TABLE OF CONTENTS

Contents **Page**

1.0 Introduction

slideshow is an Audio-On-Demand client which is capable of decoding and playing audio streams with in-lined URLs. These URLs (if any) are added to the audio stream at the time of encoding and the audio client detects the presence (or absence) of these. The URLs are added along with the time (which is calculated as the file offset relative to start of the audio file - the audio file refers to the un-encoded audio file) at which they are to be opened while the audio stream is being played at the client site. If URLs are present and if Netscape is not running on the client site, the client spawns a process to launch Netscape. The child process then remotely controls Netscape and opens the URLs as and when appropriate.

The server used is the VOD (Video-On-Demand) Server cheetah, designed and implemented in Northeast Parallel Architectures Center . cheetah is capable of streaming multiple media streams irrespective of their nature. It uses TCP/IP protocol.

Section 2.0 gives an overview of the VOD server cheetah especially the protocol used for the client - server communication. Section 3.0 gives an overview of the Intel/DVI ADPCM codec which is the codec used by the application. Section 4.0 elaborates on the design and implementation of the client while Section 5.0 gives the results obtained. Section 6.0 is on Future Work to be undertaken while Section 7.0 gives the references used.

2.0 Overview of the VOD Server - cheetah

The VOD server cheetah was originally designed and implemented to support MPEG 1 movies over an Ethernet. It is capable of sourcing multiple MPEG streams concurrently. It was subsequently scaled to support low-bit-rate video streams as well using H.263 video compression standard and in the current implementation cheetah is capable of supporting multiple media streams, be they video, audio, etc.

The VOD server has been implemented both on UNIX (SGI IRIX 5.3) and Windows NT platforms. Further, clients can cross - connect, that is, a client running on a Windows NT platform can connect to and receive service from the server running on a SGI machine with IRIX and vice - versa. Fig. 1 shows a sample client - server connection setup.

