INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Université d'Ottawa · University of Ottawa

Design and Implementation of a Java H.263/G.726 Decoder

By
Spencer Leung

A thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements for the degree of

Master of Applied Science
In Electrical Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering
School of Information Technology and Engineering

Faculty of Engineering
University of Ottawa

Ottawa, Ontario, Canada, 2002

© **Spencer Leung, Ottawa, Canada, 2002**

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-72776-9

Canada

Abstract

In this thesis, we present a decoder that is capable of performing real-time decoding of video and audio streams on Java-enabled platforms. We decompose the design phase into two stages: the declaration of design objectives and the proposition of a high level design. The design objectives are to build a decoder that is efficient, small in size, and adaptable to different platforms and network environments. The high level design is composed of key algorithms and major building blocks of the decoder. We document some challenges encountered during the implementation phase, some of which required revision of the initial design. We conduct test cases under some representative network environments to demonstrate the adaptability of the decoder. Lastly, we identify the work areas to be extended.

Acknowledgements

I would like to express my appreciation to my supervisor, Professor Nicolas D. Georganas, for being a valued source of guidance and commentary. I have benefited very much from the generous help, guidance and many fruitful discussions I had with Dr. Jauvane C. de Oliveira, who provided his H.263 Java decoder source, and also a voice decoder software. Many thanks are given to my parents, Dr. & Mrs. Kar-Koi Leung, and to my brother, Elton, for their encouragement. I also owe a great debt to Mr. & Mrs. Matthew Wong for their generous hospitality during my stay in Ottawa. Finally, I would like to thank my classmates - Jacqueline Chan, Ada Chow, Matthew Ho, and Wai-Pong Lai - who have offered help and support in various stages of my master's.

Table of Contents

Abstract I

Acknowledgements II

Table of Contents III

Chapter 1 - Introduction 1
- **1.1 The Objective of the project** 1
- **1.2 Background of the Project** 1
- **1.3 Design Goals** 2
  - *1.3.1 100% Java implementation* 2
  - *1.3.2 Broad Audience Reachability* 3
  - *1.3.3 Support of Open Standards* 4
- **1.4 The Applications of the JPlayer** 4

Chapter 2 - Overview of the Technologies 7
- **2.1 Audio and Video Fundamentals** 7
- **2.2 Audio and Video Coding and Compression** 7
  - *2.2.1 Pulse Code Modulation (G.711)* 8
  - *2.2.3 Video Coding for Low Bit Rate Communication (H.263)* 13

Chapter 3 - Design requirements 16
- **3.1 Design Goals** 16
- **3.2 The Use of the Java Language** 16
  - *3.2.1 100% Java Implementation* 17
  - *3.2.2 The Use of the Standard APIs* 18
  - *3.2.3 Compliance with Personal Java* 18
- **3.3 Wide Adoptability of Different Network Environments** 20
  - *3.3.1 Networks of Different Bandwidths* 20
  - *3.3.2 The Effects of Disturbance on IP Networks* 21
- **3.4 Implementation of Open Standards** 22

Chapter 4 - High Level Design 24
4.1 The G.726 Codec 24
4.2 Request Algorithm 24
4.3 Four Major Building Blocks 30
4.3.1 The Session Controller 31
4.3.2 The Decoder 34
4.3.3 The Buffer 35
4.3.4 The Scheduler 38
4.4 Network Monitoring 41
4.4.1 The Average Round-trip Time 41
4.4.2 The Average Interframe Separation 43

Chapter 5 - Implementation Challenges 44
5.1 TCP vs. UDP 44
5.2 The Multi-thread Approach 46
5.2.1 Priority Tuning 47
5.2.2 Critical Regions 48
5.3 Issues with Applets 48
5.3.1 Applet Behaviors in Netscape and Internet Explorer 49
5.3.2 Performance and Java Version 51
5.3.3 Security Restrictions of Applets 51
5.4 Playing Audio with Java 52
5.4.1 The sun.audio Package 53
5.4.2 The .AU File Header 53
5.4.3 The Java Sound API 55
5.4.4 Identifying Java Version 56
5.5 Audio Interruption Handling 58
5.5.1 Approach 1: Silence Filling 58
5.5.2 Approach 2: Repetition of Audio Data 59
5.5.3 Approach 3: Blocking the Audio Player 60
5.6 Optimization for Speed 61
5.6.1 Objects Allocation 61
5.6.2 Integer vs. Shorts 62
5.6.3 The Nagle's algorithm 62
5.7 Audio/Video Synchronization 63

Chapter 6 – Test Cases and Results 67
6.1 Test Cases 67
6.1.1 The performance of The JPlayer Audio Decoder 67
6.1.2 Synchronization of the Audio and the Video Decoders 68
6.1.3 CPU Utilization 69
6.2 Test Environments and Reference Machines 70
6.3 Test Results 71
6.3.1 The Performance of the Audio Decoder 71
6.3.2 Synchronization of the Audio and the Video Decoders 72
6.3.3 CPU Utilization 73
6.4 Analysis of the test results 74
6.5 Areas for Improvements 78

Chapter 7 – Conclusion 80

References 82

Appendix A - Competitive Analysis 84

Appendix B - List of Figures and Tables 89

Chapter 1 - Introduction

1.1 The Objective of the project

The objective of this project is to design and implement a decoder that is capable of decoding G.726 audio streams and H.263 video streams. The G.726 and H.263 decoder is expected to run on various platforms. Java has been chosen as the language of the implementation. The rest of this chapter will focus on the background, the design goals, as well as the applications of the project.

1.2 Background of the Project

To avoid needless repetition and complication, the name "JPlayer" rather than "G.726 and H263 decoder" will be used through out this thesis.

One major component of the JPlayer, the Java H.263 video decoder, (hereafter to be referred as the "JStreaming Video Decoder" or "video decoder") is an earlier work of Jauvane C. de Oliveira. [1] Access to source code of the JStreaming Video Decoder is granted by Jauvane C. de Oliveira to the author of this thesis. Enhancements such as the robustness improvement to the JStreaming Video Decoder, the addition of G.726 audio streaming functionality which has been given the name "JPlayer Audio Decoder", and the

synchronization mechanism between the video and audio decoders are implemented by the author. In achieving the additional functionalities and extensions, some minor code changes are made to the JStreaming Video Decoder. These changes are kept to a minimum to permit the existing users of the JStreaming Video Decoder which provides only video decoding, to upgrade easily to the new JPlayer with both video and audio decoding capabilities. The author of this paper has contributed mostly on the JPlayer Audio Decoder as well as the synchronization mechanism. Therefore, the body of this thesis will mainly focus upon these two mentioned aspects.

1.3 Design Goals

The objective of the project stated above is dissected into a few design goals. In this section, the design goals are identified and each of them is discussed in detail in turn.

1.3.1 100% Java implementation

The JPlayer takes advantage of the benefits inherent in the Java language. By using Java, the player is portable, customizable, and robust. Java is platform agnostic. Not only does Java allow the JPlayer to reach the largest audience on the web without the need to install any specialized media player, it also reduces the amount of development effort needed to

produce cross-platform streaming media applications.

1.3.2 Broad Audience Reachability

As of this writing, none of the popular media players are supported on every platform. (See Appendix A for details) The multimedia support on the palm size devices is especially limited due to the demanding requirements of most media players. The JPlayer on the other hand gives unparalleled audience reach when compared to standalone media players. Most popular browsers such as Internet Explorer from Microsoft, Netscape, and Opera are Java-enabled. This implies that most people surfing on the Internet today are able to view streaming content using the JPlayer immediately.

While a lot of standalone media players such as Real Player require installation to be performed on each client computer or device, which limits the popularity of those media players, the JPlayer is a 100% Java implementation- it can easily be packaged in the form of an applet. The JPlayer applet is extremely small in size comparing to most media player as shown in Table A-1 in Appendix A and can be seamlessly downloaded along with web contents. Viewers will always have the correct version for the encoded multimedia content without the concern of additional software or plug-ins installation.

1.3.3 Support of Open Standards

The JPlayer is designed to decode H.263 video streams and G.726 audio streams. Both H.263 and G.726 standards are open standards set by the International Telecommunications Union (ITU). The H.263 standard is a video compression algorithm, based on the Block Discrete Cosine Transform (DCT) and motion compensation, for low bit-rate environments. [2] G.726 is an ITU recommendations for digitizing and compressing speech signals using Adaptive Differential Pulse Code Modulation (ADPCM) operating at very low bit rates. [3] By complying with H.263 and G.726, multimedia contents can be created, delivered and manipulated by products of different vendors without concern of hardware or software compatibility.

1.4 The Applications of the JPlayer

The application of the JPlayer is broad. Today, a lot of devices are connected to networks of every kind. With the introduction of the third generation wireless network, as well as other newly established media access technologies such as Firewire, IEEE 802.11a/b, Bluetooth, etc.; the declining cost of CPU power; and the growing usage of the Java language in the appliance product market, even more devices will be expected to be

"connected" and capable of receiving digitized video and audio streams. The JPlayer is very portable due to the 100% Java implementation, and its small footprint.

The JPlayer can be ported to virtually all Java-enabled consoles with a graphical display, minimal audio support, and low bit-rate network support. These low requirements allow the application of the JPlayer to be broad. From desktop computers to palm-sized devices, to wired terminal, to Java-enabled mobile phones, the list of possible candidates to receive multimedia content using the JPlayer are growing. Figure 1-1 shows some examples of the applications of the JPlayer.

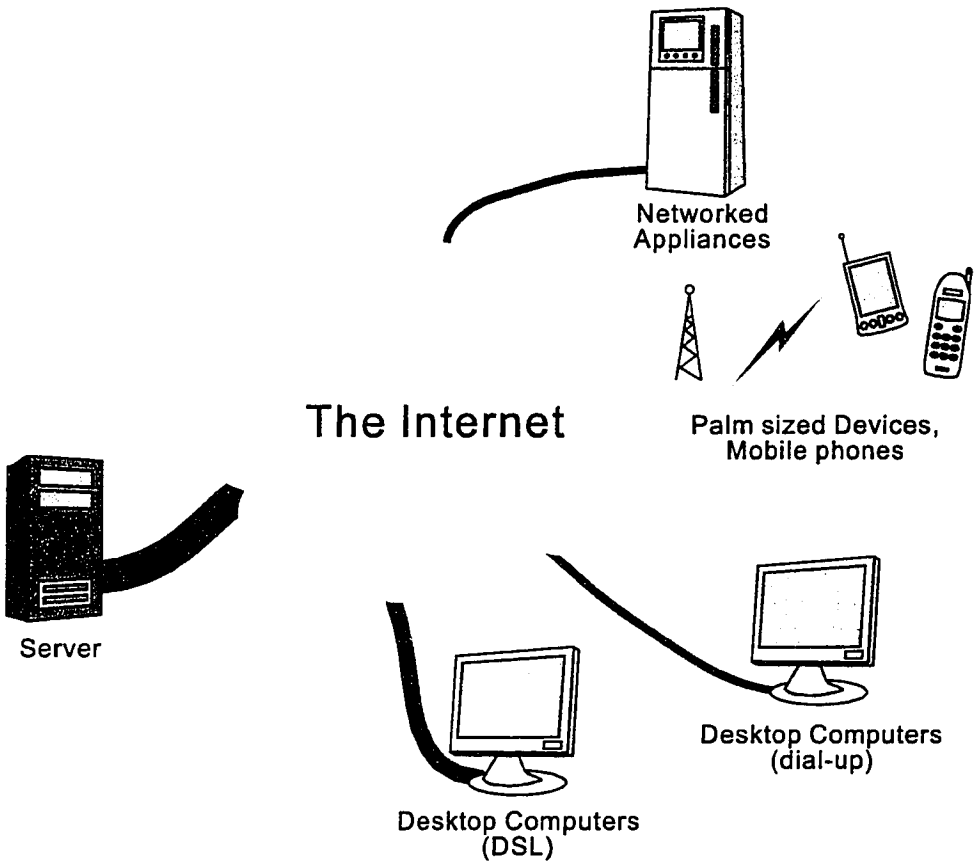

Figure 1-1 - Examples of the applications of the JPlayer

1.5 Organization of this Thesis

The rest of this thesis is organized as follows. In Chapter 2, some audio and video encoding technologies relevant to the project are reviewed. This includes a brief introduction to Pulse Code Modulation (G.711), Adaptive Differential Pulse Code Modulation (G.726), and Video Coding for Low Bit Rate Communication (H.263). In Chapter 3, design requirements of the project are listed and the importance of each is also examined. The high level design of the project is covered in Chapter 4 in which different aspects of the initial design are reviewed. In Chapter 5, some challenges encountered during the implementation stage are documented. Some of these problems have called for major redesigns in order to be resolved. The performance results of the JPlayer are presented in Chapter 6. The thesis is summarized in Chapter 7.

Chapter 2 - Overview of the Technologies

2.1 Audio and Video Fundamentals

Often the bandwidth requirements of uncompressed video or audio data are huge in comparison to the available bandwidth of the underlying network. It is sometimes necessary to perform data compression to reduce the bandwidth requirements prior to transmission over communication channels. In most cases, the decompression of data on the client side must consume the minimum amount of time and resources, in order to satisfy the tight timing and resources constraints of the real-time streaming applications. The coder/decoder pair (codec), which is the centerpiece of any communication systems, performs the functions of compressing, and decompressing video and audio data, as well as other forms of media.

2.2 Audio and Video Coding and Compression

This section of the thesis lists the encoding and compression technologies used in the JPlayer. They include Pulse Code Modulation (G.711), Adaptive Differential Pulse Code Modulation (G.726), and Video Coding for Low Bit Rate Communication (H.263).

2.2.1 Pulse Code Modulation (G.711)

Analog audio signal is digitized through a process called Pulse Code Modulation (PCM). [4] The sampling rate and the number of bits per sample are the two major parameters of the process to control the fidelity of the digitization. Sampling rate is the number of samples taken in one second and the number of bits per sample is the number of bits used to represent each sample value. Optionally, a PCM signal can have more than one channel- it can have one for mono, two for stereo, or more for some high-end systems. To digitize audio signal, the analog audio signal is to be passed through several stages. First, the signal is low-pass filtered to remove the unwanted, high frequencies. Audio signals, which have a frequency above half of the sample rate, will be "aliased", or translated to lower frequencies, and result in a distorted reproduction of the original sound wave. Although humans can perceive sound frequencies between 20 Hz and 20 kHz, videoconferencing systems are typically designed to deliver telephone quality audio signal which encompasses merely a small range of frequencies extending from 300 to 3300 Hz. After filtering, the signal is sampled. The amplitude values of an analog audio signal, which represents the loudness of the signal, are continuously varying in time. To encode this signal digitally, the amplitude value of the signal is measured at regular intervals. According to the Nyquist Theorem, to have lossless digital representation of the

analog signal, the sampling rate must be at least twice that of the highest frequency present in the analog waveform which is termed the Nyquist rate. Quantization is the step that follows sampling. It requires the sampled values to be mapped into one of a set of fixed values, which are usually binary coded. The errors, which result from the mapping of analog values to quantized levels, are given the name of quantization noise. The degree of quantization noise drops as the quantization levels become closer together. Thus, more bits of quantization resolution translate to less quantization noise, and hence greater dynamic range. Some implementations of the quantizer have a linear quantization which the quantization levels are distributed evenly across the range. Other approaches may use a logarithmic quantization, which have more quantization levels in the lower range. The logarithmic quantization has a greater dynamic range comparing to linear quantization and achieves better voice quality over silence or small amplitude pars of speech. An illustration of the linear and logarithmic quantization approaches is shown in Figure 2-1.

