



---

# Spoken Dialogue Systems: Architectures and Applications

---

Doctoral Thesis

**Javier Mikel Olaso Fernandez**

Advisor

**María Inés Torres Barañano**

Leioa 2017

# CONTENTS

---

<b>List of Figures</b> . . . . .	<b>v</b>
<b>List of Tables</b> . . . . .	<b>vii</b>
<b>List of Symbols</b> . . . . .	<b>ix</b>
<b>1 Introduction</b> . . . . .	<b>1</b>
<b>2 Spoken Dialogue Systems</b> . . . . .	<b>5</b>
1 Overview . . . . .	5
2 Components of a Spoken Dialogue System . . . . .	6
2.1 Automatic Speech Recognition . . . . .	7
2.2 Natural Language Understanding . . . . .	8
2.3 Natural Language Generation . . . . .	9
2.4 Text-To-Speech Synthesis . . . . .	9
2.5 Dialogue Management . . . . .	10
2.5.1 State-Based . . . . .	10
2.5.1.1 Finite State-Based . . . . .	11
2.5.1.2 Frame-Based . . . . .	12
2.5.2 Agent-Based . . . . .	13
2.5.3 Information State-Based . . . . .	14
2.5.4 Statistical . . . . .	14
2.5.4.1 MDP and POMDP . . . . .	15
2.5.4.2 Stochastic Finite State Bi-Automata . . . . .	18
2.5.4.3 Neural Networks . . . . .	22
2.6 Other Components . . . . .	22
3 Examples of Spoken Dialogue Systems . . . . .	23
<b>3 Frameworks and Toolkits</b> . . . . .	<b>25</b>
1 PC under Windows OS . . . . .	25
2 PC under Unix-like OS . . . . .	28
3 PC under Windows or Unix-like OS . . . . .	28
4 Other Platforms . . . . .	31
<b>4 Preliminary Systems</b> . . . . .	<b>35</b>
1 Introduction . . . . .	35
2 INMA . . . . .	35
2.1 Architecture . . . . .	36
2.2 Configuring and Launching the System . . . . .	38
2.3 Modules Implementation Details . . . . .	40
2.3.1 Automatic Speech Recognition and Text to Speech . . . . .	40
2.3.2 User Interface . . . . .	41
2.3.3 Data Backend . . . . .	42

	2.3.4	Comprehension . . . . .	44
3	RTMIVA . . . . .		45
	3.1	Architecture Definition . . . . .	46
	3.2	Installing and Configuring the Architecture . . . . .	46
	3.2.1	Installing Asterisk PBX . . . . .	47
	3.2.2	Installing no-ip . . . . .	49
	3.2.3	Installing PHPAGI . . . . .	51
	3.2.4	Installing TTS . . . . .	51
	3.2.5	Installing ASR . . . . .	52
	3.2.6	Configuring Asterisk PBX . . . . .	53
	3.2.6.1	Creating Users . . . . .	53
	3.2.6.2	Configuring Extensions . . . . .	59
	3.2.7	Applications Interface . . . . .	60
	3.2.8	Installing and Configuring the Clients . . . . .	62
	3.2.8.1	Clients Using SIP and IAX Protocols . . . . .	62
	3.2.8.2	Clients Using H.323 Protocol . . . . .	63
	3.3	Weather Forecast Retrieval Application . . . . .	63
	3.3.1	Application File Definition . . . . .	64
	3.3.2	Integrating the Application into Asterisk . . . . .	69
4	Concluding Remarks . . . . .		69
<b>5</b>	<b>Bus Information System Based on EDECAN Framework . . . . .</b>		<b>71</b>
1	Introduction . . . . .		71
2	RavenClaw Dialogue Manager . . . . .		72
3	Data Analysis . . . . .		74
4	Bus Information System Based on Olympus . . . . .		75
	4.1	Architecture . . . . .	76
	4.1.1	Automatic Speech Recongition . . . . .	76
	4.1.2	Natural Language Understanding . . . . .	77
	4.1.2.1	Parser Grammar Definition . . . . .	78
	4.1.3	Dialogue Manager . . . . .	80
	4.1.4	Data Backend . . . . .	81
	4.1.5	Natural Language Generation . . . . .	83
	4.1.5.1	Templates Definition . . . . .	83
	4.1.6	Speech Synthesis . . . . .	84
	4.1.7	Telephone Interface . . . . .	85
5	Bus Information System Based on EDECAN . . . . .		85
	5.1	Architecture . . . . .	85
	5.2	Configuring and Launching the System . . . . .	88
	5.2.1	Audio interface configuration . . . . .	88
	5.2.2	Configuring and Launching EDECAN Services . . . . .	89
	5.2.3	Configuring and Launching Olympus Services . . . . .	90
	5.3	Modules Implementation Details . . . . .	91
	5.3.1	Automatic Speech Recognition . . . . .	91
	5.3.2	Semantic Parser . . . . .	93
	5.3.3	Natural Language Generator . . . . .	93
	5.3.4	Text to Speech . . . . .	94

5.3.5	Phone Interface . . . . .	94
5.3.6	Dialogue Manager . . . . .	94
5.3.7	Dialogue Manager Interface and Data Backend . . . . .	94
5.3.8	Controller . . . . .	97
5.3.9	DateTime . . . . .	98
5.4	Experimental Evaluation . . . . .	98
5.4.1	System Performance . . . . .	99
5.4.2	Usability . . . . .	100
5.4.3	Speech Assessment . . . . .	101
5.4.4	Attractiveness . . . . .	103
6	Concluding Remarks . . . . .	106
<b>6</b>	<b>Systems Based on ActiveMQ . . . . .</b>	<b>107</b>
1	Introduction . . . . .	107
2	vAssist . . . . .	107
2.1	Architecture . . . . .	108
2.1.1	Speech Recognition . . . . .	108
2.1.2	Semantic Parsing . . . . .	109
2.1.3	Natural Language Generation . . . . .	109
2.1.4	Text to Speech . . . . .	109
2.1.5	Semantic Unification and Resolution . . . . .	110
2.1.6	Dialogue Act Mapping . . . . .	110
2.1.7	Dialogue Management . . . . .	110
2.2	Adding Languages and RavenClaw DM Integration . . . . .	111
2.2.1	Adding Languages . . . . .	111
2.2.2	RavenClaw DM Integration . . . . .	112
2.3	Experimental Evaluation . . . . .	112
2.3.1	Task and Experimental Scenarios . . . . .	112
2.3.2	Experimental Evaluation . . . . .	113
2.3.2.1	System Performance . . . . .	114
2.3.2.2	Disco-LFF and RavenClaw DM Comparison . . . . .	114
3	Personal Assistant Robot Dialogue System . . . . .	115
3.1	Architecture . . . . .	116
3.2	Configuring and Launching the System . . . . .	117
3.2.1	Configuring and Launching ActiveMQ Services . . . . .	117
3.2.2	Configuring and Launching Olympus Services . . . . .	118
3.3	Modules Implementation Details . . . . .	119
3.3.1	ActiveMQ Services Programming . . . . .	119
3.3.2	Automatic Speech Recognition . . . . .	120
3.3.3	Semantic Parser . . . . .	121
3.3.4	Natural Language Generator . . . . .	121
3.3.5	Output . . . . .	121
3.3.6	Face recognition . . . . .	121
3.3.7	Dialogue Manager Interface . . . . .	122
3.3.8	Controller . . . . .	122
3.3.9	RavenClaw Dialogue Manager . . . . .	123

3.4	Demonstrator . . . . .	125
<b>7</b>	<b>Conclusions . . . . .</b>	<b>127</b>
<b>A</b>	<b>EDECAN Architecture Specification . . . . .</b>	<b>129</b>
1	Introduction . . . . .	129
2	Communication Protocol and Routing Modes . . . . .	130
2.1	Communication Protocol . . . . .	130
2.2	Routing Modes . . . . .	131
3	System Configuration . . . . .	132
3.1	Communication Manager . . . . .	132
3.2	Server . . . . .	133
3.3	Service . . . . .	134
4	Controller Commands . . . . .	135
<b>B</b>	<b>Systems Based on EDECAN: Configuration Files . . . . .</b>	<b>139</b>
1	Campus SDS System . . . . .	139
2	INMA . . . . .	145
<b>C</b>	<b>RTMIVA Applications Configuration Files . . . . .</b>	<b>151</b>
0.1	Applications Interface Code . . . . .	151
0.2	TTS: text2audio source code . . . . .	155
0.3	Weather Information Retrieval XML File . . . . .	156
<b>D</b>	<b>Usability Questionnaires . . . . .</b>	<b>159</b>
1	System Usability Scale . . . . .	159
2	Subjective Assessment Speech System Interfaces . . . . .	161
3	AttrakDiff 2 . . . . .	163
	<b>Bibliography . . . . .</b>	<b>165</b>

# LIST OF FIGURES

---

2.1	SDS basic architecture. . . . .	6
2.2	ASR basic architecture. . . . .	7
2.3	DM components example. . . . .	11
2.4	Finite State Based dialogue manager example. . . . .	11
2.5	Finite State Based dialogue manager example. . . . .	12
2.6	Frame Based dialogue manager example. . . . .	13
2.7	Block diagram of a Spoken Dialogue System proposed in [1]. . . . .	15
4.1	INMA: Final system. . . . .	36
4.2	Architecture of the INMA SDS System. . . . .	37
4.3	INMA: User Interface. . . . .	42
4.4	RTMIVA system architecture. . . . .	47
4.5	Structure of the dialogues for the meteorological information retrieval system with RTMIVA architecture. . . . .	64
4.6	Language Model for the Weather Forecast retrieval application in RTMIVA. . . . .	65
5.1	RavenClaw dialogue tree example. . . . .	72
5.2	Let's Go Spanish Port: System architecture, based on architecture proposed in [2]. . . . .	77
5.3	Basic task specification for the Campus SDS system. . . . .	81
5.4	Task Tree for the Bus Information System implemented with Ravenclaw. . . . .	82
5.5	Architecture of the Campus SDS System. . . . .	86
5.6	SASSI. Mean values and standard errors. . . . .	103
5.7	AttrakDiff 2. Mean values and standard errors. . . . .	106
6.1	vAssist: System architecture. . . . .	109
6.2	Roberta: System architecture. . . . .	116
6.3	Roberta: Architecture implementation with ActiveMQ. . . . .	117
6.4	Roberta SDS System: Dialogue structure. . . . .	125
6.6	Roberta: Screenshot of the Roberta SDS demonstrator. . . . .	125
6.5	Roberta SDS System: Dialogue structure for when users ask questions. . . . .	126
A.1	EDECAN architecture block diagram. . . . .	129



# LIST OF TABLES

---

3.1	Summary of SDS frameworks and toolkits. . . . .	33
4.1	Set of commands (and arguments) for the different modules in INMA SDS. . . . .	37
4.2	RT-MIVA dialplan. . . . .	60
5.1	User goals detected in the 010 Corpus. Sub-functions observed in the Let's Go grammar are marked with (*). . . . .	75
5.2	Set of commands (and arguments) for the different modules in the Campus SDS System. . . . .	87
5.3	Description of the main characteristics and profiles of the set of users taking part in the experiments . . . . .	99
5.4	Task Completion and Average Dialogue Length of the user-system interaction metrics measured during the User Experience evaluation for the Campus SDS System. . . . .	99
5.5	Results for a subset of the SUS questionnaire. . . . .	101
5.6	SASSI questionnaire results: Habitability subset. . . . .	102
5.7	SASSI questionnaire results: Speed subset. . . . .	102
5.8	Attrakdiff 2 (Pragmatic Quality) results. . . . .	104
5.9	Attrakdiff 2 (Hedonic Quality-Identity) results. . . . .	105
6.1	TC and ADL of the vAssist system using the Disco-LFF DM. . . .	114
6.2	Comparing the Disco-LFF and RavenClaw DMs. . . . .	115
D.1	SUS questionnaire. . . . .	160
D.2	SASSI questionnaire. . . . .	162
D.3	ATTRAKDIFF 2 questionnaire. . . . .	164





# LIST OF SYMBOLS

---

<b>ADL</b> Average Dialogue Length . . . . .	98, 99, 114, 115
<b>AGI</b> Asterisk Gateway Interface . . . . .	51
<b>ASR</b> Automatic Speech Recognition . . . . .	5–8, 14, 19, 20, 25, 26, 29, 30, 36, 38, 40, 44, 46, 51, 52, 67, 68, 85, 87, 88, 94, 97, 98, 100, 108, 109, 117, 120–122, 128
<b>CMU</b> Carnegie Mellon University . . . . .	25
<b>DA</b> Dialogue Act . . . . .	110
<b>DHCP</b> Dynamic Host Configuration Protocol . . . . .	50
<b>DM</b> Dialogue Manager . . . . .	2, 6, 7, 10–15, 17, 19–21, 25–27, 29–31, 36, 44, 46, 60, 71, 72, 74, 76, 83, 85, 86, 88, 94, 95, 97, 98, 106–108, 110, 116, 117, 121–124, 128
<b>DME</b> Dialogue Move Engine . . . . .	30, 31
<b>DNS</b> Domain Name Server . . . . .	50
<b>DSP</b> Digital Signal Processing . . . . .	10
<b>FXO</b> Foreign eXchange Office . . . . .	47
<b>FXS</b> Foreign eXchange Subscriber . . . . .	47
<b>GPL</b> GNU General Public License . . . . .	26, 30
<b>HMM</b> Hidden Markov Models . . . . .	8, 10
<b>IAX</b> InterAsterisk eXchange protocol . . . . .	47, 56
<b>IETF</b> Internet Engineering Task Force . . . . .	53
<b>IP</b> Internet Protocol . . . . .	47, 49, 50
<b>ISDN</b> Integrated Services Digital Network . . . . .	47
<b>LFF</b> Linked Form Filling . . . . .	108, 110, 111
<b>LM</b> Language Model . . . . .	52, 53, 61, 62, 64, 67, 68
<b>LSTM</b> Long Short Term Memory . . . . .	22
<b>MDP</b> Markov Decision Process . . . . .	14–18

<b>NAT</b>	Network Address Translation . . . . .	55, 56
<b>NLG</b>	Natural Language Generation . . . . .	6, 7, 9, 25, 29, 30, 36, 83, 85, 93, 108, 111, 121
<b>NLP</b>	Natural Language Processing . . . . .	9, 10
<b>NLU</b>	Natural Language Understanding . . . . .	5, 6, 8, 9, 14, 25, 26, 29, 85, 108
<b>OAA</b>	Open Agent Architecture . . . . .	30
<b>PBX</b>	Private Branch Exchange . . . . .	46, 47, 49–51, 54, 55, 59, 60
<b>POMDP</b>	Partially Observable Markov Decision Process . . . . .	14, 15, 18, 22
<b>RAD</b>	Rapid Application Developer . . . . .	27
<b>RTMIVA</b>	Real-Time Multimedia Interactive Voice Architecture . . . . .	45, 63, 66
<b>RTN</b>	Recursive Transition Network . . . . .	78, 80
<b>RTP</b>	Real-time Transport Protocol . . . . .	53, 55, 56
<b>SASSI</b>	Subjective Assessment of Speech System Interfaces . . . . .	99, 101
<b>SDK</b>	Software Development Kit . . . . .	32
<b>SDP</b>	Session Description Protocol . . . . .	53
<b>SDS</b>	Spoken Dialogue System 1–3, 5, 6, 10, 14, 15, 18, 19, 22, 23, 25, 27–29, 35, 36, 45, 46, 69, 72, 76, 85, 106–108, 127, 128	
<b>SF</b>	Semantic Frame . . . . .	109, 110
<b>SIP</b>	Session Initial Protocol . . . . .	47, 53–55, 94
<b>SP</b>	Semantic Parser . . . . .	36, 44, 93, 100, 109, 111, 112, 117, 121, 122
<b>SUS</b>	System Usability Scale . . . . .	99, 100
<b>TC</b>	Task Completion . . . . .	98–100, 114, 115
<b>TTS</b>	Text To Speech 6, 7, 9, 25, 26, 29, 30, 36, 38, 41, 44, 46, 51, 52, 85, 88, 94, 97, 108, 121	
<b>UA</b>	User Agent . . . . .	54
<b>UAC</b>	User Agent Client . . . . .	54
<b>UAS</b>	User Agent Server . . . . .	54
<b>UDP</b>	User Datagram Protocol . . . . .	55, 56
<b>VAD</b>	Voice Activity Detector . . . . .	92

---

<b>VoIP</b> Voice over IP . . . . .	46, 47, 53, 56, 62
<b>WOZ</b> Wizard of Oz . . . . .	76, 123



# 1

## INTRODUCTION

---

In a world in which technology and technological devices have become habitual and omnipresent, humans need to learn to communicate with all kind of devices. Until recently, and in the face of this need for communication with the different devices, humans needed to learn how the devices express themselves to communicate with them, which often, may not be intuitive enough. But in recent times the tendency has become to make communication with these devices in more intuitive ways. An obvious example of this trend is the great success obtained with the use of touch screens available in all kinds of devices.

