EMBEDDING SPEECH IN WEB INTERFACES

Samuel Bayer

The MITRE Corporation, M/S K329
202 Burlington Rd.
Bedford, MA 01730
sam@mitre.org

ABSTRACT

In this paper, we will describe work in progress at the MITRE Corporation on embedding speech-enabled interfaces in Web browsers. This research is part of our work to establish the infrastructure to create Web-hosted versions of prototype multimodal interfaces, both intelligent and otherwise. Like many others, we believe that the Web is the best potential delivery and distribution vehicle for complex software applications, and that the functionality of these Web-hosted applications should match the functionality available in standalone applications. In this paper, we will discuss our approach to several aspects of this goal.

1. INTRODUCTION

For many years, speech interaction has been seen as a critical aspect of multimodal human-computer interaction, along with more traditional graphical tools (for a summary of the range of such tools, see [4]). Advances in Internet technologies have increased considerably the range of "host" options for multimodal interfaces, moving from standalone applications to Web-based alternatives such as locally-based "helper" applications spawned by document type, HTML-based interfaces to remote applications driven by CGI scripts (for a particularly sophisticated example, see [7]), Web-hosted applications in languages like Java or Python enabled by mobile code technology, and more recently, locally-based applications set up as libraries to "plug in" to browsers like Netscape.

The HCI group at the MITRE Corporation has been involved for many years in the design and construction of advanced multimodal interfaces involving language understanding and generation, speech recognition and synthesis, interpretation of deixis and direct manipulation, and graphic design and presentation. One of our current research goals is to duplicate, in Web-hosted interactions, the entire range of communication supported in our previous work. In this paper, we'll describe some of the components of this research program, concentrating on the role of speech.

2. COMPARISON WITH PREVIOUS WORK

Talking to Web browsers is not a new idea. For instance, Novick and House [6] describe a minimal modification of NCSA Mosaic which allows names of links to be selected by being spoken. Since then, "remote control" of Web browsers has made it possible to "drive" a browser without modifying it at all, by opening URLs, new windows, etc. through a limited external command language. However, there is a fundamental difference between this sort of speech-enabled browser behavior and being able to address Web-hosted applications themselves. The reason is that the Web browser need not even know that it's being spoken to, since the speech interaction itself is an application completely external to the browser. The speech "proxy" reads the page being displayed via the browser's "remote control" language, parses it with its own HTML parser, creates its speech model, digests the user's speech, translates it back into the "remote control" language, and relays the command to the browser. For all the browser knows, the "remote control" command was typed in directly to the shell.

This situation contrasts fairly dramatically with the case of a multimodal application written in Java or Python which is downloaded in the course of Web navigation. If this application desires speech input, it (and not some external application) must somehow enable a connection between it, some speech recognition service, and a microphone connected to the user's console; it (and not some external application) must ensure that the speech recognition service is aware of the application's language model; and it (and not some external application) must ensure that the appearance of the browser's viewer conforms to the requests the user makes. Furthermore, these requests are specific to the downloaded application, not general to the browser or the syntax of HTML. In other words, speech input to Web browsers can remain external only as long as the addition of speech input does not depend at all on the hosted application itself. At that point, speech input must be enabled and driven from inside the browser, not from outside. So while it might be possible, for instance, to drive a forms interface via speech by using an external application to digest the contents of an HTML page which describes the forms interface and allow the user to navigate this interface with voice commands, and while it might even be possible to signal the availability of speech to the user within the Web browser by replacing the current page (once digested) with a slightly "edited" page which describes the available speech capabilities, that's as far as this technology goes.

3. EXAMPLE APPLICATIONS

The MITRE Corporation has been involved in research in intelligent human-computer interaction for approximately ten years. Our initial NL understanding system, King Kong, and its multimodal sibling AIMI (An Intelligent Multimodal Interface) ([2], [3]) are *completely interpreted* interfaces. That is, virtually every input and output is translated into (or out of) a rich, medium-independent semantic representation which exploits a model of the application domain. These systems share this property with systems like CUBRICON [5], CHORIS [10], etc. More recently, we have turned back to traditional "dumb" graphical interfaces, with an interest in developing strategies for migrating from these systems to their intelligent counterparts.

