UNIVERSIDADE FEDERAL DO PARÁ

INSTITUTO DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Avanços em Reconhecimento de Fala para Português Brasileiro e Aplicações: Ditado no LibreOffice e Unidade de Resposta Audível com Asterisk

Pedro dos Santos Batista

UFPA / ITEC / PPGEE

Campus Universitário do Guamá

Belém - Pará - Brasil

2013

UNIVERSIDADE FEDERAL DO PARÁ

INSTITUTO DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Avanços em Reconhecimento de Fala para Português Brasileiro e Aplicações: Ditado no LibreOffice e Unidade de Resposta Audível com Asterisk

Autor: Pedro dos Santos Batista

Orientador: Aldebaro Barreto da Rocha Klautau Júnior

Dissertação submetida à Banca Examinadora do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Pará para obtenção do Grau de Mestre em Engenharia Elétrica. Área de concentração: **Telecomunicações**.

UFPA / ITEC / PPGEE

Campus Universitário do Guamá

Belém, PA

2013

UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Avanços em Reconhecimento de Fala para Português Brasileiro e Aplicações: Ditado no LibreOffice e Unidade de Resposta Audível com Asterisk

AUTOR(A): PEDRO DOS SANTOS BATISTA

DISSERTAÇÃO DE MESTRADO SUBMETIDA À AVALIAÇÃO DA BANCA EXAMINADORA APROVADA PELO COLEGIADO DO PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA, DA UNIVERSIDADE FEDERAL DO PARÁ E JULGADA ADEQUADA PARA A OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA ELÉTRICA NA ÁREA DE TELECOMUNICAÇÕES.

Prof. Dr. Aldebaro Barreto da Rocha Klautau Júnior
(Orientador - UFPA / ITEC)

Prof. Dr. Nelson Cruz Sampaio Neto
(Coorientador - UFPA / ICEN)

Prof. Dr. Dionne Cavalcante Monteiro
(Membro - UFPA / ICEN)

Profa. Dra. Yomara Pinheiro Pires
(Membro - UFPA / Castanhal)

Agradecimentos

Agradeço primeiramente à minha família, que esteve sempre ao lado. Em especial a minha mãe Rosilda Batista e meu pai Alfonço Batista, por me mostrar o caminho até aqui, e pelo apoio incondicional em todos os momentos da minha vida. Agradeço meus irmãos, tios, tias e primos que estiveram sempre ao lado.

Gostaria de agradecer também a família da Elielza Reis, pelo apoio, sem o qual essa caminhada seria um tanto quanto mais difícil.

Não poderia deixar de agradecer meu orientador, professor, conselheiro, terapeuta e amigo Aldebaro Klautau e meu coorientador Nelson Neto, pelos inúmeras conselhos e total apoio no decorrer desse trabalho. Agradeço todos os professores da EngComp e do PPGEE, que construíram a base que precisei para desenvolver esse trabalho.

No trilhar desse caminho tive inúmeros amigos, os quais não me atrevo a citar nomes, pois não sei se teria espaço suficiente, em especial o pessoal de Concórdia, por nunca deixar de lado nossa amizade.

Agradeço também a todo o pessoal do LaPS, que como costumamos dizer é nossa segunda família. Cujo apoio foi essencial em todos os momentos desse trabalho.

Gostaria também de agradecer a comunidade *open source* pelo ótimo trabalho, disponibilizando recursos para que trabalhos como este sejam possíveis.

Sou grato também as agências governamentais financiadoras de pesquisa pelo investimento feito em mim. Assim como empresas colaboradoras do LaPS.

A banca examinadora sou grato, pelo tempo investido na avaliação deste trabalho. Ajudando a aprimorá-lo, e melhor contribuir com a comunidade acadêmica.

Dedico este trabalho aos meus pais,

e meus amigos

que torceram pelo meu sucesso...

Resumo

O reconhecimento automático de voz vem sendo cada vez mais útil e possível. Quando se trata de línguas como a Inglesa, encontram-se no mercado excelentes reconhecedores. Porém, a situação não é a mesma para o Português Brasileiro, onde os principais reconhecedores para ditado em sistemas *desktop* que já existiram foram descontinuados. A presente dissertação alinha-se com os objetivos do Laboratório de Processamento de Sinais da Universidade Federal do Pará, que é o desenvolvimento de um reconhecedor automático de voz para Português Brasileiro. Mais especificamente, as principais contribuições dessa dissertação são: o desenvolvimento de alguns recursos necessários para a construção de um reconhecedor, tais como: bases de áudio transcrito e API para desenvolvimento de aplicações; e o desenvolvimento de duas aplicações: uma para ditado em sistema *desktop* e outra para atendimento automático em um *call ceter*. O Coruja, sistema desenvolvido no LaPS para reconhecimento de voz em Português Brasileiro. Este, além de conter todos os recursos para fornecer reconhecimento de voz em Português Brasileiro possui uma API para desenvolvimento de aplicativos. O aplicativo desenvolvido para ditado e edição de textos em *desktop* é o SpeechOO, este possibilita o ditado para a ferramenta Writer do pacote LibreOffice, além de permitir a edição e formatação de texto com comandos de voz. Outra contribuição deste trabalho é a utilização de reconhecimento automático de voz em *call centers*, o Coruja foi integrado ao *software* Asterisk e a principal aplicação desenvolvida foi uma unidade de resposta audível com reconhecimento de voz para o atendimento de um *call center* nacional que atende mais de 3 mil ligações diárias.

Abstract

Automatic speech recognition has been increasingly more useful and feasible. When it comes to languages such as English, there are excellent speech recognizers available. However, the situation is not the same for Brazilian Portuguese, where the few recognizers for desktop dictation that existed, are no longer available. This dissertation is aligned with a goal of the Signal Processing Laboratory at the Federal University of Pará, which is the development of a complete automatic speech recognizer for Brazilian Portuguese. More specifically, the main contributions of this dissertation are: the development of some resources needed to build a speech recognizer such as transcribed audio database and speech API; and the development of two applications: one for desktop dictation and another for automatic service in a call center. The system developed in-house for automatic speech recognition in Brazilian Portuguese is called Coruja, and besides all the resources that makes automatic speech recognition in Brazilian Portuguese available, the Coruja also contains an API for application development using speech recognition. The application for desktop dictation is called SpeechOO. The SpeechOO enables dictation and text editing and formatting by voice for the LibreOffice Writer. Other contribution of this work is the use of Coruja in call centers. Coruja was integrated with the Asterisk software, which is the main open source software for call centers. The main application developed for automated service in call center was an interactive voice response which is deployed nationally and receives more than 3 thousand daily calls.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

AGI *Asterisk Gateway Interface*

AM *Acoustic Model*

ARPA U.S. *Department of Defense Advanced Research Project Agency*

ASR *Automatic Speech Recognition*

API *Application Programming Interface*

BSD *Berkeley Software Distribution*

CAPES Coordenação de Aperfeiçoamento de Pessoal de Nível Superior

CETUC Centro de Estudos em Telecomunicações

CLI *Common Language Infrastructure*

CLR *Common Language Runtime*

CMU Carnegie Mellon *University*

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico

COM *Component Object Model*

DDD Discagem Direta à Distância

DDN Disque Denuncia Nacional

EAGI *Extended* Asterisk *Gateway Interface*

FISL Fórum Internacional de Software Livre

FXS *Foreign eXchange Subscriber*

GSM *Global System for Mobile Communications*

HMM *Hidden* Markov *Model*

HTK *Hidden* Markov *Model Toolkit*

IBGE Instituto Brasileiro de Geografia Estatística

ICEN Instituto de Ciências Exatas e Naturais

IEEE *Institute of Electrical and Electronics Engineers*

IP *Internet Protocol*

ISBN *International Standard Book Number*

ITEC Instituto de Tecnologia

IVR *Interactive Voice Response*

JNI Java *Native Interface*

JSAPI Java *Speech Programming Interface*

JSGF Java *Speech Grammar Format*

JVM Java *Virtual Machine*

LM *Language Model*

LVCSR *Large Vocabulary Continuous Speech Recognition*

MAP *Maximum A Posteriori*

MFCC *Mel-Frequency Cepstral Coefficients*

MLLR *Maximum Likelihood Linear Regression*

NLP *Natural Language Processing*

OGI Oregon *Graduate Institute*

PB Português Brasileiro

PBX *Private Branch eXchange*

PCI *Peripheral Component Interconnect*

PPGEE Programa de Pós-Graduação em Engenharia Elétrica

PROPOR *International Conference on Computational Processing of the Portuguese Language*

PSTN *Public Switched Telephone Network*

xRT *Real-time factor*

SAPI *Speech Application Programming Interface*

SIP *Session Initiation Protocol*

SDK *Software Development Kit*

SR *Speech Recognition*

SRI *Stanford Research Institute*

SRILM *The* SRI *Language Modeling Toolkit*

TTS *Text to Speech*

UFPA Universidade Federal do Pará

UNO *Universal Network Objects*

URL *Unified Resource Locator*

WER *Word Error Rate*

XML *Extensible Markup Language*

Capítulo 1

Introdução

A interface homem-máquina encontra-se cada vez mais amigável. O que antigamente era feito através de cartões perfurados, hoje é possível através do que conhecemos como teclados (um conjunto de teclas pré-definidas), *mouses* e a tecnologia *touch screen*, onde a tela mutável é sensível ao toque. Consideráveis esforços vêm sendo reunidos para melhorar cada vez mais essa comunicação, visando tornar realidade o que é mostrado em filmes de ficção científica, onde uma máquina é capaz de se comunicar (através da fala), pensar e agir como um ser humano.

Acredita-se que através do reconhecimento automático de voz (ASR) e da síntese de voz (TTS) seja possível alcançar essa tão sonhada interação rápida e prática com a máquina. Sistemas TTS [2] são módulos de *software* que convertem texto em linguagem natural em fala sintetizada. ASR [3] pode ser visto como o processo inverso ao TTS, no qual o sinal de fala digitalizado, capturado por exemplo por um microfone, é convertido em texto. Várias são as aplicações que podem usar a voz como interface, principalmente dentro das áreas de acessibilidade e automação.

Atualmente, existem no mercado alguns bons sintetizadores de voz com suporte ao Português Brasileiro (PB), como é o caso do eSpeak [4]. No entanto, quando se fala em ASR, não se tem a mesma disponibilidade. Os principais sistemas ASR para ditado em *desktop* desenvolvidos com suporte ao PB foram o ViaVoice da IBM [5] e o FreeSpeech 2000 da Nuance [6], ambos descontinuados. Em contrapartida, quando se fala em sistemas ASR para outras línguas, como a inglesa, existem reconhecedores que possuem excelente desempenho, entre eles podemos citar o Dragon Naturally Speaking [7] e o Windows Speech Recognition [8].

Visando contribuir para o desenvolvimento de aplicativos utilizando reconhecimento de voz em PB, este trabalho apresenta uma interface de programação de aplicativos (API) para o reconhecedor automático de voz para PB desenvolvido no Laboratório de Processamento

de Sinais da Universidade Federal do Pará (UFPA). O sistema proposto é chamado Coruja. É importante salientar que o Coruja é livre, ou seja, programadores poderão usar ASR em suas aplicações sem nenhum custo. Outra solução proposta neste trabalho consiste na integração entre o *software* Asterisk e o Coruja, possibilitando o desenvolvimento de sistemas de atendimento automático em *call centers* com um módulo de reconhecimento de voz baseado no Coruja. Por fim, dois aplicativos de voz foram desenvolvidos: o primeiro é uma extensão que permite ditado para a ferramenta Writer do pacote LibreOffice chamada SpeechOO; e o segundo é um sistema de atendimento automático de um *call center* baseado na solução proposta neste trabalho. Esses aplicativos usam o Coruja para ASR em PB. As próximas seções discutem as principais motivações que nortearam a elaboração desta dissertação e trazem um resumo das atividades em processamento de voz no âmbito da UFPA.

1.1 Motivação

Duas grandes áreas motivaram este trabalho, uma delas é a acessibilidade que utiliza ASR de forma exaustiva, já que este é visto como uma das melhores soluções nos casos onde possa ser utilizado. A outra área são os *call centers* já que por meio de ASR os tradicionais menus solicitando a escolha do usuário por meio de dígitos são quase sempre evitados, adicionando mais naturalidade e eficiência no atendimento eletrônico. A seguir algumas seções são dedicadas a essas discussões.

1.1.1 Acessibilidade

A performance dos computadores pessoais vem evoluindo com a produção de processadores cada vez mais velozes, fato que possibilita, por exemplo, a utilização de processamento de voz em tarefas de ensino assistidas por computador. Existem várias tecnologias de voz que são efetivas no campo da educação, entre as quais TTS e ASR são as mais proeminentes, como pode ser visto em trabalhos realizados na área [9, 10]. Entretanto, todos esses projetos só foram possíveis através do uso de *engines*, que são módulos de *softwares* dependentes de língua que executam ASR ou TTS.

A incorporação de tecnologias computacionais no processo educacional tem gerado poderosas aplicações multimídia para aprendizagem assistida por computador [11]. A internet também tornou-se uma importante ferramenta para ensino e fonte de consulta [12]. Contudo, o custo financeiro e a pouca disponibilidade de computadores e *softwares* são dois dos principais obstáculos para o aprendizado baseado em computadores, especialmente em países em

desenvolvimento como o Brasil [13].

A situação é ainda mais complicada quando falamos de pessoas com necessidades especiais, incluindo visuais, auditivas, físicas, de fala e cognitiva. Eles encontram sérias dificuldades para ter acesso a esse tipo de tecnologia e sobretudo ao conhecimento. Por exemplo, de acordo com o censo realizado em 2010 pelo Instituto Brasileiro de Geografia e Estatística (IBGE), 14,5% da população brasileira tem algum tipo de necessidade especial, como detalhado na Tabela 1.1. Dessa forma é importante compreender como tecnologias de voz podem ajudar na educação e inclusão digital de pessoas com necessidades especiais [10].

Deficiência	Descrição	Número de pessoas
Visual	incluindo cegos, baixa visão, e baixa percepção de cores	16.644.842
Auditora	incluindo dificuldade auditória parcial ou total	5.735.099
Física	dificuldade motora, pode incluir fraqueza e limitação do controle muscular (como movimentos involuntários, falta de coordenação motora, ou paralisia)	9.355.844
Mental	incluindo dificuldade cognitiva e neurológica	2.844.936

Tabela 1.1: Perfil da população brasileira com deficiências, baseado em dados de [1] (a população total brasileira é 190.732.694).

Diante deste contexto, a disponibilização de *engines* com bom desempenho é fundamental para o uso efetivo de aplicações baseadas em voz em tarefas educacionais e sistemas assistivos [14, 15]. Além disso, o custo dos programas de computador e equipamentos não podem ser inviáveis. Assim, este trabalho propõe um conjunto de ferramentas e recursos livres para desenvolvimento de aplicativos com interface aural em PB, tendo a disponibilização e disseminação do conhecimento para as pessoas com deficiência como principal incentivo. As contribuições desta pesquisa assumem um papel importante na diminuição do abismo digital existente entre os falantes de línguas comercialmente atraentes e as sub-representadas, como

o Português Brasileiro.

1.1.2 *Call centers*

Em 2012 foi traçada uma parceria entre o Grupo FalaBrasil e a empresa Comunix, sediada em Brasília. A Comunix trabalha com soluções de *call centers* usando o *software* Asterisk e tem interesse em integrar ASR em seus produtos.

Ainda em 2012 um projeto foi iniciado para adaptar os recursos de ASR do grupo para os canais de telefonia e integrá-los com o *software* Asterisk. Assim começaram os trabalhos do Grupo FalaBrasil para atendimento eletrônico automático em *call centers* com reconhecimento de voz em PB.

Tradicionalmente o atendimento eletrônico em *call centers* é feito por meio de menus, onde é fornecida ao usuário uma série de opções associadas a números, e este escolhe a opção desejada discando o número correspondente. Por exemplo se em um determinado atendimento é necessário saber se o usuário quer falar com o setor de recursos humanos ou de suporte, o sistema pode falar o seguinte *prompt*: "para falar com o setor de recursos humanos digite 1, para falar com suporte digite 2", o usuário então pode escolher a opção desejada discando o número correspondente.

Uma das principais motivações para usar ASR em atendimento eletrônico é a naturalidade da interação com usuários, além de existirem situações onde seria inviável a solução por métodos tradicionais. Um exemplo de aplicação onde é inviável o uso de menus tradicionais é a venda de passagens aéreas, pois se em um determinado momento o usuário precisa escolher seu destino, seria inviável listar todos os possíveis destinos e pedir para o usuário discar o número correspondente. Para isso pode-se usar ASR e o usuário simplesmente fala o destino desejado. O que torna o atendimento mais natural e eficiente.

Dessa forma, o Grupo FalaBrasil tem trabalhado com o intuito de prover recursos para o suporte a reconhecimento de voz em *call centers* usando o Asterisk.

1.2 Resumo das atividades em processamento de voz na UFPA

O primeiro esforço do Grupo FalaBrasil no Laboratório de Processamento de Sinais da UFPA em termos de recursos a serem disponibilizados foi a criação do UFPAdic versão 1, um

dicionário fonético para PB com 11.827 palavras transcritas manualmente [16]. Esse conjunto de transcrições foi então utilizado como base de treino de um módulo de conversão grafema-fone (G2P), capaz de extrair automaticamente regras de transcrição de um léxico através de técnicas de aprendizado de máquina por indução (árvore de decisão) e Naïve Bayes. O resultado foi a criação e disponibilização de um novo dicionário fonético de grande vocabulário para PB com aproximadamente 60 mil palavras, chamado UFPAdic versão 2.

Em [17], buscou-se criar um sistema de referência utilizando as bases de voz: Spoltech [18] e OGI-22 [19]. O principal objetivo foi desenvolver e disponibilizar recursos para PB, a fim de criar *baselines* que permitam a reprodução dos resultados em diferentes locais. Para isso, são disponibilizados recursos (modelos acústicos e de linguagem) e resultados atualizados na página do Grupo FalaBrasil[1]. O dicionário UFPAdic 2 foi usado para a construção dos modelos ocultos de Markov (HMMs) a partir dos *corpus* OGI-22 e Spoltech, totalizando 32 fones e um modelo de silêncio. Em seguida, modelos trifone foram construídos a partir dos modelos monofone e uma árvore binária de decisão fonética foi elaborada para vincular os modelos trifone com as mesmas características. Nos experimentos realizados, após um trabalho de correção dos *corpora*, a base de treino do Spoltech foi composta por 5.246 arquivos de áudio, que correspondem a 180 minutos, e a base de teste com os restantes 2.000 arquivos, que totalizaram 40 minutos de gravação. Já a base de treino do OGI-22 consistiu de 2.017 arquivos de áudio, correspondendo a 184,5 minutos de gravação, e o conjunto de teste com 209 arquivos, totalizando 14 minutos.

Com a finalidade de avaliar o modelo acústico do OGI-22, o primeiro experimento consistiu na construção de modelos de linguagem bigrama utilizando as próprias frases do *corpus* OGI-22 (ou seja, uma gramática viciada). O número de Gaussianas por mistura foi gradualmente incrementado até o total de 10 componentes (limite de decréscimo da taxa de erro). A taxa de erro por palavras (WER) foi de 21% para 3.285 palavras. Da mesma forma, para avaliar o modelo acústico do Spoltech, modelos de linguagem bigrama foram construídos apenas com as frases dessa base. A WER com 10 componentes de Gaussianas por mistura foi 18,6% para 793 palavras. Em um segundo momento, frases da base de texto CETENFolha [20] foram formatadas e incluídas na construção dos modelos de linguagem. Primeiramente, para o treino dos modelos de linguagem, foram selecionadas e fixadas 32.100 frases das bases OGI-22 e CETENFolha. O vocabulário foi aos poucos incrementado com as palavras mais frequentes no conjunto de treino. Assim, vários modelos bigrama foram construídos. Usando o modelo acústico do *corpus* OGI-22 com 10 componentes de Gaussianas por mistura, a melhor WER encontrada para 30.000 palavras foi 35,87%.

[1] http://www.laps.ufpa.br/falabrasil

Modelos trigrama e técnicas de suavização foram usadas em [21] para construção de modelos de linguagem mais elaborados. O modelo acústico do OGI-22 apresentou uma WER de 23,22% e 20,40% para bigramas e trigramas viciados e suavizados via técnica Witten-Bell, respectivamente. Da mesma forma, modelos de linguagem n-grama foram construídos apenas com as frases da base Spoltech, com vocabulário de 1.104 palavras. A WER com 14 componentes de Gaussianas por mistura foi 6,13% e 5,27% para bigramas e trigramas, respectivamente, apresentando consideráveis melhoras em relação os resultados obtidos em [17].

Outro experimento, realizado em [21], consistiu em avaliar o desempenho do sistema variando o número de sentenças usadas para o treino dos modelos de linguagem e, dessa vez, completamente livres de contexto, ou seja, usando apenas frases do *corpus* CETENFolha. Assim, bigramas e trigramas, com suavização Kneser-Ney, foram elaboradas variando o número de frases para treino entre 10.000 e 180.000 e o vocabulário mantido constante, contendo apenas as palavras presentes no OGI-22. A melhor taxa de erro por palavra usando o modelo acústico do *corpus* OGI-22 com 10 componentes de Gaussianas por mistura foi 47,39% para o sistema com trigrama, treinado com 180.000 sentenças.

Em [22], contribui-se com um algoritmo baseado em regras para conversão G2P com determinação de vogal tônica para PB. A estrutura de regras utilizada baseou-se nas regras G2P descritas em [23], sendo que algumas correções e adições foram propostas. Para avaliar a eficácia das regras G2P com determinação de vogal tônica, foi construído um dicionário fonético composto pelas 679 palavras presentes no *corpus* West Point [24]. Em seguida, esse dicionário foi usado para treinar um modelo acústico com 38 fones. Para efeito de comparação, o mesmo procedimento foi usado para elaborar outros dois modelos acústicos. O segundo modelo desenvolvido usou um dicionário fonético fiel às regras fonológicas descritas em [23], ou seja, sem considerar as alterações propostas por [22]. Já o último modelo acústico, composto por 34 HMMs, empregou o dicionário UFPAdic 2. Os melhores resultados foram obtidos com os sistemas que utilizaram um dicionário baseado em regras. Contudo, diante de um modelo de linguagem independente do contexto, ficou mais nítido que as mudanças sugeridas às regras de [23] melhoraram a performance do reconhecedor.

Até então, o *corpus* de texto CETENFolha e as transcrições ortográficas das bases de voz disponíveis eram utilizados para construir os modelos de linguagem usados nos experimentos. Visto que o CETENFolha não é um *corpus* tão recente, sua utilização para a tarefa de reconhecimento de voz em tempo real fica prejudicada. Assim, textos recentes de outros jornais foram adicionados ao *corpus*. Isso foi feito através de um processo totalmente automatizado de coleta e formatação diária de três jornais, disponíveis na internet. Aproximadamente dois meses de textos coletados e processados encontram-se disponíveis na página do Grupo FalaBrasil, que correspondem a aproximadamente 120 mil frases.

Dentre as dificuldades encontradas em se produzir grandes *corpora*, tem-se a coleta de dados (voz) e transcrição ortográfica. Visando contornar tal problema em [25], foi construído um novo *corpus* de voz baseado em *audiobooks*. Os *audiobooks* são livros falados disponíveis na internet. Com os arquivos de áudio e suas respectivas transcrições (livros) tem-se uma redução considerável na tarefa de produção de um *corpus*. Inicialmente, foram obtidos 5 livros com aproximadamente 1 hora de duração cada. Os arquivos de áudio foram re-amostrados para 22050 Hz com 16 bits por amostra, em seguida os mesmos foram divididos em arquivos menores, com 30 segundos de duração cada, e por fim transcritos. O *corpus* foi composto por 2 locutores do sexo masculino, e 3 do sexo feminino, totalizando 5 horas e 19 minutos de áudio. Um problema encontrado na utilização de *audiobooks* foi o ambiente de gravação, que é bastante controlado, sendo assim, os arquivos não possuem qualquer tipo de ruído, diferente dos arquivos de teste real, onde o ambiente acústico não é controlado. Contudo tal problema pode ser contornado com técnicas de adaptação ao ambiente [25].