Taking intuitiveness into consideration, the ideal way to communicate with devices would be the natural way of communication between humans, the speech. In this context, humans have long been investigating and designing systems that use this type of communication, giving rise to the so-called Spoken Dialogue Systems (SDSs).

In the beginning, SDSs were relatively simple systems designed to carry out simple tasks such as the provision of air travel or bus schedule information, or even simpler systems based on speech commands recognition for executing some associated actions. Nowadays, and although it's still a technology that has much to develop, there are more complex systems used for performing more complex tasks. Examples of these current systems, which prove the great future that the SDSs have, are systems for Intelligent Environments, communication systems with intelligent robots or the increasingly popular virtual assistants such as Apple's Siri or Microsoft's Cortana.

In the described context, based on different contacts carried out with the enterprise world, the research group to which the author of this thesis is assigned detected a need for such dialogue systems, which awoke in the group and the author of this thesis the interest for this type of systems.

Initially, the research team undertook an academic course focused on building systems based on statistical methods for dialogue management, mainly POMDPs and neural networks. However, it was seen that this approach, nowadays, is not able to solve the implementation needs required by companies. There is a significant difference between the technologies employed by enterprises and those used in the academic environment, being the first ones, in general, simpler and focused on functionality and reliability. This fact en-

tailed a course change for the research group focused on engineering and in building functional and reliable systems. With this in mind, the primary goal of this thesis is to show how these systems were implemented, and by extension how SDSs can be built. Also, to show how the difficulties encountered can be solved. This thesis additionally serves as a review of the state-of-the-art regarding architectures and toolkits focused on the development of SDSs. Finally, this document is intended to serve future system developers as a guide for their construction. Therefore, details are given both for the installation and for the configuration of the different required components.

In this context, on the one hand, an analysis about what SDSs consist of and the elements that compose them has been carried out. On the other hand, we have built several of these systems to learn how to do it and to analyse the existing problems.

For first testing works, we have explored the implementation of systems under the EDECAN framework, a framework designed for the development of distributed applications and focused on the development of SDSs, through an application to get meteorological forecasts and news from digital newspapers. We have also developed a new architecture for the implementation of SDSs focused on the interaction through mobile devices on which we have built a system aimed to get meteorological forecasts.

The dialogue management task of these systems is distributed between several components. With our experience, we have seen the need for its centralisation in a unique part. This need for a centralisation of the dialogue management task led us to the idea of integrating a Dialogue Manager (DM) into the systems and investigate the Olympus framework and its RavenClaw DM. During this investigation, an adaptation of the Let's Go system to the Spanish language has been carried out. That has allowed us an in-depth knowledge of the RavenClaw DM. Then, we have decided to integrate this DM into the EDECAN framework and a new system has been developed, consisting of a telephone information system for obtaining bus schedules.

In addition, and with the knowledge acquired about the RavenClaw DM, two additional works have been carried out. One consists in the development of an SDS prototype for the creation of an assistant robot for the elderly and the other in the integration of the RavenClaw DM in a pre-existing system, with the purpose of comparing its performance with the system's own DM.

Several contributions have been made during the development of these systems:

- A revision of the frameworks and toolkits available to develop SDSs.
- Development of several components to build SDSs under EDECAN framework.
- Make possible to use the RavenClaw DM in other frameworks apart from its native one, Olympus.
- Development of a new framework to implement SDSs focused on inter-

action through mobile devices.

- Development of a set of SDSs aimed to deal with a variety of applications, e.g. weather forecast retrieval, bus schedules information retrieval, personal and elderly assistance, under a variety of frameworks, namely, EDECAN, Olympus, RTMIVA and ActiveMQ.
- Evaluation of the user's experience in handling with a bus information telephone system.

The rest of the thesis is structured as follows:

- Chapter 2 describes what SDSs are and the components of which they consist of, analysing the main components, their characteristics and ways of implementation.
- Chapter 3 gives a review of the various existing toolkits for the implementation of SDSs.
- Chapters from 4 to 6 are devoted to the description of the systems that have been implemented during the development of this thesis.
- Finally, in Chapter 7 some conclusions obtained during the elaboration of the thesis are given.





# 2

## SPOKEN DIALOGUE SYSTEMS

---

### 1 Overview

---

SDSs can be defined as informatics systems that allow humans to interact with such systems using natural language. In a general way, it can be said that SDSs are systems that get spoken utterances as input from the user and generate spoken utterances as output. The primary purpose of such systems is to facilitate natural interaction between humans and computerised systems when trying to perform some task.

SDSs are complex systems that require the interaction of several components to recognise the user's speech input utterance, extract the relevant information from the speech transcriptions, retrieve information from backend services, decide the next step in the dialogue, generate system responses and synthesise them, etc.

Ideally, an SDS system is desired to have the following capabilities:

- Being able to recognise any speech, that is, spontaneous speech, not being dependent on speaker or gender, etc.
- Being able to understand sentences in any domain without content restrictions.
- Giving well-formed answers, that is, meaningful, grammatically well-formed and pragmatically suitable.
- Having an entirely natural voice.

But given the limitations of the different technologies involved in the development of such systems, it's not possible to get these capabilities, among others, the following reasons:

- There are limitations in the Automatic Speech Recognition (ASR) technology. Nowadays, technology doesn't allow to recognise speech entirely, i.e., there are problems to recognise spontaneous speech, problems with phonetic pronunciation differences between speakers, problems with environmental interferences, etc.
- The Natural Language Understanding (NLU), the component responsible for analysing the output from the ASR, is limited to restricted domains, i.e., getting train schedule information, buying aeroplane tickets, etc.

- Natural Language Generation (NLG) technology isn't capable of emulating the mechanisms humans use to generate sentences in any kind of situation.
- Human dialogues are characterised for being natural and coherent and by the fact of not needing confirmation to every question. To emulate such dialogues, non-existent complex DMs are required.
- Text To Speech (TTS) technology still doesn't allow to create synthesised voices that can be confused with that of a person. Neither allows simulating the prosodic realisation of natural voice sentences as humans do.

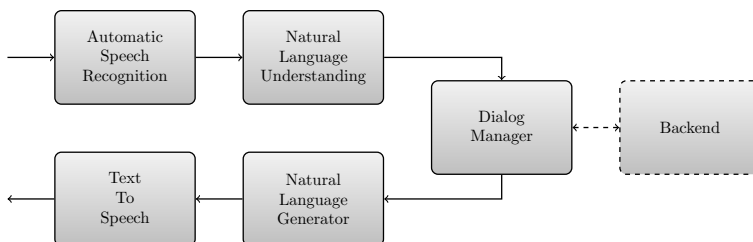
Nevertheless, despite the limitations, the development of such systems is proliferating.

## 2 Components of a Spoken Dialogue System

An SDS needs to deal with several tasks to emulate human's dialogues. For a simplified and primary SDS the following tasks are necessary:

- Recognition of users' utterances.
- Linguistic analysis of users' utterances.
- Dialogue management.
- Generation of sentences.
- Text to speech conversion.

Usually, SDSs are built in a modular way in which each of the modules is responsible for performing each of the mentioned tasks, interacting together to carry out the dialogue.



**Figure 2.1:** SDS basic architecture.

Figure 2.1 shows the architecture of such a primary system. It consists of an ASR module responsible for transforming the speech signal into a suitable representation, usually a text string. From that representation, the NLU extracts a semantic representation. Once the semantic representation has been obtained, it's sent to the DM to decide the next step in the dialogue. The DM, using the semantic representation of the speech signal and, in case of being necessary, information retrieved from a data backend server, decides the next

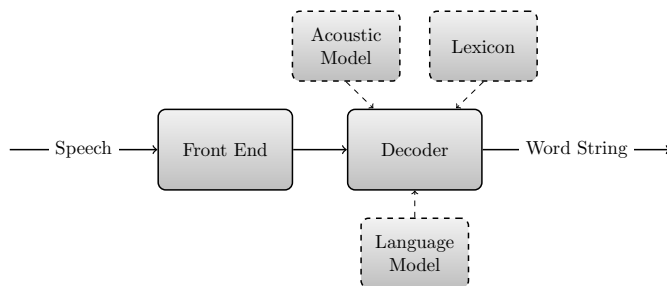
step in the dialogue, which mainly consists in giving some information to the user. For that, the DM sends the required information to the NLG module. The NLG module builds a sentence with the received information and sends it to the TTS module to be synthesised and given the information to the user.

Following sections will briefly describe each of the modules mentioned.

## 2.1 Automatic Speech Recognition

The ASR module is responsible for converting users' speech utterances into sequences of words.

Figure 2.2 shows the basic architecture of an ASR system. To convert speech into a sequence of words, it uses phonetic, lexical and language models.



**Figure 2.2:** ASR basic architecture.

Depending on several criteria ASR modules can be classified into several categories:

- *Style and type of speech:* Depending on the style and type of speech it can recognise ASRs can be classified as *isolated words*, *continuous speech* or *spontaneous speech* recognisers. Isolated word recognisers are the simplest ones and are characterised for recognising words only. On the other hand, continuous and spontaneous speech recognisers are distinguished for recognising full sentences with the difference that continuous ones recognise well-spoken sentences only, and spontaneous ones can recognise sentences including repetitions, interjections, pitch, etc.
- *Speaker dependence:* An ASR can be classified as *speaker dependent* or *speaker independent*. Speaker dependent recognisers are designed to work optimally with a small group of users and, on the other hand, speaker independent recognisers are designed with the goal of having similar performance whoever the user.
- *Channel type:* Speech recognisers can get their input from microphones or telephones, in which the audio signal has differences in quality.
- *The dimension of the vocabulary:* Depending on the number of different words a recogniser can deal with, it can be classified as *sort vocabulary* or *large vocabulary* speech recogniser. Usually, the number of words

limits the quality of a dialogue due to non-existent words in the lexicon, anyway, this issue can be minimised using specific language models for each topic in the application and loading them depending on the state of the dialogue.

- *Language*: Some recognisers are designed to recognise one language only, and others are designed to recognise multiple languages.

Some of the most known toolkits to develop ASR systems are:

- HTK [3], a toolkit for building and manipulating hidden Markov models. Although it's primarily used for speech recognition research, it has been used for other applications such as speech synthesis, character recognition or DNA sequencing. It provides a set of tools for speech analysis, Hidden Markov Models (HMM) training, testing and analysis. It supports HMMs using both continuous density mixture Gaussians and discrete distributions and can be used to build complex HMM systems.
- CMU Sphinx [4], is a set of tools that include several ASR decoders, an acoustic model trainer (SphinxTrain), a Language model compilation tool and a public domain pronunciation dictionary (cmudict). The different decoders are targeted for different situations such as real-time recognition, high-accuracy with non-real-time recognition or recognition within embedded systems.
- Kaldi [5], a toolkit that provides a speech recognition system based on finite-state transducers. It also provides a set of scripts for building complete recognition systems and supports modelling of arbitrary phonetic-context sizes, acoustic modelling with subspace Gaussian mixture models (SGMM) as well as standard Gaussian mixture models, together with all commonly used linear and affine transforms.

## 2.2 Natural Language Understanding

The goal of the NLU module is to interpret an input given as a text string, usually obtained from the ASR module, and translate it to an abstract semantic representation understood by the system. Roughly speaking there are two implementation approaches: knowledge-based and statistical.

First NLU systems were knowledge-based ones. Based on the representation schemes and reasoning mechanisms used can be classified as follows:

- *Procedural semantics based*: Knowledge is represented as an executable program in a computer language. Systems using this approach can be found in [6] and [7].
- *Semantic Networks based*: Meaning of words and sentences is represented as a set of nodes linked in a graph. An example of a system using this approach is the MARGIE system [8].
- *Frames based*: Knowledge is represented as data structures called frames. CMU's Phoenix [9] is an example of a system using this approach.

As an alternative to knowledge-based systems, statistical ones were developed. Statistical NLU systems can be understood as a pattern recognition problem and formulated as such [10]. Examples of systems using this approach are systems that use Hidden Understanding Models [11], Fertility Models [12] or Hidden Vector State (HVS) Models [13]. Almost all statistical NLU systems use the pattern recognition approach although it's possible to use different methods. As an example, [14] shows a system where the grammar is induced semiautomatically based on statistical clustering techniques.

### 2.3 Natural Language Generation

NLG is the process of getting computer programs to generate readable text in an ordinary language from non-linguistic inputs. Several techniques exist for NLG development, some of them are:

- *Canned text*: It's the simplest way of generating natural language. A database of text sentences is created in the development step and then used during the dialogue execution. This approach is trivial to implement but has the disadvantage of being very inflexible. This approach is usually used in systems to give error or warning messages.
- *Template-based*: The next level of sophistication can be achieved using template based systems. Such systems use pre-defined templates or schemas with slots that can be filled by looking up the relevant information in a table. This approach is used mainly for applications whose texts are regular in structure. Some systems using this approach are TG/2 [15], D2S [16] and YAG [17].
- *Phrase-based*: This technique uses a first phrasal pattern to match the top level of the input. In a second step, the parts of this pattern are expanded recursively into more specific phrasal patterns matching subportions of the input. In [18] is presented a system that uses this approach.
- *Feature-based*: In this framework, sentences are associated with a unique set of features. Features used usually are of the type positive/negative, interrogative/declarative, past/present, etc. A system using this approach is shown in [19].
- *Other techniques*: More sophisticated techniques include synchronous parsing [20] or methods based on statistical planning, adaptive and incremental at the turn level, that optimises NLG actions with respect to a data-driven objective function [21][22].

### 2.4 Text-To-Speech Synthesis

The TTS module is responsible for converting the sentence generated by the NLG module in an audio signal to be presented back to the user.

TTS can be seen as a two-stage process, Natural Language Processing (NLP)

and Digital Signal Processing (DSP). The NLP stage comprises a first pre-processing step, usually called text normalisation, in which the raw text is processed to convert non-word symbols, like numbers or abbreviations, into words. On a second stage, usually called text-to-phoneme or grapheme-to-phoneme conversion, phonetic transcriptions and prosodic units are assigned to the text.

Once the NLP stage has finished, the DSP stage is responsible for generating an audio signal from the data provided by the NLP stage. To this end, usually, one of two methods is used. The simplest one, called *concatenative synthesis*, consists in concatenating pieces of prerecorded speech that are stored in a database. Usually, the speech is created from phones or diphones concatenation, but also can be found systems that concatenate entire words or sentences. This kind of systems has the advantage of being easy to implement but lacks human's speech naturalness.

Alternatively, *formant synthesis* is widely used. In this case, the speech obtained is purely synthetic and is obtained using additive synthesis and an acoustic model of the vocal tract. The main disadvantage of this method is that it generates robotic artificial voices, but on the other hand, has the advantage of being highly intelligible, and being able of avoiding the acoustic glitches, one of the main disadvantages of concatenative synthesis.

Although *concatenative* and *formant synthesis* are two of the most used techniques others like *articulatory synthesis* [23], based on models of the human vocal tract and the articulation processes occurring there, or techniques based on the use of HMM [24] have been used improving the quality of the generated voices.

## 2.5 Dialogue Management

The DM is a fundamental component of an SDS. Its primary task is to control the dialogue flow. Taking into account the state of the conversation is responsible for selecting the appropriate actions to be taken by the system.

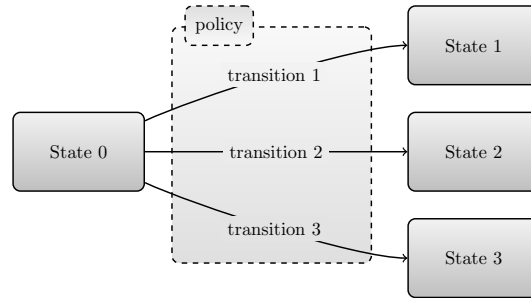
According to the implementation approach taken, the DMs can be classified as State-Based, Agent-Based, Information State-Based and Statistical.

### 2.5.1 State-Based

Formally, a State-Based DM can be defined using three concepts: *dialogue state*, *dialogue state transition* and *dialogue policy*.