So far, our experiments in Web-hosted speech services have encompassed an example of each type of system. The Intelligent Project Planner (IPP) enables the user to get information about employees, their charge distributions, schedules, supervisors, travel plans, etc., via graphical and natural language interaction. The On Line Library Interface (OLLI), on the other hand, is a standard graphical interface to on-line card catalogs, which we've enhanced with limited speech functionality. Our initial experiences with migrating OLLI beyond standard current capabilities is documented in [1].

4. ENABLING SPEECH FROM WITHIN WEB BROWSERS

There is more than one way to extend the run-time behavior of Web browsers. The first is via mobile code. A mobile code language is nothing more than a programming language with an accompanying infrastructure for shipping and executing source-level or byte-code-level code on a variety of platforms in real time. Currently, the most common infrastructures for this code distribution are Web browsers. Examples are Java, supported by Netscape and Sun's HotJava browser, and Python, supported by CNRI's Grail browser. Special HTML tags are interpreted as instructions to download code (called an *applet*) and run it, rather than display it as text or images. Using such technology, it is possible to download applets which implement arbitrary window configurations within the browser's viewer, and associates arbitrary behavior with these interfaces.

There is another way to supply arbitrary code to Web browsers, and this is through so-called plug-ins. These are libraries which can be loaded at run-time and accessed as part of the browser's installed capabilities. These plug-ins are most useful as "enabling technology" which can be repeatedly reused, as opposed to dedicated, specific applications, which are more appropriate to mobile code. For instance, plug-ins for Netscape include document readers such as Adobe Acrobat for displaying different document types, and plug-ins for Grail can enable new file protocols, such as CNRI's handle protocol. As an added twist, it's also possible via plug-ins to enable Netscape to handle additional mobile code languages; this has already been done for Python in Netscape under Windows.

There are a number of ways to use these facilities to supply speech services to Web-hosted applications. The first issue to address is whether the speech services should be "inside" or "outside" the Web browser. If the answer is "inside", these services should probably be provided via plug-ins, since they are intended to be reusable, the code which implements the service tends to be large, and at least portions of the service are not amenable to mobile code because they are platform-dependent (for instance, those which deal with connecting to the local audio device, for which there are no standards as yet). While at least one recent Macintosh-based approach has opted for this strategy, we suspect that in the long run, many applications in addition to those hosted by the browser will desire speech services, and it seems more useful to implement them "outside" the browser. At this point, we must address an additional issue: how the application within the browser accesses these services. We will discuss this issue in the next section.

5. THE IMPLEMENTATION OF DISTRIBUTED OBJECT-ORIENTED SERVICES

Throughout this paper, we've talked about speech *services*, and we hope to give this notion substance in both the conceptual and implementational sense. For instance, MITRE's Glacier speech recognition system can be configured as a garden-variety server, which accepts and manages multiple connections from multiple applications. Each application notifies the speech server of its language model, and the speech server processes audio input according to these various language models and selects a "winner" to receive the speech input according to some (potentially complex) algorithm.[1] The initial implementation of the Glacier speech server uses UNIX message queues for communication between client and server, with the ensuing restriction that client and server must be on the same machine.

In practice, this sort of restriction is potentially inconvenient. In communicating with the speech server, we encounter all the problems which typify traditional client-server communication. Some communications layers, like message queues or pipes, require client and server to be on the same machine; others, like sockets, do not, but require, in the most general case, that the implementer develop a low-level protocol which the client and server "speak" to each other. When a standard protocol such as RPC is introduced above the socket layer, it is typically not shared across platform types, and the client is still required to know the location of the server. In general, what is desired is a service model which is transparent with respect to location, machine and OS type, and implementation language.

One answer to this set of problems is an architecture for distributed object services such as OMG's CORBA. In such an architecture, an *object request broker* (ORB) accepts client requests for services and provides services in the form of *object proxies*, location-, language-, and platform-independent pointers to the servers themselves. Clients use the types and object models of their own language (C++, Java, Common Lisp, Modula 3, Smalltalk, Python,

