

Spoke: A Framework for Building Speech-Enabled Websites

by

Patricia Saylor

S.B., Massachusetts Institute of Technology (2014)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2015

© Massachusetts Institute of Technology 2015. All rights reserved.

Signature of Author.....

Department of Electrical Engineering and Computer Science

May 22, 2015

Certified by.....

James Glass

Senior Research Scientist

Thesis Supervisor

Accepted by.....

Professor Albert R. Meyer

Chairman, Masters of Engineering Thesis Committee

Spoke: A Framework for Building Speech-Enabled Websites

by

Patricia Saylor

Submitted to the Department of Electrical Engineering and Computer Science

on May 22nd in partial fulfillment of the requirements for the degree of

Master of Engineering

Abstract

In this thesis I describe the design and implementation of Spoke, a JavaScript framework for building interactive speech-enabled web applications. This project was motivated by the need for a consolidated framework for integrating custom speech technologies into website backends to demonstrate their power. Spoke achieves this by providing a Node.js server-side library with a set of modules that interface with a handful of custom speech technologies that can perform speech recognition, forced alignment, and mispronunciation detection. In addition, Spoke provides a client-side framework enabling some audio processing in the browser and streaming of the user's audio to a server for recording and backend processing with the aforementioned integrated speech technologies. Spoke's client-side and server-side modules can be used in conjunction to create interactive websites for speech recognition, audio collection, and second language learning, as demonstrated in three sample applications.

Though Spoke was designed with the needs of the MIT Spoken Language Systems group in mind, it could easily be adopted by other researchers and developers hoping to incorporate their own speech technologies into functional websites.

Thesis Supervisor: James Glass

Title: Senior Research Scientist

Acknowledgements

From my first visit to MIT, before even applying for admission, I felt at home. Now after 5 years at this wonderful institution, immersed in a fun, ambitious, and intelligent community, I am sad to leave, but ready for the next chapter of my life. I would like to thank my thesis advisor Jim Glass and the MIT SLS group for providing support and encouragement over the course of this thesis. An additional thanks to Quanta Computing for their continued interest and support in the lab.

Over the years there were many others who helped make MIT my home. Thank you to my parents, Dana Saylor and Julie Stuart, my sister, KayLynn Foster, and my aunt and uncle, Fred and Chris Lalonde, for their love and support, which helped me transition to life in Cambridge and at MIT.

Thank you to Jiayi Lin and Vedha Sayyaparaju, my closest friends during undergrad, to Jonathan Lui and Marcel Polanco, my frequent classmates and periodic project partners, and to Divya Bajekal and Sunanda Sharma, my friends and cooking coconspirators during grad school. A special thank you goes to my best friend, Samvaran Sharma, for his unwavering support, understanding, and faith in my abilities.

Contents

1	Introduction	18
1.1	Motivation	19
1.2	Purpose.....	20
1.3	Outline.....	21
2	Background	22
2.1	Related Work	22
2.1.1	WAMI.....	23
2.1.2	CMU PocketSphinx.....	23
2.1.3	Recorder.js.....	24
2.1.4	The Web Speech APIs.....	24
2.2	Kaldi.....	25
2.3	Challenges.....	25
2.4	The Modern Web	26
2.4.1	JavaScript.....	27
2.4.1.1	Node.js.....	27
2.4.1.2	Asynchronous JavaScript: Callbacks and Promises.....	28
2.4.1.3	Managing Module Dependencies: CommonJS and RequireJS.....	30
2.4.2	Express.js, Ractive, Socket.io, and WebSockets.....	30
2.4.3	Web Audio and Web Speech APIs.....	32
3	System Components and Design	34
3.1	System Overview	35
3.2	Spoke Client-Side Framework	36
3.2.1	CrossBrowserAudio.....	37

3.2.2	Microphone Volume Meter.....	39
3.2.3	Recorder.....	42
3.2.4	Player.....	45
3.2.5	Recognizer and Synthesizer.....	47
3.2.6	CSS Library.....	49
3.2.7	Package Distribution and Usage.....	49
3.3	Spoke Server-Side Library	50
3.3.1	Audio Utilities.....	51
3.3.1.1	Recorder.....	52
3.3.1.2	Player.....	52
3.3.2	Integrated Speech Technologies.....	53
3.3.2.1	Recognizer.....	54
3.3.2.2	Forced Alignment.....	55
3.3.2.3	Mispronunciation Detection.....	56
3.4	SoX Audio Processing Module.....	58
4	Sample Applications.....	63
4.1	Online Speech-Enabled Nutrition Log	64
4.1.1	Nut.....	64
4.1.2	The Nutrition System.....	67
4.2	Amazon Mechanical Turk Audio Collection.....	68
4.3	Orcas Island: Mispronunciation Detection.....	71
5	Future Work.....	79
5.1	Streaming Speech Recognition	79
5.2	Reducing Bandwidth Usage and Dropped Audio	80
6	Conclusion.....	82

List of Figures

- Figure 2-1: Promise States:** This diagram depicts the three possible states of a Promise object – pending (where the operation has yet to be completed), fulfilled (where the operation has completed successfully), and rejected (where the operation has completed unsuccessfully). An asynchronous function can return this type of Promise object instead of taking a callback parameter, enabling easier handoffs of information with other asynchronous functions.....29
- Figure 2-2: AudioNodes (from Mozilla Developer Network):** A diagram depicting the layout of AudioNodes within an AudioContext as part of the Web Audio APIs [31]....32
- Figure 3-1: Spoke Module Overview:** Spoke is split into two main components – a client-side framework and a server-side library. The client-side framework includes feature and helper modules to enable the creation of interactive UI elements (e.g. play and record buttons, volume meters, front-end connections for recognition and recording). The server-side library includes modules to help with things such as recognition, alignment, recording, and more. These server-side modules can be used as part of a web server, or in batch processing scripts.....35
- Figure 3-2: Resolving Vendor-Prefixed APIs with Modernizr:** CrossBrowserAudio uses Modernizr to find browser APIs under their vendor-prefixed names and make them available under a non-prefixed name to simplify cross-browser development.....38
- Figure 3-3: Sample Microphone Volume Meters:** Top: The VolumeMeter enables volume visualizations on a height-adjustable HTML element. Typically a microphone icon is

used. Bottom: This frequency spectrum visualization illustrates how the VolumeMeter extracts the frequency bins corresponding to the dominant human speech range (from 300 to 3300 Hz, i.e. telephone bandwidth, in dark blue) when computing the volume level. Frequencies above 16 kHz rarely appear in the FFT.....39

Figure 3-4: Microphone Volume Meter Design: The VolumeMeter creates an audio processing graph of AudioNodes to compute the volume level. The AnalyserNode performs an FFT on the user’s audio stream, and the ScriptProcessorNode extracts the frequency data corresponding to the dominant frequency range of human speech, 300 to 3300 Hz. The final volume level is computed by averaging over this range and then normalizing.....40

Figure 3-5: Microphone Volume Meter Usage: VolumeMeter instances can be initialized with an HTML element and an options object to configure certain properties of the AnalyserNode’s FFT, the ScriptProcessorNode, and the meter element.....41

Figure 3-6: Client-Side Recorder Usage: The Recorder is initialized with an HTML element that will toggle recording and an optional options object where the developer can specify the buffer length for the Recorder’s ScriptProcessorNode and the metadata to be sent to the server along with the stream of audio.....42

Figure 3-7: Client-Side Recorder Design: The Recorder creates an audio processing graph consisting of a MediaStreamAudioSourceNode connected to a ScriptProcessorNode. The audio source for the MediaStreamAudioSourceNode is supplied by promiseUserMedia. The ScriptProcessorNode operates on a buffer of audio data, converting it to the right encoding and the right type of buffer before writing it to the socket stream.....44

Figure 3-8: Client-Side Player Usage: The Player is initialized with an audio stream and will automatically play the audio as soon as it is ready. An options object may be passed in during initialization to prevent this default behavior.....45

Figure 3-9: Client-Side Player Design: The Player routes audio from an AudioBufferSourceNode to an AudioDestinationNode that plays the audio over the user's speakers. The Player accumulates the audio data from the stream and consolidates it into one ArrayBuffer. Using the decodeAudioData API, it transforms the ArrayBuffer into an AudioBuffer to provide to the source node.....46

Figure 3-10: Recognizer Configuration: Spoke's Recognizer interface can be used to interact with any recognizer that implements a basic command line interface that accepts a wav filename and outputs the recognition results on stdout.....54

Figure 3-11: Example Output from Forced Alignment: Running forced alignment on a wav and txt file results in a timing file with a simple column format as shown above. This output contains the timing data for each phoneme of each word in the txt file, but word-level timing can be extracted by taking the start sample from the first phoneme and the end sample from its last phoneme.....55

Figure 3-12: Forced Alignment Result Chaining: The output of forcedAlignmentAsync is a Promise that will be fulfilled with the name of the timing file. This Promise can be awaited with 'then' to perform some action on its completion, namely to parse the timing file with getAlignmentResultsAsync, which also returns a Promise that can be awaited and chained in a similar manner.....56

Figure 3-13: Mispronunciation Processing Chain: Using Promises, the processing steps of mispronunciation detection can be chained for sequential asynchronous execution. First we execute forced alignment, then parse the timing file, then preprocess the utterance, and finally perform mispronunciation detection. The last step involves a streaming pipeline between the output of mispronunciation detection and a parser that turns results into JavaScript objects.....58

Figure 3-14: Example Complex SoX Command: Here we use the special filename "|program [options] ..." to create SoX subcommands that trim the start or end off of a single file. Inside this subcommand, the -p flag indicates that the SoX command should be used as in input pipe to another SoX command. The overall SoX command has 3 inputs that it concatenates into a single output file: utterance_0.wav trimmed to start at 4.03 seconds, utterance_1.wav untrimmed, and utterance_2.wav trimmed to end at 2.54 seconds.....59

Figure 3-15: Initial Use of SoX Transcoding in JavaScript: Our early usage of SoX just created a String command to perform transcoding of a raw audio file to a wav audio file and executed it with Node's child_process module.....60

Figure 3-16: Example SoxCommand for Streaming Transcoding: In this example a 44.1 kHz raw audio stream of signed 16-bit integer PCM data is converted on the fly to a 16 kHz wav audio stream.....61

Figure 3-17: Example SoxCommand using SoxCommands as Inputs to Trim and Concatenate Audio Files: This example leverages SoxCommands as inputs to other SoxCommands, allowing for the results of one to be piped as input into the

other. The main SoxCommand concatenates three audio files, while the two sub commands handle trimming the first and last files respectively.....61

Figure 4-1: Nut Application: The small Nut webpage displays a volume meter on a microphone icon that doubles as the record button. When the icon is clicked, it changes from blue to red to indicate the recording status. At the end of recording, the audio is run through a recognizer on the server, and the client is notified of the recognition result, which it places below the icon.....65

Figure 4-2: Nut VolumeMeter and Recorder Instances: Nut makes an instance of the VolumeMeter and of the Recorder, both of which are attached to the same HTML element (the microphone icon). We can listen for certain recorder and socket events to appropriately update the UI.....66

Figure 4-3: Nut Server Recorder and Recognizer Setup: The Nut server creates a new Recorder and Recognizer instance when a new socket connection is established. Then it listens for an 'audioStream' event on the socket and handles it by passing the stream to the Recorder for transcoding to a wav file. When the Recorder finishes, the wav file is passed to the Recognizer for recognition.....67

Figure 4-4: The Nutrition System: The microphone icon acts as volume meter and triggers both recognition with the Web Speech APIs and recording to the Nut server when clicked. The nutrition system determines which parts of the utterance correspond to food items, quantities, descriptions, and brands, and tries to provide nutritional information about the identified food.....68

Figure 4-5: Audio Collection Task on Amazon Mechanical Turk: The Turkers are presented with a set of 10 sentences or sentence fragments to record (5 shown in this screenshot). While recording one utterance, the record button turns into a stop button and the other buttons on the page are disabled. After recording, the UI is updated with feedback about the quality and success of the recording. Sentences marked with “Redo” must be re-recorded before submission.....70

Figure 4-6: Orcas Island Homepage: Orcas Island features a selection of short stories from Aesop’s Fables. The homepage displays our current selection of stories and links to each one.....72

Figure 4-7: Reading a Short Story on Orcas Island: The story is broken into small, readable fragments of one sentence or less. When the user records a sentence, the utterance is processed on the server and after a short delay is highlighted in light gray to indicate successful processing. When the user hovers over a fragment or is reading a fragment, its control buttons appear and the text is bolded to make it easier to read.....73

Figure 4-8: Orcas Island Mispronunciation Processing Chain: The server uses Promise chaining to create a sequential asynchronous processing pipeline. This pipeline first waits on the wav and txt files to be successfully saved, then performs forced alignment, gets the alignment results and saves them, notifies the client of success, and performs mispronunciation preprocessing.....74

Figure 4-9: Orcas Island Mispronunciation Analysis UI: The results of mispronunciation detection are displayed in a table with the more significant

mispronunciation patterns near the top. The client uses the Spoke Synthesizer to enable a comparison between the user's pronunciation and the correct pronunciation of the word. The magnifying glass will highlight all instances of that word in the read fragments.....75

Figure 4-10: Orcas Island Recording and Processing Diagram: This diagram illustrates how the spoke-client Recorder streams audio in buffers to the server, where the server-side Recorder converts the audio to a wav file and then passes it into the processing pipeline outlined in Figure 4-8.....76

Figure 4-11: Final Mispronunciation Detection Step With A Stream Pipeline: Instead of using Promises, this method handles the final mispronunciation detection step by using Streams. The stdout output from the mispronunciation detection system is transformed into a Stream of objects representing mispronounced words.....77

Chapter 1

Introduction

Speech recognition tools have a wide range of applications, from informational discourse for checking the weather, to multimodal interfaces for playing a language learning game. Many such applications were developed by the MIT Spoken Language Systems group (SLS) [1,2] as part of a larger movement towards enabling a broad range of computer interaction through natural spoken language. To this end, the group regularly builds both new speech recognition tools and new applications demonstrating the power of these tools.

1.1 Motivation

For example, Ann Lee, a Ph.D. student in the group, is developing a mispronunciation detection system that discovers patterns of mispronunciation made by a speaker in a set of multiple utterances [3]. Her backend work needs a frontend application to highlight its true potential in the domain of language learning. An application for this system will crucially need to be able to record audio from a live speaker for immediate processing on the backend, but audio recording on the web, our preferred application domain, is nontrivial and unstandardized.