A *dialogue state* represents the state of the dialogue in every moment. To maintain such state, the DM keeps an internal copy of the full dialogue history, the context and the entire sequence of input observations for the dialogue. The *dialogue state* depends on a *dialogue state transition function* that is

responsible for updating the *dialogue state* taking into account the user and system acts. Finally, the *dialogue policy* is in charge of deciding the behaviour in each state, or in other words, deciding which transition function apply to a state between a group of transition functions. Figure 2.3 shows the DM definition graphically.

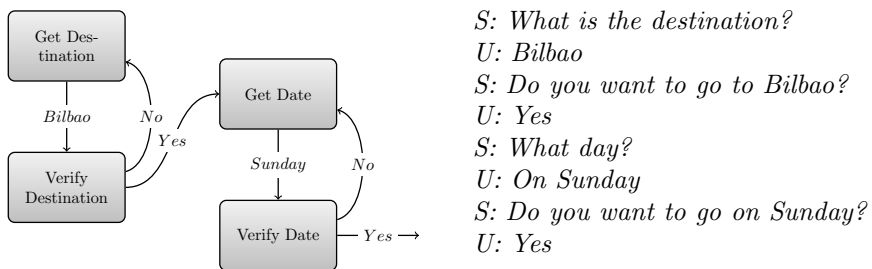


**Figure 2.3:** DM components example.

There are two main implementation approaches for State-Based DMs: Finite State-Based and Frame-Based.

### 2.5.1.1 Finite State-Based

The Finite State-Based architecture is one of the simplest methods. Dialogue states are represented by nodes in a graph and dialogue state transition functions are represented by the edges in the graph. As to dialogue policy, it depends entirely on the user acts, depending on them one transition function or another will be chosen. Figure 2.4 shows an example of a Finite State-Based DM.



**Figure 2.4:** Finite State Based dialogue manager example.

This type of architecture has the advantage of being relatively easy to implement. Another advantage is that the grammars and vocabularies for each of the states are foreknown. But there are some disadvantages. One of these disadvantages is that neither over-informative nor unexpected answers are allowed. In case of receiving an over-informative input, the information not



associated with the current state will be discarded. In case of receiving an unexpected input, the behaviour of the system will depend on the error handling strategies implemented. As an example let's take into account the dialogue in Figure 2.5 and let's suppose "Get Destination" is the current state. In such situation, the DM is expecting to receive "Bilbao" or "Madrid" as input. In case of receiving an over-informative input, for example, "Bilbao on Sunday", the DM will only be able to interpret "Bilbao" and will discard "Sunday", asking for "Date" when "Get Date" is the current state. On the other hand, if the DM receives "Barcelona" as input, in this particular example, the dialogue will remain in the current state and will ask for "Destination" again.

Other disadvantages are, on the one hand, that the user ability to ask questions and taking the initiative is discarded, and on the other hand, that the kind of dialogues obtained are not very natural.

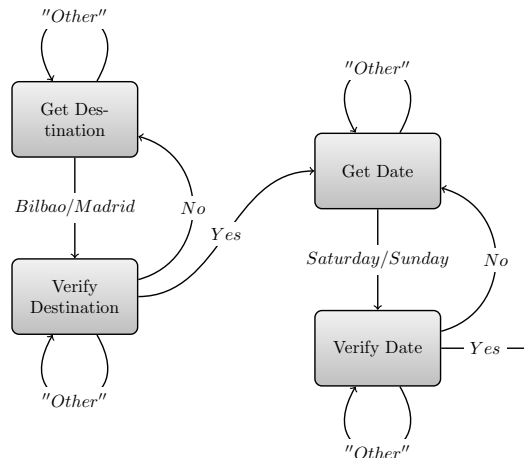


Figure 2.5: Finite State Based dialogue manager example.

### 2.5.1.2 Frame-Based

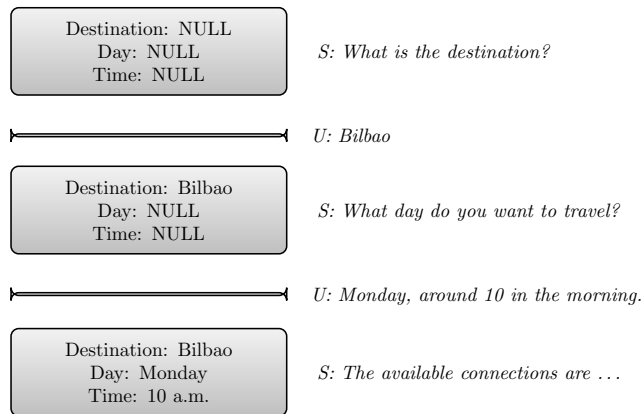
An alternative to the Finite State-Based DM is the *Frame-Based* DM, also known as *Form-Filling dialogue managers* [25]. In its basic form, this kind of DM uses the concept of *frames*. A frame is a data structure consisting of a set of *slots*, concepts the user can talk about, which can take on predefined values. The DM asks the user for values for the slots by using predefined actions for each set of known slots. Figure 2.6 shows a dialogue example using frames.

An issue with Frame-Based DM is that dialogues, often, are composed of smaller sub-dialogues, and frames cannot model the flow of the dialogue easily. To face this issue, it's usual to use extended versions of the basic frames concept using additional data structures. As examples, in [26][27] are presented systems that use an additional control table to decide the operations to perform based on the content of the frame and in [28][29] systems that use

three different data structures. Some other implementations can be found in [30][31][32].

Speaking regarding states, transfer functions and policies, it can be considered that the DMs state is represented by the frame and the transfer function by the updating process of the frame, that is, the filling of the slots. Regarding the policy it can be implemented in several ways, in fact, almost each implementation of a DM has its policy. As an example, in [25] is shown a DM whose policy assigns priorities to the slots and the system asks the user for information based on these priorities.

With this implementation approach, more natural dialogues are achieved because over-informative user inputs are allowed and, on the other hand, for the possibility the user has for speaking about any slot at any moment, giving more freedom to the user. As a disadvantage, it can be said that given the freedom the user has, grammars become harder to develop.



**Figure 2.6:** Frame Based dialogue manager example.

### 2.5.2 Agent-Based

In this kind of DM, also known as *plan-based dialogue managers* [33], the dialogue is viewed as an interaction between intelligent agents capable of reasoning. Each of the agents has a plan (a goal) and tries to achieve it interacting with the other agents when necessary. To define this class of models in an easier way let's take a look at an intuitive description taken from [34]:

*A has a goal to acquire some information; he creates a plan that involves asking B a question whose answer will provide the information. A then executes his plan, asking B the question. B receives the question and attempts to infer A's plan. In this plan, there may be goals that A cannot achieve without assistance. These are the obstacles in A's plan. B can accept some of these obstacles*

*as his own goals and create a plan to achieve them. B's response is generated when he executes this plan.*

The main advantages of this type of DM is that is suitable for the development of SDS in complex domains and also allows the user the use of a relatively unconstrained natural language. On the other hand, these type of systems are usually hard to build, and the agents themselves are very complex. Examples of this kind of DM are CMU's RavenClaw [35] and DISCO [36].

### 2.5.3 Information State-Based

The Information State-Based DM [37] is intended to combine the strengths of both State-Based and Agent-Based DMs. It models the dialogue with the following components:

- An *Information State* consisting of data structures representing the dialogue components.
- A set of *dialogue moves* that trigger the update of the Information State.
- A set of *update rules* responsible for updating the Information State.
- An *updating strategy* responsible for selecting which update rule(s) to use.

The central idea of information state based DM is that dialogue acts correspond to dialogue moves and are used to update the information state. The information state represents the history of everything happened in the dialogue, and the DM uses it to choose the next dialogue step based on an update strategy. Several methods for the update strategy can be used. As an example in [38] one based on logic programming is presented.

### 2.5.4 Statistical

The Statistical Dialogue Managers tend to be the state-of-the-art for SDSs. Statistical DMs were initially based on the Markov Decision Process (MDP) [39]. MDPs came into the Partially Observable Markov Decision Process (POMDP) [40] where unknown user goals are modelled by an unknown probabilistic distribution over user states that encodes the uncertainty about both the ASR and the NLU. This approach is up-to-now considered as the state-of-the-art in SDSs [41][42] even if the global optimisation shows significant computational problems that have to be addressed when dealing with real users.

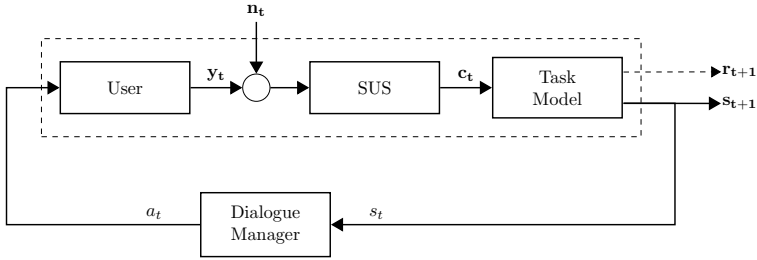
End-to-end learning for task-oriented dialogue systems was also proposed in [43] under an interactive pattern recognition approach that allowed dialogue generation based on stochastic bi-automata generative models. Recently other machine learning methods have been proposed to get optimal dialogue strategies from data such as deep neural networks and reinforcement learning [44] or hierarchical recurrent neural networks to generate open domain dialogues

[45] and build end-to-end SDSs.

#### 2.5.4.1 MDP and POMDP

Within the MDP and POMDP framework an SDS can be defined as follows:

*An SDS is controlled by a DM [1] which generates a sequence of actions  $a_t$  dependent upon the system state  $s_t$ . The goal of the DM is to change the system from some initial uninformed state to a sufficiently informed state so that the user's information need can be satisfied [1]. The actions are primarily questions to the user although they can also result in accesses to a database. Questions to the user result in acoustic responses  $y_t$  which are corrupted by noise  $\mathbf{n}_t$  before being input to the Speech Understanding System (SUS). The concepts  $\mathbf{c}_t$  output from the SUS cause the system memory to be updated leading to a new dialogue state  $s_{t+1}$ . The relationship between the dialogue state and the concepts output by the SUS is determined by the particular task. Also, each step taken by the DM results in the generation of a reward  $r_{t+1}$ .*



**Figure 2.7:** Block diagram of a Spoken Dialogue System proposed in [1].

The rewards generated at each dialogue step represent the system design objectives and the overall goal of the system design process is to maximise the total reward  $R = \sum_{t=1}^T r_t$ . Thus, the Dialogue Design is defined as an optimisation problem [46]. Typically, each user interaction will incur a small negative reward and successfully meeting the user's information need will generate a large positive reward [46]. The complete sequence of user interactions leading from the initial state to the final state is called a dialogue transaction and the primary goal is to satisfy the user's requirements while minimizing the transaction time and possible additional design criteria to be incorporated into the reward function. Within this framework, the joint distribution for the state, action, speech and concept sequence can be decomposed as:

$$P(s_{t+1}, a_t, \mathbf{y}_t, \mathbf{c}_t | s_t, \mathbf{n}_t) = P(s_{t+1} | \mathbf{c}_t, s_t) P(a_t | s_t) P(\mathbf{y}_t | a_t, s_t) P(\mathbf{c}_t | \mathbf{y}_t, a_t, s_t, \mathbf{n}_t) \quad (2.1)$$

where

- $P(s_{t+1} | \mathbf{c}_t, s_t)$  is the Task Model.

- $P(a_t|s_t)$  (DM) represents the dialogue control strategy: at each state of the dialogue, what is the best action to take next?
- $P(\mathbf{y}_t|a_t, s_t)$  (User) represents the response of each user to receiving a specific query action in a given state. A model of the typical user or the statistics of a real user population.
- $P(\mathbf{c}_t|\mathbf{y}_t, a_t, s_t, \mathbf{n}_t)$  (SUS) represents the speech recognition and understanding processes. The interpretation of a user input  $\mathbf{y}_t$  can depend on the query action, the dialogue state and the ambient noise conditions.

The SUS term can be expanded as:

$$P(\mathbf{c}|\mathbf{y}, a, s, \mathbf{n}) \approx \max_{\mathbf{w}} P(\mathbf{c}|\mathbf{w}, a, s) P(\mathbf{w}|\mathbf{y}, a, s, \mathbf{n}) \quad (2.2)$$

Term  $P(\mathbf{w}|\mathbf{y}, a, s, \mathbf{n})$  defines the conventional speech recognition problem which, ignoring the noise term, can be written as:

$$P(\mathbf{w}|\mathbf{y}, a, s) = \frac{P(\mathbf{y}|\mathbf{w})P(\mathbf{w}|a, s)}{P(\mathbf{y}|a, s)} \quad (2.3)$$

where  $P(\mathbf{y}|\mathbf{w})$  is the acoustic likelihood of the observed speech given the word sequence  $\mathbf{w}$  and  $P(\mathbf{w}|a, s)$  is the prior probability of the word sequence  $\mathbf{w}$  given the action and dialogue state. The interesting aspect is the dependence of  $P(\mathbf{w})$  of action  $a$  and state  $s$ . This is the crucial feature that makes complex interactive SDS feasible. The language model used at each question/answer step can be highly context dependent. For example, it can be tightly constrained to a specific set of possible input utterances or it can be a stochastic context dependent n-gram.

The mapping of words to concepts as represented by  $P(\mathbf{c}|\mathbf{y}, a, s, \mathbf{n})$  is typically done by mapping phrases to semantic tags or predicates. This mapping can be done manually using explicit grammars or using n-grams. These approaches can also be combined. Most current approaches identify the salient phrases to extract and map manually. However, if the task is simple enough and there is a large body of data to learn from, it's possible to identify the salient phrases automatically.

## MDP

A MDP is a 4-tuple  $(S, A, T, R)$  where

- $S$  is a finite set of states.
- $A$  is a finite set of actions (alternatively  $A_s$  is the finite set of actions available from state  $s$ ).
- $T$  is a transition function, i.e., is a description of each action's effects in each state. Deterministic actions: for each state and action we specify a new state,  $T : S \times A \rightarrow S$ . Stochastic actions: for each state and action we specify a probability distribution over next states,  $P(s'|s, a)$  represents the distribution;  $T(s', a, s) = P(s_{t+1} = s' | a_t = a, s_t = s)$ .

- $R$  is the reward (expected) received after transition to state  $s'$  from state  $s$  with transition probability  $P(s'|s, a)$ .  $R : S \times A \rightarrow \mathcal{R}$ .  $R(s', a, s) = \mathcal{E}(r_{t+1} | s_{t+1} = s', a_t = a, s_t = s)$

From the point of view of the DM the components within the dotted box in Figure 2.7 can be regarded as a single system driven by the input actions  $a_t$  [1]. Two assumptions are necessary [46] to describe a dialogue system as a MDP for which techniques exist for finding the optimal strategy.

- First assumption. It concerns assigning a probabilistic model to state transitions. When an action  $a_t$  is taken at time  $t$  while in state  $s_t$ , the MDP state changes to  $s_{t+1}$  according to transition probabilities with the following Markovian property:

$$P(s_{t+1} | s_t, s_{t-1}, \dots, s_0, a_t, a_{t-1}, \dots, a_0) = P_T(s_{t+1} | s_t, a_t) \quad (2.4)$$

- Second assumption. It concerns stochastic modelling rewards. If we define the *session reward* as a sum of all the rewards experienced by the system during a dialogue session (a path in the state space starting in the initial state, and ending in the final state), the objective function for MDP is the expected session reward:

$$\mathcal{R} = \sum_{t=1}^T R(s_{t+1}, a_t, s_t) \quad (2.5)$$

And the goal is usually to find a strategy (*policy*) which maximises this function. For the infinite time horizon the expected cumulative reward  $\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})$  can include a discount factor  $\gamma$  that satisfies  $0 \leq \gamma \leq 1$  (typically close to 1). Discount  $\gamma$  ensures that rewards accrued later are counted less than those accrued earlier. The discount factor can be used to model processes that can be terminate at any time with probability  $1 - \gamma$ .

The assumption underlying this model is that the user always complies with the system, i.e. always provides, as a response, only the information he was asked for, and his answer does not depend on any other information but the one described by the current state and action pair (the Markovian property) [46].

The core problem of MDP is to find an optimal strategy, i.e. *policy*, for the decision maker: a function  $\pi$  that specifies the action  $\pi(s)$  that the decision maker will choose when in state  $s$ . A *policy* is a mapping between states and actions. Thus, in MDP and in order to decide which is the optimal action to take in a given state, there is no need to consider other information except that included in the state itself, e.g. no need to consider past states and actions. Note that once a Markov decision process is combined with a policy in this way, this fixes the action for each state and the resulting combination behaves like a Markov chain. The policy matrix  $\pi(s, a) = P(a_t = a | s_t = s)$  represents the dialogue management strategy and the reward represents the

objective function. The goal is to choose a policy that will maximize the reward. Because of the Markov property, the optimal policy for this particular problem can indeed be written as a function of  $s$  only, as assumed above.

