INSTITUTE FOR SIGNAL AND INFORMATION PROCESSING

summary of deliverables for

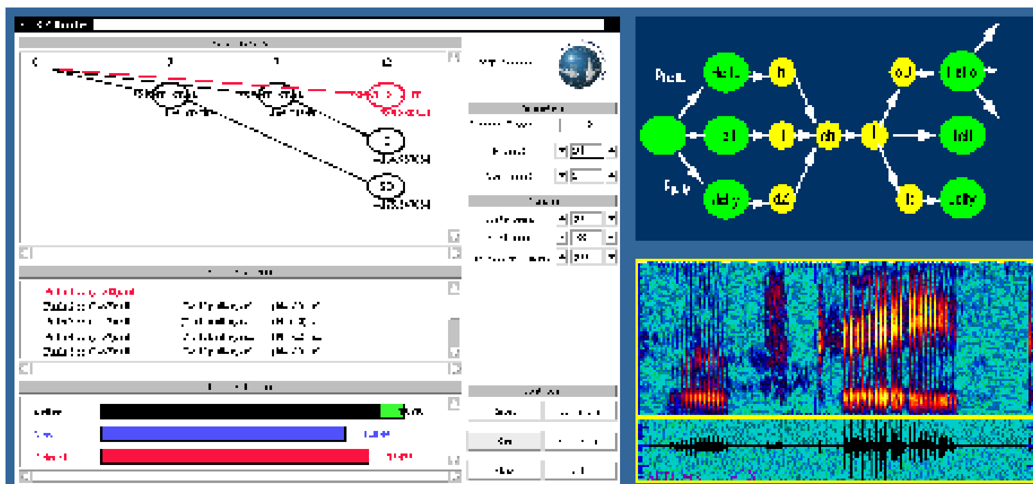
Robust Low Perplexity Voice Interfaces

Subcontract No. 43556

submitted to:

Dr. Fred J. Goodman
Signal Processing Center
The MITRE Corporation
1820 Dolley Madison Blvd., M/S W622
McLean, Virginia, USA 22102-3481

August 15, 2001



submitted by:

F. Zheng and J. Picone
Institute for Signal and Information Processing
Department of Electrical and Computer Engineering
Mississippi State University
Box 9571, 413 Simrall, Hardy Road
Mississippi State, Mississippi 39762
Tel: 662-325-3149, Fax: 662-325-3149
primary email contacts: {zheng, picone}@isip.msstate.edu

EXECUTIVE SUMMARY

This report summarizes the second set of deliverables for our one-year collaboration with the MITRE Corporation on a project titled “Robust Low Perplexity Voice Interfaces.” The goal for the second portion of this project was to develop a Resource Management speech recognition server demo running within DARPA Communicator architecture. The longer term goal of this portion of the project is to produce a real-time recognition server that provides a robust and flexible command and control voice interface in realistic tactical noisy environments.

In the second phase of this project, we have successfully completed three milestones:

- Designed and implemented (in C++) a speech recognition API for the ISIP prototype system to support a recognition server running within the DARPA Communicator framework.
- Developed a Resource Management recognition system from 8 kHz sampled data, based on the 16kHz baseline system of last deliverable. This system delivers a WER of 4.4% at ~9 xRT on a 600 MHz processor.
- Delivered a recognition server demo which allows a user to decode audio data and display the resulting hypothesis in a text window.

In order to better support the DARPA Communicator hub/server architecture, we added a *communicator* utility to our prototype system distribution. We also added the Communicator API for the recognition server to our standard distribution so that it would be officially supported. In order to avoid the repetition of code in utilities such as *extract_feature* and *trace_projector*, we added C++ classes for these functions and modified our utilities to use these classes instead of in-line code. Thus, the prototype system now has a much improved architecture and significantly more flexibility (for example, feature extraction is an option inside the decoder). To make the system run in a streaming audio real-time mode, we changed the implementations of cepstral mean subtraction and energy normalization in the front end. We are currently evaluating the performance impact of these changes on the real-time system delivered last quarter.

