João Daniel da Silva Cardoso

Sistema de Reconhecimento de Fala via Web

Dissertação de Mestrado em Engenharia Eletrotécnica e de Computadores

Fevereiro de 2016

Universidade de Coimbra

Departamento de Engenharia Eletrotécnica e de Computadores

Faculdade de Ciências e Tecnologia

Universidade de Coimbra

Dissertação de Mestrado em Engenharia Eletrotécnica e de Computadores

Sistema de Reconhecimento de Fala via Web

João Daniel da Silva Cardoso

Desenvolvido com a Supervisão de

Professor Doutor Fernando Santos Perdigão

e coorientada por Jorge Proença

Júri:

Professora Doutora Teresa Martinez dos Santos Gomes (Presidente)

Professor Doutor Fernando Santos Perdigão (Orientador)

Professora Doutora Rita Cristina Girão Coelho da Silva (Vogal)

Professora Adjunta Carla Alexandra Calado Lopes (Vogal)

Setembro de 2015

Agradecimentos

Em primeiro lugar gostaria de agradecer ao meu orientador de dissertação, Prof. Dr. Fernando Santos Perdigão, pelo apoio prestado e a inteira disponibilidade ao longo da realização da dissertação. Também agradeço a colaboração e auxílio prestado pelo Jorge Proença. A motivação e exigência colocada, pelos dois, permitiu retirar o máximo proveito da dissertação e adquirir novas competências.

Agradeço à minha família. Sem eles não teria sido possível chegar a esta etapa. A motivação e suporte prestados foram essenciais ao longo de toda a minha vida.

Um agradecimento especial à minha namorada Liliana Almeida, por todo o amor, atenção e carinho proporcionado nos bons e maus momentos.

Por fim quero ainda agradecer a todos os meus amigos, com os quais partilhei muitos momentos de alegria e experiências académicas inesquecíveis.

Resumo

Com a especificação do HTML5 deu-se uma revolução das tecnologias *web*. A criação de novas APIs tem vindo a substituir, progressivamente, a utilização de *plugins*. Assim, a *web development* passou a estar em código aberto para a comunidade. O reconhecimento de fala também passa a estar disponível segundo as novas especificações do HTML5. Esta dissertação contribui com um sistema de reconhecimento de fala alternativo ao dos fabricantes de *browsers*.

O objetivo desta dissertação consiste na implementação de um demonstrador de reconhecimento de fala, através da web. O reconhecedor está integrado num servidor e este gere os acessos dos clientes, feitos através de páginas HTML. Para que tal aconteça, o cliente deverá enviar o áudio juntamente com a lista de possíveis palavras a reconhecer. O microfone é acedido através da MediaStream API, sendo analisado pela WebAudio API. Cada segmento de áudio gerado é enviado para o servidor, via *streaming*. Este utiliza a tecnologia Node.js, que por sua vez utiliza a arquitetura *event driven*, na gestão de pedidos.

O motor de reconhecimento de fala usado é o *Julius*, um *software* de domínio público. Os modelos acústicos, sistema de transcrição de letras para fonemas e dicionário fonético foram desenvolvidos previamente no laboratório onde ocorreu esta dissertação.

O sistema permite satisfazer pedidos de reconhecimento em simultâneo e oriundos de qualquer sítio da internet. Também é compatível com os principais *browsers* de internet. Para isso, basta que as páginas HTML sigam um protocolo simples de indicação de objetos que podem ter uma entrada por fala, além dos habituais dispositivos de entrada, rato e teclado.

Palavras Chave: Reconhecimento de fala, Julius Speech Recognizer, Web API, Node.js

Abstract

With the specification of HTML5 begun a revolution in web technologies. The creation of new APIs have been gradually replace the use of plugins. Thus, web development is open sourced to the community. Speech recognition is now available under the new specifications of HTML5. This work contributes with an alternative speech recognition system from browser makers.

The goal of this dissertation is to implement a demonstrator of speech recognition, over the web. The recognizer is part of a server and it manages clients' accesses made through HTML pages. To make it happen, the customer must send the audio along with the list of possible words to recognize. The microphone is accessed through the API MediaStream, being analyzed by the Web Audio API. Each generated audio segment is sent to the server, via streaming. This technology uses Node.js, which in turn uses the event driven architecture, to manage the requests.

The speech recognition engine Julius is an open-source software. The acoustic models, letters to phonemes transcription system and phonetic dictionary were previously developed in the laboratory where this dissertation occurred.

The system can satisfy simultaneous recognition requests anywhere from the internet. It is also compatible with all major web browsers. This is done by HTML pages if they follow a simple protocol indicating objects that can have an input from speech, besides the usual input devices, mouse and keyboard.

Keywords: Speech Recognition, Julius Speech Recognizer, Web API, Node.js

Índice

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

AM – Acoustic Model (Modelo Acústico)

API – Application Programming Interface (Interface de Programação de Aplicações)

HMM – Hidden-Markov Model (Modelos de Markov não observáveis)

HTK – Hidden Markov Model Toolkit

HTML – HyperText Markup Language

I/O – Input/Output

IP – Internet Protocol

LM – Language Model (Modelo de Linguagem)

MFCC – Mel-Frequency Cepstral Coefficients

MSAPI – Media Stream API

NPAPI – Netscape Plugin Application Programming Interface

NPM – Node Package Manager

PHP – PHP: Hypertext Preprocessor

RTC – Real Time Communication (Comunicação em Tempo Real)

TCP – Transmission Control Protocol

WAAPI – Web Audio API

WebRTC - Web Real Time Communication

Capítulo 1

Introdução

O reconhecimento de fala é uma tecnologia que permite ao computador transcrever o sinal acústico de fala humana em termos da sequência de palavras que foram proferidas [1].

Esta área tem permanecido um tema quente, nas últimas cinco décadas, sendo considerada uma tecnologia auxiliar importante na promoção de uma melhor comunicação Homem-Máquina. Contudo, nos seus primórdios, não era vista como uma forma fiável de comunicação. Isto porque a capacidade de processamento existente não era suficiente para fazer reconhecimento de fala em tempo real. Outros periféricos, como teclado e rato, ultrapassavam facilmente a eficácia comunicativa de um reconhecedor de fala [2].

Nos últimos anos, o reconhecimento de fala tem-se vindo a tornar, progressivamente, a primeira escolha na interação Homem-Máquina. Esta tendência deve-se ao progresso feito em algumas áreas chave. Primeiro, o poder computacional de hoje é muito superior ao que existia no início desta área (o que vai de encontro à Lei de Moor [2]). Na prática, passa a ser possível conceber reconhecedores de fala mais robustos, logo mais complexos (e exigentes computacionalmente). Em segundo lugar, graças às permanentes evoluções da Internet, é possível aceder a uma quantidade maior de dados. Passa a ser possível elaborar modelos (que servem de base aos reconhecedores) através de uma maior amostra de dados, que poderão ser adquiridos em cenário de uso real [2]. Isto traduz-se num acréscimo de qualidade, neste tipo de sistemas. Por último, *smartphones*, casas inteligentes, veículos inteligentes tornaram-se populares. Uma vez que interagir com estes sistemas através de um teclado é pouco conveniente, surge aqui uma procura de alternativas. A resposta a essa procura tem sido dada, através de sistemas de reconhecimento de fala, não fosse a fala, a forma mais natural de comunicar.

Com a revolução das tecnologias web, o reconhecimento de fala foi progressivamente integrado no seu meio. As duas tecnologias mais populares eram o VoiceXML e JAVA APPLETS [3]. Ambas viriam a ser abandonadas devido a problemas de latência e eficácia [3]. Neste momento, graças aos esforços produzidos pelo World Wide Web Consortium (W3C), está a ser desenvolvido um *standard* que permite reconhecimento de fala. Este *standard* encontra-se ainda numa fase embrionária, não sendo extensível a todos os *web browsers* [4].

A integração do reconhecimento de fala, na web, tem várias aplicações no mundo real. Esta pode melhorar drasticamente a forma como os utilizadores (especialmente os portadores de

deficiência), interagem com as páginas *web* [3]. Os utilizadores, no geral, passam a beneficiar de uma navegação mais cómoda e fluida. Submissão de formulários, alterações de visualização, botões, navegação, entre outros, passam a estar disponíveis através da voz.

1.1 Motivação

Esta dissertação decorre como a evolução natural do Projeto TICE.Healthy[1], no qual o IT e DEEC participaram com um sistema de reconhecimento de fala. Para melhor entender a necessidade de reformular o projeto anterior, é conveniente fazer uma pequena abordagem ao mesmo.

A participação do IT no Projeto TICE.Healthy teve como objetivo principal, o de possibilitar a navegação na plataforma web dedicada (WE.CAN) através da fala. Para que tal acontecesse, foi necessário desenvolver e adicionar módulos aos *browsers* de internet, de forma a ter acesso aos microfones e integrar um sistema de reconhecimento automático de fala. Estes permitiram a coerência de navegação, em páginas web, com os eventos de reconhecimento [5]. Para tal, foi desenvolvido um *plugin* para *browsers* de internet, que possibilita o reconhecimento de comandos no cliente, através de uma página web.

Para aceder ao microfone e/ou *webcam*, em browsers, a solução recorrente até então era utilizar as tecnologias Flash ou Silverlight. Estes *plugins*, que são instalados no SO, seriam responsáveis pela recolha de áudio e imagem. Com o passar do tempo, estas tecnologias provaram ter graves falhas de segurança para os seus utilizadores, comprometendo-lhes a sua privacidade. Paralelamente, no início de 2008, o W3C (consórcio de empresas de tecnologia que coordena os padrões da internet em relação à linguagem) anunciou a primeira especificação do HTML5, que tem sido desenvolvida até aos dias de hoje. Dentro de um leque de novidades que surgiu no HTML5, a mais relevante (para este caso) é a criação de novas API's (Interface de Programação de Aplicações) para áudio e vídeo. O desenvolvimento destas API's tem vindo a eliminar, progressivamente, a necessidade de *plugins* nas aplicações multimédia dos *browsers* [6]. A captação de áudio/vídeo viria também a ser resolvida no HTML5, através da MediaStream API. No entanto, não existiu uma versão estável que pudesse ser incluída no projeto Tice.Healthy, que terminou em 2013. À exceção do Google Chrome, que desenvolveu a Web Speech API, nenhum dos outros *browsers* seria capaz de oferecer uma implementação nativa de reconhecimento de fala [5] . Para colmatar esta deficiência, optou-se por criar um *plugin* para os principais *browsers* e em vários sistemas operativos. À exceção das versões superiores ao Internet Explorer superior a 5.5,

[1] LINK: http://tice.healthy.ipn.pt/

a maioria dos *browsers* era compatível com a API Netscape Plugin Application Programming Interface (NPAPI). Isto possibilitou o desenvolvimento de um *plugin*, capaz de funcionar em vários *browsers*.

No entanto, a solução acabou por se revelar efémera. As recomendações do HTML5 são de abandonar o uso de *plugins* em detrimento das API's desenvolvidas e estandardizadas pelo W3C [6]. Por consequência, a Google abandonou o suporte do NPAPI em 2014 [7] e a Mozilla planeia fazê-lo em 2016 [8]. Podemos concluir que o sistema se tornou obsoleto, num curto espaço de tempo.

Pelo descrito, a principal motivação foi revitalizar o sistema previamente desenvolvido, seguindo as recomendações do W3C. Serviu ainda como motivação, a premissa de que todos os desenvolvedores deverão ter a possibilidade de criar os seus próprios sistemas de reconhecimento.Com a consecução deste trabalho, criar-se-á um meio para que os utilizadores usufruam de um sistema de reconhecimento remoto, como alternativa às implementações disponibilizadas pelos fabricantes de *browsers*.

1.2 Objetivos

Na secção anterior ficou clara a intenção do W3C em abandonar o uso de *plugins* nos *browsers*. A tendência aponta para a integração de todas as funcionalidades necessárias ao cliente diretamente no *browser*, mais concretamente, através de API's.

O objetivo desta dissertação é, assim, o de desenvolver um sistema de reconhecimento de fala através da web. O reconhecedor deverá estar integrado num servidor e não no cliente. Este gere os acessos dos clientes, feitos através de páginas HTML. Para que tal aconteça, o cliente deverá enviar o áudio juntamente com a lista de palavras a reconhecer. O acesso e algum processamento de áudio deve ser feito no cliente, através das API's disponibilizadas pelo W3C. Cada segmento de áudio gerado é enviado para o servidor, via *streaming*. Este utiliza a tecnologia Node.js, mais concretamente o *addon BinaryJS*, permitindo o *streaming* dos segmentos de áudio e gerir os pedidos e processos de reconhecimento de fala em simultâneo.

O motor de reconhecimento de fala que foi utilizado é o Julius [9], que é um *software* de domínio público. Os modelos acústicos, sistema de transcrição de letras para fonemas e dicionário fonético foram desenvolvidos previamente, no âmbito do projeto Tice.Healthy. A implementação efetuada no projeto Tice.Healthy foi aproveitada, reformulada e integrada no servidor Node.js. A

integração é feita através de um *addon*, sendo escrito em código nativo através das bibliotecas: V8, libuv e biblioteca nativas do Node.js.

O sistema deverá satisfazer pedidos de reconhecimento em simultâneo e oriundos de qualquer sítio da internet. Também deverá de ser compatível com os principais *browsers* de internet [10].

1.3 Estado da Arte

A Web Speech API tem o intuito de oferecer aos desenvolvedores ferramentas de reconhecimento de fala e síntese de voz [11]. Na vertente do reconhecimento, a API está projetada para possibilitar reconhecimento tanto de comandos, como de fala contínua. Os resultados de reconhecimento de voz serão fornecidos para a página web, através de uma lista de hipóteses, juntamente com outras informações relevantes para cada hipótese (por exemplo, o índice de confiança)[11].

Contudo, à data da dissertação, esta solução não satisfaz alguns dos objetivos propostos para este projeto.

Primeiro, há que assinalar que esta API apenas é parcialmente suportada num único *browser* - o Chrome. Ainda assim, um dos recursos que ainda não está disponível é, justamente, a possibilidade de construir uma lista de palavras a reconhecer a cada instante [12]. Apesar de esta funcionalidade estar previsto pela estandardização do W3C, ainda não foi implementada. Em segundo lugar, não há qualquer tipo de suporte por parte do *browser* Firefox. Neste momento, a Mozilla apenas dá suporte para o Firefox OS e Firefox Nightly Edition (edição especial para desenvolvedores) [13]. A intenção de criar uma solução versátil (para os dois gigantes de browsers) e através da Web Speech API, cai por terra.

1.4 Organização da Dissertação

Esta dissertação está organizada em 7 capítulos. O Capítulo 2 é dedicado à explicação das tecnologias utilizadas. Neste, começaremos por abordar a captação e processamento de áudio no cliente. Esta temática divide-se em dois grandes temas: MediaStream API e Web Audio API. Seguidamente, será feita uma abordagem geral à tecnologia utilizada no servidor. Esta designa-se por Node.js, sendo dado foco às questões de programação *event driven* e utilização de *closures*. Finalmente, serão abordadas as questões relacionadas com o reconhecimento de voz. Nestas destacam-se a modelação acústica com Modelos de Markov não Observáveis e o *software Julius*.

O Capítulo 3 serve de guião às rotinas criadas para o cliente. Serão expostos os processos utilizados na captação e processamento de áudio e implementações que visam a melhoria global do sistema de reconhecimento.

No Capítulo 4 serão debatidas as questões referentes ao servidor. Aproveitando os conceitos teóricos, alicerçados no Capítulo 2, será explicada como a arquitetura por eventos é implementada no servidor. Questões referentes ao controlo de acesso ao servidor e escalabilidade do serviço serão abordadas neste capítulo.

O Capítulo 5 serve para expor a forma como o reconhecedor é integrado no servidor. Neste capítulo, começaremos por debater as principais limitações do *Julius* e forma como estas influenciaram a integração com o servidor. Posteriormente, é exposta a API criada para a integração do *Julius* com o servidor. Finalmente, serão expostas as principais modificações efetuados no *Julius* e as implementações que visam a melhoria global do sistema de reconhecimento.

Para concluir, seguem-se os Capítulos 6 e 7. No primeiro será apresentado um possível esboço do resultado final, juntamente com considerações pertinentes. No segundo, serão expostas as principais conclusões, acompanhadas de sugestões de implementações futuras.

Capítulo 2

Sistema de Reconhecimento de Fala via Web

2.1 Captura e Processamento de Áudio no Cliente

Um dos últimos grandes desafios na web foi o de implementar comunicação em tempo real (RTC, Real Time Communication) de vídeo e voz. A RTC tem sido dominada pelas companhias detentoras dessas tecnologias, através do uso de *plugins*, tais como: o Silverlight (pela Windows) e o Flash Player (pela Adobe). Consequentemente, integrar estas tecnologias com outras funcionalidades é particularmente difícil, uma vez que elas não se encontram em licença de código aberto [14][15].

Numa procura de alternativas e no resultado da primeira especificação do HTML5, viria a nascer em Maio de 2011 a Web Real Time Communication (WebRTC). Os princípios base da WebRTC são: manter as suas API's em código aberto, gratuitas, estandardizadas e embebidas nos *web-browsers* [16]. Muitos dos dispositivos que usamos diariamente, como *smartphones, tablets, smart TV's* e computadores pessoais estão conectados à internet e com a WebRTC, todos estes dispositivos terão a capacidade de partilhar voz, vídeo e dados em tempo real através de uma plataforma comum. Para que tal aconteça, a WebRTC criou três API's: RTCPeerConnection, RTCDataChannel e MediaStream [16].

Nesta Secção, faremos uma abordagem teórica relativa à MediaStram API (MSAPI), uma vez que foi através dela que se criou o acesso ao microfone do cliente. Posteriormente, explicar-se-ão os pressupostos da Web Audio API, enquanto aplicativo que permite o acesso às amostras de som capturadas pela MSAPI.

2.1.1 Media Stream API

A Media Stream API (MSAPI) está projetada para permitir ao *browser* o acesso a dispositivos de entrada, tais como câmaras e microfones. Esta API oferece meios de controlar onde os dados são adquiridos/consumidos e dá a possibilidade de controlar variáveis relacionadas com a aquisição em si mesma (por exemplo, a resolução da *webcam*, *framerate*, etc.). No entanto, há que referir que ainda não se encontra totalmente suportada pelo *browser* Chrome. Para que funcione corretamente, terá de ser invocado, uma interface de adaptação entre a versão antiga da MSAPI e

aquela que é a recomendada. Esta interface, "adapter.js", também é válida para as versões do Firefox mais antigas[17].

As duas componentes principais da MSAPI são as interfaces *MediaStreamTrack* e *MediaStream* (Figura 2.1). O objeto *MediaStreamTrack* representa conteúdo *media* de um determinado tipo, consoante o dispositivo de entrada que lhe dá origem, ou seja, representa vídeo se for capturado pela *webcam* e áudio se for capturado pelo microfone. Por sua vez, o objeto *MediaStream* é utilizado para agrupar os diferentes *MediaStreamTrack,* num só [18].

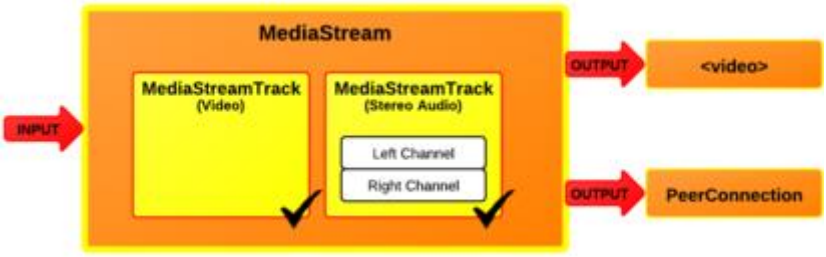

Figura 2.1 – Objeto MediaStream resultante de dois MediaStramTrack [14]

Cada *MediaStreamTrack* pode representar múltiplos canais (como por exemplo, o canal esquerdo/direito de áudio ou vídeo estereoscópico). No entanto, o seu acesso apenas é possível através de outras API's, como a WAAPI[2] (como se verá mais à frente) [18].