### POMDP

A POMDP is defined as a tuple  $\{S, A, T, R, O, Z, \lambda, b_0\}$  where:

- $S$  is a set of states describing the agent's world
- $A$  is a set of actions that an agent may take
- $T$  defines a transition probability  $P(s'|s, a)$
- $R$  defines the expected reward  $r(s, a)$
- $O$  is a set of observations the agent can receive about the world
- $Z$  defines an observation probability  $P(o'|s', a)$
- $\lambda$  is a geometric discount factor  $0 \leq \lambda \leq 1$
- $b_0$  is an initial belief state  $b_0(s)$

Thus a POMDP is a MDP  $\{S, A, T, R\}$  plus  $\{O, Z, \lambda, b_0\}$ . The POMDP operates as follows. At each time-step, the world is in some unobserved state  $s \in S$ . Since  $s$  is not known exactly, a distribution over states is maintained called "a belief state",  $b$ , with initial belief state  $b_0$ . We write  $b(s)$  to indicate the probability of being in a particular state  $s$ . Based on  $b$ , the machine selects an action  $a \in A$ , receives a reward  $r(s, a)$ , and transitions to unobserved state  $s'$ , where  $s'$  depends only on  $s$  and  $a$ . The machine then receives an observation  $o' \in O$  which is dependent on  $s'$  and  $a$ . At each time-step, the belief state distribution  $b$  is updated as follows:

$$b'(s') = k p(o'|s', a) \sum_{s \in S} p(s'|a, s) b(s) \quad (2.6)$$

This equation consists of the observation function  $Z$ , transition matrix  $T$  and current belief state  $b(s)$ . At each time-step the agent receives reward  $r_t$ . The cumulative, infinite-horizon, discounted reward is called the *return*:

$$\Theta = \sum_{t=0}^{\infty} \lambda^t r_t \quad (2.7)$$

where  $\lambda$  is the geometric discount factor  $0 \leq \lambda \leq 1$ . The goal of the machine is to choose actions in such a way as to maximize the expected return  $\mathcal{E}[\Theta]$  to construct a *plan* called *policy* which indicates which actions to take at each turn (we assume the planner has a model of system dynamics,  $T$ ,  $R$  and  $Z$  are known or can be estimated from training data, i.e. model-based learning). A *policy* can be viewed as a mapping from belief state to action  $\pi(s) \in A$  and an optimal policy  $\pi^*(b) \in A$  is one which maximizes  $\mathcal{E}[\Theta]$ .

#### 2.5.4.2 Stochastic Finite State Bi-Automata

A SDS can be considered as an interactive pattern recognition system [47] in which the system derives an hypothesis from some input stimulus, according to some previously obtained model.

In this framework, let  $h$  be the hypothesis derived by the DM of a SDS and  $f$  the input stimulus, i.e. user's feedback. The hypotheses of the dialogue manager are usually called actions. These actions typically consist of machine turns that include queries to a database to get the information required by the user, questions to the user to complete the data the system needs to fulfill user goals, strategies to recover recognition or understanding errors, turns providing information to the user as well as greeting turns.

To simplify, we can ignore the user feedback except for the last hypothesis  $h$ . Assuming the classical minimum-error criterion, the Baye's decision rule is simplified to maximize the posterior  $P(h|h', f)$ , and a *best* hypothesis  $\hat{h}$  is obtained as follows:

$$\hat{h} = \arg \max_{h \in \mathcal{H}} P(h|h', f) \quad (2.8)$$

This maximisation procedure defines the way the dialogue manager of an SDS chooses the best hypothesis in the space of hypotheses  $\mathcal{H}$ , i.e. the best action at each interaction step, given the previous hypothesis  $h$  and the user feedback  $f$ .

In a SDS, the interpretation of the user feedback  $f$  can not be considered a deterministic process. In fact the space of decoded feedbacks  $\mathcal{D}$  is the output of an ASR system. Thus a *best* hypothesis can be obtained as follows [47][48]:

$$\hat{h} = \arg \max_{h \in \mathcal{H}} \sum_{d \in \mathcal{D}} P(h, d|h', f) \simeq \arg \max_{h \in \mathcal{H}} \max_{d \in \mathcal{D}} P(h|d, h') P(f|d) P(d|h') \quad (2.9)$$

where  $f$  is the user turn,  $d$  is the decoding of the user turn,  $h$  is the hypothesis or the output produced by the system and  $h'$  is the history of the dialogue. A suboptimal approach can be considered through a two step decoding: find first an optimal user feedback  $\hat{d}$  and then, use  $\hat{d}$  to decode system hypothesis  $\hat{h}$  as follows:

$$\hat{d} = \arg \max_{d \in \mathcal{D}} P(f|d) P(d|h') \quad (2.10)$$

$$\hat{h} \simeq \arg \max_{h \in \mathcal{H}} P(h|\hat{d}, h') \quad (2.11)$$

On the other hand, the development of a complete SDS requires an online learning to train the DM strategy. Therefore, a large amount of dialogues is needed together with real users with different goals, expectations and behaviours. Thus, statical DMs are usually trained by simulated users [46].

A *simulated* user must provide the feedback  $f$  to the system at each interaction step. The user feedback  $f$  depends on its previous feedback  $f'$  according to some unknown distribution  $P(f|f', h)$ , which represents the user response



to the history of system hypotheses and user feedbacks. This distribution considers the user behaviour and stands for the user model  $\mathcal{M}_u$  and can also be defined considering now the user point of view. However, feedback  $f'$  produced by the user in the previous interaction is not corrupted by any noisy channel, such as an ASR system, before arriving to the user again. Thus, a deterministic decoding  $d : F \rightarrow D$  maps each user turn signal into its corresponding unique decoding  $d' = d(f')$  before arriving to the user model. Consequently the best decoded user feedback  $\hat{d}$  is the one that maximizes the posterior  $P_{\mathcal{M}_u}(d|d', h)$ ,

$$\hat{d} = \arg \max_{d \in \mathcal{D}} P(d|d', h) \simeq \arg \max_{d \in \mathcal{D}} P_{\mathcal{M}_u}(d|d', h) \quad (2.12)$$

where  $\hat{d}$  is estimated using only the hypothesis produced by the system and the feedback produced by the user in the previous interaction step according to its user model. Equation 2.12 represents the way the user model decides the feedback to be produced at each interaction step.

Both the DM hypothesis probability distribution  $P(h|d, h')$  and the user feedback probability distribution  $P(d|h, d')$  can be modelled with Stochastic Finite State Bi-Automata.

### Model Definition

We first summarise the basic definitions of bi-string and stochastic regular bi-language provided in [49].

*Definition 1. A bi-language is a set of strings over an extended alphabet  $\Gamma$ , i.e., a set of strings of the form  $\mathbf{b} = b_1 \dots b_k$  such that  $b_i \in \Gamma$  for  $0 \leq i \leq k$ . A string over an extended alphabet  $\Gamma$  will be called bi-string.*

*Definition 2. Given two finite alphabets  $\Sigma$  and  $\Delta$ , a stochastic bi-language  $\mathcal{B}$  is a probability distribution over  $\Gamma^*$  where  $\Gamma \subseteq (\Sigma^{\leq m} \times \Delta^{\leq n})$ ,  $m, n \geq 0$ . Let  $\mathbf{z} = z_1 \dots z_{|\mathbf{z}|}$  be a bi-string such that  $z_i \in \Gamma$  for  $1 \leq i \leq |\mathbf{z}|$ . If  $P_{r\mathcal{B}}(\mathbf{z})$  denotes the probability of the bi-string  $\mathbf{z}$  under the distribution  $\mathcal{B}$  then  $\sum_{\mathbf{z} \in \Gamma^*} P_{r\mathcal{B}}(\mathbf{z}) = 1$ .*

Let  $\Sigma$  be the finite alphabet of semantic symbols provided by some speech understanding system. Thus,  $\tilde{d}_i = d_1 \dots d_{|\tilde{d}_i|} \in \Sigma^{\leq m}$  represents the decoding of a user feedback  $f$ . Let now  $\Delta$  be the finite alphabet of dialogue acts that compose each of the hypotheses  $\tilde{h}_i = h_1 \dots h_{|\tilde{h}_i|} \in \Delta^{\leq n}$  provided by the dialogue manager. Let  $z$  be a bi-string over the extended alphabet  $\Gamma \subseteq \Sigma^{\leq m} \times \Delta^{\leq n}$  such as  $\mathbf{z} : \mathbf{z} = z_1 \dots z_{|\mathbf{z}|}$ ,  $z_i = (\tilde{d}_i, \tilde{h}_i)$  where  $\tilde{d}_i = d_1 \dots d_{|\tilde{d}_i|} \in \Sigma^{\leq m}$  and  $\tilde{h}_i = h_1 \dots h_{|\tilde{h}_i|} \in \Delta^{\leq n}$ . Extended symbols  $(\tilde{d}_i, \tilde{h}_i) \in \Gamma$  have been obtained through some alignment between  $\Sigma^{\leq m}$  and  $\Delta^{\leq n}$ , i.e. between pairs of user feedbacks decoding provided at a user turn and dialogue manager hypotheses provided at the next machine turn.

Let us now define a Dialogue Model  $\mathcal{DM}$  as a Deterministic and Probabilistic Finite-State BiAutomaton [49]  $\mathcal{DM} = (\Sigma, \Delta, \Gamma, Q, \delta, q_0, P_f, P)$  where

- $\Sigma$  and  $\Delta$  are two finite alphabets representing semantic symbols provided by the user and dialogue acts provided by the DM respectively,  $\Gamma$  is an extended alphabet such that  $\Gamma \subseteq (\Sigma \leq m \times \Delta \leq n), m, n \geq 0$ .  $\epsilon$  represents the empty symbol for both alphabets, i.e.,  $\epsilon \in \Sigma, \epsilon \in \Delta$  and  $(\tilde{\epsilon}, \tilde{\epsilon} \in \Gamma)$ . To simplify let  $\tilde{\epsilon} \in \epsilon$ .
- $Q = Q_{\mathcal{M}} \cup Q_{\mathcal{U}}$  is a finite set of states labelled by bi-strings  $(\tilde{d} : \tilde{h}) \in \Gamma$ . The set  $Q_{\mathcal{M}}$  includes machine states before a machine turn providing an hypothesis and the set  $Q_{\mathcal{U}}$  includes user states before providing a feedback.
- $\delta \subseteq Q \times \Gamma \times Q$  is the union of two sets of transitions  $\delta = \delta_{\mathcal{M}} \cup \delta_{\mathcal{U}}$  as follows:
  - $\delta_{\mathcal{M}} \subseteq Q_{\mathcal{M}} \times \Gamma \times Q_{\mathcal{U}}$  is a set of transitions of the form  $(q, (\epsilon : \hat{h}_i), q')$  where  $q \in Q_{\mathcal{M}}, q' \in Q_{\mathcal{U}}$  and  $(\epsilon : \hat{h}_i) \in \Gamma$ .
  - $\delta_{\mathcal{U}} \subseteq Q_{\mathcal{U}} \times \Gamma \times Q_{\mathcal{M}}$  is a set of transitions of the form  $(q, (\hat{d}_i : \epsilon), q')$  where  $q \in Q_{\mathcal{U}}, q' \in Q_{\mathcal{M}}$  and  $(\hat{d}_i : \epsilon) \in \Gamma$ .
- $q_0 \in Q_{\mathcal{M}}$  is the unique initial state and it's labelled as  $(\epsilon : \epsilon)$ .
- $P_f : Q \rightarrow [0, 1]$  is the final-state probability distribution.
- $P : \delta \rightarrow [0, 1]$  defines transition probability distributions  $(P(q, b, q') \equiv P_r(q', b|q))$  for  $b \in \Gamma$  and  $q, q' \in Q$  such that:

$$P_f(q) + \sum_{b \in \Gamma, q' \in Q} P(q, b, q') = 1 \quad \forall q \in Q \quad (2.13)$$

where a transition  $(q, b, q')$  is completely defined by  $q$  and  $b$ . Thus,  $\forall q \in Q, \forall b \in \Gamma, |q' : (q, b, q')| \leq 1$ .

Let  $\mathbf{z}$  be a bi-string over the extended alphabet  $\Gamma \subseteq \Sigma^{\leq m} \times \Delta^{\leq n}$  such as  $\mathbf{z} : \mathbf{z} = z_1 \dots z_{|\mathbf{z}|} = (\hat{d}_i : \hat{h}_i)$ .  $\mathbf{z}$  represents a dialogue when  $z_i$  is of the form  $z_i = (\epsilon : h_i)$  for machine turns  $m_i$  and  $z_i = (\hat{d}_i : \epsilon)$  for user turns  $u_i$ . Both, user and machine turns can also be null bi-strings of the form  $(\epsilon : \epsilon)$ . Let now  $\theta = q_0, z_1, q'_1, z_2, q_2, \dots, q'_{|\mathbf{z}|-1}, z_{|\mathbf{z}|}, q_{|\mathbf{z}|}, q_i \in Q_{\mathcal{M}}, q'_i \in Q_{\mathcal{U}}$ , be a path for  $\mathbf{z}$  in  $\mathcal{DM}$ . The probability of generating  $\theta$  is:

$$P_{r_{\mathcal{DM}}}(\theta) = \left( \prod_{j=1}^{|\mathbf{z}|} P(q_{j-1}, z_j, q'_j) \right) \cdot P_f(q_{|\mathbf{z}|}) \quad (2.14)$$

$\mathcal{DM}$  is unambiguous. Then, a given bi-string  $\mathbf{z}$  can only be generated by  $\mathcal{DM}$  through a unique valid path  $\theta(\mathbf{z})$ . Thus, the probability of generating  $\mathbf{z}$  with  $\mathcal{DM}$  is  $P_{r_{\mathcal{DM}}}(\mathbf{z}) = P_{r_{\mathcal{DM}}}(\theta(\mathbf{z}))$ .

### 2.5.4.3 Neural Networks

Initially the neural networks were used together with the POMDP in an approach in which the neural networks were used to track the dialogue state represented by the Markov process. A far more ambitious approach is the *end2end* approach, in which a neural network is in charge of processing the input (in text) produced by the user to generate the output, i.e. the response phrase (also in text) of the system.

The architecture of these *end2end* systems is based on the *seq2seq* machine translation network proposed by Google [50]. This network was designed to improve the Google Translate service, and allows to produce a variable length output sequence from a variable length input sequence. Its structure consists of an encoder and a decoder. Both the encoder and the decoder are recurrent neural networks, usually formed from Long Short Term Memory (LSTM) cells. The encoder provides the abstraction of the input sequence, which the decoder takes as input to produce the output sequence.

Several versions have been created from the original *seq2seq* architecture, being one of the most commonly used the *seq2seq* architecture with attention mechanisms [51], which is inspired by how humans perceive the environment. The general idea is to pay attention only to the most informative or relevant parts of the input sequence.

The training process of these *seq2seq* networks usually consists of two phases. The usual thing is to start with a supervised classification phase from a dialogue corpus [52], where the network is trained so that, given a turn in the corpus, it generates the next turn in the dialogue. In a second phase another training is usually carried out through reinforcement learning, where the behaviour of the network is adapted to improve, for example, semantic coherence in sentences or the introduction of new information with respect to the previous turns.

*End2end* architectures have demonstrated a great capacity for abstraction and understanding of dialogue. Even so, the challenge of adapting these systems to dialogue problems with user-oriented objectives remains difficult to address. There are approximations such as [53][54], which proposes a learning adaptation by reinforcement that allows interaction learning to address this challenge.

---

## 2.6 Other Components

The task of building and emulating dialogues with persons is a very complex task. So far we have seen the essential components that are part of a primary SDS. But systems tend to be increasingly complex and to emulate more precisely the interaction with users. That is achieved by adding new functionalities to the systems and, therefore, new components to manage these functionalities. Examples of these components could be: Users' gesture de-

tectors [55], Emotion Classifiers [56] or Topic Classifiers for Customer Service Dialogue Systems [57].

