to appear, 1993 AVIOS Conference, San Jose CA.

Speech Recognition Architectures for Multimedia Environments

Eric Ly[1], Chris Schmandt and Barry Arons

Speech Research Group
MIT Media Laboratory
20 Ames Street, Cambridge MA 02139
ly@media-lab.mit.edu

1 Abstract

Computer workstations have recently become powerful enough to support speech recognition entirely in software, but speech recognizers still vary in their functionality, and each vendor offers their own programmatic interface. Developing recognition applications currently means writing to non-portable protocols. As new improved recognizers become available, such applications will need to be rewritten for new protocols. A recognition server can abstract such differences from client applications, while supporting the use of different classes of recognizers available today.

This paper describes the design and implementation of an asynchronous recognition server; the asynchronous server allows a client application to continue operation and possibly attend to other input and output events while waiting for recognition to complete. Its internal architecture is based on an object-oriented engine application programming interface (API). The server is designed to support speaker-independent connected-speech recognition and speaker-dependent, isolated-word recognition paradigms, while offering applications a consistent programmatic interface for accessing this functionality. As recognition engines with better performance arrive, they can be incorporated into to the server, via a standardized engine API, and automatically become available to all client applications without modification.

2 Motivations and Applications

Today, sophisticated multimedia environments include not only graphical input but also audio input, and applications employing the use of speech recognition are beginning to appear in such environments. Unfortunately, while many of the state-of-the-art recognition systems offer similar functionality, they also have non-standard programmatic interfaces. These systems also assume that recognition will be the primary audio function, seizing control of the entire audio input stream and in some cases the computational resources of the workstation. For multimedia environments, these assumptions make recognizers an unfriendly component in a community of mixed and shared resources. While graphical window systems already have standard interfaces [8], application developers using speech recognition cannot yet write to portable interfaces. Adapting an application for a new recognizer may involve a significant rewrite of code. In addition, more work may be required to obtain asynchronous operation for recognizers which have been designed to take program control from an application until a word is recognized.

More important, multimedia environments also invite the use of recognition with other audio functions, such as recording and playing back messages and monitoring environmental speech [10, 6]. Speech recognizers must therefore co-exist as one of many resources in an asynchronous, event-driven environment, rather than being only the primary one. For example, a user may want to record, playback and send audio messages from his desktop but may also want to drive the window system using speech commands [9, 4, 5]. Speech recognition may be turned on or off depending on whether the user is recording a message. A recognizer must cooperate by sharing the audio input stream with other applications and by working in an asynchronous manner.

Like window system servers, a recognition server can provide a suite of functionality based on an application's needs while allowing a device-independent, asynchronous means of accessing such functionality. Such a server can also offer multiple client applications the simultaneous ability to perform speech recognition, intermixed with playback/record functions, from a single audio stream.

The server concept described in this paper has already been used to provide recognition capability for several

1. The author is currently affiliated with Stanford University but can still be reached at the above email address.

projects, including the a voice-controlled window system [9], a speech-only hypermedia system [2] and a hand-held note taker [11]. The current version of the server was implemented for a work group messaging system call Chatter, which is used over a telephone [7]. This system runs on a workstation, where telephone audio is captured, recognized and processed. These applications are all examples in which the audio stream is used not only to recognize user commands but also to record and play back audio files or messages.

An early predecessor of this server relied on external recognition hardware communicating asynchronously to a single client [10]. This hardware configuration was also used in a subsequent version of the server described in [3]; it provided a speaker-dependent recognizer for client applications. This paper describes the work on the internal architecture of the recognition server, making it possible for the server to be outfitted with several recognizers and yet retaining the same API to external clients. The server architecture can both support software-based and hardware-based speech recognizers, although software-based recognizers are only currently used.

3 Design Issues

For the recognition server to be general enough to accommodate many classes of speech recognition algorithms, it must be able to deal with similarities and differences of recognition technologies available today. The main differences among recognition technologies can be represented along the three dimensions of speaker-dependence, connectedness of the recognition, and the recognizer's possible vocabulary size.