O objeto *MediaStream* tem uma entrada, que como já foi dito, representa a combinação de todos os *MediaStreamTracks* que lhe dão origem. Já o tipo da sua saída determina a forma como este objeto é renderizado, ou seja, se uma saída for para o elemento *vídeo* (do HTML5) terá menores exigências de qualidade, do que se tratar de um *download*. Assim, os processos de renderização serão necessariamente diferentes [18].

Os objetos anteriormente referidos só passam a ter valor se for implementado o método *MediaDevices.GetUserMedia()*. Este método solicita permissão ao utilizador para aceder à câmara e/ou microfone, tal como está indicado na Figura 2.2. Se a permissão for aceite por parte do utilizador, será devolvido o objeto *MediaStream* resultante. Este objeto será composto das *MediaStreamTrack* resultantes das permissões de acesso concedidas. Se o utilizador optar por negar o acesso aos seus dispositivos multimédia, será devolvida a mensagem de erro consequente.

[2] Web Audio API

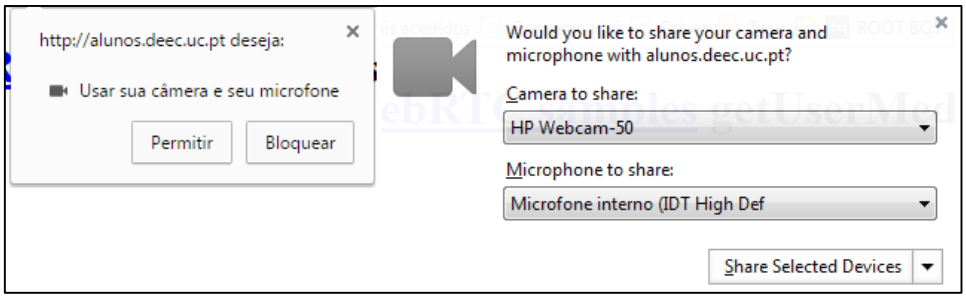

Figura 2.2 – Pedidos de permissão de acesso dos Browsers

Após o pedido de permissão, o utilizador consegue verificar, em qualquer altura, se o *browser* está a aceder aos dados dos seus dispositivos. Os *browsers* têm esta ferramenta de alerta, por definição. Esta pode ser verificada na Figura 2.3.

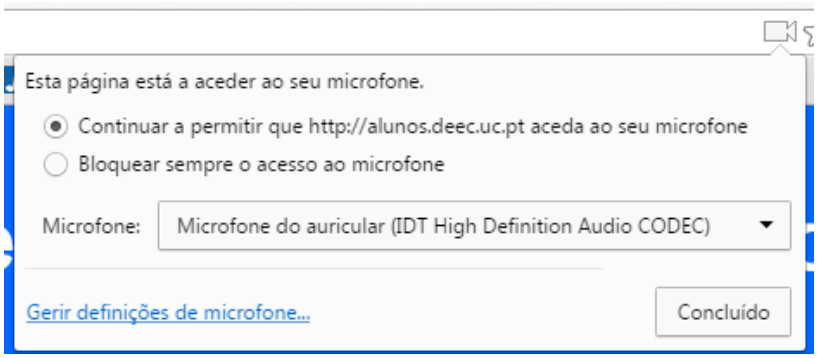

Figura 2.3 – Aviso de acesso ao Microfone

2.1.2 Web Audio API

Antes da existência do elemento HTML5 <*audio*> era necessário utilizar *plugins* para quebrar o silêncio dos *browsers*. Contudo, apesar deste avanço, o elemento <*audio*> continua a ser limitado, se for necessário criar aplicações mais complexas [19]. Desta necessidade nasce a Web Audio API (WAAPI), tornando-se possível o processamento e sintetização do áudio em aplicações web. Esta tecnologia já se encontra numa fase amadurecida, sendo suportada por todos os *browsers* de internet [20].

A WAAPI está construída em torno do conceito contexto de áudio. O contexto de áudio é compreendido pela representação gráfica dos vários nodos de interligação desde a fonte (microfone) até ao destino. À medida que o áudio flui através de cada nó as suas propriedades são alteradas/lidas, de acordo com as características do nó correspondente [21]. Consequentemente, a complexidade do contexto de áudio é proporcional à quantidade de nós e conexões existentes. Graficamente, a inicialização do contexto de áudio equivale a criar uma tela em branco, na qual podemos declarar e conectar os nodos pertinentes.

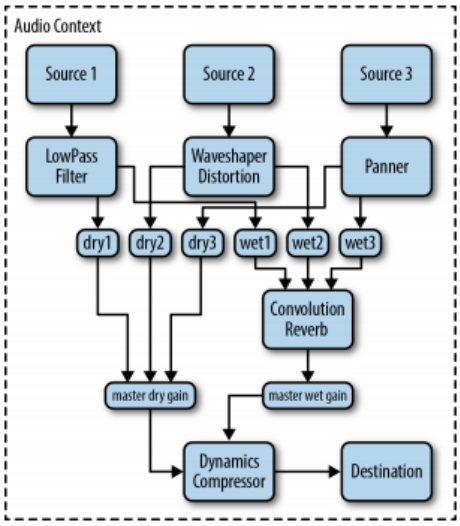

Figura 2.4 – Exemplo de um Contexto de Áudio [21]

A Figura 2.4 mostra um contexto de áudio complexo, onde está representado uma grande quantidade de nodos. É importante referir que estes se dividem em quatro categorias [21]:

- *Source Nodes* – Nós que atuam como fontes de áudio. Podendo ser buffers de áudio, a *tag* de <*audio*>, osciladores e entradas de áudio ao vivo (microfone, acedido pela MSAPI).
- *Modification Nodes* – Nós onde o áudio é alterado nas suas propriedades. Estes podem representam filtros, *JS Processors*, *convolvers*, *panners*, etc.
- *Analysis Nodes* – Nós onde as propriedades do áudio são lidas. Tipicamente usadas para representação gráfica do áudio. Estes podem ser *Analysers* ou *JS Processors*.
- *Destination Nodes* – Nós que representam a saída de áudio.

Isoladamente os nodos não têm qualquer efeito sobre o áudio (tanto a nível de leitura, como de modificação). Como se pode verificar na Fig.2.4 eles devem estar ligados. Para que tal aconteça, deve ser utilizado o método *connect()*. Inversamente a isto, a WAAPI também prevê o método *disconnect()*, o que permite que o contexto de áudio seja mudado dinamicamente[21].

2.2 Servidor

A maioria das aplicações web têm uma implementação do lado do cliente (*front end)* e do lado do servidor (*back end*) [22]. A implementação do *back end* é, tradicionalmente, bastante complexa. De modo a criar um simples servidor eram necessários vastos conhecimentos de *multithreading* e escalabilidade. Outra complicação, para o desenvolvedor, é o facto do *front end* estar implementado através das linguagens HTML e JavaScript, enquanto o *back end* estar

implementado noutras linguagens, como PHP, C#, .NET, JAVA, Ruby, entre outros. Esta diferença obriga o desenvolvedor a ter de dominar várias linguagens de programação [23].

Há alguns anos era impensável implementar um servidor em JavaScript. Esta linguagem tinha um desempenho muito baixo, gestão de memória pouco eficiente e falta de integração com o sistema operativo. Todas estas falhas deveriam ser corrigidas para que o JavaScript fosse considerado uma solução viável na implementação de servidores [23]. A solução para os dois primeiros problemas veio a ser resolvida pela Google, em Setembro de 2008. Nasce assim a V8: uma máquina JavaScript, de alta performance, escrita em C++ e de código aberto [24].

Figura 2.5 – Logotipo Node.js

Ryan Dahl viu a oportunidade em trazer o JavaScript para o *back end* e resolveu o último problema: embutiu a V8 numa camada capaz de integrar a V8 com o Sistema Operativo. Em Maio de 2009, nasce o Node.js (Figura 2.5) e o sucesso foi instantâneo. De tal maneira que grandes empresas como IBM, Microsoft, Yahoo, Paypal, LinkedIn o utilizam [25]. A comunidade de desenvolvedores cresceu exponencialmente, contribuindo num total de mais de 21000 extensões (à data). Concluindo, com NodeJS passou a ser possível aos *web devellopers* utilizar a mesma linguagem no *front end* e *back end* [23].

2.2.1 *Node.js*

O Node.js é um *software* multiplataforma e em código aberto, para o desenvolvimento de aplicações de servidor. As suas aplicações estão escritas em JavaScript e podem ser executadas em OS X, Microsoft Windows, Linux, entre outras. O Node.js oferece *event-driven architeture* e uma *non-blocking I/O* [26] *API*, projetadas para otimizar a escalabilidade de aplicações *web* em tempo real [25].

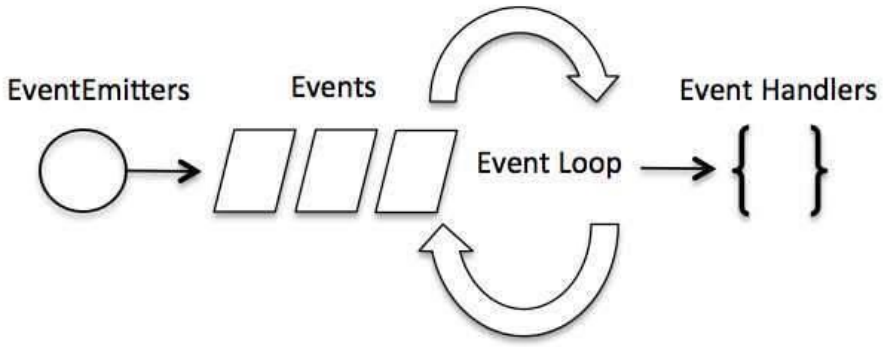

Figura 2.6 – Diagrama da programação Event-Driven [27]

Na programação *event-driven*, a execução das rotinas é determinada por eventos que são tratados por *event callbacks*, também designados por *event handlers*. Desta forma, um *event callback* é uma função que é invocada assim que o evento que lhe dá origem é desencadeado [28] (Figura 2.6). Na prática, isto significa que, no Node.js, uma rotina não causará bloqueio (de todo o programa) ao fazer operações I/O. Portanto, várias operações I/O podem ocorrer em paralelo, e o respetivo *event callback* será invocado assim que a operação termine [28]. Para que tal aconteça, esta técnica de programação necessita de um *event loop*. O *event loop* é um mecanismo que realiza duas tarefas num ciclo contínuo: deteção de eventos e desencadeamento de *event handlers*. Em qualquer execução do ciclo, este mecanismo deverá detetar quais os eventos desencadeados, seguidamente deverá determinar o *event handler* correspondente e invocá-lo [28]. O *event loop* é apenas uma *thread* a correr dentro de um processo, pelo que podemos tirar duas ilações: apenas um *event handler* estará a correr a uma determinada altura e qualquer *event handler* será executado até ao fim, sem que seja interrompido [28]. Isto torna-se vantajoso para o desenvolvedor que desta maneira não necessita de se preocupar com *threads* a serem executadas em simultâneo, que podem alterar o estado da memória partilhada [28] (Figura 2.7).

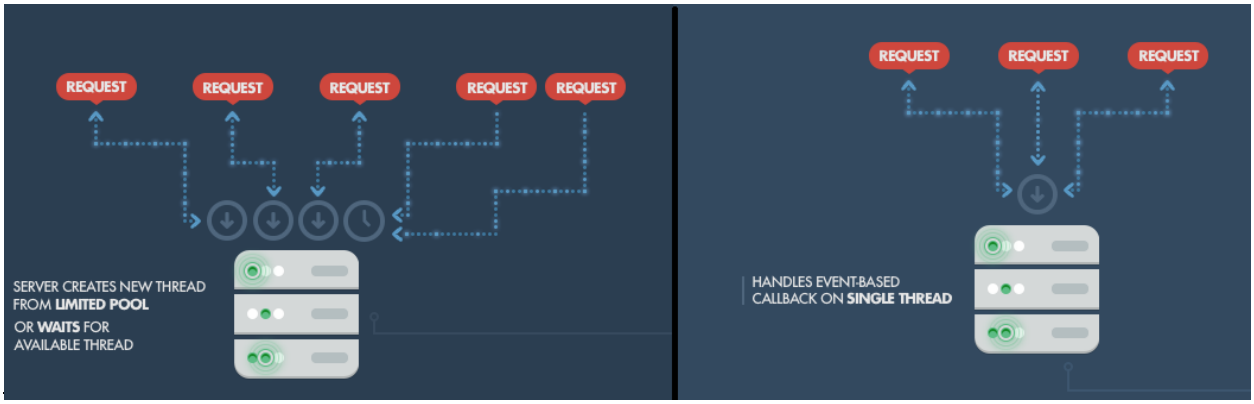

Figura 2.7 – Acesso ao Servidor (à esquerda a via tradicional e à direita através do Node.js) [29]

Outra técnica de grande utilidade (e herdada do JavaScript) é a utilização de *closures*. As *closures* são funções que herdam variáveis do seu ambiente envolvente (*enclosing environment*) [28]. O ambiente envolvente é definido por todas as variáveis globais, variáveis declaradas no próprio alcance e variáveis declaradas no alcance de funções que lhe são pai [30][28]. Quando uma função *callback* é passada como argumento de outra função, será chamada por esta a determinada altura, para ser executada. Ao ser executada, esta função terá a particularidade de se lembrar do contexto em que foi chamada, sendo capaz de aceder e manipular todas as variáveis alocadas no seu ambiente envolvente [30]. Por esta razão uma função *callback* é considerada, na prática, uma *closure* [31]. A situação inversa poderá não se verificar.

Posto isto, na programação *event-driven* começa-se por definir que código será executado para um determinado evento, sob a forma de *event handlers* e o sistema encarregar-se-á de as invocar, assim que o respetivo evento seja desencadeado [28]. Ao passarmos a função *callback* como argumento é expectável que esta seja executada mais tarde, sendo capaz de ler e manipular qualquer variável do seu ambiente envolvente. Isto acontece porque, como já vimos, se trata de uma *closure*. Ao utilizar esta técnica consegue-se implementar a programação *event-driven* sem que a função *callback* perca o contexto na qual foi executada [28]. Desta maneira deixa de ser necessário andar a passar variáveis entre funções e guardar o seu estado.

Figura 2.8 – Logotipo do Node Package Manager

Para uma melhor organização do código o Node.js dispõe as suas funcionalidades em módulos. Estes módulos deverão ser invocados através do método *require()* e no início do script (tal como analogamente o C/C++ faz o *include*). Ao fazer *require()* de um módulo, será retornado um objeto que representa a API JavaScript, exposta por esse módulo [28]. Oficialmente existem dois tipos de módulos. Em primeiro lugar, aqueles que já fazem parte do *core* Node.js. Em segundo, os que estão disponíveis no Node Package Manager (NPM, Figura 2.8). Este programa representa o repositório oficial de módulos desenvolvidos por terceiros, permitindo a sua instalação no servidor [32].

2.2.2 *BinaryJS*

No âmbito desta dissertação, houve um módulo que se tornou bastante relevante, estando disponível no NPM: o *BinaryJS*. O *BinaryJS* é definido como um módulo de comunicação bidirecional de dados binários em tempo real, através de *websockets* [33]. Este módulo foi desenvolvido pela comunidade e tem como base as especificações ilustradas na WebSocket API. Deste modo torna-se possível o uso desta API, através de uma interface mais simples e intuitiva. Este módulo cria uma ligação TCP entre servidor e cliente, permitindo também que um servidor se comporte como cliente (perante um servidor pai). O seu funcionamento geral obedece aos pressupostos apresentados na Secção 2.2.1, ou seja, gira em torno da programação *event-driven*, portanto, são os desenvolvedores que têm de definir os *event callbacks* pertinentes de cada evento.

Na Tabela 2.1 estão referenciadas as classes do *BinaryJS* existentes, respetivas propriedades e métodos. Nela podemos perceber que há uma implementação ao nível da página do cliente e outra ao nível do servidor. No fundo, o *BinaryJS* pega na Web Socket API de modo a criar um canal de comunicação entre a página web e o servidor Node.js.

Tabela 2.1- Referência da BinaryJS API, adaptado de [34]

Classe: binaryjs.BinaryServer [Apenas Node.js]	**Classe: binaryjs.BinaryClient** [Node.js e browsers]	**Classe: binaryjs.BinaryStream** [Node.js e browsers]
Construtor: *new binaryjs.BinaryServer([options],[call back])*	Construtor: *new binaryjs.BinaryClient(address, [options])*	*stream.pause()* - Pausar o envio de dados
Evento: "error" - Quando existe erro na ligação	*client.createStream([meta])* - Cria uma nova stream, partilhando o campo meta	*stream.resume()* - Prosseguir o envio de dados
Evento: "connection" - Quando um cliente se liga	*client.close()* - Encerra a ligação	*stream.end()* - Acabar o envio de dados
Server.clients() – Informação de todos os clientes ligados num dado instante.	Evento: "*open*" - Quando é estabelecida a ligação	*stream.write(data)* - Enviar dados
	Evento: "*stream*" - Quando é criado um novo canal de *stream*	Evento: "*data*" - Quando um Buffer de dados chegou
	Evento: "*error*" - Quando existe mensagens de erro	Evento: "*pause*" - Quando foi utilizado *stream.pause()* remotamente
	Evento: "*close*" - Quando a ligação é fechada	Evento: "*error*" - Quando é emitido mensagem de erro

A sintaxe utilizada no *BinaryJS* está, genericamente, representada na Tabela 2.2. Nela devemos especificar o objeto que emite determinado evento. Esse objeto designa-se por *event emitter*. Associado a este deverá estar o nome do evento. Só desta maneira o *event loop* reconhece qual o evento desencadeado. Finalmente, o *event handler* é a função *callback* invocada para determinado evento.

Tabela 2.2- Sintaxe do BinaryJS na programação event-driven

```
eventEmitter.on('eventName', eventHandler);
```

2.2.2 *Node.js Addon*

Como foi abordado anteriormente, o servidor Node.js apenas faz operações I/O [26] sem bloqueio. O resto ocorre em apenas uma *thread*. Isto poderá ser um transtorno, caso essas rotinas sejam computacionalmente pesadas, pois provoca bloqueio do *event-loop* e consequentemente do servidor. Este tipo de rotina designa-se por síncrona, já que está a ser executada no *event loop*. Rotinas síncronas de longa duração devem ser evitadas uma vez que só quando finalizadas permitem ao servidor executar novas operações.

Uma vez que o *Node.js* é baseado no motor JavaScript V8, que está escrito em C/C++, é possível adicionar código nativo ao servidor, através do uso de *addons* [35]. Um *addon* é, no fundo, um módulo para o servidor *Node.js* escrito em código nativo. Apesar de ser uma abordagem mais complexa é possível ganhar mais flexibilidade e desempenho [35]. Isto porque a linguagem C/C++ é reconhecidamente a mais rápida e é possível invocar as rotinas assincronamente. Este último especto é muito importante, uma que vez ao tratar rotinas mais complexas assincronamente, evita-se o bloqueio do servidor [36].

Construir um *addon* implica conhecimento bibliotecas de programas, tais como as que se seguem [37]:

- *V8 C++* - usada para servir de interface entre a linguagem C/C++ e Javascript. Toda a sua documentação e código encontram-se em aberto para a comunidade
- *Libuv* - esta libraria serve para fazer operações I/O [26] (o código está aberto à comunidade). É através dela que se invoca o *addon* assincronamente.
- Bibliotecas internas do Node.js - uma das mais importantes, para o projeto, foi a *node_buffer*. Esta permite a passagem de dados binários entre o JavaScript e o C/C++.

