A Portable, Server-Side Dialog Framework for VoiceXML

Bob Carpenter *Sasha Caskey* *Krishna Dayanidhi* *Caroline Drouin* *Roberto Pieraccini*

SpeechWorks International, Inc.
http://www.speechworks.com/

{ bob.carpenter, sasha.caskey, krishna.dayanidhi, caroline.drouin, roberto }@speechworks.com

ABSTRACT

We describe a spoken dialog application framework that combines the power and flexibility of server-side Java Servlets and Java Server Pages (JSPs) with the deployment portability, reliability and scalability of standard web (HTTP) servers and VoiceXML clients. Applications are developed by extending a framework of Java classes in order to define dialogs through lower level actions such as speech recognition, audio prompting, speech synthesis, and backend data access. The framework delegates session data management to servlets, embedding frame-based representations for the application's global and session data. Dialog flow is controlled through general constructions such as loops, conditionals, scoped sub-dialogs, along with scoped command, error, and exception handling. Prompting and grammars are configured through simple JSP templates that generate the VoiceXML instructions for the server to return to the client. The framework is designed to be extensible, as demonstrated by the implementation of customizable backup and repeat commands integrated with session data, command handling and grammar scoping.

VOICEXML CLIENTS & SERVERS

Like an HTML graphical web browser, a VoiceXML client proceeds by requesting data from a VoiceXML server using the HTTP protocol (see Figure 1). The server returns content in the form of a VoiceXML document [1]. The client interprets the document, which may include local computation with ECMAScript [2], synthesizing speech, playing pre-recorded prompts, and performing speech recognition [3]. As part of its VoiceXML processing, the client may make further requests to the same server or to other servers. The client may provide information about speech recognition results or local variables to the server along with a request for a new page. The server updates its state based on the client request and sends an HTTP response containing the appropriate VoiceXML document to the browser.

VoiceXML specifies its own dialog control, which controls prompting and synthesis, recognition, telephony event handling, and local ECMAScript evaluation. Control flow may also be transferred to a document specified by a URL either through a direct transfer or a subdialog invocation. Despite this rich set of control structures, it is impossible to build a sophisticated, data-driven application with all of its logic expressed in VoiceXML on the client side, because there is neither a way to store data across sessions, nor a way to access backend services and resources such as databases. Without non-standard extensions to a VoiceXML client, as in [4], it is not possible to provide personalized dialog flow, customized language models based on registered preferences, inventory availability information, or to record any data that persists after the call terminates. To facilitate data-driven applications, we choose to control the dialog on the server side, employing VoiceXML for its standardized, portable speech and telephony interface.

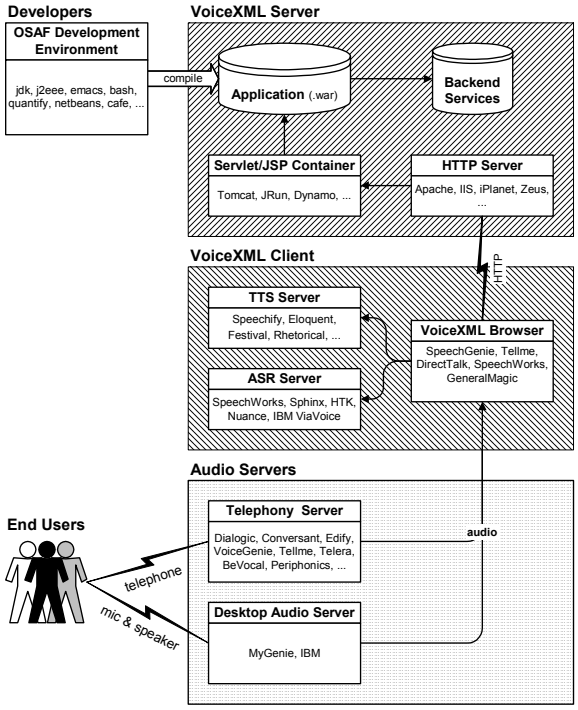

Figure 1: **VoiceXML Application Architecture**

SERVER-SIDE DIALOG CONTROL & SERVLETS

A fundamental challenge we face in implementing server-side dialogs for VoiceXML is that HTTP is stateless, in that each request is independent. In order to maintain session information, the client and server may exchange cookies as part of their HTTP requests and responses. Alternatively, session identifiers may be embedded directly in the URL of the request. Rather than supply a custom solution to this thorny problem, as in [4], we chose to take advantage of the portability provided by Java Servlets and Java Server Pages [5].

