MTALK - A Multimodal Browser for Mobile Services

Michael Johnston, Giuseppe Di Fabbrizio, Simon Urbanek

AT&T Labs Research
180 Park Avenue, Florham Park, NJ 07932 - USA
{johnston,pino,urbanek}@research.att.com

Abstract

The MTALK multimodal browser is a tool which enables rapid prototyping for research and development of mobile multimodal interfaces combining natural modalities such as speech, touch, and gesture. MTALK integrates a broad range of open standards for authoring graphical and spoken user interfaces and is supported by a cloud-based multimodal processing architecture. In this paper, we describe MTALK and illustrate its capabilities through examination of a series of sample applications.

Index Terms: multimodal, browser, speech, gesture

1. Introduction

Multimodal interfaces supporting natural modalities such as speech, touch, and gesture have been shown empirically to make it easier for users to interact effectively with applications and services, especially on mobile devices [1]. However, while numerous prototypes have been demonstrated over the last thirty years [2, 3, 4, 5] the barrier to entry for building multimodal systems both for research and deployment remains high. Typically these systems involve a range of different components, from speech recognition, gesture recognition, and language understanding to multimodal integration, presentation planning and dialog management. Building an effective multimodal interface remains a complex and specialized task often requiring a team with a range of diverse skill sets. In this paper, we describe MTALK, a multimodal browser and the cloud-based multimodal application framework that supports it. The MTALK browser runs on the iPhone and iPad and is freely available for research purposes upon request. The MTALK framework integrates a broad range of different open standards, including HTML, Cascading Style Sheets (CSS), JavaScript for GUI development, SRGS, and SSML for speech support. The framework also supports EMMA [6] for representation of multimodal input processing. Section 2 describes the capabilities of MTALK, Section 3 outlines the components of a MTALK multimodal application, and Section 4 provides case studies that illustrate the capabilities of MTALK. Section 5 describes related work.

2. MTALK multimodal browser architecture

Moving away from traditional telephony services, mobile interaction with automated services no longer requires establishing a phone call. In fact, modern devices offer direct access to rich multimedia capabilities with flexible programming environments and constant wireless network connectivity. This facilitates the development of mobile applications that directly orchestrate the user interface, the local device capabilities, and communicate with cloud-based services through the data network. In these terms, mobile services are gradually converging towards client-server based web applications, where the client runs the user interface on the device, and most, or part, of the service resides in the network cloud.

For mobile multimodal interaction, in addition to the graphical user interface, an application needs to manage gestures, touches, and real-time data streams from and to the audio subsystem. Additionally, advanced services may need to harvest the rich sensory data available on these devices, such as geographical location (GPS), direction (digital compass), motion (3-axis accelerometer), environmental conditions (proximity, ambient light sensors), etc. Although most of today’s high-end mobile phones (smartphones) provide application programming interfaces (API) to expose these features, the wide variety of software development toolkits (SDK) available on the market makes this task difficult across different models and brands.

Most mobile application development frameworks create a common layer of software on top of the specific vendors API that emphasize software component reuse and reduce application development cost, possibly preserving similar user experience even with diverse screen sizes and keyboard layouts. In general, three main approaches are found:

Native libraries - Applications rely on the native device SDK and the multimodal capabilities are wrapped into a reusable software library. This option is usually the best in terms of efficiency; memory footprint and direct access to the underlying device resources. However, native applications are tied up to a specific operating system (OS) and require special expertise for each OS or class of devices. The Android SDK, for instance, allows developers to access the device capabilities through common API and development tools, including desktop simulators and debuggers, but it only applies to Android-based phones. The AT&T Speech Enabler Services SDK is another example of speech native library that supports both user interface and speech service capabilities for Apple iOS, Android, and the RIM BlackBerry family of smartphones.

Widgets - A widget-based interface provides a mechanism to describe, package, install, and maintain software components accordingly to a W3C recommendation. Widgets usually implement simple tasks and act together in the hosting environment with other components to achieve more complex interactions. Portability across devices is handled in the hosting environment, which is usually implemented as native mobile application. The Opera Mini and the Opera Mobile projects,

http://developer.android.com/sdk
http://apple.com/ios
http://w3c.org/TR/widgets
http://opera.com/mobile

International Conference on Spoken Language Processing (2011)
Browser-based - This approach extends traditional web browser environments by exposing specific multimodal device capabilities through plug-ins. A browser-based application uses HTML markup language to describe the user interface content, CSS to define the presentation semantics, and the JavaScript to implement the application logic. In this case, device-dependent features are confined into a typically small JavaScript API, which is easy to replicate across browsers, while the presentation and the application logic remain almost identical on different devices. As a drawback, running a full-fledged browser may impact responsiveness and can degrade the user experience, but the constant improvements in CPU computation power reduce this issue over time.

The mTalk browser belongs to the latter category and has been implemented by extending the WebKit 9 open source web browser engine running on the Apple iOS for iPhone. Since WebKit lies at the core of mTalk, mTalk could readily be ported to other devices supporting WebKit by reimplementing the components for audio, video, and sensor capture. Figure 1 illustrates the general architecture.