Uma vez que o *addon* é apenas um módulo, escrito em código nativo, a sua invocação é feita pelo método *require()*, sendo que este devolverá todos os métodos definidos (pelo *addon*) [28].

Para o desenvolvimento do *addon* utilizou-se a ferramenta Microsoft Visual Studio 2013. Esta trás uma versão recente do compilador, o que permite obter as características do *standard* C++2011. No contexto desta dissertação, o C++2011 foi relevante para o uso da *std::thread* (explicado no Capítulo 5). Para efetuar *debug* do *addon* compilou-se o código fonte do Node.js. A

compilação é feita de forma a obter os símbolos de *debug* no executável resultante [38]. Desta forma, podemos integrar o executável nas opções de *debug* do Visual Studio 2013 [38].

2.3 Reconhecimento de Fala

Os sistemas de reconhecimento de fala têm tipicamente quatro componentes, tal como representado na Figura 2.9 e que consistem em:

- Processamento de sinal e extração de características,
- Modelo Acústico (AM),
- Modelo de Linguagem (LM) e a
- Descodificação ou busca de hipóteses [2].

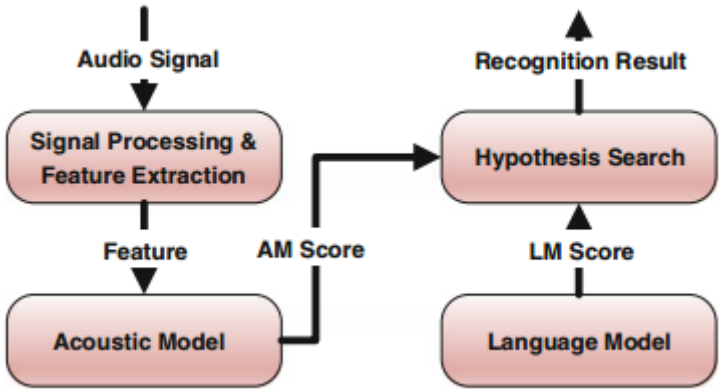

Figura 2.9 – Arquitetura geral de um Sistema de Reconhecimento [2]

O processamento de sinal e extração de características tem como entrada o sinal de fala que será convertido do domínio do tempo para o domínio da frequência. Neste processo são extraídos os vetores adequados ao modelo acústico (AM) utilizado [2]. O AM integra a relação entre a acústica e a fonética. Por outras palavras, estabelece a relação entre o sinal de áudio e os fonemas correspondentes (ou outras unidades linguísticas de fala utilizadas). O AM toma a sequência de vetores de características geradas de forma a calcular a probabilidade da sequência de fonemas para aquele sinal acústico. O modelo de linguagem (LM) estima a probabilidade de uma possível sequência de palavras. O LM oferece um contexto, de forma a distinguir palavras e frases que soem familiarmente [39]. Finalmente, a busca de hipóteses combina os resultados devolvidos pelo AM e LM, devolvendo a sequência com maior pontuação. Sendo O a sequência de observações acústicas e W a sequência de palavras embebidas no sinal de fala, o reconhecimento de fala calcula a probabilidade da sequência de palavras W ter sido proferida sabendo que O é o sinal observado e segundo o modelo acústico M. Pelo teorema de Bayes,

$$\Pr(W|O,M) = \frac{\Pr(O|M)\Pr(W)}{\Pr(\mathrm{O})}$$

Nesta equação $\Pr(O|M)$ é a probabilidade calculada pelo modelo acústico e $\Pr(W)$ a probabilidade calculada pelo modelo de linguagem. A descodificação consiste em obter o valor máximo do primeiro termo da equação, ou seja, a sequência de palavras mais verosímil, pelo que o denominador do segundo termo não varia e é ignorado neste cálculo. No caso de reconhecimento de comandos, o modelo de linguagem é mais simples, assumindo-se usualmente que os comandos são equiprováveis.

2.3.1 Parametrização do Sinal de Fala

Como se verifica na Figura 2.8, o primeiro passo para efetuar reconhecimento é a extração das características do sinal de voz. Para isso são calculados os Mel Frequency Cepstral Coefficients (MFCCs). Tal como o nome indica, estes parâmetros têm como base a escala de Mel. Esta é uma escala melódica que simula o funcionamento do sistema auditivo. Para isso, ela tem em conta que o ouvido humano é mais sensível a mudanças de tom a baixas frequências [40]. A parametrização foi introduzida na década 1980 e é uma das mais aplicadas, ao sinal de fala [40]. A sua implementação obedece aos passos expostos na Tabela 2.3.

Tabela 2.3- Parametrização MFCC, adptado de [40]

1) Agrupar o áudio em janelas de 25ms, com espaçamento de 10ms. 2) Cálculo dos coeficientes de magnitude para cada janela, através da DFT. 3) Multiplicação, de cada janela, por um banco de filtros (12). Estes têm a frequência central distribuída ao longo da escala de mel. Somar os valores obtidos à saída de cada filtro. Esta soma define o comportamento espetral da banda correspondente, para cada janela. 4) Calcular o logaritmo de cada soma resultante, em 3). 5) Calcular os parâmetros cepstrais dos valores resultantes em 4), através da DCT (Discrete Cosine Transform).

Cada janela resultará num vetor de características, um vetor de coeficientes MFCC que vai constituir numa observação do modelo acústico. Este é composto por 12 parâmetros, resultantes do cálculo da DCT, mais um resultante do logaritmo da energia da janela, logo existem 13 parâmetros (ditos estáticos) que definem um segmento de sinal de fala [41]. Contudo, pode haver pequenas variações neste processo. É comum juntar a estes 13 parâmetros os chamados coeficientes dinâmicos, também conhecidos por parâmetros delta. A partir destes parâmetros, ainda são adicionados os parâmetros delta-delta, também conhecidos por parâmetro de aceleração. No total, são 39 parâmetros a serem interpretados na componente AM, da Figura 2.9.

2.3.1 Modelação Acústica do Sinal de Fala

Um dos principais problemas, ao lidar com a componente acústica (ver Figura 2.9) é a variabilidade dos vetores MFCCs [2]. A variabilidade está relacionada com as características do sinal de fala e do canal - microfone, questões de dicção, sotaque, ambiente de captação entre outros. Esta variabilidade é usualmente modelada estatisticamente através dos Modelos de Markov Não Observáveis (HMM, "Hydden Markov Models") [2].

A introdução dos HMMs e métodos estatísticos associados ocorreu em meados da década de 1970 [2]. Todavia, apenas foi disseminada em meados da década de 1980 [42], tornando-se a maior mudança de paradigma, na área da modelação acústica [2]. Esta solução tem sido, desde então, uma das principais referências do estado da arte [42]. Desde 2010, soluções baseadas em redes neuronais começaram a mostrar maior precisão nos resultados apresentados [2]. Contudo, os avanços obtidos são mais relevantes para tarefas de reconhecimento contínuo, onde teoricamente se pode dizer qualquer palavra da língua, o que implica uma computação exaustiva [43]. No caso desta dissertação, a utilização de HMMs consiste na opção disponível para demonstrar o reconhecimento via web.

O HMM descreve um fonema da língua, através de uma cadeia de Markov que representa uma sequência de vários estados. Cada estado tem associada probabilidade de transitar para outro, ou de manter-se no mesmo [44][45]. Associado a cada estado existe uma função densidade de probabilidade das observações vistas por esse estado. A sequência dos estados não é diretamente observável, daí o nome de Modelos de Markov Não Observáveis, ou seja, a sequência dos estados está implícita, não sendo conhecida. Apenas a probabilidade dessa sequência de estados pode ser calculada [44].

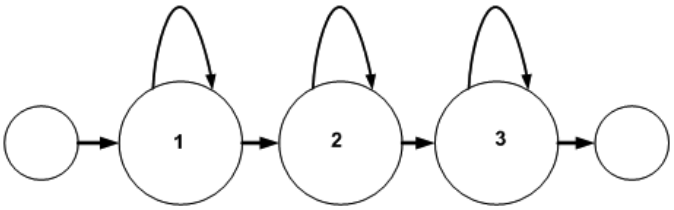

Figura 2.10 – Cadeia de Markov com estrutura "esquerda-direita"

Nesta dissertação foram utilizadas cadeias de Markov esquerda-direita com apenas 3 estados, conforme está representado na Figura 2.10. Tipicamente o modelo esquerda-direita representam um trifone (fone com contexto esquerda e direita), difone (fone com contexto à esquerda ou à direita) e monofone (fone independente do contexto), onde os seus estados representam o início, parte estável e seu fim. A função densidade de probabilidade associada a

cada um destes estados é descrita em termos de médias e variâncias de distribuições Gaussianas. Cada uma dessas descrições designa-se por mistura. Na prática, vamos ter as propriedades estatísticas dos MFCCs a serem comparadas com a descrição do estado atual. Dependendo do sinal acústico, o estado mantém-se ou transita para o seguinte. Conclui-se que, até um certo limite, quanto maior for o número de misturas, maior é a tolerância à variabilidade dos MFFCs e consequentemente, mais robusto é o AM. No projeto Tice.Healty (e nesta dissertação) utilizaram-se 16 misturas.

O número de trifones é muito superior ao de monofones. Ao considerar 39 fones, teremos um total de 39^3 trifones, assumindo que qualquer fone pode anteceder e suceder a cada fone. Isto pode tornar-se num problema de performance computacional. Para o contornar, partilha(m)-se a(s) parte(s) semelhante(s) entre os vários trifones. Esta técnica designa-se por *clustering* [41].

2.3.2 *Speech Recognition Engine Julius*

O *Julius* é um *software* em código aberto, de alta performance, usado para fins de investigação científica e aplicações industriais [9]. A primeira versão foi lançada em 1998, no seguimento do estudo sobre implementações eficientes para reconhecimento de fala contínua, com vocabulários extensos (LVCSR, Large Vocabulary Continuous Speech Recognition) [9]. Nasceu desta maneira o *Julius*, sendo atualmente considerada uma referência nas tecnologias da fala. Este incorpora as técnicas mais vulgares de reconhecimento de fala, sendo capaz de efetuar eficazmente LVCSR em tempo-real e com escassos recursos computacionais. Do ponto de vista de um reconhecedor de fala, o *Julius*, é visto como uma solução versátil e escalável [9]. Versátil por que é possível criar um sistema de reconhecimento, para qualquer língua, através da combinação de um AM, LM e dicionário apropriados. Escalável, por que o sistema tem a capacidade de reconhecer desde uma palavra, até às sessenta mil [9].

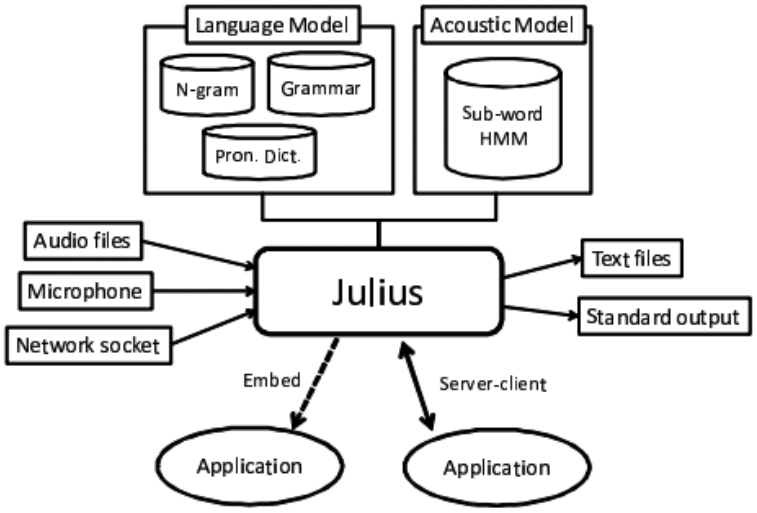

Figura 2.11 – Visão geral do sistema Julius [9]

Como já foi dito anteriormente, o *Julius* necessita que sejam criados externamente modelos acústicos e de linguagem para a língua em questão, tal como apresentado na Figura 2.11. É na dependência em relação aos modelos, que está o facto de o *Julius* ser independente (em si mesmo) à língua utilizada [9]. Com modelos apropriados, este será capaz de trabalhar com qualquer língua. Posto isto, conclui-se que o *Julius* é um mero descodificador do sinal de fala. Isto significa que os resultados apenas dependem da entrada de áudio e modelos utilizados (Figura 2.11). A precisão dos resultados depende (sempre e largamente) da qualidade dos modelos acústicos [9].

O *Julius* suporta entrada de som em tempo real bem como em ficheiros de áudio. Os ficheiros de áudio poderão ser WAV, RAW (WAV sem *header*), ou mesmo parametrização de áudio em MFCC. Os ficheiros de áudio deverão representar apenas um canal (*mono*), 16 bits por amostra e frequência de amostragem igual à que foi utilizada no treino de modelos acústicos (16000Hz no nosso caso) [46]. A entrada de microfone também é possível. No entanto, o acesso ao microfone depende do SO onde o *Julius* está a ser executado. Para aceder ao microfone, nos vários tipos de SO, o *Julius* contém um conjunto de API's designadas para esse efeito. Finalmente, também é possível aceder a áudio através da sua receção via *sockets*. Estes *sockets* representam, no emissor, uma das possíveis entradas do *Julius* (ficheiros ou microfone).

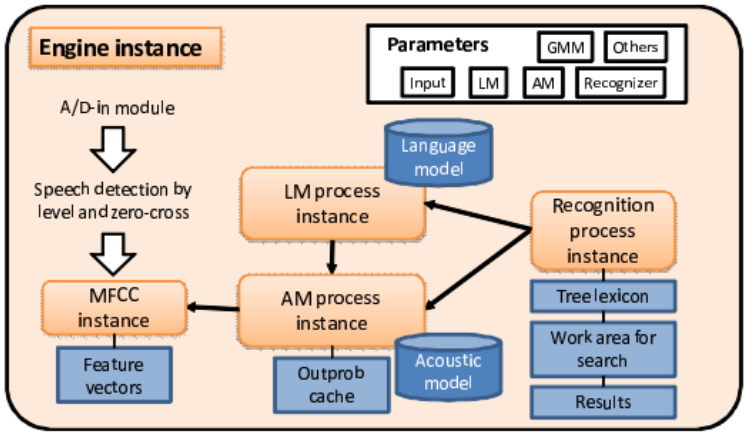

Figura 2.12 – Arquitetura do Sistema Julius

A arquitetura do *Julius* está representada na Fig. 2.12. Nesta verificam-se as semelhanças com o sistema de reconhecimento genérico, da Figura 2.9. Na verdade, estão representadas as mesmas instâncias. Apenas está adicionada a "instanciação do motor" (de reconhecimento). Este é responsável por alocar na memória as restantes instâncias, de acordo com os parâmetros de entrada [42].

Capítulo 3

JuliusClient

Neste capítulo serão apresentadas todas as questões que estão ligadas ao uso do sistema de reconhecimento de fala, no lado do cliente. Serão discutidas as estratégias adotadas para a criação de uma página web que permita interação com o utilizador através da fala. Para isso, começaremos por discutir a problemática do acesso ao microfone. Desta forma, será abordada a forma de aceder às amostras do cliente. Finalmente, serão expostas as implementações que visam melhorar a eficiência de todo o sistema: Deteção de Atividade de Voz (VAD, "Voice Activity Detection"), Buffer Circular e Sistema de Decimação com Filtragem AntiAliasing.

3.1 Acesso ao Microfone

O acesso ao microfone é possível através das API's facultadas pelos *browsers*. No entanto, como todas as implementações são revistas ao longo do tempo pelo W3C, foi obrigatório retificar a forma como se acede ao microfone do utilizador.

O método *getUserMedia()*, referido na Secção 2.1.1, começou por ser implementado através de *callback*:

```
Navigator.getUserMedia(constraints, successCallback, errorCallback);
```

Contudo, este método nunca chegou a reunir consenso, entre elementos do consórcio. Pelo que, no decorrer do projeto, foi marcado como "descontinuado" [47]. Na prática passou a haver o risco de que uma nova atualização, nos *browsers*, inviabilizasse o acesso ao microfone. Sendo assim, teve de se adotar a nova recomendação do W3C. Passou-se a aceder ao microfone através de uma *promise*. A *promise* representa uma operação que ainda não foi completada, mas espera-se que venha a ser no futuro [48]. Por essa razão pode ter 3 estados: *pending* (se estiver no estado inicial), *fulfilled* (operação completada com sucesso) e *rejected* (operação falhada, deverá devolver a razão da falha) [48]:

```
Navigator.mediaDevices.getUserMedia(constraints)
.then(function(mediaStream) { ... })
.catch(function(error) { ... })
```

Contudo viria a ser aplicada mais uma alteração, mais concretamente no *browser* Chrome. A Google decidiu seguir, mais uma vez, as normas do consórcio. Estas apontam o HTTP como um protocolo de transferência inseguro, devendo ser usado o protocolo seguro HTTPS [49]. Na prática, o Chrome passou a impedir o acesso ao microfone em páginas acedidas por HTTP [50]. Esta mudança obrigou a efetuar alterações no servidor Node.js. Anteriormente este disponibilizava as páginas web através de HTTP, no entanto também permite que sejam disponibilizadas em HTTPS [51]. Para que tal seja possível, recomenda-se a compra de um certificado TLS/SSL (Transport Layer Security/Secure Sockets Layer), a uma autoridade certificada. Tal não foi necessário, para provar a viabilidade do projeto. Recorreu-se assim ao OpenSSL para gerar o certificado e satisfazer os objetivos da dissertação. A ferramenta OpenSSL é uma implementação em código aberto para os protocolos SSL/TLS, escrita em C e com funções básicas de criptografia [52].

3.2 Acesso às Amostras de Som

Com o acesso ao microfone resolvido, o próximo passo é criar mecanismos para guardar o áudio e enviá-lo para o servidor. Numa primeira abordagem utilizaram-se bibliotecas facultadas pela WebRTC, que visam apoiar os desenvolvedores na construção de aplicações *web*. Foi utilizada a *RecordRTC*, definindo-se como biblioteca de gravação de dispositivos media, baseada em JavaScript e para *browsers* modernos [53]. A *RecordRTC* pressupõe um conjunto de métodos que permitem gravar faixas de áudio WAV. No fim da gravação estar concluída poderá guardar-se no disco do cliente e/ou fazer *upload* para o servidor. Como o *Julius* suporta ficheiros WAV, poderá parecer à primeira vista uma solução plausível. Tal não é o caso.

Primeiro, o envio do áudio para o servidor só é efetuado no fim da gravação. Isto provoca graves problemas de latência numa aplicação que se pretende em tempo real. Segundo, a frequência de amostragem do ficheiro WAV está fixada nos 48000Hz. Isto significa que a decimação teria de ser feita no servidor, já que o AM requer 16000Hz de frequência de amostragem. Isto provocaria uma sobrecarga adicional no servidor. Terceiro, um ficheiro WAV a 48000Hz, com 16 bits por amostra, ocupa num só segundo 768044 bits. Tendo em conta que só um terço desta informação é realmente aproveitada, estamos a desperdiçar tempo (no *upload*) e largura de banda.

Felizmente a Web Audio API proporciona formas de resolver os problemas anteriormente indicados. Existe um nó (na WAAPI) preponderante na implementação das bibliotecas WebRTC e que assumiu igual importância neste projeto. Esse nó designa-se por *ScriptProcessorNode*.

