and generate DTMF tones.

Ethereal
Gerald Combs and contributors
http://www.ethereal.com/

Ethereal is a low-level packet analyzer that let us examine packets as constructed by our software. Uses libpcap to capture packets.

GNU Compiler Collection (gcc)
http://gcc.gnu.org/

We used the GNU C compiler to build much of our software for cross-platform testing. gcc is also used within MinGW.

Linux 2.4 and 2.6
Linus Torvalds and contributors
http://www.kernel.org/

Much of our software was developed on systems running the Linux kernel, versions 2.4 and 2.6. The Directory Service uses new functionality of the 2.6-series kernel.

Microsoft Visual Studio and MSDN
Microsoft Corporation
http://www.microsoft.com/

We used Visual C++ and Visual Basic, along with the MSDN reference material, in order to build our graphical user interface. Visual C++ was also used to compile many of our Dynamic Link Libraries (DLLs).

Microsoft Windows NT
Microsoft Corporation
http://www.microsoft.com/

Our final VoIP application was developed on the Windows NT platform, and relies on the Windows API for facilities such as audio I/O.

Minimalist GNU For Windows (MinGW)
http://www.mingw.org/

MinGW provides software that allows UNIX-like software to compile for Windows

OpenSSL
Eric A. Young, Tim J. Hudson, and contributors
http://www.openssl.org/

The OpenSSL library provides a Secure Sockets Layer (SSL) and Transport Layer Security (TLS) used in some of our software.

tcpdump/libpcap
The Tcpdump Group
http://www.tcpdump.org/

tcpdump is a command-line program that displays packets captured by libpcap. We used tcpdump to analyze UDP packets under Linux.

Third Party and Directory Service
http://www.nathack.net/
Berkes, Czyrnyj, Kaye, Schaub
(Advisor: Dr. R. D. McLeod, University of Manitoba)

This software provided the facilities to establish direct connections through firewalls and NAT. Specifically, our VoIP software relies on the 'Third Party' and 'Directory Service' modules.

WinIO
Yariv Kaplan
http://www.internals.com/

Used with written permission from the author
WinIO provides Windows drivers and a library that allow low-level hardware I/O. This provides the access to the parallel port, which is used to digitally interface our software to our hardware device.

VITAE

NAME: Jem E. Berkes

PLACE OF BIRTH: St. Catharines, Ontario

YEAR OF BIRTH: 1982

SECONDARY EDUCATION: St. John's-Ravenscourt School (1996-2000)

HONOUR and AWARDS: Dean's Honour List 2002, 2003, 2004

NSERC University Undergraduate Research Award

Canada Millennium Scholarship Foundation Excellence Award

Advanced Placement and International Baccalaureate Scholarship Enhancement

Engineering Academic Excellence Award – Computer Engineering

APEGM Engineering Entrance Scholarship

Dr. Kwan Chi Kao Scholarship in Electrical and Computer Engineering

Faculty of Engineering Second Year Scholarship

UMSU Scholarship

Isbister Scholarship in Engineering

EECOL Electric Inc. Scholarship

Technical Communication Report Prize – Engineering

Donald George Lougheed Memorial Scholarship

Grettir Eggertson Memorial Scholarship

NAME: Timothy Eugene Czyrnyj

PLACE OF BIRTH: Winnipeg

YEAR OF BIRTH: 1982

SECONDARY EDUCATION: Oak Park High School (1997-2000)

HONOUR and AWARDS: Hogg Centennial Entrance Scholarship

Advanced Placement and International Baccalaureate Scholarship Enhancement

Canadian National Undergraduate Scholarship

NSERC University Undergraduate Research Award

Dean's Honour List 2002, 2003, 2004

NAME: Justin David Olivier

PLACE OF BIRTH: Winnipeg

YEAR OF BIRTH: 1982

SECONDARY EDUCATION: River East Collegiate (1997-2000)

HONOUR and AWARDS: University of Manitoba Entrance Scholarship

Dean's Honour List 2002, 2003, 2004

NSERC Undergraduate Award

NAME: Dominic Etienne Schaub

PLACE OF BIRTH: Winnipeg

YEAR OF BIRTH: 1982

SECONDARY EDUCATION: St. John's-Ravenscourt School (1996-2000)

HONOUR and AWARDS: Dean's Honour List 2002, 2003, 2004

University of Manitoba Entrance Scholarship

Advanced Placement and International Baccalaureate Scholarship Enhancement

Technical Communication Report Prize – Engineering

NSERC Undergraduate Award

Easton I. Lexier Award for Community Leadership