View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Arrow@dit

The ITB Journal

Volume 5 | Issue 1 Article 2

2004

Developing a Distributed Java-based Speech Recognition Engine

Tony Ayers
Institute of Technology Blanchardstown, tony.ayers@itb.ie

Brian Nolan
Institute of Technology, Blanchardstown, brian.nolan@tudublin.ie

Follow this and additional works at: https://arrow.tudublin.ie/itbj

Part of the Computer and Systems Architecture Commons

Recommended Citation

Ayers, Tony and Nolan, Brian (2004) "Developing a Distributed Java-based Speech Recognition Engine," *The ITB Journal*: Vol. 5: Iss. 1, Article 2.
doi:10.21427/D7GX7H
Available at: https://arrow.tudublin.ie/itbj/vol5/iss1/2

This Article is brought to you for free and open access by the Journals Published Through Arrow at ARROW@TU Dublin. It has been accepted for inclusion in The ITB Journal by an authorized administrator of ARROW@TU Dublin. For more information, please contact yvonne.desmond@tudublin.ie, arrow.admin@tudublin.ie, brian.widdis@tudublin.ie.

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 License

Developing a Distributed Java-based Speech Recognition Engine

Mr. Tony Ayres, Dr. Brian Nolan
Institute of Technology Blanchardstown, Dublin, Ireland
tony.ayres@itb.ie brian.nolan@itb.ie

Abstract

The development of speech recognition engines has traditionally been the territory of low-level development languages such as C. Until recently Java may not have been considered a candidate language for the development of such a speech engine, due to its security restrictions which limited its sound processing features. The release of the Java Sound API as part of the Java Media Framework and the subsequent integration of the Sound API into the standard Java development kit provides the necessary sound processing tools to Java to perform speech recognition.

This paper documents our development of a speech recognition engine using the Java programming language. We discuss the theory of speech recognition engines using stochastic techniques such as Hidden Markov Models that we employ in our Java based implementation of speech signal processing algorithms like Fast Fourier Transform and Mel Frequency Cepstral Coefficients.

Furthermore we describe our design goal and implementation of a distributed speech engine component which provides a client server approach to speech recognition. The distributed architecture allows us to deliver speech recognition technology and applications to a range of low powered devices such as PDAs and mobile phones which otherwise may not have the requisite computing power onboard to perform speech recognition .

1. Introduction

In the past speech recognition engines have been developed using low level programming languages such as C or C++. Early versions of the Java programming language would not have been candidate languages for the development of such speech systems. The sandbox security model employed by the Java platform prevents rogue Java code from damaging system hardware but it also limits access to the hardware needed to perform speech recognition, namely the sound card. This problem could be overcome by using native code (i.e. C, C++, etc.) to implement functionality that Java could not and then use the Java Native Interface to link the Java and native code together. With this solution the platform independence that Java brings to software development is lost and the complexity is increased. When Sun released the Java Sound API [1] and the Java Media Framework [2] (JMF), these extensions to the Java platform allowed developers to create applications that took advantage of not only sound processing but also advanced video and 3D functionality also, moreover these extensions provide this added functionality without compromising the Java security model. With these APIs Java is not only a capable development platform for building speech recognition engines, but an engine developed in Java inherits its benefits of platform independence and object oriented development.

The proliferation of the Java platform from the desktop computer to handheld devices and set-top boxes provides part of the motivation for developing a Java based speech recognition engine, given that Java 2 Micro Edition is present on over 100 million mobile phones world wide, speech recognition applications could be delivered to a large number of people users. Other motivating factors include the project from which this development has stemmed, which is entitled Voice Activated Command and Control with Speech Recognition over WiFi. In this project we are developing speech recognition software for the command and control of a remote robotic device. Our requirements for this project included a Linux based distributed speech recognition engine and a speech recognition engine for an iPaq Pocket PC PDA. Java's proficiency for operating seamlessly across multiple platforms coupled with its networking capabilities through RMI and sockets made it an obvious choice for developing our software.