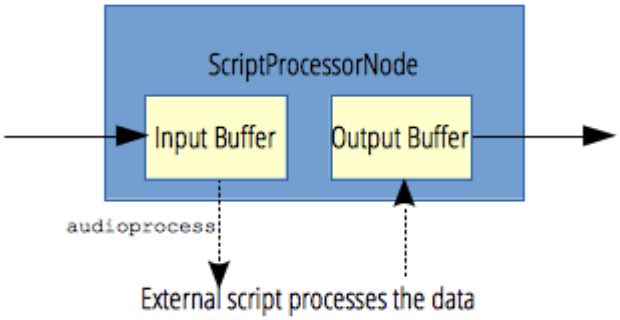

Figura 3.1 – Esquema do ScriptProcessorNode

O *ScriptProcessorNode* é uma interface que permite o processamento, ou análise de áudio, utilizando JavaScript [54]. Este está ligado a dois *buffers*, um que contém o áudio de entrada e outro que terá o áudio de saída. Um novo evento é desencadeado assim que o *buffer* de entrada seja totalmente preenchido com amostras. É no *event handler* correspondente (*onaudioprocess*) que poderemos ter acesso às amostras de som captadas. Assim que o *buffer* de saída seja preenchido, o *event handler* termina as suas operações. O processo repete-se tantas vezes quantas aquelas em que o evento seja desencadeado. Desconectando este nó da fonte (microfone) não haverá mais eventos. A sintaxe do *ScriptProcesssorNode* apresenta a seguinte forma:

```
Var audioCtx = new AudioContext();
audioCtx.createScriptProcessor(buffer_size, n_input_channels, n_output_channels);
```

Pegando em todas as potencialidades da WAAPI, especialmente do *ScriptProcessorNode*, construiu-se um *AudioContext* (Figura 3.2). Este procedimento permite combater as deficiências identificadas anteriormente. O *AudioContext* representado na Figura 3.2 ilustra os blocos de processamento que foram utilizados no projeto.

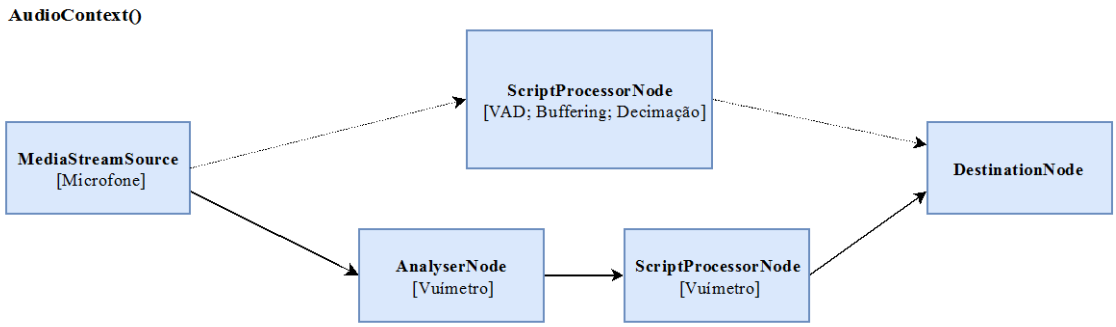

Figura 3.2 – AudioContext Utilizado

O *AudioContext* utilizado começa por instanciar o nó *MediaStreamSource*. Este é um dos *source nodes* possíveis na WAAPI e representa a interface que acede ao microfone. Depois disto, destacam-se dois ramos, que convergem no destino. O contexto de áudio começa por apresentar uma conexão a tracejado. No primeiro ramo implementou-se uma máquina de estados, de modo a criar Deteção de Atividade de Voz (VAD). Para isso usou-se a capacidade do *ScriptProcessorNode* em aceder às amostras de som, de modo a calcular a energia do seu *buffer* de entrada. Dependendo do valor de energia devolvida, são decididas as rotinas a executar. Estas poderão ser: i) alimentação de um *buffer* circular (que guarda as últimas frações de segundo, do áudio) ou ii) *streaming* dos buffers gerados para o servidor (através do *BinaryJS*). Em ambos os casos é feita decimação do áudio gerado. A ligação encontra-se a tracejado, uma vez que é dinâmica; entre o fim da locução e a chegada da resposta, o nó é desconectado.

O *AnalyserNode* destina-se à criação de um Vuímetro. Para que o utilizador tenha a perceção do volume com que diz os comandos. O *AnalyserNode* é uma interface que disponibiliza leitura, em tempo real, de propriedades no domínio do tempo e frequência [55]. Este nó é opcional, caso o desenvolvedor não queria mostrar o vuímetro só necessita de o desconectar. Finalmente, o *DestinationNode* representa o dispositivo de reprodução do cliente (colunas ou auriculares, dependendo da predefinição) [56]. Faz parte das normas da WAAPI que cada *AudioContext* termine no *DestinationNode*, mesmo que não sirva para reprodução de som. Isto acontece porque o *AudioContext.sampleRate()* é igual ao do dispositivo de reprodução pré-definido [57].

Esta implementação resolve todos os problemas levantados anteriormente. Primeiro, o áudio capturado é decimado para 16000Hz, o que significa que o volume de tráfego produzido é três vezes menor, libertando o servidor desta tarefa. Segundo, o áudio é transmitido em tempo real, resultando numa diminuição, evidente, da latência. Terceiro, a máquina de estados implementada (Secção seguinte) confere menor sobrecarga do servidor e menor desperdício de tráfego.

3.3 Deteção de Atividade de Voz

A Deteção de Atividade de Voz (VAD) é uma técnica utilizada no processamento de fala que tem a função de detetar a presença ou ausência de fala humana. Dada a complexidade dos algoritmos de reconhecimento de fala, esta técnica torna-se muito importante neste tipo de sistemas [58], evitando o processamento de silêncios entre locuções de fala. Percorrendo uma faixa de áudio, o VAD tem a capacidade de eliminar a informação irrelevante. Assim, o reconhecedor atua sobre uma faixa mais pequena, o que torna o resultado mais rápido e preciso [58].

Quando se decidiu fazer *streaming*, duas questões se colocaram: enviar o sinal captado ininterruptamente ou apenas as faixas de interesse. A primeira opção obriga a que os mecanismos de deteção de voz sejam implementados no servidor, através do *Julius*. A segunda opção obriga à implementação do VAD no cliente. Uma vez que se pretende um serviço eficiente, optou-se pela segunda via. Deste modo há uma menor sobrecarga de tarefas no servidor e diminui-se o volume de tráfego. Aliado a este facto está o baixo processamento que o VAD envolve, logo a máquina de estados implementada não será um transtorno ao utilizador.

Para o VAD utilizou-se uma máquina de estados. Eles são: "Áudio OFF", "Possível Áudio ON", "Áudio ON" e "Possível Áudio OFF" (Figura 3.3). Para que se faça a transição entre dois estados, deverão ser respeitas determinadas condições em torno do limiar. O limiar é único para todos os estados e representa um valor de energia que é multiplicado por uma constante e que varia de forma adaptativa. Para isso, é tomado um valor mínimo de energia que cresce linearmente até que se encontre outro valor mínimo. O processo de crescimento linear repete-se cada vez que é encontrado um novo valor mínimo de energia. No fundo, o limiar representa a variação dinâmica do mínimo, multiplicada por uma constante. Desta forma, o limiar adapta-se a uma situação em que exista ruído envolvente (ex. som de uma ventoinha). A constante (multiplicada pelo valor mínimo) e o crescimento linear são definidos pelo desenvolvedor ao implementar o *JuliusClient*, numa página HTML.

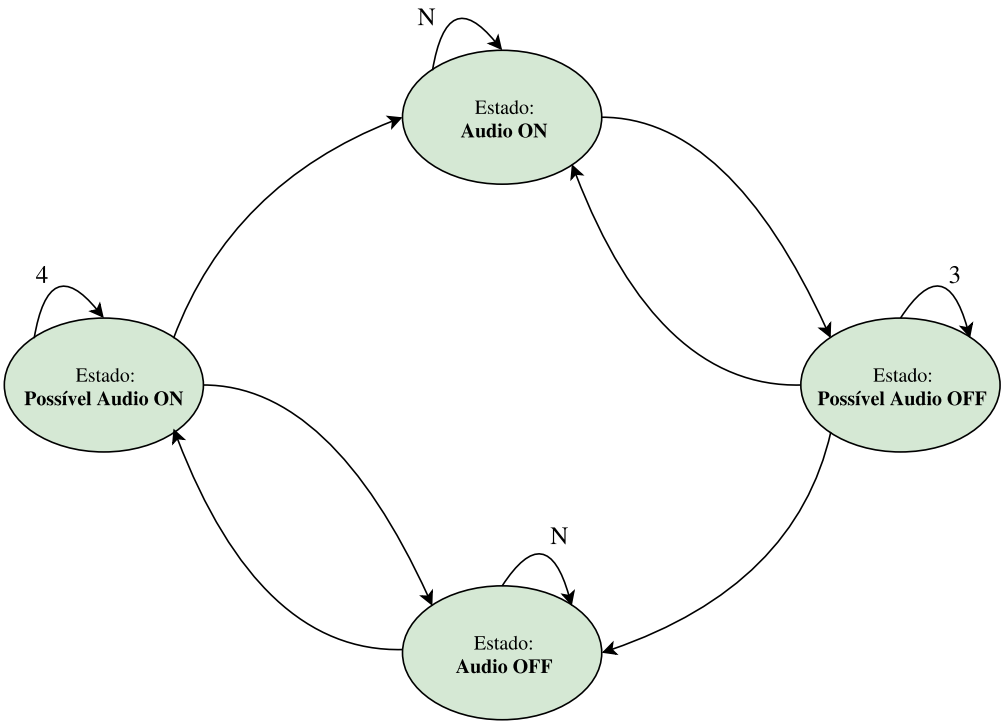

Figura 3.3 – Máquina de Estados responsável pelo VAD no cliente

A implementação do VAD começa por ser feita através do *ScriptProcessorNode,* pois este permite aceder às amostras de som captadas. Para isso, utilizaram-se *buffers* de 4096 amostras, um canal de entrada (*Julius* requer áudio *mono*) e um canal de saída (que é irrelevante, mas tem de ser alocado). Cada vez que o *buffer* de entrada é preenchido, calcula-se a energia desse sinal através de (3.1), onde $x[n]$ representa um buffer de áudio. Para justificar a fiabilidade da WAAPI, gravou-se um ficheiro WAV medindo-se a energia em tramas de 4096 amostras. Carregou-se a mesma gravação na WAAPI, fazendo o cálculo (3.1), através do *ScriptProcessorNode*. As diferenças revelaram-se inferiores a 0.01%, comparando trama a trama.

$$E = \sum_{n=0}^{4095} x^2[n] \tag{3.1}$$

As energias calculadas são guardadas em memória, através de um *array*. O último índice desse *array* representa o valor de energia atual. Ele serve de referência porque é comparado com valores anteriores, de modo a lhe ser atribuído um estado. Antes da atribuição de um estado, o último índice do *array* é comparado com o valor mínimo de energia obtido até ao momento. Desta forma, atualiza-se o limiar adaptativo antes da atribuição do estado.

O estado "Áudio OFF" representa a ausência de som ou fala com muito baixo volume (energia baixa). Isto quer dizer que a energia *E* se encontra abaixo do limiar. O estado devolvido, ao *ScriptProcessorNode,* dita que apenas se deve efetuar *buffering*. Este estado mantém-se inalterado N vezes, até que o limiar seja ultrapassado. Desta forma, transita-se para o estado "Possível Áudio ON".

No estado "Possível Áudio ON" o limiar de energia foi ultrapassado. Apesar de o ser, considera-se que o som captado tem uma origem ambígua. Isto porque não basta que o nível de energia seja ultrapassado, é necessário que o seja durante algum tempo. Caso contrário, o estado transita para "Áudio OFF". Desta maneira, é possível prevenir que estalidos ou outros ruídos de curta duração sejam enviados para o servidor. O estado devolvido, ao *ScriptProcessorNode,* dita que apenas se continua a efetuar *buffering*.

Em "Áudio ON", estamos perante a situação em que se considera a presença de fala, sendo cessadas as operações de *buffering*. Transmite-se o *buffer* previamente guardado, seguido de todos os *buffers* gerados neste estado. É iniciado o *streaming* para o servidor através da aplicação *BinaryJS* para clientes. O estado permanece inalterado N vezes, até que o valor de energia caia para valores abaixo do limiar.

Finalmente, o estado "Possível Áudio OFF" existe para absorver pequenas pausas entre as sílabas de palavras ou entre palavras. Por essa razão, o comportamento do *ScriptProcessorNode* (primeiro ramo da Figura 3.2) é igual ao descrito em "Áudio ON". Daqui existem duas possibilidades: a energia do áudio é recuperada e a locução continua (o estado regressa a "Áudio ON") ou a energia nunca chega a ser recuperada e transita para "Áudio OFF".

3.4 *Buffer* Circular

Como se viu na Secção anterior, não se deve enviar áudio para o servidor até que haja as garantias mínimas de que o limiar foi ultrapassado, durante um certo período de tempo. À primeira vista, poderá pensar-se que bastaria começar a preencher um *buffer* temporário no estado "Áudio Possível ON", sendo posteriormente enviado na transição para "Áudio ON". Esta solução revelou-se pouco precisa. Desta maneira, tornar-se-ia difícil o envio do segmento de baixa energia, que antecede o comando proferido. Este segmento é necessário porque antes de uma energia alta, poderá existir uma fricativa de baixa energia ou outro segmento fraco, que deve ser associado ao comando a reconhecer.

Para resolver esta questão foi implementado um *buffer* circular. A duração, no tempo, deste *buffer* tem de ser superior ao tempo gasto no estado "Possível Áudio ON". Isto porque é no estado "Possível Áudio ON" que o limiar é ultrapassado. Só desta forma se garante a captura do segmento de energia que está abaixo do limiar e que deve ser associado ao comando a reconhecer. O seu funcionamento é simples: o *buffer* começa por ser preenchido até à sua capacidade máxima, através da concatenação de novos *buffers*, gerados no *ScriptProcessorNode*; quando este atinge a sua lotação máxima, descarta o segmento mais antigo e concatena o mais recente.

3.5 Sistema Decimação com Filtragem Anti-Aliasing

Tal como está descrito na Secção 2.3.1, o *Julius* requer áudio do tipo mono, frequência de amostragem a 16000Hz e inteiros de 16 bits por amostra. Por sua vez, na WAAPI, o áudio é interpretado com frequência de amostragem igual à do dispositivo de reprodução [57] (48000Hz é standard [59]) e cada amostra está representada em *floats* de 32 bits. Mais uma vez se levanta a questão: a conversão deverá ser feita ao nível do cliente ou ao nível do servidor? Como nos casos anteriores, o argumento de poupar o servidor, aplica-se a esta situação.

Criou-se então um sistema de decimação (por 3), com filtragem *anti-aliasing*. Esta operação é efetuada tanto no *buffering*, como no momento que antecede o *streaming* dos *buffers* de áudio. O diagrama de blocos na forma direta 2 encontra-se indicado na Figura 3.4 e a equação (3.2) indica que se trata de um filtro IIR de ordem 3. Por sua vez, a resposta em frequência está indicada na Figura 3.5.

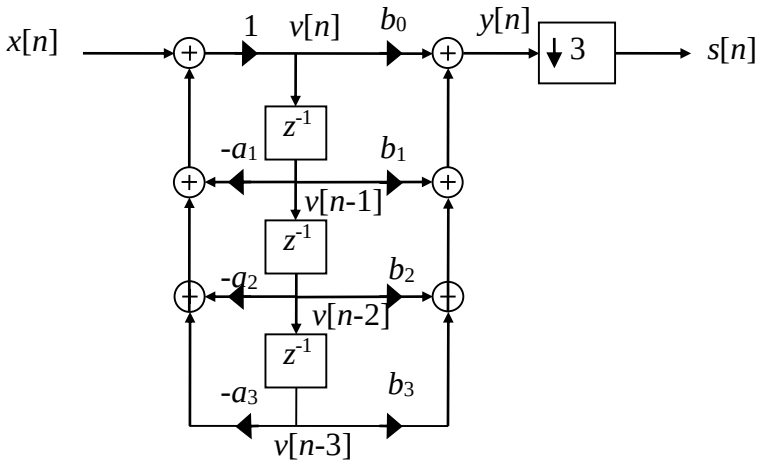

Figura 3.4 – Filtragem de ordem 3 seguida de decimação por 3

$$H(z)=\frac{B(z)}{A(z)}=\frac{b_0+b_1z^{-1}+b_2z^{-2}+b_3z^{-3}}{1+a_1z^{-1}+a_2z^{-2}+a_3z^{-3}} \tag{3.2}$$

Para que a fiabilidade do sistema fosse confirmada na WAAPI, começou-se por implementar o código em MATLAB. Em ambos os casos mediu-se a resposta a impulso do sistema (Figura 3.5), sendo que as diferenças são completamente irrelevantes (<< 0.001%).

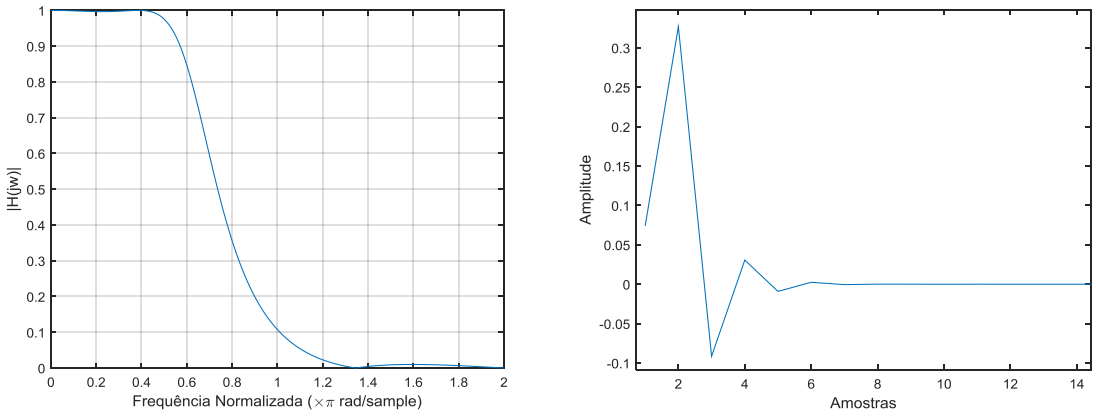

Figura 3.5 – Resposta em Frequência do filtro (Esquerda) – Resposta a Impulso (Direita)

Contudo, pode haver situações em que o utilizador não tenha definida a frequência de 48000Hz no seu dispositivo de saída. Nesse caso, o utilizador começa por ser alertado que a

frequência de amostragem não é a desejada. Caso não mude as configurações da sua placa de som, a página utiliza automaticamente a biblioteca *audiolib.js*. Esta libraria disponibiliza um conjunto de métodos, para manipulação de áudio (na WAAPI) [60]. O método utilizado, para este caso, foi o *resample()*:

```
Var r_buffer = audioLib.Sink.resample(input ,current_value_Hz/desired_value_Hz);
```

Este método permite a conversão de quaisquer frequências de amostragem. No entanto deve ser evitado, porque não há garantia que o *aliasing* seja eliminado. Futuramente a WAAPI não terá mais este problema. É intenção do W3C, que seja o próprio desenvolvedor a definir a frequência de amostragem no *AudioContex* [57][61].

A conversão de *Float32* para *Int16* é feita através de JavaScript puro. As amostras da WAAPI estão representadas em *Float32Array* [61]. Os valores deste tipo de *array* são representados em ponto flutuante de 32 bits, mas por definição da WAAPI, eles apenas variam entre -1 e 1. No entanto, o *Julius* requer que os valores das amostras estejam representados por inteiros de 16 bits (modulação PCM que varia de -32,728 a 32,767). Portanto, basta multiplicar as amostras negativas por -32,728 e as positivas por 32,767. No fim da multiplicação, as amostras são guardadas num *Int16Array*, onde os seus valores são representados por inteiros de 16 bits [62]. Guardando o resultado da multiplicação num *Int16Array*, a conversão para inteiros de 16bits é feita automaticamente.