The μLaw and the aLaw are two examples of logarithmic encoding schemes. The μLaw encoding scheme is used on North American and Japanese phone systems, and is coming into use for voice data interchange, and in private branch exchanges (PBXs), voice-mail systems, and most computer systems. In μLaw encoding, 14 bits of linear sample data are reduced to 8 bits of logarithmic data. The aLaw encoding scheme is used

in Europe and the rest of the world.

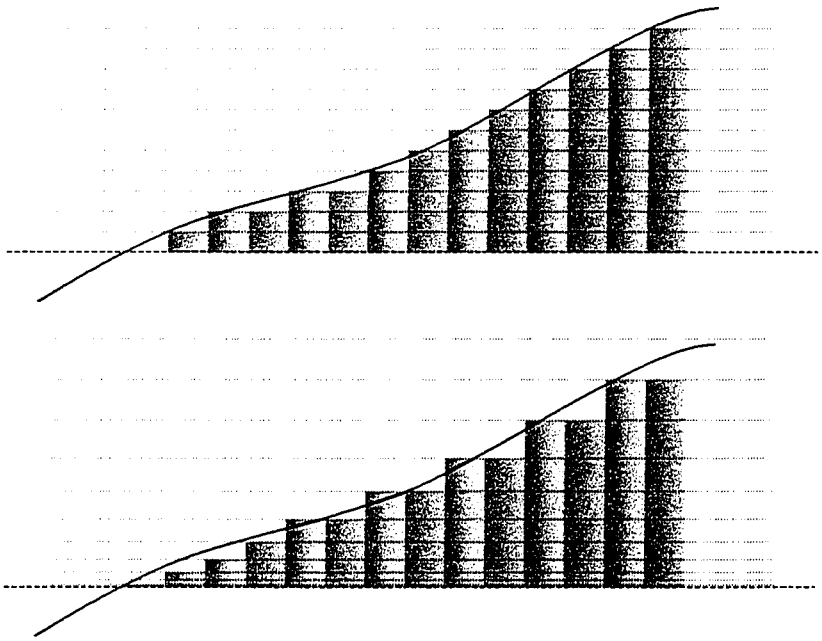

Figure 2-1 - Linear (above) and logarithmic (below) quantizations
(Note: Figures are not drawn to scale.)

The total bit rate for a mono PCM signal can be computed by multiplying the sample rate by the number of bits per sample. A stereo signal would require twice the bit rate. A typical voice transmission of telephone quality requires about 3 kHz of audio bandwidth and 256 quantization levels, resulting to a 64 kbps channel with a sample rate of 8 kHz and 8 bits per sample. ITU-T has standardized the 8 kHz, mono, 8 bits per sample version of PCM into the G.711 recommendation. The standardization, however does not impose the use of a particular quantization method.

2.2.2 Adaptive Differential Pulse Code Modulation (G.726)

Adaptive Differential Pulse Code Modulation (ADPCM) is a compressed version of PCM, which requires a lower bit rate than the standard PCM to transmit voice information of the same quality. The ADPCM takes advantage of the property of patterns present in speech samples, where previous values of PCM samples are used to predict the value of the current sample. This prediction algorithm is performed at both transmitting and receiving ends. The transmitter compares the actual sampled value to the predicted value, and computes the error. Because the error signal will have a lower variance than the original speech samples, it can be quantized with fewer bits than the original speech signal. This error signal is then transmitted. Because the prediction algorithm is performed at the receiving end as well, the receiver also knows what the predicted value is based on the previous samples. It uses the error signal to make an adjustment to the predicted value and reproduces the original sample. This predict-compare-adjust process is repeated for each input sample, reproducing the original PCM samples at the output. The system is called "adaptive" because the prediction parameters and the quantization levels of the error signal can change dynamically depending on the rate of change of the sample values (i.e. signal level). Figure 2-2 shows the difference between PCM and

ADPCM in a pictorial form.

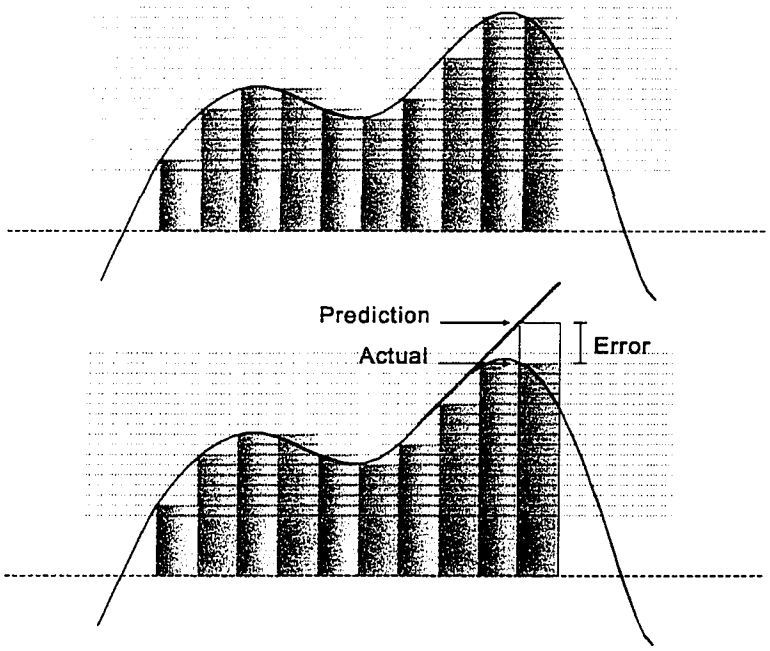

Figure 2-2 – PCM (above) and Adaptive Differential PCM (below)

(Note: Figures are not drawn to scale.)

Different flavors of ADPCM encoder/decoders vary the way the predicted value is calculated and how the predictor or quantizer adapts to signal characteristics. This results in various levels of compression. Many ITU-T videoconferencing recommendations incorporate ADPCM as encoding method. Standards include G.722, G.726, and G.729. The Java audio codec implements the 24kbps, 32kbps and 40kbps version of thte G.726 standards. G.726 32kbps version gives a 2:1 compression ratio. The 24kbps and 40kbps version give an 8:3 and an 8:5 compression ratios respectively. Various characteristics of

these standards are summarized in Table 2-1.

Standard	Sample Rate	Decoded Sample Size	Encoded Sample Size	Bandwidth Requirement
G.726 (4 bit)	8000 Hz	8 bit	4 bit	32 kbps
G.726 (3 bit)	8000 Hz	8 bit	3 bit	24 kbps
G.726 (5 bit)	8000 Hz	8 bit	5 bit	40 kbps

Table 2-1 - Characteristics of different G.726 audio encoding standards

Higher quality speech (50 Hz -7 kHz, 14 bit resolution) may be encoded by dividing the audio spectrum into two sub-bands and performing separate ADPCM coding on each. The technique is covered in G.722 and is called "Sub-Band ADPCM". [5] G.722 specifies three modes of operation: 64, 56 and 48 kbps. The discussion on this standard is beyond the scope of this thesis.

2.2.3 Video Coding for Low Bit Rate Communication (H.263)

The H.263 is a video codec for narrow telecommunications channels (less than 64 Kbps). The formal name of H.263 is "Video Coding for Low Bit Rate Communication." It is a backward compatible enhancement of the H.261 standard [7] with many performance and error recovery improvements. As with H.261, the support of the Quarter Common

Intermediate Format (QCIF) is mandatory. (Refer to Table 2-2 for the details on QCIF and other formats.) H.263 also optionally supports several other formats. Sub-QCIF is supported for very low-resolution images. Also supported are 4CIF, which has four times the resolution of CIF, and 16CIF, which has a resolution sixteen times that of CIF. Table 2-2 presents a summary of the video picture formats.

Picture Format	Number of Luminance Lines	Number of Luminance Pixels	Supported in H.261	Supported in H.263
Sub-QCIF	96	128	Not supported	Optional
QCIF	144	176	Optional	Mandatory
CIF	288	352	Mandatory	Mandatory
4CIF	576	704	Not supported	Optional
16CIF	1152	1408	Not supported	Optional

Table 2-2 - Supported picture formats of H.261, H.263 video standards

While the H.261 standard was limited to full pixel precision for motion compensation, H.263 has required support of half-pixel precision. The half-pixel refinement greatly improves the picture quality, particularly in low-resolution video. New to the H.263 standard are negotiable coding options offering improved performance. One of these options is the support of P-B frames used in interframe encoding. This is a technique that

is also used in MPEG video. Although it is computationally more expensive, it allows much higher compression, and therefore a potentially higher frame rate.

Chapter 3 - Design requirements

3.1 Design Goals

The goal of this project is to design and implement a video/audio decoder that is appropriate for wide range of platforms, with the streamed content transmitted over various media. This chapter of the thesis outlines the design requirements of the project that would allow this goal to be achieved.

3.2 The Use of the Java Language

The JPlayer should be implemented in a computer language that is well-supported on various platforms to ensure maximum portability. Java is a good candidate for the task.

Java is well-supported on most popular platforms. Most web page browsers nowadays have build-in support for Java applets. Mobile devices such as PDAs and cellular phones are starting to incorporate the Connected Limited Device Configuration (CLDC) [8] to allow execution of Java code on these devices. The CLDC is the foundation of the Java runtime environment that targets small, resource-constrained devices, such as mobile phones, personal digital assistants, and small retail payment

terminals. The CLDC could be viewed as a subset of the Standard Java runtime environment. At the heart of CLDC is the K virtual machine (KVM), a virtual machine designed for inexpensive mobile devices such as devices running the PalmOS. Java programs that are compliant to CLDC are certified to run on all the CLDC devices. With the exception of a handful powerful mobile devices such as those Windows CE devices which are capable in running real-time multimedia applications, today's most CLDC devices are still too limited in processing power to drive any practical multimedia applications. However, newer, more capable CLDC devices are expected to emerge in the market in the very near future. Palm has plans to ship its new models in early 2002 with OS 5, which will run on ARM compliant processors from Texas Instruments. [9] Palm devices that feature the powerful ARM processor will be potent enough to drive multimedia streaming on KVM.

3.2.1 100% Java Implementation

The JPlayer must be implemented solely in Java. The standard Java runtime environment permits the interworking with the native operating systems using the C language to allow easy access to the underlying system resources such as graphics, memory, multimedia

mixer and the likes. Although such offering of the standard Java runtime environment is very useful in implementing efficient multimedia application, the JPlayer must not gain access to these resources by such means. Doing so would violate Java's "write once, run everywhere" paradigm and limit the JPlayer to execute only on one platform.

3.2.2 The Use of the Standard APIs

Apart from the restriction that the JPlayer should be implemented solely in Java, it should use the functions provided by the APIs from the Java standard environment only and should not utilize any optional APIs or APIs that are offered by third parties. If non-standard APIs are used, users of the JPlayer must first download and install the APIs before using the decoder. These restrictions are therefore necessary to ensure utmost compatibility with most Java enabled platforms without installation of additional software. For this very reason, the Java Multimedia Framework API could not be used to take advantage of its built-in decoding and playing functionalities.

3.2.3 Compliance with Personal Java

The Java H.263/G.726 decoder should be in accordance with the PersonalJava

environment to ensure maximum compatibility with most Java enabled platforms. The PersonalJava environment is a Java environment that contains a smaller set of APIs than the standard Java environment. It addresses the software needs of networked applications running on personal consumer devices such as set-top boxes and smart phones rather than desktop computers. The PersonalJava API has been derived from the Java 1.1.6 API, supplemented by a small number of PersonalJava-specific APIs designed to meet the needs of networked embedded applications and it has diverged from the more recent runtime environments since then. However, all APIs put in to the standard JDK prior to Java 1.1.6 are available for use. [10] Figure 3-1 illustrates the relationships between the different versions of the Java Runtime Environments and the PersonalJava API.

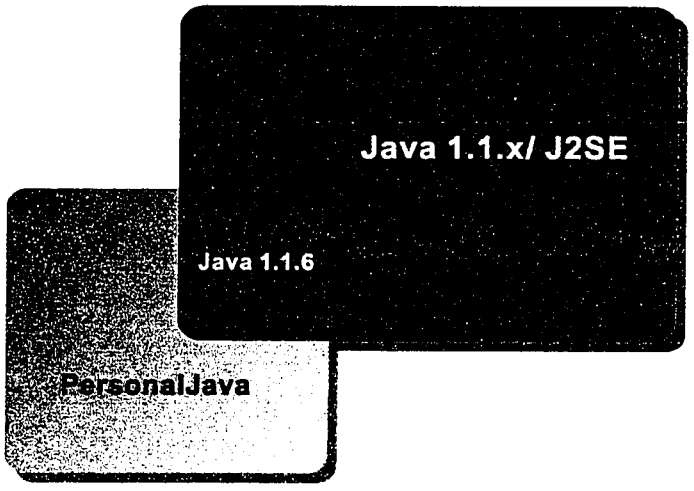

Figure 3-1 - PersonalJava and other Java Runtime Environments

Most Java-enabled palm-sized devices and embedded devices often do not implement the J2SE environment due to the scarcity in memory, but mostly comply with the PersonalJava environment. Thus, applications that are conformed to PersonalJava should run on most Java-enabled platforms with the exception of a few devices that lack floating point support. By complying with PersonalJava, the JPlayer will run on most Java-enabled platforms including most palm-sized devices and embedded devices.

3.3 Wide Adoptability of Different Network Environments

3.3.1 Networks of Different Bandwidths

The JPlayer is aimed at providing multimedia streaming to various platforms via different kinds of networks. Thus, it must be able to cope with transmission across networks with different characteristics. While the round trip time on a local area network (LAN) is in the range of milliseconds, wireless access using the wireless access protocol (WAP) could take as much as 10 seconds to respond. In the context of this paper, modern networks are classified into four groups over which the JPlayer should provide reasonable quality of service. Networks in Group 1 are those of which with a throughput higher than that of the 10BaseT LAN. Group 2 is meant to cover the high-speed internet accesses of all forms which have become very popular in the last few years. They range from 1 Mbps up to 10

Mbps. Traditional dial-up connections over the telephone network with 56 kbps and ISDNs are represented by Group 3. Networks with bandwidth under 28.8 kbps are grouped under Group 4. WAP on mobile phones and other forms of limited wireless accesses are represented in this category.

These most typical network scenarios are meant to be selected as samples of all networks environments. By achieving reasonable performance under each of the test cases, one could reasonably conclude that the JPlayer could satisfy most network environments with different characteristics. The following table summarizes the characteristics of each of the network types.