The remainder of this paper describes our approach to developing speech software in Java. We explain the theory behind speech recognition and map the theory to a Java based implementation and code. We also describe our design of a speech distributed recognition engine.

2. Speech Recognition and Java

As has been detailed already, pure Java based speech recognition software would not have been feasible prior to the release of the Java Sound and Media APIs. Despite this, speech processing capabilities have been available to Java based applications since the release of the Java Speech API 1.0 [4] in 1998. The API offers speech recognition and synthesis functionality leveraged through a speech engine provided on the host operating system. For example, on the Windows platform the Java Speech API can take advantage of the recognition and synthesis capabilities provided by the Microsoft SAPI [5] engine. Other engines supported by JSAPI include IBM Via Voice [6], Dragon Naturally Speaking [7] and Phillips Speech SDK 2.0 [8]. JSAPI provides no speech processing functionality of its own, therefore it is not a solution to the question of Java based speech processing tools.

The Sphinx [9] project taking place at Carnegie Mellon University, offers a large scale, set of speaker independent speech recognition libraries which can be used in various application scenarios. Sphinx II, is designed for fast real time speech applications and Sphinx III offers a slower more accurate speech engine. Both of these engines are written in the C programming language. Speech recognition continues to be an area of much research in the Java community, Sphinx 4 is being developed in Java, with support from Sun Microsystems. The source code for Sphinx project is open source, much of the implementation of our speech system is based on

this code and related Java code from an open source project entitled OCVolume [10]. OCVolume is a small Java based speaker dependent speech recognition engine which is also based on the front end of Sphinx III.

3. The Speech Recognition Process

Automatic Speech Recognition (ASR) is the ability of computers to recognise human speech. There are two distinct types of speech recognition engine namely, continuous and non-continuous. Continuous speech recognition engines allow the user to speak in a normal conversation style, in contrast non-continuous engines require speech to be input in a more constrained manner, for example some require longer than average pauses between words. Both continuous and non-continuous recognition engines can be classified as either speaker dependent or speaker independent.

Speaker dependent engines require a user to train a profile of their voice before the engine can recognise their voice; speaker dependent systems offer the greatest accuracy as the engine is tailored to the characteristics of the users voice. Speaker independent engines do not require any training, while a speaker dependent engine has a profile of the users voice, speaker independent systems use databases of recorded speech. The databases contain voice samples of many different users, words, phrases and pronunciations.

Hidden Markov Models to date represent the most effective and accurate method of performing speech recognition [11]. A Hidden Markov Model is specified by the states Q, the set of transition probabilities A, defined start and end states and a set of observation likelihood's B [12]. Section 3.1 describes Markov models in greater detail.

Figure 1 diagrams the speech recognition process from the speech input at the microphone to the recognised speech output.

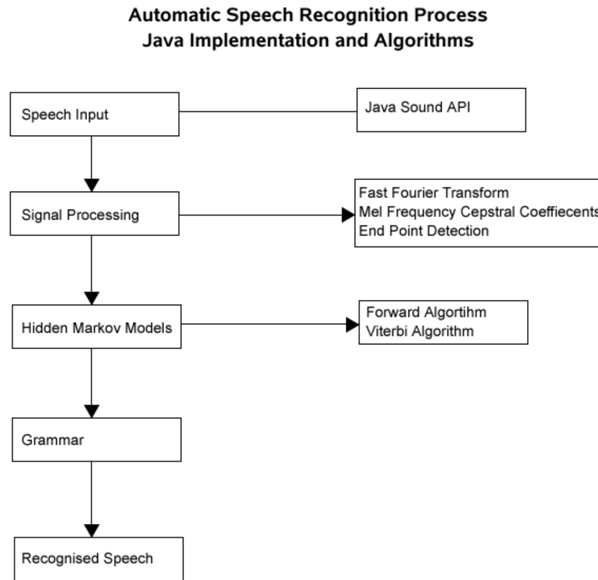

Figure 1 The Speech Recongition Process

3.1 Speech Input

Speech input is taken from a microphone attached to the system sound card, the sound card handles the conversion of the analogue speech signal into digital format. Depending on the recognition software type i.e. Continuous or non-continuous the engine may need to listen continuously for sound input, in this case a continuous audio stream will need to be opened and read from (Section 3 documents our implementation of this procedure using the Java Sound API).