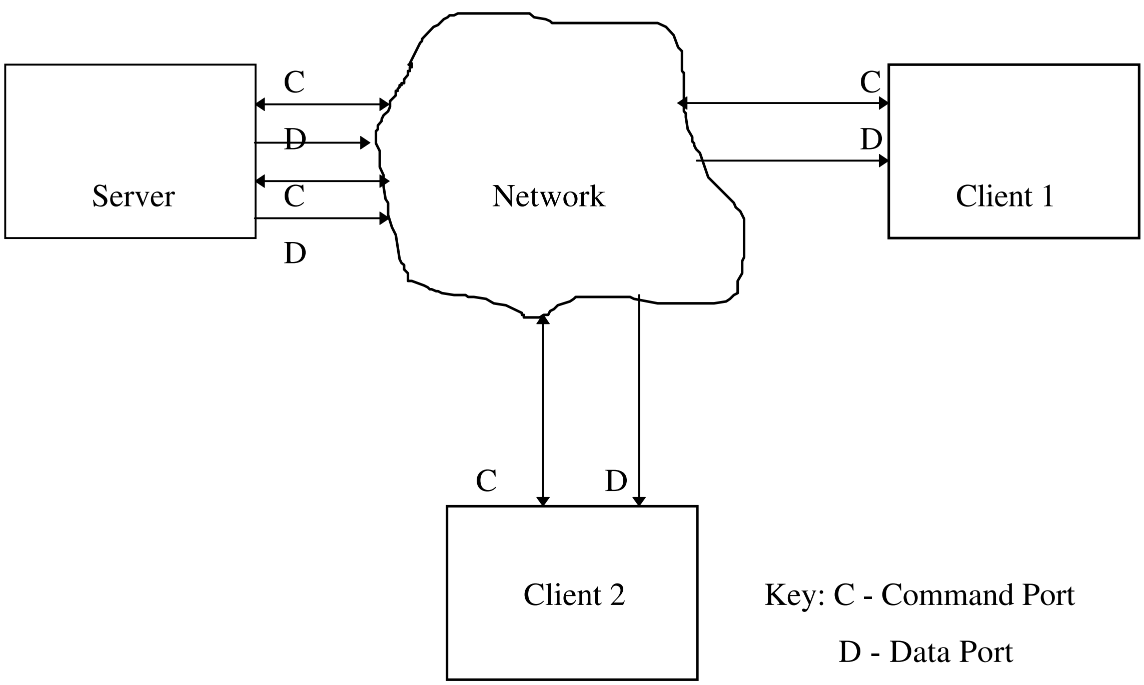

Fig. 1 An Illustrative Server - Client(s) Setup

2.1 Client - Server Communication Protocol

The server has defined two ports for communication - a Data Port and a Command Port. As such the client also opens two ports for communication. The actual data is sent over the Data Channel (formed by connecting the two Data Ports) while all control information is sent over the Command Channel (formed by connecting the two Command Ports). As mentioned earlier, the protocol used for transmission is TCP/IP.

The following are some of the commands used in the client-server communication:

VODLoadMovie : This is the first command that is sent from client to server when connection is established. It tells the server to read (n+1) bytes from the network immediately following this command, where n is the length of the filename specified in the first byte. This file may be for video, audio or for anything else; the server does not check to establish the contents of the file. The server then opens the named file.

VODSeekToStart : This command tells the server to seek to 0L file offset, that is, go to beginning of the file.

VODSeekToEnd: This tells the server to seek to the end of the file.

VODSeekToOffset: The server seeks to the desired file offset relative to the start of the file.

VODPlay: This command instructs the server to start playing the file previously specified through *VODLoadMovie*.

VODPause: The server suspends reading and transmitting of file data over the Data Channel until the *VODPlay* command is specified again.

VODStop: The server stops reading the file and transmits all pending data.

VODAckStopMessage: This message is sent from server to client when it (server) has stopped reading the input file and transmitted all pending data. Upon receipt of this command the client may terminate the connection using the command given below.

VODTerminateConnection: The server closes all open files being used for the client from whom this message was received, transmits all pending data and closes the connection with the client.

3.0 Encoding and Decoding of Audio Files

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 very good quality for voice signals (8KHz sampling) and reasonably good quality for music (11.025KHz, 22KHz and 44.1KHz). 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

This section describes the file format used for slideshow, the connection setup and call management, receiving of audio, decoding it and playing it, and finally design of random access to the audio file to provide true audio - on - demand capabilities.

4.1 File Format

In order that the client application can read and decode the audio file appropriately, it is necessary to add all the relevant audio information as header in the audio file. This addition of all the file information is done at the time of encoding. The pertinent information required by the IRIS Digital Media Library (described elsewhere in the report) are:

- *Channels* : channels is the number of channels (1, 2, or 4) used in the original recording and consequently the number of channels to be used in the audio playback. The current implementation supports only mono (1) channel.

- *Rate* : This is the sampling rate at which audio was sampled for recording - 8KHz, 11.025KHz, 22KHz, 44.1KHz . The current implementation plays the decoded audio at the same rate at which it was recorded and supports all the four sampling rates.

- *Width* : This is the width in bits used to represent an audio sample, in other words, it is a measure of resolution of the sampled audio signal. This may be 8, 16, or 24 bits. The current implementation supports only 16 bits since the encoder and decoder work with only this value.

- *Format*: This represents the format in which the samples are written - two's complement, floating point or double - precision floating point. The current implementation supports only the two's complement format.

Further, if it is desired to provide URLs in the audio then the url(s) and its/their time of activation must also be provided in the audio file. This may be done in one of the two following ways:

- inline the URL string at the file offset corresponding to its activation time. The advantage of this scheme is the reduction in the number of bytes since the time of activation of URL doesn't have to be provided. The major disadvantage is that to enable this, the decoder will have to be changed since it should be able to recognize the URL while decoding the audio stream. Also for the decoder to be able to recognize the URL string a special flag or some other suitable means will have to be provided, and this more than offsets the above mentioned advantage.
- add the URL string(s) along with its/their time of activation represented as a corresponding file offset. This scheme is simple to implement and does not involve the modification of the decoder.

Hence the latter scheme was chosen. The file header format is shown in figs. 2 for one URL.