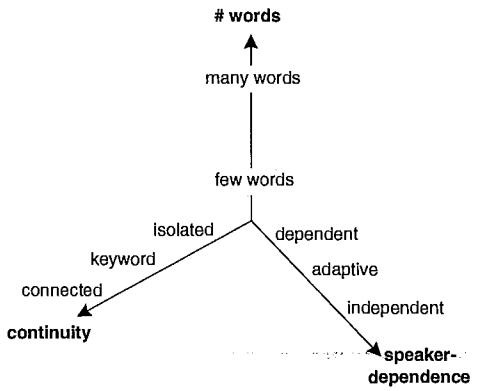

Figure 1: Recognizer functionality space

Each point in this space can be considered to represent a particular class of recognizer. For example, it is possible to have a large-vocabulary, continuous, speaker-dependent recognizer. Each point is also distinguished by its own operational requirements; some recognizers require language grammars while others require the training of word templates.

The server acts as an intermediary between client applications and the actual speech recognizers, henceforth called *recognition engines*. The server must therefore provide for clients to select the capability they want to use and to relay any additional data between the engine and client necessary for successful operation in a device-independent manner.

Although each point is a possible class of recognizer, current recognizers fall into roughly three categories:

- *Speaker-dependent, isolated-word, small-vocabulary.* Recognizers in this class are perhaps the simplest in implementation. Users of such recognizers supply training samples by saying each word of the vocabulary several times, and these samples are used as templates in matching speech at the recognition stage. A special kind of recognizer in this class includes the word-spotting recognizer, which attempts to pick up all or any keywords in a specified vocabulary from a continuous speech stream [14]. Such recognizers may be more useful in certain domains where commands may be given as complete sentences, in which case they can be used to pick out the essential information without much regard to analysis of syntactic structure.

- *Speaker-independent, isolated-word, small-vocabulary.* These recognizers are good for simple recognition tasks where accuracy and widespread speaker applicability is desired. Perhaps the most prevalent use of these recognizers are with over-the-phone consumer services, where particular items from a menu, a binary choice, or even a digit are to be spoken by the caller.

- *Speaker-independent, connected-word, large-vocabulary.* This class of recognizers represents the state-of-the-art in current speech recognition technology. Speech samples are collected from a large set of speakers and statistics of word frequencies and phoneme characteristics are distilled. Such recognizers typically use some form of Hidden Markov Models on a constraint grammar as the basic recognition algorithm.

It is therefore useful for the server to provide roughly three sets of functionality for clients along these classes.

To accomplish these ends, the server's architecture is constructed in an object-oriented inheritance-based manner. The motivation for organizing the server architecture in this way stems from the differing capabilities among

recognition engines. The server must provide a common set of functionality regardless of the engine employed, which implies that its implementation must make up for deficiencies of particular engines. At the same time, the server must also be able to defer to those engines which providing their own (or better) means of performing part of the recognition task. Also, while all engines of a particular class from different vendors are different in how they are accessed, they are in many ways structurally the same except for minor differences. Different classes of recognizers may have significantly different methods of usage. For example, a recognition result from a connected-word recognizer may be an entire string of words, but an isolated-word recognizer result consists of only a single word.

The current design of the server already incorporates three recognition engines—from Texas Instruments [13], Command Corp. [5], and Agog [4]. The TI recognizer offers speaker-independent, connected-word, large vocabulary recognition. The Command Corp recognizer, called IN^3, offers speaker-dependent, keyword recognition, while the Agog engine offers speaker-dependent, isolated word and small vocabulary recognition.

A version of the server is built for each engine, so there are currently three distinct recognition servers, all with the same communication protocol. It is currently up to the user to invoke the correct server before running the client applications.

4 Using the Recognition Server

The recognition server operates as a separate process and communicates with clients over Unix sockets. Connections and requests to the server are performed using remote procedure calls from the client to the recognition server, and recognition results are returned asynchronously over a socket.

4.1 Transport mechanism

Messaging between clients and the server relies on a set of tools developed described in [3] called the Socket Manager (SM) and the Byte Stream Manager (BSM), which simplifies the implementation of inter-process communication (IPC). SM handles low-level connection I/O, supporting a callback mechanism for connection initiation, termination and data transfer. BSM is an abstraction above SM, providing an RPC compiler and run-time library that supports synchronous and asynchronous calls. On each end, application code uses the BSM to communicate with the other process. The client-server architecture is illustrated in this diagram:

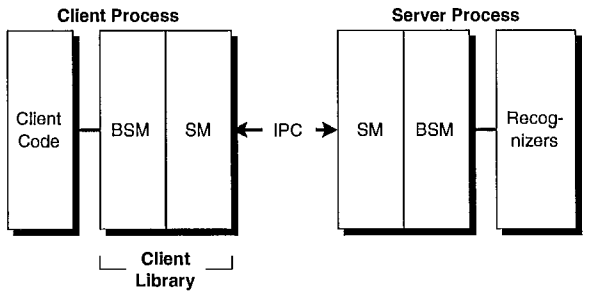

Figure 2: Client-server model

The client protocol consists chiefly of the following areas:

- Establishing and closing a connection with the server.