Group	Examples	Bandwidth	Estimated Round Trip Time
Group 1	10BaseT or faster	> 10 Mbps	< 10 ms
Group 2	High-speed access	1 – 10 Mbps	10 ms – 100 ms
Group 3	Dial-up	56 Kbps	100 ms – 1 s
Group 4	Wireless access	< 28.8 Kbps	> 1 s

Table 3-1 - Networks with different characteristics

3.3.2 The Effects of Disturbance on IP Networks

The JPlayer is aimed to run over IP networks primarily. An IP network is an example of a

best effort network over which successful delivery is not guaranteed, nor is the quality of service provided. Congestion on IP networks could result in delay in packet propagation or even packet lost. Some constant bit rate networks, such as ATM which allow smooth transport of multimedia content, usually have a steady data rate. IP networks however usually suffer variable delay and jitter which are undesirable for streaming applications.

Human ears are very sensitive to even the slightest amount of disturbance in audio streaming. Human eyes on the other hand, allow much greater tolerance to interruptions in video streaming. Hence, delay and jitter control for the audio component of a multimedia application must be very well engineered to avoid major degradation that affects users' experiences. One way to combat variable delay and jitter is to include a large buffer. This buffer should hold a large number of data to compensate the possibility of data congestion at any time.

3.4 Implementation of Open Standards

Among all the different formats of multimedia content in the market, many have their own proprietary formats. Employing proprietary media formats gives flexibility in designing applications but prohibits compatibility with others, nor does it allow users to create their own contents with other third party tools. By complying with an open

standard, the JPlayer would permit interoperability with other media tools. Hence users would have greater freedom in manipulating their own media clips with wide range of tools available in the market.

Chapter 4 - High Level Design

4.1 The G.726 Codec

One of the major pieces of the JPlayer Audio Decoder is the decoder that transcodes G.726 audio data to G.711 audio data which can be fed to Java audio facility directly. The G.726 codec was obtained by translating from a reference implementation in ANSI-C language to Java language by the author. The original source code is released by Sun Microsystems, Inc. to the public domain. The ANSI-C language reference implementation supports G.711 and G.726 voice compressions. They have been tested on Sun SPARCstations and have passed 82 out of 84 test vectors published by ITU-T (Dec. 20, 1988) for G.726. The JPlayer Audio Decoder has been proved to produce the same outputs as the ANSI-C implementation using various vector files as inputs. Although the algorithm of the decoder is acquired from the ANSI-C implementation, some parts of the code were optimized in Java to achieve better performance. (See Section 5.6 for details.)

4.2 Request Algorithm

The JPlayer Audio Server sends out audio data upon requests made by the client. The JPlayer Audio Decoder therefore must be equipped with the intelligence to make an

educated estimation of the number of frames to be requested based on the network characteristics and its processing power.

The author has derived the formula for the number of frames requested at any given time based on the number of frames that are stored in the buffer on the client side. This section of the thesis shows the steps for the derivation. The variables used in the derivation are defined as follows and a pictorial illustration is shown in Figure 4-1.

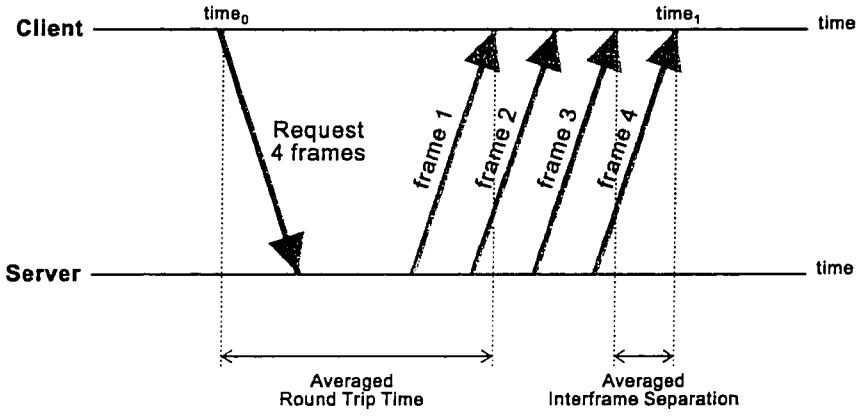

Figure 4-1 - The timing diagram of messages exchanged between the server and the client

- *buffered*$_i$ is the number of frames stored in the buffer at time$_i$.
- *average roundtrip time (ART)* is defined as the period of time between the moment a request for frames is sent and the first frame arrives;
- *average interframe separation (AIFS)* is defined as the interval between arrivals

of two adjacent frames;

- *frame departing period (FDP)* is the period at which the frames are leaving the buffer to be played.
- $request_i$ is the number of frames requests made at $time_i$.

The very first step of the derivation is to state the initial conditions. As shown in Figure 4-2 (top), at $time_0$, there is $buffered_0$ number of frames in the buffer and a request of $request_0$ number of frames is being made.

$$time = time_0 \tag{4.2.1}$$

$$buffer = buffered_0 \tag{4.2.2}$$

We simplify the calculation by assuming the first requested frame arrives when the *average roundtrip time* elapses, neglecting the small processing time on the server. Assume that the client decides to request $request_0$ number of frames at $time_0$ based on the algorithm to be found. The first frame is received after one *ART*. If each frame arrives approximately one *AIFS* after the arrival of the previous frame, then the last frame of the group should arrive ($request_0$ – 1) x *AIFS* seconds following the first frame. At the reception of each frame, a small amount of time is needed to process the frame before it is inserted into the buffer. Since the amount of processing time is in a same scale

comparing to the *AIFS*, the calculation is further simplified by assuming all the requested frames are received, processed and placed in the buffer ($request_0$ x *AIFS)* seconds after the arrival of the first frame.

As shown in Figure 4-2 (bottom), $time_1$ marks the point in time at which the last frame of the group that is sent in response to the previous request, arrives at the buffer. The equation for $time_1$ is shown as follows.

$$time_1 = time_0 + ART + (request_0 \times AIFS) \qquad (4.2.3)$$

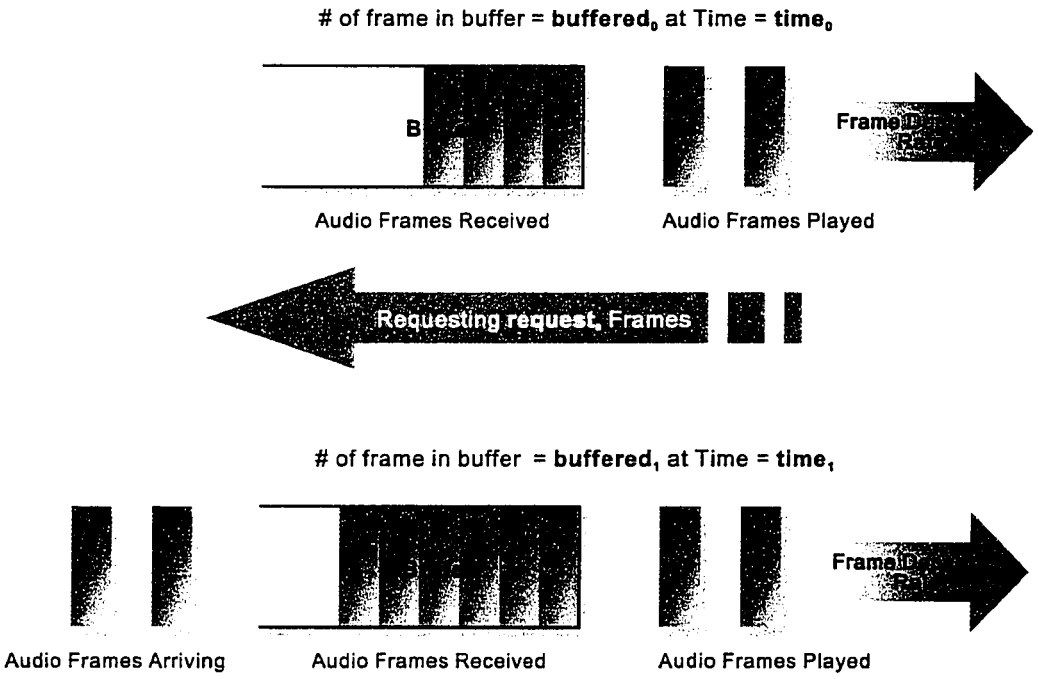

Figure 4-2 - The number of frames stored in the buffer at $time_0$ (top) and $time_1$ (bottom)

The decoder needs to be fed with a frame every *FDP* to maintain a steady flow. In

any arbitrary period of ΔT, the decoder needs $\Delta T/FDP$ number of frames. In other word, $\Delta T/FDP$ number of frames must leave the buffer within the time period in order to satisfy the real-time requirement of the decoder. In the time period between $time_0$ and $time_1$, $(time_1 - time_0)/FDP$ number of frames would leave the buffer.

The previous assumption states that at $time_0$, there is $buffered_0$ number of frames in the buffer. There is an influx $request_0$ number of frames and a departure of $(time_1 - time_0)/FDP$ number of frames occurring in the time period. As the result, $buffered_1$ which indicates the number of frames stored in the buffer at $time_1$ could be represented by the following equation.

$$buffered_1 = buffered_0 - \frac{time_1 - time_0}{FDP} + request_0 \qquad (4.2.4)$$

Substituting equation 4.2.3,

$$buffered_1 = buffered_0 - \frac{ART + (request_0 \times AIFS)}{FDP} + request_0 \qquad (4.2.5)$$

Rearranging equation 4.2.4, we get

$$buffered_1 = buffered_0 - \frac{ART}{FDP} - \frac{request_0 \times AIFS}{FDP} + request_0 \qquad (4.2.6)$$

Isolating $request_0$ to one side of the equation,

$$request_0 = \frac{FDP \times (buffered_1 - buffered_0) + ART}{FDP - AIFS} \qquad (4.2.7)$$

Generalizing 4.2.7 and the formula is obtained as follows.

$$request_i = \frac{FDP \times (buffered_{i+1} - buffered_i) + ART}{FDP - AIFS} \qquad (4.2.8)$$

The variable $buffered_{i+1}$ in the formula could be interpreted as the projected or preferred number of frames stored in the buffer at $time_{i+1}$. This number is particularly important as it represents the preferred number of frames to be stored in the buffer and it should be assigned with a value that is considerably greater than the buffer's lower threshold according the *ART*. When the decoder makes a request, the first frame will have to wait for approximately one *ART* before the first frame comes in. Hence, the lower threshold of the buffer represents the minimum number of frames needed to sustain a continuous flow to the decoder without the influx of additional frames during the waiting period. For that reason, the lower threshold is simply equal to the *ART* divided by the *FDP*. Hence, equation 4.2.8 has become the final equation 4.2.9 as shown below. The implementation of the JPlayer Audio Decoder sets the preferred buffer threshold to twice of the minimum buffer threshold.

$$\boxed{request_i = \frac{FDP \times (buffer_threshold - buffered_i) + ART}{FDP - AIFS}} \qquad (4.2.9)$$

Since the frame departing period is a constant, the final formula for $request_i$ is a function of the *FDP*, the *ART*, the *AIFS*, and $buffered_i$, the number of frames stored in the buffer.

Note that the bottom of the fraction is the main factor in the formula that affects $request_i$. The denominator signifies that the number of frames the decoder ought to request is inversely proportional to the difference between the *FDP* and the *AIFS*.

The subtraction in the denominator signifies that the *AIFS* must be less than the *FDP*. If this fundamental requirement is not met, the continuous flow to the decoder would be disturbed. Since the *FDP* is a constant, the subtraction also suggests that the larger the *AIFS* is, the greater number of frames needs to be requested to maintain the uninterrupted flow.

4.3 Four Major Building Blocks

The JPlayer Audio Decoder is composed of four main components: the session controller, the decoder, the buffer and the scheduler. The session controller controls and maintains the connection with the server; the decoder translates encoded bytes to audio data; the buffer holds audio data to be played; the scheduler feeds audio data to the audio player in a timely manner. Figure 4-3 illustrates the interactions of the components.

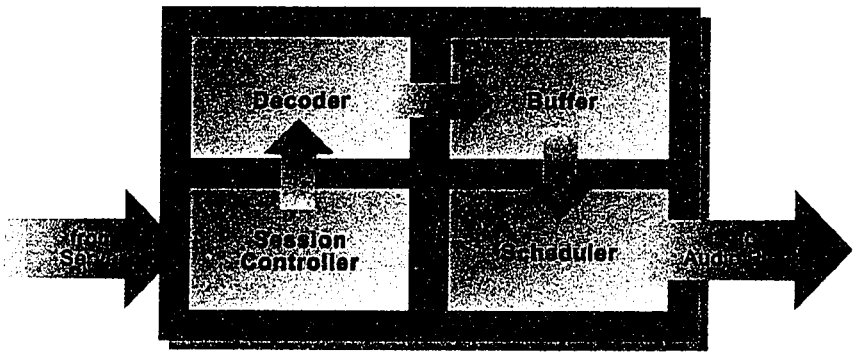

Figure 4-3 - The four main components of the JPlayer Audio Decoder

4.3.1 The Session Controller

The main function of the session controller is to provide the liaison between The JPlayer Audio Server and the JPlayer Audio Decoder. The session controller has a state machine to store the state information of the connection. The session controller has three states: SESSION_CLOSED is the initial state which indicates that the connection with the server has not been setup; SESSION_OPENING is the state that specifies the connection is being opened; as the name suggests, SESSION_OPENED is used when the connection with the server has been successfully established. The transitional diagram of the session controller is shown in Figure 4-4.

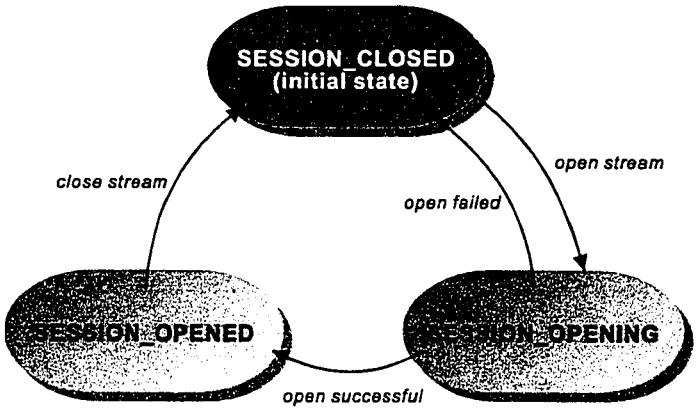

Figure 4-4 - The state diagram of the session controller

Between every pair of client and server, there exists a set of signals or a protocol in order to communicate. The protocol agreed by the JPlayer Audio Decoder and the server has ten operations. They are defined in the `StreamTransaction.class`. These operations are listed as follows.

```
public final static int QUERY = 0;
public final static int OPENSTREAM = 1;
public final static int CLOSESTREAM = 2;
public final static int REQUEST = 3;
public final static int KEEPALIVE = 4;
public final static int QUERYREPLY = 5;
public final static int OPENSTREAMREPLY = 6;
public final static int REQUESTREPLY = 7;
public final static int KEEPALIVEREPLY = 8;
public final static int DATA = 9;
public final static int ERROR = 10;
```

In Figure 4-5 on the next page, the exchange of operations between the JPlayer

Audio Decoder and Server is illustrated.