Within SLS there is high demand for live audio recording on the web—particularly to gather custom audio training data by creating an audio collection task on Amazon Mechanical Turk—but currently there is no in-home support for this feature after the previous outdated framework was phased out [4]. There are also many researchers seeking to build small demo websites using their own tools or those built by others in the lab, but in general there is no standard way of incorporating one’s speech recognition tools into a website. Setting up this interaction independently for each project in the lab is inefficient and makes it harder to share and reuse tools on other people’s sites. Thus we need a consolidated, reusable, and modern solution to using and sharing internal speech recognition tools for sites throughout the lab.

The group also seeks to expand into new areas for fluid speech interactions. With the near ubiquity of mobile devices, we strive to bring our speech recognition demos to mobile browsers and native apps on phones and tablets. With increasing demand for responsive, real-time applications, we strive to create our own continuous speech recognition system that outputs recognition hypotheses even as audio is still being streamed and processed. Both of these goals will benefit from a consolidated framework, while adding their own challenges to its development.

1.2 Purpose

The purpose of this thesis is to develop Spoke, a framework for the Spoken Language Systems group to build spoken language enabled websites. This framework leverages modern web technologies such as the Web Audio APIs, WebSockets, and Node.js to provide a standardized and streamlined way to build website demos in the lab featuring our own spoken language research. Spoke is implemented entirely in Javascript.

Spoke has two overarching components, one for the client (the browser) and one for the server, and it provides a decoupled server library that can be used independently of a website. The client-side framework provides a set of tools for building front-end speech recognition websites:

- recording audio to the server for processing
- playing audio from the server
- visualizing audio information in a volume meter
- speech recognition (e.g. using Google's SpeechRecognition, available in the Chrome browser, or custom backend recognizers)
- speech synthesis (e.g. using Google's synthesizer, or custom synthesizers)

Meanwhile the server-side component of Spoke provides a library of speech recognition technologies built in the lab and a utility-belt of audio tools. The library's utilities enable recording raw or wav audio, transcoding audio files or live audio streams, and playing back streaming audio from a file. The library also provides standardized access to and usage of technologies built in the lab, such as

- domain-specific speech recognizers
- forced alignment of an audio sample to expected text
- mispronunciation detection on a set of utterances from the same speaker

This is a subset of technologies built in the lab, and it is easy to add new tools to the Spoke library so that everyone in the lab can use them. One technology we expect to add to Spoke within the next year is continuous streaming speech recognition, which will benefit from the current utilities for streaming audio recording and streaming audio transcoding. This library is decoupled from client interactions, allowing the developer to use the speech recognition and audio tools for batch processing and scripting in Javascript on the backend.

Creating Spoke with the goals of SLS in mind meant confronting a few big challenges. The ideal of incorporating the broad range of speech technologies built in the lab, and those soon to be built, requires a framework with great modularity and extensibility. Making our demos work on mobile devices, and more generally in limited bandwidth conditions, encourages reducing data usage by using a small client side framework and by downsampling the audio before streaming, but ultimately robust handling of communication errors on both sides is still necessary. Later in this thesis we will explore these challenges and others further.

1.3 Outline

In Chapter 2 I discuss some related work in the area of building speech-enabled websites and provide some background about modern technologies for building web pages and web servers with JavaScript. Chapter 3 gives a high-level overview of Spoke before detailing the design and implementation of its client-side and server-side components. Chapter 4 describes three sample applications built with Spoke, highlighting our ability to enable interactivity with custom backend speech technologies. Chapter 5 presents areas for future work and specific suggestions for their undertaking. Chapter 6 concludes this thesis.

Chapter 2

Background

2.1 Related Work

Enabling speech recognition on the web is far from a new idea, but it is also far from a solved problem. The variety of available solutions do not fully meet our needs, either because they are outdated (WAMI) or they clash with using our own spoken language systems (CMU Sphinx). After all, we want to demonstrate the power of the tools we ourselves researched and developed. At times, other solutions could be used alongside our technologies, such as the Web Speech APIs, but at other times it makes more sense to build our own solution to have full control over it.

2.1.1 WAMI

The WAMI (Web-Accessible Multimodal Interfaces) toolkit was developed in 2008 with the goal of enabling multimodal interfaces, particularly speech interfaces, on the web [4]. WAMI importantly provided speech recording and speech recognition as a service to simplify and thus encourage development of speech-enabled websites. An important component of WAMI was the AudioController, which was responsible for recording the user’s audio from the browser and transmitting it to a remote recognizer, and also playing back audio. Although these two features are in high demand in the lab, the technology underpinning the AudioController is outdated and we no longer use WAMI.

Initially the AudioController was built as an embedded Java applet, but it was later rebuilt with the Flash browser plugin [5]. Aside from the poor performance and poor user experience of Java applets, they have lost support from most major browsers, and Flash, though still supported widely, is quickly becoming outdated. Flash has limited access to the browser’s technologies, and also enforces the use of an ugly, un-customizable security panel for granting audio permissions. The drawbacks and workarounds implemented in order to use Flash are now unnecessary given the state of the modern browser APIs for accessing a user’s audio, and the ability to stream binary audio data over WebSockets from the browser to a backend server.

2.1.2 CMU PocketSphinx

The CMU Sphinx speech recognition toolkit provides a few different packages for different tasks and speech applications, including a distribution called PocketSphinx built in C for use on computers and mobile devices [6]. In 2013, they ported PocketSphinx to JavaScript, empowering developers with a way to run a real-time recognizer for their websites

completely on the client-side [7]. This project is a great boon to web developers, but to other speech technology researchers, it is less useful. It necessarily utilizes CMU's own speech recognition technologies, making it irreconcilable with our goal of building websites to demonstrate our own technologies.

2.1.3 Recorder.js

A small, independently built project, Recorder.js provides a library for recording audio using the modern Web Audio APIs (discussed later) and exporting that recording as a wav file [8]. As it is available on GitHub under the MIT License, it is frequently referenced as a demonstration of how to use the Web Audio APIs for audio recording on the client, though the library stops short of sending that audio to a server. Given this shortcoming, this library barely scratches the surface of the capabilities we need for our own websites. This makes the library an instructive example, but ultimately we built audio recording into Spoke, giving us full control over that process.

2.1.4 The Web Speech APIs

The Web Speech API was introduced in late 2012 with the ambition of integrating speech technologies into the HTML5 standard and ultimately providing web developers with tools for speech recognition and speech synthesis directly in the browser [9]. So far, only the Chrome browser has implemented this API (full support starting in version 33), though support for Firefox seems to be underway. These APIs empower web developers with a black box recognizer or synthesizer, with options for finer control over their capabilities if desired. For example, the SpeechRecognition API can recognize a single utterance after it completes, or you can configure the recognizer to perform continuous recognition and output intermediate recognition results as the user speaks. Though the Web Speech APIs are not sufficient for all of our needs, they have

proven to be useful in the lab and can easily be used alongside our own audio tools for the browser, so they have been integrated into the Spoke client-side framework.

2.2 Kaldi

Kaldi is a free speech recognition toolkit written in C++ available under the Apache License [10]. Many, if not most, of the core speech technologies being developed in SLS use the Kaldi toolkit for its state-of-the-art capabilities and extensible design. The end result of developing speech technologies with Kaldi is usually a bash script that runs a bunch of other scripts with the right inputs, and this makes interfacing with Kaldi-built systems very flexible. The Spoke server-side library was designed with this executable script paradigm in mind, and with future extensions to Kaldi in sight. Though Kaldi provides support for “online decoding” in some fashion [11, 12], we have not yet set up our own instance of their online decoders operating on audio streams. However, within the next year the lab hopes to build a continuous speech recognizer on top of Kaldi and this tool can be incorporated into Spoke in much the same way as existing Kaldi tools.

2.3 Challenges

Creating a framework for integrating speech technologies into web applications meant confronting a few big challenges. Ideally we would want all speech technologies built in the lab to be easily incorporated into Spoke, allowing anyone else in the lab to discover and use that technology, either for offline processing or for a website backend. To this end, it is important to pick a simple, flexible interface between the Spoke library and non-JavaScript tools developed in the lab. Many of these tools are built using Kaldi, which lends itself to

command line usage. This prompted a command line script interface with Spoke, which should be generic enough to extend to tools built in other languages and on other frameworks, so long as the tool can be used from the command line.

With mobile devices and slow Internet connections comes the need for reducing data usage to likewise reduce latency. When a webpage is first opened, it usually needs to fetch some extra JavaScript and CSS files, and how these files are provided can have a significant effect on the load time of the page. External libraries should be loaded from a CDN, a Content Delivery Network that distributes static content, meaning you only have to load each file once and then it is cached in the browser. The website's custom files, on the other hand, have to be loaded every time, so to reduce latency we must minify the JavaScript and CSS files and even then we should try to load them asynchronously if possible. After the webpage is ready, we expect audio streams to consume the most bandwidth, so this is the next area for improvement. We can reduce the size of the audio stream by downsampling and/or compressing the audio before transmission. On the server-side, we use the SoX audio processing command line tool to accomplish this, but on the client-side we will later have to implement our own downsampling since it is not built into the Web Audio API.

2.4 The Modern Web

In the past few years the web community has seen dramatic improvements in the functionality of modern web browsers, the power of JavaScript, and the interactivity of communication between a client and server through WebSockets.

2.4.1 JavaScript

JavaScript is a cross-platform, object-oriented scripting language that is incredibly flexible but sometimes tricky. Its flexibility comes from being a dynamically typed language that supports both the functional paradigm and object-oriented programming without classes. The core language includes an impractically small standard library of objects, from Array to Date to String. Most of its power, then, comes from its host environment and how properties and methods can be added dynamically to objects [13]. Inside a host environment, JavaScript can be connected to the objects of its environment to provide programmatic access and control over them, while the core libraries themselves are frequently extended to provide more utility.

On the client side, this translates to an extended language with new objects for some complex interactions with the browser and servers. Far beyond the basic objects for interacting with a browser and its DOM (Document Object Model), browsers now extend JavaScript to provide powerful APIs ranging from server communication to audio processing. For example, an XMLHttpRequest allows you to send asynchronous HTTP requests to a server, while WebSockets enable a persistent TCP connection to the server for bidirectional communication. Moreover, the Web Audio API [14] provides a powerful interface for controlling audio sources, effects, and destinations, enabling web developers to record audio from the browser with no hacks or plugins. With the rise in server-side JavaScript implementations, and the increasing power of JavaScript in the browser, we can now use the same language in both places to great effect.

2.4.1.1 Node.js

Server-side JavaScript has been around since 1994, but has recently seen a huge resurgence with frameworks like Node.js. Node is a platform for building and running fast, scalable JavaScript applications [15]. The platform is built on Chrome's JavaScript runtime, allowing you to run JavaScript outside of a browser environment, and it extends the core language with full-fledged modules for interacting with the server's file system and for creating and controlling child

processes, just to name a few. With Node, you can easily develop command-line tools or full-featured dynamic HTTP and HTTPS servers, and I hope that members of SLS soon use it for both.

The two central themes of Node are event-driven, non-blocking I/O and Streams [16]. JavaScript itself does not have a built-in way to do I/O, but web developers were already familiar with asynchronous callback-based I/O in the browser (AJAX is a good example of this). So Node built out I/O support following this same single-threaded asynchronous, non-blocking model using a library called libuv [17]. This makes Node very well suited for data-intensive real-time applications where the limiting factor is I/O, but not appropriate for CPU-intensive operations. So performing speech recognition in Node.js is out of the question, but we can create a web server in Node capable of handling thousands of concurrent connections, and then spin up other processes for performing speech recognition upon request, which is what Spoke does.

The Stream module provided by Node.js is also central to their evented asynchronous model. Stream instances are very reminiscent of Unix pipes, and they can be readable, writable, or duplex (both readable and writable). The ReadableStream interface allows you to process a large stream of data a chunk at a time by subscribing to the “data” event, where you can add a function to read, process, and even emit some transformation of that data. Additionally, you can pipe readable streams into writable streams with “pipe”, allowing you to chain together a processing pipeline, and you can pipe one readable stream into two separate writable streams. This proves useful for dealing with readable streams of audio data, allowing us to handle incoming raw audio in two different ways at once: saving the raw audio directly, and performing streaming downsampling of the audio using SoX [18].

2.4.1.2 Asynchronous JavaScript: Callbacks and Promises

Asynchronous callbacks in JavaScript are great for simple operations, but they can get messy very quickly when callbacks are nested inside of other callbacks. In Node, this type of nesting is especially common since all I/O is asynchronous. Luckily the continuation passing

style of callbacks can be subverted by using Promises, a new pattern for representing the results of an asynchronous operation [19]. Instead of an asynchronous function taking a callback parameter, it can return a Promise object, which can be in one of three states:

- pending—the initial state of a promise before the operation has completed
- fulfilled—the state of a promise indicating a successful operation
- rejected—the state of a promise indicating a failed operation

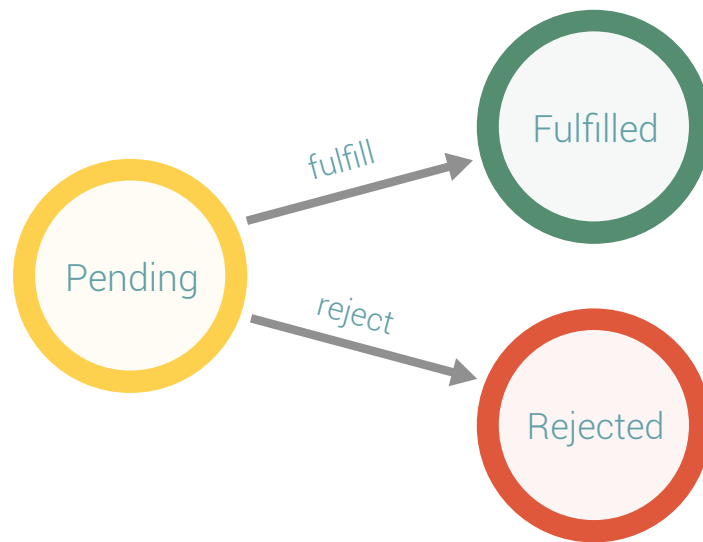


Figure 2-1: Promise States: *This diagram depicts the three possible states of a Promise object – pending (where the operation has yet to be completed), fulfilled (where the operation has completed successfully), and rejected (where the operation has completed unsuccessfully). An asynchronous function can return this type of Promise object instead of taking a callback parameter, enabling easier handoffs of information with other asynchronous functions.*

Promises can easily be composed using their “then” and “catch” methods to take action on the fulfillment or rejection of a promise, respectively. There are many JavaScript implementations of the Promise API for both the client and server, and the most comprehensive ones also build out support for “promisifying” callback-based functions, including those for I/O in Node.js [19, 20].