Ainda em [25] com o intuito de obter uma boa avaliação de desempenho e possibilitar a comparação de resultados com outros grupos de pesquisas, foi construído um *corpus* composto por 500 frases exclusivamente para testes. Para construção do *corpus* foram utilizadas frases retiradas de [26]. O *corpus* utilizado em [25] possuía 25 locutores com 20 frases cada, sendo 17 homens e 8 mulheres, que correspondem a aproximadamente 42 minutos de áudio. Todas as gravações foram realizadas em computadores pessoais utilizando microfones comuns, a taxa de amostragem utilizada foi de 22050 Hz com 16 bits por amostra. O ambiente não foi controlado, existindo assim a presença de ruído nas gravações, caracterizando ambientes onde *softwares* de reconhecimento de voz são utilizados.

Nos primeiros trabalhos, buscou-se criar um sistema de referência utilizando as bases de voz: Spoltech, OGI-22 e West Point. Porém, o sistema se limitava a um reconhecimento "controlado", ou seja, treino e teste com características semelhantes (locutores, ambiente de gravação, etc). Já em [25], o objetivo foi criar um sistema apto a trabalhar em tempo real utilizando o decodificador HDecode do pacote HTK [27]. Foram construídos modelos acústicos e de linguagem voltados para tal tarefa. Além disso, após as etapas de construção dos modelos, técnicas de adaptação ao ambiente foram utilizadas visando melhorar o desempenho do sistema. Assim, modelos acústicos utilizando 14 Gaussianas por mistura e trifones dependentes de contexto foram criados utilizando a base de voz Spoltech e os *audiobooks*. Um modelo de linguagem trigrama com perplexidade igual a 169 foi elaborado utilizando 1,6 milhões de frases retiradas das bases: CETENFolha, Spoltech, OGI-22, West Point e *audiobooks*; além dos textos recentemente extraídos da internet. Todas as simulações foram realizadas com o novo *corpus* de teste e com peso do modelo acústico igual a 2 no processo de decodificação.

O melhor resultado foi obtido com o modelo acústico treinado somente com o Spoltech,

onde a WER foi igual a 44,3% e fator de tempo-real (xRT) igual a 1,5. O pior resultado foi obtido com o modelo acústico treinado apenas com os *audiobooks*, com 55,9% de WER e 3 em xRT. Resultados esperados, já que os *audiobooks* foram produzidos em um ambiente acústico totalmente diferente do *corpus* de teste (ausência de ruído). Diante disso, buscou-se através de técnicas de adaptação ao ambiente, aumentar a precisão do sistema. Com o auxílio da ferramenta HTK, foram utilizadas duas técnicas de adaptação: *maximum a posteriori* (MAP) e *maximum likelihood linear regression* (MLLR). Partindo do modelo acústico produzido a partir dos *audiobooks*, fez-se uma adaptação (introdução de ruído) utilizando o *corpus* Spoltech. Em geral, a combinação MLLR e MAP tende a apresentar melhores resultados, fato que se confirmou, já que o melhor resultado foi obtido combinando as duas técnicas: 29,6% de WER e 2,4 em xRT.

Outro campo de pesquisa concentra-se no estudo e desenvolvimento de aplicativos com interface aural a partir da Microsoft Speech Application Programming Interface (SAPI). Por exemplo, em [28], é apresentado um *software* para aprendizado de língua inglesa com o auxílio do computador (CALL). Resumidamente, o aluno é induzido a responder oralmente, ou mesmo manualmente, os questionamentos através de estímulos escritos, falados e visuais. Além dos exercícios objetivos e subjetivos propostos, existe a possibilidade de um treinamento prévio, onde o aluno pode enriquecer seu vocabulário escutando, via síntese de voz, palavras individualizadas, ou frases, sempre com o respectivo retorno visual, ilustrando o significado da palavra, ou a ação associada à frase.

O aplicativo CALL é capaz de interagir com o usuário através das línguas portuguesa (síntese de voz) e inglesa (síntese e reconhecimento de voz) utilizando os seguintes *engines* providos pela Microsoft: Microsoft English Recognizer e Lernout & Hauspie TTS. Outra funcionalidade é o agente personalizado, Merlin, criado a partir do componente Microsoft Speech Agent para prover *feedback* e assistência quando necessário. Posteriormente em [14], o reconhecedor de voz: Microsoft Speech Recognition Sample Engine for Portuguese (Brazil), em sua versão beta, foi incorporado ao aplicativo CALL, acrescentando interface de reconhecimento de voz em PB.

Em outro trabalho [29], discute-se a integração de diferentes tecnologias disponíveis para o desenvolvimento de sistemas de diálogo falado em PB. Como exemplo, o trabalho apresenta o protótipo de um sistema para a navegação não-visual na Web, em ambiente Windows. Com base na interface SAPI, o sistema estabelece um diálogo falado com o usuário, questionando-o sobre o *site* e a palavra que deseja consultar. Como resposta, o conteúdo principal da página resultada da busca pelo usuário é lido. O próprio sistema coordena as interações com o usuário e, atualmente, é limitado à busca pelo nome de países.

Em 2009, com o intuito de promover o amplo desenvolvimento de aplicações com suporte a reconhecimento de voz em PB, o Grupo FalaBrasil, disponibilizou a comunidade de desenvolvedores seu sistema para reconhecimento automático de voz, o Coruja [30], um *engine* de voz específico para o PB. Esse software, é composto pelo decodificador livre Julius; modelos acústico e de linguagem para o reconhecimento de voz em PB; e uma API própria, implementada na linguagem de programação C/C++ chamada LaPSAPI. Essa API pode ser utilizada tanto na plataforma Linux, a partir de implementações em C++, quanto na plataforma Windows, através de qualquer linguagem da plataforma Microsoft.NET (C#, Visual Basic, entre outras), garantindo certa flexibilidade tanto quanto a plataforma, quanto a linguagem de programação.

Posteriormente, com o objetivo de tornar o Coruja um recurso mais abrangente no que diz respeito as plataformas de desenvolvimento suportadas, em [31], uma API compatível com a especificação JSAPI da Sun Microsystems, chamada JLaPSAPI, foi desenvolvida e disponibilizada a comunidade como parte do Coruja. Com a adição desta API ao Coruja, tornou-se possível o desenvolvimento de aplicativos com suporte a reconhecimento de voz através da linguagem de programação Java a qual até então, o Coruja não oferecia suporte.

Em trabalho recente [32], modelos acústicos para o PB foram construídos utilizando o pacote de ferramentas CMU Sphinx. Esse trabalho foi desenvolvido tendo como objetivo principal gerar recursos que permitissem o desenvolvimento de aplicativos com suporte a reconhecimento de voz em plataformas móveis, como Android e IOs. Os modelos acústicos foram treinados utilizando parte da base West Point, somada a LaPSStory e a base de áudio do Centro de Estudos em Telecomunicações (CETUC), totalizando aproximadamente 143 horas de áudio. Para os testes referentes à tarefa de ditado, a base LaPSBenchmark foi utilizada. O melhor resultado obtido para ditado, com WER de 16,41% e xRT em torno de um, foi obtido pelo modelo acústico com 2.000 estados vinculados e 8 Gaussianas por mistura. Já para a tarefa de comando e controle, aferida no contexto de uma aplicação de correio eletrônico utilizando a base LaPSMail, o melhor resultado, com WER de 0,67%, foi obtido pelo modelo acústico com 500 estados vinculados e 64 Gaussinas por mistura.

1.3 Contribuições

Este trabalho possui diversas contribuições, a seguir será apresentado um resumo das mais importantes. Porém é importante notar que estes trabalhos foram desenvolvidos dentro do Grupo FalaBrasil no Laboratório de Processamento de Sinais da Universidade Federal do Pará, e por se tratar de um grupo com mais de 10 pessoas nem sempre é possível extrair a

exata contribuição de cada membro, isto é, em alguns momentos a contribuição dos autores deste trabalho foram de implementação, em outras de projeto, de orientações e até sugestões. Com isso em mente, as principais contribuições do autor e de relevância para essa dissertação são citadas abaixo.

Um completo sistema para desenvolvimento de aplicativos utilizando reconhecimento de voz em Português Brasileiro foi desenvolvido, disponibilizando modelos acústico e de linguagem para PB, assim como uma API para desenvolvimento de aplicativos em plataforma Microsoft .NET (LaPSAPI). Esse *engine* é chamado Coruja e será detalhado no Capítulo 3. O autor foi responsável pela criação das bases de áudio transcritas, desenvolvimento da LaPSAPI e realização dos testes e validação do sistema.

Outra contribuição é o SpeechOO que é uma extensão que possibilita ditado para a ferramenta Writer do pacote LibreOffice. Utilizando o SpeechOO pessoas que possuem alguma dificuldade motora podem elaborar textos usando ditado e comandos de voz para formatação do texto e controle do Writer. O SpeechOO é apresentado no Capítulo 4. O autor foi o primeiro desenvolvedor do SpeechOO e um dos autores do projeto para financiamento do *software*, posteriormente ajudou no projeto e implementação da JLaPSAPI e do incremento de novas funcionalidades ao SpeechOO, como adição de comandos e integração com novos *softwares*.

O Coruja e o SpeechOO foram publicados na International Conference on Computational Processing of the Portuguese Language (PROPOR), a conferência mais importante na área de processamento de linguagem natural na língua portuguesa. O SpeechOO foi publicado no Workshop de Software Livre 2012, e o Coruja foi utilizado como base para publicações no PROPOR e em um capítulo de livro. As publicações no período de desenvolvimento deste trabalho são mostradas a seguir comprovando o impacto de seu desenvolvimento.

1. Pedro Batista; William Colem; Rafael Oliveira; Hugo Santos; Welton Araújo; Nelson Neto e Aldebaro Klautau. *SpeechOO: Uma Extensão de Ditado para o LibreOffice*. In: Workshop de Software Livre, 2012.

2. Pedro Batista; Hugo Santos; Welton Araújo; Nelson Neto e Aldebaro Klautau. *SpeechOO: A Dictation Extension for the LibreOffice Writer*. In: International Conference on Computational Processing of the Portuguese Language, 2012.

3. Nelson Neto; Pedro Batista e Aldebaro Klautau. *VOICECONET: A Collaborative Framework for Speech-Based Computer Accessibility with a Case Study for Brazilian Portuguese*. In: Modern Speech Recognition Approaches with Case Studies. ISBN 978-953-51-0831-3, 1ed., 2012, p. 303-326.

4. Rafael Oliveira; Pedro Batista; Nelson Neto e Aldebaro Klautau. *Baseline Acoustic Models for Brazilian Portuguese Using CMU Sphinx Tools*. In: International Conference on Computational Processing of the Portuguese Language.

Na área de *call centers* a maior contribuição deste trabalho foi a integração do decodificador Julius e consequentemente os recursos do Coruja com o Asterisk, possibilitando o uso de ASR em PB para atendimento eletrônico em *call centers*. Usando os recursos desenvolvidos, uma unidade de resposta audível (IVR) foi implementada para o serviço disque 100 da Secretária de Recursos Humanos da Presidência da República, sistema esse que atende todo o Brasil e recebe mais de 3 mil ligações diárias. Essas contribuições são detalhadas nos Capítulos 5 e 6. No desenvolvimento do sistema, o autor foi responsável pelo projeto e implementação dos recursos apresentados.

1.4 Síntese dos conteúdos

Este capítulo fez uma breve introdução sobre reconhecimento de voz e mostrou os recursos desenvolvidos pelo Grupo FalaBrasil, assim como as principais contribuições deste trabalho. A seguir é apresentada uma síntese dos conteúdos abordados nos próximos capítulos.

Capítulo 2. Fundamentos de reconhecimento automático de voz. A fim de fazer uma introdução ao leitor, este capítulo apresentará os principais conceitos básicos envolvidos no reconhecimento automático de voz. Em seguida formas para avaliar o ASR serão apresentadas, e por fim serão descritas as principais ferramentas utilizadas nesse trabalho.

Capítulo 3. Aprimoramento de um *engine* ASR em PB. Neste capítulo serão apresentados os recursos desenvolvidos para a construção de um *engine* ASR em PB, assim como uma das principais contribuições deste trabalho, que é o desenvolvimento da LaPSAPI, que permite a utilização do *engine* ASR Coruja em plataforma Microsoft .NET.

Capítulo 4. SpeechOO: ditado no LibreOffice. Neste capítulo será feita uma breve introdução a JSAPI. Essa que é a especificação de uma API para o desenvolvimento de aplicativos com interface aural em Java. Uma implementação dessa API foi desenvolvida e também será apresentada, essa é chamada JLaPSAPI. Finalmente será descrito o SpeechOO que permite o ditado e edição de texto no aplicativo Writer do pacote LibreOffice.

Capítulo 5. ASR em uma IVR usando Coruja e Asterisk. Este capítulo apresenta o Asterisk e suas aplicações. É mostrado como este pode ser conectado a PSTN e como é feito o

desenvolvimento de IVR. Em seguida é detalhado a integração deste com o decodificador Julius, e por fim alguns exemplos do uso de ASR no Asterisk são apresentados.

Capítulo 6. Implementação da IVR do disque 100. Com os recursos desenvolvidos neste trabalho uma IVR foi desenvolvida para atender o serviço disque 100 da Secretária de Recursos Humanos da Presidência da República, o desenvolvimento da IVR é detalhado mostrando gramáticas e dialplans criados.

Capítulo 7. Conclusão. Esse capítulo apresentará as principais contribuições deste trabalho e uma descrição de sua importância para a comunidade tanto acadêmica quanto profissional.

Capítulo 2

Fundamentos de reconhecimento automático de voz

Primeiramente, o presente capítulo faz uma breve introdução sobre sistemas para reconhecimento automático de voz (ASR). Em seguida, são apresentas as formas de avaliação e as ferramentas de desenvolvimento empregadas neste trabalho.

2.1 Reconhecimento automático de voz

Um sistema ASR típico adota uma estratégia estatística, baseada em modelos ocultos de Markov (HMM) [33], e é composto por cinco blocos: *front end*, dicionário fonético, modelo acústico, modelo de linguagem e decodificador, como indicado na Figura 2.1. As duas principais aplicações de ASR são comando e controle e ditado [3]. O primeiro é relativamente simples, pois o modelo de linguagem é composto por uma gramática que restringe as sequências de palavras aceitas. O último, tipicamente, suporta um vocabulário com mais de 60 mil palavras e exige mais processamento.

O processo de *front end* convencional extrai segmentos curtos (janelas, ou *frames*) de um sinal de voz e converte, a uma taxa de *frames* constante (tipicamente, 100 Hz), cada segmento para um vetor $\mathbf{x}$ de dimensão L (tipicamente, $L = 39$). Assumimos aqui que T *frames* são organizados em uma matriz $\mathbf{X}$ de dimensão $L \times T$, que representa uma sentença completa. Existem várias alternativas no que diz respeito à parametrização do sinal de voz. Apesar da análise dos coeficientes cepstrais de frequência da escala Mel (MFCC) ser relativamente antiga [34], essa provou ser efetiva e é geralmente usada como entrada para os blocos de *back end* do sistema ASR [3].

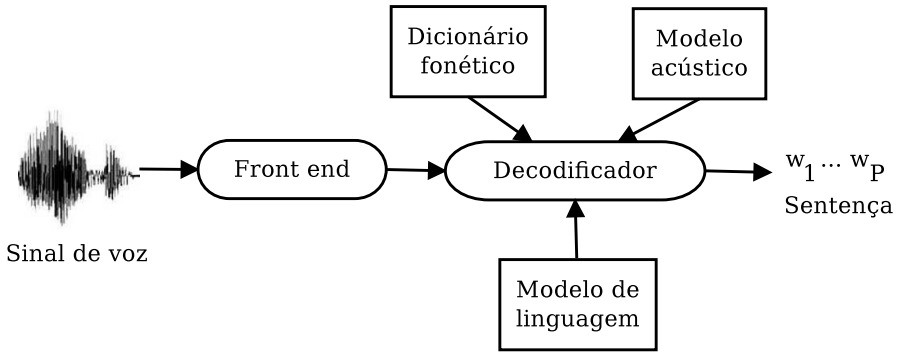

Figura 2.1: Principais blocos de um tipico sistema ASR.

O modelo de linguagem de um sistema de ditado, fornece a probabilidade $p(\mathcal{T})$ de observar a sentença $\mathcal{T} = [w_1, \ldots, w_P]$ de P palavras. Conceitualmente, o decodificador tem como objetivo achar as sentenças $\mathcal{T}^*$ que maximizam a probabilidade *a posterior* dada por (segundo [35])

$$\mathcal{T}^* = \arg\max_{\mathcal{T}} p(\mathcal{T}|\mathbf{X}) = \arg\max_{\mathcal{T}} \frac{p(\mathbf{X}|\mathcal{T})p(\mathcal{T})}{p(\mathbf{X})}, \tag{2.1}$$

onde $p(\mathbf{X}|\mathcal{T})$ é dada pelo modelo acústico. Como $p(\mathbf{X})$ não depende de $\mathcal{T}$, a equação anterior é equivalente a

$$\mathcal{T}^* = \arg\max_{\mathcal{T}} p(\mathbf{X}|\mathcal{T})p(\mathcal{T}). \tag{2.2}$$

Na prática, uma constante empírica é usada para ponderar a probabilidade do modelo de linguagem $p(\mathcal{T})$ antes da mesma ser combinada com a probabilidade dos modelos acústicos $p(\mathbf{X}|\mathcal{T})$.

Dado o grande volume de sentenças concorrentes (hipóteses), a Equação 2.2 não pode ser calculada independentemente para cada hipótese. Portanto, os sistemas ASR usam estruturas de dados como árvores léxicas, usando o artifício de separar as sentenças em palavras, e as palavras em unidades básicas, que aqui chamaremos de fones [3]. A busca por $\mathcal{T}^*$ é chamada decodificação e, na maioria dos casos, hipóteses são descartadas ou podadas (*pruning*). Em outras palavras, para tornar viável a busca pela "melhor" sentença, algumas candidatas são descartadas e a Equação (2.2) não é calculada para elas [36, 37].

Um dicionário fonético (conhecido também como modelo léxico) faz o mapeamento das palavras em unidades básicas (fones) e vice-versa. Para uma melhor *performance*, HMM contínuas são adotadas, onde a distribuição de saída de cada estado é modelada por uma mistura de Gaussianas, como mostrado na Figura 2.2. A topologia típica de uma HMM é a *left-to-right*, onde as únicas transições permitidas são de um estado para ele mesmo ou para o estado seguinte.

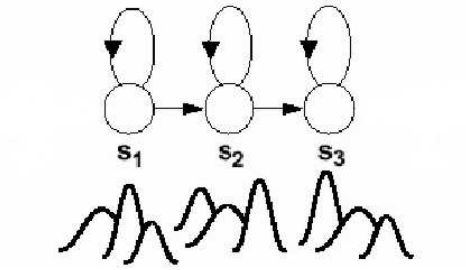

Figura 2.2: Representação gráfica de uma HMM contínua *left-to-right* com três estados e uma mistura de três Gaussianas por estado.

Dois problemas clássicos com relação à modelagem acústica são: a inconstância dos fones devido à co-articulação e à insuficiência de dados para estimar os modelos. O método de compartilhamento de parâmetros (*sharing*) visa combater esse último problema, melhorando a robustez dos modelos. Em muitos sistemas, o compartilhamento é implementado no nível de estado, ou seja, o mesmo estado pode ser compartilhado por HMM diferentes.

O modelo acústico pode conter uma HMM por fone. O que seria uma boa suposição caso um fone pudesse ser seguido por qualquer outro, o que não é verdade, já que os articuladores do trato vocal não se movem de uma posição para outra imediatamente na maioria das transições de fones. Nesse sentido, durante o processo de criação de sistemas que modelam a fala fluente, busca-se um meio de modelar os efeitos contextuais causados pelas diferentes maneiras que alguns fones podem ser pronunciados em sequência [38]. A solução encontrada é o uso de HMM dependentes de contexto, que modelam o fato de um fone sofrer influência dos fones vizinhos. Por exemplo, supondo a notação do trifone a-b+c, temos que b representa o fone central ocorrendo após o fone a e antes do fone c.

Segundo [3], existem basicamente dois tipos de modelos trifones: *internal-word* e *cross-word*. As diferenças entre os mesmos é que no caso do *internal-word* as co-articulações que extrapolam as durações das palavras não são consideradas, sendo assim, menos modelos são necessários. Já no caso do *cross-word*, que considera a co-articulação entre o final de uma palavra e o início da seguinte, a modelagem é mais precisa, porém o número de modelos trifones gerados cresce muito, o que dificulta o trabalho do decodificador e gera uma necessidade de mais dados para treino. Alguns exemplos de transcrição podem ser conferidos na Tabela 2.1.

A escassez de dados de treino também afeta a modelagem da língua que estima (segundo [35])

$$P(\mathcal{T}) = P(w_1, w_2, \ldots, w_P) \tag{2.3}$$

$$= P(w_1)P(w_2|w_1)\ldots P(w_P|w_1, w_2, \ldots, w_{P-1}). \tag{2.4}$$

Sentença	arroz com bife
Monofones	sil a R o s k o∼ b i f i sil
Internal-Word	sil a+R a-R+o R-o+s o-s k+o∼ k-o∼ b+i b-i+f i-f+i f-i sil
Cross-Word	sil sil-a+R a-R+o R-o+s o-s+k s-k+o∼ k-o∼+b o∼-b+i b-i+f i-f+i f-i+sil sil

Tabela 2.1: Exemplos de transcrições com modelos independentes e dependentes de contexto.

É impraticável estimar a probabilidade condicional $P(w_i|w_1,\ldots,w_{i-1})$, mesmo para valores moderados de i. Assim, o modelo de linguagem para sistemas ASR consiste de um modelo n-gram, que assume que a probabilidade $P(w_i|w_1,\ldots,w_{i-1})$ depende somente das $n-1$ palavras anteriores. Por exemplo, a probabilidade $P(w_i|w_{i-2},w_{i-1})$ expressa um modelo de linguagem trigrama.

Resumindo, após o treinamento de todos os modelos estatísticos, um sistema ASR na etapa de teste usa o *front-end* para converter o sinal de entrada em parâmetros e o decodificador para encontrar a melhor sentença $\mathcal{T}$.

Os modelos acústicos e de linguagem podem ser fixos durante a fase de teste, porém adaptá-los pode gerar uma melhor *performance*. Por exemplo, o domínio de aplicação pode ser estimado, e um modelo de linguagem especifico usado. Isso é crucial para aplicações com vocabulário técnico, como relatórios médicos de raios X [39]. A adaptação do modelo acústico possui igual importância [40], este pode ser adaptado, por exemplo, a um locutor, ou ao sotaque de uma determinada região.

Sistemas ASR que usam modelos independentes de locutor são convenientes, porém devem ser robustos o suficiente para reconhecer, com boa *performance*, áudio de qualquer locutor. Com o custo de exigir que o usuário leia algumas sentenças, técnicas de adaptação ao locutor podem melhorar os modelos HMM para um locutor específico. Técnicas de adaptação podem também ser usadas para compensar variações no ambiente acústico, reduzindo o descasamento causado pelo canal ou efeitos de ruído aditivo.

Neste trabalho, nos referimos a um *engine* ASR, como um decodificador e todos os recursos necessários para sua execução (modelos de linguagem e acústico, etc.).

2.2 Avaliação do reconhecimento automático de voz

Na maioria das aplicações que usam reconhecimento de voz (inclusive para ditado) a medida de desempenho utilizada é a taxa de erro por palavra (WER). Dado que geralmente

as transcrições correta e reconhecida possuem um número diferente de palavras, elas são alinhadas através de programação dinâmica [41]. Dessa forma, de acordo com [3], a WER pode ser definida como

$$\text{WER} = \frac{D + R}{W} \times 100\%,$$

onde W é o número de palavras na sentença de entrada, R e D são o número de erros de substituição e deleção na sentença reconhecida, respectivamente, quando comparada com a transcrição correta.