Servlets provide a powerful abstraction over HTTP not only by automatically handling the numerous details of the HTTP protocol itself, but also by maintaining ses-

sion data. A single instance of each servlet handles all of the requests for the URL to which it is assigned (so it must be thread safe and cannot store call-specific session information directly in member variables). Each request is handled by a method call to the servlet instance, an argument to which provides any data supplied by the client in the request and any session data on the server associated with the request. At the beginning of each session a fresh session object is created by the servlet container and initialized by the framework. Although the servlet response methods are quite general, the framework handles most of the details so we need only supply the text of the VoiceXML document for the client.

Servlets are managed by a servlet container, of which several are available, including the Tomcat, JRun, and Dynamo. The application provides the servlet container with compiled classes for each servlet, typically in a web archive (WAR) format. The servlet container manages the life cycle of servlets, constructing instances when they are needed and freeing them when they are not. JSPs are also included in the archive, but are transformed into servlets and then compiled on the fly by the servlet container. This archive format is portable across servlet containers, and the servlet containers themselves are typically portable across servers.

APPLICATION COMPONENTS

A significant advantage of VoiceXML is that an application can be hosted on a standard web server, such as Apache or IIS, with the telephony, speech recognition and speech synthesis being handled on the client side, which may even be maintained by a third party. In general, the relation between servers and clients is many to many, but applications may be deployed with all of the services shown in Figure 1 on a single machine. On the client side, VoiceXML browsers and audio platforms are available in many different configurations. Several browsers are available bundled with different speech recognizers, speech synthesizers, and telephony deployment options, including offerings by Tellme, VoiceGenie, BeVocal, and HeyAnita. MyGenie and DirectTalk are available on the desktop, and SpeechWorks provides an open source browser, VXI, for which Carnegie Mellon University has integrated Sphinx, its open-source recognizer, and Festival, a widely used open source speech synthesizer. VoiceGenie and IBM also offer platforms which bypass telephony altogether through desktop computer audio and internet access. This allows applications to be tested on the desktop before deployment without equipping each desktop with telephony equipment.

ETUDE DIALOG MANAGER

For specifying call flow, we employ the ETUDE dialog framework, which is more fully described in [6]. Rather than extend VoiceXML, as in [7], or require specialized servers, as in [4], we designed ETUDE in such a way that it could be controlled through servlets. Before we describe how that is accomplished, we describe the ETUDE framework itself. The class hierarchy of ETUDE is shown in Figure 2 in Unified Modeling Language (UML) format [8]. The fundamental building block of the framework is the `Action` interface, which contains an `execute` method that operates on the `Session` data. Developers will implement the interface in classes that handle both the interaction with the caller through speech and the interaction with the backend data services. A `Condition` supplies a `test` method over the session data returning a `boolean`.

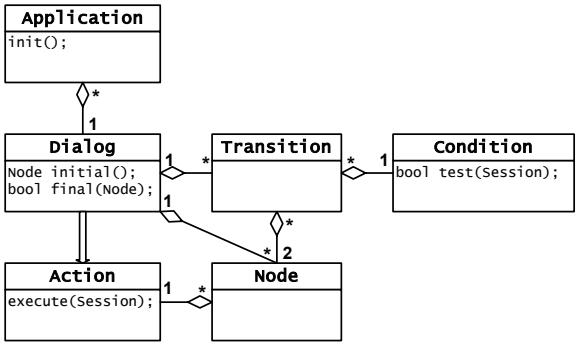

Figure 2: `Dialog` Classes

Following the standard pattern for generic data manipulation found not only in servlets themselves, but in dialog systems such as MITRE's Galaxy Communicator [9], our framework provides an associative-array-like data structure, called a `Frame`, for managing application session data. Typically an `Action` or `Condition` will only access the `Frame` portion of the `Session`. A `Frame` is a map from keys, which are of type `String`, to values, which implement the interface `Slot`. Instances of `Slot` are available as wrappers for strings, numbers (integer and floating point), booleans, other frames, and even arbitrary Java objects. The `Slot` interface handles copying and comparison, and other generic object bookkeeping details. A set of classes implementing `Action` is provided for manipulating frame values and a similar set of classes is provided implementing `Condition` for testing the contents of a `Frame`. A simple boolean expression language over frames is available for creating these conditions.