The basic browser functionalities are extended by adding custom JavaScript bindings to WebKit. mTalk leverages the JavaScriptCore framework available in the native iOS to expose low-level device capabilities to the JavaScript environment. API calls from the JavaScript layer are captured and managed by the TalkViewController object, which is responsible for dispatching requests to the right object and to manage the visual state of the application to reflect the current actions.

Requests are initiated from the web application and follow the message-based command-response paradigm. Some commands (e.g., location services activation) can define a callback, which will be called asynchronously for the life of the service to deliver results (e.g., global positions for the location service). Others, such as setting operational parameters or speech recognition, produce only one response after the command has been processed. Requests can create complex chains of asynchronous objects that are entirely managed by the TalkViewController. For example, a speech recognition command instantiates a new Recorder object in a separate thread of execution, selects an audio encoder object according to the request such as SpeexEncoder in another separate thread, creates an audio sink destination (typically an HTTPAudioConnection) to the network endpoint where the automatic speech recognizer (ASR) is running, and starts streaming encoded audio samples in background until either the ASR returns a result or the transaction is interrupted by the user. Since all operations are performed in the background on separate threads, the web application remains responsive at all times. The end of the command is then delivered asynchronously to the JavaScript interpreter, which can react to the ASR event by executing a callback.

In addition to ASR, mTalk supports text-to-speech (TTS) synthesis, location services (position, compass), network availability (notification on change of network infrastructure, e.g., WiFi, cellular, VPN), multi-media playback, image and video capture. Modular and flexible design makes it easy to expose the underlying SDK functionality through new service objects interfaced to the browsers with JavaScript APIs. mTalk enhances the default look and feel of a web browser with additional elements that are programmable by the application. One example is the “ASR” button conveniently located in the mTalk task bar for an easy thumb operation outside of the web view (see Figure 2). This allows an application to carry out an ASR request by pressing the button without the need for explicit UI or code. It will also handle automatic end-pointing if needed.

A variety of protocols are available for real-time media transmission over IP networks. To deliver media over wireless networks, it is important that mobile clients be carefully designed to reduce latency by compressing media and reducing the number of retransmissions. mTalk uses HTTP [8] as transport protocol to deliver speech over the network. Latency evaluation experiments described in [9] show that HTTP 1.1, when used to deliver media streams with chunked transfer encoding mechanism, achieves real-time performance in the majority of working conditions. In essence, HTTP, although not strictly designed for real-time media streaming, turns out to provide a practical trade-off between simplicity and efficiency.

3. Multimodal application structure

To author a multimodal application using mTalk a number of different resources need to be built. The general structure of a mTalk application is presented in Figure 3.

The UI of a mTalk application is authored in HTML, CSS, and JavaScript hosted on an application server and downloaded to the mTalk browser when the application is accessed. A variety of different packages and libraries can be used to enhance

---

9 http://webkit.org

---

Figure 1: mTalk browser architecture

Figure 2: mTalk task bar

Figure 3: mTalk application structure
the visual UI. For example, either the IU10 package or the jQuery Mobile framework11 can be used to simulate the look and feel of a native iPhone application. Given the use of web technologies for the application, there is a great degree of flexibility in where the logic of an application can be captured and in the choice of programming language. The application logic can either be in JavaScript downloaded to the client, or coded in a range of different possible languages on the application server. Several of our sample applications use the former strategy and keep most of the functionality together in the client code.

The speech capabilities of the mTALK browser can be accessed through JavaScript APIs. These allow for audio to be captured, coded, and streamed over HTTP to a speech recognizer in the cloud. Both grammar-based (SRGS) and statistical N-gram models are supported and can be built by uploading grammars or corpora to the AT&T Speech Mashups portal [9]. The details of the recognition are made available to JavaScript through a callback and the client can act on the response. The mTALK TTS JavaScript API enables developers to specify a simple string or SSML markup to be rendered by a speech synthesis service in the cloud. Media synchronization between the speech output and the display can be achieved using bookmarks in SSML. When these are received by mTALK, the trigger JavaScript function calls synchronized with the speech output. Applications may also use multimodal integration capabilities accessed over HTTP. For example, the Restaurant Search sample application below uses finite state multimodal integration [10]. In this approach, a sequence of finite state compositions are used in order to combine speech and gesture inputs and assign a combined meaning. A representation of the gesture stream can be built in JavaScript on the client page (e.g., based on touches on objects in the UI) and shipped along with a speech string to the multimodal integration service. Once the client has received the results of speech or multimodal understanding, for many applications the next step is to access some external data source (e.g., an index of restaurant listings, flights, or movie titles). This can be achieved through asynchronous HTTP queries made to a proxy on the server hosting the web application. The speech recognition server can return results in EMMA XML [6], similarly many external data sources will return results in XML. Document object model (DOM) operations in JavaScript can be used to manipulate these results and formulate the next stage of processing or directly update the user interface displayed on the client by access to its DOM.