0	1	2	3	4
Channels	Rate	Width	Format	URLflag

5	6	7	n+8	n+9	n+10	n+11
num	n+5	URL string	offset3	offset2	offset1	offset0
1	1	n + 1	1	1	1	1

Fig 2. File Header Format for the encoded Audio File

URLflag is a flag which denotes the presence or absence of URLs. If URLflag is true (i.e. 1) the string shown in fig. 2 follows it else the file header is terminated after the URLflag. num is the number of URLs in the file, (n+5) represents the number of bytes following the current location, where (n+1) is the length of the null-terminated URL string. The four offsets represent the 4-byte file offset into the (decoded) audio file. These have been unpacked into 4 separate bytes in order to provide network byte order.

num such strings, such as shown in fig. 2, are concatenated to give the URL strings and their time of activation.

As mentioned previously all the above is added at the time of encoding the audio file. The encoded audio file is appended to the header.

Random access has not been implemented but the design for the same is given here. This can be implemented by somehow preserving the encoder state at some fixed intervals and then using these as markers.

Thus, while encoding the audio file, the encoder inserts the state of the encoder which consists of the previous output value, an index into the stepsize table, and the file offset corresponding to the time at which this state occurs. The encoder state is inserted in the audio file every 500 ms of play time. To enable the decoder to distinguish the encoder state it is preceded by a flag.

Fig. 3 shows an inserted state in the encoded audio file.

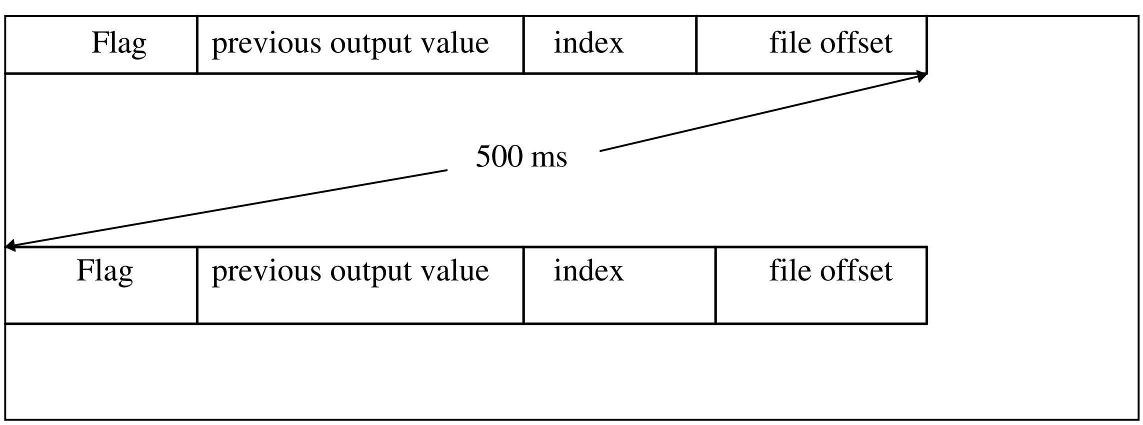

Fig 3. Shows the position of encoder state in a chunk of encoded audio

4.2 Connection Setup And Call Management

The client initializes all data structures and opens the relevant sockets. It then attempts a connect with the server and on being successful sends a VODLoadMovie command to it along with the audio file to be played. This is shown fig. 4

Thereafter it continues reading from the network until the EOF is reached. The client then sends a VODTerminateConnection message to the server whereupon the connection is severed.

Random access to enable rewind, forward and pause has not been implemented. However the design for the same has been included here and elsewhere in the report.

If the client receives a seek message from the user, it reads in the corresponding time to which the seek is desired, converts it into the file offset and locates the nearest encoder state to it. While reading from the network the encoder state is stored by the decoder in a table along with its file offset. Once the nearest encoder state has been located the client sends a VODStop message to the server and then waits for the VODAckStopMessage from it (see fig. 4). When it receives this message it sends VODSeekToOffset command along with the file offset. Then it sends VODPlay command to the server and the normal sequence of read, decode and play resumes.

