AD______________

Award Number: W81XWH-10-1-0391

TITLE: A System Solution for Voice Applications

PRINCIPAL INVESTIGATOR: Dr. Gregory Gadbois

CONTRACTING ORGANIZATION: Handheld Speech, LLC
Amesbury, MA 01913

REPORT DATE: December 2011

TYPE OF REPORT: Final

PREPARED FOR: U.S. Army Medical Research and Materiel Command
Fort Detrick, Maryland 21702-5012

DISTRIBUTION STATEMENT: Approved for public release; distribution unlimited

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* 01-12-2011	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* 7 Jul 2010 - 6 Nov 2011

4. TITLE AND SUBTITLE
A System Solution for Voice Applications

5a. CONTRACT NUMBER

5b. GRANT NUMBER
W81XWH-10-1-0391

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Dr. Gregory Gadbois

E-Mail: greg@handheldspeech.com

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Handheld Speech, LLC
Amesbury, MA 01913

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Medical Research and Materiel Command
Fort Detrick, Maryland 21702-5012

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This project was the first implementation of a system of speech applications sharing the recognition resource (via a Multi-Voice architecture). There were improvements in the recognition engine for more efficient running of multiple recognizers and to support the coordination of system level resources.

15. SUBJECT TERMS
a system solution of speech applications using Multi-Voice recognition

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** USAMRMC
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	12	**19b. TELEPHONE NUMBER** *(include area code)*

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Table of Contents

Introduction:

HandHeld Speech invented a new speech recognition architecture we call Multi-Voice. The Multi-Voice recognition architecture allows a simple design for a system service supporting multiple simultaneous speech applications. A recognition service of that design was created. But prior to this project that Multi-Voice service had only been used with applications running singley (not multiple simultaneously applications). Until this new service is used with simultaneous applications there will be uncorrected oversights in functionality and API's. The service will be immature until a real use of simultaneous speech applications is explored. The purpose of this project was to do that exploration. And as expected, there were oversights in functionality discovered and inefficiencies corrected.

The project needed multiple applications, the choice of applications was irrelevant from a research point of view. But, the more useful the applications are, the more they show how such a system solution is crucial for the future. So choosing relevant applications mattered. Making prototypes of real solutions to real problems underlines the strength of the Multi-Voice architecture. We partnered with Elintrix to improve the user interfaces of their mesh network radios. We chose two applications to run on their mobile devices, a map application and a low bandwidth communication application.

This project has been an engineering research project. It is an exploration of user interface ideas. The basic technology had been developed earlier. In this project we made that technology work.

Body:

Background – Multi-Voice Speech Recognition:

Prior to the Multi-Voice architecture, all speech recognition system services have been modeled on the keyboard service. With the keyboard service, one particular application has the focus. To illustrate the keyboard service, suppose there are two programs running, one a word processor and the other, an email client. And suppose one is currently writing an email. When an 'x' is typed, the 'x' goes into the email being written, not into to the word processor window. The email composition window has the "focus". To select the word processor window, that window must be tapped, bringing it to the foreground, giving it the "focus", before the typing will go into that window. The window with the focus gets the keyboard input.

In a similar manner, only one application at a time had the recognition engine's "focus". The application with the focus got all the recognition events. If two applications are alive and they both want to interact with the speech recognition service, only one of them wins, the one with the "focus". You must tap the window of the other application, bringing it to the foreground, allowing it to capture the "focus", making it the new destination for speech input. If the wrong application had the focus, when the user speaks, things go awry. Worse some applications, periodically set the speech focus back to themselves, just in case they had lost it. Multiple voice aware applications don't "play well together" (they fight over the resource). Except for our new Multi-Voice service, that is the current state of speech recognition services.

In a completely different domain, we at HandHeld Speech were trying to find a better solution for speaker independence. We developed a new architecture for a speech recognition engine. We designed a solution where we run multiple recognizers in parallel. Each recognizer is speaker dependent (they only model one speaker's voice). If we have a catalog of models covering the spectrum of voices, using all of them simultaneously, we achieve speaker-independent recognition. When a speaker says something, all the recognizers run in parallel, the models that best fit the speaker "wins", and accurate recognition results.

