TABLE OF CONTENTS

1.0 Introduction

bat-cheet which means light talk in Hindi, is a Voice Communication Tool. It provides packet-voice communications across a Local Area Network (LAN). It operates currently in the unicast (point - to - point) mode using the UDP transport protocol. The current implementation works on SGI platform.

This report describes the design and implementation of bat-cheet which was implemented as a class project in CSE791, Advanced Programming in UNIX.

Section 2.0 gives an overview of the IRIS Digital Audio Library in IRIX 5.3 and describes the interaction of the application with it. Section 3.0 gives an overview of the codec used (Intel/DVI) and its characteristics. Section 4.0 describes the design and implementation of bat-cheet. Section 5.0 gives the results obtained. Section 6.0 enunciates on future work and enhancements to the voice communication tool while section 7.0 gives the references.

2.0 IRIS Digital Audio Library in IRIX 5.3

The Audio Library (AL) in SGI provides an API for audio input to and audio output from Silicon Graphics workstations that feature high-quality digital audio systems. It configures the audio system, manages audio I/O between the application program and the audio hardware, specifies attributes of digital audio data, and facilitates real-time programming. Fig. 1 diagrams the interaction between an audio application and the audio library, the device driver and the audio hardware.

The AL comprises routines that provide these basic capabilities:

- creating digital audio input and output connections
- reading and writing digital audio data
- querying and controlling digital audio data attributes
- querying and controlling the configuration of the audio system
- handling errors