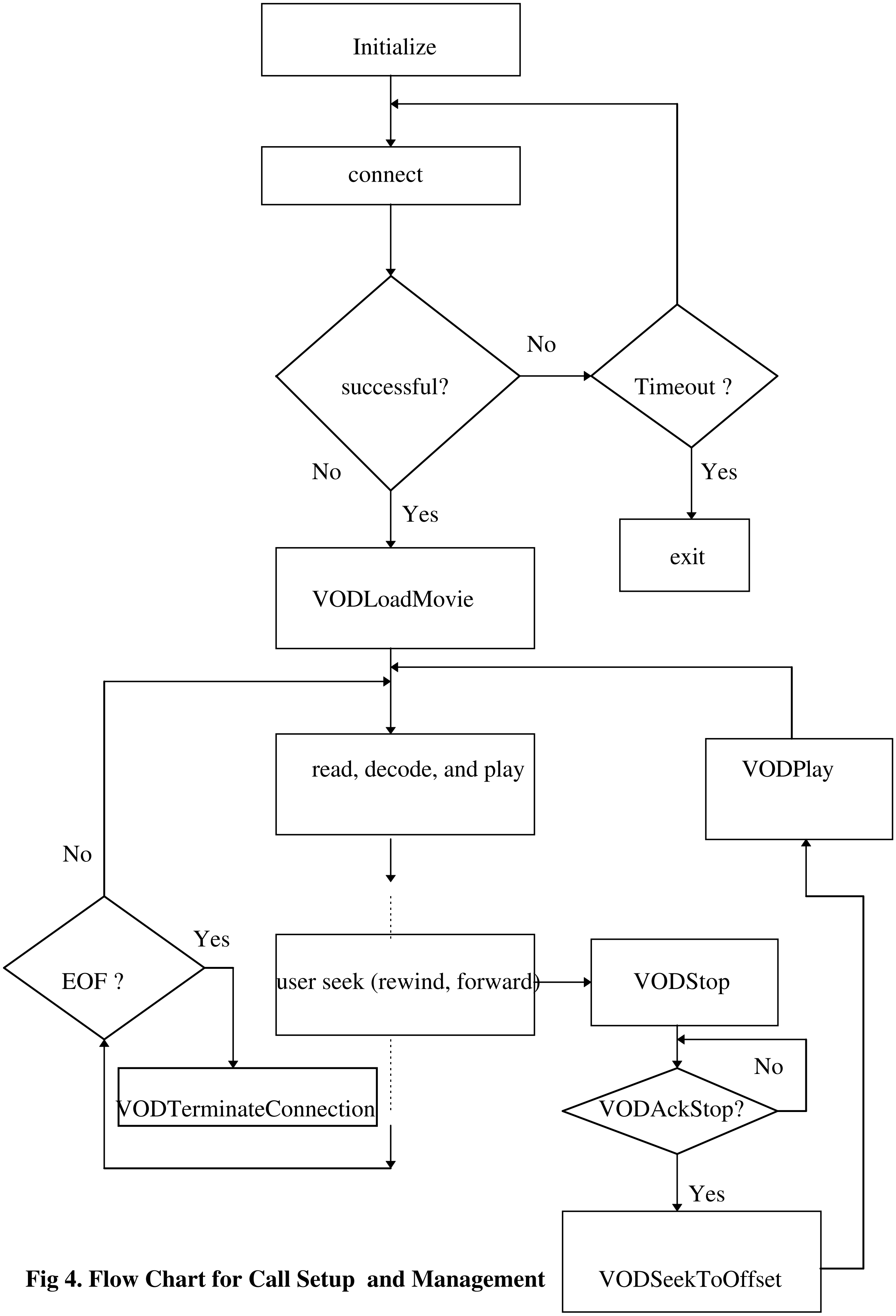

Fig 4. Flow Chart for Call Setup and Management

4.3 Program Flow

This sub - section covers the audio receiving, decoding and playing, and opening of URLs, if present. Fig. 5 gives a bird's eye view of the application and its environment. Section 4.3.1 gives an overview of the IRIS Audio Library, section 4.3.2 covers the audio handling and section 4.3.3 covers the URL handling.