### 3 Examples of Spoken Dialogue Systems

Following is a summary of several existing SDSs and speech-based applications:

- Let's Go! a project working in the domain of bus information, providing information such as schedules and route information for the city of Pittsburgh's Port Authority Transit (PAT) buses [58].
- SimSensei Kiosk, a virtual human dialogue system that conducts interviews related to psychological distress conditions such as depression, anxiety, and post-traumatic stress disorder (PTSD) [59].
- JUPITER [26] was one of the first SDS released to the public. It's a conversational interface that allows users to obtain worldwide weather forecast information over the telephone using spoken dialogue.
- The Philips automatic train timetable information system [60], a system for train schedules information over the phone that provides actual connections between 1200 German cities in which the caller can talk to it in free, natural, and fluent speech.
- Speak&Eat, San Francisco. Is a conversational application to have a natural conversation to find restaurants in San Francisco, developed by the European FP7 project "Parlance" [61][62].
- The Alex Public Transit Information System is an experimental Czech spoken dialogue system providing information about all kinds of public transit in the Czech Republic [63].
- CMU Communicator [28] acts as a travel planning agent and can arrange multi-leg itineraries and make hotel and car reservations.
- TOOT [64] gives spoken access to train schedules.
- Presenter [65] provides a continuous listening command and control interface to PowerPoint presentations.
- WITAS [66] provides a spoken language interface to an autonomous robotic helicopter.
- AdApt [67] provides real-estate information in the Stockholm area.
- TRIPS [68] is a spoken-language enabled planning assistant.
- Hound [69], Siri [70], Cortana [71], Alexa [72] and Google Now [73] are applications for general information retrieval such as looking up the weather, placing phone calls, sending text messages, finding hotel reservations, navigating to an address, etc.



# 3

## FRAMEWORKS AND TOOLKITS

---

This chapter summarises a set of existing frameworks for the development of SDSs. That is an important task because depending on the specifications of the system we want to develop we can opt for the use of one framework or another. Among these specifications to take into account the platform and Operating System (OS) on which the framework works or the set of components it provides. Following sections summarise the main features of each of the frameworks. In Table 3.1 is shown a summary of them.

### 1 PC under Windows OS

---

#### **Olympus**

Olympus [74] is a complete framework for building SDS developed at Carnegie Mellon University (CMU) based on works by Alex Rudnicky, leading investigator at CMU and director of the Carnegie Mellon Speech Consortium, developed during late 2000. It incorporates all the necessary modules for building SDSs:

- The DM is called *RavenClaw* [75] and is based on AGENDA [29] that supports mixed-initiative dialogues. The dialogues are represented by a tree which nodes represent the hierarchical structure of the dialogue, and its depth levels encapsulate dialogue actions (questions, answers, databases access, etc.) and by an Agenda consisting of a list of agents that via a dispatcher operates on the dialogue tree. It can be seen as a plan based architecture.
- For ASR, CMU SPHINX speech recogniser [76] is used.
- NLU is achieved by *Phoenix* a semantic parser based on Context Free Grammars (CFG).
- Responsible of NLG is *Rosetta* a template based system written in Perl. Language generation is specified by a form, a sub-form and attribute values that are substituted into the templates.
- TTS module is called *Kalliope* and can use Festival, SAPI, CMU's Flite and CEPSTRAL Swift speech synthesisers.
- Communication between modules is done via Galaxy Communicator architecture [77].

Anyway, it's possible to develop custom modules using the libraries provided by the distribution. The primary programming language is C/C++, but it's also possible to develop modules using Java or scripting languages like Perl. It's under modified BSD Open Source license. Last stable version is 2.6.1, dated 05/2015 and is still maintained.

### **Flipper MMDS**

Flipper Multi-Modal Dialogue System (FlipperMMDS) [78] is a framework that makes it easier to develop fully featured Dialogue Systems. It's an addition to the Flipper DM. It has been drawn up by Siewart van Wingerden and Jelte van Waterschoot at the Humand Media Interaction group at the University of Twente. Flipper is an Information State DM originally developed by Mark ter Maat. Using XML-templates, the preconditions, effects and behaviours of dialogue rules can be described. FlipperMMDS last updated version is from 10/08/2016 and Flipper last updated version is from 27/10/2016.

### **IrisTK**

IrisTK [79] is a Java-based framework for developing multi-modal dialogue systems.

According to developers website it provides:

- A framework for defining event-based dialogue systems with modules running asynchronously.
- A statechart-based XML formalism for representing the dialogue flow (similar to SCXML).
- Generic interfaces for speech recognition (ASR) and speech synthesis (TTS) for easy deployment of these in dialogue systems.
- A set of modules for Windows ASR & TTS, Microsoft Kinect, Nuance Recognizer 9, Nuance NDEV Cloud-based ASR (Dragon), Facial animation, The Furhat robot head.
- Support for distributed systems (over processes and/or machines).
- Tentative support for incremental processing.
- Logging of events and audio.

IrisTK is released as open source under GNU General Public License (GPL) license. Last version update is from 18/09/2015.

### **CSLU Toolkit**

CSLU toolkit [80] was developed at Center for Spoken Language Understanding (CSLU) OGI Campus at Oregon Health & Science University (OGI/OHSU) by John-Paul Hosom. It provides a framework and tools for interactive language systems development. These systems can incorporate ASR, NLU, TTS and facial animation technologies.

The core of the toolkit is written in C, but it offers a Tcl-based environment to develop spoken language systems and tools to perform related tasks (extracting acoustic features, training acoustic models, etc.). It also incorporates a tool called Rapid Application Developer (RAD) that graphically enables system development.

The DM uses a finite state dialogue management approach. To develop a system the developer needs to define a set of states and the set of words or sentences to trigger each state transition. On the other hand, the system provides a set of templates to handle events not reflected in the dialogue graph.

The license is limited to non-commercial use, where noncommercial use relates only to educational, research, personal or evaluation purposes. Last version update is from 14/02/2007 and, although it's possible to download it, seems to be unattended.

### **AT&T Statistical Dialogue Toolkit**

AT&T Statistical Dialogue Toolkit (ASDT) [81] is a statistical toolkit for building SDS. It maintains a distribution over multiple dialogue states. The core is a belief update engine that keeps a distribution over application-specific objects. To develop an application it's necessary to create these objects through the implementation of a set of methods for each object which the engine calls to update the distribution. The objects represent a partition of user goals. The updating method used by the engine is described in [82].

The distribution of ASDT provides an extensive set of examples and an end-to-end system. It's possible to use any speech recognition and text-to-speech engines, or use the provided by the AT&T Speech Mash-ups platform. The toolkit is written in Python. The license is limited to non-commercial use. Last version update is from 01/2010.

### **Ariadne**

Ariadne is an SDS developed by Dr Matthias Denecke at the Interactive System Laboratories at Carnegie Mellon University in USA and University of Karlsruhe in Germany.

The Ariadne distribution contains all the required components for developing a full SDS in Windows with SAPI 5.1. It requires Visual Studio C++ 6 for compilation and execution.

The architecture is a layered microkernel that allows replacing almost every component other than the main dialogue processing component. Last version update is from 06/09/2004. It's unattended and is not possible to download it from developer's site.



## 2 PC under Unix-like OS

---

### **vAssist (ActiveMQ based)**

In [83] is presented an open source framework, vAssist, for the implementation of SDSs providing a set of modules and tools. Modules communication is done via ActiveMQ [84], a message queuing protocol available for Java, C and other programming languages and which architecture is built around a broker whose task is to connect the registered clients, collect messages and distribute them to receivers of topics and queues. Development of new modules is possible just using the ActiveMQ API and the internal framework messaging protocol. Last version update is from 02/2015.

Systems using the vAssist framework and the ActiveMQ message queuing protocol are presented in Chapter 6.

### **Alex Dialogue Systems Framework**

Alex Dialogue Systems Framework [85] is a framework developed by the dialogue systems group at UFAL, the Institute of Formal and Applied Linguistics, Faculty of Mathematics and Physics, Charles University in Prague, the Czech Republic, with the purpose of to facilitate research into and development of SDSs.

It has the following features implemented:

- VOIP using PJSIP 2.1 with some modifications.
- ASR using GoogleASR or KALDI.
- VAD using Gaussian Mixture Models or Feed-Forward Neural Networks.
- NLU using a set of logistic regression classifiers for detecting dialogue acts.
- DM using probabilistic discriminative dialogue state tracking and hand-crafted policies.
- NLG using template based generation possibly with efficient inflexion into the correct surface form for morphologically rich languages.
- TTS using Flite, VoiceRSS and SpeechTech.

It uses Python as the programming language. Last version update is from 23/06/2016.

## 3 PC under Windows or Unix-like OS

---

### **EDECAN**

EDECAN [86] is a communication interface for development of distributed applications, based on TCP/IP protocols and XML messages interchange. It has been developed by the Communication Technologies Group (GTC group) of the University of Zaragoza. EDECAN architecture aims to provide the

developer with a multiplatform and easy to use interface. It's possible to add any service just developing a dynamic library and providing its configuration files. Although it's feasible to develop any distributed application, it was initially drawn up with SDSs implementation in mind.

It's written in C programming language. The technical specifications of the EDECAN framework can be found in Appendix A. Chapters 4 and 5 present some systems built using this framework.

### **OpenDial**

OpenDial [87] is a toolkit for developing domain-independent spoken dialogue systems written in Java. It combines logical and statistical approaches to model dialogues. The domain models are represented by probabilistic rules, and unknown parameters can be estimated using reinforcement learning techniques. The architecture is designed as a blackboard in which all modules are connected to a central information hub which represents the dialogue state.

The toolkit provides a set of plugins for integrating ASR, NLU, TTS and other components. Other custom modules also can be implemented. Last version update is from 05/04/2016.

### **Trindikit**

Trindikit [88] is a toolkit for building and experimenting with DMs based on information states. It was developed at Dialogue Technology Lab at the Centre for Language Technology at the University of Gothenburg by D. Traum and S. Larsson for TRINDI [89] and SIRIDUS [90] projects and then updated for TALK [91] project. TRINDI (Task Oriented Instructional Dialogue) is an European project developed by the Dialogue Technology Lab at the Centre for Language Technology at the University of Gothenburg, the Human Communication Research Centre (at the University of Edinburgh), the Computational Linguistics Department at the Universität des Saarlandes, SRI international in Cambridge and by Xerox Research Centre Europe. Trindikit is focused on dialogues between humans and machines that enable the human to make choices in the performance of a certain task, i.e. task-oriented instructional dialogues. SIRIDUS (Specification, Interaction and Reconfiguration in Dialogue Understanding Systems) is an FP5 project aimed at developing computational tools to support the development of robust, functional and user-friendly dialogue systems. It was developed by the Department of Linguistics at Goteborg University, Telefónica I&D, Department of Computer Linguistics at the University of Saarland and the Department of English Language at the University of Sevilla.

Although it's possible to develop custom modules the toolkit provides some useful ones like ASR, TTS, NLU, NLG and DM.

Main components of a system are:

- The Total Information State (TIS): consisting of the Information State

(IS) variable, the module interface variables and the resource interface variables.

- Modules with their behaviour defined by module algorithms.
- The Dialogue Move Engine (DME) which is responsible for updating the IS and selecting actions to be performed by the system.
- A controller responsible of intercommunicating the rest of the modules.
- External resources, such as databases, etc.

The IS is an object which type is defined by the developer, with its variables and properties, and determines the conditions and operations that can be performed on it. It can be accessed and updated by all the modules in the system.

When implementing a system, the developer must define the IS, modules with their respective TIS update rules and algorithms and a controller with its control algorithm.

Trindikit is under GPL but requires the use of SICStus Prolog that is under proprietary license. Last updated version is from 03/10/2007 but is still maintained nowadays.

### **DIPPER**

DIPPER (Dialogue Prototyping Equipment & Resources) is an initiative of the Language Technology Group at The University of Edinburgh and CSLI Stanford [92]. It's an architecture to develop dialogue systems based on information states and plan dialogue management approaches and integrated into an Open Agent Architecture (OAA). It takes some of the core ideas of Trindikit but simplifying them. On the other hand, uses Java as programming language and is more close-fitting to OAA than Trindikit. DIPPER is not a dialogue system itself but provides interfaces to ASR (Nuance), TTS (Festival), NLG and automated reasoning.

Because of using an interpreted language is OS independent. It's under GPL license, and last updated version is from 2004.

### **Midiki**

Midiki [93] is a toolkit developed by MITRE Corporation, a not-for-profit company that operates multiple federally funded research and development centres (FFRDCs) [94]. As Trindikit it's based on information states, in fact, Midiki is based on Trindikit but with the main difference of being written in Java and therefore there's no need for specialised languages for writing rules.

Midiki is just a DM and external components must be developed to create dialogue systems. Some modules are supplied in the distribution but are just test modules, anyway, it's possible to develop modules using Galaxy Communicator [77] and OAA.

The dialogue management algorithm is rule-based. Each rule evaluates the

information state and if all conditions are satisfied associated actions are executed and information state and environment updated. The algorithm resides in a process called DME that operates using two different algorithms, the first that updates the information state with information provided by users and a second one that selects the actions to be performed by the system.

It's OS independent because of using an interpreted language and is under an Open Source license. Last updated version is from 18/08/2005 and is not maintained anymore.

### **Jaspis**

Jaspis [95] is a framework for user and environment adaptive dialogue systems developed at the University of Tampere in Finland by Markku Turunen. It's designed with multilingual applications in mind. The architecture is based on the use of agents which shares the dialogue information, these agents are informed of information changes via triggers. It seems to be outdated and is not possible to download it.

### **Daisie**

Daisie (Diaspace's Adaptive Information State Interaction Executive) [96] is a dialogue management framework based on the use of agents and information states. It has the characteristic of not having a DM as a main module. Instead, the logic of the DM is coded as a library of functions that can be invoked from the application.

## **4** Other Platforms

---

### **Houndify Developer Platform**

Houndify [97], developed by SoundHound Inc., is a platform for developing speech-based applications.

According to developer's website:

- Understands text and voice input.
- Provides large-scale, fast and accurate speech recognition.
- Speech recognition and natural language understanding are performed simultaneously in real time, achieving very high speeds and accuracy.
- Understands queries with compound criteria and uses context to enable conversational interactions.
- Understands complex and compound queries.

It's possible to program in Android, iOS, C++, Web, Python, Java, C Sharp programming languages and runs on iOS, Android, Windows, Unix, Raspberry Pi, and others.

### ATOM Spoken Dialogue SDK

Atom SDK is a Software Development Kit (SDK) for interactive spoken dialogue systems development developed by Agilingua, LLC [98] and it's a proprietary system.

According to developer's website, the system has next features:

- Opened to any speech recogniser and TTS that provides APIs.
- The dialogue engine can communicate with external software components of various applications, virtually any application.
- Works with most major operating systems, including Windows Vista, 7 and 8, Windows Phone, Mac OS X, iOS, Android, Windows Mobile, Windows CE and QNX.
- Infrastructure is flexible, open and scalable, making it easy to develop, extend, update and maintain.
- Can be deployed embedded or in the cloud.

### Alexa Skills Kit

Alexa Skills Kit [99], is a toolkit for providing Alexa, the Amazon's speech interface, with new skills. It consists of a collection of self-service APIs, tools, documentation and code samples that make fast and easy adding skills to Alexa.

According to developer's website, for adding new skills to Alexa, it's necessary:

- An account on the developer portal to create a configuration for the new ability that includes the information the Alexa service needs to route requests to your service.
- The ability to develop and deploy a cloud-based service to an Internet-accessible endpoint. The service processes user intents and returns responses to the user.
- A development environment appropriate for the language to use. Possible languages are Node.js, Java or any language suitable for web services.

	Programming Language	OS	DM Approach	License
Olympus	C++	Windows	Agents	Open Source
Flipper MMS	Java	Windows	Information States	
IrisTK	Java	Windows	Finite State	Open Source
CSLU Toolkit	TCL/Tk, C	Windows	Finite State	Non Commercial
AT&T SDT	Python	Windows	Statistical	Non Commercial
Ariadne	C++, Java	Windows	Agents	Open Source
vAssist (ActiveMQ Based)	Java, C	UNIX-like	Agents	Open Source
Alex Dialogue Systems Framework	Python	UNIX-like	Statistical	Open Source
EDECAN	C	PC MultiOS		
OpenDial	Java	PC MultiOS	Statistical and Logical	Open Source
Trindikit	Prolog, Python	PC MultiOS	Information States	Open Source
DIPPER	C++, Java, Prolog	PC MultiOS	Information States	Open Source
Midiki	Java	PC MultiOS	Information States	Open Source
Jaspis	Java	PC MultiOS	Agents	Open Source
Daisie	Java	PC MultiOS	Agents and Information States	Open Source
Houndify	Android, iOS, C++, Web, Python, Java, C Sharp	MultiOS		
ATOM Spoken Dialog SDK		MultiOS		Proprietary
Alexa Skills Kit	JavaScript, Java, Others (Web Services Friendly)	MultiOS		Open Source

**Table 3.1:** Summary of SDS frameworks and toolkits.



# 4

## PRELIMINARY SYSTEMS

---

### 1 Introduction

---

This chapter shows the first systems developed by the author of this thesis. Section 2 presents a system aimed to give weather forecast information and news obtained from several digital publications. It was developed in collaboration with a company with the goal of having a prototype for a functional dialogue system that would serve as a sample of what these systems are capable of doing. Just the prototype was needed, and so no formal evaluation was carried out. The system was built under EDECAN framework [86], and the required components developed from scratch.