3.2 Signal Processing

Speech input is taken from a microphone attached to the system sound card, the sound card handles the conversion of the analogue speech signal into digital format. At this point we have a digitised version of the speech signal. Speaking comes naturally to people, when we are spoken to we hear individual words, sentences and pauses in the speech, more so our understanding of language allows to interpret what was said. Consider what happens when we hear people speaking in a language which is foreign to us, we don't hear the individual words in the language and the speech sounds like one continuous stream of noise. The same scenario is true when we speak to computers for the purposes of speech recognition. The process of finding word boundaries is called segmentation.

Applying this knowledge to our digitised signal data, we need to process this signal in order to determine the word boundaries in the signal and also to extract relevant data which we use to

model the signal and eventually perform recognition through Hidden Markov Models [section 3.3]. We use the Fast Fourier Transform algorithm, an implementation of the discrete Fourier transform, to quickly calculate the Fourier equation in real time.

3.3 Hidden Markov Models

Hidden Markov Models (HMM) have proven to date to be the most accurate means of decoding a speech signal for recognition. HMM are stochastic in nature, that is, they generate a probability that an outcome will occur. In our speech system the input to the HMM will be a speech signal sampled at a particular moment in time, the output from the Markov Model will be a probability that the inputted signal is a good match for a particular phoneme. We can create numerous Hidden Markov Models to model speech signal input at time samples of the signal, each Markov Model can represent a particular phoneme. By combining the probabilities from each Markov model we can produce a probabilistic output that a given speech sequence is a representation of a particular word.

The fundamental concept in a HMM is the Markov assumption, this concept assumes that the state of the model depends only on the previous states. Given this concept, we can introduce the Markov process, a process which moves from state to state depending only on the previous n states. [11]

HMM = (∏ , A , B)

∏ = the vector of initial state probabilities
A = the state transition matrix
B = the confusion matrix

Figure 2 Definition of a Hidden Markov Model

Figure 2 shows a formal definition of a Hidden Markov Model, ∏ represents the initial set of states. A represents the transition matrix or the probability of transiting from one state to another. B represents the confusion matrix or the set of observation likelihood's, which represent the probability of an observation being generated from a given state. [11].
Having modelled the speech input utterances using Hidden Markov Models we need to determine which observation sequence produces the best probability, we use the Viterbi algorithm to do this. The goal of the Viterbi algorithm is to find the best state sequence q given the set of observed phones o.

3.4 Grammar

Given a speech corpus which consists of many thousands of speech samples, during the recognition process the hidden markov model may have to search the entire corpus before finding a match. This searching problem is known as perplexity and can cause the speech engine to operate inefficiently, which translate into delays in recognition for the user. Grammars can be used to reduce perplexity. Grouping Markov Models of phonemes together we can form words. For example, a grammar can specify a particular word order, such that if we can recognise word A and we know that word B never follows word A we can eliminate searching the corpus for word B.

3.5 Recognised Speech

The output from these steps of the process is a String of text which represents the spoken input, at this point the recognised speech can be applied to some application domain such as in a dictation systems or, in our case, a command and control scenario.

4. Implementation

The implementation of our speech system is a speaker dependent system which requires the words to be recognised be recorded or sampled by the user prior to the use of the system.

The system accomplishes its goal by use of the process described in section 3, with one additional component. Given the need to distribute the speech process we developed a Java class to handle network communications between the client and the server. The client application is responsible for signal capture, while the server handles the process of recognition.