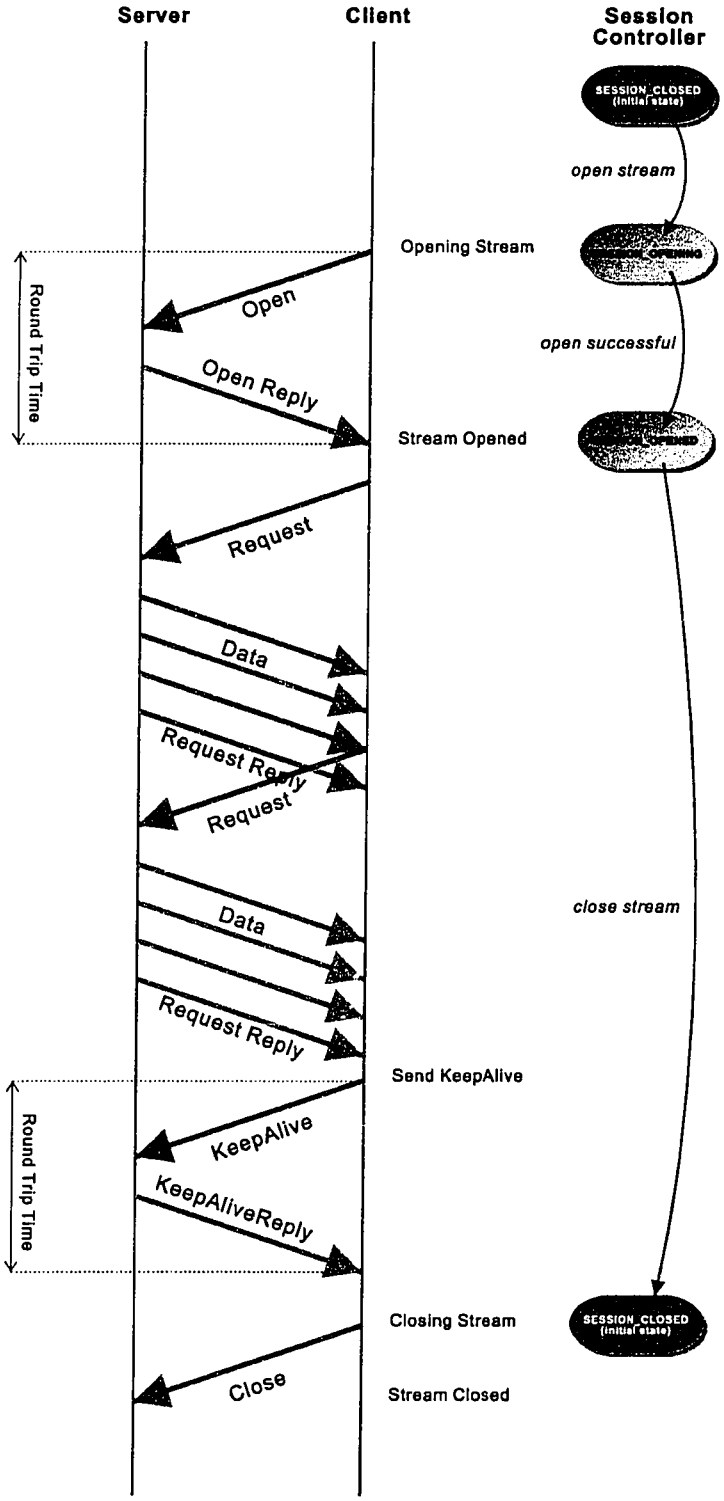

Figure 4-5 - The exchange of operations between the JPlayer Audio Decoder and Server

When the client wishes to establish an audio stream with the server, the client first transmits an OPEN operation to the server. The server reacts by issuing an OPEN REPLY to indicate a stream has been successful set up. The client controls the flow of the streaming content by sending the REQUEST operation. The REQUEST allows the client to specify the number of frames requesting so as to avoid buffer overflows. The server sends encapsulated audio data in frames of a configurable size to suit the underlying network so as to provide the most efficient use of the network.

The server is designed to terminate any connections that are idle for more than three minutes. This safety feature provides necessary robustness to ensure problems occur on the client side do not affect the reliable operation of the server. To avoid the accidental cut-off by the server, the client sends a KEEP ALIVE operation if the connection should become idle for more than two minutes. The client sends a CLOSE operation when the streaming connection is no longer needed.

4.3.2 The Decoder

The decoder component of the JPlayer Audio Decoder is based on a G.726 decoder. The decoder performs as one independent component. It obtains G.726 encoded frames from the filter buffer, decodes them using the G.726 decoder, and places the decoded frames in

the buffer. The JPlayer Audio Decoder only implements the G.726 decoder but is not limited to G.726. The design of the decoder component is modular- it can be fitted with decoders of other codecs to transcode other formats.

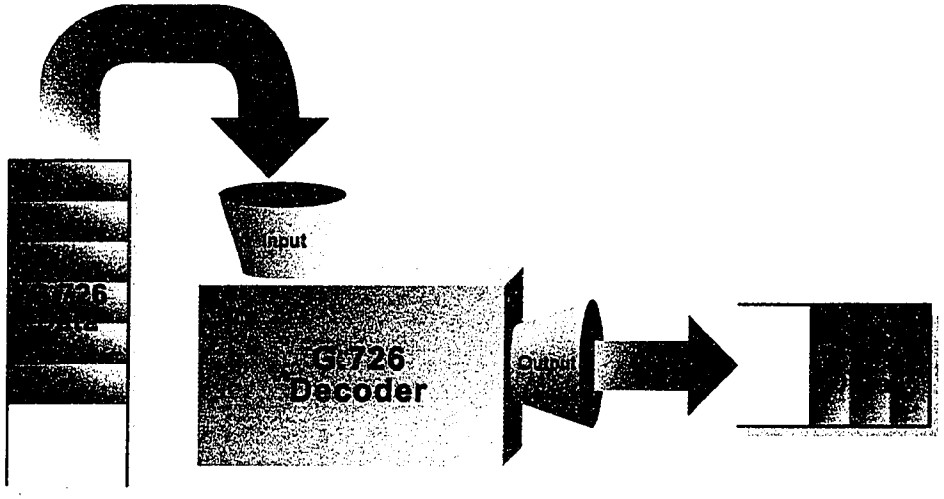

Figure 4-6 - The operation of the G.726 decoder

4.3.3 The Buffer

The buffer is the simplest of all components in the JPlayer Audio Decoder. It is the only passive part which acts as a mean of storage and does not manipulate data. It provides sanity checking and arranges decoded data into segments of proper size for the scheduler. Under normal circumstance, when the decoder passes a frame to the buffer, the timestamp of the frame is checked to ensure the validity of the data. Then the data are arranged into

segments in the size of one frame departure period and placed into the queue ready to be served. When the frame departure period elapses, one audio frame is popped from the queue. The behaviour of the buffer is illustrated in Figure 4-7.

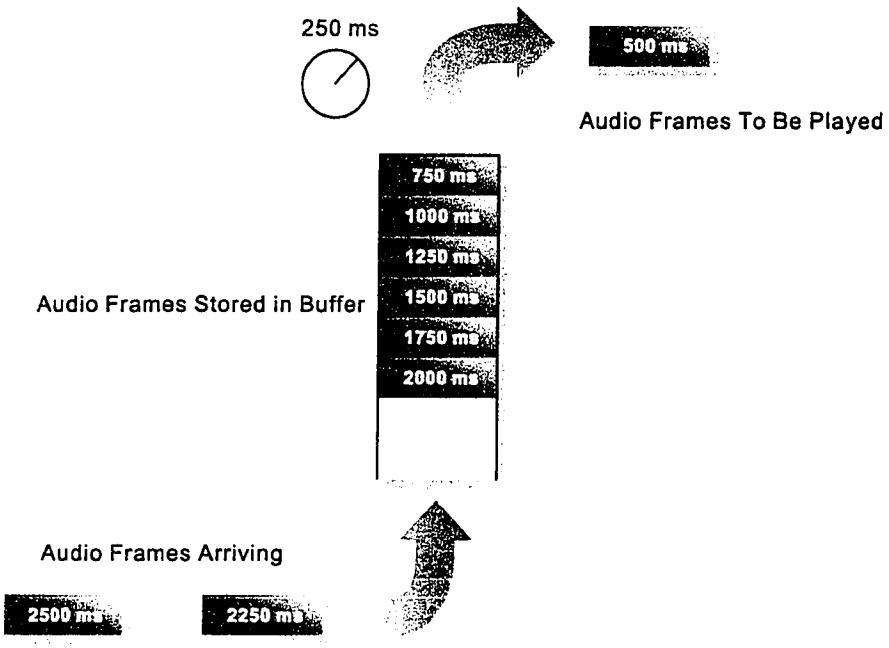

Figure 4-7 - The flow of frames into and out of the buffer

Internally, the buffer has a state machine that indicates the status of its operation. The buffer has three states. They are: BUFFER_INACTIVE which is the initial state that indicates the inactivity of the buffer; BUFFER_BUFFERING signifies that the frames are in short supply. Hence, the JPlayer Audio Decoder is paused to wait for more frames to arrive; BUFFER_IN_EQUILIBRIUM is set when the rate of frame arrival and departure

are balanced.

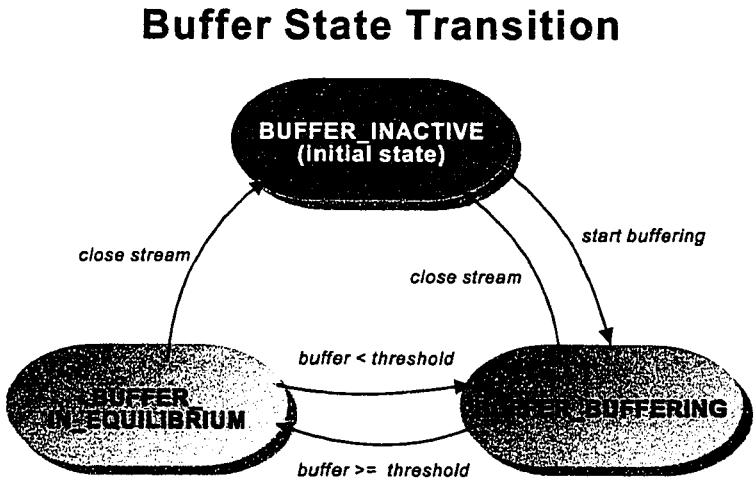

Figure 4-8 - The state diagram of the buffer

Under normal condition, the buffer state is in the BUFFER_INACTIVE state when the JPlayer Audio Decoder is idle. When the JPlayer Audio Decoder starts requesting frames from the server, the state jumps to BUFFER_BUFFERING and stays until the number of frames arriving is more than the threshold. The JPlayer Audio Decoder begins feeding frames to the audio player once the threshold is reached and the flows into and out of the buffer is said to be in equilibrium, hence this gives the name BUFFER_IN_EQUILIBRIUM.

4.3.4 The Scheduler

The scheduler of the JPlayer Audio Decoder manages the pace at which audio frames are sent to the audio player and provides a time reference to the video controller. The scheduler is complex, partly because of the requirement of precise timing and partly to provide auxiliary functions such as pausing and delaying. Much as the buffer and session controller have their own state machine, the scheduler too has its own. The scheduler has a total of six distinct states. They are SCHEDULER_INACTIVE, SCHEDULER_STOPPED, SCHEDULER_PLAYING, SCHEDULER_PAUSED, SCHEDULER_DELAY, and SCHEDULER_FINISH_SERVING.

The SCHEDULER_INACTIVE is the initial state. When the JPlayer Audio Decoder opens a streaming session and waits for audio data to arrive, the schedule is put in the SCHEDULER_STOPPED state. The SCHEDULER_PLAYING, SCHEDULER_PAUSED states are set when user commands for play and pause are received respectively. The video controller might request for a delay if it is very behind in scheduler and fails to catch up. Hence, this creates the need for the SCHEDULER_DELAY state. Due to the slight delay between the time a frame is fed to the audio player and the time the frame is heard, the SCHEDULER_FINISH_SERVING state is only set when all the audio stream has reached the end but the audio player has

not yet finished player. This is necessary in order to provide timing information.

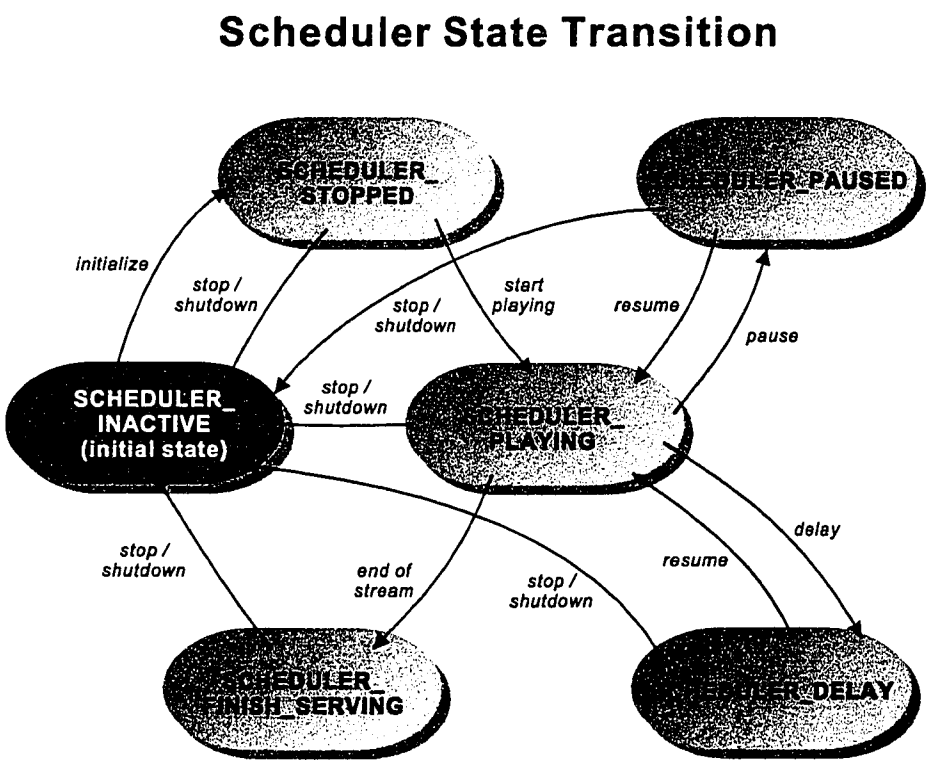

Figure 4-9 - The state diagram of the scheduler

The scheduler is run as an independent component separated from other parts of the decoder. Its primary duty is to serve decoded audio frames of a fixed size to the audio player at a constant rate. The precise time tracking of the audio codec is critical for it also serves as a time reference to the video counter part. The code of the scheduler is fairly complex- the scheduler constantly adjusts itself against the system clock. This is vital as a small misalignment could result in a huge accumulated time difference between the audio

and video controller. A sample of the pseudo code is presented as follow.

```
1   while (!endOfStream & !exit) {
2      // break the loop if endOfStream is reached or exit issued
3      // send a frame to sun.audio.AudioPlayer
4     referenceTime = currentTime() - startTime;
5      adjustment = referenceTime - audioPlayerPosition;
6      sleepTime = frameDeparturePeriod - adjustment;
7      if (sleepTime > 0) {
8         // sleep for sleepTime
9         sleep(sleepTime);
10     }
11     // increment audioPlayerPosition
12     audioPlayerPosition += frameDeparturePeriod;
13  }
```

The gut of the audio media controller lies within a while loop. It is repeated until one of the terminating conditions occurs- either the audio stream has reached the end or an exit instruction has been issued. In the beginning of the while loop (line 1), a frame is sent to the `sun.audio.AudioPlayer` to be played. The `referenceTime` is extracted by comparing against the system time (line 4) and the little `adjustment` to be made is found by comparing the `referenceTime` to the `audioPlayerPosition` (line 5) which holds the position of the last played frame. The amount of time the thread should wait until playing another frame is the `interDepartureTime` minus `adjustment` (line 6). The thread then goes into sleep for the amount of time calculated (line 9). The player position is incremented (line 12) and the thread repeats itself as it jumps back to the beginning of the

while loop.