3.6 Implementação *BinaryJS* no Cliente

Para diminuir a latência do serviço de reconhecimento, optou-se por uma solução que faça *streaming* de áudio para o servidor. A aplicação utilizada para este fim é o *BinaryJS*, já que esta permite o *streaming* de *arrays* e o áudio é enviado sob a forma de *Int16Array* [33]. Para implementar os métodos do *BinaryJS*, ao nível do cliente, é necessário invocação do script "*binary.min.js*" na página HTML. Este é disponibilizado pelos autores da API. A parte importante da API está referenciada na Tabela 2.1, da Secção 2.2.2.

A corrente Secção destina-se à contextualização dos métodos e eventos existentes, para a compreensão do *streaming* de áudio e receção de respostas.

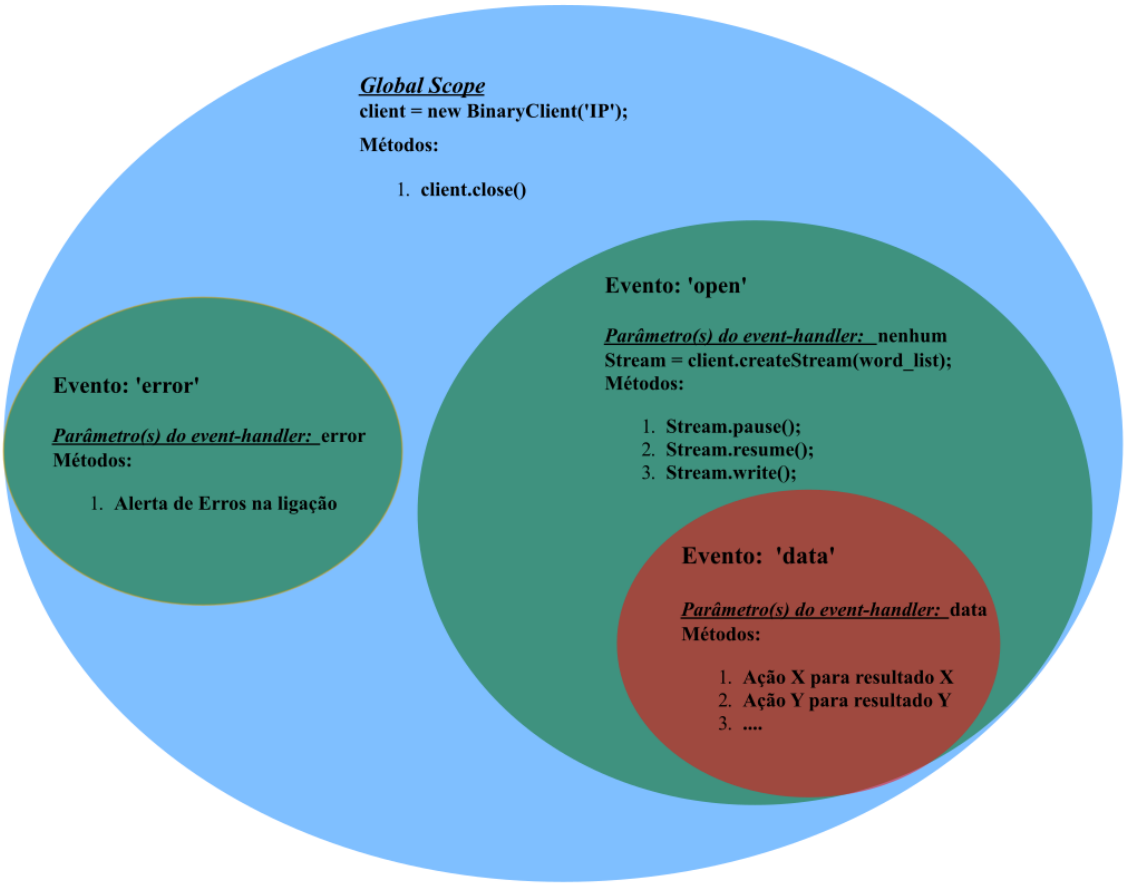

Figura 3.6 – Arquitetura event-driven no BinaryClient

Tal como está indicado na Figura 3.6, começa-se por invocar a inicialização da classe *BinaryClient*. Esta recebe como parâmetro de entrada o IP e porto do servidor e é criada a ligação com o *BinaryServer*. Após a inicialização dois cenários são possíveis: i) o servidor de reconhecimento está disponível e o evento *open* é desencadeado, ou ii) existem problemas na ligação e é desencadeado e evento *error*.

Assim que a ligação seja bem-sucedida, o microfone e o *AudioContext* são inicializados e é utilizado o método *createStream([meta])*, onde o parâmetro *meta* será uma *string* com a lista de palavras a reconhecer neste instante. Este método devolve um objeto *binaryjs.BinaryStream*, ao qual ficam associados os respetivos métodos e eventos. O método *resume()* é utilizado cada vez que o áudio começa a ser transmitido, ou seja, na transição de "Áudio Possível ON" para "Áudio ON". Ele dá a conhecer ao servidor o início da chegada de um novo comando. O método *write()* é utilizado cada vez que o *ScriptProcessorNode* tem um *buffer* de áudio filtrado, pronto a ser enviado, ou seja, é responsável pelo envio dos sucessivos fragmentos de áudio para o servidor. Este método é sucessivamente utilizado durante o estado "Áudio ON" e "Áudio Possível OFF". Por último, o método *pause()* é invocado cada vez que se dá o fim de uma locução. O que significa que é utilizado na transição de "Áudio Possível OFF" para "Áudio OFF".

Assim que o envio do comando seja concluído, o *ScriptProcessorNode* é desconectado. O objetivo desta desconexão é o de prevenir o envio de novo áudio, sem que o servidor tenha despachado a resposta anterior. Essa resposta surgirá através do evento *data*. É no *event handler* de *data* que se implementam as ações a efetuar na página, nomeadamente, a abertura de hiperligações, alteração da configuração da página, entre outras. Desencadeado o evento *data*, o *ScriptProcessorNode* volta a ser conectado ao *AudioContext*.

Questões técnicas, em relação à arquitetura *event driven*, serão abordadas no âmbito do Capítulo 4.

Capítulo 4
JuliusServer

Neste capítulo, serão expostas todas as questões relacionadas com o acesso ao servidor Node.js. Serão discutidas as estratégias adotadas na gestão do tráfego produzido nos clientes. Para isso, começaremos por referir a forma como se organiza a arquitetura *event-driven* utilizada. Com estas considerações seremos capazes de perceber a proveniência dos eventos gerados. Finalmente, será abordada uma solução com vista à escalabilidade do sistema. Para tal, foi aproveitado o facto de o *BinaryJS* conferir grande flexibilidade aos desenvolvedores. Esta é uma questão fulcral nas aplicações *web*. Assim, o sistema fica apenas limitado aos recursos físicos disponíveis.

4.1 Implementação *BinaryJS* no Servidor

O funcionamento do *BinaryJS* é abordado na Secção 2.2.2. Em suma, esta API utiliza estratégias de programação *event-driven* e *closures*. Estas duas técnicas são abordadas na Secção 2.2.1. A parte importante da API está referenciada na Tabela 2.1. Esta secção destina-se à contextualização dos métodos e eventos existentes, para a resolução das seguintes questões: distinção dos clientes, receção de áudio, resposta ao cliente e limitação de conexões.

Há uma questão de nomenclatura que merece ressalva. Como está descrito na Secção 2.2.1, os eventos são tratados por *event handlers* [28]. Esta é a designação utilizada para funções de *callback* que por sua vez se comportam como *closures* [28][30]. No contexto da dissertação, estes três conceitos são tecnicamente a mesma coisa. Será escolhida a designação *event handler* (“processador de eventos”, “tratador de eventos”), já que é a que melhor se adequa ao processamento de eventos.

A Figura 4.1 ilustra a arquitetura da programação *event-driven* aplicada. A imagem está organizada de modo a demonstrar os vários níveis de *event handlers* existentes. Desta forma, fica claro quais são as variáveis e métodos que cada *event handler* consegue aceder. Sendo assim, o círculo mais exterior representa o escopo global, considerado o nível mais alto. Os restantes círculos formam-se a partir deste e referem-se a eventos aos quais estão associados *event handlers*. Em cada círculo estão contidos os métodos e propriedades definidos pelos mesmos. Círculos da mesma cor (que representa o mesmo nível) não partilham variáveis ou métodos inicializados por si mesmos. Apenas conseguem aceder aos que estão no *enclosing environment*.

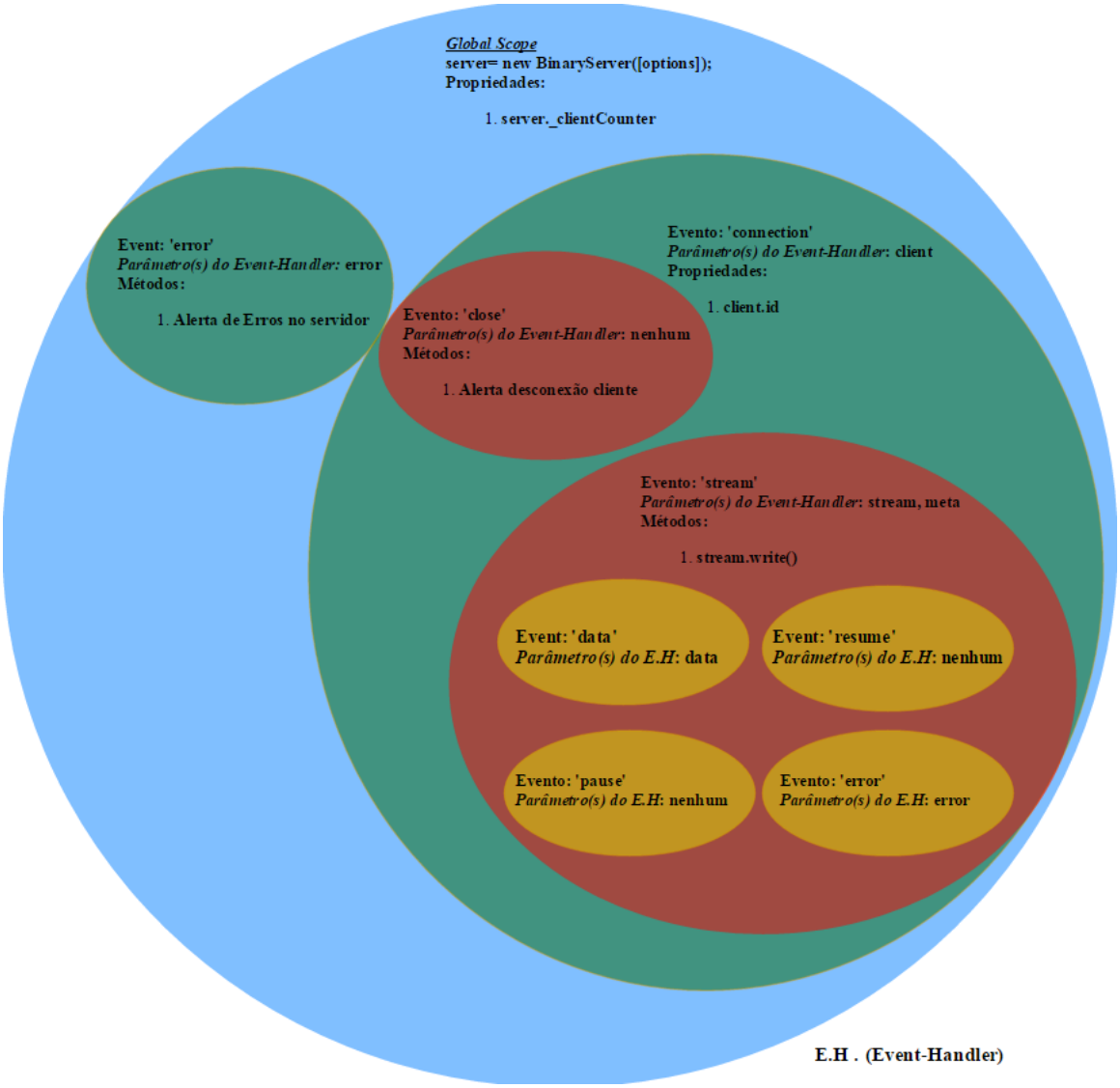

Figura 4.1 – Implementação BinaryJS no Servidor

Tal como se pode observar na Figura 4.1, verificamos a existência de *event handler* dentro de *event handler*. Tendo presente os pressupostos da Secção 2.2.1, isto significa que o *event handler* de mais baixo nível conseguem ler/editar as variáveis dos *event handlers* hierarquicamente acima. O contrário já não acontece. Este aspeto é muito importante, pois desta maneira, se a identificação dos clientes for atribuída em alto nível, passa a ser visível nos níveis que lhe estão abaixo; ou seja, qualquer que seja o nível do evento é possível perceber quem o gerou.

A situação exposta em cima descreve exatamente o que acontece na prática. O *event handler* do *BinaryServer* é uma função *callback* com o parâmetro *client*. Este parâmetro é um objeto com vários métodos e propriedades associadas. O mais importante para este caso é que *client* tem a propriedade *id*. O *id* é um identificador (inteiro) que identifica cada cliente, que está conectado ao

servidor. Por sua vez, este é visível em qualquer *event handler* de igual ou mais baixo nível, logo *cliente.id* é a forma que nos permite identificar a origem dos eventos gerados.

4.1.1 Interação entre *BinaryServer* e *BinaryClient*

Para avançar com a explicação prática do *BinaryServer* é preciso ter presente o funcionamento do *BinaryClient*, descrito na Secção 3.7. As duas instâncias interagem diretamente uma com a outra. Por essa razão, os eventos desencadeados no servidor dependem dos métodos executados no cliente (e vice-versa). Esta relação de ação-reação está ilustrada na Figura 4.2. Nela podemos verificar que cada método (ação) desencadeia um evento (reação) na extremidade oposta.

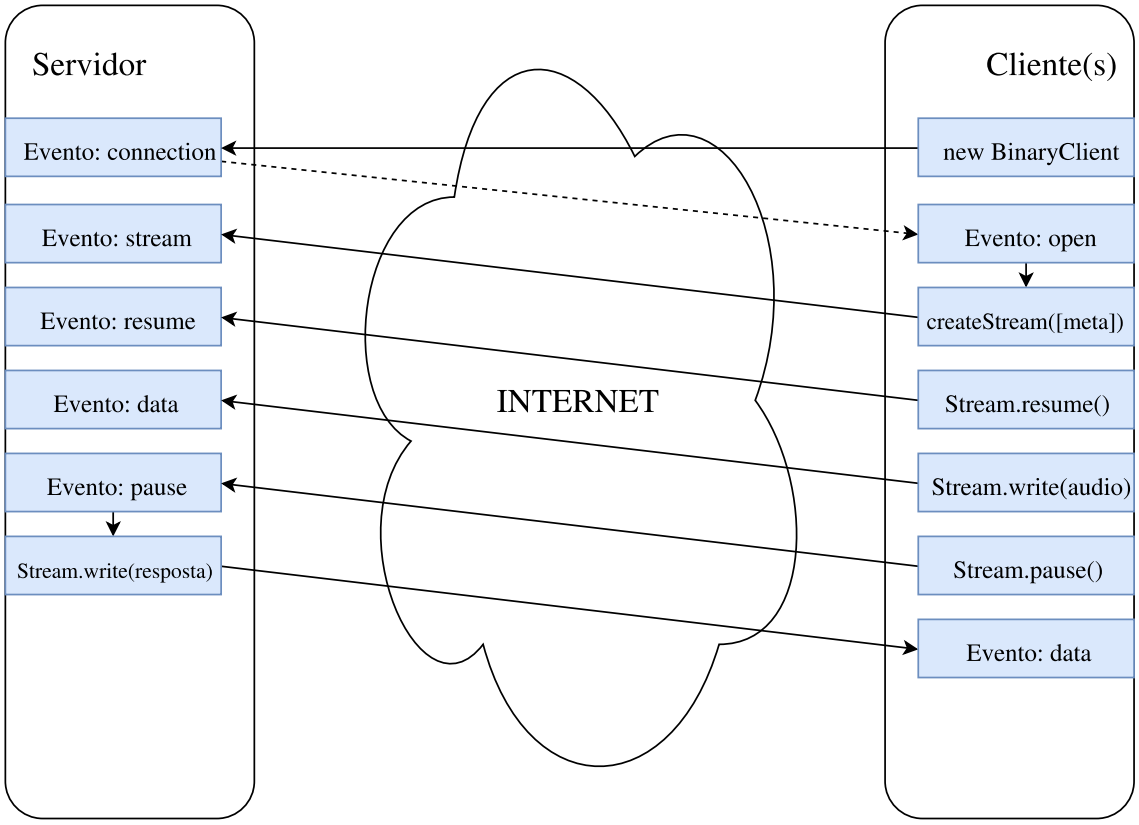

Figura 4.2 – Interação entre Servidor e Cliente, através do BinaryJS

Tal como está indicado na Figura 4.1, começa-se por invocar a inicialização da classe *BinaryServer*. Após a inicialização existem dois desfechos possíveis: o servidor dá erro na inicialização (desencadeando o evento *error*) ou a inicialização é bem-sucedida e aguarda ligações de clientes.

Quando o utilizador abre a página *web* (onde está a inicialização de *BinaryClient*) é desencadeado o evento *connection*, no servidor. É no seu *event handler* que é atribuído um *id* ao cliente que se conectou. Por sua vez, o *BinaryServer* faz desencadear o evento *open*, no cliente.

No *event handler* de *open* é inicializado o objeto *stream*. Esta inicialização desencadeia o evento *stream*, no servidor. No seu *event handler* é guardada a gramática utilizada por um determinado cliente. Como vimos anteriormente, tal é permitido, pois é possível aceder ao *id* do cliente que gerou o evento.

A determinada altura o cliente começará a fazer *streaming* de áudio. Essa intenção é indicada ao servidor, através do método *resume()*, desencadeando o evento *resume* (no servidor). Seguidamente, o cliente começa a enviar os segmentos de áudio. Cada segmento de áudio que chega desencadeia o evento *data*. Este *event handler* tem o parâmetro *data*, que é um *buffer* de dados binários. Estes, por sua vez, correspondem à gravação no cliente.

Finalmente, quando o cliente finaliza a locução do comando é desencadeado o evento *pause*. Nesta altura o *Julius* (que está integrado no servidor) terá de gerar uma resposta, com o áudio adquirido. A forma como este processo é feito será abordada no Capítulo 5. Neste momento, apenas importa reter que a resposta gerada é enviada através do método *write()*. Quando a resposta chega ao cliente, desencadeia-se o evento *data*. Embora estejam vários utilizadores ligados, as respostas nunca são trocadas. Tal acontece pois o *BinaryServer* encarrega-se de guardar as informações necessárias nos objetos *stream* e *cliente*. Ao implementar os métodos destes objetos, estamos a fazê-lo tendo em conta as propriedades guardadas nos mesmos. Essas foram delineadas de acordo com os pressupostos da Websocket API. Daí que o *BinaryJS* disponibilize uma interface simples e intuitiva para questões de *streaming*.

4.1.2 Número Máximo de Clientes

Poderá haver a necessidade de limitar o acesso ao servidor. Esta operação é muito importante. Como veremos no Capítulo 5, o número de reconhecimentos em simultâneo tem um máximo permitido. O *BinaryJS* permite esta limitação de modo eficaz. Para isso, o *BinaryServer* tem uma propriedade designada por *_clientCounter*. Esta propriedade mantém atualizado o número de clientes conectados. Qualquer ligação que cause um incremento de *_clientCounter*, acima do máximo pretendido, poderá ser rejeitada. Para isso, basta utilizar o método *close*, quando o máximo é ultrapassado. Desta forma, será desencadeado o evento *close* no cliente. Este terá no *event handler* o alerta: "A quota do servidor de reconhecimento foi ultrapassada. Tente mais tarde."