4. Sample multimodal applications in mTalk

The sample applications, ‘Movie Search’, ‘Restaurant Search’, and ‘Newsreader’, that appear in the initial screen in mTALK, each illustrate different multimodal interaction paradigms.

**Movie Search** - This application allows users to search for movies by voice and receive results and details graphically. It illustrates the paradigm of using a graphical interface for multimodal error correction. The user presses a Speak button and utters a query e.g., *Harry Potter*, multiple possible competing recognition hypotheses are displayed (Figure 4). In this case, the request for recognition to the mTALK Speech API, in addition to specifying the grammar to be used, also specifies that N-best recognition results are needed (e.g., ‘decoder.nbest=10’) and that the results should be returned in EMMA (‘format=emma’). The results that come back to the client JavaScript are in an EMMA XML document consisting of sequence of <emma:interpretation> elements contained within an <emma:one-of>. The client code renders this as a spinnable list of HTML elements for the user to select among (Figure 4 (Left)). When the user clicks on one of the items in the list it results in a query being sent to a movie search backend and details on the chosen movie are displayed (Figure 4 (Right)).

**Restaurant Search** - Like Movie Search, the user hits a Speak button in order to initiate capture of audio in the mTALK browser and streaming of that content to the recognizer, in this case using a model that supports recognition of restaurant names (e.g., *Babbo*), categories (e.g., *Italian restaurants*), and locations (e.g., *lower east side*). The sample application supports search for restaurants in New York City. The application also uses natural language understanding for queries that specify multiple constraints, and utilizes the JavaScript interface to GPS provided by mTALK to support search based on the current device location. The UI includes a map display, illustrating how mTALK enables combination of speech capabilities with other web services such as mapping APIs like Google Maps and Yahoo! Maps.

![Figure 4: Movie Search N-best Selection](image-url)

![Figure 5: Local search interface map/refline](image-url)
specifies their location of interest by gesturing on the map, e.g., "pizza restaurants near here" (touch on map). When the speech result is received by the client, it is sent by HTTP along with a list of symbols representing the gesture to a multimodal understanding component that uses a multimodal grammar in order to integrate speech and gesture and determine their combined interpretation. Here, the combined semantics is returned in an EMMA document to the JavaScript running in the client, a query is then issued to the search engine, and pizza places near the point are shown and map zooms in on that area. Simple cases of multimodal integration, such as this where there is a single gesture could also be handled directly in JavaScript code. More complex multimodal inputs with multiple gestures, or cases involving handling of ambiguity in the gesture and speech streams can be handled with more powerful methods accessed over HTTP such as finite-state multimodal integration.

**Multimodal Refinement:** This application demonstrates a multimodal interface for error correction and iterative refinement of queries. The 'Select' screen in the UI allows users to use text or voice input to correct or refine individual components of their query (Figure 5 (Right)). If one of fields, e.g., the category or location, is wrong or otherwise needs to be changed, the user can correct it by tapping on the button next to the field and speaking the new value. More restrictive field specific models can be used for recognition. Text input can also be used, and for more limited sets of values all of the capabilities of HTML such as radio buttons, drop down menus, spinnable lists of items etc can be used for quick correction and refinement.

**Newsreader** - The third application is a news reader that reads RSS newsfeeds. It demonstrates the use of TTS and the media synchronization capabilities of MTalking multimodal output generation. The string sent to the TTS engine can be in either plain text or SSML. In SSML, the `<mark>` element can be used to indicate specific points in the text. As the text is rendered in addition of the audio stream, the TTS engine receives a stream of mark events indicating the progress of TTS. These can be captured in JavaScript and used to coordinate graphical actions with speech output. In Newsreader, as each phrase is spoken it is highlighted in the HTML page.

Figure 6: Newsreader user interface

### 5. Related work

The first mobile multimodal browsers were built in connection with early efforts to standardize the addition of speech to HTML markup. As part of the SALT (Speech Application Language Tags) initiative [11] there was a mobile version of Pocket Internet Explorer with voice support. Several mobile browsers including Netfront and Opera had early implementations of the X-w-v proposal for combination of XHTML and VXML specification [12]. Neither of these provided the cloud-based framework for multimodal application authoring found in Talk though. More recently, the WAMI toolkit was released by MIT [13]. Like MTalking, it combines an HTML browser with access to cloud-based speech resources through a JavaScript API. One important difference is the availability of advanced capabilities such as multimodal output synchronization in the MTalking framework, and the adoption of open standards such as EMMA, SSML, and SRGS.

### 6. Conclusion

The MTalking multimodal browser along with the cloud services that support it have the potential to dramatically reduce the barrier of entry for prototyping and experimentation with mobile multimodal interfaces. MTalking integrates a broad range of open standards, including HTML, CSS, Javascript, SRGS, SSML, and EMMA, and supports sophisticated multimodal interaction paradigms such as multimodal integration, multimodal error correction, and synchronized multimodal output generation combining graphical actions with synthetic speech.

### 7. References