2.4.1.3 Managing Module Dependencies: CommonJS and RequireJS

Modularity is critical in software design, but it is not built into the core JavaScript language. Currently there are two main flavors of modularization for JavaScript: CommonJS and RequireJS. CommonJS is the standard for Node.js development, but it is not well suited for the browser environment [22]. In the CommonJS style, every file is given a `module.exports` object to fill with that module's contents; the exports object can be a function, a value, or a plain JavaScript object containing a plethora of other functions, values, and objects. When you require another module with `mod = require('otherModule')` you get back its `module.exports` object. This model is simple and easy to use on the server-side, and though it can be used on the client-side, there is a better option.

Though browsers themselves provide no way to manage JavaScript module dependencies on the client side, RequireJS is optimized for exactly this [23]. In the earlier days of web development, script dependencies were managed by just loading scripts in a particular order, which is a fragile sort of modularity. RequireJS allows you to explicitly list module dependencies in a file with the “define” function, and then at runtime it asynchronously loads those dependencies for you. Besides the asynchronous module loading, lower latency can be achieved by using the `r.js` optimizer for packaging your main JavaScript file and all its dependencies into one minified file [24, 25]. Spoke uses CommonJS for the server side code and RequireJS for the client side code.

2.4.2 Express.js, Ractive, Socket.io, and WebSockets

Part of the success of Node.js and its revival of server-side JavaScript can be attributed to its large and active ecosystem, which allows anyone to develop and publish a Node.js

module through NPM, Node's package manager [26]. Express.js and Socket.io are two high-quality projects born out of this open ecosystem that are geared towards building great websites.

Express.js is a Node.js web application framework with support for routing, middleware, RESTful APIs, and template engines [27]. It is a core component of the now popular MEAN stack for web development, which includes MongoDB, Express.js, Angular.js and Node.js, but you can use it just as effectively with other database and templating solutions. Express supports a vast array of templating solutions, and it can be hard to find the right one to use, balancing the features you need with ease of use and current maintenance of the project. For my web applications, I chose to use Ractive, a Mustache-based templating engine featuring a simple but powerful templating syntax, two-way data binding, and template partials [28].

Socket.io is a JavaScript library for building real-time web applications with event-based bidirectional communication between the client and server [29]. Traditionally the client initiates communication with the server through a single request, and receives a single response; but the bidirectional model means either side can initiate communication, so the server can push data updates to the client when they are available instead of the client asking for updates periodically. WebSockets [30] provide the underpinnings for Socket.io, enabling a persistent TCP connection to the server for the bidirectional communication, but Socket.io can also fallback to AJAX longpolling if a WebSocket connection cannot be established. Socket.io provides the best abstraction around the communication layer, and with support for binary streaming added in version 1.0 in 2014, it is a great solution for audio recording. The client-side and server-side components of the Spoke framework sit on top of the corresponding client-side and server-side components of Socket.io.

2.4.3 Web Audio and Web Speech APIs

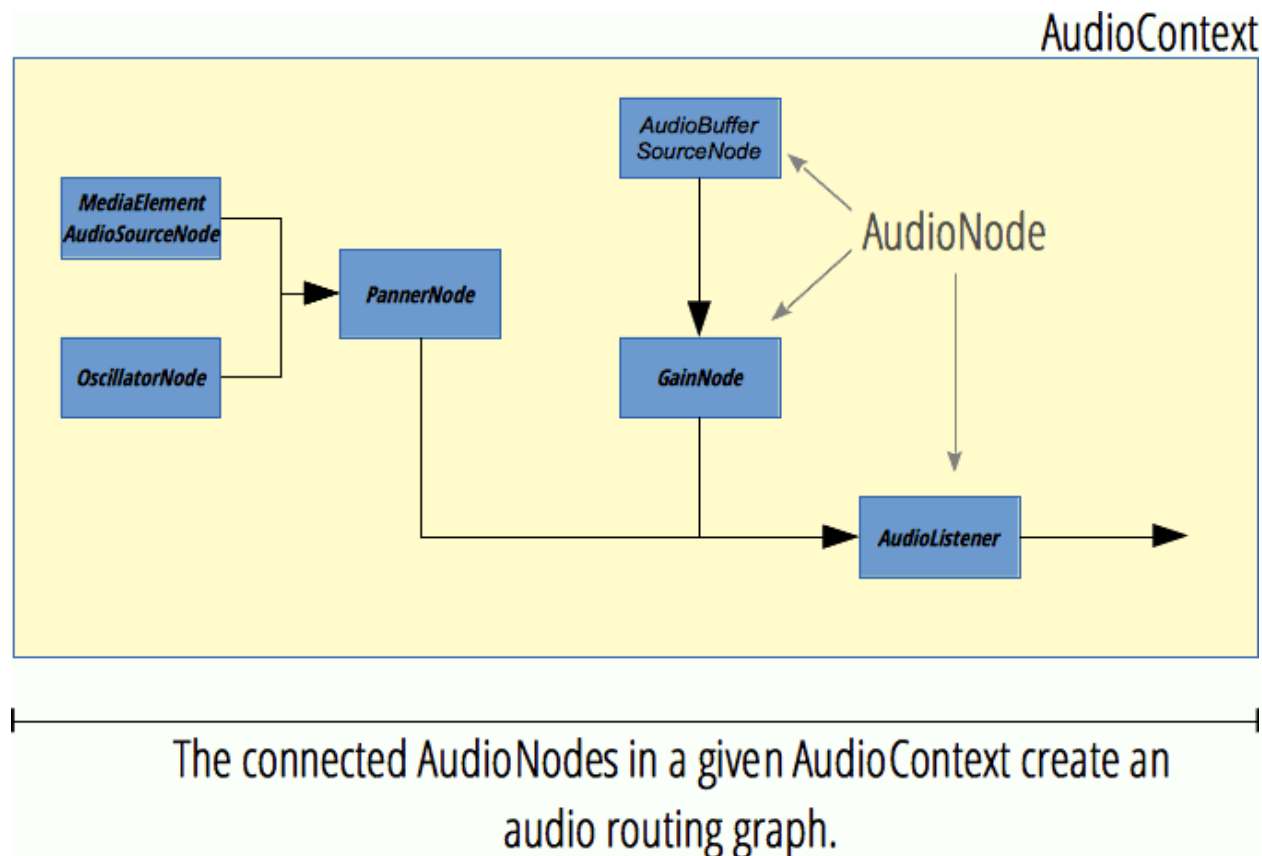


Figure 2-2: AudioNodes (from Mozilla Developer Network): A diagram depicting the layout of AudioNodes within an AudioContext as part of the Web Audio APIs [31].

With the new Web APIs for audio and speech, JavaScript in the browser is powerfully positioned for audio processing and speech recognition. The Web Speech API includes the SpeechRecognition and SpeechSynthesis objects [9], which provide speech recognition and speech synthesis, respectively, as a service (only in Chrome). This is great for black-box recognition, but we need something else in order to use our own speech recognition tools.

Enter the Web Audio API, a versatile interface for controlling audio in the browser [14]. Through this API, developers can generate audio, access the user's audio stream (with permission), and perform built-in analysis or custom processing of the audio stream. The main constructs are the `AudioContext` and the `AudioNode`. Within an `AudioContext`, many `AudioNodes` performing different functions can be linked together in a processing graph. For example, hooking an `AudioBufferSourceNode` into an `AnalyserNode` will perform a configurable FFT on the audio stream from the source node, which can then be used to produce visualizations of the audio's frequency spectrum or volume level. Linking an `AudioBufferSourceNode` to a `ScriptProcessorNode` allows you to define your own custom audio processing function, which could transform the raw audio in some way (changing the encoding format or downsampling) or send the audio data over to a backend server. The Web Audio API underpins the client-side recording functionality of Spoke.

Chapter 3

System Components and Design

Spoke is a framework for building real-time, interactive speech applications on the web backed by speech technologies built in the SLS group. To simplify development of these applications, Spoke has a piece for each side of the web application, the client-side and the server-side. Spoke also enables the development of backend processing scripts in JavaScript that could batch process audio offline. In this chapter I further describe Spoke's design, functionality, and implementation, and sox-audio, an adjacent project that further bolsters application development in the lab.

3.1 System Overview

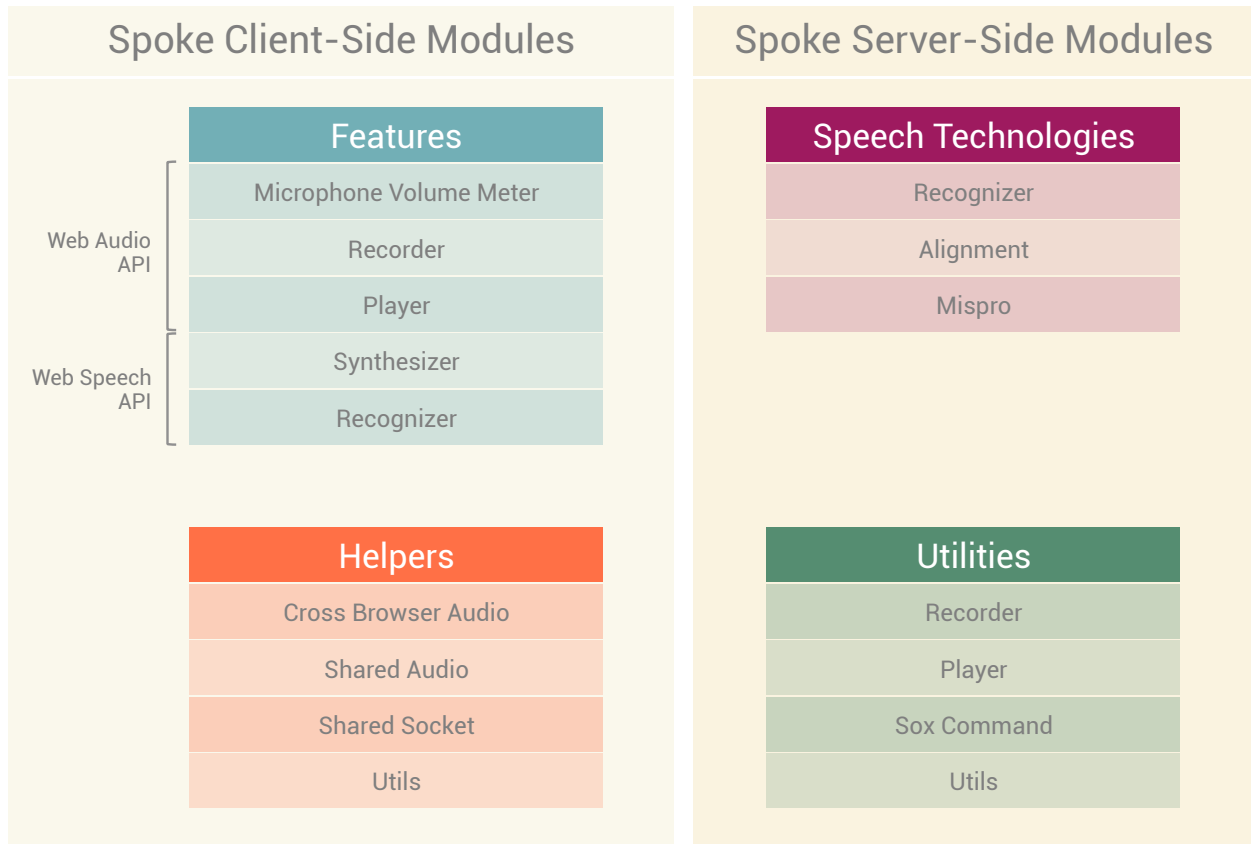


Figure 3-1: Spoke Module Overview: *Spoke is split into two main components – a client-side framework and a server-side library. The client-side framework includes feature and helper modules to enable the creation of interactive UI elements (e.g. play and record buttons, volume meters, front-end connections for recognition and recording). The server-side library includes modules to help with things such as recognition, alignment, recording, and more. These server-side modules can be used as part of a web server, or in batch processing scripts.*

The two pieces of Spoke, the client-side framework and the server-side library, should be used in conjunction to build real-time, interactive speech applications demonstrating our research. The Spoke client-side framework provides tools for building the front-end of

speech applications and communicating with the server, including recording audio to the server, playing audio, and visualizing volume level. The server-side of Spoke provides a library that integrates some speech technologies built in the lab, such as recognition and mispronunciation detection, and a utility belt of audio tools such as recording, transcoding, and playback.

The implementations of these components follow the same high-level design principles and paradigms for flexibility, abstraction, and consistency. First and foremost is modularity of the components, breaking each component into logically distinct modules, and breaking down the actions of each module into logically distinct functions for maximum flexibility. For further flexibility, every module object should accept an (optional) options parameter, allowing customization to a large extent. One salient example of both of these principles is the volume meter module: you can customize the minimum height of the volume meter with ‘options’, and you can customize the translation of volume level into meter height by overwriting one method of a VolumeMeter instance. Finally, the design principally follows prototype-based object-oriented programming, so each module is constructed as an object with properties and methods; this allows stateful variables to be encapsulated in the object and accessed by all its methods, without adding excessive parameters to function signatures that the developer then becomes responsible for managing.

3.2 Spoke Client-Side Framework

The Spoke client-side framework, spoke-client, is composed of a CSS library, a few low-level helper modules and many high-level feature modules (see *Figure 3-1*). The high-level modules employ prototype-based object-oriented programming and the publish-subscribe

pattern to create modules with encapsulated state that operate rather autonomously after setup and then notify the developer about key events. This event-driven model is implemented using jQuery for event emitting, and events follow the standard namespacing convention `eventName.package.module` (e.g. `start.spoke.recorder`). Some of these modules are driven by user interaction, so they accept an HTML element in the constructor and register themselves as click listeners on that element. All of them accept an options object for customizing the configuration. Through this framework the following functionality is enabled:

- Recording audio to the server for processing with Spoke's integrated speech technologies (technologies built in the lab and incorporated into the Spoke library)
- Speech recognition and speech synthesis through the Web Speech APIs
- Playing audio from the server
- Visualizing audio information, such as volume level

In this section I will explain some of the key client-side modules in more detail.

3.2.1 CrossBrowserAudio

The low-level helper modules in `spoke-client` help simplify and decouple the feature modules. In particular, the `crossBrowserAudio` module resolves two key issues: vendor prefixing and audio sharing.

Many of the feature modules depend on vendor-prefixed APIs, such as `navigator.getUserMedia` and `window.AudioContext`. These APIs can alternatively be provided under prefixed names such as `webkitGetUserMedia` or `mozGetUserMedia` in a browser-dependent way. The `crossBrowserAudio` module uses a library called `Modernizr` to ensure these APIs will be available under their non-prefixed names.

```
1 window.AudioContext = Modernizr.prefixed('AudioContext', window);  
2 navigator.getUserMedia = Modernizr.prefixed('getUserMedia', navigator);
```

Figure 3-2: Resolving Vendor-Prefixed APIs with Modernizr:
CrossBrowserAudio uses Modernizr to find browser APIs under their vendor-prefixed names and make them available under a non-prefixed name to simplify cross-browser development.