Outra métrica de avaliação é o fator de tempo-real (xRT). O fator xRT é calculado dividindo o tempo que o sistema despende para reconhecer uma sentença pela sua duração. Assim, quanto menor for o fator xRT, mais rápido será o reconhecimento.

Durante o processo de construção dos modelos de linguagem, existem várias características que os diferenciam, como o número de palavras distintas e o número de sentenças usadas para estimar os modelos. No entanto, a métrica mais comum para avaliar os modelos de linguagem é a probabilidade $p(\mathbf{T})$ que o modelo atribui para alguns dados de teste $\mathbf{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_B\}$ compostos de B sentenças. Independência entre as sentenças é assumida, o que leva ao produto das probabilidades $p(\mathbf{T}) = p(\mathcal{T}_1) \ldots p(\mathcal{T}_B)$. Duas medidas são derivadas dessa probabilidade: perplexidade e entropia cruzada (*cross-entropy*) [3]. A entropia cruzada $H_p(\mathbf{T})$ é definida como

$$H_p(\mathbf{T}) = -\frac{1}{W_\mathbf{T}} \log_2 p(\mathbf{T}), \tag{2.5}$$

onde $W_\mathbf{T}$ é o número de palavras em $\mathbf{T}$.

A perplexidade (PP) representa o número médio de palavras que podem seguir uma dada palavra e está relacionada com a *cross-entropy* $H_p(\mathbf{T})$ por

$$PP = 2^{H_p(\mathbf{T})}. \tag{2.6}$$

Para uma dada tarefa (por exemplo, com o tamanho do vocabulário fixado), baixos valores de perplexidade e *cross-entropy* indicam menor incerteza na predição da próxima palavra e, tipicamente, revelam um melhor modelo de linguagem.

2.3 Ferramentas para desenvolvimento de aplicações com ASR

O ponto de partida para a disponibilização de aplicativos utilizando reconhecimento de voz, é a utilização de um *engine* ASR. Existem soluções comerciais de empresas como a

Microsoft [42] e a Nuance [6]. Hoje, nem todos idiomas são suportados, especialmente os chamados sub-representados. Assim, um aspecto chave é a existência de *engines* para a língua alvo.

Além do interesse para pesquisa, duas razões para investigar o desenvolvimento de um *engine* próprio são: a falta de suporte a uma dada língua e a necessidade de minimizar o custo. Existem pacotes de ferramentas (*toolkit*) disponíveis para construir os recursos necessários para o desenvolvimento de *engines* destinados ao reconhecimento automático de voz. O *toolkit* para construir e adaptar modelos acústicos mais amplamente utilizado é o HTK [27]. Já os modelos de linguagem podem ser criados e manipulados com o SRI Language Modeling Toolkit (SRILM) [43].

Quanto aos decodificadores livres, podemos citar o Julius [44] e o Sphinx-4 [45], os quais podem ser usados com os devidos recursos para criar um *engine* ASR na língua desejada. Para aplicações embarcadas (como *smartphones*), os decodificadores Julius e PocketSphinx [46] são os mais populares. Grupos de pesquisas disponibilizam robustos *engines* para inglês, japonês e outras línguas, baseados nesses decodificadores.

Uma vez que o *engine* está disponível, uma API facilita a tarefa de desenvolver aplicações com interface aural para usuários finais. As APIs especificam uma interface *cross-platform* para suportar reconhecedores, sistemas comando e controle e ditado. Como tal, as API não incluem somente as funcionalidades de um ASR, mas também vários métodos e eventos que possibilitam ao programador visualizar e customizar características do *engine*. Um *engine* tipicamente tem sua própria API, porém existem pelo menos duas API que foram projetadas para uso padronizado: a Microsoft Speech API [47] e a Java Speech API [48].

Devemos notar também a existência de *toolkits* para o desenvolvimento de interfaces voltadas para a acessibilidade, como o Microsoft Active Accessibility Platform e o Assistive Technology Service Provider Interface (Linux). Esses *softwares* são projetados para ajudar no desenvolvimento de produtos com tecnologia assistiva, como leitores de tela e interações com elementos do sistema operacional.

2.4 Conclusão

Este capítulo apresentou uma breve descrição sobre a tecnologia ASR descrevendo os principais blocos dos sistemas ASR construídos neste trabalho. O capítulo seguinte descreve os recursos desenvolvidos para PB e as bases de dados não proprietárias usadas para realizar os experimentos.

Capítulo 3

Aprimoramento de um *engine* ASR em PB

ASR é uma tecnologia guiada pelos dados (*data-driven*) que requer uma quantidade relativamente grande de informações rotuladas. Diante de tal dificuldade, os pesquisadores dependem fortemente de bases de dados (*corpora*) públicas e outros recursos específicos para expandir o estado da arte.

Neste contexto, o presente capítulo apresenta bases de áudio e texto construídas no decorrer deste trabalho, além de uma API que permite a fácil integração do decodificador Julius com aplicativos .NET. Por fim, o *engine* Coruja será detalhado.

3.1 LaPSStory

A criação de um *corpus* de voz não é trivial, já que é preciso a transcrição correta do áudio, contratação de locutores, enfim, atividades que despendem muito tempo e recurso humano. Visando contornar essa situação, buscou-se montar um *corpus* a partir de *audiobooks*, que são livros narrados por locutores profissionais. Desse modo, podemos elaborar facilmente as transcrições do texto. Uma dificuldade encontrada nessa abordagem é o fato do texto ser lido por apenas um locutor, característica que torna o *corpus* menos eficiente, já que não se consegue obter diferenças na fala, como os sotaques. Outro problema em bases de livros falados é que os arquivos de áudio possuem em média 70 minutos de duração. Com isso, esses arquivos precisam ser divididos em arquivos menores com cerca de 30 segundos, para que possam ser usados no processo de construção do modelo acústico.

Desta forma foram organizadas 5 horas e 19 minutos de áudio em arquivos com cerca de 30 segundos com suas respectivas transcrições ortográficas. Atividade feita manualmente. Este *corpus* foi utilizado para a construção dos modelos probabilísticos usados primeiramente em [25].

3.2 Corpus da Constituição

Outra solução encontrada para aumentar a base de áudio é semelhante a dos livros falados (Seção 3.1) e consiste na aquisição da Legislação Brasileira em áudio, mais especificamente a Constituição Federal, fornecida pelo Senado Federal em [49]. Foram então convertidos em arquivos com até 30 segundos cerca de 9 horas de áudio da Constituição e 1,5 horas do Código de Defesa do Consumidor. Juntamente com o LaPSStory, esses *corpora* foram utilizados para a construção dos modelos probabilísticos usados em [30].

3.3 LaPSNews

Os modelos de linguagem são essenciais dentro de um sistema ASR, já que assumem a função de caracterizar a língua. Esses modelos são tipicamente construídos utilizando-se de textos interpolados de transcrições de áudio e textos de jornais. O *corpus* de texto inicialmente utilizado pelo Grupo FalaBrasil foi o CETENFolha [20]. Essa base de dados é composta por textos extraídos do jornal Folha de S. Paulo e compilado pelo Núcleo de Linguística Computacional da Universidade de São Paulo, Brasil.

O CETENFolha vem sendo complementado com textos recentes de outros jornais. Isso vem sendo feito através de um processo totalmente automatizado de formatação e coleta diária de jornais disponíveis na internet (*crawling*). Foi então construído o *corpus* de texto: LaPSNews. Em sua primeira versão usada em [25], o LaPSNews possuía cerca de 120 mil sentenças. Hoje, contém mais de 1 milhão de sentenças formatadas.

3.4 LaPSAPI

Ao promover o amplo desenvolvimento de aplicações baseadas em reconhecimento de voz, o Grupo FalaBrasil observou que não era suficiente apenas tornar recursos disponíveis, tais como modelos de linguagem. Esses recursos são úteis para pesquisadores, mas o que

a maioria dos programadores desejam é a praticidade oriunda de uma API. Por isso, foi necessário complementar o código que faz parte do pacote de distribuição do Julius [44].

Recentemente, foi disponibilizada uma versão do pacote Julius totalmente SAPI *compliant*, mas somente na língua japonesa. Logo, é extremamente complicado o uso do Julius com SAPI no caso de outras línguas, como o português. O Julius também não suporta a especificação JSAPI, contudo possui sua própria API, implementada na linguagem de programação C/C++.

Este trabalho apresenta uma API para facilitar a utilização do decodificador Julius, chamada LaPSAPI. A API proposta busca flexibilidade com relação à linguagem de programação. Além disso, o objetivo é prover suporte à plataforma Windows, Linux e qualquer outro potencial sistema operacional. Diante de tal cenário, a solução encontrada foi desenvolver uma API simples, com as funcionalidades necessárias para operar o Julius em aplicativos C++ no Linux e plataforma Microsoft .NET no Windows. O suporte à plataforma Windows é baseado na especificação *common language runtime*, que permite comunicação entre as linguagens suportadas pela plataforma .NET (C#, Visual Basic, J#, entre outras).

A LaPSAPI permite o controle em tempo real do decodificador Julius e da interface de áudio do sistema. Como pode ser visto na Figura 3.1, as aplicações interagem com o Julius através da API. Basicamente, a LaPSAPI abstrai do programador detalhes de baixo nível relacionados à operação do *engine*.

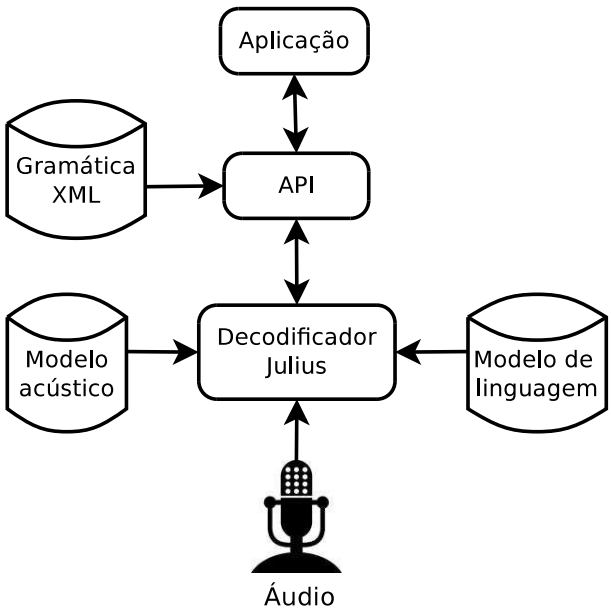

Figura 3.1: API desenvolvida para facilitar a tarefa de operar o decodificador Julius.

Visto que a LaPSAPI suporta objetos compatíveis com o modelo de automação *com-*

ponent object model (COM), é possível acessar e manipular (i.e. ajustar propriedades, invocar métodos) objetos compartilhados que são exportados por outras aplicações. Do ponto de vista da programação, a API consiste basicamente de uma classe principal denominada *SREngine*. Essa classe expõe à aplicação o conjunto de métodos e eventos descritos na Tabela 3.1.

Métodos e Eventos	Descrição Básica
SREngine	Método para configurar e controlar o reconhecedor
loadGrammar	Método para carregar gramática SAPI XML
addGrammar	Método para carregar gramática nativa do Julius
startRecognition	Método para iniciar o reconhecimento
stopRecognition	Método para parar o reconhecimento
OnRecognition	Evento chamado quando alguma sentença é reconhecida
OnSpeechReady	Evento chamado quando o reconhecimento é ativado

Tabela 3.1: Principais métodos e eventos da LaPSAPI.

A classe *SREngine* permite que a aplicação controle aspectos do decodificador Julius. É através dessa classe que a aplicação carrega os modelos acústico e de linguagem, inicia e pára o reconhecimento, e recebe eventos e resultados do processo de reconhecimento de voz.

Através do método *loadGrammar* é possível carregar uma gramática livre de contexto[1] especificada no formato SAPI XML. Para tornar isso possível, um conversor gramatical foi desenvolvido e integrado ao método *loadGrammar*. Essa ferramenta permite que o sistema converta automaticamente uma gramática de reconhecimento especificada no padrão SAPI Text Grammar Format [50] para o formato suportado pelo decodificador Julius[2]. O procedimento de conversão usa as regras gramaticais SAPI para encontrar as conexões permitidas entre as palavras, usando o nome das categorias como nós terminais. Isso também define as palavras candidatas em cada categoria, juntamente com as suas respectivas pronúncias.

É importante salientar que a atual versão do conversor não suporta regras gramaticais do tipo recursivas, facilidade suportada pelo decodificador Julius. Para esse tipo de funcionalidade deve-se carregar a gramática nativa do Julius através do método *addGrammar*. A especificação para esse tipo de gramática pode ser encontrada em [44].

O método *startRecognition*, responsável por iniciar o processo de reconhecimento de voz,

[1] A gramática livre de contexto age como o modelo de linguagem, provendo ao reconhecedor regras que definem as palavras e frases que podem ser ditas.

[2] O Julius suporta tanto modelos n-gram como gramáticas livre de contexto.

basicamente ativa as regras gramaticais e abre o *stream* de áudio. Similarmente, o método *stopRecognition* desativa as regras e fecha o *stream* de áudio.

Alguns eventos também foram implementados. O evento *OnSpeechReady* sinaliza que o *engine* está ativado para reconhecimento. Em outras palavras, ele surge toda vez que o método *startRecognition* é invocado. Já o evento *OnRecognition* acontece sempre que o resultado do reconhecimento encontra-se disponível, juntamente com o seu nível de confiança. Assim, o programador tem a facilidade de aceitar ou rejeitar o resultado do reconhecimento em função do nível de confiança. A sequência de palavras reconhecidas e o seu nível de confiança são passados da API para a aplicação através da classe *RecoResult*.

Através do conjunto reduzido de métodos e eventos apresentados acima é viável construir compactas aplicações com ASR usando o decodificador Julius. A Listagem 3.1 apresenta um código simples que reconhece a partir de uma gramática livre de contexto no formato XML e mostra o resultado na tela. O código-fonte e as bibliotecas da API, assim como alguns aplicativos, encontram-se publicamente disponíveis [51].

```
namespace Test {
  public partial class Form1 : Form {
    private SREngine engine = null;
    public Form1() {
      SREngine.OnRecognition += handleResult; }
    public void handleResult(RecoResult result) {
      Console.Writeline(result.getConfidence() + " | "
        + result.getUterrance() + "\n") }
    private void but_Click(object sender, EventArgs e) {
      engine=new SREngine(@".\LaPSAM1.5.jconf");
      engine.loadGrammar(@".\grammar.xml");
      engine.startRecognition(); }
  }
}
```

Listagem 3.1: Exemplo de código C# usando a API implementada.

3.5 Coruja: Um sistema para ASR em PB

A seguir é feita uma breve descrição do processo de criação dos modelos acústico e de linguagem, que juntamente com a LaPSAPI, formam o *engine* para reconhecimento automático de voz em Português Brasileiro chamado Coruja [30].

3.5.1 Recursos utilizados

Além dos recursos desenvolvidos neste trabalho, outros foram usados para a criação e avaliação dos modelos estatísticos acústico e de língua, são eles: um dicionário fonético e dois *corpora* de áudio. Em [22], é descrito um conversor grafema-fone baseado em regras. A partir desse conversor, um dicionário fonético com mais de 65 mil palavras foi gerado.

A base de dados Spoltech [18] foi revisada de acordo com o procedimento descrito em [21]. Após essa revisão, restaram 4,3 horas de áudio faladas por 477 locutores. Já o *corpus* de voz LaPSBenchmark [25] foi utilizado como referência para avaliar o desempenho do Coruja. Atualmente, o LaPSBenchmark é composto por 35 locutores, com 20 sentenças faladas por cada um deles, o que corresponde a 54 minutos de áudio. As gravações do LaPSBenchmark foram feitas em ambiente acústico não controlado, usando computadores pessoais e microfones comuns.

O pacote de ferramentas HTK [27] foi usado para construir e adaptar o modelo acústico. Já o SRILM [43] foi usado para construir o modelo de linguagem trigrama no formato ARPA. O processo de criação desses modelos estatísticos é descrito na próxima seção.

3.5.2 Sistema base

O *front-end* adotado foram os amplamente utilizados 12 MFCC (*Mel-Frequency Cepstral Coefficients*), com suas primeiras e segundas derivadas. O modelo acústico inicial para 39 fones (38 monofones e um modelo de silêncio) usou HMM *3-state left-to-right*. Em seguida, foram criados trifones *cross-word* a partir dos modelos monofones. As matrizes de transição dos trifones que possuem o mesmo fone base foram compartilhadas. Os trifones com características fonéticas similares também foram compartilhados, para isso foi utilizado uma árvore de decisão fonética [21].

Após o processo de vínculo, o número de componentes por mistura nas distribuições foi gradualmente incrementado até 14 Gaussianas por mistura, finalizando o processo de treino. Depois de cada passo, os modelos foram re-estimados usando o algoritmo de *Baum-Welch*.

Duas técnicas de adaptação do modelo acústico foram utilizadas. A primeira é o *maximum likelihood linear regression* (MLLR), que computa um conjunto de transformações que visa reduzir a diferença entre um modelo inicial e os dados de adaptação. A segunda técnica usa a abordagem *maximum a posteriori* (MAP). Na implementação do HTK, a média de cada Gaussiana é atualizada por MAP usando a média da distribuição a priori, os pesos da Gaussiana e os dados do locutor.

Foram usadas 1,6 milhões de frases para treinar o modelo de linguagem. O vocabulário foi composto por mais de 65 mil palavras dos *corpora* CETENFolha e LaPSNews. O modelo trigrama foi desenvolvido com suavização Kneser-Ney e obteve perplexidade igual a 169. Um total de 10 mil frases, disjuntas do conjunto de treino, foram usadas para medir a perplexidade.

3.5.3 Resultados experimentais

O modelo acústico foi inicialmente treinado usando os *corpora* LaPSStory e Constituição. Em seguida, o modelo foi adaptado com a base de dados Spoltech usando as técnicas MLLR e MAP em treinamento supervisionado [25].

Para medir a robustez do Coruja, uma comparação foi feita com outros dois sistemas: HDecode (parte do HTK) e o *software* comercial IBM ViaVoice [5]. O processo de avaliação foi feito em duas etapas: usando modelos dependentes e independente de locutor. Em todas elas, o *corpus* LaPSBenchmark foi utilizado para comparar os sistemas. As medidas de desempenho foram a WER e o xRT. Os resultados são apresentados na Tabela 3.2, onde os modelos acústicos e de linguagem do Coruja foram utilizados com os decodificadores HDecode e Julius, sendo este último o decodificador usado pelo LaPSAPI.

Decodificador	**Modelos independente**		**Modelos dependentes**	
	WER(%)	**xRT**	**WER(%)**	**xRT**
Julius	39,57	0,7	22,29	0,7
HDecode	29,37	0,9	15,4	0,8
IBM ViaVoice	29,29	-	17,3	-

Tabela 3.2: Comparação dos sistemas usando modelos dependente e independente de locutor.

Uma sessão de adaptação ao locutor é necessária para o funcionamento do IBM ViaVoice. Assim, para o primeira etapa (modelos independentes de locutor), o processo de adaptação do IBM ViaVoice foi realizado usando vozes de seis locutores, 3 homens e 3 mulheres, que correspondem a 10 minutos de áudio. A medida de xRT não é disponível no IBM

ViaVoice. Observamos que o HDecode e o IBM ViaVoice tiveram quase o mesmo resultado, enquanto o Julius obteve resultado um pouco inferior, porém foi mais rápido.

Dois locutores foram usados na segunda etapa do teste e a adaptação foi feita com 10 minutos de áudio para cada locutor. As técnicas MLLR e MAP foram novamente utilizadas para a adaptação dos modelos usados no Julius e HDecode. No caso do IBM ViaVoice, o processo de adaptação padrão foi executado. O HDecode mostrou resultados satisfatórios quando comparado ao IBM ViaVoice. Embora o decodificador Julius tenha apresentado a pior *performance* nas duas etapas, em alguns casos, se configurado propriamente, pode eventualmente superar o HDecode, como descrito em [52].

3.6 Conclusão

Neste capítulo foram apresentadas as bases de áudio e ferramentas utilizadas na criação do Coruja, bem como os recursos aqui desenvolvidos que compõe o mesmo. A seguir, o *software* SpeechOO é descrito, bem como os recursos utilizados na construção do mesmo.

Capítulo 4

SpeechOO: Ditado no LibreOffice

Este capítulo apresenta o SpeechOO, uma extensão para o programa Writer do pacote LibreOffice que permite sua utilização a partir da interface de voz. Inicialmente, será descrita a Java Speech API, especificação para o uso de tecnologias de voz em ambiente Java. Em seguida, é descrita a especificação Universal Network Objects (UNO), que permite o desenvolvimento de extensões para o LibreOffice. É apresentada também a JLaPSAPI, interface desenvolvida para permitir o uso do Coruja através da especificação JSAPI. Por fim é descrito o SpeechOO com todas suas funcionalidades e detalhes de implementação.

4.1 JSAPI: Java Speech API

A Java Speech API (JSAPI) é uma especificação que permite aos desenvolvedores incorporarem tecnologia de voz aos seus aplicativos escritos no ambiente de programação Java. A JSAPI surgiu de uma parceria entre grandes empresas, como a Apple Computer, AT&T, Dragon Systems, IBM Corporation, dentre outras. Dessa cooperação surgiram, além da especificação, *engines* de voz implementando o padrão proposto, ambos visando facilitar o desenvolvimento de aplicativos baseados em voz na plataforma Java.

A JSAPI é independente de plataforma com suporte a sistemas para ditado, comando e controle e síntese de voz. Desse ponto em diante, o termo JSAPI será utilizado como referência a parte ASR dessa API. Basicamente, a JSAPI é utilizada para fazer a interface entre a aplicação Java e o reconhecedor, disponibilizando um conjunto de classes que representam a visão do programador sobre o *engine*, conforme a Figura 4.1. A maior parte dos produtos que implementam JSAPI são comerciais, e a Sun (líder do projeto), não possui nenhuma implementação de referência. A única implementação para ASR de código livre encontrada é

o Sphinx-4.

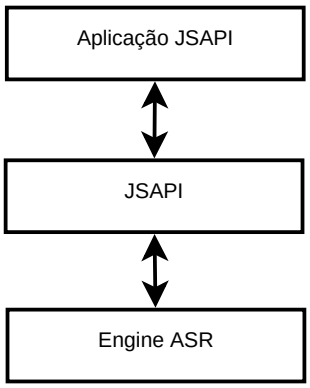

Figura 4.1: Arquitetura de auto nível de uma aplicação usando JSAPI.

4.1.1 Usando um *engine* ASR através da JSAPI

Para uma melhor compreensão da especificação JSAPI, serão descritos a seguir seus principais recursos que possibilitam a utilização de reconhecimento de voz em aplicativos Java.

A classe *Central* da JSAPI é usada para encontrar e criar o reconhecedor. Essa classe espera encontrar o arquivo `speech.properties`, provido pelo *engine*, nos diretórios `user.home` ou `java.home/lib` do sistema operacional. Seu método *createRecognizer* é responsável por instanciar um reconhecedor, descrito pela classe *RecognizerModeDesc* que é o responsável por receber as características que o reconhecedor deve possuir.