In Section 3 is presented an architecture the author developed with the idea of having a framework in which SDSs can be implemented easily. The architecture is focused on the development of applications in which users interact through portable devices such as smartphones and tablets. The section also presents a weather forecast retrieval application developed using this architecture. As in the previous system, no formal evaluation was carried out as it was just an initial approach for building a full framework for SDSs and, as will be shown, some aspects had to be improved.

As has already been said, one of the objectives of this thesis is that it serves as a guide to other developers for the implementation of dialogue systems. Therefore, details about installation and configuration of the different components involved in the systems are given. The non-interested reader can ignore them.

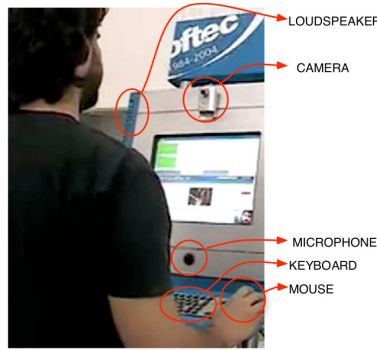
### 2 INMA

---

The *INMA (Conversational interface for e-administration)* system consists of a stand that allows getting current and past news published on the Internet by several newspapers and sites, and also to get information about the weather, initially of Spanish cities, although it can be easily extended to other cities around the world. The final goal is to provide with valuable information and entertainment to people queuing or just passing around. The system was developed along with the University of Zaragoza, under contract for the



Softec Ibermática enterprise and was funded by the Basque Government and GAITEK. Figure 4.1 shows an image of the system.



**Figure 4.1:** INMA: Final system.

This section describes the system architecture, detailing its configuration and giving development details for the different components that are part of the SDS.

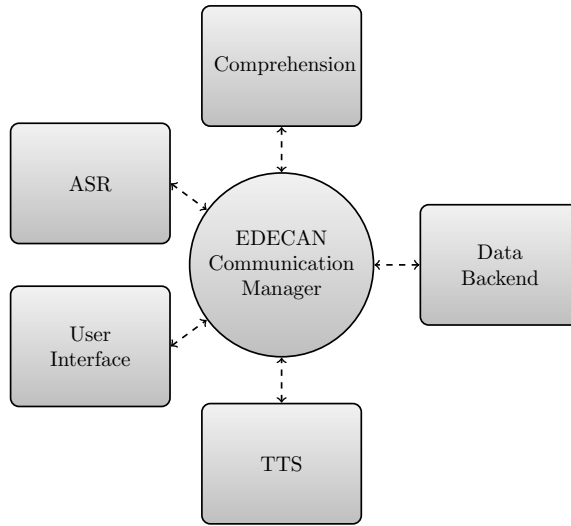
## 2.1 Architecture

The system was implemented under the EDECAN [86] framework. The architecture of the system is shown in Figure 4.2. As can be seen, it does not follow the usual architecture of the SDSs, that is, having TTS, ASR, Semantic Parser (SP), NLG and DM as independent modules. In this case, we have the TTS and ASR as independent modules, but the SP, NLG and DM are integrated into a single module which has been called *Comprehension*. Additionally, the system has a *Data Backend* module and a *User Interface* module. The *Data Backend* module is responsible for getting the information required by the user from the internet. On the other hand, the *User Interface* is responsible for showing the obtained information to the user. Additionally, this last module is responsible for detecting users' presence to init a dialogue. That is done via a face recogniser integrated on it.

The input and output commands (messages) for each of the modules are the ones shown in Table 4.1. The flow of these messages during the execution of the system is as follows:

When the *User Interface* detects the presence of a user via the face recogniser, it generates a *tts\_run* message with a welcome sentence embedded directed to the TTS. The TTS synthesises the sentence and sends an *asr\_run* message to activate the ASR. Once the ASR has got the recognition hypothesis from user's audio input sends it to the *Comprehension* module with an *asr\_end* message.

The *Comprehension* module analyses the received recognition hypothesis, and



**Figure 4.2:** Architecture of the INMA SDS System.

	INPUT	OUTPUT
ASR	asr_run	activate_voice_signal deactivate_voice_signal asr_end (utterance)
TTS	tts_run (text)	asr_run set_avatar_speak set_avatar_no_speak
Data Backend	action_news (newspaperID) action_meteo (cityID)	view_news (html_data) view_meteo (html_data)
Comprehension	asr_end (utterance)	reinit action_news (newspaperID) action_meteo (cityID) asr_run tts_run (text)
User Interface	view_news (html_data) view_meteo (html_data) activate_voice_signal deactivate_voice_signal reinit set_avatar_speak set_avatar_no_speak	tts_run (text)

**Table 4.1:** Set of commands (and arguments) for the different modules in INMA SDS.

if it determines that all the necessary information for a request is present (taking into account both the actual hypothesis and the dialogue history), it sends either an *action\_news* or an *action\_meteo* message to the *Data Backend* to obtain the requested information. Once the information has been obtained, the *Data Backend* module sends either a *view\_news* or a *view\_meteo* message that the *User Interface* receives to show the information to the user. Once the information has been showed to the user, the *User Interface* sends a *tts\_run* message to the TTS to ask the user if a new request is required, repeating the TTS, ASR and *Comprehension* path. At this point, if the user decides to finish the dialogue the *Comprehension* module will send a *reinit* message to the *User Interface* module to reinit all the components. In the other case, it will send an *asr\_run* to retrieve the new user's request.

On the other hand, at the point of the *Comprehension* module determining if all the necessary information for the user's request is available, if it determines that it isn't, a *tts\_run* message is sent to the TTS to ask the user for the missing information, repeating again the TTS, ASR and *Comprehension path*.

Finally, in addition, both the ASR and the TTS send other messages to the *User Interface* to modify the information presented to the user. The ASR sends *activate\_voice\_signal* and *deactivate\_voice\_signal* messages that are used to inform the user when the ASR is active. On the other hand, the TTS sends *set\_avatar\_speak* and *set\_avatar\_no\_speak* messages to activate or deactivate the motion of the avatar present in the user interface. An image of the user interface is shown in Figure 4.3.

## 2.2 Configuring and Launching the System

In order to make the system work it will be necessary, on the one hand, run the services corresponding to the different modules of the system, and on the other hand, run the communication manager. Here's how to do it.

### Services

Running the system services is done via the *super\_server* with the following command:

```
super_server ini increment end
```

where the parameters specify the listening ports range as follows:

- ini: Initial port
- increment: Port increment
- end: Final port

For our system only one listening port has been used:

```
super_server 6611 1 6611
```

The *super\_server* is responsible for launching the services. It reads the *config\_files/services/services.xml* file in which the configuration files corresponding to each of the services are listed, for our system the one shown in Listing 4.1. These files specify, on the one hand, the dynamic library containing the corresponding service implementation, and on the other hand, the input commands each service reacts to. An example of such a file is shown in Listing 4.2. Once the *super\_server* has read these files, it launches an instance of each of the services listed. Appendix B shows the configuration files for all the modules of the system.

Once the services have been launched the *super\_server* keeps waiting for connection requests from the communications manager. Once the connection between the communication manager and the *super\_server* is established, the message exchange can begin.

```
<?xml version="1.0"?>
<services>
  <tts>config_files/services/tts.xml</tts>
  <asr>config_files/services/asr.xml</asr>
  <ui>config_files/services/ui.xml</ui>
  <comprehension>config_files/services/comprehension.xml</comprehension>
  <databackend>config_files/services/databackend.xml</databackend>
</services>
```

**Listing 4.1:** INMA services configuration file.

```
<?xml version="1.0"?>
<service name="asr">
  <so>libASR.so</so>
  <command name="conexion">
    <function>entrada</function>
  </command>
  <command name="desconexion">
    <function>salida</function>
  </command>
  <command name="desconexion_caidagestor">
    <function>salida_caidagestor</function>
  </command>
  <command name="asr_run">
    <function>asr_run</function>
  </command>
</service>
```

**Listing 4.2:** INMA ASR EDECAN module configuration file.

## Communication Manager

The communication manager is launched with the following command:

```
comm_manager controller_port DNS_port DNS_ip config_file
```

The only parameter we are interested in is the last one that corresponds to a configuration file that will have to be provided to the communication manager.

For the rest of parameters, we will put any value since they correspond with elements of the EDECAN framework that we will not use.

So, for our system:

```
comm_manager 6633 6644 localhost config_files/router/manager.xml
```

Regarding the configuration file, it details such things as the different services that make up the system, its location, initialisation parameters to provide them when the connection is established or a table indicating possible message routing between services. Appendix B details the complete file used in the system, and Listing 4.3 shows a reduced version of this one on which the main components are explained.

As can be seen, this file consists of two main sections, a `<services>` section and a `<routing_table>` section. The `<services>` section specifies the different services that make up the system together with some configuration parameters. From these parameters, `port` and `address` are used to specify the location of the corresponding service and `configuration` is used to include parameters that will be provided to the corresponding service when establishing a connection with it.

On the other hand, the `routing_table` section is used to specify the possible paths for the messages. Looking, for example, at the controller section we see that it's composed of the name of the different services within the system which have associated a one or a zero. A one means that the controller can send messages to the corresponding module and a zero that can not. The EDECAN framework allows dynamic routing to be used, including the name of the destination service in the message structure, although in this system it has been chosen to use only static routing, which just will use the routing scheme defined in the configuration file.

## 2.3 Modules Implementation Details

### 2.3.1 Automatic Speech Recognition and Text to Speech

For the ASR, the speech recogniser developed by the group *Pattern Recognition and Speech Technology* of the Department of Electricity and Electronics of the Faculty of Science and Technology of the University of the Basque Country was used. It's based on the Viterbi algorithm for decoding and HMM for acoustic modelling. It was ported as an EDECAN module.

In addition to an `asr_end` output command emitted when the speech recognition process has finished, two output commands were added, `activate_voice_signal` and `deactivate_voice_signal`. The `activate_voice_command` is emitted just before the ASR starts the audio capture, and the `deactivate_voice_signal` just after the audio capture has finished. These commands are directed to the User Interface and are used to make the user know when to speak via an indicator included in the user interface.

```
<?xml version="1.0"?>
<manager>
  <services>
    <service name="asr">
      <port>6611</port>
      <address>localhost</address>
      <location>LAB</location>
      <configuration>NONE</configuration>
      <service_type>asr</service_type>
    </service>

    <service name="tts">
      <port>6611</port>
      <address>localhost</address>
      <location>LAB</location>
      <configuration>NONE</configuration>
      <service_type>tts</service_type>
    </service>

  <routing_table>
    <tts>
      <tts>0</tts>
      <asr>1</asr>
    </tts>

    <asr>
      <tts>0</tts>
      <asr>0</asr>
    </asr>
  </routing_table>
</manager>
```

**Listing 4.3:** Reduced configuration file for the communication manager in the INMA SDS System.

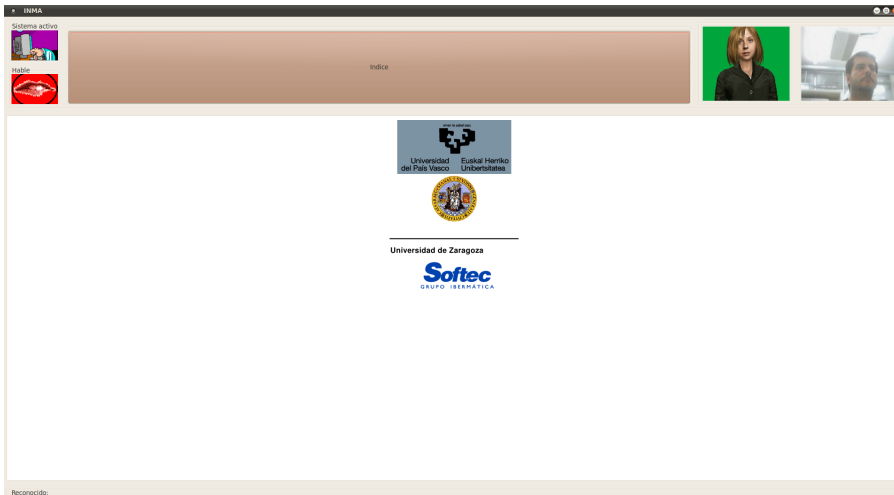
As for the TTS, Nuance’s Loquendo TTS [100] was used. It’s a multi-lingual TTS with a broad set of voices that also allows the creation of custom ones. The integration into the architecture was done using the API provided by the Loquendo software.

### 2.3.2 User Interface

The User Interface module is the responsible for showing the information requested to the user. Additionally, it also shows some extra information to make the system more user-friendly. Figure 4.3 shows a picture of the User Interface.

This extra information includes an animation to indicate to the user when the system is capturing audio and an avatar that is animated when the system is speaking.

Additionally, this module has a face recognition system. This feature is used



**Figure 4.3:** INMA: User Interface.

to detect users' presence so that once a user has been detected the interaction starts.

### 2.3.3 Data Backend

The purpose of this module is, once a user request has finished, retrieve the required data from the internet. It receives *action\_meteo* and *action\_news* messages depending on the type of the user's request.

For the case of receiving an *action\_meteo* message, the module will download the data from *The Weather Channel* web site. For that, it will extract the name of the city of which the user wants to know the weather from the received message, and with the city name will perform an HTTP GET request to obtain the identifying code needed for retrieving the data. Listing 4.4 shows and example of this request. This request returns, as a result, an XML file with all the possible matching cities and taking into account that the system was developed for Spanish cities, the module extracts the code for the city containing "Spain".

Once the city code has been obtained, the next step is to obtain the weather data. For that, again, an HTTP GET request is done. An example is shown in Listing 4.5. This request returns an XML file with the corresponding data. This file is then transformed by the module to an HTML text string using an XSLT template. Finally, the data are sent to the User Interface to be shown to the user via a *view\_meteo* message.

For the case of receiving an *action\_news* message, the module will extract the name of the newspaper required by the user from the message, and from a

```

REQUEST:
GET /weather/search/search?where=Madrid HTTP/1.1
Host: soap.weather.com

RESPONSE:
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<search ver="3.0">
  <loc id="SPXX0050" type="1">Madrid, M, Spain</loc>
  <loc id="USIA0518" type="1">Madrid, IA</loc>
  <loc id="USNE0299" type="1">Madrid, NE</loc>
  <loc id="USNY0847" type="1">Madrid, NY</loc>
  <loc id="CASK0766" type="1">Madrid, SK, Canada</loc>
  <loc id="RPXX0729" type="1">Madrid, SUR, Philippines</loc>
  <loc id="COXX8921" type="1">Madrid, CUN, Colombia</loc>
  <loc id="DRXX1034" type="1">Madrid, 06, Dominican Republic</loc>
  <loc id="EKXX1013" type="1">Madrid, LI, Equatorial Guinea</loc>
  <loc id="SWOT0583" type="1">Madrid, E, Sweden</loc>
</search>

```

**Listing 4.4:** INMA SDS: HTTP GET request for obtaining city codes.

```

REQUEST:
GET weather/local/SPXX0050?cc=*&unit=m HTTP/1.1
Host: soap.weather.com

RESPONSE:
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

XML_DATA

```

**Listing 4.5:** INMA SDS: HTTP GET request for obtaining weather data.



```
REQUEST:
GET /rss/elpais/portada.xml HTTP/1.1
Host: ep01.eping.net

RESPONSE:
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

XML_DATA
```

**Listing 4.6:** INMA SDS: HTTP GET request for obtaining newspaper data.

database will retrieve the *URL* where the news are stored for that newspaper on the Internet. Such database is a simple plain text file containing newspaper-url pairs, one per line. The URLs correspond with RSS feeds provided by the different newspapers.

Once the URL has been extracted, an HTTP GET request is used to get the data (example in Listing 4.6). These data, as in the previous case, are obtained as an XML file, so they are transformed into an HTML text string using an XSLT template. Finally, the HTML data are sent to the User Interface via a *view\_news* message.