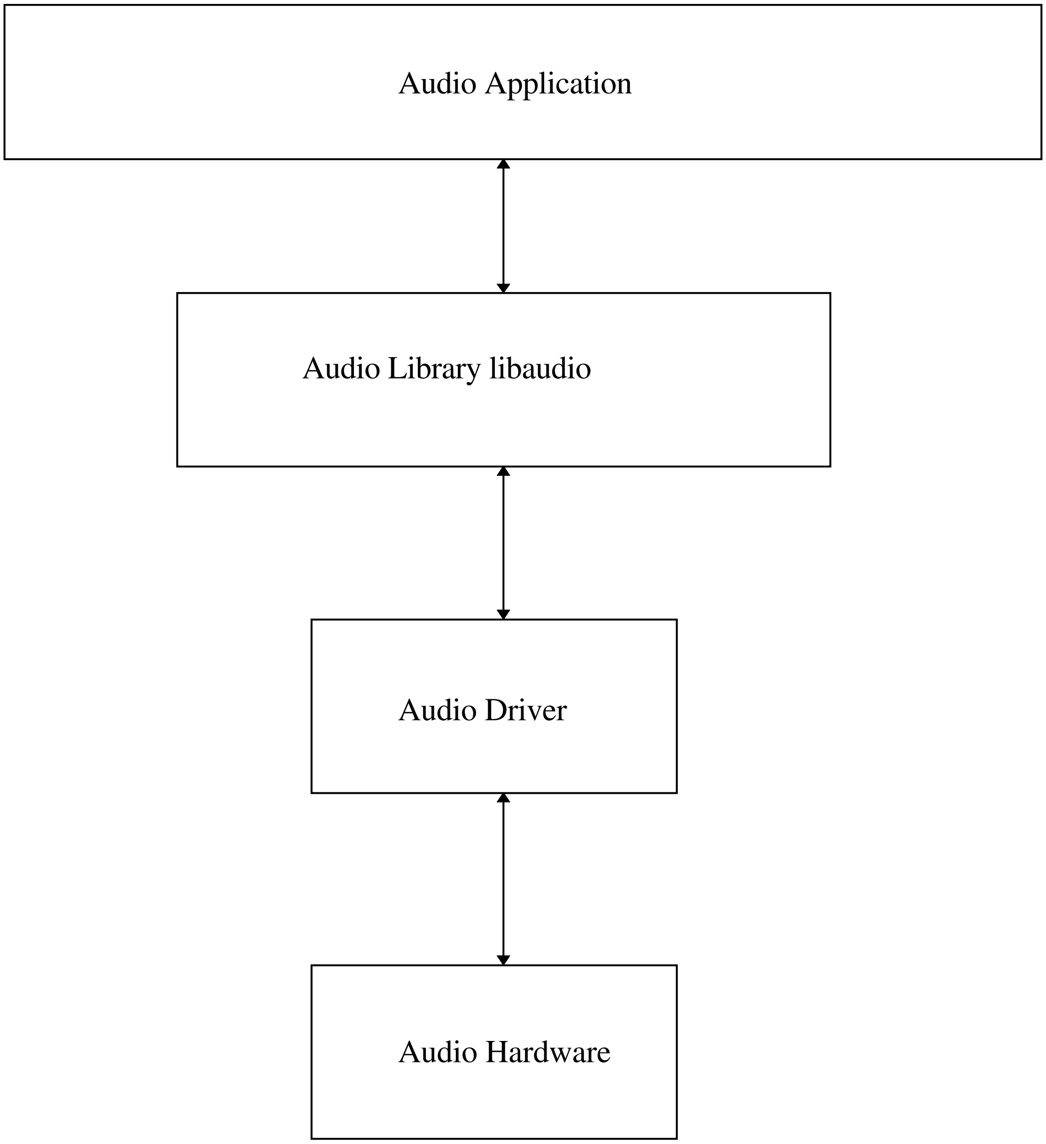

Fig 1. Interaction of Digital Audio System Components

2.1 Audio Library Programming Model

The AL programming model gas three basic objects:

- Audio Device: The audio hardware used by AL, which is shared among audio applications.

- ALport: A one-way (input or output) audio data connection between an application program and the host audio system. An ALport contains:
    - an audio sample queue, which stores audio samples awaiting input or output
    - settings pertaining to the attributes of the digital audio data it transports
- ALconfig: An opaque data structure for configuring these settings of an ALport:
    - audio device (static setting)
    - size of the audio sample queue
    - number of channels (static setting)
    - format of sample data (dynamic setting)
    - width of the sample data (dynamic setting)
    - range of floating point sample data (dynamic setting)

Digital Audio Data Representation

The digital representation of an audio signal is generated by periodically sampling the amplitude(voltage) of an audio signal. The samples represent periodic "snapshots" of the signal amplitude. Digital audio information is sampled at a frequency that is at least double the highest interesting analog audio frequency (Nyquist Theorem).

Digital Audio Sample Frames

A sample frame is a set of audio samples that are coincident in time. A sample frame for mono data is a single sample (this is used in bat-cheet). A sample frame for stereo data consists of a left-right pair. Fig 2 shows the relationship between the number of channels and the frame size of audio sample data.

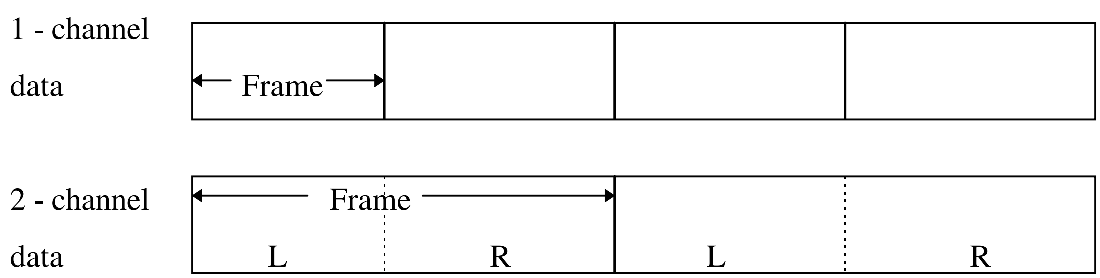

Fig 2. Audio samples and frames

Digital Audio Sample Formats:

The AL uses PCM to represent digital audio samples. The formats supported by the AL and the audio system are:

- 8-bit and 16-bit signed integer
- 24-bit signed, right justified within a 32-bit integer
- 32-bit and 64-bit floating point

For this application 16-bit signed integer format was chosen.