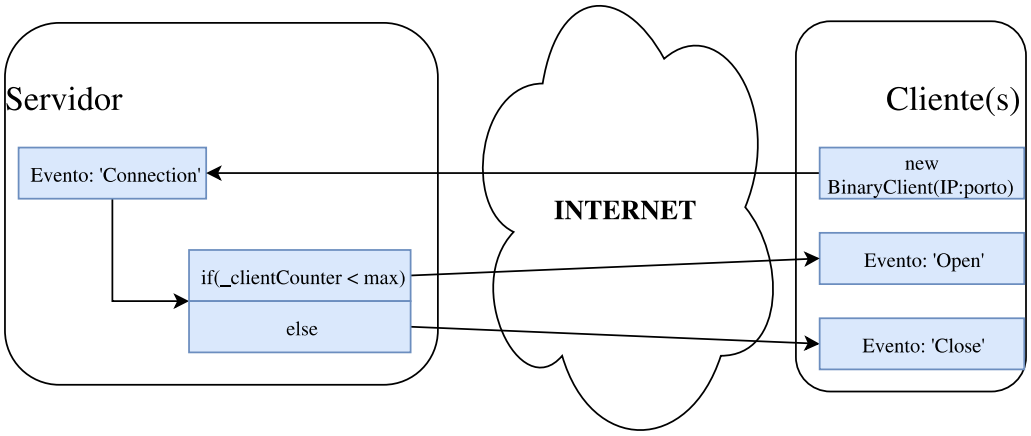

Figura 4.3 – Limite de acessos ao Servidor

4.2 Escalabilidade do Servidor Node.js

Escalabilidade é uma característica desejável em todo o sistema. Esta indica a capacidade de manipular uma porção crescente de trabalho de forma uniforme, ou estar preparado para crescer [63]. A escalabilidade de um sistema deve ser analisada, consoante o propósito do mesmo. Em linhas gerais, um sistema cujo desempenho aumenta com o acréscimo de *hardware*, proporcionalmente à capacidade acrescida, é considerado escalável [64]. Para auxiliar esta avaliação existem várias dimensões, tais como [64]:

- Escalabilidade Administrativa – A capacidade de um número crescente de utilizadores, serem suportados pelo sistema.

- Escalabilidade Funcional – A capacidade melhorar o sistema, adicionando novas funcionalidades ao mínimo esforço.
- Escalabilidade Geográfica – Capacidade de manter o desempenho, independentemente da expansão do sistema ser feita no mesmo local geográfico, ou de forma mais distribuída.
- Escalabilidade de Carga – Capacidade de um sistema expandir ou contrair os recursos utilizados, consoante a procura dos mesmos.
- Escalabilidade de Geração – Capacidade do sistema continuar a ser integrado em novas componentes. Também é a capacidade do sistema ser compatível com diferentes produtores.

Como está descrito na Secção 2.2, o Node.js é uma ferramenta multiplataforma. Ao funcionar em vários SO's podemos afirmar que a Escalabilidade de Geração é garantida. Outro aspeto que verificámos foi a arquitetura *event driven*. Ao funcionar através de eventos, o Node.js apenas necessita de manter o *event loop*. Todos os recursos computacionais utilizados, para além do *event loop*, são devido aos eventos gerados pelos clientes, logo a Escalabilidade de Carga é satisfeita. O Node.js permite a atualização do código sem que o servidor tenha de ser desligado. Após a atualização deverá ser reiniciado. Além disso, a adição de uma nova funcionalidade não obriga à alteração integral do código. Isto vai de encontro à Escalabilidade Funcional.

4.2.1 Resolução da Escalabilidade de Administrativa e Geográfica

Foi implementada uma solução que resolve o problema da Escalabilidade Geográfica e Administrativa. Para esse fim, foi tido em conta que um servidor *BinaryJS* se pode comportar como cliente perante o servidor pai. O esquema da solução apresentada está na Figura 4.4.

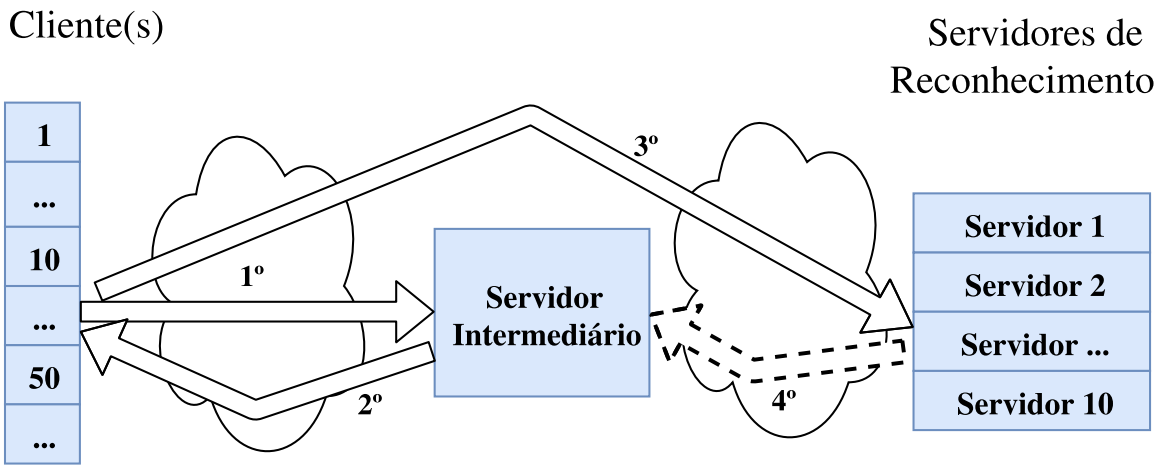

Figura 4.4 – Solução BinaryJS para a escalabilidade

A estratégia consiste em utilizar um servidor que atue como intermediário entre o cliente e o servidor de reconhecimento. Posto isto, o cliente começa por ligar-se ao servidor intermediário. Este mantém atualizado o número de clientes conectados a cada servidor de reconhecimento. Ao receber a conexão de um novo cliente, o intermediário devolve a localização do servidor de reconhecimento mais desocupado. Quando o cliente recebe a localização, é criada a conexão com o servidor de reconhecimento correspondente. Por sua vez, este envia a atualização do número de clientes que tem ligados. O processo repete-se aquando da entrada ou saída de um cliente.

Desta maneira resolve-se a Escalabilidade Administrativa. Quanto mais servidores de reconhecimento forem ligados ao intermediário, maior será a capacidade de resposta a um aumento de utilizadores. Os servidores de reconhecimento podem correr em qualquer máquina, com IP visível na Internet. Por essa razão, o sistema passa a ter Escalabilidade Geográfica.

Capítulo 5
JuliusAddon

Como já foi abordado na Secção 2.3.2, o sistema de reconhecimento *Julius* está escrito na linguagem C. No entanto, o servidor Node.js é implementado em JavaScript. Este capítulo descreve esta (aparente) incompatibilidade, abordando a integração do *Julius* no servidor. Serão expostos os principais entraves à integração do *Julius*. Consequentemente serão referidas as principais alterações efetuadas, de modo a combater esses entraves. A principal motivação das alterações efetuadas relaciona-se com o aumento da eficiência da interação com o Node.js, sem comprometer as funcionalidades originais. Finalmente, serão expostas as modificações que permitem um consumo de recursos computacionais mais eficiente.

5.1 Limitações do *Julius*

Antes de passar à integração do *Julius* no servidor é fulcral compreender algumas das limitações impostas pelo *Julius*. Só desta forma se entendem algumas das decisões tomadas na abordagem executada.

Em primeiro lugar, apenas duas das possíveis entradas de áudio oferecem garantias do áudio ser processado em tempo-real: o acesso direto ao microfone e os *sockets* TCP/IP. A primeira, não faz sentido numa implementação de servidor. A segunda foi utilizada numa primeira abordagem e sem sucesso, logo o *Julius* não oferece uma solução próxima do processamento em tempo-real e que possa ser utilizada na integração com o Node.js. Para contornar esta limitação, foi adicionada um novo tipo de entrada ao *Julius*. Ela designa-se por *BinaryStream* e será discutida mais à frente.

A próxima limitação tem que ver, com a forma como o *Julius* deve ser utilizado. O *Julius* enquanto *software*, tem rotinas de inicialização utilizadas para alocação de memória para modelos acústicos e de linguagem, de memória para *buffers* de processamento de áudio, de memória para dicionários e gramáticas, entre outros [9]. Consequentemente, também apresenta rotinas para libertar da memória todas as instâncias previamente mencionadas.

Para testar a forma como o *Julius* deve ser utilizado, no Node.js, experimentaram-se duas vias. Ambas garantem o correto funcionamento do reconhecedor. Na via A, colocaram-se as

rotinas de alocação, reconhecimento e libertação da memória dentro de um ciclo. Cada iteração corresponde a um resultado. Na via B, fez-se alocação de memória e colocaram-se as rotinas de reconhecimento dentro de um ciclo. Ao fim de N iterações, efetuou-se libertação da memória. Ambos os esquemas estão representados na Figura 5.1.

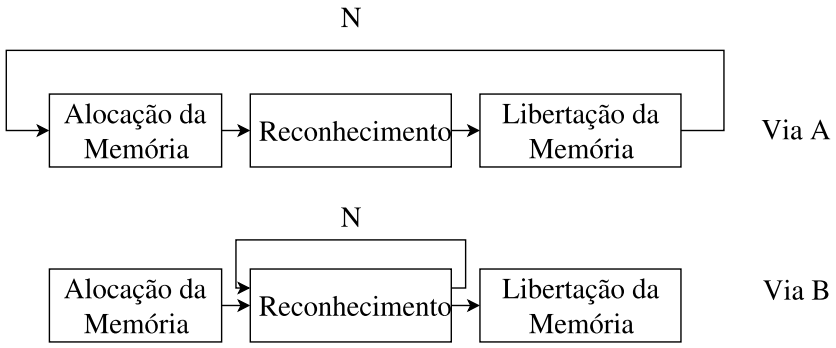

Figura 5.1 – Esquemas de utilização do Julius

Os resultados obtidos, ao fim de N interações, encontram-se na Tabela 5.1, na qual consta a memória utilizada pelo programa ao fim de N reconhecimentos. Considera-se como referência a memória utilizada ao fim de um reconhecimento. O lixo é calculado através da subtração da memória ao fim de N reconhecimentos com o valor de referência. Desta forma, verifica-se que a Via A tem um crescimento de lixo proporcional ao número de reconhecimentos, enquanto que na via B não se verifica acumulação de lixo.

Tabela 5.1- Comportamento da Memória após N reconhecimentos

Número de Reconhecimentos	Via A	Lixo de A	Via B	Lixo de B
1	4,348 MB	-	4.246 MB	-
50	7,538 MB	≈3.2 MB	4.248 MB	≈0 MB
300	17,314 MB	≈13 MB	4.243MB	≈0 MB
10000	370.756 MB	≈366 MB	4.243 MB	≈0 MB

A consequência prática desta medição é que se optou pela Via B na integração do *Julius* com o Node.js, ou seja, os objetos de reconhecimento são alocados apenas uma vez. Os reconhecimentos serão efetuados na memória previamente alocada. Seguir pela Via A implicaria a instanciação de um novo objeto a cada pedido. Quando o pedido terminasse, seriam invocadas as rotinas de libertação da memória. Consequentemente, ao fim de 10000 reconhecimentos o servidor teria aproximadamente 366 MB de lixo. Mesmo que esta instanciação seja feita numa nova *thread*, a libertação de memória nunca seria eficaz. Matar uma *std::thread* C++2011 não liberta a memória alocada pela mesma [65]. Foram efetuados esforços na compreensão do comportamento da memória, seguindo a Via A. Contudo, não se descobriu a origem do lixo gerado.

Outra limitação foi encontrada no *Julius*. Este não está preparado para efetuar reconhecimentos em paralelo. No projeto TICE Healthy esta questão não era relevante. O *plugin* é utilizado individualmente na máquina de cada cliente. Neste caso, o servidor tem de lidar com vários clientes em simultâneo, logo é necessário que o reconhecimento seja feito em paralelo. Para que tal seja possível, foram efetuados pequenos ajustes no *Julius*. Eles serão descritos na Secção 5.4.

5.2 *Node.js Addon*

É nos *event handlers*, gerados pelo *BinaryJS* que são executadas as rotinas referentes ao *Julius*. Inicialmente testou-se a ideia do *Julius* ser executado fora do servidor. Ela revelou-se ineficaz pelas seguintes razões: i) se múltiplos clientes estiverem ligados, torna-se difícil perceber a correspondência entre o resultado e cliente, ii) a solução do problema i) vai contra as estratégias indicadas na Secção 2.2.1 e iii) a implementação da lista de palavras a reconhecer, num dado instante, implicaria um esforço inexequível. Houve a necessidade de uma alternativa. Esta surgiu através da implementação de uma *interface* entre o *Julius* e o Node.js. Uma *interface* (API) deste tipo é vulgarmente designada por *addon* (ver Secção 2.2.3).

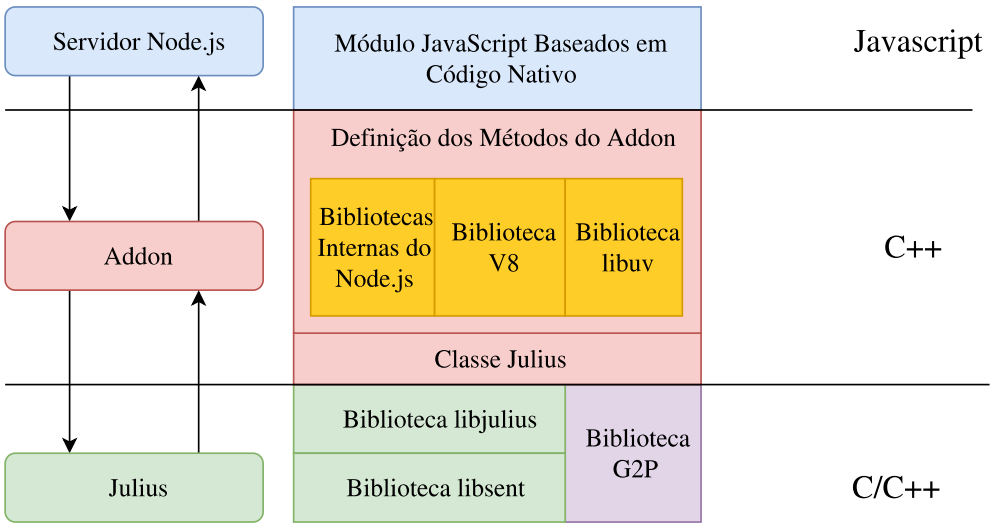

Figura 5.2 – Esquema Integração do Julius com o Servidor

Na Figura 5.2 podemos observar a arquitetura do servidor, no que diz respeito à integração do reconhecedor de fala. Nela podemos observar o papel que o *addon* tem ao nível desta integração. Ela serve de intermediário entre o nível mais alto do servidor e o *Julius* em si mesmo, ou seja, tem a função de traduzir os pedidos que chegam ao servidor em processos de reconhecimento no *Julius*. Inversamente, recebe os resultados do reconhecedor e transforma-os

em respostas para o cliente. Apesar do reconhecedor *Julius* ser escrito em C, integra uma API que permite a sua utilização numa classe C++ [9]. A metodologia da "Classe *Julius*" que será abordada na Secção 5.3.

Nesta secção serão abordados os métodos invocados pelo servidor, a sua função e o seu porquê. A Tabela 5.2 expõe os métodos implementados.

Tabela 5.2- Definição dos Métodos invocados pelo Servidor Node.js

Métodos	Parâmetros	Tipo	Função
`init_g2p`	Caminho para o Dicionário de Pronunciação; Caminho para o Modelo	Síncrono	Carrega a funcionalidades de conversão de fonemas para grafemas.
`init_julius_objects`	Número de Objetos	Síncrono	Faz alocação das variáveis de reconhecimento, para um determinado objeto.
`add_commands`	Dicionário do Cliente; Tamanho do Dicionário; ID do Cliente	Síncrono	Faz conversão de grafemas para fonemas, mediante o Dicionário do Cliente. Adiciona o resultado ao Julius.
`init_recog_thread`	ID do Cliente	Síncrono	Inicia a *thread* de reconhecimento
`buffering`	Áudio; Tamanho do Segmento de Áudio; ID do Cliente	Síncrono	Carrega o áudio que chega ao servidor, no *Julius*.
`julius`	ID do Cliente	Assíncrono	Finaliza o reconhecimento. Devolve, ao servidor, o resultado e a confiança do mesmo.
`stop_recognition`	ID do Cliente	Assíncrono	Termina o reconhecimento, se houver uma saída súbita do cliente.

A invocação do *addon* inicia-se através do método *require*, devolvendo os métodos definidos pelo mesmo. Seguidamente, utilizam-se os métodos *init_g2p* e *init_julius_objects*. Como se pode constatar, através da Tabela 5.1, estes métodos foram implementados sincronamente. Neste contexto dizer que o método é síncrono, significa que está a ser executado no *event loop*, logo bloqueia o servidor. À primeira vista, pode parecer que isto é contra as recomendações do Node.js (ver 2.2.1). Contudo, está propositadamente desta forma. A ideia desta escolha é que todos os objetos têm de estar carregadas na memória do servidor, antes de este poder receber ligações de clientes. Esta é a consequência prática de seguir a Via B, apontada na Secção 5.1. Assim, em primeiro lugar carregam-se na memória todos os objetos. Os reconhecimentos serão feitos na

memória alocada. Ao inicializar um novo objeto, para um novo pedido, estaríamos a seguir a Via A. Desta forma, quanto maior fosse o número de reconhecimentos, maior seria o lixo acumulado.

Seguidamente é inicializada a classe *BinaryServer*. Portanto, o servidor fica apto em receber ligações de clientes. Quando os clientes utilizam o método *createStream()* é desencadeado o evento *stream* (ver Figura 4.2). Como já vimos, o seu *event handler* tem como parâmetro o dicionário utilizado pelo cliente. É neste momento que se utiliza o método *add_commands*. Mais uma vez, o método é síncrono. Mas se analisarmos a Tabela 5.3, verificamos que o tempo médio de execução é demasiado pequeno para que compense a implementação no modo assíncrono.

Tabela 5.3- Tempo médio consumido por cada método

Métodos	Tempo Médio Consumido [ms]
`init_g2p`	144
`init_julius_objects`	400 (10 objetos)
`add_commands`	2
`init_recog_thread`	0
`buffering`	1
`julius`	>80
`stop_recognition`	>2

Antes de o cliente começar o envio de áudio é desencadeado o evento *resume*. Neste evento é inicializada a *thread* onde se desenrola o reconhecimento. Seguidamente, começam a chegar os segmentos de áudio, através do *event handler* de *data*. Estes são passados para o *Julius* através do método *buffering*. Ambos os métodos são síncronos. A razão para serem deste tipo é igual à que foi exposta no método *add_commands*. Mais ainda, o método *buffering* faz concatenação de segmentos de áudio. Este é um tipo de tarefa que deve ser feito sincronamente. Caso contrário, as invocações de *buffering* poderiam colidir, na escrita do *buffer* [66]. Na prática, o método *init_recog_thread* não causa bloqueio. Isto deve-se ao facto da inicialização de uma *std::thread* C++2011 não causar bloqueio. As *threads* serão discutidas, com maior detalhe, na Secção 5.4.2.

Quando o *stream* de áudio acaba é desencadeado o evento *pause*. Neste *event handler* é executado o método *julius*. O método *julius* tem a função de terminar a *thread* de reconhecimento, devolvendo o resultado e confiança ao servidor. A duração deste método depende do tamanho do comando de voz. Por essa razão, é do tipo assíncrono. Neste contexto, assíncrono quer dizer que o método é executado fora do *event loop*, logo não bloqueia o servidor. Quando o reconhecimento termina, é gerado um evento. O seu *event handler* encarregar-se-á de enviar o resultado ao cliente.