The main area that needed decoupling was the shared dependence of the VolumeMeter and Recorder on the user's audio stream. Accessing the user's audio stream through `getUserMedia` is an asynchronous task that requires the user's explicit permission, so instead of returning the user's audio stream, `getUserMedia` accepts a success callback and an error callback to invoke in the event of success or failure respectively. Typically, any action involving the user's media stream happens inside the success callback function provided at the time of the request.

In Spoke, there are multiple modules that need to take action on the user's media stream once it is available, and those modules live in separate files and ought to be completely independent from one another. Instead of each module making its own request for the user's audio stream (which requires permission to be granted again unless the connection is secure), we can make one request in a helper module and then multiplex access to the resulting stream when it is available using Promises. `CrossBrowserAudio` does this by defining `promiseUserMedia`, a promisified version of `getUserMedia` that returns a Promise that will be fulfilled with the user's media stream if permission is granted, or will be rejected if permission is denied.

3.2.2 Microphone Volume Meter

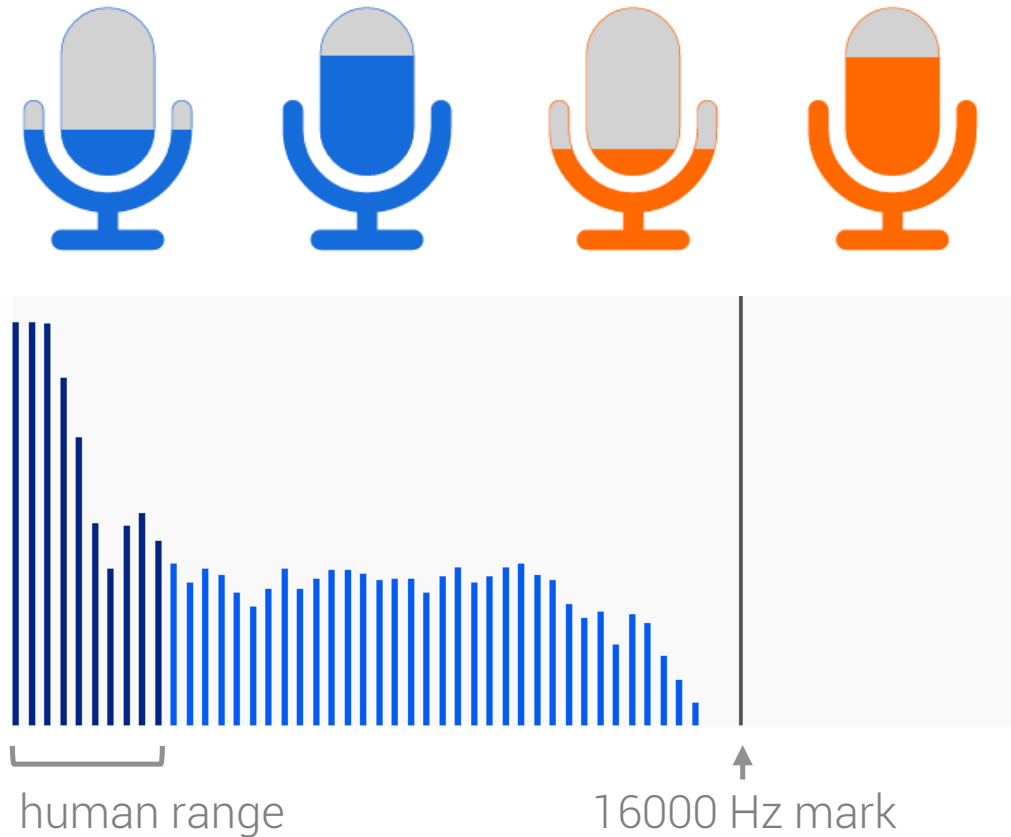


Figure 3-3: Sample Microphone Volume Meters: *Top: The VolumeMeter enables volume visualizations on a height-adjustable HTML element. Typically a microphone icon is used. Bottom: This frequency spectrum visualization illustrates how the VolumeMeter extracts the frequency bins corresponding to the dominant human speech range (from 300 to 3300 Hz, i.e. telephone bandwidth, in dark blue) when computing the volume level. Frequencies above 16 kHz rarely appear in the FFT.*

The microphone module includes a VolumeMeter object for computing and visually representing the volume level of the user’s audio. This module is intended to provide various such objects for processing and/or visualizing audio from the user’s microphone, for example it could be extended to include a live spectrogram analysis.

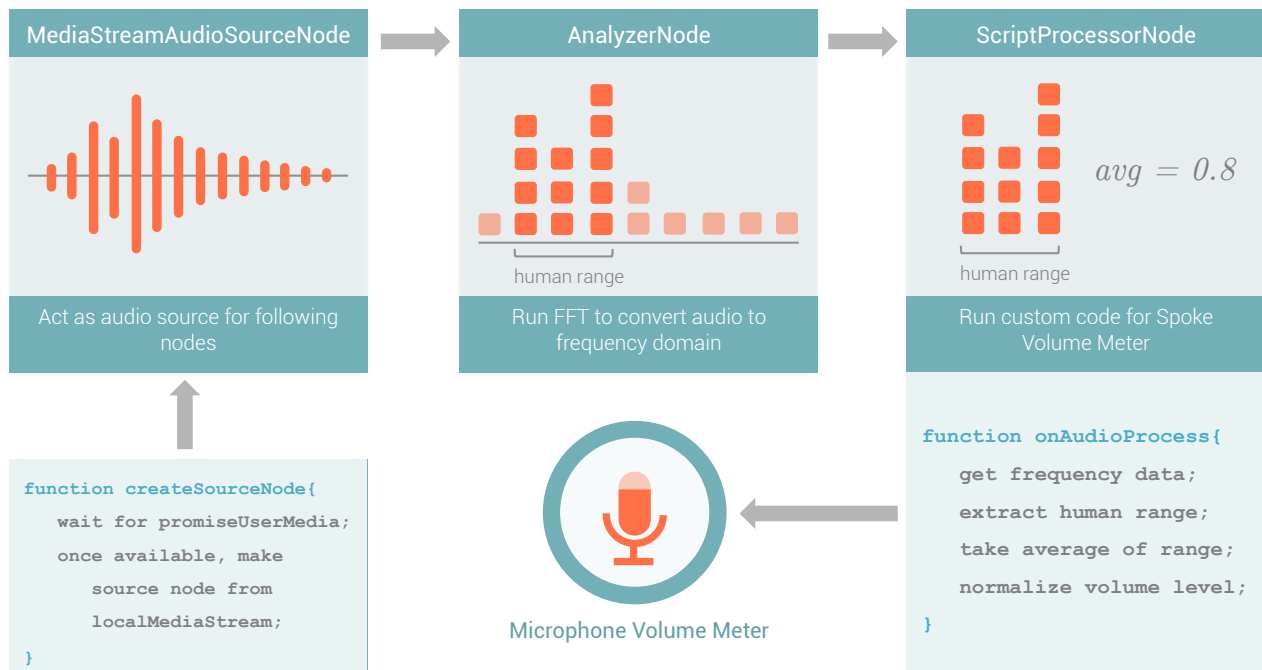


Figure 3-4: Microphone Volume Meter Design: *The VolumeMeter creates an audio processing graph of AudioNodes to compute the volume level. The AnalyserNode performs an FFT on the user’s audio stream, and the ScriptProcessorNode extracts the frequency data corresponding to the dominant frequency range of human speech, 300 to 3300 Hz. The final volume level is computed by averaging over this range and then normalizing.*

The VolumeMeter employs the Web Audio API to create an audio processing graph of AudioNodes (Figure 3-4). The first node in the processing graph is a MediaStreamAudioSourceNode created from the user’s local media stream, which is fulfilled by promiseUserMedia. The source node connects to an AnalyserNode, which performs an FFT of a given size on a small buffer of audio data from the source node. The result of this FFT is an array of frequency data in which each bin (array index) corresponds to a moderate range of frequencies and the bin’s value corresponds to the intensity of the frequencies in the bin.

In order to process the FFT frequency data, I connect the `AnalyserNode` to a `ScriptProcessorNode`, which calls a custom processing function with a buffer of audio data. This processing step computes the current volume level by averaging over the slice of the frequency array corresponding to the dominant frequencies of human speech (300 to 3300 Hz), and then normalizing this average value to get a volume level between 0 and 1.

Usage: The `VolumeMeter` component is extremely modular—the developer can configure every step in the audio processing pipeline and even swap out whole functions for more customization. For example, the `VolumeMeter` is initialized with an HTML element to be height-adjusted as the volume changes (*Figure 3-5*), and this UI update is factored out into the `adjustVolumeMeter(volumeLevel)` method. The developer can overwrite this method on their volume meter instance to change how the computed volume level translates into a UI update, and in fact this customization is done frequently in the demos built with a templating engine. For other plain actions the developer wishes to take when the volume level is computed, she can register a listener on the `volumeMeter` instance for the `volumeLevel.spoke.volumeMeter` event.

```
1 var options = {minMeterHeight: 15, analyserFftSize: 128};  
2 var volumeMeter = new mic.VolumeMeter(element, options);
```

Figure 3-5: Microphone Volume Meter Usage: *VolumeMeter* instances can be initialized with an HTML element and an options object to configure certain properties of the `AnalyserNode`'s FFT, the `ScriptProcessorNode`, and the meter element.

3.2.3 Recorder

```
1 var rec = new Recorder(element, options);
```

Figure 3-6: Client-Side Recorder Usage: *The Recorder is initialized with an HTML element that will toggle recording and an optional options object where the developer can specify the buffer length for the Recorder’s ScriptProcessorNode and the metadata to be sent to the server along with the stream of audio.*

The Recorder module hooks audio recording onto an element of the webpage and records audio to the server. In the constructor, the recorder accepts an HTML element and then registers itself as a click listener on that element, such that the recorder is toggled on and off with each click (*Figure 3-6*). When toggled on, the recorder begins streaming raw audio data to the server over socket.io and emits a ‘start’ event. Similarly, the recorder emits a ‘stop’ event when the user ends recording and a ‘result’ event when the audio stream is successfully saved on the server.

Similar to the VolumeMeter, the Recorder creates an audio processing graph with a few AudioNodes (*Figure 3-7*). The processing graph begins with a MediaStreamAudioSourceNode created from the user’s local media stream, which is fulfilled by promiseUserMedia. This audio source feeds directly into a custom ScriptProcessorNode where the raw audio data is processed one buffer at a time. This processing step is two-fold, first transforming the audio samples and then writing a buffer of transformed samples to the socket stream.

1. **Audio Sample Transform:** The browser provides the user's audio as a buffer of raw PCM 32-bit floating-point samples. The buffer size is configurable but defaults to 2048 bytes, meaning it holds 512 Float32 samples. Each of these samples is converted to a 16-bit signed integer, resulting in a 1024 byte buffer. Given the standard browser audio sample rate of 44.1kHz and the recorder's default buffer size, one buffer contains approximately 12 milliseconds of audio. This custom processing function is called every time the buffer fills with new audio data, which translates to approximately 86 times a second.
2. **Writing to the Stream:** Audio streaming from the recorder is underpinned by `socket.io-stream`, a wrapper for `socket.io` that allows you to transfer large amounts of binary data from the client to the server or vice versa with the Node.js Stream API. Right before recording starts, the recorder creates a new stream with `socket.io-stream` and emits an event on the socket with that stream and some metadata to the server. Then in the processor function, after the audio samples are converted to 16-bit integers, the samples need to be written to the stream; however, the audio buffer cannot be written directly to the stream. The stream, an implementation of the Node.js Stream API, requires a Node-style Buffer object as well, which is not interchangeable with the browser's `ArrayBuffer` object. So a Node-style Buffer is created from the data in the browser-style audio buffer, and then this Node buffer is directly written to the socket stream.

Each buffer of audio data is transmitted over socket individually, but the `socket.io-stream` wrapper implements the streaming interface around these transmissions. When `socket.io` is able to establish a `WebSocket` connection to the server, we are guaranteed the in-order delivery of each audio buffer transmitted.

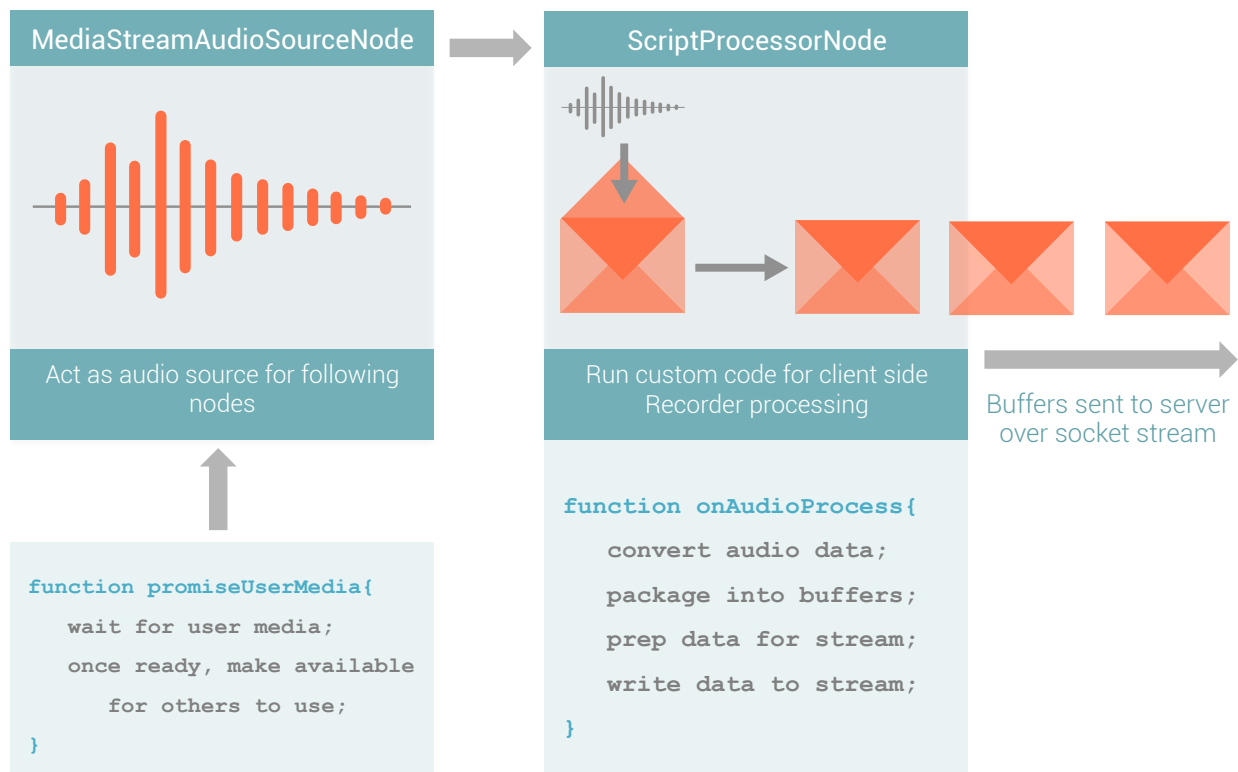


Figure 3-7: Client-Side Recorder Design: *The Recorder creates an audio processing graph consisting of a `MediaStreamAudioSourceNode` connected to a `ScriptProcessorNode`. The audio source for the `MediaStreamAudioSourceNode` is supplied by `promiseUserMedia`. The `ScriptProcessorNode` operates on a buffer of audio data, converting it to the right encoding and the right type of buffer before writing it to the socket stream.*

Usage: To integrate a recorder instance with the user interface and other JavaScript components, the developer subscribes listeners to a few recorder events: ‘start’, ‘stop’, and ‘result’. The two most likely values for configuration are the buffer length and the audio metadata. The buffer length refers to the size of the buffer (in bytes) that is passed into the `ScriptProcessorNode` for processing. The shorter the buffer, the lower the latency between some audio being produced and the recorder processing that audio; however, longer buffers may help reduce audio breakup and glitches if the processing behaves differently at

the boundaries of the audio. The audio metadata is a preset object sent along with the recorder's audio stream to the server every time a recording is made through this recorder instance. This is primarily useful for when you have multiple recorder instances on the page with different associated data that the server needs to know, such as prompted text.