In order to support MITRE’s Java-based Audio server (beta version), which collects audio data at an 8 kHz sample rate, we downsampled 16 kHz Resource Management data to 8 kHz, and then we retrained models based on the optimal parameters of our 16 kHz baseline system. The performance degraded from 3.4% to 4.3% in WER. We feel this degradation is a little high and are analyzing the results to better understand this.

Our Communicator recognition server provides the necessary C wrappers and event handlers that will allow the recognition server to conform to an application programming interface defined by MITRE. This simple demo demonstrates that the recognition server is able to receive a stream of audio data from an audio server and outputs the resulting hypothesis into a text window using a printout server. At present, we support a push-to-talk interface, though more complex audio interactions are possible. We are now beginning work on the proper format for the output hypotheses so that this system can be easily interfaced to a database query system. Applications under development require two forms of output: an SQL format that can be used directly to query a database, and an N-best format that can be postprocessed by other natural language (NL) processing tools. Eventually, we would like to implement a flexible NL parsing module to server these needs.

TABLE OF CONTENTS

1. **INTRODUCTION** 1

2. **A COMMUNICATOR RECOGNITION SERVER** 1

   2.1. A Communicator Recognition Server API. 2

   2.2. Prototype System Modifications 2

3. **FUTURE WORK** 3

4. **ACKNOWLEDGMENTS** 5

5. **REFERENCES** 5

6. **INSTAIIATION INSTRUCTIONS** 7

1. INTRODUCTION

The DARPA Communicator [1] is an open source architecture for constructing dialogue systems developed by MITRE Corporation. Its plug-and-play approach enables developers to combine architecture-compliant commercial software and cutting edge research components. It allows the different components of speech systems from different sites to communicate to each other using a client/server/hub architecture.

Last quarter we developed a real-time system on the DARPA Resource Management database [2] (1000 words, bigram perplexity of 60). Our baseline system had a WER of 3.4% running at 9.7 xRT on a 600 MHz Pentium processor. This system has been modified to run at near real-time (~1 xRT) with a WER of 5.0% [3]. This quarter, we have integrated the baseline system into a recognition server that supports the DARPA Communicator framework. We provide the speech recognition functionality for this architecture by establishing an API to interface the ISIP prototype system [4] to the Communicator architecture. We also provide the necessary C wrappers and event handlers that will allow the system to conform to an application programming interface defined by MITRE.

This deliverables for this portion of the project consist of a recognition server demo that supports MITRE’s Java-based audio server, and includes a flexible front end and a parameter file-driven configuration option. We also support one-best and N-best output formats (the latter was requested by Lockheed Martin). The system runs on both SPARC and x86 platforms. In order to run this demo, one must install the v5.10 [5] or greater of the ISIP prototype system, which includes an integrated API for the speech recognition server. This API supports audio data in a raw data format, three flexible processing modes (whole-utterance, on-the-fly or user-defined), and a front end conducive to real-time processing.

The recognition demo consists of three parts: the core recognition engine, an audio test server, and a display box (printout server). The audio test server simulates a real-time audio server: it takes a raw audio file and sends the data to the recognition server using a buffered approach. The recognition server decodes audio data and displays the resulting hypothesis in a text window of the printout server. Using MITRE’s Java-based audio server, it is easy to run a live push-to-talk demo using microphone input with this server. The default demo allows users to read sentences from the Resource Management task and displays the resulting hypothesis. In this demo, we use models trained on 8 kHz data. The instructions for installing the demo are given in Appendix A.

2. A COMMUNICATOR RECOGNITION SERVER

Our efforts to develop a Communicator recognition server initially focused on two things: understanding the required API and modifying the prototype system to be more amenable to a real-time demo. The Communicator notes made available by the Center for Spoken Language Research at University of Colorado Boulder [6] were extremely useful in understanding the basic API. The Java-based audio server supplied by MITRE made incorporating live microphone input particularly easy. In order to make the recognition server run at real-time, we had to make some modifications to the front end processing. The API is described in detail in the next section, followed by a description of the changes we made to the prototype system.