[1] Currently, Glacier's algorithm is a straightforward comparison of output scores from the speech recognition itself, but in other work with CMU's Janus recognizer server we are experimenting with algorithms guided at least partially by mouse and keyboard focus.

```
TYPE SpeechDevice = OBJECT
        METHODS
        Subscribe (app : SpeechAwareApplication,
                   path : String)
                RAISES InaccessibleServerError,
                       ParameterFileAlreadyOpenedError,
                       SpeechServerFailureError END,
        AddExpansions (app : SpeechAwareApplication,
                       expansionName : String,
                       terminals : StringSequence)
                RAISES DeadServerError, DeadClientError, BadServerReplyError,
                       SpeechServerFailureError END,
        ClearExpansions (app : SpeechAwareApplication,
                         expansionName : String)
                RAISES DeadServerError, DeadClientError, BadServerReplyError,
                       SpeechServerFailureError END,
        Help (app : SpeechAwareApplication)
                RAISES DeadServerError, DeadClientError, BadServerReplyError,
                       SpeechServerFailureError END,
        Unsubscribe (app : SpeechAwareApplication)
                RAISES BadServerReplyError END
```

Figure 1: An abstract interface to a speech device.

etc.) to interact with the object proxies as if they were local objects. The *object interface* – that is, the set of methods each CORBA object exports – serves as the abstract "bandwidth" through which communication with the object happens; the particular CORBA implementation provides data translation and communications protocols.

Our implementation consists of three parts. First, we have adopted Python, an interpreted object-oriented scripting language, as our primary implementation language. Second, we have adopted Grail, a Web browser written in Python, as our application host. Finally, we have adopted the Inter-Language Unification (ILU) system, a CORBA-compatible ORB provided for free by Xerox PARC, as our distributed object infrastructure. ILU provides bindings for C, C++, Modula 3, Python, and Common Lisp, and its latest release supports the CORBA IIOP (Internet Inter-Operability Protocol), which enables communication among CORBA-compatible ORB implementations. With such a tool, we can approach our goal of having network-accessible speech services from any client implementation language.

How, then, do we make speech services available from Web-hosted applications? First, we endow the Web browser with an ILU substrate. This can be provided in one of two ways: either we make the browser ILU-aware via a plug-in or a compile-time library, or we provide the browser with the ILU substrate via mobile code. The latter is less desirable, but in the version of the Grail browser we conducted our experiments with, it was the most elegant solution. This latter strategy introduced another problem. While ILU provides bindings for the substrate in a variety of languages, the ILU substrate itself is written in C; in order to provide the substrate to Grail via mobile code, we needed a version of the ILU substrate which is written in the mobile code language supported by our browser (in this case, Python). Fortunately, a version for this purpose exists, called SYLU; this is the version we use.

Second, we define an abstract interface for speech interaction. A number of such proposals are currently being developed, among them the SRAPI specification [8]. However, these proposals are very narrowly tailored for near-term applications like dictation and menu navigation, and are not being developed with distributed object services in mind. We define a barebones abstract interface for the speech server in Figure 1. This protocol covers subscribing and unsubscribing to speech services, altering the application's grammar, and requesting speech help.

This protocol must be paired with a method for notifying the application about the status of speech input. In a true distributed object model, the application itself is also an object service, and among the methods it supports is the notification protocol shown in Figure 2. In such a model, the application obtains a proxy for the speech server and subscribes to speech, passing the server a proxy for the application itself. When the server detects the onset of speech, it notifies all applications, and when it detects the end of speech and determines a "winner", it notifies that application and notifies all other applications that they've lost. We have implemented this communications model in our OLLI interface.

6. SECURITY CONSIDERATIONS

So far, we've discussed two strategies for augmenting the capabilities of Web browsers: via plug-ins and via mobile code. The latter, ironically enough, is as dangerous as it is convenient. In principle, mobile code presents the same threat to the client computer that any

```
TYPE SpeechAwareApplication = OBJECT
        METHODS
        SpeechStart (),
        SpeechLose (),
        SpeechEnd (),
        DeadServer (),
        SpeechResult (input : String)
        END;
```

Figure 2: An abstract interface to a speech-aware application.

installed, untrusted code does. Allowed to run without restriction, a mobile code application can freeze the keyboard, erase and corrupt the local disk, etc.; therefore, restrictions have been imposed on the execution of such code in Netscape, and to a lesser extent, Grail. Grail's Python interpreter operates in "restricted execution mode", which prohibits most interactions with the local operating system, such as opening and creating files, etc. In this way, the local machine is "protected". The Java security story is much more sophisticated; instead of arbitrarily restricted a particular set of operations, Java users can designate certain code as trusted, thereby relaxing the execution restrictions imposed on it (for details, see [9]).