To produce VoiceXML output, the framework provides a class `OutputAction`, that implements `Action`. It is configured with the URL of a VoiceXML subdialog. As in Galaxy [9], with an `OutputAction` it is possible to specify the input parameters from the `Frame` and specify the location in the `Frame` to return the result parameters. The framework supplies templates to implement simple VoiceXML subdialog calls through JSPs, which are configured to pack up recognition and telephony events for

return to the server, which can then handle them appropriately as regular results, exceptions or errors.

Dialogs are created through the `Dialog` class, which extends `Action`. A `Dialog` consists of a number of `Node` and `Transition` objects, including one `Node` singled out as the initial node and any number of nodes specified as final nodes. A `Node` provides exactly one `Action`, whereas a `Transition` requires one source `Node`, one destination `Node` and a `Condition` on the session.

Developers do not write `execute` methods for dialogs or manipulate `Node`s independently of dialogs. Instead, they construct dialogs by specifying a set of nodes, each with its own action, and an ordered set of transitions that specify the flow of control between nodes. Execution of a dialog begins with its initial node being active. At each stage during execution, the active node's action is executed. After a node's action is executed, its transitions' conditions are evaluated in turn until one succeeds, at which point the new active node becomes the target of the transition. A dialog's execution terminates after the execution of one of its terminal nodes.

If the action to be executed for a node is itself a dialog, the framework pushes the currently active node onto its stack, which is stored along with the session data, and then executes the dialog constituting the node's action recursively. When the embedded dialog terminates execution, the stack is popped and the node containing the action becomes active so that its transitions may be evaluated. In the interest of modularity, dialogs constituting actions may be defined to work on their own local frame variables rather than on the frame of its containing dialog. As with the calls to external subdialogs through JSPs, the results of embedded dialogs on nodes on the server side are returned analogously to Galaxy [9], with mappings from local dialog's frame to the embedding dialog's frame.

An abstract `Application` class is provided, and the developer extends it by providing a method that returns a top-level dialog and by implementing an initialization method. The initialization method allows the management of any dialog-specific server-side resources, and a matching teardown method is also provided.

This flow of control, coupled with the isolation of output actions allows the server to maintain state and session data. The framework implements a top-level controller servlet that is configured with an `Application`, and controls the dialog by manipulating the Dialog objects directly. This servlet will be provided to the servlet container as the handler for requests to the dialog specified by the `Application`.

ERRORS AND EXCEPTIONS

Before evaluating ordinary transitions leaving nodes, ETUDE first determines if an exception has been thrown and not handled. When an exception is thrown, either by the underlying speech engine through a return value, or by an action in the framework, it is given a name. Actions are available to handle exceptions, and each node can have a special transition to another node based on the type of exception found. Typically, a dialog will handle all exceptions of a given type in a single place, for which convenience methods exist for specification, and then either eliminate the exception and recover, simply return, or throw the same or a different exception to a containing dialog. This allows for a general, scoped exception-handling mechanism, much like in Java itself.

REPEAT AND BACK-UP

In addition to the basic action/dialog architecture, ETUDE also supports several reusable patterns of call flow, including backup, repeat, and global command navigation, as well as scoped exception and error handling (see [6] for details). Any node in a dialog may be marked as a backup anchor. Every time a node marked as a backup anchor is visited, it is pushed on a stack of potential return points, which involves making a copy of the frame using the `Slot` interface. Any node may have backup activated, meaning that if backup command (whose grammar is configurable) is recognized in speech from the user, control will return to the node indicated on the top of the backup stack (excluding the current node). Repeat is similar, but does not require a stack, controls for the fact that only prompts can be repeated, and the current node is not excluded.

VALIDATION AND VISUALIZATION

Once a dialog has been specified and compiled, it will be automatically validated before it is used. Validation tests that all nodes are reachable, all transitions are on valid nodes, simple infinite loop transitions don't exist, at most one else-case transition exists per node, and that exceptions and commands are not handled twice.

In addition to validation, which is a very weak test, simply meant to supplement the compiler, we also provide a visualization of the node, action and transition structure of a dialog through GraphViz, an open-source graph rendering program [10] which produces output in a variety of formats, including postscript.