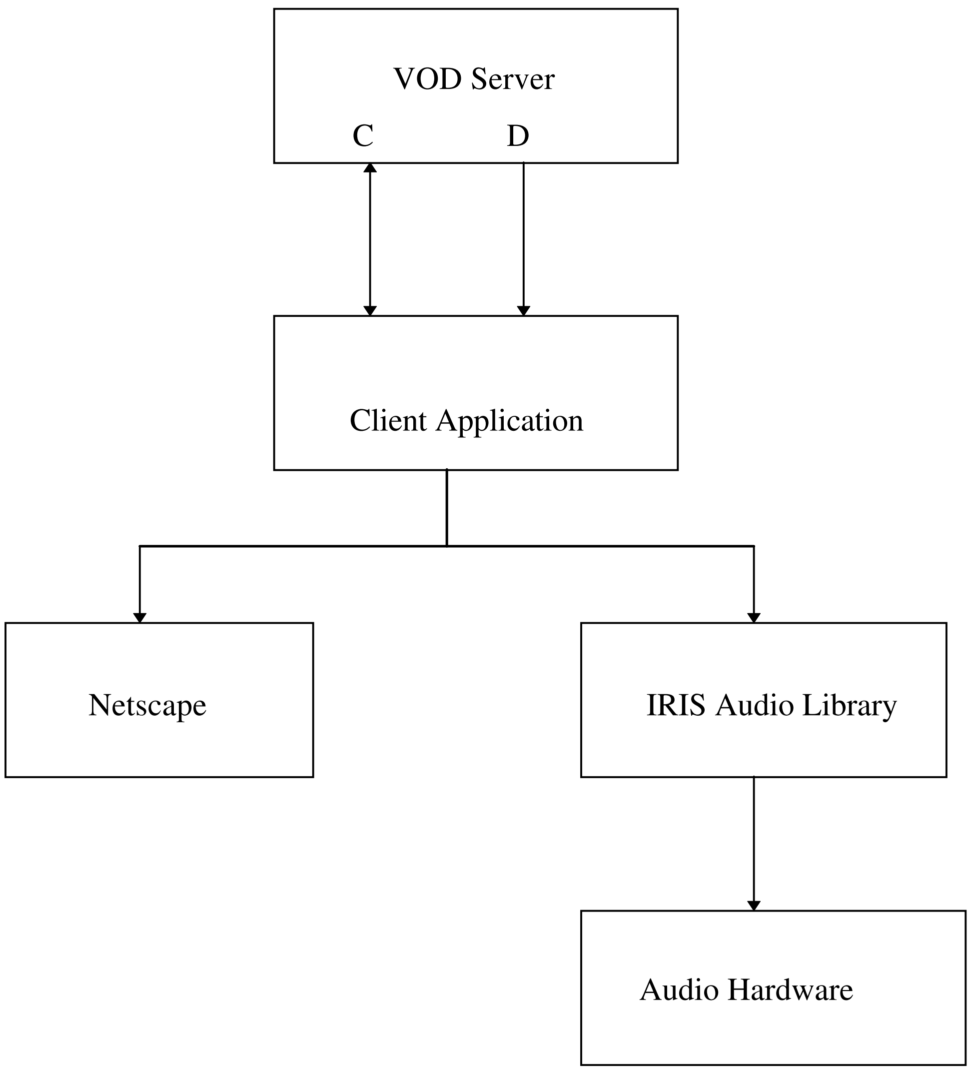

Fig. 5 Bird's Eye View of the Application And Its Environment

4.3.1 An Overview of the IRIS Audio Library

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

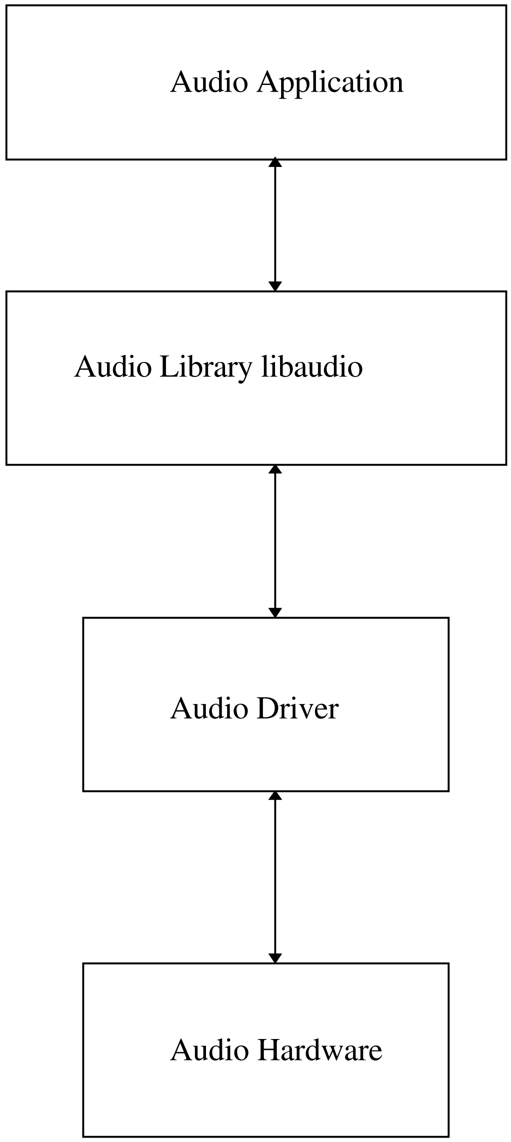

Fig 6. Interaction of Digital Audio System Components

4.3.1.1 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 slideshow). 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.

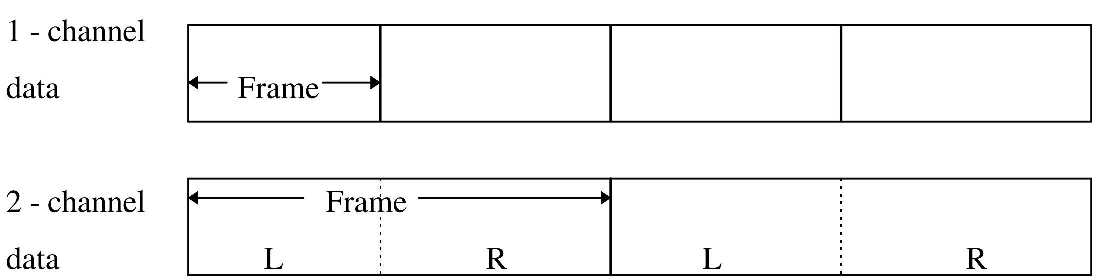

Fig 7. Audio samples and frames

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 slideshow, a 16-bit output port has been used.

4.3.1.2 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 ALseterrorhandler().
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 :

- 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