Digital Audio Input and Output Sample Resolutions:

The native data format used by the audio hardware is 24-bit two's complement integers. The audio hardware sign-extends each 24-bit quantity into a 32-bit word before delivering the samples to the AL.

Microphone input samples come from A/D converters, which have 16-bit resolution. These samples are treated as 24-bit samples with 0s in the lower 8 bits.

For audio output, the AL delivers samples to the audio hardware as 24-bit quantities sign-extended to fill 32-bit words. The actual resolution of the samples from a given output port depends on the application program connected to the port. In bat-cheet, a 16-bit output port has been used.

2.2 Audio Library Application Programming Concepts

Typically an AL program must:

- initialize data structures
- set up buffers for passing data between application and CPU
- query for available features
- configure and open audio connections
- pass data to and from the ALport and operate on the data
- process errors
- close audio connections

- free system resources

The following explains some of the concepts:

Initializing an Audio Library Application

To enable audio input and output, the application must create and configure the required audio I/O connections. The AL provides an opaque data structure called an ALport for audio I/O connections. An ALport provides a one-way (input or output) mono, stereo, or 4-channel audio data connection between an application program and the host audio system. An ALport consists of a sample queue and static and dynamic state information. For audio input, the hardware places audio samples in an input port's queue at constant rate, and the application program reads the samples from the queue.. Similarly, for audio output, the application writes audio samples to an output port's queue, and the audio hardware removes the samples from the queue.

An ALport can be opened with the default configurations or an ALconfig can be customized for configuring an ALport. The default ALconfig has:

- a queue size of 100000 samples
- stereo data
- a two's complement sample format
- a 16-bit sample width

Steps involved in configuring and opening an ALport are listed below:

1. turn off the default error handler ALseterrrorhandler().
2. if default settings are acceptable call ALopenport() routine, else create a new ALconfig by calling ALnewconfig( ).
3. If non default values are needed for any of the ALconfig settings, set the desired values as follows:
    - call ALsetchannels( ) to change the number of channels
    - call ALsetqueuesize( ) to change the sample queue size
    - call ALsetsampfmt( ) to change the sample format

- call ALsetwidth( ) to change the sample data width

4. Open an ALport by passing the ALconfig to the ALopenport ( ) routine.
5. Create additional ALports with the same settings by using the same ALconfig to open as many ports as desired.

Some Common AL Routines Used in the Program

- creating a new ALconfig:

  ALconfig ALnewconfig(void)

- Setting and getting the number of channels for an ALconfig

  int ALsetchannels (ALconfig config, long channels)

  where channels : 1, 2, or 4

  long ALgetchannels (ALconfig config)

- Setting and getting the sample queue size for an ALconfig

  int ALsetqueuesize (ALconfig config, const long size)

  long ALgetqueuesize (ALconfig config)

- Setting and getting the sample data format for an ALconfig

  The AL allows you to choose between three sample formats:

  - two's complement (default)
  - floating point
  - double-precision floating point

  int ALsetsampfmt (ALconfig config, long sampleformat)

  long ALgetsampfmt (ALconfig config)

- Setting and getting the integer sample width for an ALconfig

The sample width represents the degree of precision to which the full-scale range of an audio signal can be sampled. You can only specify the width of two's complement integer sample data.

The following sample widths are available:

- 8-bit samples : 28 quantized signal values, signed chars
- 16-bit samples : 216 quantized values, shorts
- 24-bit samples : 224 quantized signal values, longs

int ALsetwidth (Alconfig config, long samplesize)

long ALgetwidth (ALconfig config)

- Freeing Resources Associated with an ALconfig

  If the application is not going to open any more ports with an ALconfig, it is then no longer needed and as such must be freed to minimize memory consumption.

  int ALfreeconfig (ALconfig config)

- Opening and Closing Audio Ports

  ALopenport opens an ALport while ALcloseport closes it.