3.2.4 Player

```
1 var player = spoke.Player(audioStream);
```

Figure 3-8: Client-Side Player Usage: *The Player is initialized with an audio stream and will automatically play the audio as soon as it is ready. An options object may be passed in during initialization to prevent this default behavior.*

The Player module enables playing an audio stream (such as one from the server) over the user's speakers with the Web Audio APIs. Unlike the previously discussed feature modules, the Player does not hook onto an HTML element but rather an audio stream. The developer writes her own playback request to emit over the Socket.io socket on a particular user interaction, and then adds a socket listener for the server's audio stream response. In this listener, the developer can create a new Player instance to play the audio stream returned by the server and register listeners for when the player is 'ready' to begin playing and when it is 'done' playing.

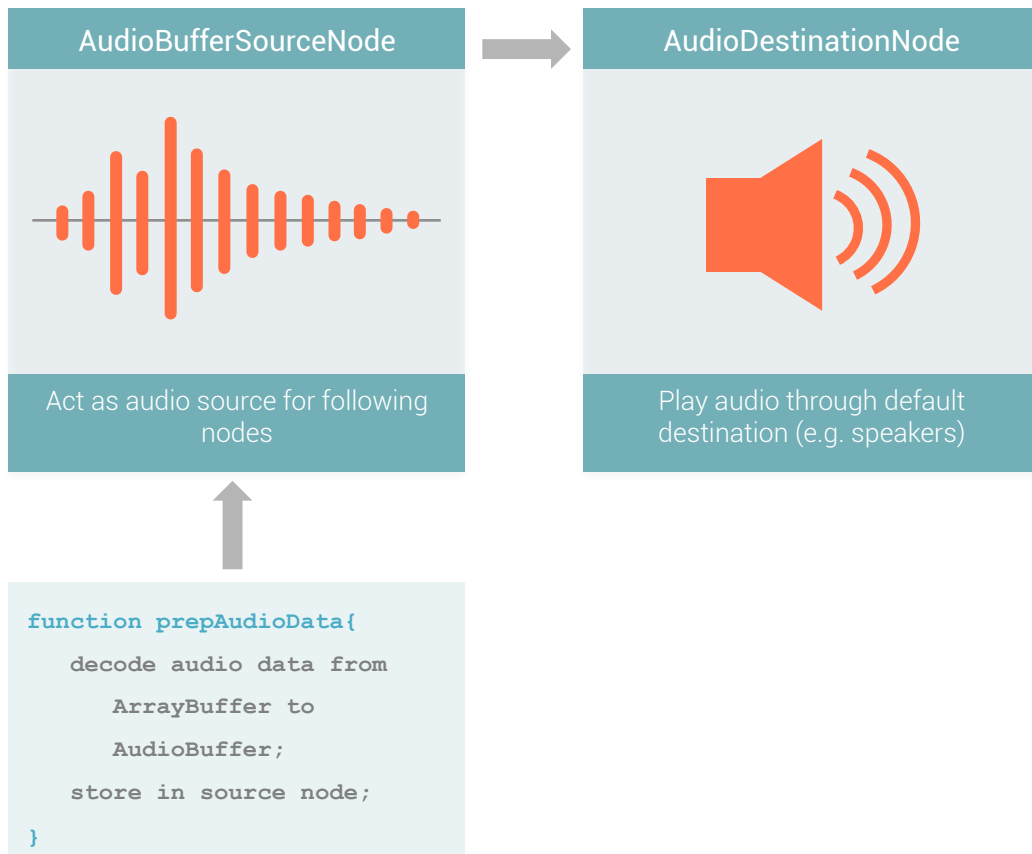


Figure 3-9: Client-Side Player Design: *The Player routes audio from an `AudioBufferSourceNode` to an `AudioDestinationNode` that plays the audio over the user's speakers. The Player accumulates the audio data from the stream and consolidates it into one `ArrayBuffer`. Using the `decodeAudioData` API, it transforms the `ArrayBuffer` into an `AudioBuffer` to provide to the source node.*

Over the stream, audio data arrives a buffer at a time. Internally the Player listens to the provided audio stream, consolidating all these audio data buffers into one buffer for playback. The main buffer construct on the browser is the `ArrayBuffer`, a fixed-length raw binary data buffer. Working with these buffers can be cumbersome because The `ArrayBuffer`'s raw contents cannot be directly manipulated and its size in bytes is fixed at initialization, which makes working with it cumbersome:

- To manipulate the data in an `ArrayBuffer`, you have to create a `TypedArray` around it, which provides an interpretation of the raw binary data as data of a certain type, e.g. `Uint8Array` is a `TypedArray` that provides a view of the data as unsigned 8-bit integers.
- To concatenate two buffers, you have to create a new larger buffer and copy over the data from each smaller buffer.

Nevertheless, the player needs to merge all the small buffers of audio data from the stream into one large buffer for playback. As the player listens to the audio stream, it gathers all the audio data buffers into an array of buffers. When the stream ends, it counts the total number of bytes of audio data from all the small buffers and makes a new empty `ArrayBuffer` of that size, then copies a `TypedArray` of each small buffer into the appropriate section of a `TypedArray` of the large buffer.

Once the audio data is consolidated, the player emits the ‘ready’ event and will proceed to play the audio using the Web Audio API. The `ArrayBuffer` of audio data first needs to be converted to an `AudioBuffer` that an `AudioBufferSourceNode` knows how to play. This is done asynchronously with the API’s `decodeAudioData` method, and then the player makes a new `AudioBufferSourceNode` for the resulting `AudioBuffer`. This source node is connected directly to an `AudioDestinationNode` representing the user’s speakers. By default the Player will automatically play the audio once it is ready, but it can be configured to wait and let the developer manually trigger playing later. The developer can also listen for the ‘done’ event to make UI updates when playback has finished.

3.2.5 Recognizer and Synthesizer

The Recognizer and Synthesizer modules are wrappers around the two core pieces of the Web Speech API, the `SpeechRecognition` and `SpeechSynthesis` interfaces, which provide

speech recognition and speech synthesis, respectively, as a service. Though most browsers do not support the Web Speech API because it is still experimental, Chrome provides full support and support in Firefox is pending. The Spoke wrapper modules abstract some of the common patterns for working with these interfaces to simplify development. At the same time, integrating these interfaces into Spoke provides greater visibility for them in the lab and an easier starting point for most developers.

The Recognizer module simplifies both hooking recognition toggling onto an element of the page and taking action on the recognizer's state changes and final results¹. In the constructor, the Recognizer sets a click listener on the HTML element that was passed in such that recognition is toggled on and off with each click. When toggled, the Recognizer either emits a 'start' or 'stop' event that the developer can listen for to make appropriate UI changes on the page. Then when the underlying SpeechRecognition object has new speech recognition results, it emits a SpeechRecognitionEvent object that contains an array of SpeechRecognitionResults, each of which contains an array of SpeechRecognitionAlternatives specifying one possible transcript and its confidence score. The Recognizer emits a 'result' event passing through this deep SpeechRecognitionEvent object, but as a convenience for developers it also searches through the event for the best transcript of the most recent final result and emits that with a 'finalResult' event.

The SpeechSynthesis interface is far less complex and thus the Synthesizer module provides only a thin wrapper for it. A Synthesizer instance is created with a string of text to translate to speech and will automatically prepare and play the synthesized speech if

¹Note that the Recognizer module can also be used alongside the Recorder module, hooked onto the same button, and listening to the same audio.

autoPlay is set to true (the default). The ‘play’ method takes care of trying to play the utterance immediately, which involves cancelling any currently-playing utterances and clearing the utterance queue before queuing the current utterance and resuming the SpeechSynthesis’s playing. In addition to the standard ‘start’ and ‘end’ events, the Synthesizer also re-emits a ‘boundary’ event triggered when the currently playing utterance reaches a word or sentence boundary.

3.2.6 CSS Library

The spoke-client package also provides a CSS library for styles shared throughout the lab. This library is small to begin with, but with greater use of Spoke we can add more styles that will help standardize the look of websites and their components built by SLS. Currently the library includes some utility classes for text styling and classes for styling a microphone icon that is used in many sites around the lab.

3.2.7 Package Distribution and Usage

Spoke-client uses RequireJS to explicitly manage its module dependencies and asynchronously load them in the browser, and it uses the r.js optimizer to build and optimize itself for production. The r.js optimizer is a command line tool that allows you to get the best of both worlds out of RequireJS—source code divided into modules in separate files with managed dependencies for development, and then one minified self-contained file for production. It takes the name of the file to build and then follows its dependency chain, building a standalone output file that concatenates the original file and all its dependencies, and then optionally minifies it. Build options can be specified on the command line or in a simple JavaScript build file. Spoke-client does the latter, defining three builds, one for the minified CSS library spoke.min.css (1 kb), and one each for a development and production

version of the JavaScript modules, `spoke.js` (198 kb) and `spoke.min.js` (79 kb) respectively. The ultimate result is a modularized, production-ready file that can be required (using RequireJS) into other projects to provide all the functionality of `spoke-client`.

3.3 Spoke Server-Side Library

The Spoke server-side library provides modules of two types—audio utilities and speech technologies—geared towards building a speech processing backend for a webserver, but the library can also be used for offline scripting and batch processing (see *Figure 3-1*). As a library, not a framework, it is up to the developer to listen for client events and then decide which pieces of the library to use. The library’s utilities enable recording raw or wav audio, transcoding audio files or live audio streams, and streaming audio from a saved file. The library also provides standardized access to and usage of some speech technologies built in the lab, such as

- domain-specific speech recognizers
- forced alignment of an audio sample to expected text
- mispronunciation detection on a set of utterances from the same speaker

These integrated speech technologies support the key functionality we wish to demonstrate through sample web applications, and any technology that has a command line interface can also be integrated into Spoke, even a hardware recognizer. This makes the technology visible, accessible, and usable for website backends and offline JavaScript processing. For technologies with complex outputs, such as forced alignment, the technology module is paired with an output parser that transforms the output into a usable JavaScript object.

Each module employs prototype-based object-oriented programming to export one object providing a clean interface to a tool or technology. Under the hood, all of these modules operate by running a command in a new process, though the utility modules do this indirectly through the SoxCommand interface. In Node’s single-threaded model, asynchronously creating these new processes is essential, allowing the developer to spin up a long running process and then wait for a callback; Node provides a `child_process` module to accomplish exactly this. Callbacks are the standard Node.js way, but they are not always the best way for handling asynchronous control flow when multiple asynchronous steps need to be chained together. In this case, the preferred option is that the command execution method returns a Promise instead of calling a callback, since Promises can easily be chained to create a sequential pipeline of asynchronous tasks². Each module in the library supports both styles, callback-based and Promise-based, leaving it to the developer’s discretion for her particular case.

3.3.1 Audio Utilities

Spoke provides a utility-belt of audio tools that complement its speech technologies with support for recording audio streams to a file, creating audio streams from a file, and building arbitrarily complex SoX commands that will be run in a new process. This last piece is enabled by `sox-audio`, a Node.js package I built as part of this thesis but that, independent of Spoke, is useful for the community of Node.js developers, so it was released as a standalone package on NPM. The Recorder and Player modules then use `sox-audio` to do most of the heavy lifting.

² Not to be confused with a streaming pipeline, which is also asynchronous but does not wait for one step to finish before the next one begins.

3.3.1.1 Recorder

The Recorder module records audio from a raw input stream to a file, either in the original raw format or in an optionally downsampled wav format. This feature of Spoke alone is greatly useful, enabling live audio recording on the web for the collection of audio samples for training and testing of new speech technologies.

At initialization, the Recorder can be configured with information about the raw input streams it will be handling, such as their bit depth and sample rate, and with the desired sample rate for converted wav files. The crucial method is `convertAndSave`, which accepts a raw audio file or raw audio stream and a wav file or stream for outputting the transcoded results; transcoding is carried out with a `SoxCommand` built from the method's parameters for input and output streams or files and the Recorder configuration for sample rates, etc. For offline processing, this can be used to transcode already saved raw files to wav files. More interestingly, as part of a web backend, this can be used to perform streaming transcoding of a raw stream to a wav stream, which could start being processed before all the audio has been received if a speech technology accepted streaming audio. As it is, all of them currently accept only saved wav files, but this should change over the next year.

3.3.1.2 Player

Turning a saved file into a stream is simple in Node.js, but turning only part of an audio file into a stream or concatenating multiple audio files, or parts of them, into one stream is not. The Player module enables all of the above, using `SoxCommands` for audio trimming and concatenation. Given a wav input (file or stream) and the start sample and end sample of the desired section, the Player builds a `SoxCommand` that cuts the audio at the specified samples and pipes out only the desired section to a provided stream. Similarly, a list of wav

files can be trimmed and concatenated such that the resulting audio stream starts at the specified start sample of the first file and ends at the specified end sample of the last file in the list, with intervening files included in their entirety.