Code	Comments
`void r_recognizer_ev(int fd, void *data, char *utterance)`	Callback function for receiving recognition events
`{`	
`  printf("%s was recognized.\n", utterance);`	Simply prints what was recognized
`}`	
`int main(int argc, char *argv[])`	Main function
`{`	
`  int fd;`	
`  fd = r_open("localhost", R_TISR);`	Opens a connection to the recognition server
`  r_set_filename(fd, "grammar.tisr");`	Sets the grammar or vocabulary file to use
`  r_register_callback(fd, R_RECOGNIZE_EV, recognizedEvent, NULL);`	Registers the above function to receive recognition events
`  r_start_recognition(fd);`	Tells the server to begin recognizing
`  r_main_loop();`	Begin the event loop
`  return 0;`	
`}`	

Figure 3: Using the recognition server

Before recognition services can be handled, clients must establish a connection to the server. Opening a connection gives the client access to one recognition engine. The client must also specify the class of recognizer desired. Currently, this specification is simply given by the name of the engine. A connection to the server is represented as an integer Unix file descriptor, which is used to address the engine in all subsequent calls. When the client is finished with recognition, the connection can be closed.

- Setting vocabulary files for the engine. All types of recognizers require some configuration file which specify the vocabulary to be recognized. Connected-speech recognizers will usually require a grammar file. Before recognition can be performed, a client must tell the server which vocabulary or grammar file to use. Because such files are still largely engine-specific, the server simply passes this information onto the engine, so the meaning and format of the vocabulary is engine-dependent. The server does not establish any standards for the format of such files.

- Starting and stopping recognition. A message to the server starts or stops recognition. When recognition is active, audio data is passed to the engine and any recognition are captured and delivered to the client.

If the given engine supports vocabulary subsetting, the server has a protocol for performing subsetting. This capability is used to improve the accuracy of recognition. For isolated-word recognizers, vocabulary words are represented as integers mapped into a vocabulary word table known to the client. For connected-word recognizers, the subset is usually given by the top level grammar symbol(s). The current implementation only supports subsetting on a per client basis, and no provisions are made for different applications having different subsets. See "Conclusions" for ideas on future work.

4.2 Receiving recognition results

Recognition results are delivered through the asynchronous callback mechanism provided by the transport subsystem. Such results are also known as recognition "events." When recognition is enabled and a recognized result is detected in the audio stream, the server generates an event, and the client's callback function is invoked with the result so it can be processed.

The server has two types of recognition events: one is a simplified result which only returns what was recognized; the other provides the recognized input and more detailed information such as confidence measures and secondary hypotheses. To obtain these events, clients can register a callback to receive one or both of these events. When registered, the client's callback will be invoked when a recognition event is detected. For isolated-word recognizers, results are integers while for connected-word recognizers, results are returned as strings representing the actual words spoken.

4.3 Training

For engines requiring training templates, the server has a protocol for performing training sessions and obtaining templates. For training to occur, the client application must be aware that its recognition engine requires it. It is necessary because training almost always requires user interaction and feedback at the top level of the interface.

When training is initiated by the client, a training "begin" event will be generated each time a training template is expected from the user. This event can be used by the client to prompt the user to speak. This event is generated several times for one word, since many recognizer require multiple training sessions to generate a reliable template.

When training has completed, a training "end" event is sent to the client. If a sequence of words is to be trained during the training session, the client can then initiate a new training session when it receives the "end" event.

The protocol also includes calls to load and save templates once they are trained.

5 Server Implementation

To support the multitude of recognition engines, the server uses an object-oriented design, in which the core functionality is embedded into an abstract root class, and actual recognizers are implemented as subclasses of this root class. These subclasses inherit functionality from the root class that they cannot provide, and they override functionality if their own is better. More often however, implementations of engines will not only use the core functionality but also provide their own to perform additional initialization or processing. For different classes of recognizers, abstract subclasses of the root class can be specified to provide a separate set of core functionality associated with that particular class of recognizers. All of these concepts can be represented naturally in an objected-oriented, inheritance hierarchy.