  ALport ALopenport (char *name, char *direction, ALconfig config)

  where

  name is an ASCII string used to identify the port to humans. This name is limited to 20 characters

  direction "r" configures port for reading (input)

  "w" configures port for writing (output)

  int ALcloseport (ALport port)

- Monitoring the audio sample queue status to provide nonblocking I/O

  The AL maintains the following status information about the queue:

filled the number of queue locations containing valid data

fillable the number of empty locations in the queue

filled + fillable = queuesize

To get filled:

long ALgetfilled (ALport port)

and to get fillable:

long ALgetfillable (ALport port)

- Reading and writing samples from and to queue

Audio input is accomplished by reading audio data samples from an input ALport's sample queue and audio output is accomplished by writing audio data samples to an output ALport's sample queue.

int ALreadsamps (const ALport port, void * samples, const long samplecount)

where

samples is a pointer to a buffer into which the samples read from the input are transferred

samplecount is the number of samples to read

int ALwritesamps (ALport port, void * samples, samplecount)

where

samples is a pointer to a buffer from which the samplecount samples are transferred to the audio port

3.0 Encoding and Decoding of Speech Signal

We will discuss waveform coding here. Waveform codecs attempt to reproduce the input signal's waveform. They are generally designed to be signal independent so as to be able to encode a wide variety of signals. They also exhibit a graceful degradation in the presence of noise and transmission errors. However they are most effective at medium bit rates. Waveform coding can be carried out in either the time or frequency domain. We will elaborate on time domain coding only.

Time Domain Coding

Pulse Code Modulation (PCM)

It is the simplest type of waveform coding. As each sample enters the coder it is quantized to one of a finite set of reconstruction levels, each level is assigned a unique sequence of binary digits and it is this sequence that is transmitted. Logarithmic quantization is the one most frequently used.

Differential Pulse Code Modulation (DPCM)

When coding speech there is a high correlation between adjacent samples. This correlation can be used to reduce the resulting bit rate. One method is to transmit only the differences between each sample. This difference signal will have a much smaller dynamic range as compared to the original speech and as such can be effectively quantized using a quantizer with fewer reconstruction levels. In this method the previous sample is used to predict the value of the present sample. This technique is called differential pulse code modulation.

Adaptive Differential Pulse Code Modulation

With DPCM both the predictor and the quantizer remain fixed in time. Greater efficiency could be achieved if the quantizer adapted to the changing statistics of the prediction signal. Further gains could be made if the predictor itself, could be adapt to the speech signal. This would ensure that the mean squared prediction error was being continually minimized independently of the speaker and the speech signal.

There are two methods to do this - feedforward and feedbackword adaption. With feedforward adaption the reconstruction levels and the prediction coefficients are calculated at the transmitter, using a block of speech. They are then, themselves, quantized and transmitted to the receiver as side information. Both the transmitter and the receiver use these quantized values to make the predictions and quantize the residual. For feedbackword adaption the reconstruction levels and predictor coefficients are calculated using the coded signal. Since this signal is known to both the transmitter and the receiver there is no need to transmit any side information, so the predictor and quantizer can be updated for every sample. This latter adaption technique is more susceptible to transmission errors than the former though it can produce lower bit rates.

For the above mentioned reasons and advantages of adpcm technique over others, it was decided to use an ADPCM codec. There are quite a few standards which use ADPCM, e,g. ITU G.721 and Intel/DVI ADPCM. It was decided to use Intel/DVI codec over G.721 as the former is less computationally intensive and produces reasonable quality for voice signals. The codec was obtained from Stichting Mathematisch Centrum, Amsterdam, Netherlands which used the algorithm proposed by IMA.

Intel/DVI ADPCM CODEC

The IMA Reference Algorithms for ADPCM compression and decompression are an implementation of adaptive quantization with fixed prediction. The specific implementation of the quantizer adaptation using table-based lookup was offered by Intel/DVI® as an open standard for use by the IMA.

This algorithm encodes only the difference between consecutive samples, allowing a wide dynamic range to be maintained with a minimum data bandwidth.

Compression Algorithm

The compression algorithm assumes the original sample is a 16-bit two's complement variable. The new sample is the resulting 4-bit ADPCM sample.