4.1 Signal Acquisition with the Java Sound API

Acquisition of the speech signal is achieved through a microphone using the Java Sound API. The audio signal is recorded as pulse code modulation (PCM) with a sample rate of 16KHz, this is implemented in Java using floating point numbers. The Java Sound API objects TargetDataLine and AudioFormat are used to create the input data line from the sound card, the method open() called on the TargetDataLine object opens the line for audio input. The AudioFormat object is used to create audio input of the specified type, in our case this is PCM signed with a frequency of 16KHz. Code fragment 1 shows the basic Java code used to open a line for audio input. The Java code in this class is implemented as a thread, which allows the system to do other work in the recognition process while continuously listening for audio input.

```
TargetDataLine SampleLine;
AudioFormat format = new AudioFormat(AudioFormat.Encoding.PCM_SIGNED, 16000.0F, 16, 1,
16000.0F, 2, false);
DataLine.Info info = new DataLine.Info(TargetDataLine.class, format);

if (AudioSystem.isLineSupported(info))
{
        int soundData = 0;
        try
        {
                SampleLine = (TargetDataLine) AudioSystem.getLine(info);
                SampleLine.open(format);
                SampleLine.start();
                soundData =  SampleLine.read(audiostream, 0, BUFFER_SIZE);
          //do some work with soundData captured e.g. Call another method
                SampleLine.stop();
                SampleLine.close();
                sampleOutputStream.close();
        }
        //catch exceptions statement Here
}
//remainder of the program
```

Code 1 Capture Audio with JavaSound

4.2 Signal Processing Algorithms

Stage two of the speech process requires the signal to be processed in order to identify key features in the speech signal e.g. word boundaries etc. The acoustic signal data from the audio recording process is used as the input to our signal processing and analysis stage. Figure 3 shows a block diagram of the phases which make up signal processing. The mathematical formulae for calculating the stages in figure 3 and the corresponding Java implementation code is shown below. As space is limited only code relevant to each formula is included rather than entire methods and objects.

The Mel Scale [14] is a perceptual scale based around the characteristics of the human ear, it attempts to simulate the operation of the human ear in relation to the manner in which frequencies are sensed and resolved. Using the Mel scale in speech recognition can vastly improve the recognition accuracy. Calculation of the Mel scale is achieved using the Mel Filter bank, which involves applying triangular filters in the signals frequency power spectrum. The purpose of the Mel Filter Bank is to smooth out the pitch harmonics and noise in the speech signal and to emphasise the formant information.
The procedure for Mel cepstrum coefficients is:

1. Divide the signal into frames
2. Obtain the power spectrum

3. Convert to Mel Spectrum
4. Use Discrete Cosine Transform to get Cepstrum Coefficients

This procedure is integrated into the feature extraction process shown in figure 3

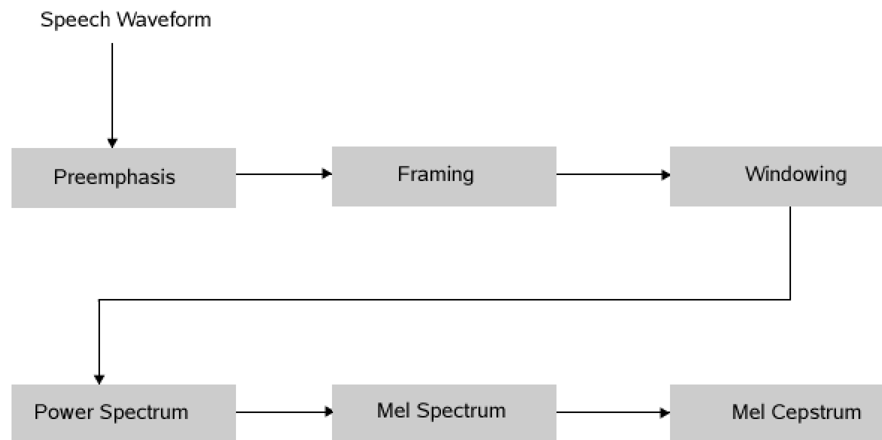

Figure 3 Signal Processing in Detail (Feature Extraction)

Pre-emphasis

In voice signals frequencies below 1KHz have greater energy than higher frequencies, this is due to the energy from the glottal waveform and the radiation load from the lips [16]. In order remove the tilt we can use a High Pass Filter, applying this filter removes the glottal waveform and radiation load from the lower frequencies and distributes the energy equally in all frequency regions.

```
FIR Pre-Emphasis
y[n] = x[n] –  x[ n - 1 ]

Java
    double outputSignal[] = new double[inputSignal.length];
    for (int n = 1; n < inputSignal.length; n++)
    {
        outputSignal[n] = inputSignal[n] - preEmphasisAlpha * inputSignal[n – 1]; //preEmphasisAlpha = 0.95
    }
    return outputSignal;
```

Equation 1 Pre-Emphasis

Framing

The framing process involves breaking up the signal into frames with a shift interval in order to create a 50% overlap with a previous part of the signal.