A classe *RecognizerModeDesc* é responsável por exemplo, por definir o idioma para o qual queremos o reconhecedor, e se este deve suportar ou não o modo de ditado. Caso a aplicação não especifique a língua do reconhecedor, a língua padrão do sistema, retornada por *java.util.Locale.getDefault()*, será utilizada. Se mais de uma *engine* suportar a língua padrão, a classe *Central* dará preferência a um *engine* que esteja *running* (em espera), e então a um que dê suporte a língua definida no *locale*. O objeto *EngineList*, descrito a seguir, surge como um possibilidade de visualização dos reconhecedores disponibilizados, facilitando o estudo e a escolha do *software* que melhor se adequá a aplicação.

```
RecognizerModeDesc required = new RecognizerModeDesc();
required.setLocale(new Locale("en", ""));
required.setDictationGrammarSupported(Bollean.TRUE);
EngineList engineList = Central.availableRecognizers(required);
for (int i = 0; i < engineList.size(); i++) {
```

```
    RecognizerModeDesc desc = (RecognizerModeDesc) engineList.get(i);
    System.out.println(desc.getEngineName() + desc.getLocale());
}
```

Escolhido o reconhecedor o próximo passo é criá-lo, através do método *createRecognizer* da classe *Central*. Os métodos *allocate* e *resume* alocam todos os recursos necessários e prepararam o reconhecedor para o reconhecimento, respectivamente. O código abaixo ilustra a criação do *engine*.

```
static Recognizer rec;
rec = Central.createRecognizer(required);
rec.allocate();
```

Uma gramática de regras suporta o diálogo discreto entre uma aplicação e o usuário. O que pode ser dito é definido explicitamente através de um conjunto de regras descritas, utilizando-se uma linguagem denominada Java Speech Grammar Format (JSGF) [53]. Outra alternativa se dá através de objetos e métodos da própria JSAPI. As regras de uma gramática podem ser carregadas em tempo de execução através de um arquivo de texto, uma URL ou uma regra por vez. O código abaixo mostra uma gramática sendo carregada a partir de um arquivo contendo regras no formato da JSGF.

```
FileReader reader = new FileReader("myGrammar.jsgf");
RuleGrammar gram = rec.loadJSGF(reader);
```

O trecho de código a seguir ilustra a criação de uma gramática e adição de uma regra com a palavra "iniciar", semelhante ao que foi feito acima, porém através de métodos que permitem a especificação de uma gramática via código fonte:

```
RuleGrammar gram = rec.newRuleGrammar("myGrammar");
RuleToken word = new RuleToken("iniciar");
gram.setRule("ruleName", word, true);
```

A linha de código a seguir cria uma instância de gramática para ditado. Tal estrutura suporta a entrada de texto sem formato pré-determinado. O usuário pronuncia as palavras uma após a outra em qualquer sequência desde que pertençam ao domínio em uso. Palavras não pertencentes ao conjunto, geram resultados incorretos ou são simplesmente ignorados.

```
DictationGrammar gram = rec.getDictationGrammar(null);
```

Após o carregamento da(s) gramática(s), deve-se ligar um ou mais ouvidores (*listeners*) ao reconhecedor eles serão responsáveis por interpretar o que foi dito, determinando qual processo se refere ao que foi reconhecido.

```
rec.addResultListener(new Teste());
```

Por fim, todas as gramáticas precisam ser disponibilizadas, salvas e ativadas, como descrito a seguir. Somente após estas três etapas é que o reconhecimento da gramática pode ocorrer. Os métodos da interface *Recognizer* que permitem iniciar e interromper o reconhecimento são, respectivamente, *resume()* e *pause()*.

```
gram.setEnabled(true);
rec.commitChanges();
rec.requestFocus();
rec.resume();
```

Neste momento, o reconhecedor espera que o usuário diga alguma coisa. Quando uma correspondência com a(s) gramática(s) ativa(s) for detectada, o reconhecedor gerará um *ResultEvent* que será recebido pelo método *resultAccepted* do *listener* (nesse exemplo *Teste*). Uma vez recebidas as informações sobre o que se "ouviu", pode-se desencadear um processamento pré-estabelecido pelo programador.

```
public void resultAccepted(ResultEvent e) {
    Result r = (Result) e.getSource();
    ResultToken tokens[] = r.getBestTokens();
    for (int i = 0; i < tokens.length(); i++)
        System.out.println(tokens[i].getSpokenText() + " ");
    rec.deallocate();
    System.exit(0);
}
```

O sucesso (*resultAccepted*) ou fracasso (*resultRejected*) do reconhecimento não é determinado por um patamar único. De fato, este índice pode ser estabelecido através do método *setConfidenceLevel* que aceita um valor variando de *0.0* até *1.0*, sendo o valor padrão igual a *0.5*. Assim, o nível de confiabilidade é o limite entre o que é aceito e o que é rejeitado. Por exemplo, em um nível de confiabilidade mínimo, todos os resultados são tratados como *resultAccepted*. O nível de sensibilidade é o limite entre o som que deve ser processado e o

que deve ser ignorado. É classificado com um valor que vai de *0.0* até *1.0*, respectivamente referente aos sons mais baixos e mais altos. Seu valor padrão é *0.5*, e quanto mais baixo for esse parâmetro, mais alto o usuário terá que falar.

```
RecognizerProperties props = rec.getRecognizerProperties();
props.setConfidenceLevel(new Float(0.0).floatValue());
props.setSensitivity(new Float(1.0).floatValue());
rec.commitChanges();
rec.resume();
```

Os reconhecedores geram, além dos resultados referentes às gramáticas, eventos de áudio (*RecognizerAudioEvent*) de forma que as aplicações possam receber informações sobre a entrada do som. São três eventos gerados: um com informações sobre o volume da entrada do som, e outros dois quando se tem o início ou o fim de um diálogo. Estes eventos são gerados automaticamente e, para serem utilizados, basta que se associe um ouvidor de áudio (*RecognizerAudioListener*) ao reconhecedor.

4.2 Desenvolvimento de extensões para o LibreOffice

As extensões (*plug-in*) são cada vez mais importantes no universo de aplicativos, pois elas possibilitam que qualquer programador adicione funcionalidades a um determinado programa sem grandes esforços. Outro motivo do sucesso das extensões é a sua fácil e rápida instalação, pois elas são apenas acopladas a um *software* principal.

Podemos observar o quanto extensões são importantes visitando o *site* da comunidade LibreOffice[1], onde podemos encontrar mais de 100 extensões para esse aplicativo. Outro exemplo de sucesso com extensões é o navegador de internet Mozilla Firefox[2], para o qual podemos encontrar milhares de extensões, com algumas sendo utilizadas por milhões de usuários.

As próximas seções descrevem o pacote de ferramentas para escritório LibreOffice e a especificação Universal Network Objects (UNO), exemplificando o uso de componentes UNO no desenvolvimento de extensões.

[1] http://extensions.libreoffice.org/extension-center
[2] https://addons.mozilla.org/en-US/firefox/

4.2.1 A suíte de escritório LibreOffice

O LibreOffice é um pacote de ferramentas para escritório livre e de código aberto, desenvolvido pela The Document Foundation surgindo de um projeto paralelo ao OpenOffice [54]. Com o LibreOffice é possível editar textos, fazer planilhas, gráficos, cálculos, apresentação de *slides*, dentre diversas outras funcionalidades. Este é compatível com os principais sistemas operacionais (Windows, MAC OS X, Linux, BSD e UNIX) e também com arquivos de outras ferramentas de escritório como o Microsoft Office, OpenOffice, iWork e GoogleDocs.

O LibreOffice foi escrito principalmente em linguagem de programação C++. No entanto, é possível desenvolver extensões em diversas linguagens, como por exemplo, Java, C++ e Python. Essa diversidade é possível graças a especificação UNO, que define componentes que são implementados em diversas linguagens e são capazes de se comunicar entre si.

4.2.2 UNO: Universal Network Objects

O LibreOffice fornece uma API para desenvolvimento de extensões independente de linguagem de programação. Isso é possível graças a especificação UNO, que permite a comunicação entre componentes implementados em qualquer linguagem, para a qual existe uma implementação UNO. Atualmente, é possível encontrar extensões em Java, C++, Python, entre outras.

O LibreOffice foi construído a partir de bibliotecas em C++, então uma mudança nessa base exigiria que grande parte do programa fosse recompilado, o que levava cerca de 2 dias, isso se nenhum problema fosse encontrado. Esse é apenas um dos problemas que levaram a criação de um modelo de componentes.

O UNO veio então para permitir a adição de componentes por meio de interfaces. O UNO é composto principalmente por uma especificação binária, onde é definido o *layout* de memória de seus tipos. Cada objeto possui seu próprio ambiente, e a comunicação com o restante do programa é feita através de uma ponte UNO. A ponte UNO permite a comunicação inclusive de programas escritos em diferentes linguagens de programação, visto que a especificação de memoria é única. Como ilustrado na Figura 4.2, um componente se comunica com o ambiente UNO que fará a ponte entre esse e qualquer outro componente. Componentes UNO são implementações da especificação UNO, que pode ser o programa principal (principal funcionalidade do programa) e extensões, que são *features* adicionais ao programa principal.

A seguir é apresentado um código em Java usando a especificação UNO. Esse código tem como principal objetivo mostrar na tela do processador de texto Writer do LibreOffice,

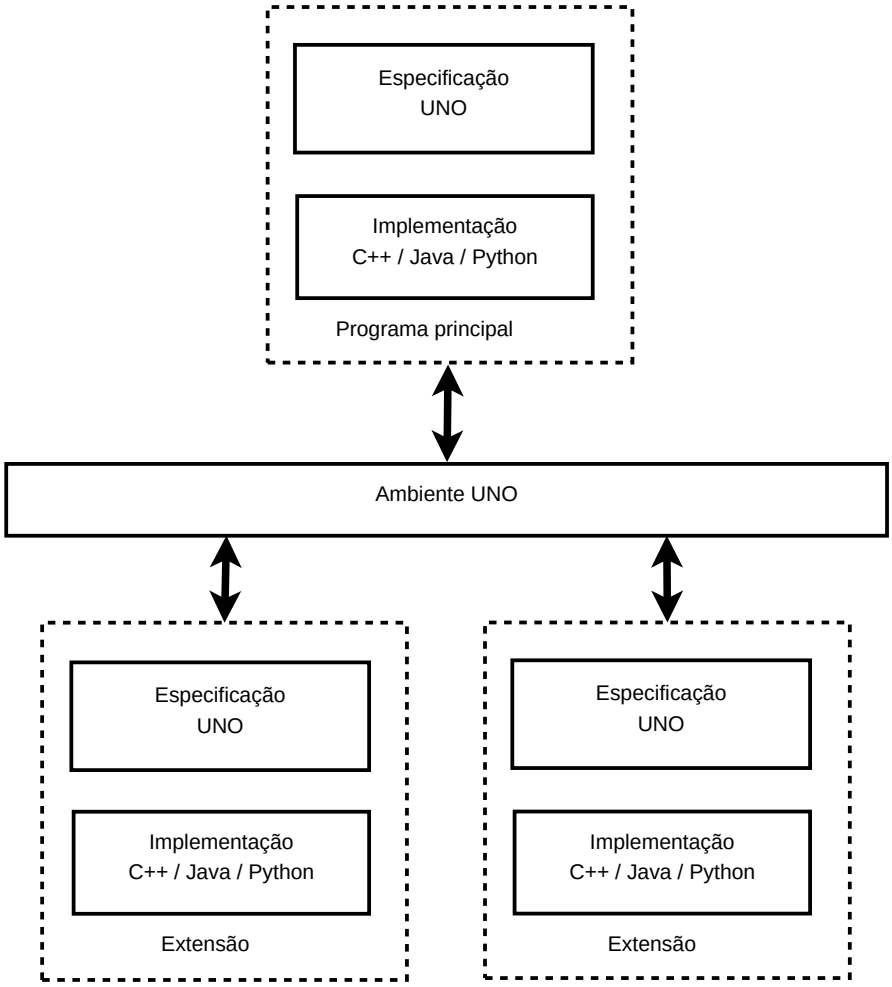

Figura 4.2: Arquitetura do UNO.

a *String* passada como argumento para o método *insertNewSentence* da classe *InputSentence* também definida abaixo.

```
package org.speechoo.inputText;
import com.sun.star.frame.XController;
import com.sun.star.text.XText;
import com.sun.star.text.XTextDocument;
import com.sun.star.text.XTextViewCursor;
import com.sun.star.text.XTextViewCursorSupplier;
import com.sun.star.uno.UnoRuntime;
import org.speechoo.SpeechOO;
public class InputSentence {
  public void insertNewSentence(String sentence) {
    XTextDocument xDoc = (XTextDocument) UnoRuntime.queryInterface(
```

```
            XTextDocument.class, SpeechOO.m_xFrame.getController().getModel());
        XText xText = xDoc.getText();
        XController xController = xDoc.getCurrentController();
        XTextViewCursorSupplier xViewCursorSupplier =
            (XTextViewCursorSupplier) UnoRuntime.queryInterface(
                XTextViewCursorSupplier.class, xController);
        XTextViewCursor xCursor2 = xViewCursorSupplier.getViewCursor();
        xText.insertString(xCursor2, sentence, true);
        xCursor2.gotoRange(xCursor2.getEnd(), false);
    }
}
```

4.3 JLaPSAPI: Utilizando o Coruja pelo padrão JSAPI

A JLaPSAPI é uma API desenvolvida a partir da LaPSAPI, que permite o uso do decodificador Julius (e consequentemente do *engine* Coruja) em aplicativos Java. Apesar da LaPSAPI descrita na Seção 3 ser suficiente e robusta para o desenvolvimento de aplicativos, esta não segue nenhuma especificação reconhecida, ou seja, um programa escrito a partir dessa API não poderia trocar seu *engine* ASR. Outro problema é que não há suporte a uma linguagem *cross-platform* como a linguagem de programação Java.

Assim, surgiu a necessidade de se construir uma nova API em cima da LaPSAPI seguindo a especificação JSAPI, a qual chamamos JLaPSAPI. A JLaPSAPI é de grande importância para o SpeechOO, pois, dessa forma, este pode usar diferentes *engines*, inclusive em diferentes línguas. Além de ser importante para que o SpeechOO possa ser utilizando em diferentes sistemas operacionais, já que seu programa principal (LibreOffice) também é *cross-platform*.

A JLaPSAPI exigiu a comunicação entre código Java e C++ (LaPSAPI). Essa comunicação que foi provida pela Java Native Interface (JNI) [55]. Nessa arquitetura o acesso ao *engine* é feito através de código especificado pela JSAPI. Como mostrado na Figura 4.3, o programador agora tem a possibilidade de alternar entre o Coruja e qualquer outro *engine* que siga a especificação JSAPI, sem a necessidade de alteração no código de sua aplicação Java. Os métodos e eventos JSAPI disponibilizados pela JLaPSAPI são apresentados na Tabela 4.1 e devem ser utilizados como descrito na Seção 4.1.

A JLaPSAPI vem sendo constantemente desenvolvida para oferecer todas as funcionalidades demandadas pelo SpeechOO, uma aplicação conceitualmente mais complexa quanto à

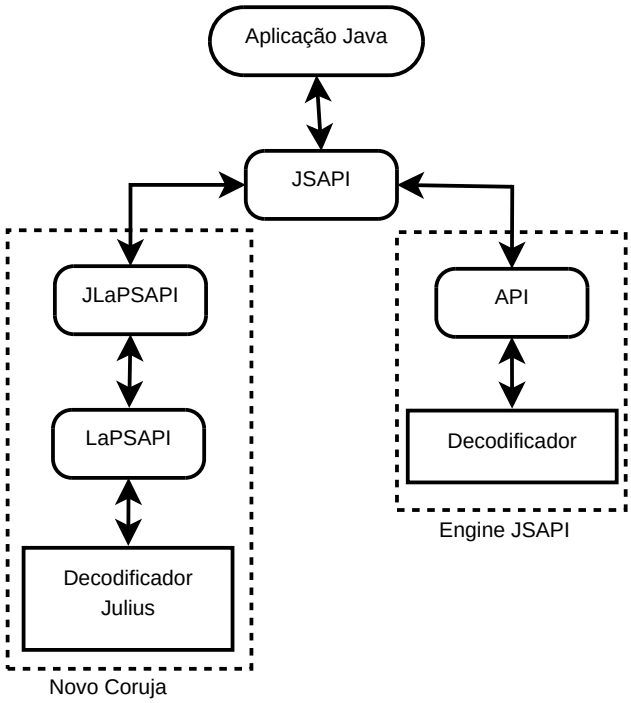

Figura 4.3: Arquitetura da JLaPSAPI.

Métodos e Eventos	**Descrição Básica**
createRecognizer	Cria uma instancia do *engine*
allocate	Aloca os recursos do *engine*
deallocate	Desaloca os recursos do *engine*
resume	Inicia o processo de reconhecimento
pause	Pausa o processo de reconhecimento
resultAccepted	Recebe o resultado do reconhecimento
setEnabled	Pausa ou inicia gramática ou ditado
getAlternativeTokens	Captura vários resultados no modo ditado
loadJSGF	Carrega gramática
getDictationGrammar	Carrega a gramática de ditado
addResultListener	Seta classe receptora dos resultados do reconhecedor

Tabela 4.1: Métodos e eventos suportados pela JLaPSAPI.

avaliação do texto reconhecido.

4.4 SpeechOO: Ditado no LibreOffice

O SpeechOO é uma extensão para o *software* Writer da suíte de escritório *LibreOffice* que permite a criação de textos através da interface de voz, o que viabiliza a escrita de textos por pessoas que, por algum motivo, não podem utilizar o teclado e o *mouse* de um computador. Atualmente esta extensão não permite que todas as funcionalidades do Writer sejam controladas somente a partir da voz, entretanto os recursos implementados até este ponto do trabalho são suficientes para que se crie e edite documentos, simples sem tabelas, figuras e gráficos entre outros elementos que não texto. A seguir, a arquitetura e funcionalidades do SpeechOO são descritas.

A Figura 4.4 ilustra a arquitetura do SpeechOO. Como se pode observar a comunicação entre o SpeechOO e o LibreOffice é efetuada através da especificação UNO. Já a interface com o reconhecedor é realizada através de JSAPI. Note que o *engine* utilizado é o Coruja, porém este pode ser substituído por qualquer outro *engine* que implemente a especificação JSAPI.

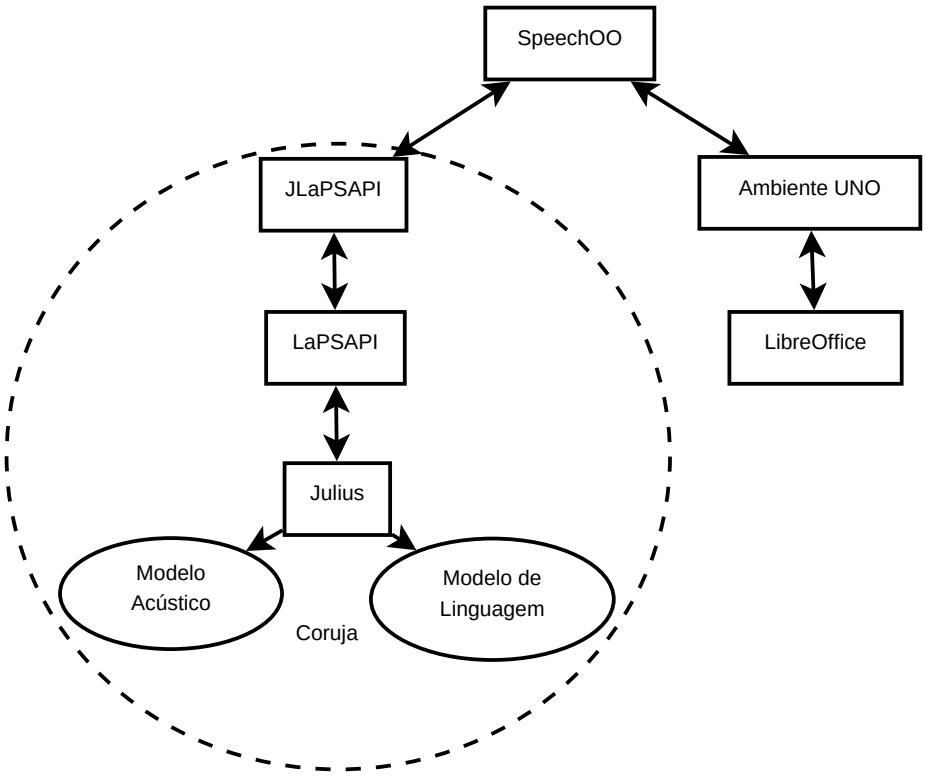

Figura 4.4: Arquitetura do SpeechOO.

O SpeechOO possui dois modos de funcionamento, um voltado para o ditado de conteúdo e outro para o controle de funcionalidades do Writer. A seguir, são descritos os modos de funcionamento do SpeechOO.

4.4.1 Modo Ditado

O modo padrão de funcionamento do SpeechOO é o modo ditado. Neste modo, a fala do usuário é convertida em texto e então é apresentada na tela de edição do aplicativo Writer. Este modo é utilizado nos momentos em que o SpeechOO não pode prever o que é dito pelo usuário, ou seja, quando o corpo do texto está sendo ditado. A Figura 4.5 ilustra um momento básico da utilização do SpeechOO colocando em destaque: (1) o botão que inicia a execução do SpeechOO na barra de ferramentas, (2) o texto reconhecido no corpo do documento e (3) a janela de *status* na parte inferior, a qual indica o modo de funcionamento do SpeechOO que encontra-se ativo.

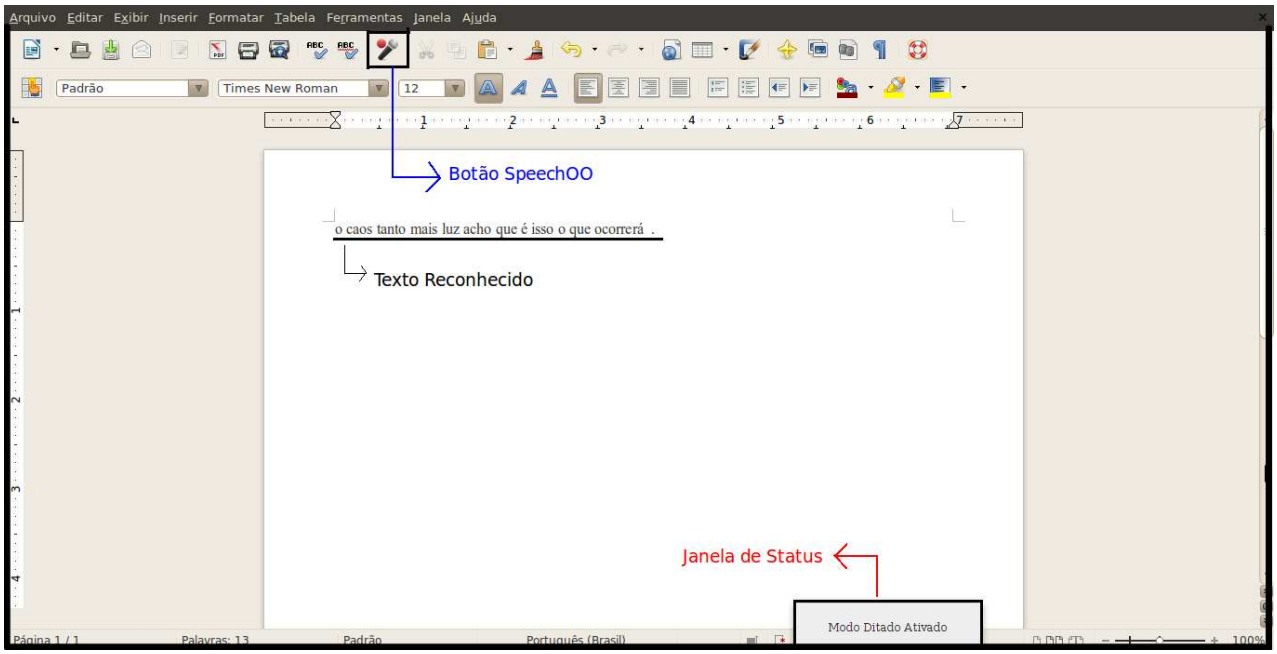

Figura 4.5: Interface do SpeechOO junto ao LibreOffice Writer

Uma alternativa para o funcionamento do *plugin* seria utilizar somente o modo ditado em qualquer momento da interação com o usuário. Línguas como o inglês possuem reconhecedores com uma taxa de acerto muito alta, além de módulos de processamento de linguagem natural (NLP) bastante maduros, o que torna possível a adoção desta estratégia sem que o usuário sofra com erros frequentes do reconhecedor. As ferramentas para PB ainda não atingiram o mesmo nível. Por conta disto, foi necessário implementar o modo comando, facilitando a tarefa do reconhecedor nos momentos em que o ditado pelo usuário é previsível, ou seja, quando comandos são ditos para a formatação ou edição do texto.