The algorithm finds the difference between the original sample and the predicted sample, which is the output of its predictor. This difference is then quantized down to a 4-bit sample, using stepsize. The new 4-bit sample has a sign-magnitude format. After the new sample has been calculated, it is uncompressed using the same quantization step size to obtain a linear difference identical to that calculated by the decompressor. In order to correct for truncation errors in the quantization, ½ is effectively added to the new sample during the expansion. This difference is added to the predicted sample to form a prediction for the next sequential (original) Sample. The new sample is used to adjust an index into the step size table.

This index points to a new step size in the step size table. The predicted sample, step size, and index must be static variables between samples.

Decompression Algorithm

The decompression algorithm assumes the original sample is a 4-bit ADPCM sample. The new sample is the resulting 16-bit two's complement variable.

The original sample is uncompressed using a quantization step size to obtain a linear difference. In order to correct for truncation errors in the quantization, ½ is effectively added to the original sample during the expansion. This difference is added to the predicted sample to form a linear new sample. The original sample is used to adjust an index into the step size table. This index points to a new step size in the step size table.
The new sample, step size, and index must be static variables between samples.

4.0 Design and Implementation

The design of the voice communication tool is quite straightforward. Fig 3 summarizes the essential components in the design.

4.1 Audio Input Path:

The audio input is taken from the microphone and is deposited by the audio driver into the audio port's input queue (refer Section 2.0). From there the AL reads block of data and deposits it into the user designated buffer. To ensure real-time operation this buffer is implemented as a ring buffer. Its size was chosen (for programming convenience, it doesn't affect the real-time operation) to be an integral multiple of the blocksize. The blocksize itself was chosen to represent 10 ms worth of audio (In literature 10 ms of audio sampling is frequently sited as the optimum size to maximize I/O efficiency). The blocksize is then calculated as:

blocksize = (input sampling rate (samples/sec) x 10) / 1000.

The input sampling rate was chosen as 8Khz. This is provides quite good quality for speech transmission. As such the blocksize works out to be 80 samples.