3.3.2 Integrated Speech Technologies

A core subset of the lab’s speech technologies have been integrated into Spoke allowing for their use in Node.js server backends or processing scripts. Each technology has its own module providing a clean JavaScript interface to its typical command line usage. Each module has a default configuration for calling a certain set of scripts to carry out the execution steps of the technology, but alternative scripts to call for these steps can be provided in the ‘options’ object; this means another implementation of the same technology can be swapped in³ at run time and used through the same JavaScript interface.

Though these technologies can be used through Spoke for offline processing, Spoke is first and foremost designed to enable interactive speech applications on top of the lab’s technologies—thus the functionality of each module is broken into small logical steps as much as possible to allow for the shortest delay before responding to the user. Technologies primed for integration into Spoke should mirror this pattern, breaking the processing into separate logical steps that will be meaningful to other developers or the end user. A great example of this is in how the mispronunciation detection technology was designed: originally it consisted of one long-running process called after all utterances had been collected, but it was split into a pre-processing step to be called on each utterance after it is saved and a shortened mispronunciation detection step after all utterances are collected.

³ As long as it adheres to the same script interface for inputs and outputs, including ordering of arguments for input and location and formatting of output.

```
1 var nutritionOptions = {
2     recognitionScript: 'my/local/path/nutrecognizer.sh',
3 };
4
5 var nutritionRecognizer = new Spoke.Recognizer(nutritionOptions);
```

Figure 3-10: Recognizer Configuration: *Spoke's Recognizer interface can be used to interact with any recognizer that implements a basic command line interface that accepts a wav filename and outputs the recognition results on stdout.*

3.3.2.1 Recognizer

The Recognizer enables JavaScript integration of any speech recognizer built in the lab with a command-line interface taking a single wav filename and outputting the recognized text on stdout. A new Recognizer instance will by default use a Kaldi recognizer trained on nutrition logs (easy to change for Spoke, but that's what we've been using mainly), but it can be configured with a new path to any recognition script with 'recognitionScript' in 'options' (*Figure 3-10*). Recognition is performed asynchronously on a wav file with either the 'recognize' or 'recognizeAsync' method, depending on whether the developer wants to use callbacks or Promises; either way, the recognition script is executed with Node's `child_process` module and the script's output on stdout will be provided to the developer asynchronously (i.e. when available).

3.3.2.2 Forced Alignment

The Alignment module provides a simplified JavaScript interface for performing forced alignment with Kaldi. The developer must provide an `outputDir` at initialization, an existing directory where the forced alignment script can put output and associated processing files, and then call the `initDir` method on the Alignment instance before any forced alignments can be performed. The ‘forcedAlignment’ method requires a wav file and a txt file to which to align the speech and performs forced alignment on them, asynchronously returning the timing results txt file; behind the scenes, the Alignment module handles also passing the `outputDir` to the alignment script, simplifying the interface presented to the developer.

1	[start sample]	[end sample]	[phoneme]	[word]	[wordboundary]
2	24960	26080	w	warm	I
3	26080	27520	aa		
4	27520	28000	r		
5	28000	28960	m	warm	E

Figure 3-11: Example Output from Forced Alignment: *Running forced alignment on a wav and txt file results in a timing file with a simple column format as shown above. This output contains the timing data for each phoneme of each word in the txt file, but word-level timing can be extracted by taking the start sample from the first phoneme and the end sample from its last phoneme.*

The timing file has a simple column format specifying on each row a phoneme, its start sample and its end sample, with two optional columns for the word the phoneme is from and whether the phoneme is at the start or end of this word. This file is difficult to use directly, so the Alignment has an associated parser that reads in this file with phoneme-

level timing information and extracts word-level timing information. The method `getAlignmentResults` handles parsing a timing file and asynchronously returns a list of objects representing words in the alignment with their start samples, end samples, and a list of their phonemes. Again, each of these methods can be chained with Promises, leading to easily readable code.

```
1 alignment.forcedAlignmentAsync(wavFilename, txtFilename)
2   .then(function (timingFilename) {
3     return alignment.getAlignmentResultsAsync(timingFilename);
4   });
```

Figure 3-12: Forced Alignment Result Chaining: *The output of `forcedAlignmentAsync` is a Promise that will be fulfilled with the name of the timing file. This Promise can be awaited with ‘then’ to perform some action on its completion, namely to parse the timing file with `getAlignmentResultsAsync`, which also returns a Promise that can be awaited and chained in a similar manner.*

3.3.2.3 Mispronunciation Detection

The `Mispro` module wraps the mispronunciation detection technology built in our lab by Ann Lee using the Kaldi toolkit. This technology processes a set of utterances from the same speaker over a set of known text and determines the n most likely mispronunciation patterns in the collected speech. An integral part of this system is performing forced alignment on each (utterance, text) pair in the set, so each `Mispro` instance takes in an `outputDir` then creates its own `Alignment` instance and exposes its methods. Similar to the

Alignment scripts, the Mispro scripts also require this `outputDir`, which the Mispro module handles passing to the script calls for the developer.

The mispronunciation detection system is broken into a few steps to enable more interactivity and shorter delays while processing. Each (utterance, text) pair undergoes a forced alignment step and then a mispronunciation preprocessing step. These two steps are separate because the forced alignment output may be relevant for enabling certain user interactions after recording (such as playing back a certain word), whereas the preprocessing step is purely for the benefit of the system to speed up the final computation over the whole set of utterances. After all utterances from the same speaker have been processed in this way, the `misproDetection` method will perform the final step, finding patterns of mispronunciation, and outputting the identified mispronunciations to `stdout`.

The `stdout` output of the mispronunciation system is not directly useable, so the Mispro module has an associated parser that transforms the output into a suitable JavaScript object. The system iteratively identifies mispronunciation patterns in decreasing order of severity (or probability), and outputs for each pattern its triphone rule and the words in the text set matching the pattern. The parser then constructs an ordered list of objects representing the mispronounced words, with the word itself, the expected and actual phoneme contributing to the mispronunciation, and the location of the phoneme in the word.

```
1 mispro.forcedAlignmentAsync(wavFilename, txtFilename)
2   .then(function (timingFilename) {
3     return mispro.getAlignmentResultsAsync(timingFilename);
4   })
```

```

5     .then(function () {
6         return mispro.preprocessAsync(wavFilename);
7     })
8     .then(function () {
9         var misproOutput = mispro.misproDetectionStream();
10        var misproResultsStream =
11            mispro.getMisproResultsStream(misproOutput);
12    });

```

Figure 3-13: Mispronunciation Processing Chain: *Using Promises, the processing steps of mispronunciation detection can be chained for sequential asynchronous execution. First we execute forced alignment, then parse the timing file, then preprocess the utterance, and finally perform mispronunciation detection. The last step involves a streaming pipeline between the output of mispronunciation detection and a parser that turns results into JavaScript objects.*

3.4 SoX Audio Processing Module

SoX is a command line utility for audio manipulation that can convert between many audio formats and apply various effects to audio files [18]. It supports streaming audio in and out of processing commands and the use of subcommands to provide input for later routines through a set of special filenames. This opens the door to interesting and complex commands that could, for example, trim two files and concatenate them all in one line without creating any intermediate files (*Figure 3-14*).

```
1 sox "|sox utterance_0.wav -t wav -p trim =4.03" utterance_1.wav "|sox  
utterance_2.wav -t wav -p trim 0 =2.54" trim_and_concat.wav --combine  
concatenate
```

Figure 3-14: Example Complex SoX Command: Here we use the special filename `"|program [options] ..."` to create SoX subcommands that trim the start or end off of a single file. Inside this subcommand, the `-p` flag indicates that the SoX command should be used as in input pipe to another SoX command. The overall SoX command has 3 inputs that it concatenates into a single output file: `utterance_0.wav` trimmed to start at 4.03 seconds, `utterance_1.wav` untrimmed, and `utterance_2.wav` trimmed to end at 2.54 seconds.

Initially we were using SoX simply for downsampling and transcoding the raw PCM 44.1 kHz audio data from client browsers to 16 kHz wav files compatible with our recognizers. For this basic usage of SoX's command line interface, creating and executing the command by hand using Node's `child_process` module was rather simple: first save the raw audio to a file, then compose the String command to perform transcoding and execute it (*Figure 3-15*). However, taking this simple processing step to the next level of *streaming* transcoding was vastly more difficult and we realized the need to abstract out the creation and management of the commands. We searched for a Node module to accomplish this, but the ones we found provided access to only a sliver of SoX's capabilities. Seeing how useful it could be to enable all of SoX's capabilities through a JavaScript interface, I chose to take this abstraction beyond just transcoding commands and make it for any SoX command in general.

```
1 var convertFileSox = function (rawFileName, saveToFileName) {
```

```

2     var COMMAND_FORMAT = 'sox -r 44100 -e signed -b 16 -c 1 %s
      -r 16k %s';
3     var commandLine = util.format(COMMAND_FORMAT, rawFileName,
      saveToFileName);
4     var command = child_process.exec(commandLine);
5 };

```

Figure 3-15: Initial Use of SoX Transcoding in JavaScript: *Our early usage of SoX just created a String command to perform transcoding of a raw audio file to a wav audio file and executed it with Node’s child_process module.*

The sox-audio module (currently open-sourced and available for public consumption on NPM) provides a complete Node.js interface to the SoX audio utilities and many usage examples. The main construct is a SoxCommand on which you progressively specify the command’s inputs, outputs, effects, and associated options for each to build arbitrarily complex SoX commands. This interface was heavily inspired by fluent-ffmpeg, a fluent Node.js interface for Ffmpeg that operates in a very similar manner [33]. The SoxCommand accepts any number of inputs and one or more outputs, where an input may be a filename, a ReadableStream, or another SoxCommand, and an output may be a filename, a WritableStream. If another SoxCommand is provided as input, it is treated as a subcommand such that its output is piped into the main SoxCommand as input. The input and output options are very similar, allowing you to specify the sample rate, encoding, file type, etc. for each input and output. With SoxCommands transcoding a raw audio stream to a wav audio stream is easily accomplished just by specifying a few input options and output options (*Figure 3-16*). Even the complex command from the beginning of this section is easy to construct using a couple SoxCommands with the ‘trim’ effect as input to the main SoxCommand and applying the ‘concat’ effect (*Figure 3-17*).

```
1 var command = SoxCommand();
2
3 command.input(inputStream)
4     .inputSampleRate(44100)
5     .inputEncoding('signed')
6     .inputBits(16)
7     .inputChannels(1)
8     .inputFileType('raw');
9
10 command.output(outputStream)
11     .outputSampleRate(1600)
12     .outputEncoding('signed')
13     .outputBits(16)
14     .outputChannels(1)
15     .outputFileType('wav');
16
17 command.run();
```

Figure 3-16: Example SoxCommand for Streaming Transcoding: *In this example a 44.1 kHz raw audio stream of signed 16-bit integer PCM data is converted on the fly to a 16 kHz wav audio stream.*

```

1 var command = SoxCommand();
2 var trimFirstFileSubCommand = SoxCommand()
3     .input('utterance_0.wav')
4     .output('-p')
5     .outputFileType('wav')
6     .trim("=4.03");
7 var trimLastFileSubCommand = SoxCommand()
8     .input('utterance_2.wav')
9     .output('-p')
10    .outputFileType('wav')
11    .trim(0, "=2.54");
12 command.inputSubCommand(trimFirstFileSubCommand)
13    .input('utterance_1.wav');
14    .inputSubCommand(trimLastFileSubCommand)
15    .output(outputFileName)
16    .concat();
17 command.run();

```

Figure 3-17: Example SoxCommand using SoxCommands as Inputs to Trim and Concatenate Audio Files: *This example leverages SoxCommands as inputs to other SoxCommands, allowing for the results of one to be piped as input into the other. The main SoxCommand concatenates three audio files, while the two sub commands handle trimming the first and last files respectively.*

Chapter 4

Sample Applications

The central purpose of Spoke is to enable demonstrations of speech technologies built in the Spoken Language Systems group. These demonstrations take the form of web applications that may apply a mix of client-side and server-side tools to create interactive speech-enabled websites. In this section we discuss three sample applications built with Spoke that together cover all of Spoke's client-side and server-side modules. These applications illustrate the power and flexibility of Spoke's modules as well as how easy they are to use for building front-end features and backend processing. Each of these applications is backed by an Express server using Socket.io and Spoke to handle processing the user's audio data.

4.1 Online Speech-Enabled Nutrition Log

The Spoken Language Systems group is currently developing a speech-enabled nutrition system whose main goal is to enable efficient dietary tracking through an interactive speech web interface. On the nutrition website the user describes a meal she had and her utterance is recognized with the Web Speech API; the nutrition system then uses language understanding techniques to determine the food items, their quantities, and their descriptions, and then fetches appropriate images and nutritional information for each food item to display to the user [32]. Ultimately we want to replace the Web Speech API recognizer on the client-side with our own server-side domain-specific recognizer trained on utterances from real user interactions with the system.

To this end, we wanted to augment the nutrition website's existing functionality with Spoke's volume meter for visual feedback, its recording framework for collecting utterances and its recognizer library for using the custom speech recognizer. However, the nutrition system's backend was already implemented in Java, so it was outside the Node.js ecosystem where the Spoke server-side library is available. Thus while incorporating Spoke's client-side framework was straightforward, incorporating its server-side libraries required setting up a separate JavaScript server for handling recording and recognition.

4.1.1 Nut

For the initial development and testing of this server, I built a small standalone application called Nut that would carry out the same interactions with Spoke that we wanted to enable in the nutrition system. The Nut webpage prominently displays a microphone icon, which acts as both a volume meter and a record button, and the recognized text (*Figure 4-1*). On the client-side, Nut creates a new `VolumeMeter` instance and a new `Recorder` instance on

the HTML element for the microphone icon (*Figure 4-2*). When the icon is clicked, the spoke-client Recorder begins sending the user's audio stream to the server over Socket.io with the event name 'audioStream'. The Nut server handles the 'audioStream' event by sending the audio stream to the Spoke Recorder (server-side) for streaming transcoding to a wav file. Once the wav file is saved, Nut uses the Spoke Recognizer to run the wav file through a Kaldi speech recognizer trained on nutrition logs, and then sends the recognition result back to the client over Socket.io.