To enable the use of different brands of recognizers in the server, an internal engine API is also defined to ease

Class definition	Comments

```
class Recognizer
{
  Recognizer(int connfd, int sfd);

  virtual ~Recognizer();

  virtual int fd();
  virtual void setUser(const char *aUser);
  virtual void setDataFile(const char *aFile);
  virtual void startAudio();
  virtual void stopAudio();
  virtual BOOL wantsAudio();
  virtual int frameSize();


  virtual void prepareReceiveAudio();

  virtual void receiveAudio(const bsm_bytes *buf);
  virtual void finishReceiveAudio();

  virtual void loadFile();
  virtual void saveFile();

  virtual void startRecognition();
  virtual void stopRecognition();
}
```

Initializes the recognizer with connection and audio file descriptors
Deallocates memory for the recognizer

Called by the server to initiate audio delivery
Called after audio delivery is suspended
Returns true if recognizer wants audio
Implemented by engines to tell the server the buffer size the engine desires for audio

Called before the first audio buffer is delivered so the engine can perform any necessary initialization
Engines implement this function to receive audio
Implemented by engines to perform any needed cleanup

Invoked by the server to load the vocabulary file

Figure 4: Core *Recognizer* engine API

the process of incorporating new recognition engines into the server. The engine API is specified as a set of member functions of the abstract *Recognizer* superclass. Future providers of recognizers can also give programmatic interfaces that make their incorporation possible more straightforward. As recognition engines with better performance become available, they can be added to the server, via this standardized engine API, and automatically become available to all applications which use the server without modification.

An abstract *Recognizer* class forms the root class. It implements functionality common to all recognizers, regardless of their type, such as the mechanisms for handling connection control with the client, device-independent means of receiving audio data, converting between the necessary audio formats, loading and saving of data files, and the maintenance of state of the recognition process. Then, abstract subclasses of *Recognizer* are defined for each of the three common types of recognizer.

Figure 3 shows the server's internal class hierarchy. The *WordRecognizer* class implements the speaker-dependent, isolated-word, small-vocabulary recognizer, and the *ConnectedRecognizer* a speaker-independent, connected-word, large-vocabulary recognizer. A subclass of *WordRecognizer* or *ConnectedRecognizer* serves as "glue" for an engine and server. The *IndependentRecognizer* subclass is not shown because no recognition engines are yet implemented using it currently. The three lowest classes in the hierarchy actually implement the current recognition engines.

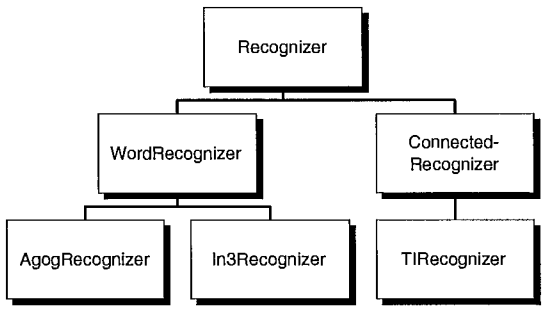

Figure 5: Internal recognizer hierarchy

The various subclasses implement the protocol necessary for the operation of that recognizer class. For example, the *WordRecognizer* class specifies that training is to begin with a message to `startTraining`, then a `train` message to train individual words and finally a `stopTraining` message to end training. The *AgogRecognizer* simply overrides these calls to prepare its own internal voice models for training. To receive audio data, the *TIRecognizer*, for example, implements a `receiveAudio` method defined as part of the protocol by the *Recognizer* class. `receiveAudio` is automati-

cally invoked with audio data when the audio stream is turned on for recognition or training.

Each connection to the server is represented as an object instance of a particular recognizer class, and calls to the server are simply translated to messages to the appropriate objects.

6 Interfacing to Audio Input

The server is written to cooperate with an audio server, described in [1], which allows several applications to use the audio device as a shared resource. The use of an audio server allows recognition to the performed without prohibiting other processes from using the audio stream. The recognition server functions as a client of the audio server, although its processing is also useful to other applications. This diagram illustrates how the recognition server fits into the general audio server framework:

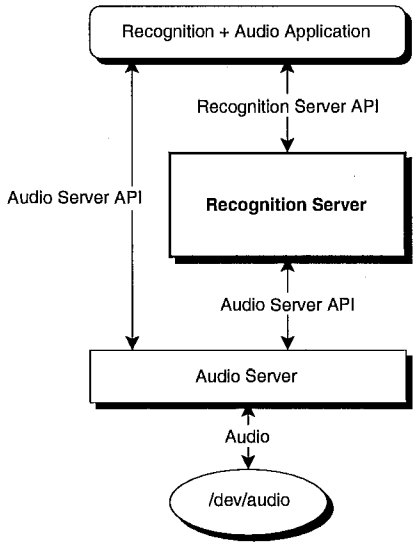

Figure 6: Audio application architecture

An application wanting recognition functionality can communicate with the recognition server to receive recognition "events" found in the incoming audio stream. It can also initiate the training of word templates to be used for later recognition. While the server acts as intermediary between the client and audio server, a client application can still receive and send control commands directly to the audio server. This framework allows applications to perform mixed-mode audio tasks, where training, recognition, record and playback can be arbitrarily intermixed during their execution.

The recognition server does not open the audio device itself but rather relies on the audio server to read from the audio device and deliver copies of the audio data for recognition analysis. In the meantime, the audio server may also be writing audio data to a file for recording purposes or delivering additional copies of the data to other applications for touch tone detection, for instance.

7 Interfacing with Recognizers

When designing programmatic interfaces for a specific recognizer, the designer should realize that the recognizer will be working in a shared-resource, asynchronous configuration. This section outlines the key issues to be considered in designing the server's engine API and how the internal server architecture facilitates the integration of new recognizers.

7.1 Receiving audio

A recognizer must work in an environment where the audio stream is a shared resource, so it must be designed to not require unique access to the audio input but be able to accept audio from an arbitrary source. It is a task of the recognition server to receive audio data from the audio server and in turn deliver this data to the recognition engine. Several methods exist for transferring audio to recognizers in the recognition server. The recognition server currently supports two methods of delivering audio.

If the engine executes as a separate process, the engine can receive audio from a Unix named pipe. The engine and recognition server can establish a shared pipe, and when audio begins to arrive at the server, it can be written to the pipe and received by the recognizer for processing. This method was used to deliver audio to the IN^3 recognizer. If the recognizer is already designed to read from the real audio device, then it is straightforward to change the device's name from which audio may be read. The drawback of this method is that it is not very efficient because of the additional overhead involved in context-switch time to transfer data using pipes to the recognizer.

A more efficient means is to pass audio to the engine via a function call that is provided with the engine. The engine defines a function for receiving audio with arguments to specify the location and length of data, and when the server receives an audio buffer, it can deliver this buffer using the function. This method was used to provide audio to the Agog and TI recognizers. Note that it is the responsibility of the recognizer to manage the received audio thereafter, since the server passes only

one copy of the stream to the recognizer. The recognizer can either process the new buffer as soon as it is received or save it in private buffers until the end of a speech segment is recognized. In the server's current design, audio will be lost if the function takes too long to return, so recognizer's API design should take into account the time it takes to process a frame of audio. However, providing a call to receive audio is the most efficient means of delivery.

Since delivery of audio can be suspended and restarted as dictated by the external functionality of the server, the recognizer should also provide a reset function which initializes its internal state for a new session of recognition. The problem is that the server sends audio to the engine only when either recognition or training is active, and the engine may receive a buffer of audio that occurs much later in time than the previous buffer it received. If the engine performs recognition on these buffers as if the audio in them were contiguous in time, it may incorrectly report spurious recognition results. A reset function provides the means to restart the recognizer in a clean state after audio delivery has been suspended.

7.2 Posting recognition results

As a recognizer receives audio, it must post recognition results to the server so that client processes may be notified of recognized speech. Since the server operates asynchronously, reporting results from recognizer to server must also occur asynchronously. Here, there are three methods for recognizers to post results: the server polls for a result; the recognizer reports results directly from the function used to pass the recognizer audio; or the recognizer invokes a callback to report the result. Once the server is notified of a recognition result, it can then generate a recognition event in the client.

In the first method, the server periodically polls the recognizer to check for any recognition events. The server may call a function or check a variable's value from time to time to determine whether any results can be obtained. This method is used to find recognition results for the IN^3 recognizer. The testing interval may be controlled by a timer or some other periodic event. Since delivery of audio data is a periodic event, the server currently checks for recognition results after several buffers have been delivered. The drawback of this approach is twofold: processor time is wasted on polling, and there may be a lag time between when the recognition result is known and when it is discovered, which may affect the interactivity of the system. However, this method may be used for recognizers that were not originally designed to be operated asynchronously.