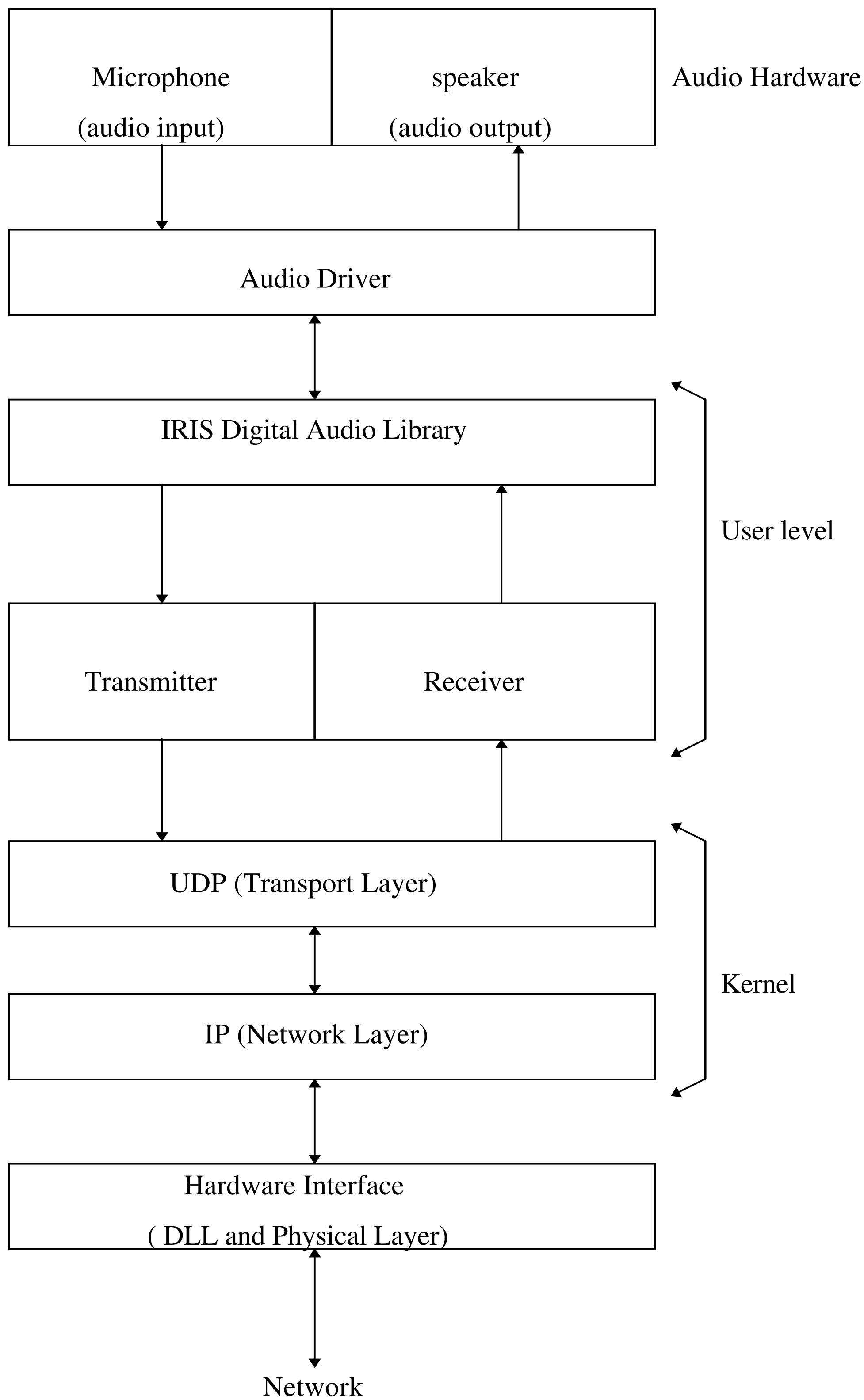

Fig. 3 Overall view of the application in the scheme of things

The transmitter has 3 processes running concurrently to provide real time support. These are readAudio, encode and transmit. These processes are created using the sproc( ) (IRIX) system call which is a variant of the standard fork ( ) system call. The difference is that the child process is a clone of the parent rather than a copy and shares the virtual address space of the parent. Though both parent and child have their respective program counter and stack pointer, text and data are visible to both. This is especially useful in the current scenario.

readAudio opens and configures an ALport through the AL library. The default settings are changed to a queuesize of 12000 samples, mono channel, two 's complement linear PCM format, and 16-bit sample width.

readAudio deposits blocksize of audio input into a ring buffer - audioInBuf using Audio Library routine ALreadsamps( ). encode reads blocksize of data from this buffer, encodes it and writes the encoded data (blocksize/2) into another ring buffer - encodeOutBuf. transmit reads blocksize/2 worth of data from this ring buffer and transmits it onto the network.

Fig. 4 shows the above mechanism.