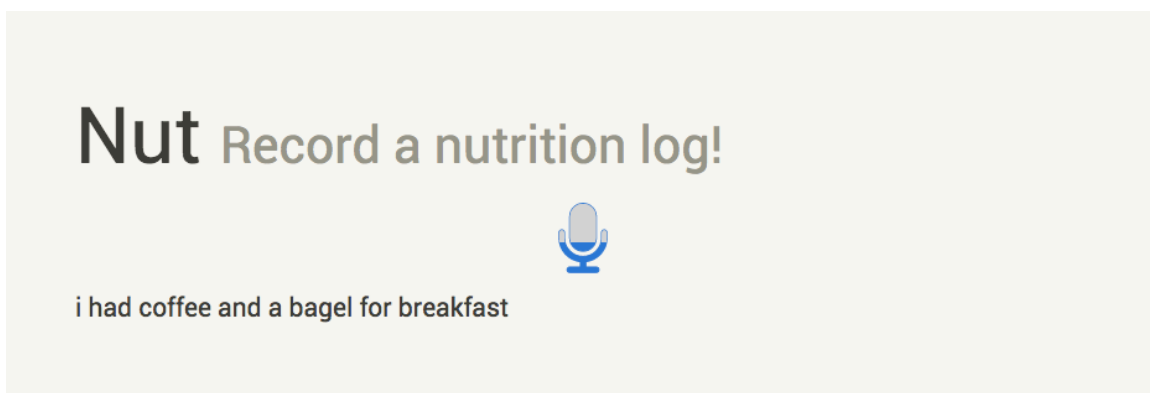


Figure 4-1: Nut Application: *The small Nut webpage displays a volume meter on a microphone icon that doubles as the record button. When the icon is clicked, it changes from blue to red to indicate the recording status. At the end of recording, the audio is run through a recognizer on the server, and the client is notified of the recognition result, which it places below the icon.*

```
1 var element = $('.volumeMeter');
2 var volumeMeter = spoke.microphone.VolumeMeter(element);
3
4 var recorder = spoke.Recorder(element);
5 recorder.on('start.spoke.recorder', {}, function (e) {
```

```

6     /* set icon color to red */
7 });
8 recorder.on('stop.spoke.recorder', {}, function (e) {
9     /* set icon color to blue */
10 });
11
12 recorder.socket.on('recognition results', function (text) {
13     /* update UI with recognition results */
14 });

```

Figure 4-2: Nut VolumeMeter and Recorder Instances: *Nut makes an instance of the VolumeMeter and of the Recorder, both of which are attached to the same HTML element (the microphone icon). We can listen for certain recorder and socket events to appropriately update the UI.*

```

1 var recorder = new Spoke.Recorder();
2 var recognizer = new Spoke.Recognizer();
3
4 ss(socket).on('audioStream', function (stream, data) {
5     recorder.convertAndSaveAsync(stream, wavFilename)
6         .then(function (resultWavFilename) {
7             return recognizer.recognizeAsync(resultWavFilename);

```

```
8         })
9         .then(function (recognitionResult) {
10             socket.emit('result.spoke.recognizer', recognitionResult);
11         });
12     });
```

Figure 4-3: Nut Server Recorder and Recognizer Setup: *The Nut server creates a new Recorder and Recognizer instance when a new socket connection is established. Then it listens for an 'audioStream' event on the socket and handles it by passing the stream to the Recorder for transcoding to a wav file. When the Recorder finishes, the wav file is passed to the Recognizer for recognition.*

4.1.1 The Nutrition System

The client-side usage of Spoke is almost exactly the same in the nutrition system as on the Nut website except for a small amount of configuration to direct Socket.io to the URL for the Nut server. Operating completely independently of the nutrition system's Java server, the Nut server handles recording the user's nutrition logs and running a custom domain-specific speech recognizer on them. These recognition results are returned to the client but are not yet used by the nutrition system because the recognizer needs more training. The recording we have enabled will help collect utterances for this training the more the system is used.

Record a nutrition log!



he had **a** **small** **Chiquita** **banana**
Quantity Description Brand Food

Food	Quantity	USDA Hits
Banana	a	Select further adjectives: <ul style="list-style-type: none"> dehydrated raw See more options

Figure 4-4: The Nutrition System: *The microphone icon acts as volume meter and triggers both recognition with the Web Speech APIs and recording to the Nut server when clicked. The nutrition system determines which parts of the utterance correspond to food items, quantities, descriptions, and brands, and tries to provide nutritional information about the identified food.*

4.2 Amazon Mechanical Turk Audio Collection

The audio recording capabilities of Spoke are particularly expedient when applied to collect audio data through Amazon Mechanical Turk (AMT) tasks [33]. Given the tens of thousands of written nutrition logs collected for the nutrition system and our desire to build a custom speech recognizer for the nutrition system, the nutrition domain was a prime candidate for this audio collection task. For the first round of audio collection, we selected

and cleaned 1,000 nutrition logs, and in the second round we selected 5,000 logs with minimal cleaning. In each round, we assigned each log a unique utterance ID and split them into groups of 10 with a unique group ID.



For this AMT task, I built a task-oriented utterance collection website with Spoke that displays instructions and a set of 10 utterances for the AMT workers (sometimes called “Turkers”) to record. Each utterance has its own record button that is hooked up to an instance of the spoke-client Recorder and configured with specific metadata to send to the server along with the audio. This metadata specifies the utterance ID and the expected utterance text that the Turker should be reading.



The server creates a new recording directory for each Socket.io connection it receives. Then the server handles an audio stream and its metadata by saving the raw audio and the transcoded wav file with the Spoke server-side Recorder, and saving the utterance text from the metadata to a txt file. The filenames for these three files includes the unique utterance ID included in the metadata. Upon successfully saving these files, the server responds to the client with an ‘audioStreamResult’ event, echoing back the metadata along with the path to the saved wav file to be included in the AMT results for the task.



As an early quality control measure, we wanted to verify that the utterances being collected at least partially matched the expected text before allowing the Turker to submit their assignment. Our nutrition recognizer could not be used for this purpose (since we were still collecting data to train it), but the recognizer available in the Web Speech API was well suited for the job. Thus while the audio is being recorded to the server, it is simultaneously being processed with the Web Speech API, used through the spoke-client Recognizer module. The final recognition results are compared to the expected text for exact matches and partial matches—if the two match exactly, we accept the recording and mark that utterance as “Perfect”; if at least 60% of the expected words are present in the


recognized text, then we accept it and mark it as “Great”; otherwise, we mark it as “Redo” and require that the Turker record the utterance again.

Please record each of the sentences below.

1.   For lunch I had a frozen pizza.

2.   I had five ounces of salted peanuts.

3.   I had six strawberries and cream.

4.  Last night I had homemade vegan lasagna.


5.  It had noodles, tomato sauce, spinach and vegan cheese.

Figure 4-5: Audio Collection Task on Amazon Mechanical Turk: *The Turkers are presented with a set of 10 sentences or sentence fragments to record (5 shown in this screenshot). While recording one utterance, the record button turns into a stop button and the other buttons on the page are disabled. After recording, the UI is updated with feedback about the quality and success of the recording. Sentences marked with “Redo” must be re-recorded before submission.*

Using AMT’s ExternalQuestion data structure we can embed this webpage in a frame on the Turker’s browser [34]. Any data we want to collect within AMT has to be in a named field in a form that gets posted to a specific URL of the Mechanical Turk website. In this

task, we save the group ID, the Turker's ID, and each of the utterance IDs, their expected and recognized texts and the paths to their wav files on the server.

Using Spoke for this high traffic application allowed us to observe the client-server audio streaming under great load while collecting thousands of utterances. Theoretically a Node.js server can handle thousands of Socket.io connections if the server is properly architected to minimize CPU usage on the application thread. The Spoke server-side library excels at this: all CPU-intensive processing methods operate by creating a new child process to handle the computation on a new thread. However, audio recording under high load was still not perfect. Some of the AMT recordings had audio artifacts or were truncated, and these issues were more pronounced with long polling connections than with WebSocket connections, suggesting they may have resulted from timed out connections or a backed-up stream to the server.

4.3 Orcas Island: Mispronunciation Detection

The mispronunciation detection technology being built in the group has great potential in the field of language learning. To demonstrate its potential we built Orcas Island, an interactive speech-enabled web application with Spoke that uses this core technology on the backend to provide specific feedback on a user's pronunciation. This application uses almost all of Spoke's client-side and server-side modules, illustrating how the features of Spoke can be combined to create novel and complex demonstrations centered around a core technology. The user can record utterances and play back a whole utterance or hone in on just one word. Ultimately the application provides feedback about which words were mispronounced, and allows the user to compare her pronunciation of the word with that of a speech synthesizer.

Orcas Island

Improve your English pronunciation by reading short stories and getting feedback from Orca!

Short Stories

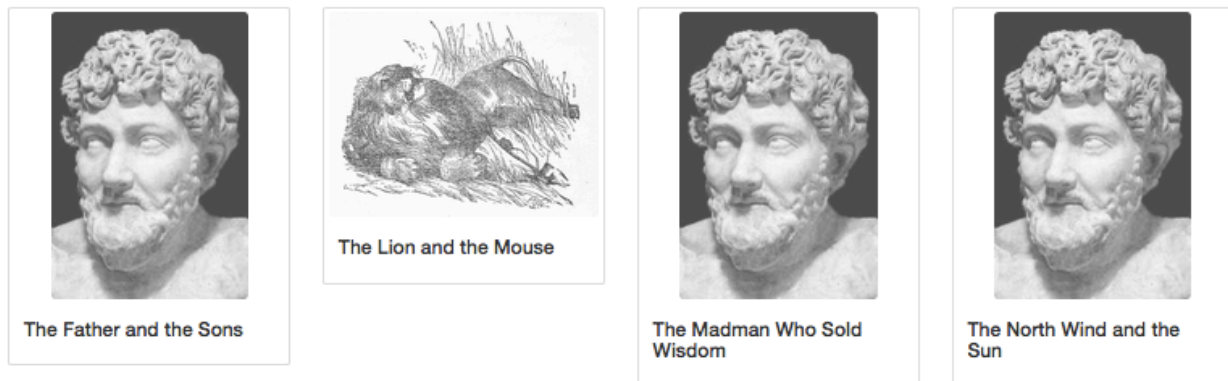


Figure 4-6: Orcas Island Homepage: *Orcas Island features a selection of short stories from Aesop's Fables. The homepage displays our current selection of stories and links to each one.*

The mispronunciation detection system looks for patterns of mispronunciation made by the same speaker over a set of utterances, so the application features a variety of short stories (from Aesop's Fables) broken into fragments for the user to read out loud. Each fragment is presented in a separate box with its own controls for recording and playback, allowing the user to focus on the story one sentence at a time. Each record button is hooked up to its own spoke-client Recorder instance configured with specific metadata about the fragment to send to the server along with the user's audio stream (e.g. the fragment number and the

fragment text). Each play button fires a Socket.io event requesting an audio stream of the user's utterance for that fragment to play back to the user.

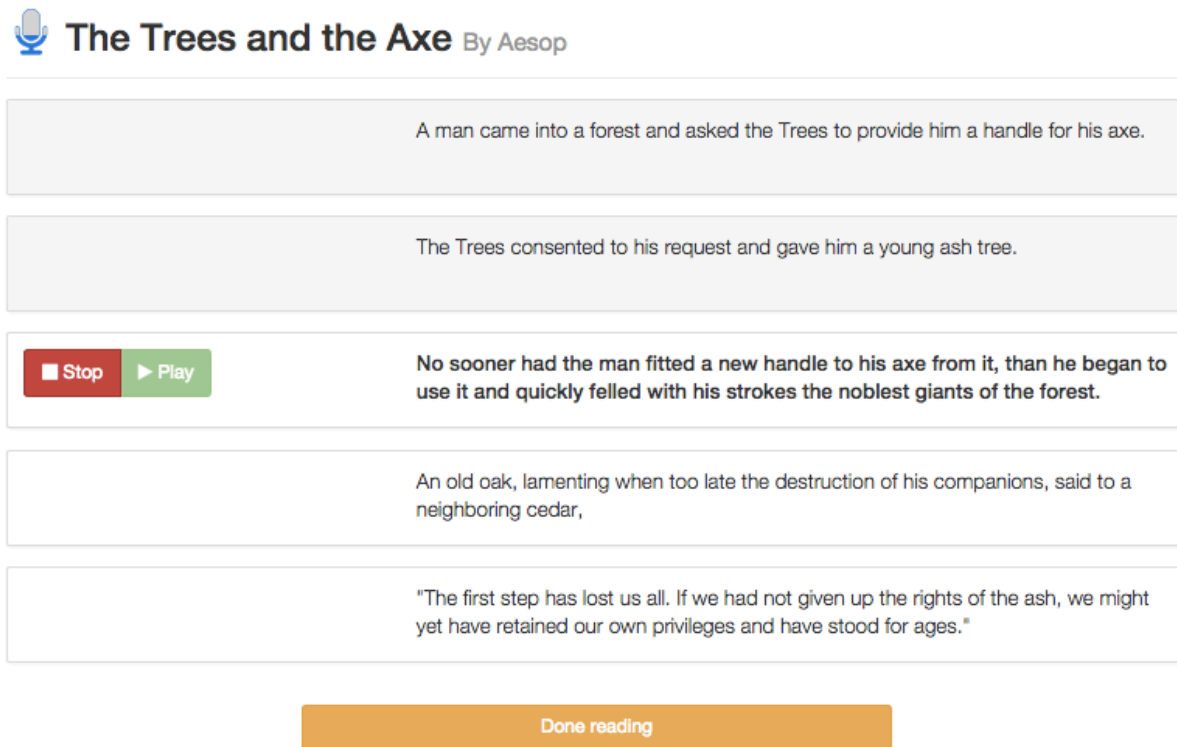


Figure 4-7: Reading a Short Story on Orcas Island: *The story is broken into small, readable fragments of one sentence or less. When the user records a sentence, the utterance is processed on the server and after a short delay is highlighted in light gray to indicate successful processing. When the user hovers over a fragment or is reading a fragment, its control buttons appear and the text is bolded to make it easier to read.*

```
1 Promise.join(recordingPromise, txtPromise)
2   .then(function () {
3     return mispro.forcedAlignmentAsync(wavFilename,
      txtFilename);
```

```

4     })
5     .then(function (timingFilename) {
6         return mispro.getAlignmentResultsAsync(timingFilename);
7     })
8     .then(function (utteranceTimingData) {
9         timingData[utteranceId] = utteranceTimingData;
10        socket.emit('success.spoke.alignment', metadata);
11    })
12    .then(function () {
13        return mispro.preprocessAsync(wavFilename);
14    })
15 });

```

Figure 4-8: Orcas Island Mispronunciation Processing Chain: *The server uses Promise chaining to create a sequential asynchronous processing pipeline. This pipeline first waits on the wav and txt files to be successfully saved, then performs forced alignment, gets the alignment results and saves them, notifies the client of success, and performs mispronunciation preprocessing.*

When the server receives an audio stream from the client, it begins transcoding the raw audio stream to a 16 kHz wav file with the Spoke Recorder and then runs the saved file through a processing pipeline comprised of Spoke’s integrated speech technologies and chained together with Promises. Given the saved wav file and the fragment text the user was reading, the server runs asynchronous forced alignment using Spoke’s Alignment

module. When this finishes, it uses the Alignment module to parse the timing results and present them in a JavaScript object. At this step the server notifies the client that playback for the recorded fragment is now available and goes on to the last stage of the pipeline, preprocessing for mispronunciation detection.