A more efficient method is to send recognition results as the return value of the function used to pass in audio. This method also assumes the particular style of audio delivery described above, and is used to determine recognition results for the TI recognizer. In this case, the function result can easily be checked to determine whether something was recognized. If nothing has yet been recognized since the last result, this condition may be represented by an empty result. While this method removes the inefficiency of busy waiting, it may still result in lag response time if the audio buffer size is large.

The best method is to use a callback mechanism. The server registers a callback using a function in the recognizer's API. When the recognizer computes a recognition result, it immediately notifies the server, which also sends the client a recognition event. This method is used to obtain recognition results from the Agog recognizer, and it avoids the disadvantages of the first two approaches.

8 Conclusions

Speech recognizers are beginning to move out of the prototyping stage into application use. As part of a multimedia platform, their functionality should be accessed through a standard programmatic interface so applications do not have to be rewritten for every recognizer. Recognizers must also cooperate with other processes using the audio resource.

The recognition server described in this paper accomplishes these ends by providing the necessary abstractions and resources for recognizers to be used effectively. It introduces an external client API and internal engine API, which are necessary for interfacing to client applications and for integrating new engines, respectively. This work also addresses the issues involved in making engines more easily adaptable to this framework.

One of the weaknesses of the current client API is that engines are specified for use rather than desired functionality. Future directions include improving the API so that clients need only to specify recognition capabilities they require, leaving the server to select the appropriate engine for the task. For connected-speech recognizers, one of the challenging issues related to this task is in specifying a common grammar format that can be used across all such recognizers, so that grammars do not have to be rewritten for different engines. The challenge is in how to map the common format to the specific for-

mat required by each engine, whose format is still different for every other. These challenges can be met by examining and integrating new engines to determine the common areas of functionality among them.

As speech recognition applications become more common, it will also become necessary to coordinate recognition requirements among them. Different applications will each desire their own vocabulary and grammar, and the server will have to dynamically update the active vocabulary as applications begin and terminate. A mechanism for distributing recognition events to different applications is also required, much as a window manager arbitrates keyboard and mouse events among applications running simultaneously.

Acknowledgments

We would like to thank Greg Cockroft of Agog, Inc. and Charles Hemphill of Texas Instruments for adapting their recognizers to work with the server presented in this paper. Lisa Stifelman also worked on earlier versions of the recognition server for use with the Voice Navigator on the Macintosh. She also helped by reviewing drafts of this paper.

This work was sponsored by Sun Microsystems, Inc.

References

[1] B. Arons. The design of audio servers and toolkits for supporting speech in the user interface. *Journal of the American Voice I/O Society*, 9:27–41, 1991.

[2] B. Arons. Hyperspeech: navigating in speech-only hypermedia. In *Hypertext '91*, pp. 133–146, ACM, 1991.

[3] B. Arons. Tools for building asynchronous servers to support speech and audio applications. Proceedings of the ACM UIST Symposium, 1992.

[4] G. Cockroft. SayIt recognizer. Marketed by Qualix Group, Inc., 1992.

[5] IN^3 Recognizer. Command Corp, Inc., 1992.

[6] D. Hindus and C. Schmandt. Ubiquitous audio: capturing spontaneous collaboration. In *CSCW '92*. ACM, 1992.

[7] E. Ly. Chatter: a conversational telephone agent. MIT Masters Thesis, Media Arts and Sciences Program, 1993.

[8] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–106, 1986.

[9] C. Schmandt, M. Ackerman and D. Hindus. Augmenting a window system with speech input. Computer, pp. 50–56, IEEE, 1990.

[10] C. Schmandt and M. McKenna. An audio and telephone server for multimedia workstations. Proceedings of the 2nd IEEE Conference on Computer Workstations, pp. 150–160, 1988.

[11] L. Stifelman. Not just another voice mail system. *In proceedings of* American Voice Input/Output Society, pp. 21–26, 1991.

[12] L. Stifelman, B. Arons, C. Schmandt and E. Hulteen. VoiceNotes: a speech interface for a handheld voice notetaker. *In proceedings of* INTERCHI '93, 1993.

[13] B. Wheatley, J. Tadlock and C. Hemphill. Automatic efficiency improvements for telecommunications application grammars. First IEEE Workshop on Interactive Voice Technology for Telecommunications Applications, 1992.

[14] L. Wilcox and M. Bush. HMM-based wordspotting for voice editing and indexing. *In Proceedings of* Eurospeech '91, pp. 25–28, 1991.