2.1. A Communicator Recognition Server API

The prototype system interfaces to the Communicator architecture through an application programming interface (API) that was implemented in C++, and provides basic control of the processing flow:

int uttproc_rawdata(short* raw_data, int num_samples, int block);

This function decodes an utterance from the input audio data. The "block" argument specifies whether the recognizer should buffer the data until all pending data have been received. If 0, it is "non-buffering", which means the recognizer will process all the received data on the fly and be able to do the real-time decoding. If -1, it is "whole-utterance-buffering", which means the recognizer will buffer all the data received and will not actually process the data until *uttproc_end*() is called. If "block" is any integer > 0 (for example, 250), the mode is called "user-defined." In this mode, the recognizer will buffer the received data until the data size is greater than the specified number of milliseconds (e.g., 250 msec).

int uttproc_mfcdata(short* mfc_data, int num_samples);

This function decodes an utterance assuming the input data is feature data. The decoder will not do the final back trace and give the hypothesis until *uttproc_end*() is called.

int load_models(char* params_file);

This function reads the parameter files, and then configures the front end and the decoder according to the user-defined parameters.

int uttproc_begin();

This function resets and re-allocates if necessary the decoder contents that need to be cleaned up before decoding a new utterance. This is called at the beginning of each utterance.

int uttproc_end();

This function marks that no more data is forthcoming in the current utterance. It processes the pending data and obtains the final hypothesis.

int uttproc_result()

This function obtains the final hypothesis for the current utterance.

int uttproc_abort()

This function aborts the current utterance immediately without final hypothesis.

2.2. Prototype System Modifications

In order to integrate the recognition API into the prototype system, we made several modifications to our prototype system: (1) We added a *communicator* utility, which encapsulates the required Communicator API; (2) We added an *extract_feature* class which implements our front end processing from a C++ function rather than a utility; (3) We added a *trace_projector* class which also runs our decoder from a function call; (4) A core function *decode_cc() was added* to *decoder*

class, which will decode the feature data directly from data in memory buffer instead of from a file. By making the changes, we accomplished two significant things: (1) the decoder now can perform feature extraction internally; (2) all related utilities share the same code base. For historical reasons, feature extraction was previously implemented as a standalone program. That error has essentially been corrected.

Our real-time Resource Management system, which achieves a word error rate (WER) of 5.0%, runs at ~1.1 xRT on a 600 MHz Pentium processor. We were able to port this code relatively unchanged into the Communicator framework for a push-to-talk demo in which all speech is buffered before being processed. However, the overall system had several undesirable features from a real-time system point of view: (1) 16 kHz sample rate and (expensive) Sennheiser microphone acoustic channel; (2) cepstral mean subtraction requires buffering the entire utterance; and (3) energy normalization requires buffering the entire utterance. The first problem leads to a model mismatch: we trained models from 16 kHz clean data recorded by Sennheiser microphone. However, the real application includes various noise conditions, compression, and packet loss, which result in an increased word error rate.

In order to make the system running within Communicator framework in a streaming audio mode, we needed to change implementations of cepstral mean subtraction and energy normalization. The proposed real-time implementations for each feature are similar — we need to use a time-adaptive filter-based approach. Our first attempt at implementing an adaptive approach to energy normalization simply normalized by the maximum value in the buffer. For large buffers, this was a reasonable approximation. However, for true streaming audio with low latency, this was not acceptable since the buffers would need to be short. Although the normalized energy would eventually converge to the true normalized energy if the maximum value occurred early in the utterance, the initial portions of the utterance would be improperly normalized. Hence, this approach produced a 30% relative increase in the WER.