4.3.2 Audio Receive, Decoding and Play

Fig. 8 shows the module interactions in handling the audio stream from the server.

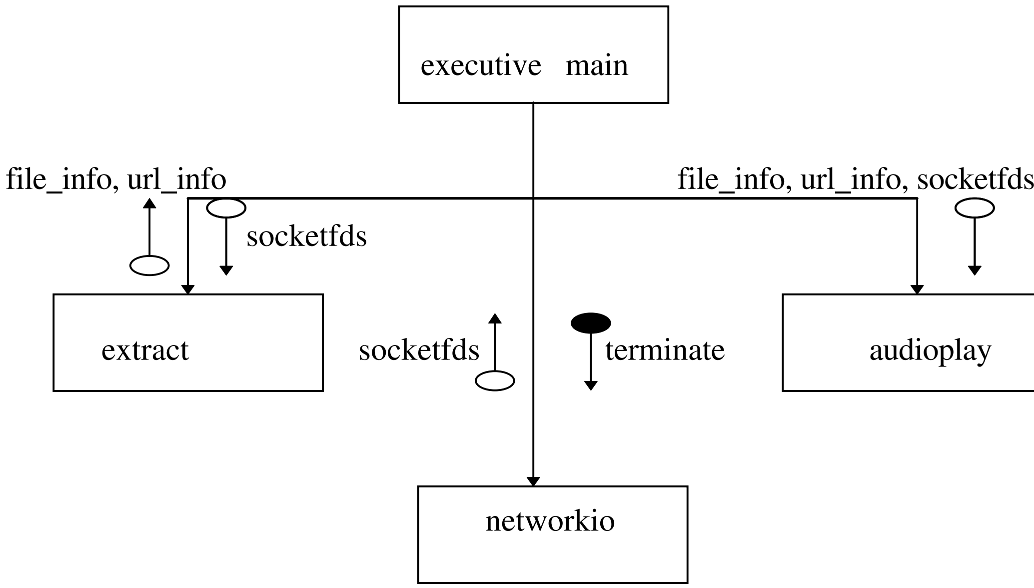

Fig. 8 Module Interaction for Audio Handling

main obtains the socket descriptors from the networkio module and then invokes the extract module passing to it these socket descriptors. The extract module extracts the file_info (channels, rate, width, format, urlFlag and number of URLs). Depending on the number of URLs found the module is invoked again by main to get url_info (an array of structures, each containing the URL string and its file offset relative to the beginning of the decoded audio file). After obtaining all the relevant information, main invokes the audioplay module passing to it all the above mentioned information.