4.4.2 Modo Comando

O modo comando é utilizado para manipular as funcionalidades do Writer. Os comandos foram separados por tipos, cada um de acordo com a sua funcionalidade. Os comandos são detalhados na Seção 4.4.3.

A seguir é apresentado um trecho de código que é executado no momento em que o comando é reconhecido pelo Coruja. Neste código o resultado é recebido e o comando a ser executado é mostrado para o usuário na janela de *status*.

```
for (int i = 0; i < tokens.length; i++) {
        if (i > 0) {
                returnTokens.append(' ');
        }
        returnTokens.append(tokens[i].getSpokenText());
        Recognized = returnTokens.toString();
        Recognized = Recognized.substring(0, 1).toUpperCase()
                .concat(Recognized.substring(1));
        SpeechOO.label.setText(Recognized);
        SpeechOO.label.setVisible(true);
        Recognized = Recognized.toLowerCase();
}
```

Para que fosse possível a execução dos dois modos de funcionamento no SpeechOO foram utilizadas *threads*. A arquitetura do SpeechOO foi projetada de forma que o funcionamento das *threads* é mutualmente exclusivo, isto é, em um determinado momento apenas uma das duas *threads* está ativa.

Para a implementação da mudança entre os modos de funcionamento, foi necessário a criação de um *Key Listener* em

utilizados em aplicativos que realizam ações quando uma tecla é pressionada. Os exemplos são os mais diversos como, por exemplo: sistemas operacionais, editores de textos, navegadores de internet dentre outros.

O primeiro passo para a utilização de um *Key Listeners* é a escolha da tecla que vai ser utilizada (*Switch Key*). No caso do Writer, existem algumas teclas que não possuem função. A tecla `Alt Graph` é um exemplo. De forma solitária, esta tecla não possui nenhuma função no *software*. Além disso, esta se localiza na parte inferior do teclado, demandando menos esforço do usuário para uma utilização constante. Por esses motivos esta tecla foi escolhida como a *Switch Key* do *plugin*.

A *Switch Key* aqui implementada, funciona de uma maneira simples: quando está pressionada, é ativado o modo comando e desativado o de ditado; quando está solta ocorre o inverso.

4.4.3 Comandos Implementados

O SpeechOO possui vários comandos que controlam diferentes funções do Writer. Esses foram divididos em vários grupos, são eles: comando de pontuação, navegação, fonte, cor, formatação e comandos alternativos. Cada um deles será detalhado em seguida. A maioria dos comandos são auto explicativos, dessa forma a maioria deles serão apenas citados e sua funcionalidade subentendida.

Os comandos de pontuação simulam a utilização do teclado, ou seja, ao falar algum comando de pontuação, a referida pontuação é inserida no corpo do documento na posição atual do cursor. Os comandos de pontuação são: “vírgula”, “ponto”, “maior”, “menor”, “dois pontos” e “ponto e vírgula”.

Alguns dos comandos de navegação tem apenas a função de mudar a posição atual do cursor, são eles: “avançar palavra”, “voltar palavra”, “avançar parágrafo” e “voltar parágrafo”. Existem também comandos de navegação que além de alterar a posição do cursor alteram também o texto, são eles: “backspace”, “enter” e “espaço”.

Na parte superior do Writer existem botões que realizam formatação do texto, tais como, sublinhar e justificar o texto. Para permitir esse tipo de ação utilizando comandos de voz através do SpeechOO, foram criados os comandos de formatação, são eles: “negrito”, “itálico”, “sublinhar”, “alinhar para a direita”, “alinhar para a esquerda”, “justificar” e “centralizar”.

Os comandos de fonte mudam o tipo de fonte ou o seu tamanho. Para mudar o tamanho da fonte é falada a palavra “fonte” seguida do tamanho desejado, por exemplo, o comando “fonte vinte” troca o tamanho da fonte atual para 20 pontos. Para trocar o estilo da fonte, o comando é “mudar fonte”. Ao reconhecer o comando “mudar fonte”, o SpeechOO mostra uma caixa com várias fontes associadas a números, o usuário deve então falar o número correspondente a fonte escolhida.

O comando para mudança de cor é semelhante ao comando para mudança do tamanho da fonte. Para mudar a cor deve ser usado o comando “cor” seguido da cor desejada. Por exemplo, para mudar para a cor azul o comando é “cor azul”. As cores implementadas são: amarelo, vermelho, marrom, azul, verde e preto.

Os comandos alternativos são, “salvar” e “corrigir”. Ao falar o comando “salvar” uma mensagem aparece pedindo o nome do arquivo desejado e, após o usuário falar o nome desejado,

o arquivo é salvo na pasta de documentos do usuário.

Já o comando alternativo "corrigir" tem como objetivo a correção de alguma sentença erroneamente reconhecida. Se em algum momento o usuário perceber que a sentença que falou foi erroneamente reconhecida, o comando "corrigir" deve ser usado. Esse mostra na tela sentenças alternativas a reconhecida. Por exemplo, supondo que o usuário falou "com fome comi duas maçãs" e o texto reconhecido foi "a fome duas maça", o usuário pode falar "corrigir" e alternativas serão apresentadas. Para esse caso é suposto: "com fome duas maças", "com fome comi maça" e "com fome comi duas maças". Se este for o caso o usuário pode escolher a hipótese número 3 e seguir com seu ditado. Os resultados alternativos são hipóteses geradas pelo reconhecedor que possuem uma taxa de confiança menor, ou seja, para o reconhecedor, essas sentenças tiveram menor probabilidade de estarem corretas. É importante notar que nem sempre a sentença correta estará entre as hipóteses alternativas.

4.4.4 Adaptação ao Locutor

A adaptação ao locutor é uma forma de otimizar a *performance* de um sistema ASR para um locutor específico. No processo de adaptação um modelo acústico genérico é adaptado às características acústicas da voz de um locutor específico. No SpeechOO para adaptar o modelo a um locutor, é necessária a leitura em voz alta de uma série de textos pré-definidos. Os quais serão usados para adaptar o modelo usando técnicas de MAP e MLLR.

A seguir é descrito o processo de adaptação ao locutor no SpeechOO. Primeiramente o usuário deve selecionar a opção de `Adaptação de Locutor` no menu do LibreOffice. Então uma tela é aberta, onde o usuário é instruído a selecionar um modelo existente ou criar um novo modelo, como mostra a Figura 4.6. A caixa de texto superior é destinada a criação de novos modelos (perfis). O *dropbox* na parte inferior lista todos os perfis existentes, inclusive o do modelo acústico genérico. Além disso, é possível continuar a gravação dos textos ainda não gravados ou mesmo regravar os áudios de algum perfil.

A criação de um perfil é iniciada pela seleção do botão `Criar` da Figura 4.6 que gera uma nova janela exibindo algumas informações sobre o processo de gravação (Figura 4.7). Nesta janela é possível gravar ou regravar um texto, além de navegar entre os textos disponíveis (que devem ser gravados).

Ao final da gravação de todos os textos, a janela mostrada na Figura 4.7 estará como mostrado na Figura 4.8, onde o botão `Adaptar` estará habilitado, e quando selecionado inicia o processo de adaptação ao locutor. Durante a adaptação algumas informações do processo serão mostradas na janela, e ao final do processo uma mensagem de sucesso será exibida, como

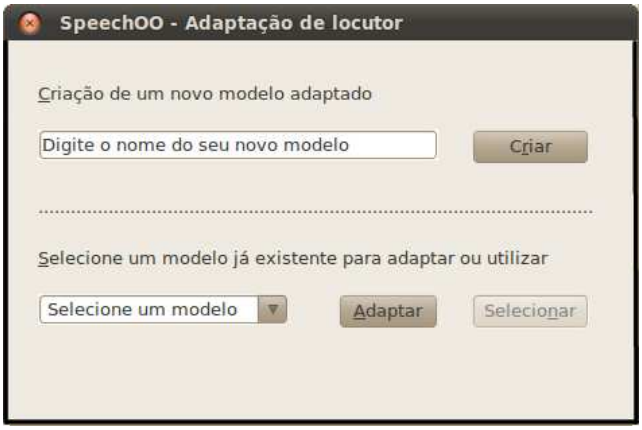

Figura 4.6: Tela para criação ou seleção de um novo perfil.

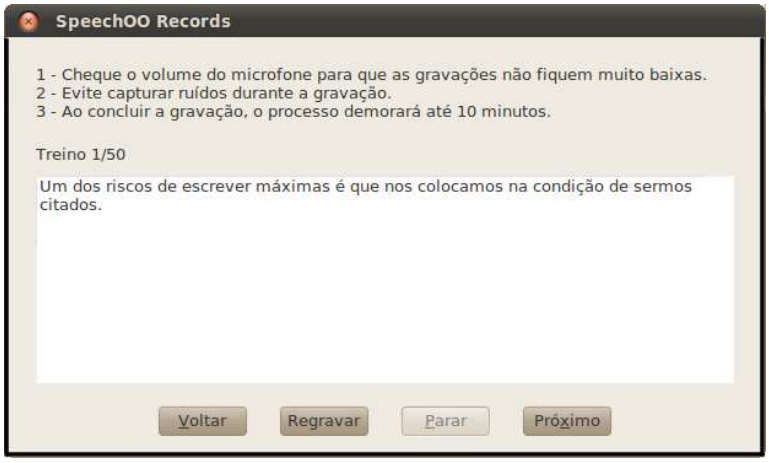

Figura 4.7: Tela com texto para gravação das amostras do perfil.

mostrado na Figura 4.9.

Figura 4.8: Tela de início do processamento dos áudios e criação do modelo acústico do perfil.

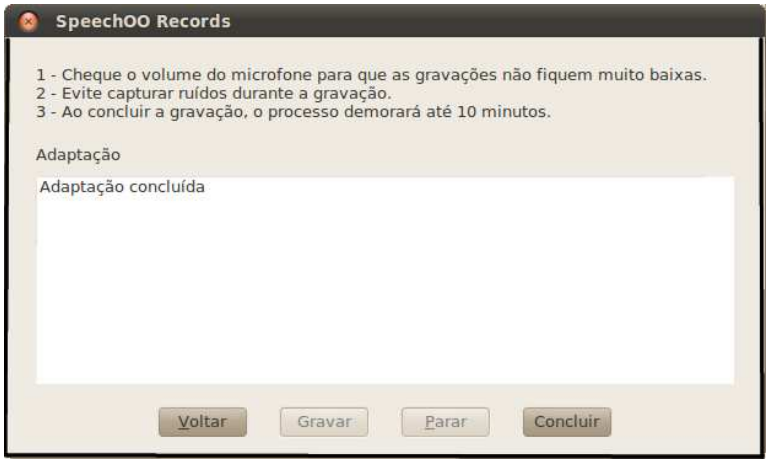

Figura 4.9: Tela sinalizando a conclusão da criação do perfil.

Neste trabalho, o módulo de adaptação desenvolvido em [56] foi utilizado. Sua integração ao SpeechOO requisitou poucas modificações, que foram a correção de erros, e algumas alterações para informar ao usuário o que ocorre durante a execução do processo de adaptação. Já que o processo pode ser demorado (alguns minutos) e se nada fosse mostrado poderia dar a impressão de que o programa estaria travado. A *performance* que pode ser obtida com adaptação de locutor foi detalhada na Seção 3.5.3.

4.4.5 Integração do CoGrOO ao SpeechOO

O CoGrOO é um corretor gramatical acoplável ao LibreOffice e também é um *software* livre que está em constante crescimento devido a grande demanda que o PB traz por ser considerado um idioma de gramaticalmente difícil. É importante notar que o Coruja não comete erros ortográficos, porém erros gramaticais são possíveis e por isso a integração com o CoGrOO.

O SpeechOO foi programado para sempre ativar o CoGrOO realizando a verificação de erros em cada sentença reconhecida. Quando detectado algum erro, uma janela como a apresentada na Figura 4.10 é mostrada para o usuário, sendo possível a escolha da sentença correta.

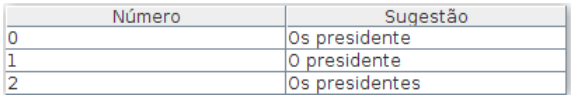

Figura 4.10: Exemplo CoGrOO

No exemplo da Figura 4.10, o SpeechOO reconheceu a frase “Os presidente são legais”

que está gramaticalmente incorreta em número. O corretor realiza a verificação de erros gramaticais e mostra para o usuário algumas possíveis correções. O usuário possui na opção zero a possibilidade de manter o texto original.

4.4.6 Pacotes e Classes

Nesta seção são descritos os pacotes e classes que compõe o projeto do SpeechOO.

4.4.6.1 Pacote `org.speechoo`

Este pacote contém as classes principais da extensão. A classe `CentralRegistrationClass` registra os componentes da extensão na interface do Writer. Já a classe `SpeechOO` é responsável pela inicialização da extensão, e é nesta classe que o reconhecedor é carregado e executado.

4.4.6.2 Pacote `org.speechoo.gui`

Neste pacote estão contidas todas as classes que lidam com a interface do SpeechOO, isto é, posicionamento de botões, tamanho de janelas e etc. A classe `Dialog` implementa a interface UNO e é um modelo para as outras classes do pacote. A classe `TrainingDialog` cria uma janela para simplificar e instruir o usuário ao longo do processo de gravação dos textos para adaptação de locutor e já inclui a API de áudio Java para capturar as amostras de som. A classe `SpeakerAdaptationDialog` é a primeira interface relacionada à criação do perfil do locutor. Esta é responsável por criar os diretórios dos novos perfis e realizar a conclusão ou reconstrução de um perfil, bem como a transição entre perfis.

4.4.6.3 Pacote `org.speechoo.inputText`

Neste pacote estão contidas as classes relacionadas com a escrita do texto na interface do Writer. A classe `InputEditor` é responsável pela execução de todos os comandos de formatação e alterações na fonte do texto utilizando a interface UNO. A classe `InputSentence` tem como função a inserção do texto reconhecido na tela. Esta classe mapeia a posição do cursor e sempre escreve a partir desta, utilizado o CoGrOO para detectar erros gramaticais antes de inserir o texto reconhecido. Outra função desta classe é salvar o arquivo de texto quando requisitado pelo usuário.

4.4.6.4 Pacote `org.speechoo.recognized`

Este pacote contém os *Listeners* do modo ditado e comando. A classe `CommandsListener` tem a função de interpretar e executar todos os comandos do SpeechOO. Já a classe `FreeDictationListener` recebe o resultado do reconhecimento (incluindo hipóteses alternativas).

4.4.6.5 Pacote `org.speechoo.util`

Este pacote contém classes utilitárias responsáveis por tarefas de caráter secundário do *software*, provendo apenas suporte às classes dos demais pacotes. A classe `AcousticModelSelector` altera o arquivo `julius.jconf` para selecionar um determinado perfil de locutor. A classe `Capture` fornece métodos para gravar áudio com as características requisitadas, isto é, é possível especificar parâmetros como taxa de amostragem, *enconding*, tamanho e *endianness* da amostra, além do número de canais. A classe `CoGrOO` recebe a string do reconhecimento e verifica se existe erros gramaticais na sentença. Caso exista, mostra uma tabela com possíveis correções para a seleção de uma delas. A classe `Dispatch` executa comandos no Writer pela interface UNO através da passagem de uma URL UNO. A classe `InputDevicesControl` simula o pressionamento das teclas *backspace* e *enter*. A classe `ReadFromXMLFile` faz a leitura dos textos para gravação (para adaptação ao lucutor) de um arquivo XML. A classe `KeyEvent` cria um *KeyListener* programado para a tecla *Alt Graph* para comutar entre os modos ditado e comando. A classe `Numbers` converte os números reconhecidos em extenso para a forma decimal. A classe `PrintAndSave` pode salvar o documento, além de enviá-lo para impressão. A classe `SpeechPropertiesCreator` cria o arquivo de cadastro da JLaPSAPI para a Java Virtual Machine (JVM). Já a classe `TableGramatical` cria a tabela para os resultados da classe `CogrOO` na interface do Writer. E por fim a classe `TableNames` cria a tabela para os nomes de possíveis fontes a serem utilizadas.

4.5 Conclusão

O SpeechOO vem sendo reestruturado e muitas funcionalidades ainda são desejadas. Para isso uma implementação mais ampla da JSAPI por parte do Coruja é cada vez mais necessária. Dessa forma, a JLaPSAPI vem sendo desenvolvida em conjunto com o SpeechOO. Este projeto é desenvolvido em parceria com Centro de Competência em Software Livre da Universidade de São Paulo, e um projeto para seu desenvolvimento foi aprovado pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) segundo o edital 09/2010 [57].

Capítulo 5

ASR em uma IVR usando Coruja e Asterisk

Este capítulo descreve como foi feita integração do Coruja com o Asterisk, implementação realizada neste trabalho com o objetivo de criar uma solução para atendimento automático de ligações utilizando reconhecimento de voz. Inicialmente o *software* Asterisk é descrito e na sequencia a estratégia adotada para a integração deste *software* ao Coruja é discutida.

5.1 Asterisk

O Asterisk é um *software* livre que pode ser utilizado para diversas aplicações, principalmente em telefonia. O Asterisk é principalmente customizado para ser usado como *internet protocol* (IP) *private branch exchange* (PBX), *gateway* de voz sobre IP (VoIP), unidade de resposta audível (IVR) e distribuidor de chamadas. Segundo a Digium (atual mantedora do Asterisk) o Asterisk é atualmente usado em mais de um milhão de sistemas em mais de 170 países.

O Asterisk deve ser utilizado em sistema operacional Linux, e por possuir uma arquitetura fortemente modular é facilmente customizável, virtualmente, este pode se comunicar com qualquer *hardware*, linguagem de programação e protocolo. Um dos ideais dos desenvolvedores do Asterisk é que o mesmo possa futuramente atender a todos os protocolos existentes.

É sabido que o Asterisk pode se comunicar com a *public switched telephone network* (PSTN) que é basicamente a rede de telefonia que usamos, composta por linhas telefônicas convencionais, cabos de fibra óptica, rede de celulares, satélites de comunicação, dentre outros.

O Asterisk pode também trabalhar como um servidor VoIP principalmente usando o protocolo *session initiation protocol* (SIP), que possibilita, dentre outros a comunicação entre *softphones*, além de ser altamente utilizado por serviços VoIP que se conectam a PSTN como o Skype por exemplo.

5.1.1 Conexão com a PSTN

Uma forma de conectar o Asterisk a PSTN é através de empresas que vendem serviços VoIP, comumente usando protocolo SIP. Através do SIP computadores ou telefones VoIP são capazes de receber e originar chamadas de / para telefones "comuns" (PSTN). Uma outra forma para acessar a PSTN é através de uma linha telefônica comum (que costumamos ter em casa) chamada de interface *foreign eXchange subscriber* (FXS), essa segunda solução foi adotada neste trabalho devido ao laboratório já possuir uma linha FXS, evitando assim custos adicionais.

A linha FXS é analógica e não é diretamente conectada ao computador. Para sua utilização, um conversor analógico digital é necessário, e neste trabalho a placa DigiVoice VB0408PCI foi utilizada. Esta placa possui quatro conectores RJ-45, dois para entrada (CN1 e CN2) e dois para saída (CN3 e CN4) que são um circuito em série com a entrada, isto é, são usadas apenas para gravação. A Figura 5.1 ilustra o esquema da placa [58].

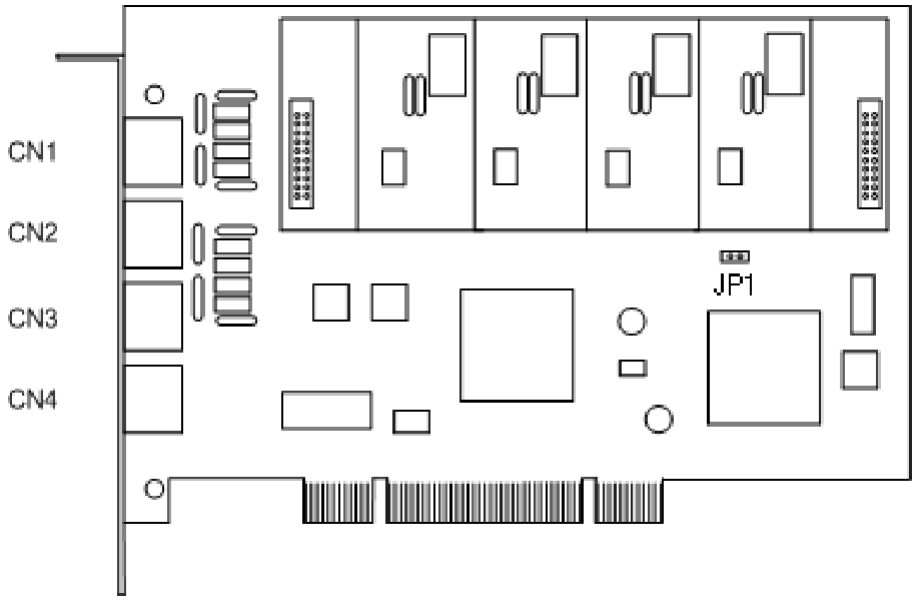

Figura 5.1: Esquema da placa DigiVoice VB0408PCI.

A Figura 5.2 mostra o esquema de numeração dos cabos no conector RJ-45. Assumindo essa convenção os pinos 1 e 2 do CN1 formam o quarto canal, 3 e 4 o terceiro, 5 e 6 o segundo

e 7 e 8 o primeiro. Já os pinos 1 e 2 do CN2 formam o oitavo canal, 3 e 4 o sétimo, 5 e 6 o sexto e 7 e 8 o quinto. Neste trabalho apenas uma linha telefônica foi utilizada no canal 3, então os pinos 3 e 4 do conector CN1 foram utilizados.

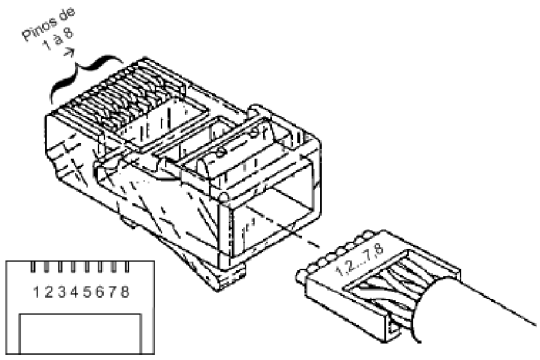

Figura 5.2: Esquema para conexão de canais FXS nos conectores RJ-45 da placa DigiVoice VB0408PCI.

As placas DigiVoice são compatíveis com sistemas operacionais Linux e Windows através do driver VoicerLib, desenvolvido pela DigiVoice. Para a integração com Asterisk um Asterisk Channel também é disponibilizado pela DigiVoice, chamado DgvChannel através deste o Asterisk pode controlar a placa VB0408PCI, com comandos como atender e desligar ligações.

A VoicerLib versão 4.2.4.0 e o DgvChannel versão 1.0.8 são compatíveis apenas com versões 2.6.x do kernel do Linux, fazendo com que este seja utilizado neste trabalho. A versão do Asterisk utilizada foi a 1.8.13.1.

5.1.2 Configuração e arquitetura do Asterisk

O Asterisk pode ser visto como um sistema com duas partes:

Núcleo do Asterisk contém os módulos que lidam com a performance do sistema, gerenciamento de chamadas, *codecs* de entrada e saída, dentre outros.

API de módulos carregáveis para abstração quatro APIs são definidas para módulos carregáveis, facilitando a abstração de *hardware* e protocolos. A Figura 5.3 ilustra esses módulos.