The maximum energy can also be computed using a time-adaptive approach [7] as follows:

$$E_{max}(n) = \alpha * E_{max}(n-1) + (1-\alpha) * max(E_{max}(n-1), E(n)) \tag{1}$$

where, the constant α falls generally between 0.9 and 1.0, and the initial value for $E_{max}(0)$ can be set to the expected average value of the maximum energy for a given application. In Figure 1 we compare the results from three different normalization schemes: **max**: normalization by the maximum value in an utterance (our non-real-time standard approach); **buffer**: normalization by the maximum value in a buffer; **adaptive**: normalization using a time-adaptive approach. The adaptive algorithm maintains the shape of the energy contour. Formal evaluations of the effectiveness of this approach are underway.

3. FUTURE WORK

In this quarter, we have delivered a recognition server demo within the DARPA Communicator architecture that runs ~10 xRT. Performance was shown to be close to our best non-real-time system. In the next quarter we will continue optimizing the real-time system to reduce the degradation in performance and enhance its flexibility. We will focus on better ways to compute

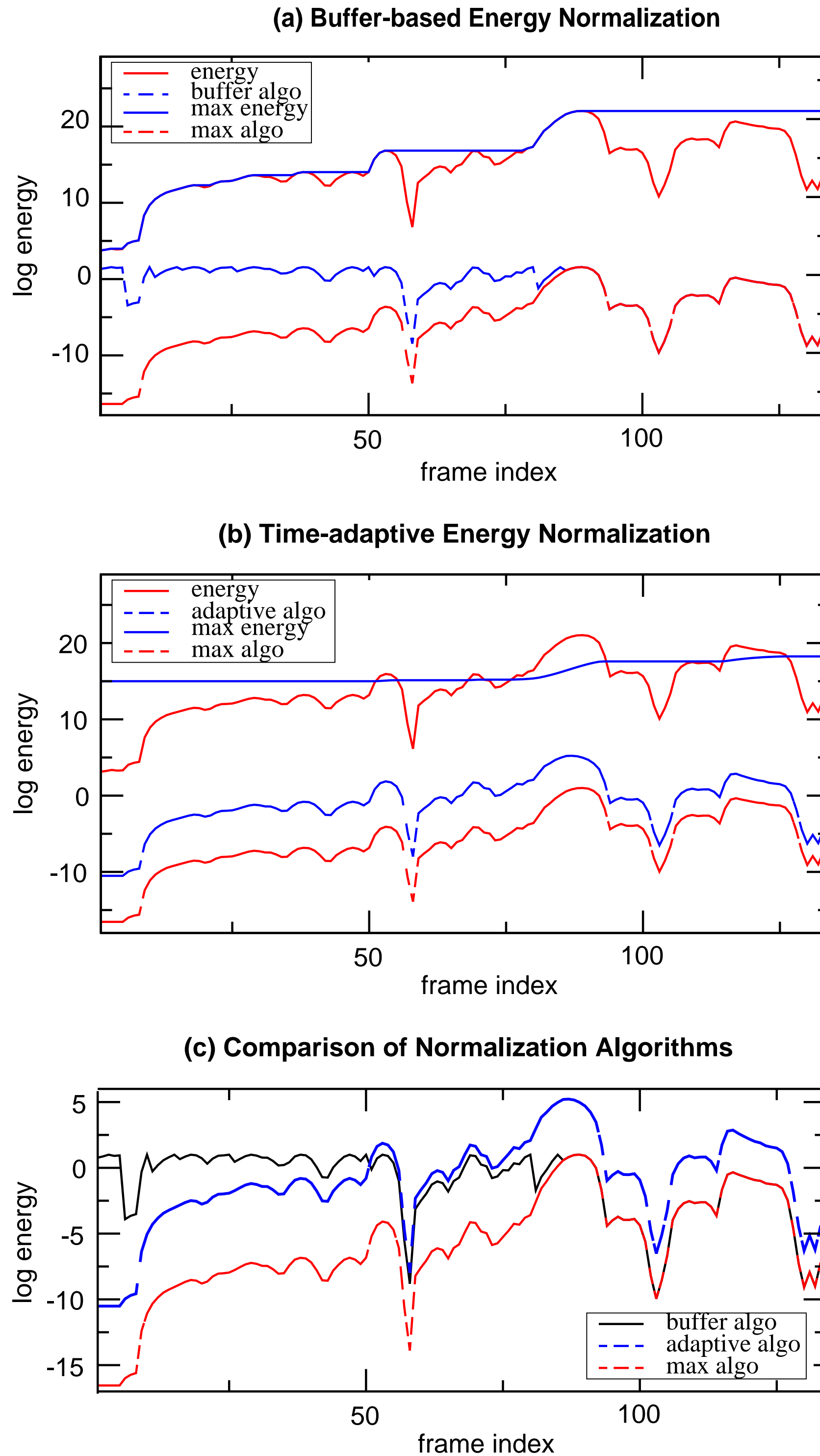


Figure 1. A comparison of an adaptive real-time energy calculation to our standard non-real-time approach. In (a), we compare the **buffer** and **max** energy normalization approaches; in (b) we demonstrate the adaptive schemes; in (c) we compare the three different normalization approaches. The adaptive scheme works well as long as the nominal level of the signal is known a priori. This is normally determined by computing an average maximum energy across the entire training database.

the mean cepstral vectors with minimal delay. We will also refine our grammar switching capability and evaluate it within the context of the Communicator framework. We will need to develop a simple protocol to provide user interface engineers control of the grammar switching process.