Windowing

The frame is multiplied by a hamming window [13]:

$$w[n] = 0.54 - 0.46 \cos\left(\frac{2\Pi n}{N-1}\right)$$

where N is the length of the frame

Java

```
double w[] = new double[frameLength];
for (int n = 0; n < frameLength; n++)
{
        w[n] = 0.54 - 0.46 * Math.cos( (2 * Math.PI * n) / (frameLength - 1) );
}
```

Equation 2 Windowing

Power Spectrum

The power spectrum is calculated by performing a discrete fouler transform through a fast Fourier Transform algorithm. The sum of the square of the resulting real and imaginary arrays from the fourier transform yields the power spectrum. [13]

$$s[k] = (\text{real}\,(X[k]))^2 + (\text{imag}\,(X[k]))^2$$

```
double pwrpectrum[] = new double[frame.length];
FFT.computeFFT( frame );
 for (int k = 0; k < frame.length; k++){
        pwrspectrum[k] = Math.pow(FFT.real[k] * FFT.real[k] + FFT.imag[k] * FFT.imag[k], 0.5);
    }
```

Equation 3 Power Spectrum

Mel Spectrum

The Mel spectrum of the power spectrum is computed by multiplying the power spectrum by each of the mel filters and integrating the result [13]. The corresponding Java implementation is not shown due to its size.

$$S[l] = \sum_{k=0}^{N/2} s[k]\, M_l\,[k] \quad l=0, 1..., L-1$$

N is the length of DFT, L is the lotal number of mel filters.

Equation 4 Mel Spectrum

Mel Cepstrum

A discrete cosine transform is applied to the natural log of the mel spectrum to calculate the cepstrum. [13]

$$c[n] = \sum_{i=0}^{L-1} \ln(S[i]) \cos\left(\frac{\pi n (2i + 1)}{2L}\right) \qquad c=0,1 \ldots C-1$$

C is the number of cepstral coefficients.

Java
```
double cepc[] = new double[numCepstra];

    for (int i = 0; i < cepc.length; i++) {
        for (int j = 1; j <= numMelFilters; j++) {
            cepc[i] += f[j - 1] * Math.cos(Math.PI * i / numMelFilters * (j - 0.5));
        }
    }
    return cepc;
```

Equation 5 Mel Cepstrum

4.3 Hidden Markov Model Implementation

The Hidden Markov Model is implemented as a Java object, the constructor of the Markov takes two integers corresponding to the number of states and number of observation symbols. The initial state is set to 1 and the transition probabilities are initialised to random values. The Viterbi algorithm is implemented to find the best (most probable) path through the Markov trellis. The input to the algorithm is an observation sequence corresponding to an input signal (i.e. speech utterance). The signal has been pre-processed by the feature extraction stage; the Viterbi algorithm returns the probability that the input utterance is a recognised word. The training process for the Hidden Markov Model involves the use of the Baum Welch Algorithm [15]. This algorithm is designed to find the HMM parameters.