Na eventualidade do cliente abortar voluntária ou involuntariamente a ligação, existe o método *stop_recognition*. Este método é invocado sempre que se desencadeia o evento *close*.

Neste caso existem duas possibilidades: o cliente abandona o servidor durante o processo de reconhecimento, ou o cliente abandona o servidor fora de um processo de reconhecimento. No primeiro caso, o *addon* aborta o reconhecimento que ficou pendente, libertando o objeto para um novo cliente. No segundo caso, apenas liberta o objeto para um novo cliente. Como se pode observar na Tabela 5.2, este método é assíncrono. Tal acontece uma vez que o primeiro caso pode consumir cerca de 30ms - desta maneira evita-se o bloqueio do servidor.

5.3 Classe *Julius*

O reconhecedor *Julius* é composto por duas bibliotecas: a *LibSent* e a *LibJulius*. A *LibSent* é a biblioteca de programas para reconhecimento de frases cuja nomenclatura decorre de "lib(rary)" em conjunto com "sent(ences)". A *LibSent* contém várias funções como gestão do áudio, pré-processamento, extração de características, gestão de modelos acústicos, cálculos de probabilidades, entre outros [5] e é utilizada pela *LibJulius.* A *LibJulius* implementa o motor de reconhecimento e apresenta uma API para expor as funcionalidades do motor de reconhecimento [5]. A API permite que bibliotecas escritas em C possam ser utilizadas por uma classe escrita noutra linguagem.

Se nos reportarmos ao apresentado na Figura 5.2, verificamos que a definição dos métodos do *addon* é feita com base na "Classe *Julius*". A "Classe *Julius*" é completamente independente da definição dos métodos do *addon*, ou seja, esta classe pode efetuar reconhecimento autonomamente num *script* C++, porque interage diretamente com o reconhecedor *Julius*. Deste modo se conclui que a "*Classe Julius*" pode ser considerada como uma interface entre o motor de reconhecimento e o *addon*. Assim, a ausência da "Classe *Julius*" inviabiliza o funcionamento do *addon*.

A "Classe *Julius*", criada no âmbito do Projeto *TICE Healthy*, é composta de todas as funções membro necessárias ao reconhecimento. Nestas incluem-se a invocação das funcionalidades do *Julius* e as da biblioteca G2P (Graphemes to Phonemes). A última biblioteca já existia no laboratório e é responsável pela conversão de grafemas para fonemas [67], tendo um papel fundamental na criação do dicionário dinâmico. A "Classe *Julius*" foi aproveitada e reajustada para o contexto da dissertação.

Na Secção 5.2 apenas foram abordados os métodos utilizados pelo servidor e a sua função. Estes estão inteiramente dependentes dos métodos da "Classe *Julius*". A Tabela 5.4 apresenta esses métodos, a sua função e parâmetros. Alguns dos membros de função criados obrigaram a

alterações na versão original do *Julius* (entenda-se *LibJulius* e *LibSent*), as quais serão discutidas na Secção 5.4.

Tabela 5.4- Métodos da Classe Julius

Método	Parâmetros	Função
`InitSREngine`	Caminho Ficheiro de Configuração; ID do Cliente	Faz alocação das variáveis de reconhecimento, para um determinado objeto.
`InitG2P`	Caminho para o Dicionário de Pronunciação; Caminho para o Modelo	Carrega, na memória, o Dicionário de Pronunciação e o respetivo Modelo.
`DynamicAddCommands`	Dicionário do Cliente	Adiciona\Altera o Dicionário de um determinado objeto.
`Callbacks`	Nenhum	Regista a função *callback* do resultado
`Audio_Buffering`	Segmento de Áudio; Tamanho do Segmento de Áudio; Flag de Início; Flag de Fim	Passa os segmentos de áudio, para o *Julius*
`Audio_recog_thread`	ID do Cliente	Inicia o reconhecimento, através de uma *thread*
`ReturnCM`	ID do Cliente	Devolve a Confiança no resultado
`ReturnWord`	ID do Cliente	Devolve o resultado

Analisando os métodos expostos na Tabela 5.4 torna-se possível antever a forma como os métodos da "Classe *Julius*" conjugam com os métodos executados pelo servidor. Esta relação está ilustrada, com mais pormenor, na Figura 5.3.

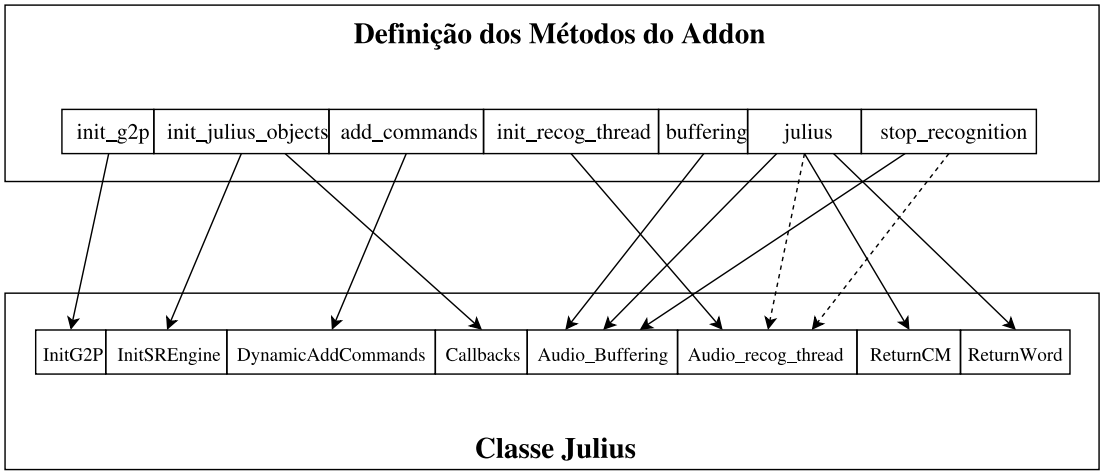

Figura 5.3 – Interação entre os Métodos do Addon e os Métodos da Classe Julius

Os métodos *init_g2p*, *init_julius_objects*, *init_recog_thread*, *buffering* e *add_comands* não oferecem qualquer dúvida: invocam o método da "Classe *Julius*" correspondente. Para que não existam dúvidas, o servidor não pode invocar diretamente os métodos da "Classe *Julius*". Tal

acontece porque é necessária a tradução dos argumentos da linguagem *JavaScript* para C++ (e vice-versa).

O método *julius* inicia-se quando o comando de voz termina invocando *Audio_Buffering*. Este último passará uma *flag* a indicar que o comando de voz acabou. Seguidamente, a *thread* que se iniciou em *init_recog_thread* é finalizada. O *init_recog_thread* executou o método *Audio_recog_thread* através de uma *std::thread* C++2011. Quando o reconhecimento é finalizado são devolvidos o comando reconhecido e índice de confiança. Estes são traduzidos para a linguagem JavaScript e enviados para o cliente correspondente. As *std::threads* C++2011 serão abordadas com maior detalhe na Secção 5.4.2.

Por último, o método *stop_recognition* tem um funcionamento muito semelhante ao do *julius*. Se for invocado durante um processo de reconhecimento, fará as mesmas ações descritas para *julius*, com a diferença de que não são devolvidos quaisquer resultados.

5.3.1 Emparelhar Clientes com Objetos da Classe *Julius*

Cada instanciação da classe, apenas permite uma tarefa de reconhecimento. Por outras palavras, cada objeto da "Classe *Julius*" apenas permite um reconhecimento a ser executado ao mesmo tempo. O número máximo de reconhecimentos, em simultâneo, é igual ao número de objetos inicializados.

Como já se apresentou, os objetos são inicializados através do método *init_julius_objects* (ver Tabela 5.2), sendo este executado antes do servidor estar disponível para receber clientes. As inicializações deixam de sobrecarregar o servidor no período em que pode receber clientes. Desta forma, basta emparelhar os clientes com os objetos que estão disponíveis. Para contornar a repetição de código optou-se por declarar um *array* de objetos. Cada índice do *array* corresponde a uma instância de reconhecimento

Como referido no Capítulo 4, *client.id* é a propriedade que identifica os clientes que estão ligados ao servidor. Este representa um inteiro, sendo único para cada cliente. Por cada nova ligação estabelecida é atribuído um novo *client.id*, gerado no servidor pelo *BinaryJS*. Como cada cliente não pode fazer pedidos em simultâneo, cada pedido está associado ao *id* do cliente. Cada objeto da "Classe *Julius*" é responsável pelo reconhecimento associado a um pedido. É necessário garantir que cada cliente fique emparelhado a um e só um objeto da classe. Este vínculo deve durar desde que o cliente é aceite no servidor, até que ele o abandone. Para fazer esta correspondência, utilizou-se um *array* bidimensional, tal como é ilustrado na Figura 5.4.

Array de Objetos	Estado	client.id	Nº Bytes Passados
Julius[0]	ocupado	17	0
Julius[1]	ocupado	25	**4096**
Julius[2]	**ocupado**	**31**	0
Julius [...]	...	...	...
Julius[N]	**livre**	**-1**	0

Figura 5.4 – Interação entre Clientes e Objetos de Reconhecimento

No exemplo representado na Figura 5.4, cada linha corresponde a um objeto de reconhecimento. As colunas contêm informações relevantes, sobre a sua utilização. A primeira coluna representa o estado, indicando se o objeto está "livre" ou "ocupado". A segunda coluna guarda o *id* do cliente que é atribuído pelo *BinaryJS*. Por último, a terceira coluna guarda o número de *bytes* de áudio recebidos por cada comando de voz. O número de bytes passados é importante para perceber quando o *stream* é iniciado e para controlar a duração do comando de voz, uma vez que o *Julius* impõe um limite máximo de 320000 bytes por locução a reconhecer [46].

A Figura 5.4 representa situações práticas da utilização da matriz. Na linha referente ao *objeto Julius[1]* verificamos que este se encontra "ocupado" e emparelhado com o cliente de *id* 25. O número de bytes passados foi atualizado para o valor 4096, o que corresponde a uma situação prática de *buffering* (ver Tabela 5.2). Na linha referente ao *objeto Julius[2]* verificamos que lhe é atribuído um novo cliente com *id* 31. Este objeto transitou do estado "livre" para "ocupado". A atribuição de um cliente a um objeto é feita através do *add_commands* (ver Tabela 5.2). Finalmente, na linha referente ao *objeto Julius[N]*, está exemplificado o caso em que um cliente abandona o servidor. Ao abandonar o servidor, o estado do objeto passa de "ocupado" para livre", ficando disponível para um novo cliente. A desocupação de um objeto é feita em *stop_recognition* (ver Tabela 5.2).

5.4 Alterações Efetuadas no *Julius*

Nesta secção são abordadas as alterações efetuadas ao *Julius*, que permitiram a resolução dos problemas apontados.

5.4.1 *BinaryStream*

Para que o *addon* funcione é necessário passar-lhe um segmento de áudio como parâmetro. Contudo, o *Julius* não está preparado para consumir áudio em segmentos, a não ser quando a origem de áudio é o microfone, o que não se aplica ao caso presente. Houve a necessidade de adicionar uma nova entrada de áudio ao *Julius*.

O novo tipo de entrada designa-se por *BinaryStream*. Esta foi adicionada ao *Julius*, que passa a ter uma nova entrada de áudio, a ser consumida por segmentos: *stream* binário, além da tradicional entrada por microfone. Assim, torna-se possível ler o áudio que é passado como parâmetro. Isso é feito na função *buffering*, que gere um *buffer* de áudio circular. As alterações efetuadas não foram demasiado invasiva porque o *BinaryStream* tem essencialmente as mesmas definições que a entrada por ficheiro e STDIN. Sendo assim, as alterações efetuadas resumem-se a: i) alteração da origem do áudio (passa a ser um buffer de áudio indexado por um *id* de cliente), ii) obrigar as rotinas que consomem áudio a referenciar o *id* do *buffer*/cliente e iii) métodos associadas à entrada *BinaryStream* que gerem um *buffer* de áudio por cada pedido.

5.4.2 *Multithreading*

Até à versão C++2011 o uso de *threads* podia ser feito através de Pthreads (POSIX *threads*) ou Visual C++. Contudo, o *standard* C++2011 apresenta um conjunto de novidades. Entre elas, está um *standard* para o uso de *threads*. O objeto inicializado *std::thread* (*worker thread*) executa as suas rotinas na forma de uma função [68]. Quando se pretende finalizar uma *worker thread* é invocado o método *join* [69]. O método *join* deve ser invocado a partir de uma *thread* diferente daquela que se pretende finalizar [68]. Todas as *worker threads* têm duas propriedades importantes: i) *id* que identifica a *worker thread* e ii) *joinable* verifica se o método *join* pode ser utilizado para uma determinada *worker thread* [70]. A Figura 5.5 representa a situação prática do uso das *worker threads*. O objeto *std::thread* começa por ser inicializado, tendo como parâmetro a "*thread function*". A "*main thread*" continua a execução das suas rotinas paralelamente com a

working thread. Quando a "*main thread*" invoca o método *join* existem dois cenários possíveis: i) a *worker thread* é finalizada pois terminou as suas rotinas ou ii) a "*main thread*" espera que a *worker thread* termine as suas rotinas. As linhas a tracejado indicam a possibilidade de um estado de bloqueio que corresponde à situação descrita em ii).

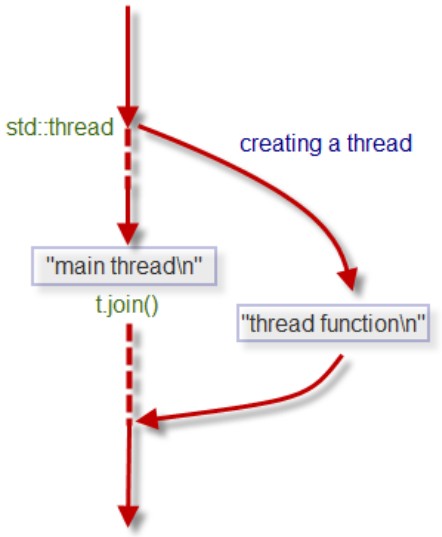

Figura 5.5 – Fluxo gerado na utilização de uma working thread [68]

A implementação do *BinaryStream*, tal como é descrita na Secção anterior, permitiu a resolução de dois problemas. Primeiro, passou a ser possível fazer processamento de áudio em tempo real. Segundo, a implementação de *handlers* e do *array* de *buffers* permite que os reconhecimentos sejam efetuados em paralelo. Desta forma, as várias *threads* não partilham variáveis entre si, mantendo-se isoladas. Para que tal acontecesse, ajudou o facto de o *Julius* ter uma quantidade mínima de variáveis globais e estáticas. O código (original) foi criado de forma a que as variáveis estejam organizadas em estruturas. Este tipo de organização permite que as *threads* não "colidam" entre si. Resumindo, os membros da "Classe *Julius*" que são relativos às estruturas do reconhecedor, são exclusivos para cada objeto inicializado. Exclusividade, neste contexto, significa que os objetos não partilham informação entre si. Para utilizar as *std:threads* basta que na sua invocação se coloquem como parâmetros: i) membro de função da "Classe Julius" que se pretende executar, ii) Objeto da "Classe Julius" e iii) parâmetros de entrada do respetivo membro de função da "Classe Julius". O membro de função da "Classe Julius" que é executado numa *std:thread* é *Audio_recog_thread* (ver Tabela 5.4). Uma vez que as *worker threads* são objetos, torna-se possível inicializar um *array* de *worker threads*. O número de elementos do *array* de *worker threads* é igual ao número de objetos *Julius*, ou seja, existe uma *worker thread* por cada objeto de reconhecimento.

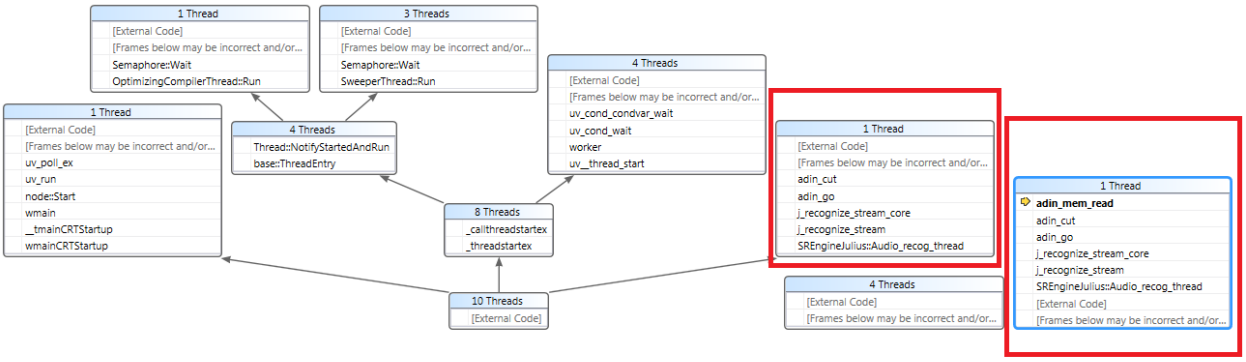

Figura 5-6 – Esquema do Parallel Stacks em Visual Studio 2013

Como foi discutido na Secção 2.2.2, o desenvolvimento do *addon* efetuou-se na ferramenta Visual Studio 2013. Fazer *debug* de *threads* era uma tarefa muito penosa. Contudo, o Visual Studio 2013 oferece uma ferramenta muito útil e que alivia esta tarefa. Tal como está ilustrado na Figura 5.6, utilizou-se a opção *Parallel Stacks*. A janela *Parallel Stacks* é muito útil quando se está a fazer *debug* de aplicações *multithreaded* [71]. Assinalado a vermelho, estão duas *threads* que correspondem ao reconhecimento de dois clientes. Desta maneira, é possível verificar que ambos os reconhecimentos se efetuam em paralelo. A utilização da janela *Parallel Stacks* pode ser explorada com maior detalhe em [71].

5.4.3 Consumo de Memória Eficiente

Em qualquer aplicação, o consumo de memória é uma questão da máxima importância. Desta forma, torna-se possível maximizar o serviço prestado. No contexto da dissertação o menor consumo de memória permite que estejam mais clientes ligados ao servidor. Na Secção 5.4.2, verificámos que os objetos inicializados são independentes entre si. Contudo, há uma quantidade de informação redundante que é comum a todos eles. O excessivo consumo de memória, provocado pela inicialização dos objetos, pode ser atenuado.

O reconhecedor implementado apenas funciona para uma língua. Este facto foi aproveitado para fazer com que o consumo de memória seja mais eficiente. Por outras palavras, todos os objetos utilizam o mesmo Modelo Acústico, Dicionário de Pronunciação e respetivo Modelo de Linguagem. Tornando esta informação global, a poupança de memória é imediata. A Tabela 5.4 compara a memória utilizada pela inicialização de quatro objetos, permitindo verificar o peso das várias componentes e a poupança global. Os cálculos da poupança foram feitos em comparação com os valores obtidos numa situação sem alterações.