The actual code of the audio controller is by far more complicated than what has been shown above. Apart from serving frames at a fixed interval, other features such as pausing, skipping of a frame and buffering are also incorporated into the controller. But the discussions on these features are not within the scope of this paper due to the complexities of these features.

4.4 Network Monitoring

The JPlayer Audio Decoder is designed to provide streaming over various networks for a variety of platforms. As discussed in the previous section, the differences of network performance of different network environments could be huge and the processing power of each platform is different. In order to achieve the best performance out of every environment, the JPlayer Audio Decoder must collect statistics of the network and the platform. The JPlayer Audio Decoder, like other streaming applications, is particularly sensitive to the response of the network, namely the round-trip time between the client and the server.

4.4.1 The Average Round-trip Time

Under most circumstances, the server is positioned at a fixed site and the link connected to the server is rarely switched. The clients, however, could be connected via links with a wide range of performance. Therefore, it is most effective to have each client measure the round trip time of the link that lies between itself and the server. Since there are several exchanges of messages between every client and the server, the format of the headers of each message could be easily enhanced to provide additional network performance data collection.

An 8-byte timestamp field is added to the header of the message. This timestamp field is set by the client at the time when the message is sent to the server and the timestamp is only relevant to the client. Upon reception of the message, the server processes the message according to the nature of the message and sends a reply message back to the client. The reply message also has a timestamp field which is assigned with the value of the timestamp of incoming messages. When the reply message arrives at the client side, the client compares the timestamp on the reply message which carries the time at which the first message was sent and the current system time. The difference is the round trip time for the previous pair of messages exchanged. Due to the fluctuating nature of the performance of IP networks, one measure at one instance could be very different from another. Hence, an average of round trip time is taken to reflect the typical round

trip time between the server and the client. The implementation of the JPlayer Audio Decoder records the ten most recent measurements of the round-trip time and averages them to obtain the estimate of network responsiveness at the moment.

4.4.2 The Average Interframe Separation

The average round-trip time is a piece of key information in evaluating the current network responsiveness. The other property, the inter-frame separation, is also critical to assess the combined effect of the throughput of the link and the processing power of the client machine. Generally, the client requests a couple of frames at a time. Every single frame is transmitted, received, processed and decoded, before it is ready to be played. Albeit, the frames arrive as a group, and there exist unavoidable little gaps between the times the frames become available. The size of the gaps, which is dependent on the network characteristic and the process power of the client machine to manipulate data, is a measure of determining how the client reacts to the environment. The larger the gaps are, the longer time the client will expect the data to become available. Much as the round-trip time changes over time, the size of the gaps or the interframe separation is also a time dependent variable. The JPlayer Audio Decoder collects the last ten most recent measurements of the interframe separation and computes an average value.

Chapter 5 - Implementation Challenges

5.1 TCP vs. UDP

A typical real-time network application requires data to be received in a timely manner. For this very reason, most IP applications choose UDP to be the data transport technology. Unlike its connection-oriented rival, TCP, the connectionless UDP has no sequence numbers nor does it have checksum to ensure data validity, thereby saving a fraction of the processing cycles for each packet. Although it is proven to be very useful in other network applications, the TCP automatic retransmission functionality might not be preferable in real-time communication systems. When a packet is lost in its course from the source to destination, the destination does not detect its absence until the sliding window expires. (The sliding window timeout is usually set as the average roundtrip time plus four times of the standard deviation of the roundtrip time.) Upon detection of a lost packet, the destination requests for a retransmission. Chances are the retransmitted packet finally arrives well after the time-sensitive data are needed and the data becomes nothing more than a sequence of meaningless bytes.

However, the advantages that UDP enjoys do come at a cost. IP delivery allows packets to travel via different paths. Without the re-ordering and automatic retransmission

capabilities that TCP has, UDP packets could arrive out of order or in some scenarios, some packets get lost before reaching the destination and the destination would have no knowledge that those lost packets were sent.

Being too sophisticated to be preferable for transporting real-time data streams, TCP does have its place in real-time communication. Often the control messages are sent with TCP in today's IP applications for its reliable, in-order transmission.

In spite of the fact that most real-time network applications utilize UDP for transporting data and TCP for control messages, in the JPlayer Audio Decoder, TCP is chosen as the transport mechanism for carrying both data and control packets instead. It is selected due to two reasons. The first of the two is the complexity of implementing a reliable UDP transport in Java. Several attempts have been made in constructing such transport. The result is unsatisfactory- packets are often dropped when the CPU utilization is high. In the end, utilizing TCP as a work-around seemed to be a wise choice giving the tight time frame of the project. It should be noted that, however, in applications that have stricter real-time requirements, UDP transport could be implemented and used for lower overhead. This thesis, being a prove-of-concept-type project at the time of writing, sufficiently proves the concept of a real-time multimedia decoder using TCP.

The second reason is the simplicity of the transport layer gained by combining the

data and control transport. Since both data and control transport requires a connection between the server and the client, it seems logical to combine the two channels into one for less transport management. The processing of data and control packets must be separated to avoid head of line blocking or starvation of either transport due to flooding of the other.

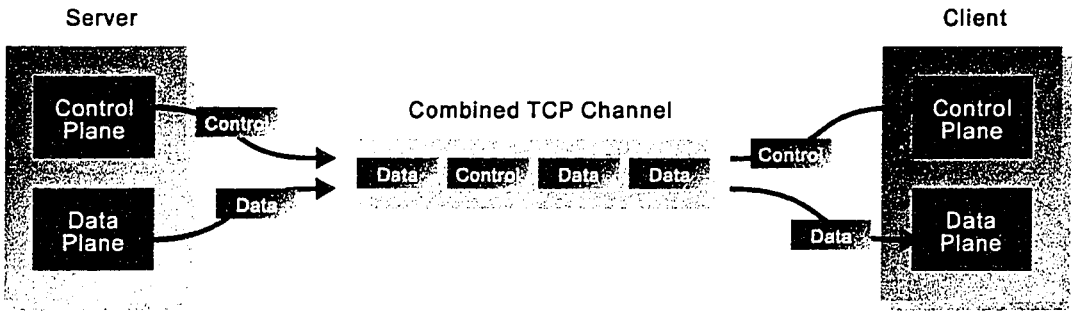

Figure 5-1 - The combined TCP Channel

5.2 The Multi-thread Approach

The JPlayer Audio Decoder has a number of components, but not all components work at the same rate and have the same timing requirement. The request algorithm has to send out frame requests from time to time to ensure an uninterrupted stream; the TCP receiver must constantly monitor the incoming packets to extract control messages; the filter processes the incoming encoded frames without much delay; and lastly, the scheduler

must wake up precisely every 250 ms to feed the audio player. To achieve good performance, components must be able to work independently. Hence, more than one thread is needed. The JPlayer Audio Decoder employs threads for each of the individual components.

5.2.1 Priority Tuning

The Java language allows each thread to be given a priority. Threads with higher priority are to be executed before the execution of the ones with lower priority. This is a key feature of any high performance computer language. However, a multi-thread program without proper tuning is destined to suffer from one of the classical multi-thread problems- starvation of resource. At the early stage of the implementation phase, the JPlayer Audio Decoder achieved sub-optimal performance. To identify what and where the problem lays, another thread is put in place to collect statistics of each component periodically. With the help of the analysing tool, the bottle neck was located before the scheduler. It was identified that the scheduler has been assigned with a priority that is too high. It has occupied most of the CPU cycles but has nothing to work with, while the other components have insufficient resource to work their own share. With careful tuning of the priorities of the threads, the JPlayer Audio Decoder achieved better performance

significantly.

5.2.2 Critical Regions

One of the disadvantages of the multiple-thread approach is the possibility of data integrity infringement. Under a single-thread scenario, only one thread can change the values of any variables within the scope of the thread. However, this is not the case for the multiple-thread approach- all threads are able to alter the values of variables within the scope. This property is convenient but possibly disastrous. One thread could rely on the consistency of the value of some variables while another thread modifies it. One solution to prevent this is the declarations of critical regions in which only one thread is allowed to enter at any given time. Java associates a lock with any objects or methods. Access to an object or a method could be made mutually exclusive by using the `synchronized` statement. Thus, data integrity is assured. The `synchronized` keyword is used only if it is absolutely necessary since a protected region takes five times longer to execute than its unprotected counterpart. (See Table 5-4 for details.)

5.3 Issues with Applets

A sample applet has been coded to serve as a demonstration in using the JPlayer as a web

component. Over the course of implementing the sample applet, a few problems have been encountered. This section of the thesis documents these issues to permit future usage of the JPlayer in the web environment to be implemented easily.

5.3.1 Applet Behaviors in Netscape and Internet Explorer

The `Applet` class of the standard Java implementation has four methods that deal with applet-browser interaction. They are `init()`, `start()`, `stop()`, and `destroy()`. The brief description of each method and its intended usage are outlined in the following table.

Methods	Details
`public void init()`	Called by the browser or applet viewer to inform this applet that it has been loaded into the system. It is always called before the first time that the start method is called.
`public void start()`	Called by the browser or applet viewer to inform this applet that it should start its execution. It is called after the init method and each time the applet is revisited in a Web page.
`public void stop()`	Called by the browser or applet viewer to inform this applet that it should stop its execution. It is called when the Web page that contains this applet has been replaced by another page, and also just before the applet is to be destroyed.
`public void destroy()`	Called by the browser or applet viewer to inform this applet that it is being reclaimed and that it should destroy any resources that it has allocated. The stop method will always be called before destroy.

Table 5-1 - The methods for applet-browser interaction and their descriptions

As a general rule, `init()` and `start()` methods are invoked when a browser loads an applet; `stop()` and `destroy()` methods are called when the browser exits or another web page is loaded. There are, however, substantial differences in how Netscape and Internet Explorer handle the life cycle of an Applet. On reload under Netscape, rather than reloading the applet as the name implies, stop() and start() methods are invoked. Refreshing under Internet Explorer forces the applet to go through a full death and rebirth by invoking `stop()`, `destroy()`, `init()`, and finally `start()`. The resizing of the browser window is also handled differently under the two browsers. Netscape treats resizing the same way as a reload, whereas it is ignored by Internet Explorer. The differences in behaviour of the two browsers are listed in Table 5-2.

Action	Methods invoked under Netscape	Methods invoked under Internet Explorer
Load	`init(), start()`	`init(), start()`
Exit	`stop(), destroy()`	`stop(), destroy()`
reload/ refresh	`stop(), start()`	`stop(), destroy(), init(), start()`
Resize	`stop(), start()`	none

Table 5-2 - Methods invoked under different browsers

Due to these differences in behaviour, users would undergo different experience under different browsers. After careful consideration, it is decided that all the

initializations of the applet are performed in the `init()` method and all the de-allocations and other actions necessary to destroy resources are placed in the `destroy()` method so as to minimize the differences in behaviour under different browsers. Even so, reload/refresh is processed differently- Netscape ignores it, but it triggers Internet Explorer to terminate and reinitialize the applet.

5.3.2 Performance and Java Version

Apart from the fact that each browser behaves differently, it is worth to note that each runtime environment also performs slightly differently. As one would expect, the Java codec performs the best when it is run under the standard Java Runtime Environment as an application. Internet Explorer 5 which features Java 1.1.3 executes with acceptable performance for an applet. Although Netscape 4.79 incorporates a Java 1.1.5 virtual machine, it has the worst performance of all. Starvation of threads is most noticeable under Netscape if the threads are not carefully engineered.

5.3.3 Security Restrictions of Applets

Running Java code in the form of an applet is bound with additional constraints. Java Virtual Machine in the browsers only allows network connection made to the web page

serving machine so as to prevent hazardous applets from communicating with unauthorized hosts. While this is a critical feature to ensure security of the client machine is not at risk, it poses additional work for implementing the JPlayer in the form of an applet. The security feature is very nit-picking at the format of the universal resource location (URL) of the server. The host portion of the URL must be exact as the URL is fed to the browser. All other URLs and IP addresses that point to the same location are treated as unauthorized host and will be rejected. The host name of the server must be obtained by the `getCodeBase().getHost()` method of the `Applet.class`.

5.4 Playing Audio with Java

The standard Java Runtime Environment 1.1 provides no straightforward way to play an audio stream. The only offering is by playing audio files using the `play(URL)` method in the `Applet` class. Although Sun Micro Systems has introduced the Java Media Framework starting from Java 1.2, which gives programmers the flexibility to carry out audio and video stream capturing, playback, as well as stream manipulation, the unpopularity of Java 1.2, especially amongst the hand held devices, has raised a big concern. An alternative must be found.

5.4.1 The sun.audio Package

After doing some research and code tracing, it was found that audio data stream could be played by using the `sun.audio.AudioPlayer` class. Sun did not make this class available in the form of source file, but every Java 1.1 compliant runtime environment has it implemented. Due to the lack of documentation to the `sun.audio` package, a great deal of effort has been put into discovering the usage of this class.

The source code of `sun.audio.AudioPlayer` is not publicly available for it is a platform dependent class. It is implemented differently on every platform since it interacts directly with native code to play raw audio streams. The `AudioPlayer` class has two public methods made available to the outer classes. They are:

```
public synchronized void start(InputStream inputstream);
```
and

```
public synchronized void stop(InputStream inputstream);
```

Audio data could be wrapped by an `InputStream` or its subclass and supplied to the `start` method to be played. The stream is stopped if the `InputStream` reaches the end or it could be stopped by calling the `stop` method.

5.4.2 The .AU File Header

The `sun.audio.AudioPlayer` has also been implemented under the standard Java 1.2

Runtime Environment. When the Java audio codec was tested on Java 1.2 Runtime Environment, it failed to play any audio streams. It was revealed later that the Java 1.2 implementation of `sun.audio.AudioPlayer` could only play file streams of well-known audio file types such as `.WAV` or `.AU`. A work-around to the problem is to feed an `.AU` file header to the player to make the audio stream look like an `.AU` file.