AudioPlay Module

Fig. 9 shows in more detail the audioplay module which is the heart of audio processing. The Play function is the public interface function for this module. (Not all functions are shown for clarity). Play sprocs the slideshow process passing to it url_info and number of URLs. This process is responsible for launching Netscape, if necessary, and opening the

URLs. It is described later in greater detail. The Play function then initializes the audio output ports through the IRIS Audio Library. It then invokes the networkio module to

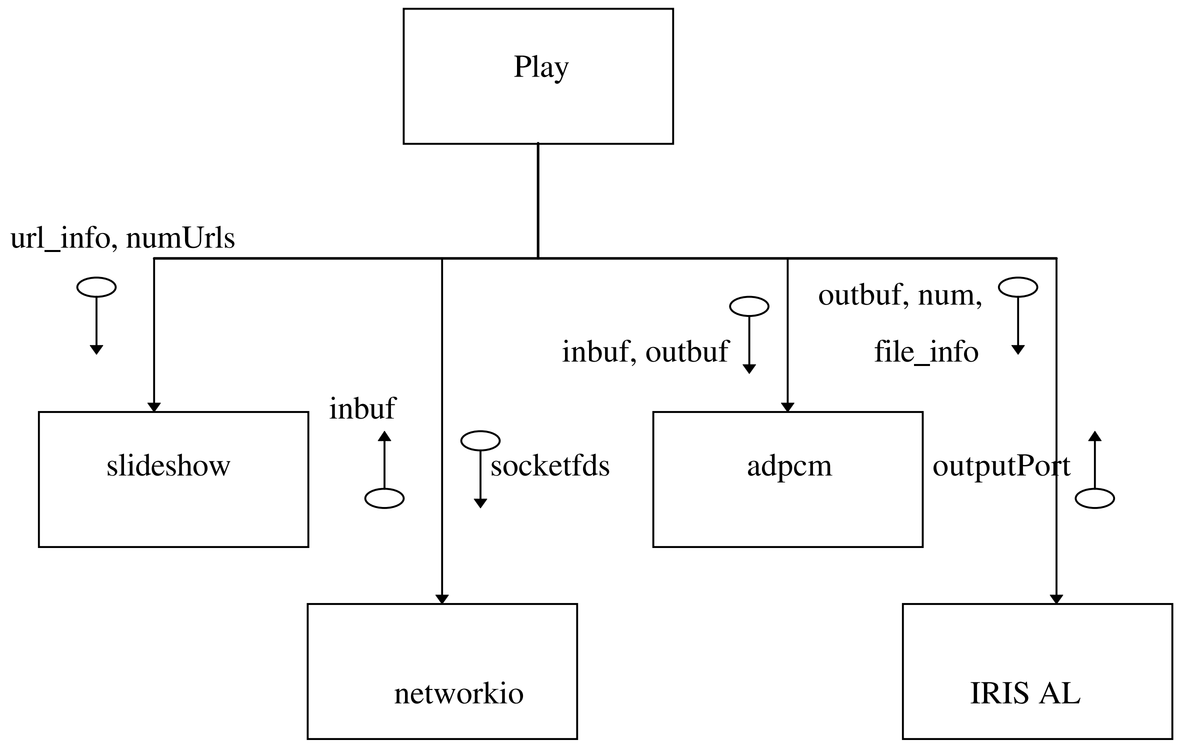

Fig. 9 AudioPlay module and Its Interaction with its Environment

read 100 ms worth of audio data from the network and put it into inbuf. This inbuf buffer is then passed to the adpcm module where the decoding takes place. The decoded output is put into outbuf. After this the Play function invokes the Audio Library again to output this data to the audio hardware (speaker). This cycle is repeated till EOF indication is given by networkio module after which the connection is terminated as described in Call Setup and management .

slideshow process

Fig. 10 shows the structure chart for the slideshow process. The slideshow function sprocs two more processes - launcher to open Netscape if it is not already running, and timer to keep track of the time at which a particular URL has to be opened.