The clever part of this process was to do it efficiently. We use competition to limit the computation. The recognizers must be normalized so that they compete fairly. To illustrate, suppose there are two recognizers running, two sets of voice models, a deep male voice and a high female voice. Suppose the speaker says a four word command. As he begins speaking, both recognizers are getting data. If the speaker actually has a deep male voice, by the time he finishes saying the first word, there is a hypothesis using the deep male models that is scoring very well. By comparison, there are no hypotheses using the female models that score anywhere close to the "good" deep male hypothesis. We prune hypotheses that are scoring very much worse than the "best" hypothesis. Very quickly all the hypotheses using the high female voice models, die. When the last hypothesis is pruned, we prune the whole recognizer. In this way we can start recognition with an array of recognizers running. Assuming one of them accurately models what is being spoken, the "correct" modeling will prune all the competing recognizers. Very quickly (in the first couple syllables) all the recognizers but one will be pruned. The process is very efficient. We can run our recognizer in this manner on small hardware (for example the hardware of a smartphone). The best detailed explanation of our Multi-Voice recognize is our patent, see reference 1.

Returning to the main point, after inventing this architecture, we realized there was a user interface innovation to explore. If we developed a system service that could "talk to" multiple programs simultaneously, we could allow each program to set up its own recognition problem, allocate as many recognizers it wanted to use, and then let them all run simultaneously. The recognizers would compete, the best modeling solution would win. When an utterance is spoken, we can look at the winning hypothesis and determine which application won, and notify that application of the recognition results. This is a new architecture to the recognition system service.

We compare this new architecture to the keyboard service. You may be running many programs that can accept keyboard input. When you type an 's', the keyboard service can't look at the value being typed (oh! this is an 's') and decide which application to give the letter to. But a speech recognition service can. It can decide very late, which application should receive the result. When you give a command to a GPS map application, it won't make sense as a medical diagnostic (competing with a medic form filling application). Similarly when you say "blood pressure 140 over 70", that won't be a sensible map command. The results of the recognition can be used to decide which program should receive those results.

This opens the door to making the speech recognition service a task switcher. The user selects which application he has in the foreground by what he says. By talking, he pulls the application that he wants to use to the foreground. The user can launch a collection of applications and speak to any of them at any time. The "right" application comes alive when it hears the command that is relevant to it. The recognition service insulates the application from commands that are not relevant to it. Better still, no application needs to "know" about the others. They can be written as if they are the only application alive that is using the recognition service. The applications are self contained and modular.

We envision a system solution where the user launches a "swarm" of applications. In the process of doing a job he speaks and the right application in the "swarm" surfaces and responds to his input. Different people may have different sets of applications in their "swarms" depending on their jobs.

The new elements of the user interface is hands free task switching and seamless simultaneous voice access to multiple applications. This is a new form of multitasking.

The next step was to create this system service and a small "swarm" of applications to test this use. This brings us to the current project. For this project we partnered with Elintrix to create a map application and a low bandwidth communication application to run on their mesh network radios.

Elintrix:

Elintrix had developed low power mesh-network radios. Their original purpose was to monitor the health of soldiers during training exercises. The monitors could non-intrusively monitor things like pulse rate, respiration, and the drinking of fluids. It would periodically send a block of measurements over the mesh-network to the instructor. On the instructor's device, alarms would be set so that should a soldier have unhealthy measurements, the instructor would be notified. The amount of data being sent through the mesh-network would be small. The required bandwidth would be low. As such, the monitors could be made very low power and have very long battery life. Devices of this type were designed. They had very low bandwidth. They were quite small with no video screen and at most a couple buttons to interact with.