LOGGING

The VoiceXML specification provides details of client-side logging, but for applications run from a server, we need to be able to log a variety of information, ranging from framework or action errors (such as session invalidation or missing keys), to warning messages (such as missing exception handlers on dialogs), to detailed event logging of particular nodes visited and transitions evaluated during a particular call.

CONFIGURATION WIZARDS

One of the most challenging aspects of launching and maintaining a robust speech applications is managing the

range of speech configurations required in terms of grammars, language models, acoustic parameters, timeout parameters, rejection and confirmation thresholds, and so on. Typical recognition turns can be configured with as many as three or four dozen parameters. To manage the complexity inherent in this process, the framework includes a wizard tailored to configuring the parametric JSPs invoked for subdialogs, constructed by extending Netbeans [11], an open-source Java Integrated Development Environment (IDE).

Another challenging configuration situation arises in provisioning the server with the appropriate archive file and configuring both the servlet container to control the servlets and the server to point to the servlet container. This includes static content such as prompts, grammars and language models. Thus we provide a configuration wizard (also in Netbeans) for setting up the configuration necessary for the open source Apache web server and Tomcat servlet container.

XML SPECIFICATION

As in [7], we provide an XML specification format sufficient for generating an entire application, from configuration through to the Java code. The XML format is designed to eliminate redundant information, especially for configuration, and to be able to generate code that will run in the framework. Naming redundancy is especially problematic in web applications, where evaluation occurs in several environments and the location of compilation is not necessarily the location of deployment.

A validating DOM parser, also written in Java, carries out the transformation, generating web and servlet container configuration, controller servlet code, the application code, code for all of the dialogs and all of the actions, and creates the requisite directory structure for all of the prompts, grammars and language models. Along with this, it creates a build file in the open source Jakarta Ant format [12] that in turn builds the web archive of the complete application.

THE VOICE WEB

By admitting external subdialog calls, our framework allows for the possibility of distributed applications, much as on the web. For instance, flight information, mapping and directions or credit card subdialogs may be hosted remotely and re-used across any number of applications.

SUMMARY

We described a framework for developing VoiceXML applications on the server side that achieves portability through implementation as standard Java servlets and flexibility through its use of standard Java coding for underlying actions, which even allow for scoped control. We show how the components fit together and how we were able to build this using standard, off-the-shelf, open-source components.

REFERENCES

[1] W3C. 2001. Voice eXtensible Markup Language (VoiceXML) 2.0 Specification. Working Draft. http://www.w3.org/TR/2001/WD-voicexml20-20011023

[2] ECMA. 1999. Standard ECMA-262. ECMAScript Language Specification. 3rd Edition. http://www.ecma.ch/ecmal/STAND/ECMA-262.HTM

[3] W3C. 2001. Speech Recognition Grammar Specification for the W3C Speech Interface Framework. Working Draft. http://www.w3.org/TR/2001/WD-speech-grammar-20010820

[4] Tsai, A., A.N. Pargellis, C.-H. Lee, and J. P. Olive. 2001. Dialog session management using VoiceXML. Eurospeech 2001. Aalborg.

[5] Sun Microsystems. 2001. Java™ Servlet 2.3 Specification, Java Server Pages 1.3 Specification. http://www.jcp.org/aboutJava/communityprocess/final/jsr053/

[6] Pieraccini, R., S. Caskey, K. Dayanidhi, B. Carpenter, M. Phillips. 2001. ETUDE: A recursive dialog manager with embedded user interface patterns. *Proceedings of ASRU 2001*.

[7] Nyberg, E., T. Mitamura, P. Placeway and M. Duggan. 2002. DialogXML: Extending VoiceXML for dynamic dialog management. *Human Language Technologies 2002*. DARPA. San Diego.

[8] Rumbaugh, J., I. Jacobson, G. Booch. 1998. *The Unified Modeling Language Reference Manual*. Addison-Wesley.

[9] MITRE Corporation. 2002. Galaxy Communicator Documentation Version 3.3. http://communicator.sourceforge.net/sites/MITRE/distributions/GalaxyCommunicator/docs/manual/index.html

[10] GraphViz. Open Source graph visualization. http://www.research.att.com/sw/tools/graphviz/

[11] Netbeans. An Open-Source Java IDE. http://www.netbeans.org/

[12] Ant. An Open Source XML-Based Make. http://jakarta.apache.org/ant