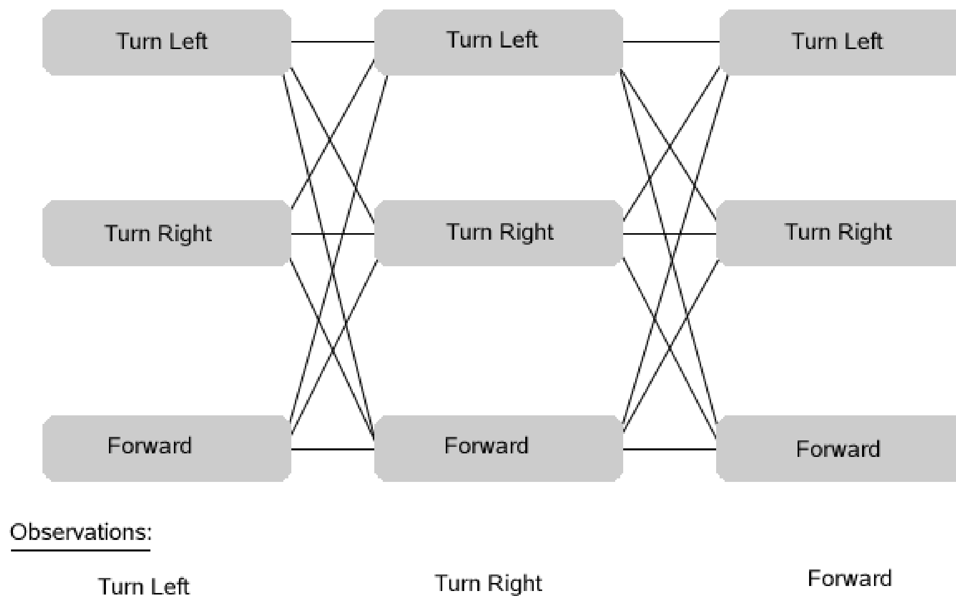

Figure 4 HMM Trellis

4.4 Distributed Engine Architecture

The speech engine we are developing is intended to be deployed across numerous disparate hardware/software systems, including desktop PCs running the Windows and Linux operating systems respectively, and an iPaq handheld PDA running Pocket PC 2003. Of these systems, the iPaq presents us with the greatest challenge, given its limited processing power and memory storage. With this in mind we devised a distributed speech recognition architecture, where the speech processing work could be performed by a more powerful server, with the Java based client on the iPaq only responsible for capturing the signal. This architecture is shown in Figure 5.

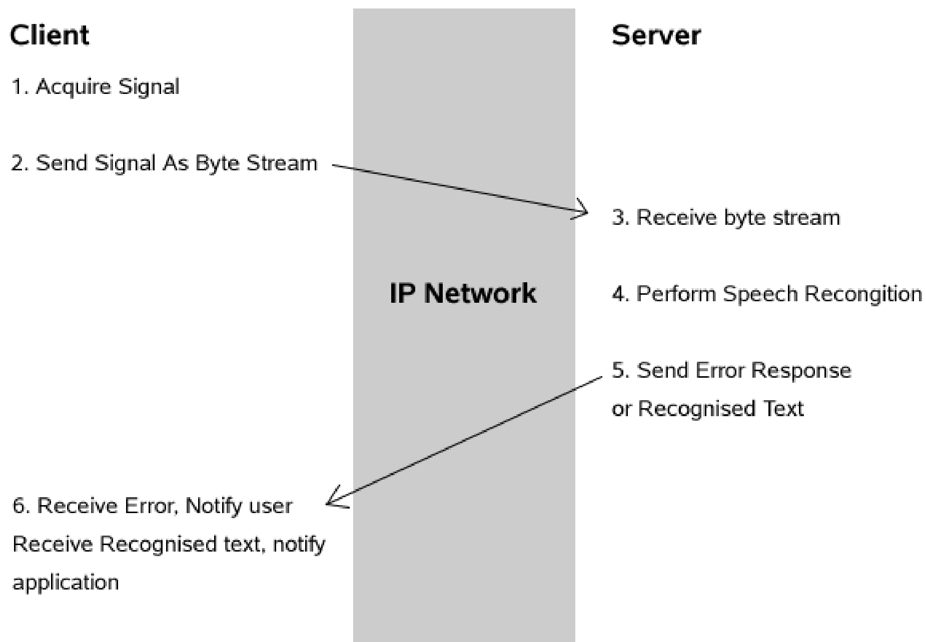

Figure 5 Distributed Speech Recognition Architecture

Using the Java Sound API, the Java Client can capture the signal as byte data which is then sent over an IP network to the server using a socket. The TCP/IP protocol is used to ensure reliability and the correct ordering of the packet data arriving at the server. The choice of TCP/IP as the protocol is not a trivial one, an alternative is be to use the UDP protocol. UDP datagram packets can arrive in different i.e. unordered sequences (or not at all since UDP does not guarantee delivery), the correct order of the speech signal is vital, as recognition errors will occur otherwise. If the UDP protocol is employed the server will need to wait for all packets to arrive and order them accordingly before processing them for recognition, this could cause delays in the recognition process. Implementing fail safes for these factors can be totally avoided with TCP/IP.