Essa arquitetura modular permite que a integração do Asterisk com novas tecnologias seja implementada sem muito esforço. Por exemplo, caso um novo protocolo ou *hardware* seja

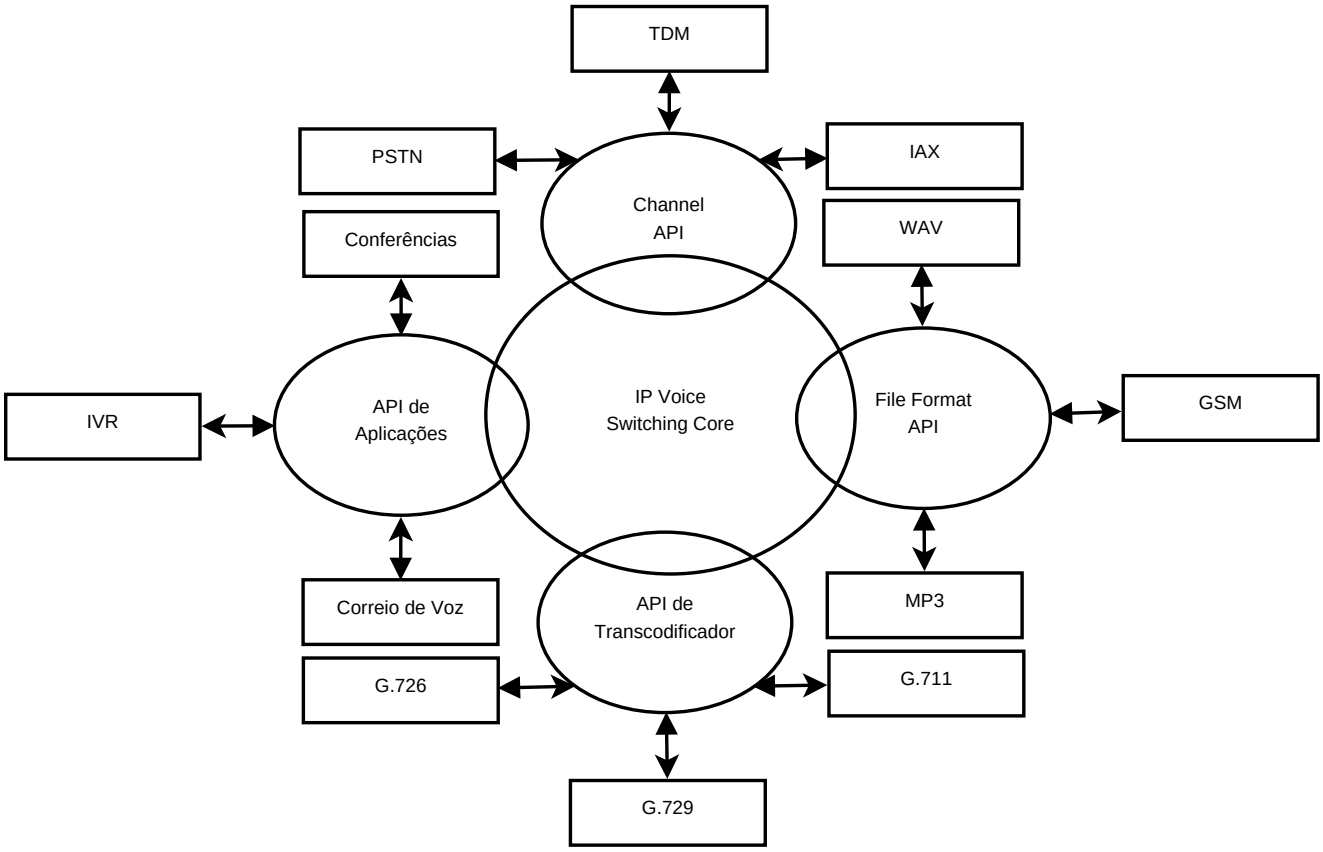

Figura 5.3: Módulos carregáveis do Asterisk.

inventado, basta escrever um módulo da Channel API para que o Asterisk possa se comunicar com essa nova tecnologia.

O Asterisk é controlado por uma série de arquivos de configuração em vários diretórios do Linux. Exemplos de configurações e recursos são: gravações de caixa postal, *prompts* de voz e arquivos de configuração de transferência e fila de chamadas, dentre outras. A seguir são descritos alguns desses diretórios de configuração (assumindo a configuração padrão do Asterisk).

`/etc/asterisk` contém os arquivos de configuração e de lógica do sistema.

`/usr/lib/asterisk/modules` contém os módulos carregáveis do Asterisk que controlam várias de suas funcionalidades.

`/var/lib/asterisk/sounds` contém os arquivos de sons usados no Asterisk, como *prompts* de voz, músicas para chama em espera, dentre outros.

`/var/lib/asterisk/agi-bin` contém os *scripts* Asterisk Gateway Interface (AGI) e Extended AGI (EAGI).

5.1.3 Unidade de resposta audível

Ao chegar no Asterisk uma chamada é direcionada a um contexto especificado em um dialplan. Esse irá especificar o que o sistema deve fazer com a chamada. Exemplos de comandos são: ligar e conectar outro usuário, atender a ligação, tocar uma música, esperar que o usuário digite um número, desligar a ligação, dentre outros. O dialplan é encontrado no arquivo `extensions.conf`.

A chamada pode ser direcionada para diferentes contextos dependendo de sua origem. Neste trabalho a origem da chamada será o canal 3 da placa VB0408PCI como citado na Seção 5.1.1. O arquivo `digivoice.conf` é responsável pelas configurações da placa, inclusive para qual contexto a chamada deve ser direcionada. A parte deste arquivo responsável por essa configuração é mostrada abaixo.

```
[port_config]
signalling=fxo
context=from-pstn
language=pt_BR
ports=>3
```

Como pode ser visto na seção `port_config` deste arquivo, é possível configurar o contexto para o qual a chamada será direcionada, nesse caso escolhemos o contexto `from-pstn`.

5.1.4 Controle da ligação com o dialplan

O dialplan é uma das partes mais importante do Asterisk, já que este é quem faz o controle da ligação. O dialplan é uma linguagem de *script* facilmente customizável, podendo interagir com sistemas externos.

O dialplan contém quatro conceitos: contextos, extensões, prioridades e aplicações, a seguir são brevemente discutidos cada um desses conceitos.

5.1.4.1 Contextos

Um dialplan é separado em seções chamadas de contextos, dentro de um contexto as extensões estão isoladas, isto é, se em um contexto o usuário digitar 0 significa confirma, em outro o digito 0 pode significar não confirma. Obviamente que isso é customizável e o contexto para qual o usuário vai ser direcionado é definido por sua origem. Como mostrado

na Seção 5.1.3 os usuários que ligam para o telefone que está conectado na porta 3 da placa VB0408PCI serão direcionados para o contexto `from-pstn`.

Contextos são definidos com seu nome entre colchetes (`[]`), abaixo é definido o contexto `exemplo`.

```
[exemplo]
```

Todas as instruções contidas depois da definição de um contexto a este pertencem até que um novo contexto seja definido. Existem dois contextos reservados que são o `[general]` e o `[globals]`, o primeiro lida com algumas configurações e o segundo define variáveis visíveis simultaneamente a todos os contextos.

5.1.4.2 Extensões

Em aplicações de telefonia o termo extensão comumente se refere a um identificador numérico que quando discado liga para um telefone. No Asterisk uma extensão é algo mais poderoso, que define passo a passo as ações da ligação.

Dentro de um contexto podemos definir tantas extensões quantas necessário. Quando uma extensão for ativada (seja por uma chamada recebida ou por dígitos discados em um canal) o Asterisk irá seguir os passos definidos para essa extensão.

A sintaxe para uma extensão é a palavra `exten` seguida por uma seta formada pelo sinal de igual (=) e o sinal de maior (>), seguido pelo número (ou nome) da extensão, sua prioridade e finalmente a aplicação (ou comando) que será executado, separados por vírgula como mostrado a seguir.

```
exten => nome,prioridade,aplicação()
```

No exemplo a seguir é mostrada uma aplicação no contexto `exemplo1` que atende uma ligação para o número 12. Nela o nome da extensão é 12, a prioridade é 1 e a aplicação é `Answer()`.

```
[exemplo1]
exten => 12,1,Answer()
```

5.1.4.3 Prioridades

Cada extensão pode conter múltiplos passos, chamados prioridades. As prioridades são numeradas e executadas sequencialmente começando de 1, e cada uma delas executa uma

determinada aplicação. Como exemplo, a extensão a seguir atende a ligação para o número 12 (prioridade 1), espera 5 segundos (prioridade 2) e então a desliga (prioridade 3).

```
[exemplo2]
exten => 12,1,Answer()
exten => 12,2,Wait(5)
exten => 12,3,Hangup()
```

5.1.4.4 Aplicações

As aplicações são as ações que as extensões vão executar. Cada aplicação faz uma determinada ação no canal aberto, como por exemplo: tocar um som, receber um número discado, consultar um banco de dados, discar para outro canal, desligar a ligação, dentre outros.

Nas Seções 5.1.4.2 e 5.1.4.3 foram apresentadas as aplicações `Answer()`, `Wait()` e `Hangup()` responsáveis por atender a ligação, esperar por um tempo determinado e desligar a ligação, respectivamente. Algumas aplicações como a `Answer()` não precisam de argumentos, porém para outras como a `Wait()` são necessários um ou mais argumentos para seu funcionamento, estes devem ser separados por vírgula.

Através das aplicações o Asterisk pode se conectar com qualquer linguagem de programação que escreva e leia em sua entrada e saída padrão respectivamente. Dessa forma durante uma ligação qualquer programa pode ser acessado pelo Asterisk, o que possibilita esse tipo de operação é a chamada Asterisk Gateway Interface (AGI).

A AGI é projetada para enviar comando para o Asterisk escrevendo na saída padrão, e receber as respostas em sua entrada padrão. Para executar um *script* AGI a aplicação AGI deve ser chamada da seguinte forma `AGI(command[,arg1[,arg2[,...]]])` os argumentos são passados para o *script* como se fossem argumentos da linha de comando. Inicialmente, além de receber os argumentos passados pela aplicação AGI, uma série de variáveis de ambiente são escritas na entrada padrão do *script* AGI, no formato `variável_agi: valor`. Dentre as variáveis de ambiente está o número do usuário que originou a chamada (`agi_callerid`) e o contexto atual (`agi_context`).

A seguir é mostrado um *script* AGI em Python (chamado `say_wait.py`) que quando chamado espera por `arg1` segundos, fala os caracteres `arg2` e retorna.

```
1 import sys, time
2
```

```
3  env_vars = dict()
4  while True:
5          line = sys.stdin.readline()
6          if line == '\n':
7                  break
8          else:
9                  var, _, value = line.partition(':')
10                 env_vars[var] = value
11
12 time.sleep(sys.argv[1])
13
14 sys.stdout.write('SAY ALPHA "%s"\n' % sys.argv[2])
15 sys.stdout.flush()
16 status, _, result = sys.stdin.readline().partition('=')
17 if status != '200 result' or result != 0:
18         sys.stderr('Erro no comando SAY ALPHA\n')
19         sys.exit(result)
```

Nas linhas 3 a 10 são lidas as variáveis de ambiente. Na linha 12 o *script* espera por `arg1` segundos. Nas linhas 14 a 16 é mostrado como o *script* AGI interage com o Asterisk, na linha 14 é escrito na saída padrão do *script* o comando a ser lido pelo Asterisk, na 15 é garantido que o comando não ficará em um *buffer* e na linha 16 é lido o resultado. O comando a ser executado nesse exemplo é o `SAY ALPHA` que falará cada um dos caracteres passados como parâmetro para o comando, após o Asterisk falar todos os caracteres é escrito na entrada padrão do *script* o retorno do comando, o formato padrão para retorno de comandos é `<code> result=<result> [data]` onde `<code>` é o retorno da comunicação com o Asterisk sendo que 200 representa sucesso e 5xx erros, já o valor `<result>` é o retorno do comando, comumente -1 significa erro e 0 sucesso, alguns comandos retornam valores extras no campo opcional `data`. O retorno do comando é processado nas linhas 17 a 19.

Uma variação da AGI é a Extended AGI (EAGI) que além de todos os recursos disponibilizados pela AGI possibilita o acesso ao canal de áudio pelo descritor de arquivo 3, para exemplificar abaixo é mostrado um exemplo (chamado `record.py`) de código que grava 5 segundos de áudio.

```
1  import sys, os
2
3  env_vars = dict()
```

```
4  while True:
5          line = sys.stdin.readline()
6          if line == '\n':
7                  break
8          else:
9                  var, _, value = line.partition(':')
10                 env_vars[var] = value
11
12 channel = os.fdopen(3, 'r')
13 with open('/tmp/file.raw', 'w') as out_file:
14         out_file.write(channel.read(2 * 8000 * 5))
```

Novamente nas linhas 3 a 10 são lidas as variáveis de ambiente, em seguida na linha 12 o canal é aberto através do descritor de arquivo 3, o arquivo `/tmp/file.raw` é criado na linha 13 e finalmente na linha 14 são lidos 80000 *bytes*, já que temos 2 *bytes* por amostra, cada segundo contém 8000 amostras e queremos gravar 5 segundos. As amostras são inteiros de 16 bits com sinal e o som possui 1 canal.

Para exemplificar a utilização de *scripts* AGI e EAGI abaixo é mostrado um exemplo de dialplan que atende a ligação, espera 2 segundos, fala os caracteres ABC, grava 5 segundos da ligação e então desliga.

```
[exemplo3]
exten => 12,1,Answer()
exten => 12,2,AGI(say_wait.py,2,ABC)
exten => 12,3,EAGI(record.py)
exten => 12,4,Hangup()
```

Existem vários comandos AGI para interação com o Asterisk, porém não iremos detalhá-los aqui. A seguir são descritas algumas aplicações utilizadas neste trabalho.

`Answer()` Atende a ligação.

`Playback(arquivo_de_audio)` Toca o áudio do parâmetro `arquivo_de_audio`.

`Set(var=valor)` Atribui o valor do parâmetro `valor` a variável do parâmetro `var`.

`Goto(contexto,extensão,prioridade)` Direciona a chamada para o contexto, extensão e prioridade especificados.

`GotoIf(expressão?destino1:destino2)` Se a expressão `expressão` for verdadeira a chamada é direcionada para o destino `destino1` se for falsa para o destino `destino2`. Qualquer um dos destinos `destino1` e `destino2` é formado pela tupla: `contexto,extensão,prioridade` sendo que o contexto e/ou a extensão podem ser omitidos e assumido o estado atual.

`Hangup()` Desliga a ligação.

`AGI(command[,arg1[,arg2[,...]]])` Executa o *script* AGI `command` que recebe como argumento os n parâmetros passados para a aplicação AGI depois de `command`.

`EAGI(command[,arg1[,arg2[,...]]])` Executa o *script* EAGI `command` que recebe como argumento os n parâmetros passados para a aplicação EAGI depois de `command`.

5.2 Integração do decodificador Julius com Asterisk

5.2.1 Conversor de gramáticas SAPI XML para Julius Grammar

O conversor de gramáticas citado na Seção 3.4 foi reescrito em linguagem de programação Python, suportando as *tags* `GRAMMAR`, `RULE`, `PHRASE`, `LIST` e `OPT`. Além disso uma nova *tag* chamada `RETURN` foi criada para possibilitar o retorno de uma *string* arbitrária para uma dada regra. Assim, ao executar o conversor de gramáticas, além dos arquivos de gramática do Julius um novo arquivo com a extensão `.ret` será criado, este contém informações para o retorno de cada uma das regras da gramática. Esse novo conversor foi chamado sapixml2julius.

5.2.2 *Plugin* para entrada de áudio do Julius

Neste trabalho optou-se por integrar o decodificador Julius com o Asterisk através de uma EAGI, para isso é necessário que o Julius utilize como entrada de áudio o descritor de arquivo 3, que é o canal de áudio disponibilizado pelo Asterisk na EAGI, como mostrado na Seção 5.1.4.4.

Para isso foi desenvolvido um *plugin* para o decodificador Julius, este além de receber a entrada de áudio no descritor de arquivo 3 salva todo o áudio em um arquivo indicado pela variável de ambiente `RAWFILE` que deve ser inicializada antes de executar o Julius com o *plugin*.

5.2.3 *Script* EAGI para integração com o Julius

O *script* EAGI desenvolvido utiliza o Julius para reconhecer de uma determinada gramática e retorna o resultado em algumas variáveis de contexto. Este deve ser chamado em uma extensão da seguinte forma `EAGI(adintool.py,arquivo_jconf,prefixo_gramática)`, onde `arquivo_jconf` é o arquivo para configuração do Julius e `prefixo_gramática` é o prefixo dos arquivos de gramática do Julius (`.dfa`, `.dict`, `.voca` e `.ret`). O decodificador Julius será executado e retornará um resultado que contém as seguintes informações: resultado do reconhecimento, retorno definido pela regra XML, confiança do reconhecimento por palavra e confiança do reconhecimento nas variáveis `RECRESULT`, `RECRET`, `RECCONFIDENCEPERWORD` e `RECCONFICENCE` respectivamente.

A seguir é apresentado o código da EAGI responsável pela integração.

```
1  import sys, os, subprocess, re, tempfile
2
3  plugindir="/projetos/voz/julius-asterisk/adinplugin"
4
5  def exec_agi(command):
6    sys.stdout.write(str(command) + '\n')
7    sys.stdout.flush()
8    sys.stdin.readline()
9
10 def main():
11   env_vars = dict()
12   while True:
13     line = sys.stdin.readline()
14     if line == '\n':
15       break
16     else:
17       var, _, value = line.partition(':')
18       env_vars[var] = value
19
20   jconf = sys.argv[1]
21   grammar = sys.argv[2]
22
23   with open(grammar + '.voca') as f:
24     voca = [w.strip() for w in f.readlines()]
```

```
25  with open(grammar + '.ret') as f:
26    ret = dict()
27    for t in re.findall('^(?P<id>\w+)\s<>(?P<ret>.*?)<>',
28        ''.join(f.readlines()), re.MULTILINE | re.DOTALL):
29      ret[t[0]] = t[1]
30
31  command = """julius -separatescore -fallback1pass -nosectioncheck -C {0}
32     -gram {1} -plugindir {2} -input psb""".format(jconf, grammar, plugindir)
33
34  raw_file = tempfile.NamedTemporaryFile(suffix='.wav', delete=False)
35  julius = subprocess.Popen(command.split(), stdin=subprocess.PIPE,
36      stdout=subprocess.PIPE, stderr=2, env=dict(RAWFILE=raw_file.name))
37
38  while True:
39    line = julius.stdout.readline()
40    if line == '':
41      break
42    sys.stderr.write(line)
43    sys.stderr.flush()
44    result = re.search('^sentence1: (.*?)$', line, re.DOTALL)
45    if result is not None:
46      for i in xrange(3):
47        line = julius.stdout.readline()
48        sys.stderr.write(line)
49        sys.stderr.flush()
50        if i == 0:
51          wseq = map(int,
52              re.search('^wseq1: (.*?)$', line, re.DOTALL).group(1).split())
53      score = re.search('^cmscore1:(?P<score>( (\d\.\d{3}))+)$',
54          line, re.DOTALL)
55      if score is not None:
56        julius.kill()
57        score = score.group('score')
58        result = result.group(1)
59        break
60
```

```
61    exec_agi('SET VARIABLE RECRESULT "%s"' % result)
62    exec_agi('SET VARIABLE RECRET "%s"' %
63        ret[re.match('^#(.*)_WORD_\d+$', voca[wseq[1]]).group(1)])
64    exec_agi('SET VARIABLE RECCONFIDENCEPERWORD "%s"' % score)
65    exec_agi('SET VARIABLE RECCONFIDENCE "%s"' %
66        (sum(map(float, score.split())) / len(score.split())))
67
68  if __name__=='__main__':
69    main()
```

Na linha 3 é definido o diretório que contém o `plugin` escrito para o Julius como apresentado na Seção 5.2.2. Nas linhas 5 a 8 é definida a função `exec_agi` que facilita a execução de comandos AGI, por simplicidade nenhuma checagem é feita no retorno do comando. Nas linhas 11 a 18 são lidas as variáveis de ambiente e armazenadas no dicionário `env_vars`. Nas linhas 20 e 21 são lidos os argumentos `arquivo_jconf` e `prefixo_gramática` respectivamente. Nas linhas 23 a 29 são lidos os arquivos `.voca` e `.ret` que especificam o retorno de acordo com a regra reconhecida. As linhas 31 e 32 compõem o comando que executará o decodificador Julius. A linha 34 cria um arquivo temporário que armazenará todo o áudio que o Julius utilizar, finalmente nas linhas 35 e 36 o Julius é executado em uma nova *thread*, observamos que este é executado com uma variável de ambiente `RAWFILE` especificando o local para armazenar o áudio, o que é esperado segundo a Seção 5.2.2. O laço das linhas 38 a 59 processam em tempo real a saída da *thread* de execução do Julius, e quando um resultado é encontrado este encerra o processo do decodificador, armazenando nas variáveis `result`, `wseq` e `score` o resultado do reconhecimento, a sequência dos identificadores das palavras reconhecidas e o *score* de cada uma das palavras, respectivamente. Por fim nas linhas 61 a 66 são atribuídos as variáveis de contexto os resultados esperados.

5.3 Reconhecendo com o Julius a partir do Asterisk

Para demonstrar a utilização do decodificador Julius com o Asterisk será apresentado um pequeno exemplo que integra a maioria dos recursos apresentados.

Imagine que em um dado momento de uma ligação é solicitado que um usuário confirme seu endereço. Se este estiver correto o usuário deve confirmar senão deve negar e a chamada será encaminhada para um atendente.

A gramática proposta para este exemplo é mostrada a seguir.

```
<GRAMMAR>
<RULE ID="1" TOPLEVEL="ACTIVE">
        <LIST>
                <PHRASE> isso <OPT> mesmo </OPT></PHRASE>
                <PHRASE> positivo </PHRASE>
                <PHRASE> correto </PHRASE>
                <PHRASE> confirmo </PHRASE>
                <PHRASE> confirmado </PHRASE>
                <PHRASE> confirma </PHRASE>
                <PHRASE> certo </PHRASE>
                <PHRASE> sim
                        <OPT> sim </OPT>
                        <OPT>
                                <LIST>
                                        <PHRASE> confirmo </PHRASE>
                                        <PHRASE> confirmado </PHRASE>
                                        <PHRASE> confirma </PHRASE>
                                </LIST>
                        </OPT>
                        <OPT> sim </OPT>
                </PHRASE>
        </LIST>
        <RETURN> 199998 1008 1 </RETURN>
</RULE>
<RULE ID="2" TOPLEVEL="ACTIVE">
        <LIST>
                <PHRASE> não <OPT> confirmo </OPT> <OPT> não </OPT> </PHRASE>
                <PHRASE> negativo </PHRASE>
                <PHRASE> incorreto </PHRASE>
                <PHRASE> errado </PHRASE>
        </LIST>
        <RETURN> 199999 1008 0 </RETURN>
</RULE>
</GRAMMAR>
```

Esta gramática possibilita que o usuário fale frases para confirmar como por exemplo “sim confirmo”, “confirmado”, “sim confirma sim”, “sim sim”, dentre outras, como ilustrado na

Figura 5.4. Para negar, o usuário pode falar frases como "não", "incorreto" dentre outras, como ilustrado na Figura 5.5. É importante notar que a regra de confirmação possuí um retorno único para qualquer que seja a frase, assim como a regra de negação.