#### 2.3.4 Comprehension

The primary purpose of this module is to extract the information from the user's input and act accordingly to it. It works, in some sense, as an SP and a DM at the same time.

The semantic parsing part is done by storing the information in frames. A frame consists of a *type* entry and a *param* entry. Listing 4.7 shows the structure of a frame and the possible values each field can take.

The module maintains two of these frames, one for the dialogue history and another for storing the last user interaction. When the module receives an input text from the ASR, it extracts the relevant information and fills the last user interaction frame. If this frame type is different to *TYPE\_YES* and *TYPE\_NO* the dialogue history frame is updated with the new information. At this point, if the history frame has all the necessary information to perform a request, the module will send a message to the TTS to request the user for confirmation of the data. On the other hand, in case not all the necessary information is present the module will send a message to the TTS to request the user the missing information.

When the last user interaction frame is of type *TYPE\_YES* a message is sent to the Data Backend module to obtain the data requested by the user, since

```
FRAME:
  type
  param

POSSIBLE TYPES:
  TYPE_NEWS
  TYPE_METEO
  TYPE_YES
  TYPE_NO

POSSIBLE PARAMS:
  PARAM_NEWSPAPER
  PARAM_CITY
```

**Listing 4.7:** INMA SDS: Comprehension module semantic parsing frame.

by the construction of the system this situation implies that the history frame is already complete. On the other hand, when the last user interaction frame is of type *TYPE\_NO* a *reinit* message is sent to the User Interface module to reinitialise the system for a next user. By the construction of the system this situation only occurs when some information has been shown to the user.

### 3 RTMIVA

---

This section defines the Real-Time Multimedia Interactive Voice Architecture (RTMIVA), an architecture designed with the idea of having a framework in which SDSs, and applications based on speech technologies in general, can be implemented easily.

The architecture is focused on the development of applications in which users interact through portable devices such as smartphones and tablets through VoIP protocol, although it could be possible to interact through any device provided that they have the appropriate client software installed.

The idea of the architecture is to simplify the process of creating new applications. So it has been designed in a way that the use of the different elements that make up an SDS is simplified and the development of the applications is done based on the utilization of a reduced set of commands and their respective parameters.

The rest of the section will focus on describing the architecture, explaining the required steps for its correct installation and developing a weather forecast retrieval application (similar to the one developed in Section 2) that in turn will serve as an example for the development of other applications.

### 3.1 Architecture Definition

---

The server side consists of an Asterisk [101] open source Private Branch Exchange (PBX) under the Linux operating system. This PBX will be configured so that on the one hand it accepts the calls of the different types of user, and on the other hand, allows the execution of various applications.

The applications have a call extension associated with each and are implemented using a simple language based on the use of XML tags. Each time the Asterisk server receives a call directed to an application gives up the control to an applications interface that will search for the corresponding XML for that application. If it does not exist it will be informed to the user, and the call will finish. In case such XML exists, the applications interface will interpret the commands included in it to execute the corresponding application. On the other hand, the applications interface, to provide the capabilities required by an SDS is connected to:

- TTS: To synthesise the answers that are given to the user.
- ASR: To recognise the spoken interactions of the users.

and plays the role of a DM maintaining the state of the dialogue with the user and managing the state changes.

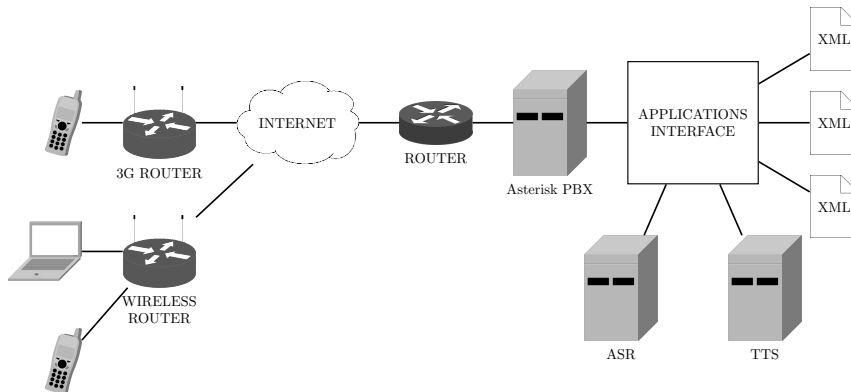
Clients wishing to connect to the applications provided by the platform may make Voice over IP (VoIP) calls to the server using any software or hardware that allows making VoIP calls based on any of the following protocols: SIP, IAX and H.323. There are many softphones (IP phones implemented by software), so no software of this type will be developed, and it will only be explained how to configure these softphones to interact with the platform. These softphones exist for a multitude of platforms like smartphones, PCs or PDAs so that through IP calls can interact with the platform any user who has a device of these with Internet connectivity. This connectivity can be any network that has an Internet connection, a Wi-Fi hotspot and even 3G.

For a better understanding of the architecture, everything mentioned is outlined in Figure 4.4.

### 3.2 Installing and Configuring the Architecture

---

This section explains the steps for the installation and configuration of all the required components for the correct operation of the architecture. In addition, when it's necessary, explanations on some concepts, whether they are technical or related to the operation of the architecture or some of its components, are given.



**Figure 4.4:** RTMIVA system architecture.

### 3.2.1 Installing Asterisk PBX

Asterisk is a free software application that provides PBX functionality. Like any PBX, it accepts connections from a certain number of devices to make calls between them and even connections from VoIP providers and Integrated Services Digital Networks (ISDNs). Asterisk includes many features previously only available in expensive proprietary PBX systems such as voicemail, conferencing, automatic call distribution, and much more. Developers can create new features by writing a dialplan in the Asterisk scripting language, or in other languages by adding the corresponding modules. To connect analogue standard phones, Foreign eXchange Subscriber (FXS) or Foreign eXchange Office (FXO) telephone cards are necessary, since a simple modem is not enough to connect the server to an external line. An interesting aspect of Asterisk is that it supports many VoIP protocols such as Session Initial Protocol (SIP), InterAsterisk eXchange protocol (IAX) and H.323. Asterisk can interoperate with Internet Protocol (IP) terminals acting as a registrar and as a gateway between them.

Before installing the Asterisk software, some libraries and kernel updates must be installed, to avoid having problems installing the required components, as follows:

```
apt-get install -qy bison flex libncurses5-dev zlib1g-dev libssl-dev libnewt-dev sox
```

Now we proceed to install all the required components.

#### Installing Asterisk

To install Asterisk we will run the following commands:

```
cd /usr/src
wget http://downloads.digium.com/pub/asterisk/releases/asterisk-1.4.17.tar.gz
tar xvfz asterisk-1.4.17.tar.gz
cd asterisk-1.4.17
./configure
make
make install
make samples
```

### Installing Asterisk Addons

Asterisk Addons is a package providing extra functionalities to Asterisk, such as:

- The possibility of having a call log in a MySQL database.
- Using MP3 files for music on hold.
- Adding the H.323 protocol.

To install it:

```
cd /usr/src
wget http://downloads.digium.com/pub/asterisk/releases/asterisk-addons-1.4.5.tar.gz
tar xvfz asterisk-addons-1.4.5.tar.gz
cd asterisk-addons-1.4.5
./configure
make
make install
make samples
```

### Installing Voices to Asterisk

This project was focused on Spanish Castillian speaking users, so we proceed to install Spanish Castillian voices to Asterisk. Asterisk's unofficial Castillian voices, licensed by Alberto Sagredo Castro and obtained using professional speakers, will be installed.

In the first place, the necessary voices will be downloaded, they will be downloaded with u-law format since they are of high quality:

```
cd /usr/src
mkdir voices
cd /voices
wget http://www.voipnovatos.es/voices/voipnovatos-core-sounds-es-ulaw-1.4.tar.gz
wget http://www.voipnovatos.es/voices/voipnovatos-extra-sounds-es-ulaw-1.4.tar.gz
```

Two voice packs have been downloaded that complement each other. It's necessary to have both so that everything is played in Spanish Castillian. In case there is no voice file when playing a piece of the dialplan in Asterisk, the corresponding English voice will be used.

Once the packages have been downloaded we decompress them:

```
tar zxvf voipnovatos-core-sounds-es-ulaw-1.4.tar.gz
tar zxvf voipnovatos-extra-sounds-es-ulaw-1.4.tar.gz
```

When decompressing these files the structure of the created directories is of type `/<directory>/es`, which does not correspond to the predefined structure in Asterisk 1.4, which is `/es/<directory>`. Therefore we must execute the following commands:

```
mkdir /var/lib/asterisk/sounds/es/dictate
mkdir /var/lib/asterisk/sounds/es/letters
mkdir /var/lib/asterisk/sounds/es/silence
mkdir /var/lib/asterisk/sounds/es/followme
mkdir /var/lib/asterisk/sounds/es/phonetic
mkdir /var/lib/asterisk/sounds/es/digits

cd /usr/src

mv voices/es/* /var/lib/asterisk/sounds/es
mv voices/digits/es/* /var/lib/asterisk/sounds/es/digits
mv voices/dictate/es/* /var/lib/asterisk/sounds/es/dictate
mv voices/letters/es/* /var/lib/asterisk/sounds/es/letters
mv voices/silence/es/* /var/lib/asterisk/sounds/es/silence
mv voices/followme/es/* /var/lib/asterisk/sounds/es/followme
mv voices/phonetic/es/* /var/lib/asterisk/sounds/es/phonetic
```

Now the voices are in the right place, just need to tell Asterisk to use them. As will be seen later, it's done using a `language=es` label in the Asterisk user's configuration file.

### Launching Asterisk at Boot Time

If we want Asterisk to start every time the computer is turned on we need to execute the following commands:

```
cd /etc/init.d
cp /usr/src/asterisk-1.4.17/contrib/init.d/rc.debian.asterisk asterisk
update-rc d asterisk defaults
```

#### 3.2.2 Installing no-ip

To establish the connection between a client device and the Asterisk PBX, the client configuration must specify the IP address where the server is located. In the case that both the client device and the server are on the same network, there will not be any problem since the server can be configured so that it has a static IP address and this way always have the same IP address. So, once the clients have been configured, they would not have to be reconfigured again.

The problem arises when the client devices want to access from outside the network in which the server with the Asterisk PBX is located, that is, when the

clients making the IP calls and the server are not on the same local network. Nowadays, Internet providers allocate the majority of public IP addresses by Dynamic Host Configuration Protocol (DHCP), so these addresses are dynamic and change over time. Therefore it's not comfortable to configure the clients with the public IP address of the server since this will change from time to time and would have to reconfigure the clients with the new IP address.

To solve the problem, we will register a Domain Name Server (DNS) name. This way each time a client tries to connect to the registered DNS name will be redirected to the machine we will indicate, that is, the one with the Asterisk PBX.

### Registering a DNS Name

To register a DNS name in the first place will access the web page <http://www.noip.com> and proceed to register as a user. In addition to the usual registration data (user, password and e-mail address), the DNS name we wish to register must be provided at the time of registration. Once the registration has finished, the DNS name will be available to use. In our case and as an example we have chosen the DNS name *rtmiva.hopto.org*, so that every time a client tries to connect to this hostname will be redirected to the PC that we indicate. The way to do the last is explained below.

### Installing no-ip Software

Once the DNS name has been registered, we must install the no-ip software in the server we want to be located using the registered DNS name. This software will be responsible for updating the IP address associated with the DNS name. While this software is running on the server, each client that wants to connect through the DNS name will be redirected to it. In case this software is not running the DNS name will probably not be translated to the current IP address of the server since a possible change in the public IP address will not be updated.

Therefore, for the server with the Asterisk PBX to be accessible from an external network using the DNS name, the no-ip software must be installed. To do this we download and install the software with the following commands:

```
cd /usr/src
wget https://www.noip.com/client/linux/noip-duc-linux.tar.gz
tar xvfz noip-duc-linux.tar.gz
cd noip-2.1.9-1
make
make install
```

During the installation process, the username and password with which the DNS name was registered are prompted, we introduce them and proceed with the installation.

As mentioned above, this software is responsible for updating the value of the

IP address associated with the registered DNS name. Therefore the next step will be to specify, in minutes, the IP address updating interval. In our system, it has been selected a 5 minutes updating interval so that if an IP change occurs, the server, at most, will be inaccessible for five minutes

Next, we must specify which of the registered machines we want to update the DNS record. In this case, we only have one, so the required field will be left empty as the default option is to update all the registered machines.

Once the installation is complete, the updating process will start automatically and the server machine with the Asterisk PBX will be accessible via the DNS name.

To verify that it works correctly, from another machine we could execute a *ping* command and see how, effectively, the packages arrive at their destination.

### 3.2.3 Installing PHPAGI

Asterisk Gateway Interface (AGI) [102] is an extension of Asterisk that allows the execution of programs and scripts external to the own Asterisk platform and its integration within the dialplan of Asterisk. It supports external programs written in various programming languages such as: Java, Perl, PHP, Python, Ruby, C, .NET, Haskell, ...

For our architecture, it has been decided to use the PHP programming language to develop some of the modules. This way, thanks to the features of PHP, we can use the AGI as an interface between Asterisk and the Linux system command line, and thus obtain the interoperation between Asterisk and the external modules necessary to make the architecture work (ASR, TTS, ...).

For the installation, we execute the following commands:

```
cd /usr/src
wget http://sourceforge.net/projects/phpagi/files/latest/ -O phpagi.tgz
tar xvfz phpagi.tgz
mkdir /var/lib/asterisk/agi-bin/phpagi
cp -r /usr/src/phpagi-<version>/* /var/lib/asterisk/agi-bin/phpagi/
```

### 3.2.4 Installing TTS

For the TTS, Loquendo TTS has been used, although it could be possible to use any other as long as it's integrated into the platform appropriately. This section will explain the installation and configuration process of the Loquendo TTS, although following similar steps any other could be integrated.

The first step is the installation of the software itself. The Loquendo software is composed of the synthesis engine plus a set of voices. In this system, *Jorge's*



voice was used. For the installation of both the engine and the voices, the following commands must be executed:

```
cd Loquendo/Loquendo_TTS.6.5-SDK_Distribution.6.5.4/Linux
chmod +x install.sh
./install.sh

cd Loquendo/Loquendo_TTS.6.5-Jorge.6.5.4/Linux
chmod +x install.sh
./install.sh
```

If nothing is specified during the installation process, the software will be installed in the `/opt/Loquendo/LTTS` directory.

Once the software has been installed, it will be necessary to activate the license. The license file will be placed in the `/opt/Loquendo/LTTS` directory, and the `default.session` file edited to add the following line in case it doesn't exist:

```
"LicenseFile"="/opt/Loquendo/LTTS/TTSLicense.txt
```

On the other hand, to integrate the TTS in the platform, a small application has been developed that is in charge of making the synthesis itself, and that will be invoked when necessary. The code for this application can be found in the C appendix. This application will receive three parameters:

- The name of the voice to use. In this system, *Jorge*.
- The path where the synthesised audio will be stored.
- The text to synthesise.

### 3.2.5 Installing ASR

For the ASR, the speech recogniser developed by the group *Pattern Recognition and Speech Technology* of the Department of Electricity and Electronics of the Faculty of Science and Technology of the University of the Basque Country has been used. However, as in the case of the TTS, any other may be employed provided that it's properly integrated.

The integration of the ASR into the platform consists of two parts. On the one hand, a Language Model (LM) generator that will be used to generate LMs needed by the decoder, which will be generated on the fly and will be dependent on the state in which the dialogue between the system and the user is. And on the other hand the decoder.

#### Language Model Generator

As has been said depending on the state in which the dialogue between the system and the user is, a LM or other will be used. To generate these LMs on the fly, a software has been developed that will be invoked before the recognition process. This software receives as parameters a text file with the

different training phrases and a directory where the file will be saved with the created LM.

```
./create_lexicon lexicon_tmp.txt lexicon.txt
```

The training phrases file will consist of one sentence without spaces per line, instead of spaces the character "\_" is used.

As an example for greetings LM the file would look like:

```
Hello
Good_morning
Good_evening
Good_night
```

### Decoder

The installation of the decoder has been divided into two parts. On the one hand the parametrisation of the audio and on the other hand the decoding.

In the process of voice recognition once the audio has been obtained it's parametrised to obtain the relevant information and this parametrisation is then sent to the decoder, which together with the LM previously generated will give a recognised sentence. To integrate these two steps, parametrisation and decoding, two applications were previously available which were integrated into the platform. To run them is done as follows:

```
./param08 audio.wav params.bin
./recogniser params.bin
```

It must be taken into account that for the parametrisation models for audio sampled at 8KHz have been used.