At that point, it was conceived that these health monitors would be more useful if the mesh-network could also be used for communication. The low bandwidth of the radios precluded their use as walkie-talkies. The next idea was to use speech recognition and text to speech as a means of reducing bandwidth. This is where HandHeld Speech joined the development effort. They wished for a low bandwidth communication application. Our requirement was for at least two speech applications. So they then suggested there was also a map application which the instructor uses to keep track of his soldiers. They suggested it would be good to enhance that application's user interface with speech commands.

So this project was formally initiated with a speech enabled Map application and a Low Bandwidth Communication application as its goal. Making the two applications work well together would prove the Multi-Voice swarm of apps multitasking architecture. And in addition we would create a prototype device solution compelling in its own right.

The Low Bandwidth Communication Application:

The mesh-network was intentionally created to only support moving small amounts of data. With the low bandwidth came low power and long battery life. By the original specification, there is not enough bandwidth to support the streaming audio of a walkie-talkie.

The initial idea was that when the instructor had something to say to one of his soldiers:

1) he would just say it, then
2) the speech recognizer would create the text of what he said, next
3) the text would be shipped over the mesh-network to the soldiers monitor, there
4) a text to speech engine would recreate a verbal command.

This idea did not take into account that the speech recognition is taking place outdoors; that the speech recognition is not happening in a pristine quiet office environment. Accurate open recognition in noise on a mobile device is beyond current state of the art. We must limit the application to a fixed set of commands. Given a fixed command set, the challenges of the application changes. The biggest hurdles will be providing the "right" set of commands and also having those commands be "learn-able" by the instructor. In general, the command set must be small otherwise it becomes difficult for the instructor to learn. Also, a small command set improves recognition accuracy.

The communication application we wrote attempts to address these issues. Everything is operable by either touch or voice. The commands are the text on the screen. You can select a command by speaking it or by touching it. When you are new to the application, you operate it by touch. Then as you become familiar with it you can operate it (eyes and hands free) by voice.

There remains the problem of having the "right" command set. We did not work too hard to create the "right" commands in this iteration. Rather we set up the application so that it would be easy to change the command set to whatever would be required. The application runs after loading the command set from an easily editable text file. If you change the text file, you change the command set. The hope is that we will empower the people close to the real use, so that they can create the proper command set.

The text file allows the definition of a state machine. The command set is envisioned as a set of questions and their answers. There is an initial question and a set of accepted answers. The recognizer will be listening for those answers. When a particular answer is accepted, the text file defines what the next question will be. The text file can be thought of as defining a system of menus and sub-menus. In this prototype, someone wanting to change the command set/state machine will have to become familiar with the syntax of the text file. In the future, we can imagine graphical tools that would allow a naïve user to edit the command set.

We give a brief discussion of the current CommApp.tsv (the text file driving the command set). The purpose of this explanation is to show the current command set and to suggest how it is easily changed. It is not meant to be a user's manual.

There are currently only two states. The 'start' state only allows you to select the radio that will be the recipient of following commands. Transitioning from the 'start' state is the 'command' state where all the commands are active (including selecting a different radio). All the comands of the 'command' state transition to the 'command' state. Below is an excerpt from the commApp.tsv file:

```
q1   "radio"          "which radio"
a    "all radios"                                     q2   setAllRadios
a    "radio &XXX"                                     q2   setRadio

q2   "command"        "what command or which radio"
a    "all radios"                                     q2   setAllRadios
a    "radio &XXX"                                     q2   setRadio
a    "drink [more] water [and acknowledge]"           q2   outRadio
a    "break for {5 10} [and acknowledge]"             q2   outRadio
a    "stop [and acknowledge]"                         q2   outRadio'
a    "head {north south east west} [and acknowledge]" q2   outRadio
```

The app always starts with q1 (question 1). There are 2 strings associated with any question, a short version of the command and a long version of the command. There are options in the app to play these strings so that the operator can "hear where he is" and so know what type of answers the app is expecting. Following the question are the expected answers.

An answer has a string that is the words of the answer. There are some special symbols accepted in these strings. The { ... } means an alternative, any one of the things in curly braces. The [...] means an optionally sequence. Following that string is the question id of the next state. And last is the action associated with this answer.