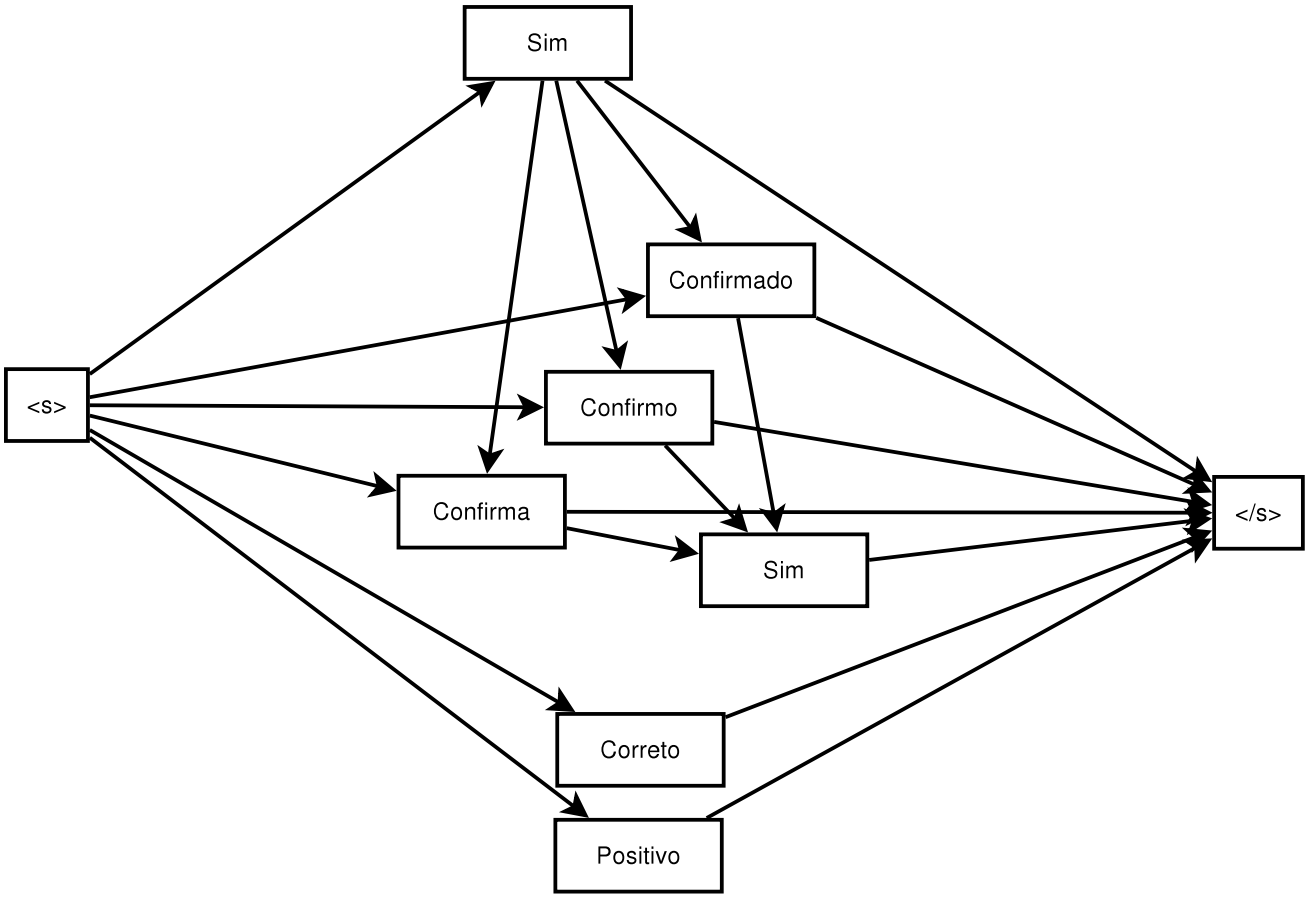

Figura 5.4: Diagrama para a regra de confirmação.

Assumindo que o arquivo que contém a gramática XML é chamado `sim_nao.xml` e que o conversor sapixml2julius está instalado no sistema, podemos compilar a gramática executando o seguinte comando `pysapixml2julius -p sim_nao` e a gramática será convertida e compilada, ficando pronta para ser usada com o Julius.

O próximo passo é definir o dialplan para receber a ligação, este deve inicialmente tocar um *prompt* confirmando o endereço do usuário, iniciar o reconhecedor para esperar a resposta e de acordo com esta direcionar ou não o usuário para um atendente. O dialplan que atende a esses requisitos é mostrado a seguir.

```
[exemplo]
exten => 12,1,Answer()
exten => 12,2,Playback(confirma_endereco)
```

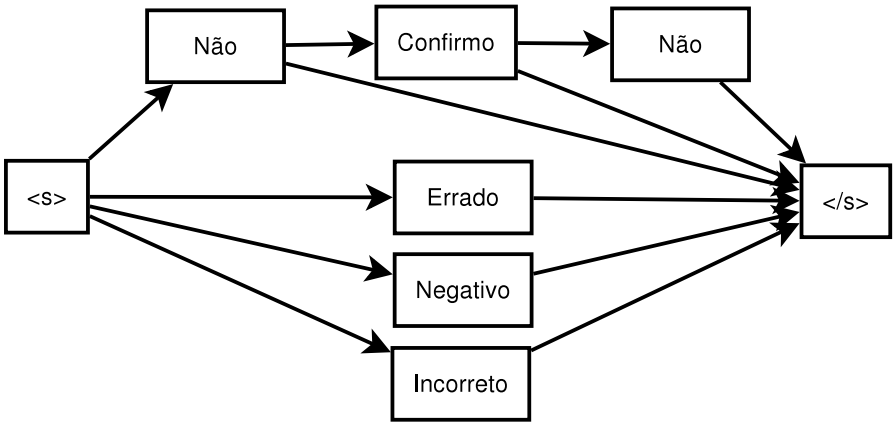

Figura 5.5: Diagrama para a regra de negação.

```
exten => 12,3,EAGI(adintool.py,coruja_asterisk.jconf,sim_nao)
exten => 12,4,Set(CONF=${RECRET:12:1})
exten => 12,5,GotoIf($[${CONF} = 1]?7:6)
exten => 12,6,Goto(atendente,1,1)
exten => 12,7,Hangup()
```

Como podemos observar a prioridade 2 toca um arquivo de áudio que deve solicitar a confirmação do endereço do usuário, em seguida a prioridade 3 faz o reconhecimento da resposta do usuário, o resultado do reconhecimento é utilizado na prioridade 4 onde é atribuído a variável `CONF` o 13º caractere do retorno da regra, isto é, para a regra de confirmação é atribuído 1 e para negação 0. Na prioridade 5 é testado se o usuário confirmou ou não, caso tenha confirmado a ligação é encerrada na prioridade 7, caso contrário a ligação é direcionada para um atendente (assumindo que existe a extensão `atendente` com nome e prioridade 1) e só quando a interação com o atendente acabar é que a ligação será encerrada na prioridade 7.

5.4 Conclusão

Neste capítulo foi feita uma introdução ao Asterisk e apresentado o desenvolvimento de dialplan para este. Com essa informação a criação de unidades de resposta audível para atendimento eletrônico de *call centers* é possível. Também foi mostrado como é realizada a conexão do Asterisk com a PSTN através de linhas analógicas, e que uma vez conectado este pode receber e originar ligações para qualquer telefone.

A integração do Asterisk com o reconhecedor Julius e o Coruja também foi discutida neste capítulo. Foi mostrado como a integração destes *softwares* foi feita e como usar o Julius

em um dialplan do Asterisk. A solução apresentada apesar de usar o Português Brasileiro pode ser utilizada em qualquer lingua desde que para esta se tenha modelo acústico para o decodificador Julius.

Capítulo 6

Implementação da IVR do disque 100

Este capítulo descreve uma aplicação para atendimento automático criada com base na solução proposta no Capítulo 5. Mais que uma prova de conceito, este capítulo expõe uma parceria de sucesso entre academia e mercado. Primeiramente, o Dique Denuncia Nacional (DDN) é descrito e em seguida a aplicação aqui desenvolvida é apresentada.

6.1 O disque 100

O Disque Denuncia Nacional (DDN) é um serviço da Secretária de Direitos Humanos da Presidência da República. Para acessar o DDN basta ligar de qualquer telefone gratuitamente para o número 100. O principal foco do DDN é a denuncia de abuso contra crianças e adolescentes. Dentre os serviços prestados pelo disque 100 está o fornecimento do número do conselho tutelar de alguma cidade.

Inicialmente pode-se pensar em uma solução clássica para este serviço que é o uso de tons, quando o sistema lista opções ao usuário e cada opção está associada a um dígito no telefone, como por exemplo: para a cidade 1 digite 1, para cidade 2 digite 2, e assim por diante. Porém, no caso do disque 100, este tipo de solução é inviável já que o número de cidades é muito grande. É justamente ai que entra a solução por reconhecimento de voz, onde o usuário pode simplesmente falar a cidade para a qual deseja saber o número do telefone.

A seguir é apresentada uma solução para o problema acima descrito.

6.2 As gramáticas para reconhecimento

O serviço de consulta ao número do conselho tutelar de municípios disponibilizado no disque 100 oferece cobertura a todos os estados do Brasil. O uso de uma única gramática com todas as cidades do Brasil pode tornar a tarefa de reconhecimento mais difícil, visto o grande número de municípios que seriam previstos. Dessa forma, para melhorar a performance do sistema aqui desenvolvido, as cidades foram agrupadas por DDD, sendo então criadas gramáticas separadas para cada grupo, permitindo que o sistema possa, com base no DDD do usuário, carregar uma gramática mais específica, facilitando a tarefa do reconhecedor. Se o usuário está ligando por exemplo do DDD 91, o sistema espera que o usuário fale uma das cidades que são cobertas por este DDD.

A seguir é mostrada uma parte da gramática para o DDD 91.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<GRAMMAR>
        <RULE ID="0" TOPLEVEL="ACTIVE">
                <PHRASE> Belém </PHRASE>
                <RETURN> 102436 1001 91 32191203 Belém </RETURN>
        </RULE>
        <RULE ID="5" TOPLEVEL="ACTIVE">
                <PHRASE> Igarapé-Açu </PHRASE>
                <RETURN> 102470 1001 91 34412017 Igarapé-Açu </RETURN>
        </RULE>
        <RULE ID="6" TOPLEVEL="ACTIVE">
                <PHRASE> Santa Maria <OPT> do Pará </OPT> </PHRASE>
                <RETURN> 102528 1001 91 34421413 Santa Maria do Pará </RETURN>
        </RULE>
        <RULE ID="62" TOPLEVEL="ACTIVE">
                <PHRASE> São Domingos <OPT> do Capim </OPT> </PHRASE>
                <RETURN> 102535 1001 91 91138896 São Domingos do Capim </RETURN>
        </RULE>
        <RULE ID="68" TOPLEVEL="ACTIVE">
                <PHRASE> Santa Luzia <OPT> do Pará </OPT> </PHRASE>
                <RETURN> 102526 1001 91 91850551 Santa Luzia do Pará </RETURN>
        </RULE>
        <RULE ID="69" TOPLEVEL="ACTIVE">
                <PHRASE> Concórdia do Pará </PHRASE>
```

```
            <RETURN> 102457 1001 91 91931804 Concórdia do Pará </RETURN>
        </RULE>
</GRAMMAR>
```

Nesta gramática é importante notar que cada regra representa uma cidade e o retorno destas é padronizado. Os 6 primeiros caracteres do retorno de cada regra significam um código de palavra (`CD_PALAVRA`); os caracteres de 8 a 11 são um código do serviço solicitado (`SERVICO`); os caracteres 13 e 14 são o código do assunto (`ASSUNTO`); os caracteres de 16 a 24 são o número do conselho tutelar (`TELEFONE`) e os caracteres restantes são a cidade correspondente (`PALAVRA`).

Dessa forma foram criadas 68 gramáticas contemplando cada região do país, e os arquivos foram nomeados de acordo com os DDD da seguinte forma `lista_xx.xml` onde `xx` representa do DDD de dois dígitos.

6.3 O dialplan para atendimento automático

O fluxograma mostrado na Figura 6.1 apresenta o dialplan desenvolvido para atender ao sistema do disque 100. A seguir é descrito e mostrado o código que implementa o sistema apresentado.

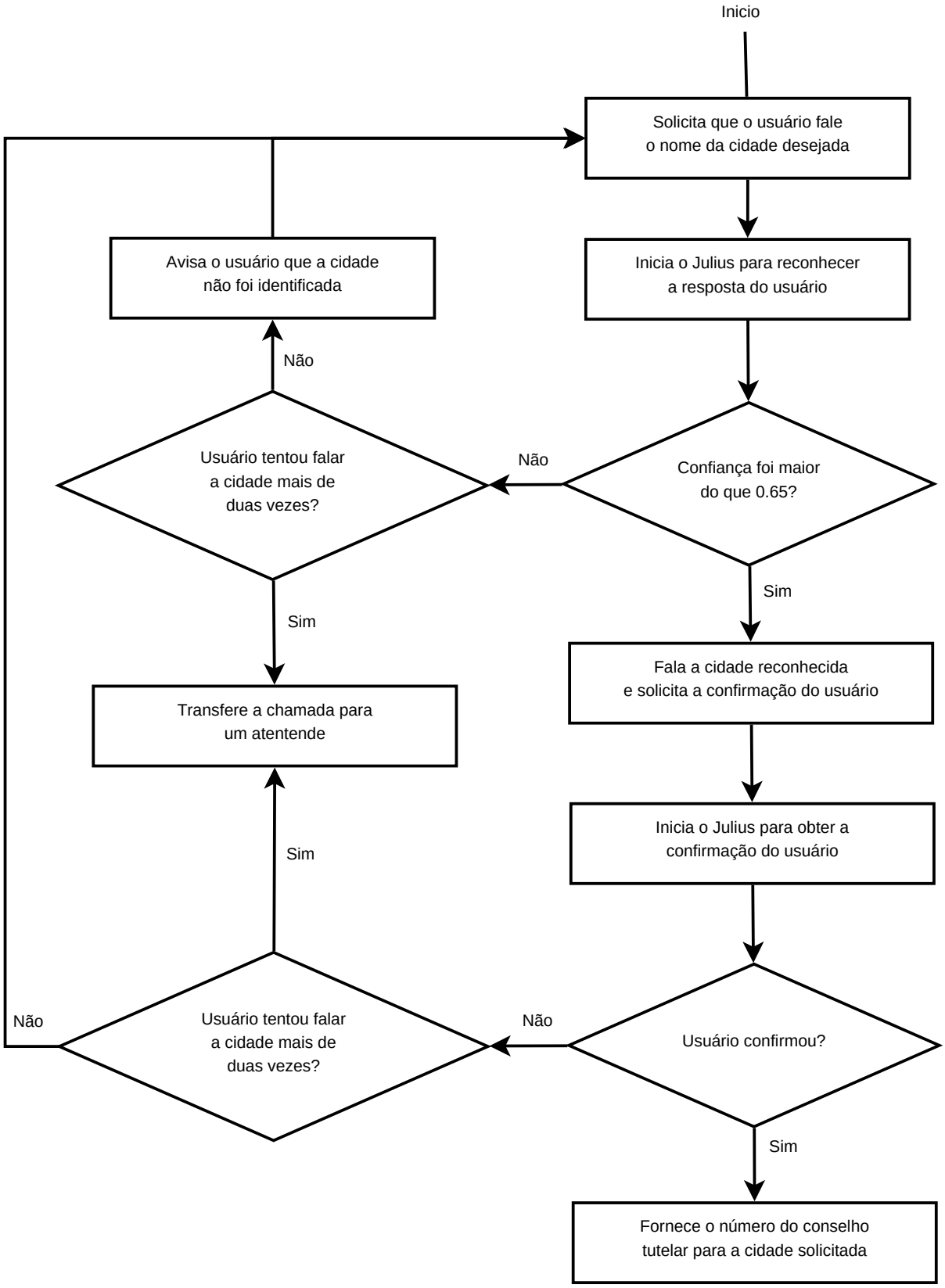

Figura 6.1: Fluxograma do sistema desenvolvido para o atendimento do disque 100.

O atendimento do disque 100 é feito da seguinte forma: é solicitado que o usuário fale o nome da cidade para o qual deseja saber o número do conselho tutelar (prioridade 2), em seguida o Julius é instanciado com a gramática referente ao DDD do usuário e seu resultado armazenado nas variáveis de contexto citadas na Seção 6.2 (prioridades 4 a 9). Caso a confiança do reconhecedor seja menor do que 0.65 (prioridade 10) o sistema checa se o usuário já tentou falar o nome da cidade mais de duas vezes (prioridades 11 e 12), caso tenha tentado mais de duas vezes o usuário é encaminhado para um atendente (prioridade 13), caso contrário o sistema avisa que a cidade não foi identificada (prioridade 14) e retorna a prioridade 2 (prioridade 15). Voltando a prioridade 10, se a confiança do reconhecedor é maior que 0.65 o sistema pede para o usuário confirmar a cidade falando a cidade identificada (prioridade 16 a 18), o Julius é então novamente inicializado (prioridade 19) agora com a gramática `sim_nao.xml` (mostrada na Seção 5.3). Se a cidade não for confirmada (prioridade 20) o sistema checa se o usuário já tentou falar a cidade mais de duas vezes (prioridades 21 e 22), se sim a chamada é encaminhada para um atendente (prioridade 13), senão o sistema retorna a prioridade 2. Voltando a prioridade 20, se a cidade for confirmada o sistema fala o número do telefone do conselho tutelar (prioridades 23 e 24) espera por 0.5 segundos (prioridade 25) e fala novamente o número do conselho tutelar (prioridades 26 a 28) e finalmente a chamada é encerrada na prioridade 29.

```
[mn_principal_sedh_ufpa]
exten => 1,1,Set(TENTATIVAS=0)
exten => 1,2,Playback(/var/lib/asterisk/sedh/Nome_cidade)
exten => 1,3,EAGI(adintool.py,lapsam.x86.jconf,lista_${EXTEN:4:2})
exten => 1,4,Set(CD_PALAVRA=${RECRET:1:6})
exten => 1,5,Set(SERVICO=${RECRET:7:4})
exten => 1,6,Set(ASSUNTO=${RECRET:12:2})
exten => 1,7,Set(TELEFONE=${RECRET:15:9})
exten => 1,8,Set(PALAVRA=${RECRET:24})
exten => 1,9,Set(SCORE=${RECCONFIDENCE})
exten => 1,10,GotoIf($[${SCORE} > 0.65]?16:11)
exten => 1,11,Set(TENTATIVAS=$[${TENTATIVAS} + 1])
exten => 1,12,GotoIf($[${TENTATIVAS} > 1]?13:14)
exten => 1,13,Goto(mn_principal_sedh_ufpa,t,1)
exten => 1,14,Playback(/var/lib/asterisk/sedh/cidade_nao_identificada)
exten => 1,15,Goto(mn_principal_sedh_ufpa,1,2)
exten => 1,16,Playback(/var/lib/asterisk/sedh/vocefalou)
exten => 1,17,VerbioPrompt(${PALAVRA})
```