As an example of the degree of subtlety required, consider an enormous (and intentional) security hole in Grail 0.2. We noted above that CORBA implementations take care of the underlying communications protocols. ILU's underlying protocol is RPC, which is in turn built on a socket layer. For an ILU-aware Web-hosted application to access local speech services, then, it must ultimately open a socket on a machine accessible to the console displaying the Web browser (perhaps on that console's machine itself) to communicate with the speech server via ILU. But this interaction is with the local operating system, which in principle ought to be disabled, since a applet could exploit security holes in any socket server protocol on the local machine. Thus the argument for trusted code: it's not the tools the applet uses, it's what it does with them that counts.

A number of considerations might contribute to the "trustworthiness" of interactions with the local machine. We know that *some* interactions between Web-hosted applications and the local machine are permitted; the most obvious examples are keyboard and mouse input and visual output, all of which may be (in fact, must be) accessed by Grail and the applets it runs. These devices are deemed trustworthy (or, perhaps more correctly, must be deemed trustworthy in order to have any interaction at all). In the long run, it might be possible to "license" other, higher-level devices as well, such as speech recognition.

Part of this task is standardization of the services provided by these high-level devices, as we have sketched above. For instance, there are at least two standardization efforts underway for speech recognition, including the Novell-led SRAPI consortium and Microsoft's SAPI effort. In addition, it is part of the long-range CORBA strategy to codify and support services at a range of levels, from simple naming services to components of tools like word-processing programs; certainly, at the level of sophistication intelligent multimodal interaction is attempting, only speech services are even close to being susceptible to this sort of codification. If access to such devices were deemed trustworthy and compiled into interface applications, much as keyboard and screen access is, the security threath posed by speech interaction in Web browsers would be reduced.

7. CONCLUSION

In our experiments, we have exploited other object services, including speech synthesis and language understanding. Indeed, interface services are merely one example of the potential impact of distributed object technology on Web applications. It is one of the most interesting, however, because in no other case is it necessary for the Web-hosted application to communicate with the client console "outside" the Web browser. We believe that this requirement is due to the absence of standardization for input and output devices besides mouse, keyboard and visual display, and that our grasp and exploitation of such alternative devices (even audio) is still, comparatively, in its infancy.

8. REFERENCES

1. BAYER, S., AND KOZIEROK, R. Bridging the gap between WIMP and intelligent interaction. Submitted to UIST, 1996.
2. BAYER, S., AND VILAIN, M. The relation-based knowledge representation of KING KONG. Presented at the AAAI Spring Symposium on Implemented Knowledge Representation Systems., 1990.
3. BURGER, J., AND MARSHALL, R. The application of natural language models to intelligent multimedia. In *Intelligent Multimedia Interfaces*, M. Maybury, Ed. AAAI Press, Menlo Park, 1993.
4. MAYBURY, M. Research in multimedia and multimodal parsing and generation. *Artificial Intelligence Review 9* (1995), 103–127.
5. NEAL, J., AND SHAPIRO, S. Intelligent multi-media interface technology. In *Intelligent User Interfaces*, J. Sullivan and S. Tyler, Eds., ACM Press Frontier Series. Addison-Wesley, Reading, MA, 1991, pp. 11–43.
6. NOVICK, D., AND HOUSE, D. Spoken language access to multimedia (SLAM): A multimodal interface to the World-Wide Web. Tech. Rep. Technical Report CSE-95-008, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, 1994.
7. RICE, J., FARQUHAR, A., PIERNOT, P., AND GRUBER, T. Lessons learned using the Web as an application interface. In *Proceedings of CHI 96* (1996).
8. SRAPI COMMITTEE. Speech recognition application program interface specification, version 5.0. Manuscript, 1994.
9. SUN MICROSYSTEMS, INC. HotJava: The security story. http://java.sun.com/1.0alpha3/doc/security/security.html, 1995.
10. TYLER, S., ET AL. An intelligent interface architecture for adaptive interaction. In *Intelligent User Interfaces*, J. Sullivan and S. Tyler, Eds., ACM Press Frontier Series. Addison-Wesley, Reading, MA, 1991, pp. 85–109.