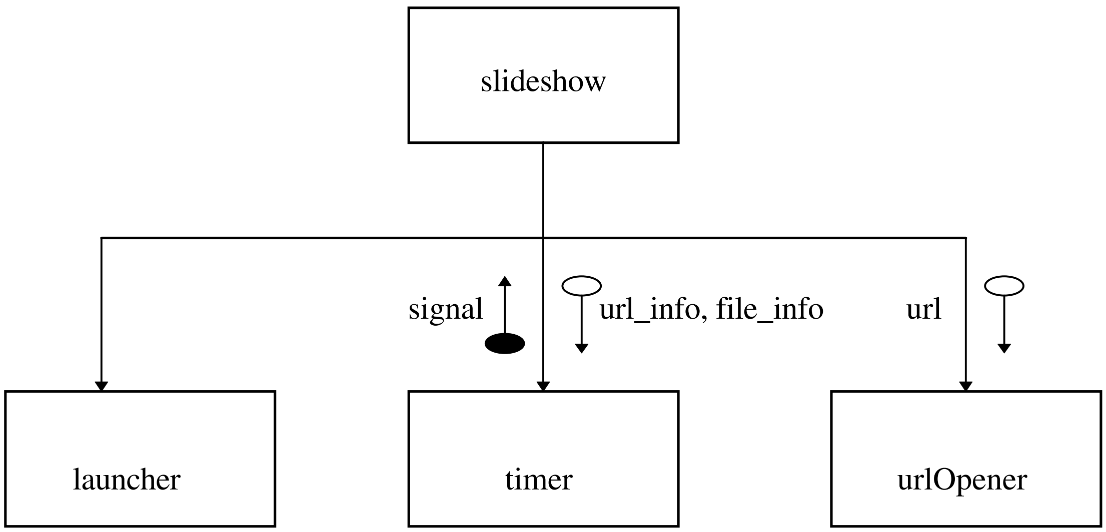

Fig. 10 URL Handling by slideshow process

launcher first checks if Netscape is running on the host machine, if yes then it sets a flag to this effect and quits, else it launches Netscape remotely. The timer process is passed the pointer to the array of url_info structures . The timer converts the file offset for a particular URL into time since it is able to view the file_info structure (which has the playing rate). It then merely sleeps till the time an ith URL has to be opened and then accordingly it signals its parent, that is, the slideshow process. Therafter it sleeps again till the next URL activation time. The slideshow process (which had previously established a signal handler urlOpener for this purpose) catches the signal. The signal handler urlOpener reads in the corresponding URL and by remotely controlling Netscape, passes the openUrl ( ) command to it telling it the URL location.

Thus the timer looks at the URL activation time (represented as file offset in the url_info structure) while the urlOpener looks at the corresponding URL. In this way URLs are opened till there aren't any more left.

networkio module

networkio module is not described here as primarily it is a reused module but modified by the author suitably. Essentially it performs the following functions:

- sets up the sockets and binds the client address
- sets up the connection facility
- connects to the server
- sends commands to the server through the Command Channel, and
- reads in a block of data from the Data Channel

5.0 Results

The slideshow was tested in NPAC and was found to perform reasonably well on sample test cases.

However the timer and urlOpener are not precision synchronized, but since this is not a binding requirement (upto 1s of out-of-sync is quite acceptable) this flaw is tolerable.

Sometimes it takes Netscape more time to launch and the audio play starts without Netscape running. This then results in missing URLs. The remedy is to start audio play only when Netscape has started. But if this time is substantial then it could result in losing audio packets from the network buffer since the server continues to send audio packets. As such a compromise ought to be reached regarding the maximum delay to be tolerated before audio starts playing.

The audio quality is quite good and comparable to that obtained from G.721 codec, though no tests were made to establish this concretely.

6.0 Future Work

Random access in the audio file has to be provided to support rewind, pause and forward. Also with this provision care will have to be taken to offset the URL and time count accordingly.

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

6.1 Remote Control of UNIX Netscape, Netscape Client APIs 2.0, Netscape Technical Library