```
exten => 1,18,Playback(/var/lib/asterisk/sedh/Voce_confirma)
exten => 1,19,EAGI(adintool.py,lapsam.x86.jconf,sim_nao);
exten => 1,20,GotoIf($[${RECRET:12:2} = 1]?23:21)
exten => 1,21,Set(TENTATIVAS=$[${TENTATIVAS} + 1])
exten => 1,22,GotoIf($[${TENTATIVAS} > 1]?13:2)
exten => 1,23,Playback(/var/lib/asterisk/sedh/fone_conselho_tutelar)
exten => 1,24,VerbioPrompt(${TELEFONE})
exten => 1,25,Wait(0.5)
exten => 1,26,Playback(/var/lib/asterisk/sedh/repetindo)
exten => 1,27,VerbioPrompt(${TELEFONE})
exten => 1,28,Playback(/var/lib/asterisk/sedh/sedh_agradece_fem)
exten => 1,29,Hangup()
```

Além das aplicações apresentadas na Seção 5.1.4.4, no exemplo apresentado foi utilizada a aplicação `VerbioPrompt` que é uma aplicação proprietária que sintetiza o texto passado como argumento.

6.4 Validação do sistema

Nesta seção são apresentados os experimentos realizados com o sistema implementado usando a gramática do DDD 91 que possui 76 cidades. Nos testes, 3 abordagens foram utilizadas: com *softphone*, com telefone fixo VoIP e com celular, visando obter resultados do sistema nas mais variadas situações. Os testes aqui reportados foram realizados por uma empresa.

Todos os testes foram conduzidos em um ambiente não controlado – um escritório – onde existem os mais variados ruídos, como por exemplo: ar condicionados, pessoas conversando, dentre outros. A seguir são detalhados cada um dos resultados.

6.4.1 Testes com *softphone*

A primeira abordagem foi a utilização de um *softphone* onde a voz do usuário é capturada através de um *headset* comum. Neste cenário, o teste foi executado por 3 locutores que repetiram cada cidade 3 vezes para o sistema, sendo dois locutores femininos e um masculino. O resultado para o locutor masculino é apresentado na Figura 6.2. Podemos observar que em 92% dos casos, o sistema acertou a cidade solicitada já na primeira tentativa. Já no geral

(acertou todas as vezes que o usuário falou a cidade) o sistema apresentou 80% de taxa de acerto. As Figuras 6.3 e 6.4 mostram os resultados para as vozes femininas, onde o resultado geral foi de 59% e 78% respectivamente. O pior resultado foi encontrado na voz feminina 1 da Figura 6.3, porém até mesmo neste resultado observamos que em pelo menos uma das tentativas o sistema acertou, o que mostra que no pior dos casos o usuário precisaria tentar 3 vezes para conseguir completar sua solicitação.

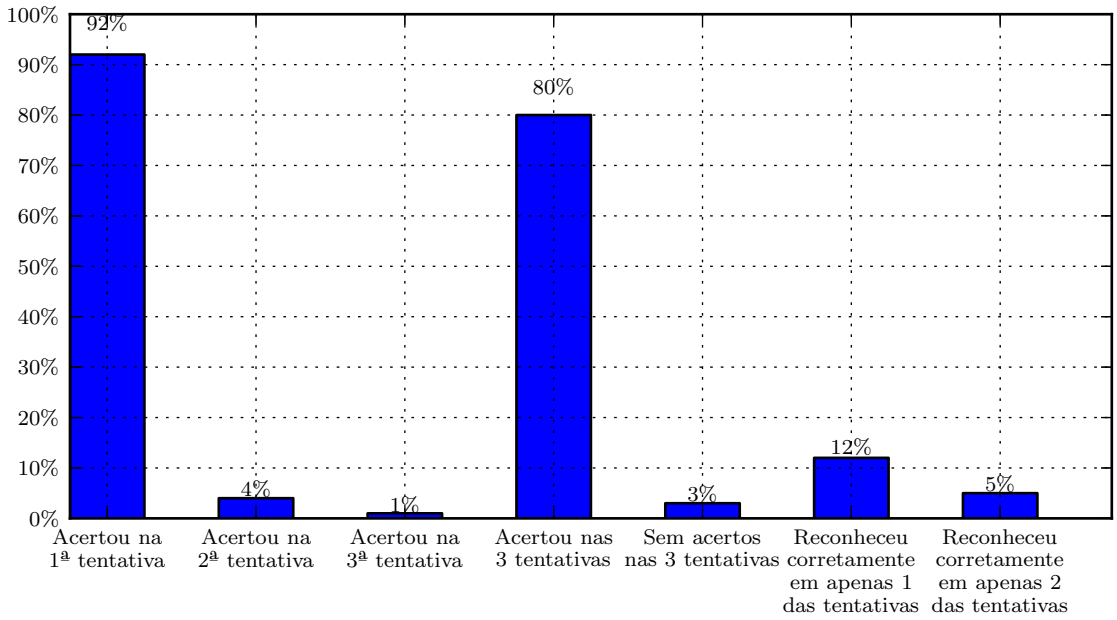

Figura 6.2: Resultado do teste falando 3 vezes cada cidade em um *headset* com a voz masculina.

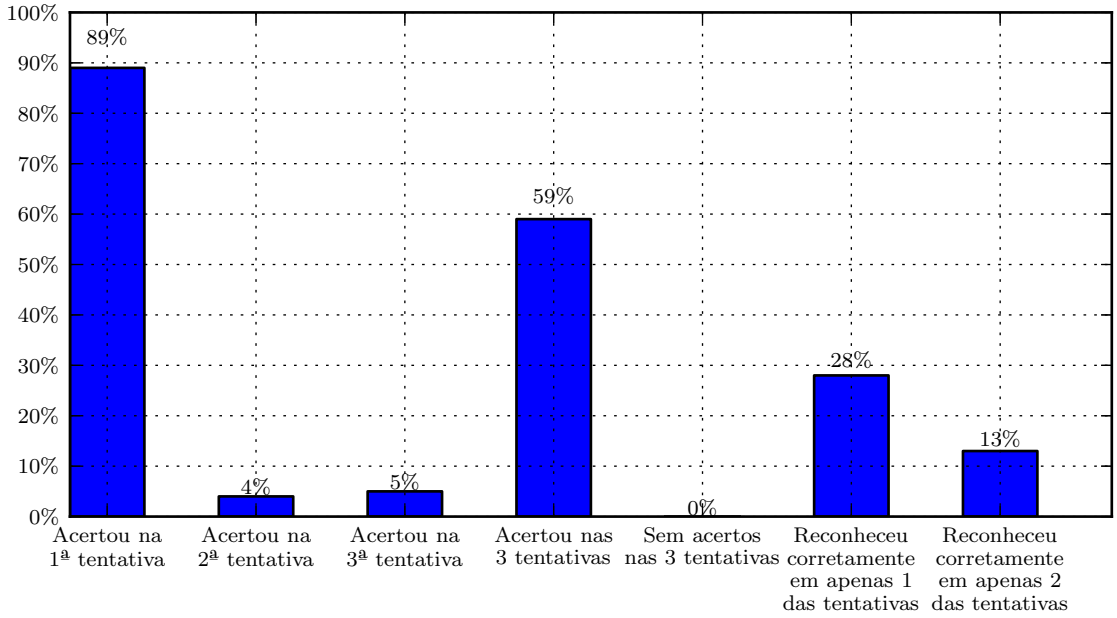

Figura 6.3: Resultado do teste falando 3 vezes cada cidade em um *headset* com a voz feminina 1.

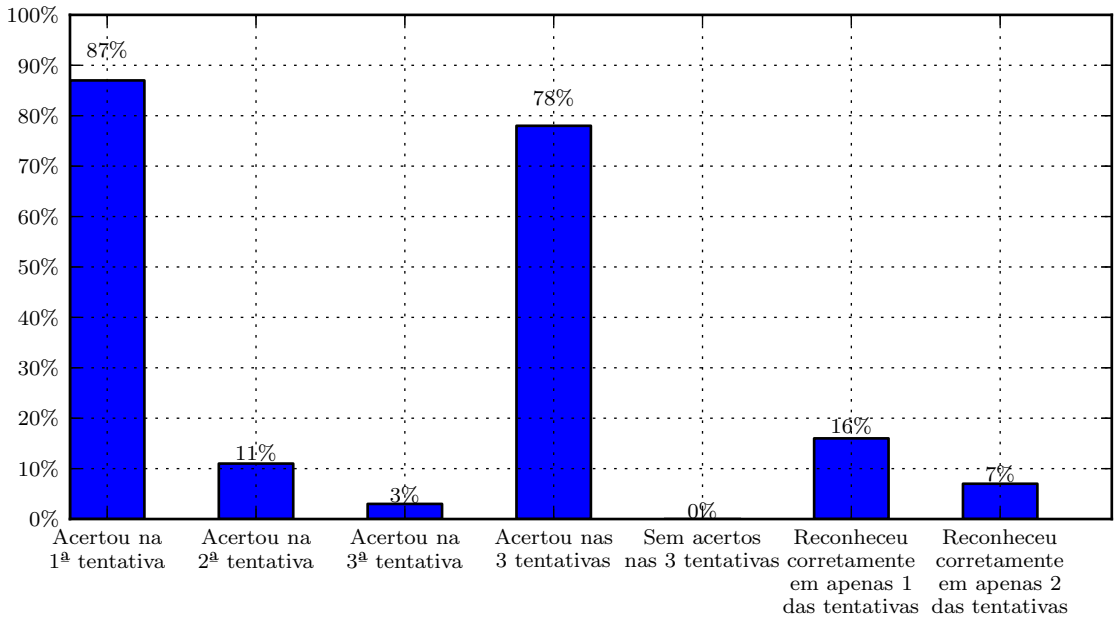

Figura 6.4: Resultado do teste falando 3 vezes cada cidade em um *headset* com a voz feminina 2.

6.4.2 Testes com telefone fixo VoIP

A segunda abordagem para o teste foi a utilização de um telefone fixo VoIP com a voz feminina 1. Para esse teste foram aleatoriamente selecionadas 10 cidades e cada uma delas foi repetida 3 vezes pela locutora. O resultado é apresentado na Figura 6.5. Pode-se observar que neste caso, ao contrário do que aconteceu com os teste com *softphone*, algumas cidades não foram corretamente reconhecidas, mesmo depois de serem repetidas 3 vezes. Porém o resultado geral foi superior (70%).

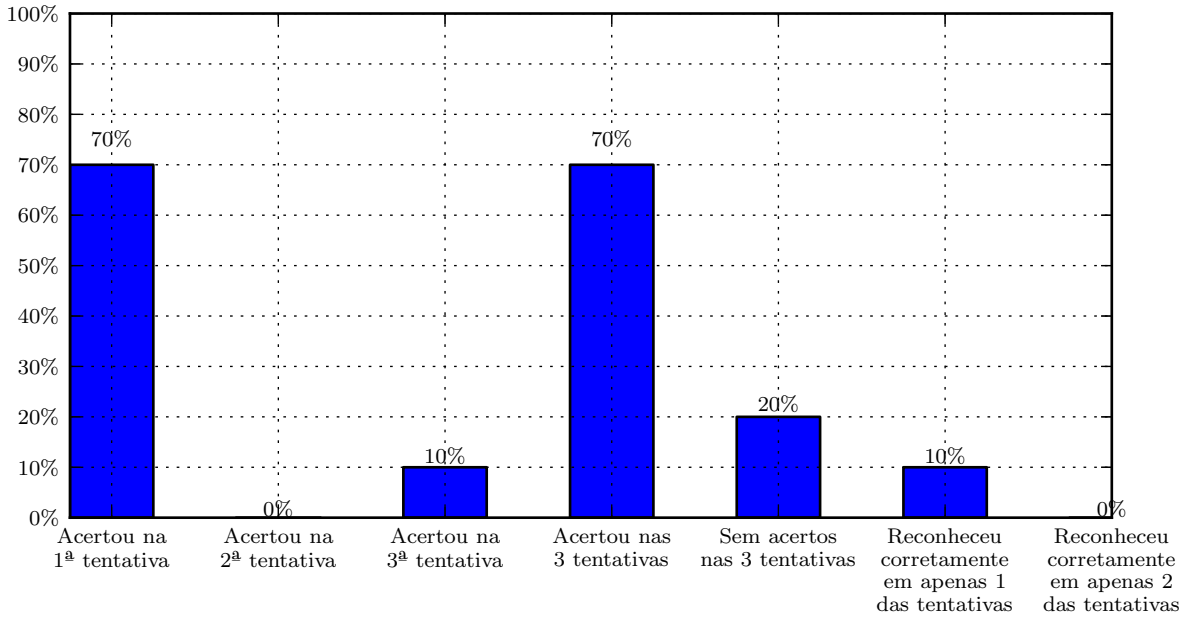

Figura 6.5: Resultado do teste falando 3 vezes 10 cidades aleatoriamente selecionadas, gerando a ligação a partir de um telefone fixo VoIP com a voz feminina 1.

6.4.3 Testes com aparelho celular na rede GSM

A última abordagem para teste foi a utilização de um aparelho celular GSM. Para este teste foram utilizadas 51 cidades sendo que a locutora falou cada uma delas apenas uma vez. O resultado mostrado na Figura 6.6 foi satisfatório, apresentado 98% de taxa de acerto. Um dos fatores que pode ter contribuído para este resultado é o fato da ligação realizada via celular ser menos ruidosa, devido o celular possuir um bom sistema para captura de áudio. Além disso, com o usuário falando muito próximo ao aparelho, a voz do mesmo acaba se sobressaindo mais facilmente ao ruído de fundo.

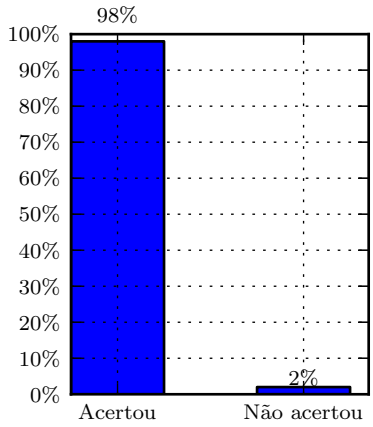

Figura 6.6: Resultado do teste falando 1 vez 51 cidades aleatoriamente selecionadas, gerando a ligação a partir de um celular com a voz feminina 1.

6.5 Conclusão

A solução apresentada atendeu de forma satisfatória as expectativas. E atualmente atende a central telefonica do disque 100, esta que recebe mais de 3 mil ligações diárias.

Capítulo 7

Conclusões

O presente trabalho contribuiu para o desenvolvimento de recursos que são específicos ao PB, com os quais se torna possível testar e disponibilizar um *framework* para ASR com grandes vocabulários em PB. De posse dos recursos e conhecimentos aqui disponibilizados, outros grupos de pesquisa podem criar novas ferramentas e aplicações e ainda contribuir para o aprimoramento das aqui propostas, visto que grande parte destes são *softwares* livres.

Por meio de listas e fóruns na internet nota-se que vários pesquisadores estão usando os recursos aqui desenvolvidos. Uma grande conquista foi a consolidação do grupo de usuários do Coruja [59]. Hoje, esta comunidade de usuários une mais de 300 membros e é essencial para o andamento e divulgação do projeto. Um exemplo de usuários externos utilizando o pacote Coruja é um projeto da Faculdade de Engenharia Mecatrônica da Universidade de Brasília (UnB), onde estão construindo um robô controlado por voz [60].

Deve-se registrar a importância teste trabalho, tanto no âmbito nacional quando internacional, pois é incontestável a necessidade de bons reconhecedores de voz para o PB, como justificado no decorrer deste trabalho. Quando se fala em outros idiomas também há contribuição, visto que, a LaPSAPI, JLaPSAPI e o SpeechOO podem ser usados em qualquer língua, desde que se tenha a mão os modelos e recursos necessários.

O SpeechOO, aqui desenvolvido, é um projeto inovador, que tem como principal funcionalidade o controle de um editor de texto com reconhecimento de voz. Por este motivo o SpeechOO é muito visado e possui vários seguidores interessados em usá-lo tanto quanto ajudar no desenvolvimento, considerando que o código deste produto é aberto. Além disso, o SpeechOO reúne anos de pesquisa em voz do grupo FalaBrasil para o PB em um aplicativo de extensão do Writer. Sendo assim, suas contribuições agora surgem realmente apoiando a comunidade brasileira em promover a acessibilidade de pessoas deficientes além de corresponder aos incentivos dado pelo governo em criar tecnologias dessa natureza no âmbito nacional.

Outra grande conquista deste trabalho foi a consolidação da parceria com a empresa Comunix. Além da arrecadação de recursos necessários para a manutenção do laboratório, esta foi uma excelente oportunidade para aplicar a teoria estudada em um produto comercial.

O desenvolvimento da solução para o disque 100, baseada em recursos criados e disponibilizados neste trabalho e, que foi bem avaliada em um teste realizado fora do ambiente do laboratório, é uma prova de que os recursos aqui desenvolvidos possuem qualidade suficiente para servir tanto a pesquisa, quando utilizados em ambiente acadêmico, quanto ao mercado, quando utilizados na indústria.

É importante ressaltar que alguns dos recursos citados encontram-se disponíveis na página do Grupo FalaBrasil [51]. Espera-se que outros grupos possam utilizar os recursos e compartilhar seus resultados.

Referências Bibliográficas

[1] "Censo Demográfico ," Visitado em Janeiro 2010, www.ibge.gov.br/home/estatistica/populacao/censo2010/. [Online]. Available: www.ibge.gov.br/home/estatistica/populacao/censo2010/

[2] P. Taylor, *Text-To-Speech Synthesis.* Cambridge University Press, 2009.

[3] X. Huang, A. Acero, and H. Hon, *Spoken Language Processing.* Prentice-Hall, 2001.

[4] "eSpeak text to speech," Visitado em Julho , 2010, http://espeak.sourceforge.net/. [Online]. Available: http://espeak.sourceforge.net/

[5] "Embedded ViaVoice," Visitado em Setembro , 2009, http://www.ibm.com/software/speech/. [Online]. Available: http://www.ibm.com/software/speech/

[6] "Nuance," Visitado em Março , 2011, http://www.nuance.com. [Online]. Available: http://www.nuance.com

[7] "Dragon NaturallySpeaking," Visitado em Julho , 2010, http://www.nuance.com/naturallyspeaking. [Online]. Available: http://www.nuance.com/naturallyspeaking

[8] "Windows Speech Recognition," Visitado em Julho , 2010, http://www.microsoft.com/speech. [Online]. Available: http://www.microsoft.com/speech

[9] M. Eskenazi, "An overview of spoken language technology for education," *Speech Communication, 51*, no. 10, pp. 832–844, 2009.

[10] J. Sealea and M. Cooperb, "E-learning and accessibility: An exploration of the potential role of generic pedagogical tools," *Computers & Education, 54*, no. 4, pp. 1107–1116, 2010.

[11] O. Saz, S.-C. Yin, E. Lleida, R. Rose, C. Vaquero, and W. Rodríguez, "Tools and technologies for computer-aided speech and language therapy," *Speech Communication, 51*, no. 10, pp. 948–967, 2009.

[12] T.-H. Wang, "Web-based dynamic assessment: Taking assessment as teaching and learning strategy for improving students' e-learning effectiveness," *Computers & Education, 54*, no. 4, pp. 1157–1166, 2010.

[13] A. Fidalgo-Neto, A. Tornaghi, R. Meirelles, F. Berçot, L. Xavier, M. Castro, and L. Alves, "The use of computers in Brazilian primary and secondary schools," *Computers & Education, 53*, no. 3, pp. 677–685, 2009.

[14] A. Siravenha, N. Neto, V. Macedo, and A. Klautau, "A computer-assisted learning software using speech synthesis and recognition in brazilian portuguese," *Interactive Computer Aided Blended Learning*, 2009.

[15] E. O'Harea and M. McTearb, "Speech recognition in the secondary school classroom: an exploratory study," *Computers & Education, 33*, no. 1, pp. 27–45, 1999.

[16] C. Hosn, L. Baptista, T. Imbiriba, and A. Klautau, "New resources for Brazilian Portuguese: Results for grapheme-to-phoneme and phone classification," *In VI International Telecommunications Symposium*, 2006.

[17] N. Neto, P. Silva, A. Klautau, and A. Adami, "Spoltech and OGI-22 baseline systems for speech recognition in Brazilian Portuguese," *International Conference on Computational Processing of Portuguese Language - PROPOR*, 2008.

[18] "CSLU: Spoltech Brazilian Portuguese Version 1.0," Visitado em Dezembro , 2011, http://www.ldc.upenn.edu/Catalog/catalogEntry.jsp?catalogId=LDC2006S16. [Online]. Available: http://www.ldc.upenn.edu/Catalog/catalogEntry.jsp?catalogId=LDC2006S16

[19] T. Lander, R. A. Cole, B. T. Oshika, and M. Noel, "The OGI 22 language telephone speech corpus," Center for Spoken Language Understanding, Oregon Graduate Institute, Tech. Rep.

[20] "Corpus de textos eletrônicos NILC/Folha de S. Paulo," Visitado em Março, 2013, Disponível em: acdc.linguateca.pt/cetenfolha/.

[21] P. Silva, N. Neto, A. Klautau, A. Adami, and I. Trancoso, "Speech recognition for Brazilian Portuguese using the Spoltech and OGI-22 corpora," *XXVI Simpósio Brasileiro de Telecomunicações*, 2008.

[22] A. Siravenha, N. Neto, V. Macedo, and A. Klautau, "Uso de regras fonológicas com determinação de vogal tônica para conversão grafema-fone em Português Brasileiro," *7th International Information and Telecommunication Technologies Symposium*, 2008.

[23] D. Silva, A. de Lima, R. Maia, D. Braga, J. F. de Moraes, J. A. de Moraes, and F. R. Jr, "A rule-based grapheme-phone converter and stress determination for Brazilian Portuguese natural language processing," *VI International Telecommunications Symposium*, 2006.

[24] "West point brazilian portuguese speech," 2008, http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2008S04. [Online]. Available: http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2008S04

[25] P. Silva, N. Neto, and A. Klautau, "Novos recursos e utilização de adaptação de locutor no desenvolvimento de um sistema de reconhecimento de voz para o Português Brasileiro," *In XXVII Simpósio Brasileiro de Telecomunicações*, 2009.

[26] R. J. Cirigliano, C. Monteiro, F. L. de F. Barbosa, F. G. V. R. Jr, L. Couto, and J. Moraes, "Um conjunto de 1000 frases foneticamente balanceadas para o Português Brasileiro obtido utilizando a abordagem de algoritmos genéticos," *XXII Simpósio Brasileiro de Telecomunicações*, 2005.

[27] S. Young, D. Ollason, V. Valtchev, and P. Woodland, *The HTK Book*. Cambridge University Engineering Department, version 3.4, 2006.

[28] N. Neto, C. Siravenha, V. Macedo, and A. Klautau, "A computer-assisted learning software to help teaching english to brazilians," *International Conference on Computational Processing of the Portuguese Language - Special Session*, 2008.

[29] J. Morais, N. Neto, and A. Klautau, "Tecnologias para o desenvolvimento de sistemas de diálogo falado em Português Brasileiro," *7th Brazilian Symposium in Information and Human Language Technology*, 2009.

[30] P. Silva, P. Batista, N. Neto, and A. Klautau, "An open-source speech recognizer for Brazilian Portuguese with a windows programming interface," *Computational Processing of the Portuguese Language*, vol. 6001, pp. 128–131, 2010.

[31] R. Oliveira, P. Batista, N. Neto, and A. Klautau, "Recursos para desenvolvimento de aplicativos com suporte a reconhecimento de voz para desktop e sistemas embarcados," *XII Workshop de Software Livre (WSL 2011)*, 2011.

[32] ——, "Recursos para desenvolvimento de aplicativos com suporte a reconhecimento de voz para desktop e sistemas embarcados," *12th International Conference on Computational Processing of the Portuguese Language (PROPOR)*, 2012.

[33] X. Huang, Y. Ariki, and M. Jack, *Hidden Markov Models for Speech Recognition*, ser. Edinburgh Information Technology Series, 7. Edinburgh University Press, 1990.

[34] S. Davis and P. Merlmestein, "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences," *IEEE Trans. on ASSP*, vol. 28, pp. 357–366, Aug. 1980.

[35] L. Rabiner and B. Juang, *Fundamentals of Speech Recognition.* Englewood Cliffs, N.J.: PTR Prentice Hall, 1993.

[36] N. Deshmukh, A. Ganapathiraju, and J. Picone, "Hierarchical search for large-vocabulary conversational speech recognition," *IEEE Signal Processing Magazine*, pp. 84–107, 1999.

[37] N. Jevtić, A. Klautau, and A. Orlitsky, "Estimated rank pruning and Java-based speech recognition," in *Automatic Speech Recognition and Understanding Workshop*, 2001.

[38] P. Ladefoged, *A Course in Phonetics*, 4th ed. Harcourt Brace, 2001.

[39] G. Antoniol, R. Fiutem, R. Flor, and G. Lazzari, "Radiological reporting based on voice recognition," *Human-computer interaction. Lecture Notes in Computer Science*, vol. 753, pp. 242–253, 1993.

[40] C. H. Lee and J. L. Gauvain, "Speaker adaptation based on MAP estimation of HMM parameters," *IEEE ICASSP*, pp. 558–561, 1993.

[41] R. Bellman, *Dynamic Programming.* Princeton University Press, 1957.

[42] "Microsoft language development center Portugal," http://www.microsoft.com/portugal/mldc/default.mspx, Visitado em Junho , 2010, http://www.microsoft.com/portugal/mldc/default.mspx. [Online]. Available: http://www.microsoft.com/portugal/mldc/default.mspx

[43] A. Stolcke, "SRILM an extensible language modeling toolkit," *7th International Conference on Spoken Language Processing*, 2002.

[44] A. Lee, *The Julius Book.* 0.0.2 ed. - rev 4.1.2, 2009.

[45] W. Walker, P. Lamere, P. Kwok, B. Raj, R. Singh, E. Gouvea, P. Wolf, and J. Woelfel, "Sphinx-4: A flexible open source framework for speech recognition," Sun Microsystems Inc, Tech. Rep., 2004.

[46] D. Huggins-Daines, M. Kumar, A. Chan, A. W. Black, M. Ravishankar, and A. I. Rudnicky, "Pocketsphinx: a free, real-time continuous speech recogntion system for hand-held devices," *Proceedings of ICASSP*, pp. 185–188, May 2006.

[47] "Microsoft speech API," www.microsoft.com/speech/, Visitado em Novembro , 2010, www.microsoft.com/speech/. [Online]. Available: www.microsoft.com/speech/

[48] "Java speech API," java.sun.com/products/java-media/speech/, Visitado em Novembro , 2010, java.sun.com/products/java-media/speech/. [Online]. Available: java.sun.com/products/java-media/speech/

[49] "Legislação em audio," Visitado em Julho , 2010, http://www2.camara.gov.br/acessibilidade/constituicaoaudio.html. [Online]. Available: http://www2.camara.gov.br/acessibilidade/constituicaoaudio.html

[50] "Microsoft speech API 5.3," msdn.microsoft.com/en-us/ms723632(VS.85).aspx, Visitado em Junho , 2010, msdn.microsoft.com/en-us/ms723632(VS.85).aspx. [Online]. Available: msdn.microsoft.com/en-us/ms723632(VS.85).aspx

[51] "FalaBrasil: Reconhecimento de Voz para o Português Brasileiro," Visitado em Julho , 2010, http://www.laps.ufpa.br/falabrasil/. [Online]. Available: http://www.laps.ufpa.br/falabrasil/

[52] T. Rotovnik, M. S. Maucec, B. Horvat, and Z. Kacic, "A comparison of HTK, ISIP and Julius in Slovenian large vocabulary continuous speech recognition," *7th International Conference on Spoken Language Processing*, 2002.

[53] "Java speech grammar format specification," Sun microsystems, Tech. Rep., 1998.

[54] "Openoffice.org community announces the document foundation," Visitado em Dezembro , 2011, http://web.archive.org/web/20100930085933/http://www.documentfoundation.org/contact/tdf_release.html. [Online]. Available: http://web.archive.org/web/20100930085933/http://www.documentfoundation.org/contact/tdf_release.html

[55] S. Liang, *The Java^{TM}Native Interface Programmer's Guide and Specification.* Addison-Wesley, 1999.

[56] R. M. e. A. K. Denise Alves, "Módulo de adaptação de locutor utilizando regressão linear de máxima verossimilhança para sistemas de reconhecimento de voz," *XII Workshop de Software Livre (WSL)*, 2010.

[57] "Resultado edital CNPq nº 09/2010 - PDI - grande e pequeno porte," Visitado em Dezembro , 2011, http://www.cnpq.br/resultados/2010/009.htm. [Online]. Available: http://www.cnpq.br/resultados/2010/009.htm

[58] DigiVoice, "Guia do programador sistema de desenvolvimento para placas digivoice," Digivoice Eletrônica, Tech. Rep.

[59] "Lista de discussão do software Coruja," Visitado em Julho , 2010, http://groups.google.com/group/coruja-users?hl=pt-BR. [Online]. Available: http://groups.google.com/group/coruja-users?hl=pt-BR

[60] "Interface homem-robô (PIBIC 2009/2010)," groups.google.com.br/group/human-robot-interaction/, Visitado em Junho , 2010, groups.google.com.br/group/human-robot-interaction/. [Online]. Available: groups.google.com.br/group/human-robot-interaction/