Mispronunciation Analysis

#	Word	Your Pronunciation	Correct Pronunciation
1	forest	forest	forest
2	noblest	noblest	noblest
3	quickly	quickly	quickly
4	gave	gave	gave
5	came	came	came
6	tree	tree	tree
7	trees	trees	trees
8	request	request	request
9	consented	consented	consented
10	than	than	than

Figure 4-9: Orcas Island Mispronunciation Analysis UI: *The results of mispronunciation detection are displayed in a table with the more significant mispronunciation patterns near the top. The client uses the Spoke Synthesizer to enable a comparison between the user’s pronunciation and the correct pronunciation of the word. The magnifying glass will highlight all instances of that word in the read fragments.*

Playback requests from the client are fulfilled with the Spoke Player on the server. If an entire utterance was requested, the Player simply creates a stream from the saved wav file for that utterance and the server sends that audio stream to the client where it is played with the spoke-client Player module. Requests for a single word of one utterance require

trimming the audio file with the server-side Player, using the timing results of the forced alignment step, before streaming.

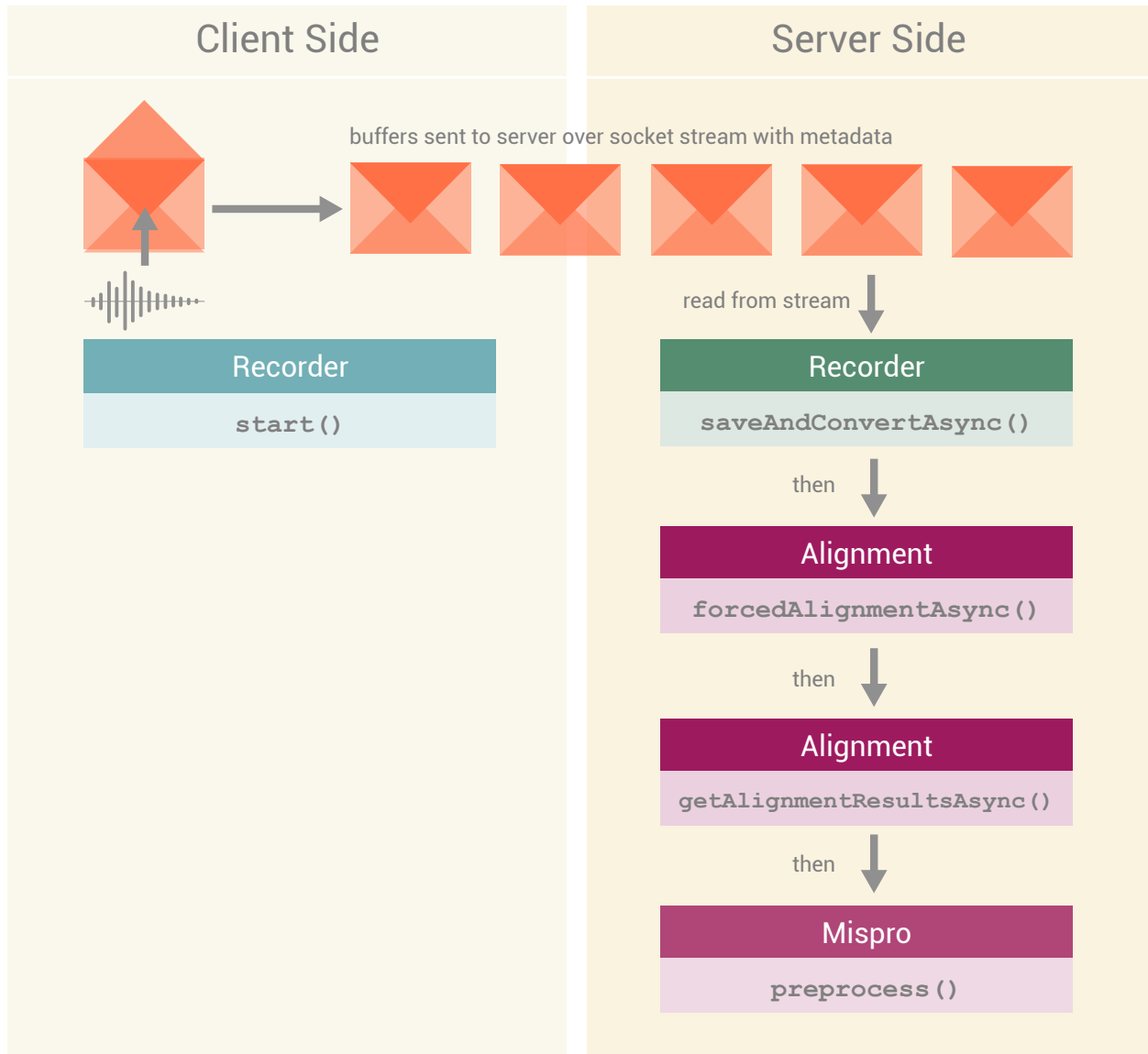


Figure 4-10: Orcas Island Recording and Processing Diagram: *This diagram illustrates how the spoke-client Recorder streams audio in buffers to the server, where the server-side Recorder converts the audio to a wav file and then passes it into the processing pipeline outlined in Figure 4-8.*

When the user is finished reading, the client notifies the server and it begins the final mispronunciation detection step, analyzing all the utterances as a set. Unlike the Promise-chained pipeline from earlier, this step forms a pipeline of streams. The Mispro module's `misproDetectionStream` method returns a stream for the output from the mispronunciation detection system, and `getMisproResultsStream` transforms the output stream into a stream of results. Each object in the results stream represents one mispronounced word, including additional information about the part of the word affected by the error and the location of one instance of the word in the recorded utterances. Passing along this information to the client, the user can see the mispronounced part of the word highlighted and can hear her own pronunciation of the word juxtaposed with the correct pronunciation generated by the Spoke Synthesizer module.

```
1 socket.on('doneReading', function () {
2     var misproOutput = mispro.misproDetectionStream();
3     var misproResultStream =
4         mispro.getMisproResultsStream(misproOutput);
5     misproResultStream.on('data', function (misproWord) {
6         socket.emit('result.spoke.mispro', misproWord);
7     });
8 });
```

Figure 4-11: Final Mispronunciation Detection Step With A Stream Pipeline: *Instead of using Promises, this method handles the final mispronunciation detection step by using Streams. The stdout output from the mispronunciation detection system is transformed into a Stream of objects representing mispronounced words.*

This application uses Spoke extensively to enable interactivity in the demonstration of the mispronunciation detection system. We showed how Spoke's modules could be combined in powerful ways to build novel features. For example, combining the Player's ability to trim audio with the Alignment's timing information, we implemented word-level playback from recorded utterances. Moreover, most methods of Spoke's server-side modules can be chained together with Promises to create a sequential, asynchronous processing pipeline.

Chapter 5

Future Work

Now that Spoke has bridged the gap between backend speech technologies and frontend web applications, we must consider areas for expansion and improvement.

5.1 Streaming Speech Recognition

With a continuous streaming speech recognizer, interactive speech-enabled web applications can be taken to the next level. This type of recognizer can operate on a stream of audio, outputting recognition hypotheses even as more audio comes in. As it receives more audio, the newest input may change the last few words of the best hypothesis, but over time

prefixes of the best partial hypothesis tend to stabilize [35]. The Web Speech API supports streaming recognition, and many developers have demonstrated its ability to make websites more responsive and engaging by providing more immediate feedback to the user. However, this browser API is currently only available in Chrome.

SLS seeks to build our own streaming speech recognizer in Kaldi to serve this purpose across multiple browsers and multiple devices. New speech technologies such as this can be integrated into Spoke through a command line interface, as has been done for non-streaming Kaldi recognizers already. A streaming recognizer could be integrated into Spoke in a very similar manner, except that instead of providing a saved wav file to the new child process executing the technology, it would set up the child process to take audio input from stdin and then pipe the user's audio stream onto stdin, much like the sox-audio SoxCommand does for streaming transcoding.

5.2 Reducing Bandwidth Usage and Dropped Audio

Though Spoke has proven its usefulness for audio recording from desktop browsers, it has not been thoroughly tested on mobile browsers. The initial tests on an Android Chrome browser produced barely intelligible audio with chunks of zeros and repeating waveforms that suggest some buffers were being duplicated and others were not getting read in time. Client-side downsampling could help reduce these occurrences by lightening the load on the client, but even so we will most likely need to move the recording processes to a background task using the Web Workers API [36]. Since these artifacts sometimes crop up even on desktop browsers, this is an important area for further investigation.

Another way we might reduce bandwidth is by transmitting FFT magnitudes instead of raw audio data. This could be achieved with an audio routing graph very similar

to that of the spoke-client VolumeMeter: a `MediaStreamAudioSourceNode` connected to an `AnalyserNode` connected to a `ScriptProcessorNode`. However, the configuration of the `AnalyserNode` would be substantially different. The VolumeMeter does not require high fidelity, so it can perform a small FFT over a larger chunk of audio data. If we plan to use the FFT output for backend processing, we will have to try a few sets of configuration parameters for the FFT size and the audio data buffer size to strike the right balance. The `ScriptProcessorNode` could easily be setup to transmit a buffer of FFT magnitudes to the server over Socket.io following the example in the spoke-client Recorder.

Chapter 6

Conclusion

As research progresses and new speech technologies are developed, it is crucial to be able to create tangible implementations and applications utilizing the advancements. However, without a convenient method for doing so, developers and researchers will be stuck re-inventing the wheel every time they wish to showcase a new technology or prepare an interface for data collection from users at scale. Spoke is a consolidated framework that allows a developer to quickly and simply take a speech-related technology and produce an application for almost any purpose, without having to tailor-make a solution from scratch.

Spoke leverages modern web technologies to enable the creation of speech-enabled websites. With an emphasis on modularity, the framework is flexible and powerful enough

to work with any spoken language system with a command line interface, while still allowing for a high level of customization by the developer. A client-side framework enables the creation of interactive UI elements, and a server-side library allows one to set up a web server to power the front end, or even use the methods in a standalone fashion to create batch-processing scripts.

In this thesis, three examples of complex speech-enabled websites created with the help of Spoke were explained, demonstrating how the various modules can be utilized to enable a multitude of applications with different goals. This included a nutrition-related site that used Spoke to help gather spoken information from users about their daily eating habits, a scalable Amazon Mechanical Turk application for large-scale data collection, and a mispronunciation detection website that provides automatic feedback to people seeking to learn to speak in a new language.

Because of the modularity inherent in the framework, it leaves a lot of room for further advancements to be made piece by piece, guaranteeing that developers that use Spoke will always have the latest tools available to them. Spoke will help greatly reduce the overhead and production time to bring a newly-developed speech-related technology into a real, working application. Ultimately we hope that this type of power in the hands of researchers and developers will go a long way in enabling a broader range of human-computer interaction through natural spoken language.

Bibliography

- [1] C. Cai, *Adapting Existing Games for Education Using Speech Recognition*, S. M. Thesis, MIT Department of Electrical Engineering and Computer Science, June 2013.
- [2] J. Liu, S. Cyphers, P. Pasupat, I. McGraw, and J. Glass, "A Conversational Movie Search System Based on Conditional Random Fields," Proc. Interspeech, Portland, Oregon, September 2012.
- [3] A. Lee and J. Glass, "Context-dependent Pronunciation Error Pattern Discovery with Limited Annotations," Proc. Interspeech, pp. 2877-2881, Singapore, September 2014.
- [4] A. Gruenstein, I. McGraw, and I. Badr, "The WAMI Toolkit for Developing, Deploying, and Evaluating Web-Accessible Multimodal Interfaces," Proc. ICMI, Chania, Crete, Greece, October 2008.
- [5] C. Varenhorst, *Making Speech Recognition Work on the Web*, M.Eng. thesis, MIT Department of Electrical Engineering and Computer Science, May 2011.
- [6] D. Huggins-Daines, et al. "Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices," Acoustics, Speech, and Signal Processing, 2006. ICASSP 2006 Proceedings.
- [7] <http://cmusphinx.sourceforge.net/2013/06/voice-enable-your-website-with-cmusphinx/>
- [8] <https://github.com/mattdiamond/Recorderjs>
- [9] <https://dvcs.w3.org/hg/speech-api/raw-file/9a0075d25326/speechapi.html>
- [10] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer and K. Vesely, "The Kaldi Speech Recognition Toolkit," in IEEE 2011 Workshop on Automatic Speech Recognition and Understanding, IEEE Signal Processing Society, 2011.
- [11] http://kaldi.sourceforge.net/online_programs.html
- [12] http://kaldi.sourceforge.net/online_decoding.html
- [13] <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction>
- [14] https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API
- [15] <https://nodejs.org/>

- [16] <https://github.com/substack/stream-handbook#introduction>
- [17] <http://docs.libuv.org/en/v1.x/>
- [18] <http://sox.sourceforge.net/>
- [19] <https://promisesaplus.com/>
- [20] <https://www.promisejs.org/>
- [21] <https://github.com/petkaantonov/bluebird>
- [22] <http://requirejs.org/docs/commonjs.html>
- [23] <http://requirejs.org/>
- [24] <http://requirejs.org/docs/optimization.html>
- [25] <https://github.com/jrburke/r.js>
- [26] <https://www.npmjs.com/>
- [27] <http://expressjs.com/>
- [28] <http://www.ractivejs.org/>
- [29] <http://socket.io/>
- [30] <https://developer.mozilla.org/en-US/docs/WebSockets>
- [31] <https://developer.mozilla.org/en-US/docs/Web/API/AudioNode>
- [32] M. Korpusik, *Spoken Language Understanding in a Nutrition Dialogue System*, M.S. thesis, MIT 2015
- [33] <http://aws.amazon.com/documentation/mturk/>
- [34] http://docs.aws.amazon.com/AWSMechTurk/latest/AWSMturkAPI/ApiReference_ExternalQuestionArticle.html
- [35] I. McGraw and A. Gruenstein, "Estimating word-stability during incremental speech recognition," 2011.
- [36] <https://github.com/fluent-ffmpeg/node-fluent-ffmpeg>