The &XXX is a place holder for the list of radio id's. The app builds and rebuilds that list automatically as the set of known radio id's changes. For example if radios 123, 456 and 291 are in the net, &XXX becomes { 123 456 291 } (where they are spoken as digit strings, 123 is pronounced one two three).

There is one last set of special symbol accepted in the answers they are (...). These allow a sequence of words to be taken as a whole. For example, the { north south east west } in the last answer might be replaced with { east west (north [{ east west }]) (south [{ east west }]) }. Then in addition to north south east and west, northeast and northwest, southeast and southwest are accepted answers.

There are some automatic CommApp commands that do not need to be set up by the command.tsv file. They are:

“Short Questions”

“Long Questions”

“No Questions”

“Train Last Command”

“Echo On”

“No Echo”

“Send It”

The Map Application:

Elintrix had already written there own map application. The challenge was to retrofit it with speech recognition. The hope was to change it as little as possible and to return the code base back to Elintrix empowering them to evolve it as they saw fit. We needed to keep the speech part as modular as possible and to change the architecture of the map application as little as possible. When we surrendered the code base back to the Elintrix developers, they would see the changes and could continue development.

The things they wished to voice enable were to change the view or to show a particular soldier on the map on demand. For example, suppose an alarm goes off saying soldier '123' is getting dehydrated. The instructor doesn't want to have to look all over the map to try to figure out where that soldier is. He uses the voice command "focus on 123" and application shows him that soldier.

This part of the project is very much a success. The Elintrix developers are in charge and are evolving the application as they see fit. They are adding their own voice commands. Less and less are we being contacted to ask how to do something. Better still they are getting a feel for what should be voice driven and what is better to leave to the touch interface.

The challenge of disturbing the existing map code base as little as possible was a key constraint. It caused important evolution to the "swarm of applications" functionality.

The Evolution of the Recognition System Services:

The CommApp was written first and was designed similar to previous single-tasking applications (see reference 2 for another such application). It was in retrofitting the MapApp that we were forced to think about a real evolution of the system services. What drove the evolution was the desire to add as little code as possible to the MapApp code base. The code base needed to be returned to the Elintrix engineers with as few changes as possible.

The first piece of code purposely excluded from the MapApp was model selection and enrollment. It would add whole modules that hadn't existed in their code base. Instead a stand-alone enrollment app is loaded on demand, and goes away when finished. Having it be a separate application lowers the total memory footprint of running multiple apps (an important feature for small hardware) With a small number of command line arguments, the launching application controls the enrollment application. The most important argument is the name of a text file containing the prompts to use during enrollment. With that, an app specifies the enrollment vocabulary. To communicate success, the enrollment app must return the successfully enrolled models. Instead of trying to capture a return value from the process, it made more sense to allow the recognition service itself to own a set of "default user models".

In many instances, applications don't have a particular need for "special" models, in fact if there are some known "good" models, it makes the most sense if all applications "share" the same best set of models. It will use less memory (not creating multiple copies of the same models) and there is also a computational efficiency during recognition if models are shared across applications. So an array of DefualtUser[ ] models was added to the system service, initially loaded with nulls. An application can ask for default user models at a particular index and will receive either a null or a pointer to models. An application can also tell the demon to load a set of models to a particular default model index. If models already exist at that location, they will be trampled, otherwise new models will be created and loaded. We suggest in the documentation that the Language Id be used for the index. For example, because ENGLISH_LANGID = 1, if an application would like to use the default English language models, he should ask for the default models at index 1. If he would like to set English default models, he should set them at index 1. An argument to the enrollment app is the index of where the newly enrolled default models should be stored. So when an app is launched it can look for existing English models, if there are none, it can launch the EnrolApp with a prompt file and the index '1' (telling it to store created models as the default English models). Then when enrollment finishes, the app can look for the new models at index 1. If they exist it can load them, if it finds a null, it can abort knowing enrollment was aborted. Using this facility when a swarm of applications is launched, if the first application sets up default models, the next apps have the choice of using those default models or not. If a 'next' app has a generic need for models, it should use the defaults. If it has a specialized purpose with a need for specific models, it can load its own private models. After creating this functionality for the MapApp, the CommApp was simplified to use it too.