`.AU` is the standard audio file format for the UNIX community. Figure 5-2 shows the structure of an `.AU` file header. An `.AU` file header starts with a magic number of `0x2e736e64` which uniquely identifies the file type, followed by header size, data size, encoding, sample rate, the number of channels, and finally audio data. The encoding field is set to 1 to indicate the data are encoded in 8 bit u-law. The sample Rate is set to 8000 for an 8000Hz sample. Data size is set to -1 for unknown file size.

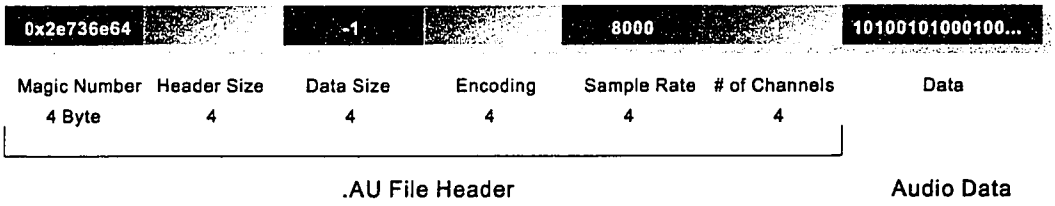

Figure 5-2 - The `.AU` file header

The `.AU` wrapper method works well for applications under Java 1.2 runtime environment but fails under Netscape6 which incorporates a Java 1.3 environment- a

security exception was thrown whenever the classes from the `sun.audio` package were accessed. An alternative is needed in order to play audio stream under Java 1.3 environments.

5.4.3 The Java Sound API

The Java Sound API has become a part of the Java Standard Edition, since the introduction of Java 1.3. Java Sound is designed to enhance the audio support of the `sun.audio` package. It provides low-level support for audio operations such as audio playback and capture (recording), mixing, MIDI sequencing, and MIDI synthesis in a framework that promotes extensibility and flexibility.

Sampled audio data can be played with the `SourceDataLine` class of the `javax.sound.sampled` package. The `SourceDataLine` class has two methods. They are listed as follows.

```
public void open(AudioFormat format, int bufferSize);
```
and

```
public int write(byte[] b, int off, int len);
```

The `open` method opens the line with the specified format and suggested buffer size, causing the line to acquire any required system resources and become operational. The

`write` method alternatively feeds audio data to the buffer waiting for its time to be played.

Table 5-3 summarizes the availability of each method on various versions of Java runtime environments.

Java Version	`sun.audio.AudioPlayer`	`Sun.audio.AudioPlayer` with `.AU` header	Java Sound API
1.1	available	Not needed	Not available
1.2	No sound output	available	Available only as a downloadable module
1.3	Security violation under Netscape	Security violation under Netscape	available

Table 5-3 - The availability of audio players on Java runtime environment of different versions

5.4.4 Identifying Java Version

The next challenge is to find out the running version of Java Virtual Machine in order to supply the JPlayer Audio Decoder with the appropriate audio player. The version number of the running Java environment could be extracted by executing `System.getProperty("java.version")` which returns a string containing the version number. For example, `"1.1.8"` is returned when the code is executed under the standard Java 1.1.8 Runtime Environment. This string could be compared to determine the version

number of the current running Java runtime environment. Any version number that starts with 1.1 and below indicates the running Java environment should play audio streams using the mechanism provided by `sun.audio.AudioPlayer.class`, Version number starting with 1.2 should use the `sun.audio.AudioPlayer.class` with the `.AU` file header. All other newer version should use the Java Sound API.

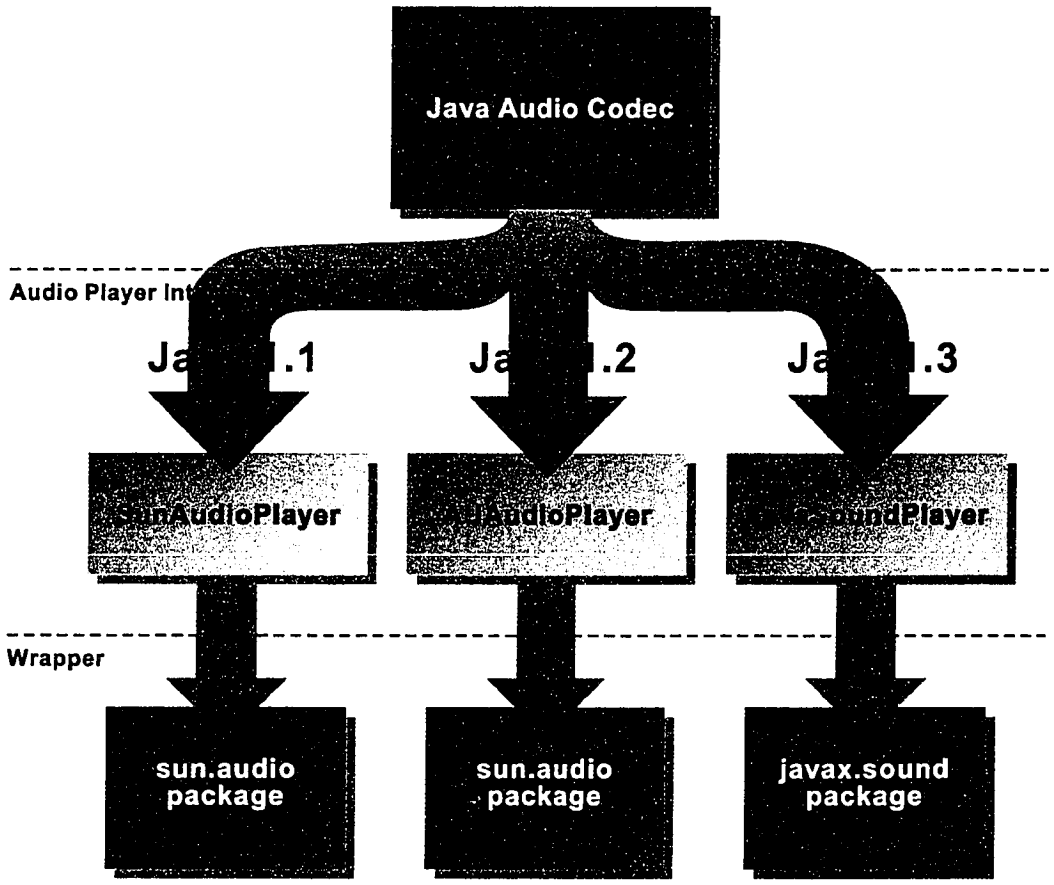

Figure 5-3 - Selecting the right audio player

5.5 Audio Interruption Handling

Often when the congestions occur in the IP network, the data are unable to be delivered to the client in time. This could result in interruptions of streaming experience in the case of multimedia applications. When audio data are not ready to be played or the decoder is paused, the JPlayer Audio Decoder must react appropriately. There are three approaches in handling such scenario. The first being the simplest of all, is to feed the audio player with silence. The second approach which is very similar to the first one is to repeatedly feed the player with the last chunk of data fed until new data becomes available. The last approach involves blocking of the audio player if audio data are temporarily unavailable. The section will discuss the benefits and the short-comings of each approach.

5.5.1 Approach 1: Silence Filling

In an 8 bit digitized audio sample, a range from 0 to 255 is used to describe the wave form of the audio stream at a particular instance in time. The data are usually captured in 2's compliment form with the range from 1 to 127 representing positive number, and the range from 128 to 255 representing the negatives. Silence in this case is represented by 127. When audio data are not available to be played, instead of triggering an end of stream notification, one approach is to feed silence to the player so that the player has

something to work with. The user under this scenario would only experience silence. The silence feeding approach is easy to implement. However, keeping track of timing precisely could be a problem due the small, indeterministic processing time the player has as previously stated.

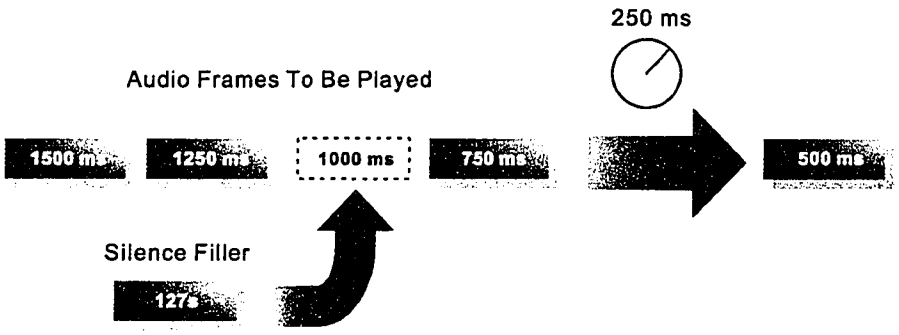

Figure 5-4 - Feeding a silence frame

5.5.2 Approach 2: Repetition of Audio Data

The second approach is similar to the first. When the data stream could not catch up with the player, the last 100 bytes that was fed to the player would be supplied again. Under this scenario, users would hardly notice the difference if the repetition lasts a very short period of time before the new data arrives. The effect of this approach could become annoying if data remain unavailable for a prolonged period. The complexity of the approach is moderate. As with the first, this approach also suffers from time tracking

issues.

5.5.3 Approach 3: Blocking the Audio Player

The third and last, being the most effective approach of all, is to pause the audio player whenever audio data become unavailable. The `sun.audio.AudioPlayer` class provides no method to pause the audio player temporarily. The trick to achieve this is to block the thread that polls the `InputStream` for data. The source code of the `ByteArrayInputStream` class has been modified so it blocks and waits if there are no data available. Time tracking of the audio player is possible with this approach since the player is temporary halted, which makes this approach a superior one.

The only problem that exists with the pause approach is that the audio player could not be shut down properly after the end of stream has been reached. The thread would be halted when the `stop` method of the `sun.audio.AudioPlayer` class is invoked. After some experience with the code, it was found that a blend of the first and the third approaches is needed to address the issue. The final solution blocks the audio player, if no data are available, but feeds silence after the end of stream has been reached. Feeding silence after the end of stream does not affect time tracking in this case since time tracking is of no concern after the audio stream has come to an end.

5.6 Optimization for Speed

The Java language, being an interpreted language, is expected to perform not as fast as compiled languages such as C. However, with proper optimization, Java code could run significantly faster. The results of a study have shown that Java code can be optimized in several ways. [11] The study summarizes the time in microseconds to execute various Java operations on a Pentium 75 (Intel Pentium 75 MHz with 16MB RAM, 256kB L2 cache, running Windows 95) under 3 Java implementations. (Sun JDK version 1.0.2, Netscape 3.0 & Internet Explorer 3.0)

5.6.1 Objects Allocation

The study is a bit outdated but it serves the purpose to prove that object creation is an expensive operation. The study has shown that object creation takes 9 to 26 ms depending of the Java virtual machine. Therefore, object creation should be minimized and objects should be reused whenever possible; most importantly, object creation should be avoided during the execution of any real-time applications. The JPlayer Audio Decoder allocates and initializes all the arrays, threads and common objects at start up. Most arrays and threads are also recycled to further reduce interruption at runtime.

5.6.2 Integer vs. Shorts

The study has also shown that the manipulation of integers is by far faster than that of bytes, shorts and longs. The benchmark results have shown that an increment of an integer on Internet Explorer takes 0.006 milliseconds to complete while an increment of a byte takes a magnitude longer. For this reason, most computations in the JPlayer Audio Decoder that involve shorts are converted to integers. This single optimization results in a significant 20% improvement in performance.

5.6.3 The Nagle's algorithm

The Nagle's algorithm is used to automatically concatenate a number of small buffer messages; this process, which is called nagling, increases the efficiency of a network application system by decreasing the number of packets that must be sent. Nagle's algorithm, defined in 1984 as Ford Aerospace and Communications Corporation Congestion Control in IP/TCP Internetworks (IETF RFC 896), was originally designed to relieve congestion for a private TCP/IP network operated by Ford, but has since been broadly deployed. [12]

Nagling has been broadly implemented across networks, including the Internet, and

is generally performed by default in many TCP/IP implementations including that of Java's. The main principle of the Nagle's algorithm is to trade network performance with bandwidth efficiency. This is sometimes considered to be undesirable in highly interactive environments. The JPlayer Audio Decoder which relies on the speedy delivery of packets exchanged between the client and the server to maintain an uninterrupted flow of streaming content, suffers from network performance degradation. Hence, nagling must be turned off through use of the TCP_NODELAY sockets option.

5.7 Audio/Video Synchronization

With no synchronization in place between the audio and video decoders, the time tracking of the two decoders could diverge given that each has its own scheduler. The next challenge is the merger with the H.263 video decoder while much of the work for the G.726 decoder had been completed. Some code is needed to provide the bridging between the G.726 audio decoder, the H.263 video decoder and the front-end interface. The glue code should provide two functions. Firstly, it should accept control information from the front-end and propagate them to both decoders. Secondly, it should facilitate the synchronization between the decoders. A wrapper class which is named `AVBridge` has been implemented to provide these functions. This is illustrated in Figure 5-5.

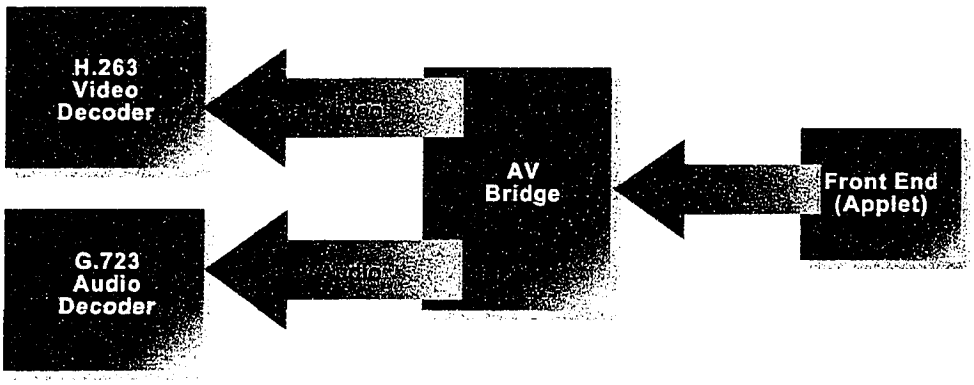

Figure 5-5 - The conveyance of command via the AVBridge

Apart from propagating commands issued by the front-end to both decoders, the `AVBridge` class, as the name implies, serves as a bridge between the audio and the video decoder. More specifically, timing signals are exchanged between the decoders so as to achieve synchronization.

The audio decoder is designed to keep a very accurate time tracking. Thus, it seems sensible for it to also serve as the master clock of both decoders. As shown in the following illustration, a heart beat is sent by the audio decoder every 250 milliseconds to the `AVBridge` class. Each heart beat contains the time stamp at which the audio decoder is currently positioned. The `AVBridge` class in turn passes the heart beat to the video decoder and the video decoder adjusts its timing according to the time stamp stored in the heart beat, thereby achieving synchronization.