5. Conclusions and Future Work

The Java programming language provides a suitable development platform with capable APIs for developing speech recognition technology. We have seen how the objects and methods provided through the Java sound API can be used to deliver continuous signal acquisition for a speech recognition process. The basis for a solid Java speech recognition engine is in place, the next phase is to investigate and make changes to the code such that it can be deployed on the iPaq and on the Linux operating system.

The current implementation of this system requires the user to train the words and phrases which will be recognised. The training process is quite intensive for the user, involving frequent manipulation of the user interface (Starting and stopping the record process), this process is not conducive to creating vocabularies of any any greater size than a few dozen words. While improving the usability of the training process could increase the potential vocabulary of the system, however a speaker independent approach would eliminate all training requirements. Speaker independent speech recognition systems do not require the user to train the system prior to use. Speaker independent systems rely on acoustic information provided to them by large speech corpus. Integrating a speaker independent component into this system through a speech corpus, would be the next logical step in the systems evolution.

6. References

1. Sun Microsystems, Java Sound API [online at] http://java.sun.com/products/java-media/sound/
2. Sun Microsystems, Java Media Framework [online at] http://java.sun.com/products/java-media/jmf/
3. A. Spanias, T. Thrasyvoulou, S. Benton, Speech parameterization using the Mel scale [online] http://www.eas.asu.edu/~spanias/E506S04/mel-scale.pdf (18/3/04)
4. Sun Microsystems Ltd, Java Speech API, [online at] http://java.sun.com/products/java-media/speech/
5. Microsoft Corporation, Microsoft Speech and SAPI 5, [online at] http://www.microsoft.com/speech/
6. IBM, Via Voice [online at] http://www-306.ibm.com/software/voice/viavoice/ (15/1/2004)
7. ScanSoft, Dragon Naturally Speaking, [online at] http://www.scansoft.com/naturallyspeaking/
8. Phillips, Phillips Speech SDK 2.0, [online at] http://www.speech.philips.com/ (20/1/2004)
9. CMU Sphinx Project, CMU Sphinx [online at] http://www.speech.cs.cmu.edu/sphinx/index.html (15/1/2004)
10. Orange Cow, OCVolume [online at] http://ocvolume.sourceforge.net (19/3/04)
11. Boyle RD, Introduction to Hidden Markov Models, University of Leeds, online at http://www.comp.leeds.ac.uk/roger/HiddenMarkovModels/html_dev/main.html
12. Jurafsky D. & Martin J.H. , Speech and Language Processing An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, 2000, Prentice Hall, New Jersey.
13. Seltzer M., Sphinx III Signal Processing Front End Specification, CMU Speech Group, August 1999.
14. S. Molau, M. Pitz, R. Schluter, H. Ney, Computing MelFrequency Cepstral Coefficients on the Power Spectrum, Proc. Int. Conf. on Acoustic, Speech and Signal Processing, Salt Lake City, UT, June 2001
15. Boyle RD, Introduction to Hidden Markov Models, University of Leeds, online at http://www.comp.leeds.ac.uk/roger/HiddenMarkovModels/html_dev/hmms/s2_pg3.html

16. Mustafa K, Robust Formant Tracking for Continuous Speech with Speaker Variability, Dept. of Electrical and Computer Engineering, McMaster University, [online] http://grads.ece.mcmaster.ca/~mkamran/Research/Thesis%20Chapters/3%20-%20The%20Formant%20Tracker%20-%20pp.%2041-70.pdf