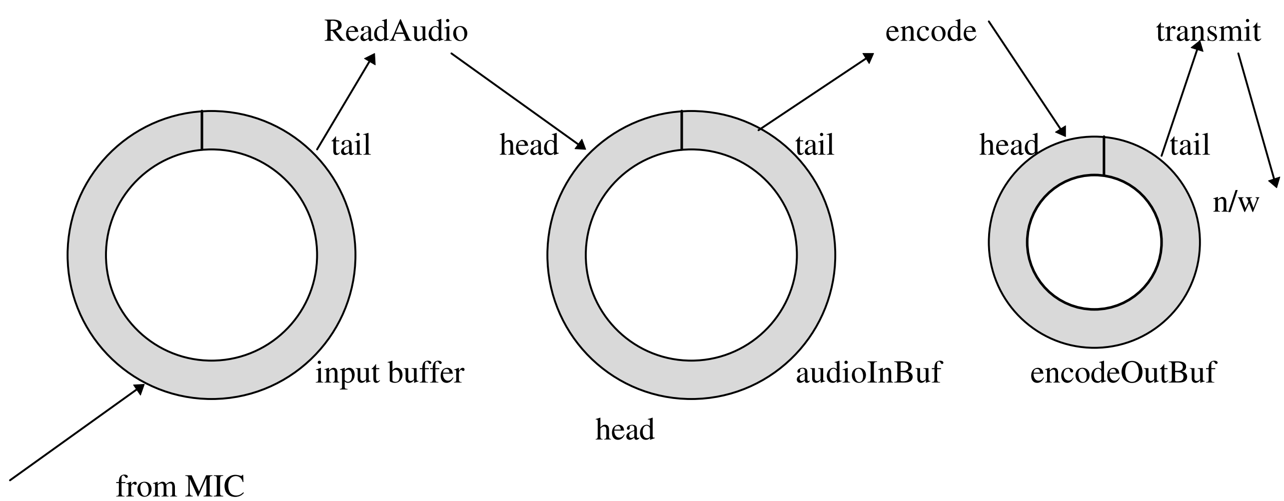

Fig. 4. The transmit path for audio data

The discontinuities in the ring buffers signifies that they 'wrap' around. The input and the audioInBuf buffers are of the same size (integral multiple of blocksize) while the encodeOutBuf is half the their size. Further the audioInbuf is of type short (audio input is sampled in this form) while the encodeOutBuf is of type char (this is the output of encode). Data in all ring buffers is inserted at the head and removed from the tail.

The transport protocol chosen was UDP due to two reasons:

1. TCP is a reliable protocol and as such provides for time-out and retransmission of erroneous and/or missing packets. For real time applications, such as this one, this can be quite disadvantageous. As such TCP is NOT suitable for real-time multimedia applications.
2. It was originally envisaged to use Real-Time Transport Protocol (RTP) with UDP/IP. Though RTP doesn't provide reliability, it does provide the mechanism by which reliability can be ensured by the application layer.

   Further RTP provides the framework to build multicast applications. As such this application could have been extended to provide (multicast) audio conferencing. However due to time constraints it became virtually impossible to provide RTP encapsulated audio data but to provide for future enhancements it was decided to use UDP instead of TCP.

There is no need for synchronization primitives or need for mutual exclusion while reading/writing data from/to the ring buffers. This is so because each process acts as a producer for one ring buffer while it acts as the consumer for the other. As an example, encode reads from audioInBuf but writes to encodeOutBuf. Before reading however it checks whether a block of data (size: blocksize) is available to be read from audioInBuf and before writing it checks whether there is space to write a block of data (size; blocksize/2) on encodeOutBuf.s

4.2 Audio Receive Path:

The audio receive path is symmetrical to the audio input path. The Receiver module receives the audio (encoded) data from the UDP layer and after decoding the audio stream delivers it to Audio Library to be output to the speaker. Like the Transmitter module the Receiver module consists of three concurrent processes - receive, decode and writeAudio. The receive process actually receives the audio data from the UDP layer and writes a block (size: blocksize/2) of it into the decodeInBuf ring buffer while the decode process reads from decodeInBuf, decodes the audio data and writes into audioOutBuf a block of decoded audio data (size: blocksize).The writeAudio process first opens an ALport for output and changes the default settings throught the AL library. The queuesize is set to 12000 samples, number of channels to 1, the sample format to two's complement linear PCM format and the sample width to 16-bit. The writeAudio process reads from the audioOutBuf and delivers it to the output (ring) buffer using the Audio Library routine ALwritesamps ( ). Of course, like in the transmit path, the processes before reading/writing verify the existence of valid data/empty space in the respective ring buffers. Fig. 5 summarizes the above operations:

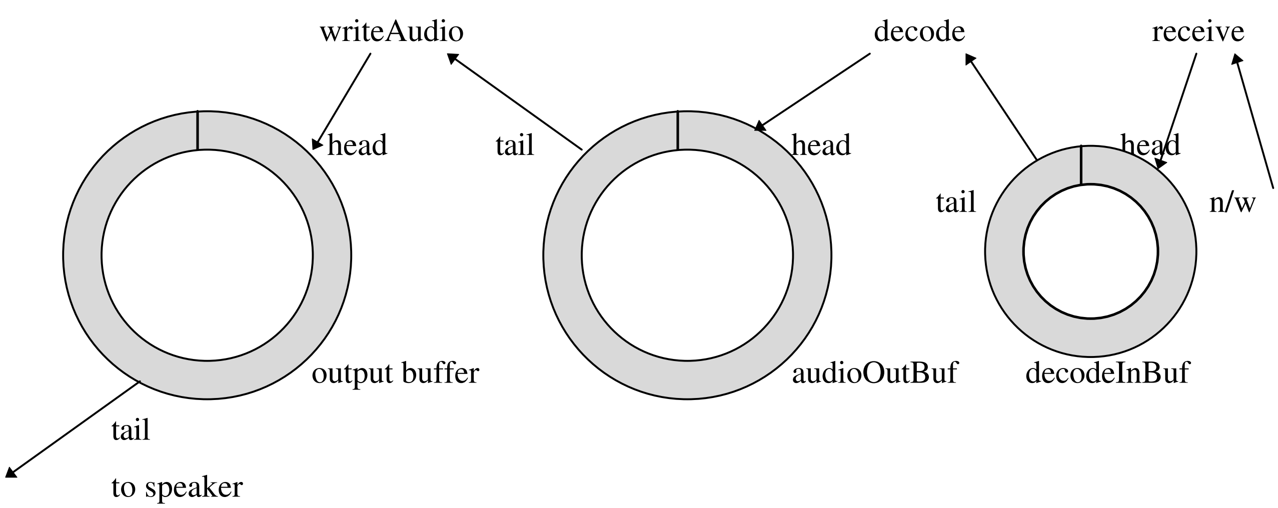

Fig. 5 The receive path for audio data

The ouput rate was chosen to be same as the input sampling rate of 8 Khz in order to provide real-time continuity. Also it may be noticed that audioOutBuf and output (sample) queue are of the same size and decodeInBuf is half their size in terms of samples. Further the output sample queue and the audioOutBuf buffer are of type short whereas the decodeInBuf is of type char.

4.3 Connection Setup and Call Management:

There are two modes of operation - the talk mode and the listen mode. The application can start in either mode. When in talk mode the remote address (the user with whom voice communication is desired) must be specified and when in listen mode the user may optionally specify the remote address . If this is so then the application will refuse connection request from all but the machine whose address was specified. If no address is specified the application will accept connection from any host.

Currently the user address has not been implemented, rather the application responds to machine addresses only.

To initiate connection with a remote host, the local host sends a UDP datagram containing the message “REQUEST”. The remote host then replies depending on whether the optional address was specified. If yes and if the addresses match then it sends back the message “ACCEPTED” else it sends the message “REFUSED”. If the optional address wasn’t specified it sends back the message “ACCEPTED”. The local host on receiving the confirmation message proceeds with the application and the users on the two hosts are then able to communicate. If a “REFUSED” message is received by the local host, the application exits.

While a conversation is going on, all requests for talk are refused by the hosts. To end a conversation the users on both ends should terminate their application by entering Ctrl+C. At present a means for termination and informing the user on the other end has not been implemented. However this is not a critical limitation as the users can agree when to terminate the connection.

As mentioned earlier reliability mechanisms have not been implemented in the current version and as such if either of these are lost or are erroneous the application is bound to fail. However the operation of the application over a LAN somewhat assuages this apprehension.

5.0 Results

The application was tested in the NPAC subnet and its performance was found quite acceptable. It was felt that the audio quality due to Intel/DVI ADPCM codec was better than Inperson, SGI's proprietary software for video and audio communication.

However there was an appreciable delay (1 - 2 s) in playout time. This can be reduced by configuring the codec to read and write directly from the ring buffers instead of a 'linear' buffer as of now.

6.0 Future Work

- encapsulation of encoded audio packets into RTP
- extending the software to provide multicast and hence audio conferencing
- intelligent management of call setup and termination

7.0 References:

1. IRIS Digital Media Programming Guide, Chapters 5, 6
2. UNIX Network Programming, W. Richard Stevens
3. Internetworking with TCP/IP, Volume III, Client-Server Programming and Applications, Douglas E. Comer, David L. Stevens
4. Guide to NEVOT 3.28, Henning Schulzrine, GMD Fokus, Berlin