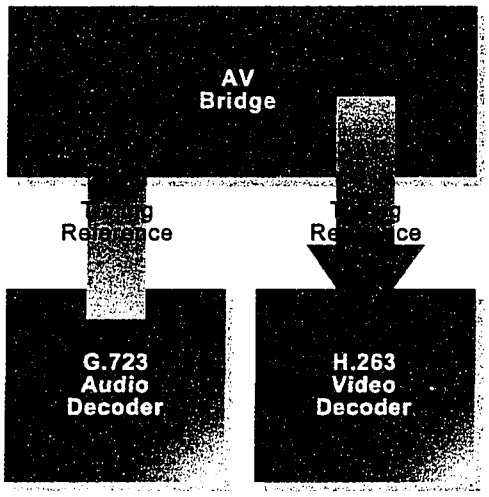

Figure 5-6 - Exchange of timing reference between the audio and video decoder

The H.263 video stream has the ability to store timing information in the stream itself. Under the original implementation, the video decoder compares the timing information with the internal clock so as to present the video stream at an accurate rate. This implementation does not allow timing adjustment base on external signalling, therefore synchronization with the audio stream cannot be achieved. Some code change was done to allow the entrainment with an external timing reference. The current design allows two modes of operation. If external timing reference is available to the video controller, the video controller would run the new timing implementation to adjust its timing against the reference. Otherwise, the video controller runs the old code under which time tracking is controlled by the internal clock only.

In order for the video decoder to adjust itself to be aligned with the external timing

reference, the video decoder must be capable to speed up or pace out the video decoding and representation. In a scenario where the video decoder is ahead of the external timing reference, the video decoder would put itself into sleep until the scheduled time for the next frame has elapsed. On the contrary, when the video decoder lags behind the external timing reference, the video decoder would reduce its work load by excluding some of the work and only perform the essentials in an attempt to catch up. In implementing the above scheme, the video controller is modified so that a boolean variable serving as a flag is used to indicate whether the video decoder is lagging behind. Every time a new H.263 video frame is decoded, the decoder checks the flag to determine if it should proceed with the process of colour space conversion and the rendering of the image on the display. The process of colour space conversion is very calculation intensive. It maps a frame in the YUV colour space to a RGB frame which can be displayed by Java. Skipping such process takes away a great portion of the workload of the video decoder, thereby saving some CPU cycles to allow the video decoder to catch up. Unlike the rendering process, the process to decode each of the H.263 video frames cannot be bypassed. This is because video frames could be dependent on the content of the previous frames. Hence, every frame must be individually decoded, although some frames could be dropped.

Chapter 6 – Test Cases and Results

6.1 Test Cases

Carrying out test cases is an essential phase of any development projects. Areas for improvement could often be found by analyzing the results of the test cases. In validating that the design requirements of the JPlayer are met, a series of test cases are run and performance statistics are collected for analysis purposes. The set of test cases is designed to evaluate the JPlayer quantitatively and qualitatively. The details of the test cases are outlined as follows.

6.1.1 The performance of The JPlayer Audio Decoder

The audio decoder streams an audio file of reasonable length, encoded in the G.726 format and decodes. The network characteristics including the average round-trip time and the average interframe separation are recorded. This test case is aimed at measuring the performance of the JPlayer Audio Decoder quantitatively. Figure 6-1 shows the JPlayer Audio Decoder playing an audio clip in G.726 encoding.

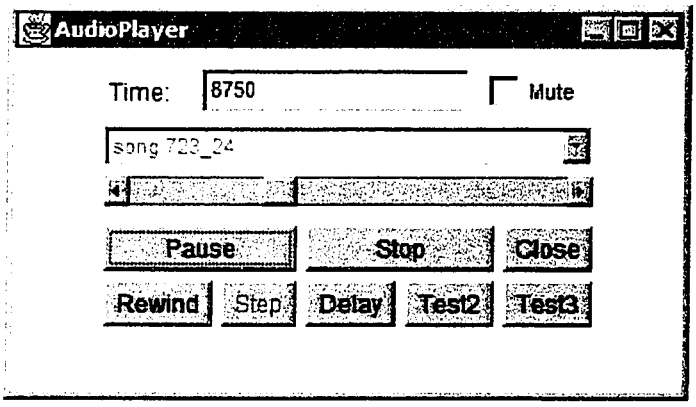

Figure 6-1 – The JPlayer Audio Decoder playing an audio clip in G.726 encoding

6.1.2 Synchronization of the Audio and the Video Decoders

An audio/video stream pair which features a man singing with the lips' movement clearly shown, is decoded using the JPlayer. Synchronization between the audio and the video decoders ensures the lips' movement is in sync with the audio track. A survey is filled by a group of testers to report the user experience. Each tester responds by choosing a grade from "Excellent", "Satisfactory", "Average", and "Poor" based upon the tester's perception with the JPlayer (audio and video). The testers are composed of 2 groups – one group with personal relationship with the author and the other group has no relationship with the author. The average temporal offset between the internal clocks of the audio and video decoder is also recoded. The goal of this test case is to reflect both the quality and the quantity aspects of synchronization.

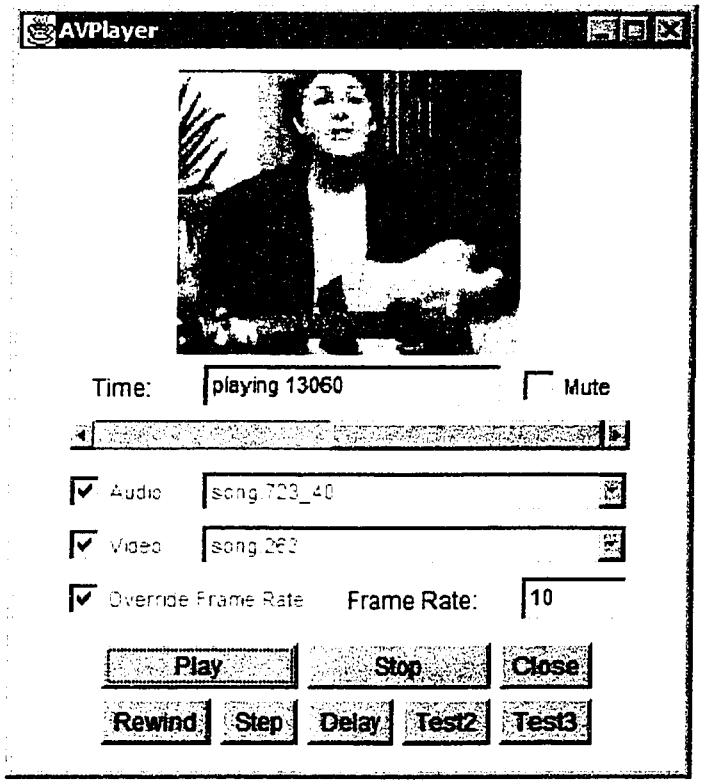

Figure 6-2 - The JPlayer showing a musical clip

6.1.3 CPU Utilization

One measure of merit of the efficiency of an application is to sample the CPU utilization of a machine. The efficiency could be measured by the CPU utilization of a machine which is the percentage of computing power that is consumed by the application at any given time. Scoring a low CPU utilization means the application is efficient and does not consume a lot of resources. Therefore, idle computing power can also be used to drive other applications. The CPU utilization on Windows machines can be shown in the Task

Manager window.

6.2 Test Environments and Reference Machines

The first test case that involves audio streaming is carried out in various network configurations representing the typical usages of the JPlayer. These configurations are listed as follow with the JPlayer Audio Decoder and the JPlayer Audio Server referred as the client and the server respectively.

- Both the client and the server are located on the same machine.
- Both the client and the server are located on the same LAN.
- Both the client and the server are connected to the Internet using high-speed broadband access.
- The server is connected to the Internet using high-speed broadband access; the client is connected using a 56k dial-up connection.
- The server is connected to the Internet using high-speed broadband access; the client is connected using a 28.8k dial-up connection.

With the exception of the first network configuration, the server machine comprises an Intel Pentium III 800 MHz processor, 512 Mb of RAM running Windows XP Professional with Java Runtime Environment 1.2.2. The client machine employs an Intel

Pentium III 650 MHz processor, 384 Mb of RAM running Windows XP Professional with Java Runtime Environment 1.1.8 and 1.3. Each test is executed on both Java Runtime Environments.

The second test case that demonstrates synchronization between the audio and the video decoders is carried out with both the client and the server located on the same machine.

The third test case which measures the average CPU utilizations is conducted on both server and client machines.

6.3 Test Results

The test results of the test cases are summarized in this section.

6.3.1 The Performance of the Audio Decoder

The audio decoder streams an audio file of 30 seconds, encoded in the G.726 format and decodes. The network characteristics including the average round-trip time and the average interframe separation are recorded in Table 6-1.

Network Configuration	Clip Encoding	Java Version	Round-trip Time (ms)			Interframe Separation (ms)		
			Low	High	Average	Low	High	Average
local	G.726 40kbps	1.1.8	262	328	**274**	12	14	**13**
		1.3	236	340	**289**	21	15	**15**
Group 1 LAN	G.726 40kbps	1.1.8	318	291	**294**	10	18	**14**
		1.3	202	374	**323**	15	18	**16**
Group 2 Broadband	G.726 40kbps	1.1.8	373	558	**438**	83	104	**93**
		1.3	323	651	**439**	87	106	**96**
Group 3 56k dial-up	G.726 24kbps	1.1.8	685	1096	**808**	172	194	**184**
		1.3	691	1468	**867**	175	206	**195**
Group 4 28.8k dial-up	G.726 24kbps	1.1.8	720	1348	**913**	171	254	**226**
		1.3	739	1287	**981**	194	269	**231**

Table 6-1 - The results of test cases conducted on various configurations

6.3.2 Synchronization of the Audio and the Video Decoders

A total of thirty (30) people were surveyed to report the experience they had with the JPlayer. The responses are organized in the pie graph shown below.

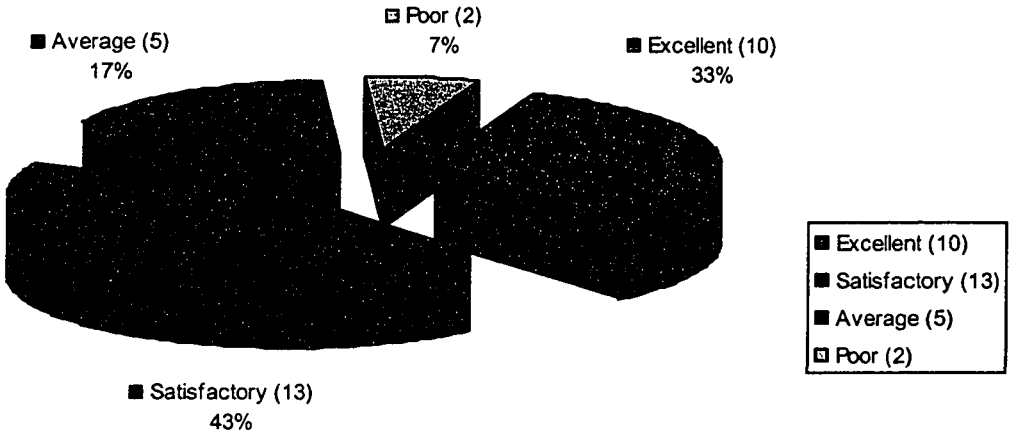

Figure 6-3 - A pie graph showing the responses of the testers

The average temporal offset between the internal clocks of the audio and the video decoders is measured to be 87 milliseconds with the video decoder lagging behind the audio decoder.

6.3.3 CPU Utilization

The JPlayer with audio and video streamings on a Pentium III-650 consumes an average of 25% CPU utilization. The JPlayer Audio Decoder that decodes audio only consumes an average of 20%. Under both scenarios, the decoder's consumption reaches its peak of 45% CPU utilization at initialization when the class files are loaded. The CPU consumption of the server is very insignificant.

6.4 Analysis of the test results

The test results of the JPlayer Audio Decoder are consistent with the expectation. The average round-trip and the average interframe separation under Java 1.1.8 are 274 ms and 13 ms respectively and both gradually increase as the configuration shifts from Group 1 towards Group 4.

It might be worthwhile to point out that Java runtime environment 1.1.8 performs better than version 1.3 - it scores tens of milliseconds less than version 1.3 under all testing configurations. This could be due to the additional resources occupied by the Java Sound API. Both Java runtime environment versions 1.3 and 1.4 are known to suffer from performance issues. Many users on the Internet have suggested changing to the Java HotSpot runtime environment which claims to be a virtual machine that delivers the highest possible performance, and allows real-time applications to execute significantly faster.

Another point worth noting is that, the fluctuations in measurements in Group 2, 3 and 4 are relatively large. This could be witnessed by the great differences between the high and low values. The greater fluctuations observed in these groups can be explained by the effect of transmissions over the Internet. The Internet, being an IP network in its greatest scale, provides best effort service. It is not a surprise that the propagation times

of each data frame diverge significantly.

Since TCP/IP transmissions within the local machine take very little propagation time, an amount that can be justifiably neglected, the average round-trip time and average interframe separation in the local configuration could serve as references for the processing time taken up by the client and the server. The average round-trip time is high comparing to the ping time in a LAN environment which is usually under 50ms. This is due to the internal overhead of the Java network facility.

Most configurations, namely local, group 1 and group 2, were tested with an audio clip in G.726 40kbps encoding. The available bandwidth of the network in these categories is well beyond the bandwidth requirements of a clip in G.726 40kbps encoding. The JPlayer Audio Decoder has performed reasonably well as expected. However, it has experienced interruptions when streaming was conducted over a 44kbps dial-up connection connected to an ISP using a V.90 modem. Contrary to the belief that a 44kbps network connection should be capable to provide real-time streaming of a 40kbps clip, it was brought to attention that there are overheads that increase that minimum bandwidth requirement. Each data frame sent by the server has a 1000 bytes of payload. Each frame is encapsulated in the JPlayer Audio header, the TCP header, and, finally, the IP header. Together, the three headers contribute an additional 80 bytes to each data frame which

translates to an 8% increase in the effective bandwidth requirement. Table 6-2 lists the effective bandwidth requirement of the G.711, different versions of G726 encodings.

Encoding	Sample Size	Bandwidth Requirement	Effective Bandwidth Requirement
G.711	8 bit	64 kbps	**69.1 kbps**
G.726 40kbps	5 bit	40 kbps	**43.2 kbps**
G.726 32kbps	4 bit	32 kbps	**34.6 kbps**
G.726 24kbps	3 bit	24 kbps	**25.9 kbps**

Table 6-2 – The effective bandwidth requirements of various audio encodings.

As shown in the above table, the effective bandwidth requirement of a G.726 (5 bit) clip is very close to the dial-up connection speed we had (44kbps). 44kbps is the maximum throughput of the connection which is unlikely to be attained at all time due to the unpredictable disruptions caused by the Internet- the transmission undergoes occasional delays and the transmission was unable to catch up with the JPlayer Audio Decoder. The streaming of G.726 (4 bit) and G.726 (3 bit) audio clips on the other hand, suffers no interruption.