Separate from enrollment, when retrofitting speech into the MapApp, we realized there were a number of other minor functionalities being duplicated in the CommApp and MapApp. The solution was to create a utility application that had those feature exposed once and for all. For example, it is important to surface a control for running half-duplex, but it need only be exposed once. Half-duplex is critical for some devices, some devices don't simultaneously support both and sound in and sound out. The mic must be toggled around playing .wav files. If half-duplex is selected, the service will automatically toggle the mic when a .wav file is to be played. Also when the mic is too close to a speaker, the mic will pick up the speaker output and think the user is speaking. Selecting half-duplex solves this unintended feedback problem.

Another functionality that is important that the system have is the facility of putting the speech recognition into hibernation, of being able to say “go to sleep”. It is very irritating when applications spontaneously do things when you are not talking to them (talking to someone else instead). If done correctly, a “sleeping” functionality need only be implement by one app and all the other apps are freed from implementing separate versions.

These two functionalities are in the current utility app. The main apps launch the utility app after they start up. The utility app will detect if it is already running and a second copy will immediately quit if it is multiply launched. We added support to the recognition service so that the utility app registers with the recognition service as a utility. When the last main app exits, the recognition service tells the utility app to quit. So, transparent to the user, the utility comes up with the main apps and goes away when the last app exits.

What remains in the both the MapApp and the CommApp is just the recognition of their respective command sets and the live training (adaptation) of those commands. The task for the developer is much more focused. He doesn't need to implement a lot of ancillary functionality anymore.

The utility app is a general place to put shared system level functionality. We can imagine more functional control of the recognition engine may one day be exposed through it. But we will await clear needs before new features are added.

Key Research Accomplishments:

Multiple applications are simultaneously defining speech recognition problems and sharing a speech recognition service. They are running as efficiently as if they were a single application. But the key feature is that the writing of these applications is simple. The application developer doesn't have to concern himself with the other apps. Each app is designed as if it were the only consumer of speech. The recognition engine does all the housekeeping, insulating each app from all the others. The system is simple and expandable. A monolithic speech application that "knows the world" is not required. The concept of "a swarm of apps" is proven. The computer's user interface has taken a big step forward.

Reportable Outcomes:

There are three distinct reportable outcomes for this project.

The first is that the two apps, the CommApp and the MapApp run seamlessly together. And they run efficiently together. The concept of "the swarm of apps" is a reality. Because there now exists separate enrollment and utility apps, the developer's task is smaller and more focused. Building speech recognition into application user interfaces has gotten simpler.

The other key accomplishment is that the Elintrix programmers were empowered. They easily grasped how to run the speech engine and how to change the source code to do additional speech commands. It is a success for the coherency and simplicity of the API's. The system is transferable.

Finally the addition of the voice driven example tasks to the Elintrix mesh-network hardware will make the Elintrix health monitors a compelling solution in its own right. The ease of use in their real tasks, and the ease that they provide an expandable open system will be critical to their winning deployment.

Conclusion:

The Conclusion is little different from the "Key Research Accomplishments" and "Reportable Outcomes". The primary conclusion is that "the swarm of applications" concept for the user interface has taken a big step forward. The only thing stopping us from launching a dozen apps and having them share the speech recognition service is there does not exist a dozen apps written to these API's.

The secondary conclusion is that it is not that difficult to write these apps. We have successfully transferred the knowledge of writing them to the Elintrix team. And they found that instruction not difficult.

References:

Greg Gadbois (2011) "Multi-Voice Speech Recognition". US patent 7899669.

Greg Gadbois (2011) "Developing Multi-Voice Speech Recognition Confidence Measures and Applying them to AHLTA-Mobile". Final report of contract W81XH-09-2-0158 to the U.S. Army Medical Research and Materiel Command.