Tabela 5.5- Poupança Global através do Consumo de Memória Eficiente

	Sem Alterações	Dicionário de Pronunciação e Modelo de Pronunciação Eficientes	Dicionário de Pronunciação, Modelo de Pronunciação e Modelo Acústico Eficientes
Consumo [MB]	203,72	114,172	56,448
Poupança [MB]	0	89,548	147,272
Poupança [%]	0	43,96%	72,29%

Capítulo 6
Sistema Final

O resultado final da dissertação pode ser avaliado numa página web com reconhecimento de voz remoto. O aspeto geral da página é uma questão menor, uma vez que o reconhecimento pode ser efetuado em qualquer página HTML. Para isso, criou-se o script "juliusClient.js". Este deverá ser invocado na página onde se pretende reconhecimento de comandos, sendo responsável pelas rotinas expostas no Capítulo 3. Além deste, deverão ser invocados os *scripts*: i) "*adapter.js*" (ver Secção 2.1.1), ii) "*audiolib.js*" (ver Secção 3.5) e iii) "*binary.min.js*" (ver Secção 3.6). O desenvolvedor deverá especificar: i) lista de palavras a reconhecer, ii) limiar para o VAD e iii) ações a efetuar na para cada evento de reconhecimento. Informações detalhadas sobre a implementação da página são ilustradas num manual de utilizador[3]. A página ilustrada na Figura 6.1 está disponível no endereço (6.1).

$$https://hades.co.it.pt:9000/pagina\_comandos.html \quad (6.1)$$

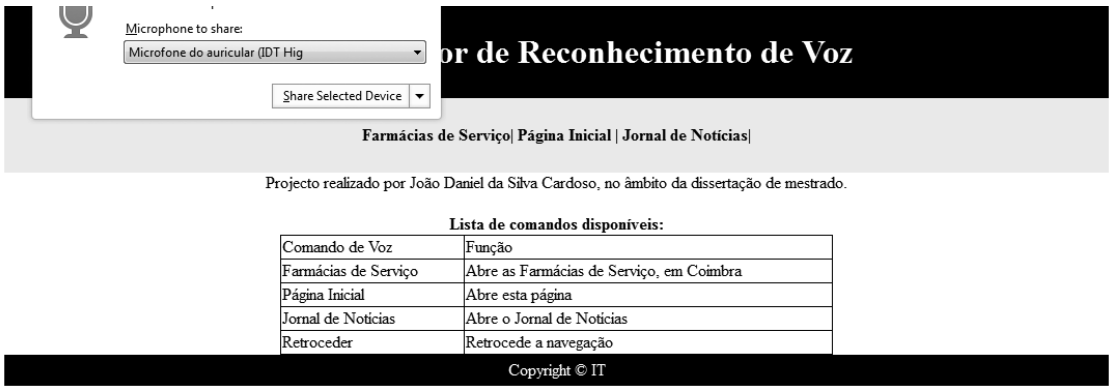

Figura 6-1 – Página de Teste para Reconhecimento de Voz

Como foi descrito na Secção 2.1, o pedido de permissão para abrir o microfone surge assim que se abre a página. Dada a permissão, basta que o utilizador dite um dos comandos possíveis. Assim que a página receba um resultado de elevada confiança, atualizar-se-á sozinha.

A latência e precisão são dados importantes neste tipo de serviço. A precisão do reconhecimento (taxa de comandos corretamente reconhecidos) depende de muitos parâmetros, como a duração do comando, tamanho da gramática, qualidade do microfone, ambiente acústico, entre outros. A latência depende de diversos fatores, como o tamanho do comando, tamanho do dicionário, qualidade da ligação do cliente e qualidade da ligação do servidor.

[3] Link: https://hades.co.it.pt:9000/manual_utilizador.pdf

Capítulo 7
Conclusão

A principal motivação desta dissertação era a de criar um sistema reconhecimento de fala compatível com as especificações do W3C. Houve a necessidade de integrar o reconhecedor de fala num servidor, já que as soluções com *plugins* estão a ser progressivamente abandonadas. Utilizou-se a MediaStream API e Web Audio API para aceder ao microfone do utilizador e às suas amostras. As amostras são enviadas para o servidor via *streaming*. O servidor implementando permite o reconhecimento simultâneo de vários clientes. Ficou disponível uma solução que permite reconhecimento de fala nos principais *browsers* de internet e que pode ser utilizada em qualquer página HTML.

A escolha do Node.js como *software* de servidor revelou ser uma boa escolha. Apesar de ser uma ferramenta muito recente, teve um impacto enorme na comunidade de desenvolvedores [28][23], de tal forma que as mais recentes investidas da Mozilla no reconhecimento de fala têm como base um servidor Node.js [4]. Assim, a solução preconizada nesta dissertação mostra-se na vanguarda dos mais recentes desenvolvimentos das tecnologias *web*.

Para trabalho futuro propõe-se que o servidor seja capaz de reconhecimento em várias línguas. Poderá também ser testada a possibilidade de utilização de outro reconhecedor de fala. Apesar de o *Julius* se demonstrar uma solução eficaz, poderá existir uma solução que se adeque melhor ao servidor Node.js. Existem várias soluções disponíveis e de código aberto, tais como: Sphinx, Kaldi e iATROS [72].

A nível pessoal, foi muito gratificante trabalhar neste projeto. Foi necessário imergir, por completo, naquilo que se tem feito na área da *web development*. É uma área de particular interesse pois é muito atual e na qual se estão a fazer grandes avanços e investimentos em investigação.

Bibliografia

[1] Wikipédia, “Speech Recognition,” 2016. [Online]. Available: https://en.wikipedia.org/wiki/Speech_recognition.

[2] Y. Dong and L. Deng, *Automatic Speech Recognition*. London: Springer-Verlag, 2015.

[3] A. de Rosa, “Introducing the Web Speech API,” 2014. [Online]. Available: http://www.sitepoint.com/introducing-web-speech-api/. [Accessed: 31-Jan-2016].

[4] MozillaWiki, “SpeechRTC,” 2015. [Online]. Available: https://wiki.mozilla.org/SpeechRTC_-_Speech_enabling_the_open_web. [Accessed: 11-Jan-2016].

[5] A. Veiga, J. Proença, and F. Perdigão, “Plugin SpeechRecoIt - Guia de Desenvolvimento,” 2013.

[6] Wikipédia, “HTML5,” 2015. [Online]. Available: https://pt.wikipedia.org/wiki/HTML5. [Accessed: 11-Jan-2016].

[7] T. C. Projects, “NPAPI deprecation: developer guide,” *2005*. [Online]. Available: https://www.chromium.org/developers/npapi-deprecation. [Accessed: 11-Jan-2016].

[8] B. Smedberg, “NPAPI Plugins in Firefox.” [Online]. Available: https://blog.mozilla.org/futurereleases/2015/10/08/npapi-plugins-in-firefox/. [Accessed: 11-Jan-2016].

[9] A. Lee and T. Kawahara, “Recent development of open-source speech recognition engine julius,” 2009.

[10] W3Schools, “Browser Statistics,” 2016. [Online]. Available: http://www.w3schools.com/browsers/browsers_stats.asp. [Accessed: 04-Feb-2016].

[11] G. Shires and H. Wennborg, “Web Speech API Specification,” 2012. [Online]. Available: https://dvcs.w3.org/hg/speech-api/raw-file/tip/speechapi.html. [Accessed: 11-Jan-2016].

[12] StackOverflow, “Grammar in Google speech API,” 2015. [Online]. Available: https://stackoverflow.com/questions/7433801/grammar-in-google-speech-api. [Accessed: 11-Jan-2016].

[13] Mozilla Developer Network, “Web Speech API - Web APIs | MDN.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API. [Accessed: 11-

Jan-2016].

[14] Wikipédia, “Adobe Flash Payer.” .

[15] Wikipédia, “Microsoft Silverlight.” [Online]. Available: https://en.wikipedia.org/wiki/Microsoft_Silverlight.

[16] S. Dutton, *Getting Started with WebRTC*. 2014.

[17] Mozilla Developer Network, “MediaDevices.getUserMedia() - Web APIs | MDN,” 2015. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia#Browser_compatibility. [Accessed: 11-Jan-2016].

[18] W3C, “Media Capture and Streams,” 2015. [Online]. Available: https://www.w3.org/TR/mediacapture-streams/#introduction. [Accessed: 11-Jan-2016].

[19] B. Smus, “Getting Started with Web Audio API - HTML5 Rocks,” 2011. [Online]. Available: http://www.html5rocks.com/en/tutorials/webaudio/intro/. [Accessed: 11-Jan-2016].

[20] “Can I use - Web Audio API,” 2015. [Online]. Available: http://caniuse.com/#search=web audio api. [Accessed: 11-Jan-2016].

[21] B. Smus, *Web Audio API*, First Edit. 2013.

[22] Wikipédia, “Front-end e back-end,” 2015. [Online]. Available: https://pt.wikipedia.org/wiki/Front-end_e_back-end#Front-end. [Accessed: 12-Jan-2016].

[23] G. Rauh, *Smashing Node.js*, vol. 53. Wiley, 2012.

[24] Google Developers, “Chrome V8,” 2015. [Online]. Available: https://developers.google.com/v8/. [Accessed: 12-Jan-2016].

[25] “Node.js,” *Wikipedia*. 2015.

[26] “libuv API documentation,” 2016. [Online]. Available: http://docs.libuv.org/en/v1.x/. [Accessed: 14-Feb-2016].

[27] TutorialsPoint, “Node.js Event Loop,” 2015. [Online]. Available: http://www.tutorialspoint.com/nodejs/nodejs_event_loop.htm. [Accessed: 28-Dec-2015].

[28] P. Teixeira, *Professional Node.js*. Wiley, 2012.

[29] T. Capan, “Why Use Node.js? A Comprehensive Tutorial,” 2014. [Online]. Available:

http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js. [Accessed: 13-Jan-2016].

[30] I. Gerchev, “Demystifying JavaScript Closures, Callbacks and IIFEs,” 2015. [Online]. Available: http://www.sitepoint.com/demystifying-javascript-closures-callbacks-iifes/. [Accessed: 09-Dec-2015].

[31] R. Bovell, “Understand JavaScript Callback Functions and Use Them,” 2015. [Online]. Available: http://javascriptissexy.com/understand-javascript-callback-functions-and-use-them/. [Accessed: 09-Dec-2015].

[32] Wikipédia, “npm,” 2016. [Online]. Available: https://en.wikipedia.org/wiki/Npm_(software).

[33] BinaryJS, “BinaryJS - Realtime binary streaming for the web using websockets,” 2013. [Online]. Available: http://binaryjs.com/. [Accessed: 10-Dec-2015].

[34] BinaryJS, “BinaryJS API Reference,” 2013. [Online]. Available: https://github.com/binaryjs/binaryjs/blob/master/doc/api.md.

[35] S. Frees, “Getting your C++ to the Web with Node.js.” [Online]. Available: http://blog.scottfrees.com/getting-your-c-to-the-web-with-node-js. [Accessed: 12-Dec-2015].

[36] S. Frees, “C++ processing from Node.js - Part 4 - Asynchronous addons,” 2015. [Online]. Available: http://blog.scottfrees.com/c-processing-from-node-js-part-4-asynchronous-addons. [Accessed: 13-Jan-2016].

[37] “Addons Node.js v0.12.9 Manual & Documentation.” [Online]. Available: https://nodejs.org/docs/latest-v0.12.x/api/addons.html. [Accessed: 09-Dec-2015].

[38] E. Khvedchenya, “How to debug node.js addons in Visual Studio,” 2015. [Online]. Available: http://computer-vision-talks.com/how-to-debug-nodejs-addons-in-visual-studio/. [Accessed: 05-Jan-2016].

[39] Wikipédia, “Language Model,” 2015. [Online]. Available: https://en.wikipedia.org/wiki/Language_model. [Accessed: 16-Jan-2015].

[40] J. Leyons, “Mel Frequency Cepstral Coefficient (MFCC).” [Online]. Available: http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/. [Accessed: 16-Jan-2016].

[41] O. Costa, “Desenvolvimento de Técnicas para a Avaliação Automática da Capacidade de Leitura das Crianças,” Universidade de Coimbra, 2015.

[42] L. Rabiner and B.-H. Juang, *Fundamentals of Speech Recognition*. Prentice-Hall Internacional, Inc., 1993.

[43] G. E. Dahl, D. Yu, L. Deng, and A. Acero, “Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition,” *IEEE Trans. Audio, Speech Lang. Process.*, vol. 20, no. 1, pp. 30–42, 2012.

[44] S. Young, G. Evermann, M. Gales, T. Hain, and G. Moore, *The HTK Book*. Cambridge University Engineering, 2009.

[45] Wikipédia, “Hidden Markov Models,” 2015. [Online]. Available: https://en.wikipedia.org/wiki/Hidden_Markov_model.

[46] A. Lee, *The Julius book*. 2010.

[47] “Media Capture and Streams.” [Online]. Available: http://www.w3.org/TR/2013/WD-mediacapture-streams-20130903/#navigatorusermedia. [Accessed: 18-Dec-2015].

[48] Mozilla Developer Network, “Promise - JavaScript | MDN,” 2015. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. [Accessed: 18-Dec-2015].

[49] The Chromium Projects, “Marking HTTP As Non-Secure.” [Online]. Available: https://www.chromium.org/Home/chromium-security/marking-http-as-non-secure. [Accessed: 18-Dec-2015].

[50] The Chromium Projects, “Deprecating Powerful Features on Insecure Origins,” 2015. [Online]. Available: https://www.chromium.org/Home/chromium-security/deprecating-powerful-features-on-insecure-origins. [Accessed: 18-Dec-2015].

[51] Node.js, “HTTPS Node.js v5.4.1 Manual & Documentation,” 2015. [Online]. Available: https://nodejs.org/api/https.html. [Accessed: 17-Jan-2016].

[52] Wikipédia, “OpenSSL,” 2015. [Online]. Available: https://pt.wikipedia.org/wiki/OpenSSL. [Accessed: 18-Dec-2015].

[53] M. Khan, “RecordRTC.” [Online]. Available: https://github.com/muaz-khan/WebRTC-Experiment/tree/master/RecordRTC.

[54] Mozilla Developer Network, “ScriptProcessorNode,” 2015. [Online]. Available:

https://developer.mozilla.org/en-US/docs/Web/API/ScriptProcessorNode. [Accessed: 19-Dec-2015].

[55] Mozilla Developer Network, “AnalyserNode,” 2015. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/AnalyserNode. [Accessed: 18-Jan-2016].

[56] Mozilla Developer Network, “AudioDestinationNode,” 2015. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/AudioDestinationNode. [Accessed: 18-Jan-2016].

[57] C. Wilson, “sampleRate WebAudioAPI,” 2015. [Online]. Available: https://stackoverflow.com/questions/34066845/webaudioapi-change-samplerate-in-windows-definitions. [Accessed: 18-Jan-2016].

[58] Wikipédia, “Voice activity detection,” 2016. [Online]. Available: https://en.wikipedia.org/wiki/Voice_activity_detection. [Accessed: 18-Jan-2016].

[59] Wikipédia, “44100Hz,” 2015. [Online]. Available: https://en.wikipedia.org/wiki/44,100_Hz#Status. [Accessed: 18-Jan-2016].

[60] J. Kalliokoski, “audiolib.js,” 2015. [Online]. Available: https://github.com/jussi-kalliokoski/audiolib.js. [Accessed: 18-Jan-2016].

[61] W3C, “Web Audio API,” 2015. [Online]. Available: https://www.w3.org/TR/webaudio/. [Accessed: 18-Jan-2016].

[62] Mozilla Developer Network, “Int16Array.” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Int16Array. [Accessed: 24-Dec-2015].

[63] Wikipédia, “Escalabilidade,” 2016. [Online]. Available: https://pt.wikipedia.org/wiki/Escalabilidade. [Accessed: 16-Jan-2016].

[64] Wikipédia, “Scalability,” 2016. [Online]. Available: https://en.wikipedia.org/wiki/Scalability. [Accessed: 16-Jan-2016].

[65] StackOverflow, “c++ - Did join() frees allocated memory? - C++11 Threads - Stack Overflow,” 2016. [Online]. Available: http://stackoverflow.com/questions/35136548/did-join-frees-allocated-memory-c11-threads?noredirect=1#comment57993210_35136548. [Accessed: 01-Feb-2016].

[66] StackOverflow, “node.js - When should be used Synchronous Function in Nodejs? - Stack Overflow,” 2016. [Online]. Available: http://stackoverflow.com/questions/35118248/when-should-be-used-synchronous-function-in-nodejs. [Accessed: 01-Feb-2016].

[67] Instituto de Telecomunicações Laboratório de Processamento de Sinal, “Grafone.” .

[68] Bogotobog, “C++ Tutorial: C++11/C++14 Thread 1. Creating Threads - 2016,” 2016. [Online]. Available: http://www.bogotobogo.com/cplusplus/C11/1_C11_creating_thread.php. [Accessed: 02-Feb-2016].

[69] cplusplus, “thread::join.” [Online]. Available: http://www.cplusplus.com/reference/thread/thread/join/. [Accessed: 19-Jan-2016].

[70] cplusplus, “thread - C++ Reference.” [Online]. Available: http://www.cplusplus.com/reference/thread/thread/. [Accessed: 02-Feb-2016].

[71] Microsoft Developer Network, “Using the Parallel Stacks Window,” 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/dd998398.aspx. [Accessed: 02-Feb-2016].

[72] Wikipédia, “List of Speech Recognition Software,” 2015. [Online]. Available: https://en.wikipedia.org/wiki/List_of_speech_recognition_software. [Accessed: 02-Feb-2016].

Anexo A

Modelo de Linguagem

Como modelo de linguagem, foi utilizada uma gramática baseada em regras e restrições. Esta é uma de três opções disponíveis no *Julius* [9]. Este é um formato próprio do *Julius* e as ferramentas de criação encontram-se disponíveis na sua distribuição [46]. A gramática consiste em dois ficheiros. O primeiro contém a lista de palavras, divididas por tipo, juntamente com a sua pronunciação. Este tem a extensão *.dict*, abreviação de *dictionary file*. O segundo ficheiro descreve uma máquina de estados finita com a extensão *.dfa* (*deterministic automation file*) e define a forma como os vários tipos de resultados se podem interligar [46].

O modelo de linguagem foi criado no contexto do projeto Tice.Healthy, sendo aproveitado para esta dissertação. A razão é simples: ambos os projetos pretendem reconhecimento de comandos de voz.

```
0       [<s>]   sil
1       [</s>]  sil
2       [lixo1] garbage
2       [lixo2] lixo_vpv
2       [lixo3] lixo_sp
2       [lixo4] lixo_vogais
3       [palavra] pronunciação
```

Figura A.1 – Ficheiro .dict utilizado

Na Figura A.1 está o esboço do ficheiro *.dict* utilizado. Nela verifica-se que ao silêncio inicial e final foram atribuídos o tipo 0 e 1. As várias possibilidades de lixo (fala indistinguível, ruído intermitente, ruído constante e vogais estendidas) são do tipo 2. As palavras são adicionadas dinamicamente, via *DynamicAddCommands*, e serão do tipo 3.

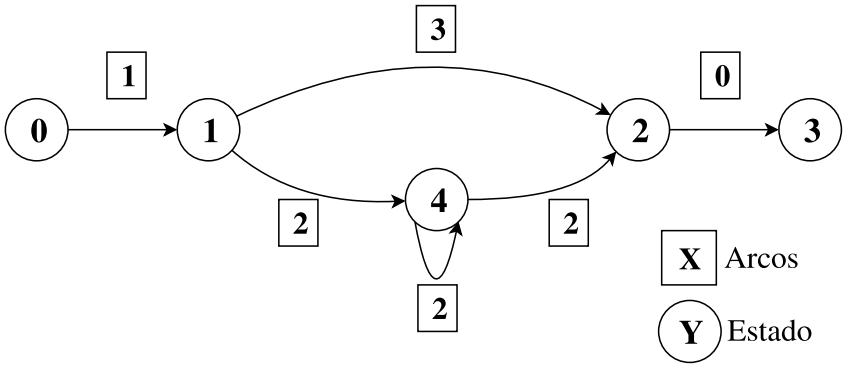

Figura A.2 – Esquema da Gramática .dfa utilizada para reconhecimento de comandos

A Figura A.2 mostra-nos a máquina de estados implementada para o LM. A transição dos estados 0-1 e 2-3 correspondem ao silêncio inicial e final, respetivamente. A partir do estado 1, existem duas possibilidades: passar pelo arco 3 (devolvendo o comando reconhecido) ou seguir para o estado 4 (devolvendo os tipos de lixo reconhecidos, que podem ser consecutivamente repetidos). Através da análise da máquina de estados fica claro que apenas uma palavra pode ser devolvida. Tal acontece, porque foi criada para o reconhecimento de comandos.