In section 4.2, we have derived a formula that computes the number of frames the JPlayer Audio Decoder should request in order to avert interruptions. Recall that

$$request_i = \frac{FDP \times (buffer_threshold - buffered_i) + ART}{FDP - AIFS} \qquad (4.2.9)$$

As it was discussed in section 4.2, the subtraction in the denominator signifies that the value of the average interframe separation ought to be smaller than the frame departure period which is assigned with a value of 250 milliseconds. The larger the value of AIFS, the more frames is needed to sustain an uninterrupted flow. As it approaches 250 ms, the formula fails to converge which implies the JPlayer Audio Decoder undergoes interruptions.

Similar results were collected when the test was conducted over a 28 kbps connection, although the fluctuations were greater in both roundtrip time and interframe separation.

Most testers who evaluate the JPlayer felt satisfied with the quality of the synchronization. While most are satisfied with the clip with a man singing, some pointed out that the slight lip-voice out of sync is most perceived with the clip of a man speaking in a rapid pace. The cause of this experience should not be viewed as an imperfection of the synchronization given that the difference of the timing reference of the two decoders is minimal. The root cause lies in the inability to capture the rapid movement of the lips

with the 10 frames per second sampling of the video clip.

The CPU utilizations recorded on both the server machine and the client machine which runs the JPlayer were low. This observable fact is very significant. Not only does it mean that machines that are limited in process power, including those palm-sized devices, will have the capacity in running the JPlayer, the JPlayer could also be run in conjunction with the collaboration tools to provide users with a full-featured interactive experience since concurrent execution of other applications on more powerful machines is possible.

6.5 Areas for Improvements

The project can be extended in several work areas which are listed as follows.

The compliance to the H.323 recommendation- H.323 is an umbrella recommendation from the International Telecommunications Union (ITU) that sets standards for multimedia communications over Local Area Networks. Complying with the H.323 allows interworking of the JPlayer with a large number of services available.

The decoupling of the components in the JStreaming Video Decoder- as of this writing, the core of the JStreaming Video Decoder employs merely one thread with which establishes connection with the server, requests video data, decodes and renders video frames. The efficiency of the decoder could be greatly improved by assigning tasks to

individual threads.

Frame loss recovery algorithm- the JPlayer runs over TCP connections. TCP trades reliability with speed which is an important factor for any real-time multimedia applications. Given the relatively big overhead of TCP, eventual migrating to a light-weight transmission mechanism becomes apparent. (Both UDP and Real-time Transport Protocol are good candidates.) Without the reliable transmission guarantee of TCP, transmissions over unreliable IP networks demands the implementation of a frame loss recovery algorithm which equips with the logic to selectively re-request late or lost frames based upon the circumstance and the real-time requirement.

The dynamic selection of data encoding- unlike networks that features bandwidth reservation such as ATM, IP networks provides no guarantee of service. This means sudden degradation (or improvement) in throughput could occur and thus introducing interruptions. With the dynamic selection of data encoding in place, the JPlayer could dynamically select the best possible encoding possible base upon the current network characteristics. It could request the server to sent audio data of higher quality if the network is unoccupied and switch to an encoding with lower bandwidth requirements when the network is congested.

Chapter 7 – Conclusion

In this thesis, we have first reviewed the objective of the project which is to design and implement a Java-based decoder that is capable of decoding G.726 audio streams. We have listed the design goals of the project and the justification for each design goals. The design goals encompass the use of the Java language, broad audience reachability, and the support of open standards. Some possible applications of the decoder have also been reviewed.

We have explained the importance of the audio and video codecs. These technologies provide compression of high bit rate sampled data to low bit rate codes that are suitable for transmission over networks. The compression technologies that are used in the project which includes G.726 and H.263, are surveyed.

We have explained the details of the design requirements which are stated to fulfill the design goals. The high level design or the initial model of the design has covered the development of the audio codec, the derivation of the request algorithm which governs the flow of streaming contents sent by the server to the client, the functions and the mechanics of the four major building blocks of the decoder- the session controller, the decoder, the buffer and finally, the scheduler, the outline of the network monitoring

feature which collects the statistics of round-trip time and interframe separation. Together, they reflect the conditions of the underlying network.

Some of the challenges encountered during the implementation phase were discussed. Most of these have demanded new designs of certain parts of the decoder. The new design has employed a TCP channel between the server and the client, a multi-thread approach, three separate audio players tailored for different Java versions, a feature to handle audio interruptions, some code optimization for better performance, and lastly, the synchronization between the audio and the video decoders.

The test cases and their results were documented. These test cases are aimed to evaluate the performance of the audio decoder, and the quality of the synchronization. The test results were generally satisfying and have shown consistence with the expectation.

The project can be extended in several work areas. These work areas include the compliance to the H.323 codec, the decoupling of the H.263 transcoder and the scheduler in the video decoder. Further study can also include the frame loss recovery algorithm for transmission over unreliable networks, the dynamic selection of data encoding with different bandwidth requirements to facilitate the real-time condition of the underlying networks.

References

[1] Oliveira, J.C., (1997, April), A Java H.263 Video Decoder, Retrieve from http://www2.mcrlab.uottawa.ca/~jauvane/elg5378/H263JavaDecoder/, University of Ottawa

[2] H.263 - Video coding for low bit rate communication, International Telecommunication Union, February 1998

[3] G.726 - 40, 32, 24, 16 kbit/s adaptive differential pulse code modulation (ADPCM), International Telecommunication Union, December, 1990

[4] G.711 - Pulse code modulation (PCM) of voice frequencies, International Telecommunication Union, December, 1990

[5] L. R. Rabiner and R. W. Schafer, Digital Processing of Speech Signals, Prentice-Hall, 1978

[6] G.722 - 7 kHz audio-coding within 64 kbit/s, International Telecommunication Union, November 1988

[7] H.261 - Video codec for audiovisual services at p x 64 kbit/s, International Telecommunication Union, March 1993

[8] Sun Microsystems Inc., Connected Limited Device Configuration, Retrieved from http://java.sun.com/products/cldc/

[9] Palm Boulevard, PalmOS 5 to Be Unveiled At PalmSource, (2002, January 17), Retrieved from http://www.palmblvd.com/articles/2002/1/2002-1-17-PalmOS-5-to.html

[10] Sun Microsystems, Inc., PersonalJavaTM Application Environment Specification Version 1.1.3 final, (1999, December 3), Retrieved from http://java.sun.com/products/personaljava/index.html

[11] Hardwick, J., (1998, March 18). Java Optimization- how to optimize your Java programs so that they're faster, smaller, and more maintainable. Retrieved from http://www.cs.cmu.edu/~jch/java/optimization.html

[12] Nagle, J., Congestion Control in IP/TCP Internetworks, RFC 896, 1984

[13] Douglas, F. G., (2001, May 16), RealNetworks, Microsoft announce media player plans, Retrieved from http://www.idg.net/idgns/2001/05/16/RealNetworksMicrosoftAnnounceMediaPlayer.shtml

[14] Grimes, C., Intel, Real deal quells 'streaming' media fears, Financial Times, March 4, 2002

[15] Rosoff, M., (2001, July 1) Windows XP Gives New Push to Windows Media Format, Retrieve from http://www.directionsonmicrosoft.com/sample/DOMIS/update/2001/07jul/0701wxgnpt.htm

[16] Microsoft Inc., Windows Media Technologies- Processors, Retrieved from http://www.microsoft.com/Windows/windowsmedia/wm7/drm/providers.asp

[17] Sun Microsystems Inc., JavaTM Media Framework API, Retrieved from http://java.sun.com/products/java-media/jmf/index.html

[18] Kendal, R., (2002, January 27), AudioCentric- Free audio player and plugin replacement applet, Retrieved from http://www.cheetahpublisher.com/AudioCentric

[19] Champeaux, D., Douglas, L., Faure, P., Object-Oriented System Development, Addison Wesley, 1993

Appendix A - Competitive Analysis

The JPlayer is not the only product on the market that provides real-time audio and video streaming. There are a number of products from different vendors that offer similar functionalities. This section of the body presents these competing products and follows by a competitive analysis.

1. RealPlayer

RealPlayer from the RealNetworks has had a big presence in the video/audio streaming area since its first release in 1995. RealPlayer was initially released on the PC platform as a revolutionary product that offers real-time audio streaming over the Internet. Subsequently, releases for other platforms such as Linux/UNIX and MAC were also delivered, and it has remained as the most popular audio streaming application on the Linux/UNIX platform. The original version 2.0 of the RealAudio format is designed to carry monophonic audio contents that are optimized for delivery over 14.4 and 28.8 kbps dial-up connections. Starting from version 3.0, the RealAudio format has incorporated new features such as multicasting and advanced encoding of audio stream optimized for high bandwidth connections. RealPlayer delivers multimedia contents using TCP but

optionally supports transmission over UDP.

RealPlayer has always targeted the desktop computer market primarily. In spite of this, Real Networks announced in May, 2001 that RealPlayer applications would be available on Sony Computer Entertainment Inc.'s Playstation 2 gaming console and on Nokia Corp.'s Media Terminal, an upcoming "infotainment" device. [13]

Intel, the world's largest chipmaker, on March 4, 2002 said it would include the Real Player on its newest PC desktop motherboards. It is a positive step for Real Networks in battling with its rival, Microsoft Inc. [14]

2. Microsoft Media Player

Microsoft has made Windows Media Player very popular by bundling with the Windows operating system. The Windows Media Player has been a long time rival of the RealPlayer- both players target primarily the desktop computer market. The Windows Media Player claims to have the best encoder that delivers near CD-quality digital audio with the least bandwidth on the market. The latest codec provides good video quality at relatively low bandwidth, CD-quality audio at 64 Kbps and near-CD-quality sound at 48 Kbps. [15]

The Windows Media Player originally was introduced along with the Windows

operating system and was only available to the Intel PC platforms. The Mac OS X version of Windows Media Player coming in early 2002 will also support playback of streamed or downloaded files using the latest Windows Media Audio and Video 8 codecs. The Windows Media Player has built-in support for digital rights management. Users will be able to download music and movies protected for purchase or promotion.

Microsoft is also working to expand the number of devices that support the WMF format, a proprietary format. The company has released a version of its Media Players for the PocketPC platform which runs Windows CE, that will play WMF 8 files; licensed the Windows Media codec to chip manufacturers such as Texas Instruments and Cirrus Logic; and demonstrated WMF video files playing on smart phones. [16]

3. Java Media Framework

The original Java Media Framework (JMF) 1.0 API was jointly developed by Sun Microsystems, Inc., Silicon Graphics Inc., and Intel Corporation. The goal of the Java Media Framework API is to define a simple, unified architecture to synchronize and control audio, video and other time-based data within Java applications and applets. The enhanced JMF 2.0 API was co-developed by Sun Microsystems, Inc. and IBM to address the shortcomings of the 1.0 release. It provides additional features such as capturing,

filtering and transcoding between different formats. [17]

The JMF API enables audio, video and other time-based media to be added to Java applications and applets. This optional package, which can capture, playback, stream and transcode multiple media formats, extends the multimedia capabilities on the Java platform, and gives multimedia developers a powerful toolkit to develop scalable, cross-platform technology. The JMF API provides H.263 and G.726 support as well as the support for most popular video and audio formats. The JMF API relies on Java Sound API to provide audio rendering. The Java Sound API is available as a standard API under J2SE 1.3 or downloadable module for the Java 1.1.x and 1.2 environments. To utilize the functionalities of the JMF API, users must download and install the JMF API on each client machine. The JMF API is available for download from Sun Microsystems and the size of which is about 2 Mb.

4. AudioCentric

AudioCentric is a module written in Java that is capable of playing audio on most Java enabled platforms. [18] It supports streaming of .AU files, Windows' .WAV file and RealAudio's version 2.0 28.8. As with the JPlayer, AudioCentric is designed to play audio of these formats without the installation of additional software. The Java-only

implementation and the small footprint allow AudioCentric to be integrated easily into web pages. Due to AudioCentric's incapability of playing video streams, it is unfair to compare AudioCentric against other products mentioned. It should be noted however, AudioCentric can only be executed under Java 1.1.x environments.

Decoder	Size	Platform	Audio Support	Video Support	Supported Formats	Note
JPlayer	~180 Kb	Most Java-enabled platforms	Yes	Yes	H.261/H.263, G.726	
RealPlayer 8 Basic	4.7 Mb	Windows, Unix/Linux, Mac, Sony's PlayStation 2, Nokia's Media Terminal	Yes	Yes	.RA, .RM, and most popular formats	
Microsoft Media Player	Bundled with Windows, 4.9 Mb for Mac, 12 Mb for Solaris	Windows, Windows CE, limited support on Mac and Solaris	Yes	Yes	WMF, and most popular formats	
Java Media Framework	~2 Mb	Java-enabled Platforms 1.1.x and above	Yes	Yes	most popular formats	JMF is available as a downloadable module to Java 1.1.x and above
AudioCentric	324 Kb	Java 1.1.x only	Yes	No	.WAV, RealAudio V2.0 28.8, .AU	Does not run on J2SE 1.2/1.3.

Table A-1 - Some competitors of the JPlayer and their characteristics

Appendix B - List of Figures and Tables

List of Figures:-

Figure 1-1 - Examples of the applications of the project

Figure 2-1 - Linear and logarithmic quantizations

Figure 2-2 - PCM and Adaptive Differential PCM

Figure 3-1 - PersonalJava and other Java Runtime Environments

Figure 4-1 - The timing diagram of messages exchanged between the server and the client

Figure 4-2 - The number of frames stored in the buffer at $time_0$ and $time_1$

Figure 4-3 - The four main components of the JPlayer Audio Decoder

Figure 4-4 - The state diagram of the session controller

Figure 4-5 - The exchange of operations between the JPlayer Audio Decoder and the server

Figure 4-6 - The operation of the G.726 decoder

Figure 4-7 - The flow of frames into and out of the buffer

Figure 4-8 - The state diagram of the buffer

Figure 4-9 - The state diagram of the scheduler

Figure 5-1 - The combined TCP Channel

Figure 5-2 - The .AU file header

Figure 5-3 - Selecting the right audio player

Figure 5-4 - Feeding a silence frame

Figure 5-5 - The conveyance of command via the AVBridge

Figure 5-6 - Exchange of timing reference between the audio and video decoder

Figure 6-1 – The JPlayer Audio Decoder playing an audio clip in G.726 encoding

Figure 6-2 - The JPlayer showing a musical clip

Figure 6-3 - A pie graph showing the responses of the testers

List of Tables:-

Table 2-1 - Characteristics of different G.726 audio encoding standards

Table 2-2 - Supported picture formats of H.261, H.263 video standards

Table 3-1 - Networks with different characteristics

Table 5-1 - The methods for applet-browser interaction and their descriptions

Table 5-2 - Methods invoked under different browsers

Table 5-3 - The availability of audio players on Java runtime environment of different versions

Table 6-1 - The results of test cases conducted on various configurations

Table 6-2 – The effective bandwidth requirements of various audio encodings.

Table A-1 - Some competitors of the JPlayer and their characteristics