Our primary task in the next quarter will be to focus on the SPINE evaluations. Last year, we also took part in these evaluations. Due to time limitations, we submitted a very basic system [8], which achieved an error rate of 56.2%. This year we are planning to demonstrate the robustness of our generic system: two-pass decoding using triphone crossword acoustic models with trigram language models. We also plan to explore the robustness of our Support Vector Machine (SVM) approach [9].

4. ACKNOWLEDGMENTS

We wish to acknowledge Drs. Fred J. Goodman, Bryan George, and George Shuttic of the Signal Processing Center at MITRE Corporation for their continued support and feedback.

5. REFERENCES

[1] “Galaxy Communicator,” *http://communicator.sourceforge.net,* MITRE Corporation, McLean, Virginia, USA, June 2001.

[2] P. Price, W. Fisher, J. Bernstein, and D. Pallett, “The DARPA 1000-Word Resource Management Database for continuous speech recognition,” *IEEE International Conference on Acoustics, Speech, and signal Processing*, pp. 651-654, 1988.

[3] F. Zheng and J. Picone, “Robust Low Perplexity Voice Interfaces,” *MITRE Corporation,* May 15, 2001.

[4] N. Deshmukh, A. Ganapathiraju, J. Hamaker, J. Picone and M. Ordowski, “A Public Domain Speech-to-Text System,” *Proceedings of the 6th European Conference on Speech Communication and Technology*, vol. 5, pp. 2127-2130, Budapest, Hungary, September 1999.

[5] “Prototype System v5.10,” *http://www.isip.msstate.edu/projects/speech/software/asr/download/asr/isip_proto_v5.10.tar.gz,* Institute for Signal and Information Processing, Mississippi State University, Mississippi State, Mississippi, USA, July 2001.

[6] “The CU Communicator,” *http://communicator.colorado.edu/*, The Center for Spoken Language Research, University of Colorado, Boulder, Colorado, USA, July 2001.

[7] S. J. Orfranidis, *Introduction to Signal Processing,* Prentice-Hall Publishing, Englewood Cliffs, New Jersey, 1996.

[8] B. George, B. Necioglu, J. Picone, G. Shuttic, and R. Sundaram, “*The 2000 NRL Evaluation for Recognition of Speech in Noisy Environments,*” presented at the SPINE Workshop,

Naval Research Laboratory, Alexandria, Virginia, USA, October, 2000.

[9] A. Ganapathiraju, J. Hamaker and J. Picone, “A Hybrid ASR System Using Support Vector Machines,” *Proceedings of the International Conference of Spoken Language Processing,* vol. 4, pp. 504-507, Beijing, China, October 2000.

APPENDIX A. INSTALLATION INSTRUCTIONS

This page describes how to download and install a recognition server based on our prototype system that integrates with the DARPA Communicator client/server architecture. This server decodes audio data and displays the resulting hypothesis in a text window. A demo program is included that makes it easy to get started.

There are five easy steps to Communicator bliss:

- Step 1: Install the DARPA Communicator system.
- Step 2: Install the ISIP Prototype System.
- Step 3: Install the recognition demo server.
- Step 4: Run the demo.
- Step 5: Modify the hub program file.

Step 1: Install the DARPA Communicator system

In order to run this server, one must have previously installed the DARPA Communicator [4] system and be somewhat familiar with its operation.

Step 2: Install the prototype system

Download the latest version of the ISIP prototype system [1] in which we provide an integrated API for the DARPA Communicator system. The system can be easily installed following the steps below:

- tar xzvf isip_proto_v5.9.tar.gz
- cd isip_proto
- ./configure --prefix=.
- source ISIP_ENV.sh
- make
- make install

In addition to installing the recognition binaries, this step builds a library containing the API that will be linked into the applications described below.

Step 3: Install the recognition server demo

Once the DARPA Communicator and ISIP prototype systems have been successfully installed, we can download the recognition server demo and proceed with building the server program:

- tar xzvf rec_demo_v1_0.tar.gz
- ln -s rec_demo_v1_0 rec_demo

- cd rec_demo
- ./configure [--prefix=Communicator install directory]
- source GC_ENV.sh
- make

Note that the --prefix option is used to specify the directory in which you installed Communicator. If you have set the Communicator environment variable, ***GC_ROOT***, you do not need to use the *--prefix* option. The default directory for the Communicator installation is */usr/local/communicator.*

Before compiling the recognition server, make sure you have sourced the ISIP environment (*source ISIP_ENV.sh*) to establish the proper run-time environment for the prototype system.

Step 4: Run the application demo

Steps 1 to 3 above must have been successfully completed for the next step to work. The recognition demo program can now be invoked as follows:

- ./recognizer.csh

Before running the recognition server, make sure you have sourced the Communicator environment in step 3 (*source GC_ENV.sh*) to establish the necessary environment.

Now you are finall ready to run an audio server that will send data to the recognizer in batch mode:

- ./demo.csh

The audio demo server reads data from a file and sends it to the recognition server. The recognition server decodes the audio data and displays the hypothesis at the end of the utterance in a text window.

Step 5: Modify the hub program file

The demo_pgm.text can be easily modified to operate in your local environment:

```
;; ------------------------------------------------
;;   SERVERS
;; ------------------------------------------------

SERVER: recognize_serv
;; HOST: 130.18.6.111
HOST: localhost
PORT: 12346
OPERATIONS: reinitialize recognize_speech
```

;; ---
;;   CONDITIONS (default order)
;; ---

RULE: :binary_port & :binary_host & :call_id --> recognizer_speech
IN: :binary_port :binary_host :call_id
OUT: none!

The tag **recognize_speech** handles incoming audio broker requests, decodes the received audio data and outputs the hypothesis. Currently, the recognizer expects the broker to send GAL_INT_16 or GAL_BINARY data and expects the audio server to send the following control messages of type GAL_STRING:

- "new_utt": audio is about to be sent
- "done": no more samples will be sent (end of utterance)

Note: A demonstration system based on a real-time Resource Management system is also included in the *final_model/* directory of the demo release. The acoustic models for this system were trained on 16kHz sampled data. Hence, it is important that the Audio server sends compatible sampled data.

Success!

Kick back, smoke a victory cigar, and recite George Peppard’s famous line from the A-Team: "I love it when a plan comes together."