## 3.2.6 Configuring Asterisk PBX

### 3.2.6.1 Creating Users

This section shows how to set up the user accounts that will be used by the users of the system. Users can connect via several protocols that implement VoIP technology, here are instructions for the most common.

#### SIP Protocol

SIP protocol was developed by Internet Engineering Task Force (IETF), with the idea of creating a signalling and control architecture for VoIP. The primary goal is to communicate multimedia devices and this communication is made using Real-time Transport Protocol (RTP) and Session Description Protocol (SDP) protocols. RTP protocol is used for voice data transmission in real time and SDP protocol for signalling.

SIP is an end-to-end signalling protocol that implies that all logic is stored in the end devices (except for routing SIP messages). In a SIP architecture there are two fundamental elements:

- User Agent (UA): Divided into User Agent Client (UAC) and User Agent Server (UAS). Both are found in all UA, thus allowing communication between different UA through client-server type communications.
  - UAC: Is a logical entity that generates SIP requests and receives responses to its requests.
  - UAS: Logical entity that generates responses to the SIP requests received.
  
- SIP Server: There are three types of servers:
  - Proxy Server: Retransmits requests and decides to which other servers forward, altering the fields of the request if necessary. It's an intermediate entity that acts as client and server for the purpose of establishing calls between users.
  - Registrar Server: Accepts user registration requests and saves the information of these requests to provide an address localisation and translation service in the controlling domain.
  - Redirection Server: Generates redirection responses to the requests it receives. This server forwards the requests to the next server.

On the Asterisk PBX, the *sip.conf* file is used to configure everything related to the SIP protocol and to add new users or connect to SIP providers. In this file, below the *[general]* tag, default settings for all users and peers are defined. Default values should be edited so they will be left as follows:

```
[general]
context=default
port=5060 ; UDP port for Asterisk responses
bindaddr=0.0.0.0 ; Specifies an Asterisk IP (eg, a computer has 3 IPs), 0.0.0.0 is valid for
  anyone
srvlookup=yes ; Enables DNS SRV server for calls outside the network
localnet=192.168.1.0/255.255.255.0 ; Specifies local IP. Required for NAT
externhost=rtmiva.hopto.org ; DNS address of the network where the PBX server is
  located. Required for NAT. If configuring the system for a local net only, not
  necessary.
```

On the other hand, for each new user to be added, its configuration will be included in the file with the following format:

```
[USERNAME] ; User's Nickname
language=es ; Language to be used by the system
type=friend ; Used for authentication purposes
secret=123456 ; Password used for user's authentication
nat=yes ; Forces Asterisk to ignore the contact information field and uses the address
           from which packets come
canreinvite=no ; Forces Asterisk to be in the middle, not allowing endpoints to exchange
                RTP messages directly
qualify=yes ; Every 30 sec sends a ping to the calling user
context=contextname ; Name of a context (defined in extensions.conf file)
host=dynamic ; IP address of user's device. "dynamic" allows all addresses
```

In the case of making the SIP calls between users of the same local network, where the Asterisk PBX is connected, no extra configuration is required.

But, on the other hand, if the calls are intended to be made from outside the network there will be problems with Network Address Translation (NAT) since SIP was not designed with that protocol in mind. NAT is a mechanism used by IP routers to exchange packets between two networks that mutually assign incompatible addresses (private addresses). It consists in converting in real time the addresses used in the transported packages. There are private address ranges that can be freely allocated to devices on a private network. This way all devices with different private addresses can go to the Internet with a single public address or a reduced set of them.

As already said, SIP was not designed to be used with NAT routers, by default SIP just sends signalling information, and audio transmission is done via RTP protocol. Signalling usually uses User Datagram Protocol (UDP) port 5060, and RTP uses random UDP ports for audio input and output which makes it necessary to have two additional UDP ports, one for audio input and another for audio output.

One of the problems that arises with NAT routers is that it's not known to which device in the internal network redirect incoming SIP calls (signalling information). To solve the problem on the NAT router, it will be necessary to specify that the SIP traffic received is redirected to the desired device. As an example, let's suppose that the IP address of the NAT router is *192.168.1.0/24*, and that the IP address of the server hosting the Asterisk PBX is *192.168.1.38*. In this case, it must be specified that all traffic of SIP calls arriving at the NAT device (traffic arriving at UDP port 5060) is redirected to the address *192.168.1.38*. Also, it will be necessary to verify that the SIP users included in the *sip.conf* file of Asterisk have the options *nat=yes* and *qualify=yes*.

The other problem has to do with audio routing. Most NAT routers are unable to associate signalling protocol packets with audio packets and are often unable to determine where to send audio. At this point when a call is made, everything seems to be normal, the call is set, but the audio does not flow in either direction. As discussed earlier, in addition to using UDP port 5060 for signalling, two additional UDP ports per call are required for RTP

audio transmission. So, we must specify these ports.

In Asterisk the range of UDP ports to use for RTP are defined in the *rtp.conf* file. By default, this range is defined as UDP ports from 10000 to 20000. This range allows 5000 simultaneous calls to be made. Although this range can be changed we decided to left it as is. On the other hand, once the range of ports has been decided it will be necessary to configure the NAT router to redirect all incoming traffic from UDP ports 10000-20000 to the server with *192.168.1.38* IP address.

Finally, clients wishing to connect to the system from outside the network should make similar configurations in their NAT routers, opening the same ports for both signalling and audio transmission.

### IAX Protocol

IAX protocol was designed as a VoIP connection protocol between Asterisk servers, although nowadays is also used in other servers apart from Asterisk. The version currently used is IAX2 as the first version has become obsolete. It's a protocol designed and intended for use in VoIP although it can support other types of connections such as video.

The primary objectives of the IAX protocol are:

- Minimize the bandwidth used in VoIP transmissions. IAX protocol is a binary protocol instead of text protocol (such as SIP) for messages to occupy less bandwidth.
- Avoid NAT problems (avoiding problems such as SIP protocol). IAX protocol transmits signalling and audio through a single port, UDP port 4569.
- Add support to transmit dialplans.

For the creation of users intended to use the IAX protocol, it will be done similarly to SIP clients but in this case in the *iax.conf* file.

For each new user some lines must be added to the file with the following format:

```
[USERNAME] ; User's nickname
language=es ; Language to be used by the system
type=friend ; Used for authentication purposes
secret=123456 ; Password used for user's authentication
context=contextname ; Name of a context (defined in extensions.conf file)
host=dynamic ; IP address of user's device. "dynamic" allows all addresses
```

To carry out IAX calls without problems from outside the network if the PBX is behind a device that implements NAT, UDP port 4569 will have to be opened. This will require accessing the router and in the NAT tables specify that the information that arrives at UDP port 4569 will be redirected to the same port but to the server machine where Asterisk PBX is.

At the other end, the calling client does not need to open any ports.

### H.323 Protocol

H.323 was designed to provide users with teleconferences that have voice, video and data capabilities over packet switching networks. The standard was designed with the following objectives:

- Rely on existing standards, including H.320, RTP and Q931.
- Incorporate some of the advantages that packet switching networks offer to transport data in real-time.
- To solve the problems that pose the sending of data in real-time on packet switching networks.

The H.323 designers defined this protocol in such a way that the companies that manufactured the equipment could add their specifications to the protocol and could define other structures that would allow the devices to acquire new characteristics or capabilities. H.323 sets the standards for audio and video compression and decompression, ensuring that devices from different manufacturers are intercommunicated. Also, the signalling procedures of the logical channels contained in the H.245 standard are used, which allow the content of each channel to be defined when it's opened. As the capabilities of the systems are determined, data transmission is not allowed if the receiver can not manage the call.

To use the H.323 protocol in the Asterisk PBX, at first, the following commands must be executed to install some dependencies:

```
apt-get install flex bison
touch /usr/include/linux/compiler.h
```

On the other hand, to configure the H.323 protocol next libraries must be installed:

- PWLIB
- OPENH323
- Chan\_h323.so for Asterisk.

To install PWLIB:

```
cd /usr/src
wget https://sourceforge.net/projects/openh323/files/pwlib/1.10.3/pwlib-v1.10.3-src-
tar.gz
tar zxvf pwlib-v1.10.3-src-tar.gz
cd pwlib_v1.10.3/
./configure
make
make install
make opt
PWLIBDIR=/usr/src/pwlib_v1.10.3
export PWLIBDIR
```

To install OPENH323:

```
cd /usr/src
wget https://sourceforge.net/projects/openh323/files/openh323/1.18.0/openh323-
v1.18.0-src-tar.gz
tar zxvf openh323-v1.18.0-src-tar.gz
cd openh323_v1.18.0/
./configure
make && make opt && make install
OPENH323DIR=/usr/src/openh323_v1.18.0/
export OPENH323DIR
```

Finally, to install chan\_h323.so:

```
cd /usr/src/asterisk-1.4.17
./configure
cd channels/h323/
make
make opt
cd /usr/src/asterisk1-4-17
./configure
make
make install
echo "/usr/local/lib" >> /etc/ld.so.conf
ldconfig
```

To configure the users the *h323.conf* file, located in */opt/asterisk* directory, must be edited. The first step is to define the general characteristics of the protocol:

```
[general]
port=1720 ; Working port
bindaddr=0.0.0.0 ; Incoming calls from any IP
allow=all ; Allow the use of all CODECs
gatekeeper=DISABLE ; Gatekeeper not used
```

For each new user to be added, its configuration will be included in the file with the following format:

```
[USERNAME]
type=friend ; Can make and receive calls
language=es ; Language for messages
host=192.168.1.34 ; IP address of user's device
context=contextname ; Name of a context (defined in extensions.conf file)
```

It must be said that for H.323 protocol the only authentication credential is the user's device IP address, so that devices must always have the same IP address, so they belong to the same user.

Finally, once the Asterisk is started, the module must be loaded as follows:

```
asterisk -rvvvv
module load chan_h323.so
```

### 3.2.6.2 Configuring Extensions

The *extensions.conf* file located in */etc/asterisk* directory is one of the most important in Asterisk since it defines the dialplan or numbering plan that will be followed by the PBX for each context. In this file will be defined for each context which resources can be accessed and by which numbering.

A dialplan can be understood as the treatment a call receives. Each phone extension of the PBX has a dialplan associated. Using a dialplan, each time a user makes a call to an extension, the associated actions are executed (making a call to another user, redirecting to a voicemail, ...).

The file *extensions.conf* is composed of contexts between brackets "[ ]". On the one hand, we have the *[general]* and *[globals]* contexts, which are always present.

In the *[general]* context the following options are configured:

```
[general]
static=yes ; In case a command "save dialplan" is executed in console, it will execute.
writeprotect=no ; If writeprotect=no and static=yes the "save dialplan" command can
                be executed in console.
autofallthrough=yes ; If an extension runs out of things to do, it will terminate the call
                    with BUSY, CONGESTION o HANGUP.
clearglobalvars=no ; Global variables are not freed when extensions are reloaded or
                    Asterisk is restarted.
```

The *[globals]* context is used to define global variables that can be used in all other contexts. The default setting will be left.

Once the common contexts have been configured we must configure a particular context for our platform, as an example, we call it *[contextname]*. When creating the users in section 3.2.6.1, *context=contextname* was defined in the configuration, meaning that those users can only execute the dialplan defined for this context. For this context, we will define the dialplan shown in Table 4.2 so that when a user makes a call to the desired extension, the associated action will be executed.

To create the dialplan in Table 4.2 the following configuration will be added in *extensions.conf*:

```
[contextname]

exten => 600,1,Playback(demo-echotest)
exten => 600,2,Echo
exten => 600,3,Playback(demo-echodone)
exten => 600,4,Hangup

exten => 601,1,Playback(Apologize)
exten => 601,2,Hangup

;SIP Calls
exten => 1001,1,Dial(SIP/USERNAME)
exten => 1001,2,Hangup
```



```

;IAX Calls
exten => 2001,1,Dial(IAX2/USERNAME)
exten => 2001,2,Hangup

;H.323 Calls
exten => 3001,1,Dial(H323/USERNAME)
exten => 3001,2,Hangup

```

Extension	Action
600	Latency and echo check.
601	One of the internal recordings is played, to verify that it is listened correctly. To test whether users behind a NAT device can hear the audio from the PBX. If they do not hear it, they should proceed to open the appropriate ports.
1001	SIP call to USERNAME.
2001	IAX call to USERNAME.
3001	H.323 call to USERNAME.

**Table 4.2:** RT-MIVA dialplan.

We have used as an example a user called *USERNAME*, when adding new users it should be done in the same way but substituting *USERNAME* with the desired usernames.

### 3.2.7 Applications Interface

The applications interface is a fundamental part of the system since it's the one that will be in charge of reading the application files and execute the instructions contained in them, besides it's the one in charge of interconnecting the different components of the system, that is, the Asterisk PBX, ASR and TTS. On the other hand, it also acts as a DM since it's the one that stores the state of execution of the application and manages the state changes. It consists of a file written in PHP language that interacts with Asterisk using the previously installed PHP AGI. This file should be located in */var/lib/asterisk/agi-bin/Central.php*.

To have an in-depth knowledge of how the applications interface works, it's recommended to look at the code in Appendix C. This section will give an overview of its operation. Broadly speaking the operation of the applications interface is as follows. When the Asterisk PBX receives a call, the call is redirected towards the applications interface which will check if the corresponding XML exists for the application associated with the call extension. If it does not exist, the user will be informed of the non-existence of the application,

and the call will be finished. That is done with the following instructions:

```
$agi->answer();
...
...
$utt="Hello $argv[2], extension $argv[1] does not have an associated application. Try
again with a different extension number.";
exec("sudo /opt/Loquendo/LTTS/./txt2audio Jorge /var/lib/asterisk/sounds/es/temp.
wav $utt", $output);
$agi->stream_file('temp');
...
...
$agi->hangup();
```

If it exists, the applications interface will load the XML file associated with the application and begin to interpret the commands included in the file. There are three predefined commands in the system:

- Synthesise
- Recognize
- Check

If the loaded command is the *synthesise* command, the audio of the text passed as a parameter will be synthesised and sent to the client:

```
$utt="$text";
exec("sudo /opt/Loquendo/LTTS/./txt2audio Jorge /var/lib/asterisk/sounds/es/temp.
wav $utt", $output);
$agi->stream_file('temp');
```

If the command to execute is *recognise*, first an audio will be recorded with the client's request, which then will be parametrised.

```
$time="" .(((int)$command[seg])*1000);
$nofich="$year$month$day$hour$min$sec";
$agi->record_file('temp/$nofich', 'wav', '#', $time, '0', '0', '2');
exec("../bin/param08 /var/lib/asterisk/sounds/es/temp/$nofich.wav ../temp/param/
$nofich.xml", $output);
```

Then, if there is no LM associated with the state in which the dialogue is located within the system cache, one is generated. Once the parametrised audio and the LM are available, the recognition result is obtained.

```
$fi=explode(".", $xml);
If(!file_exists("../temp/lexicon/" . $fi[0] . ".txt"))
{
exec("php ../bin/xmllexicon.php $xml");
exec("php ../bin/create_lexicon ../temp/lexicon/" . $fi[0] . "_tmp.txt");
exec("rm ../temp/lexicon/" . $fi[0] . "_tmp.txt");
}
copy("../temp/lexicon/" . $fi[0] . ".txt ../data/lexicon.txt");
exec("../bin/recogniser temp/params/$nofich.xml", $output);
```

If the command is *check*, the behaviour is very similar to the command *recognise*. The audio with the client's request will be recorded in the same way but, in this case, a predefined LM of checking sentences will be used. Then, the recognised text will be used to decide whether the result of the check has been positive or negative.

On the other hand, it's also possible to define custom commands for use in the applications. As an example, we can define *synthesisweb*, which synthesises a text contained in a URL. In this case, the synthesis will be carried out in the same way as for the command *synthesise*, but the text to be synthesised will be in a URL that is passed as a parameter to the command.

```
$utt=file_get_contents("$url");
exec("sudo /opt/Loquendo/LTTS/./txt2audio Jorge /var/lib/asterisk/sounds/es/temp.
wav $utt",$output);
$agi->stream_file('temp');
```

Using these commands, the applications interface will execute all instructions included in the application definition XML file until the end, at which point the application will be terminated and the call finished.

### 3.2.8 Installing and Configuring the Clients

Once all the necessary components to run the server have been installed and configured it will be necessary to configure the clients that will then connect to it. For this, it will be necessary to install a VoIP client software on them and later configure it. There are plenty of software of this type for all types of platforms. Examples include Ekiga and Empathy for desktop computers or LinPhone and 3CXPhone for smartphones. We will not give detailed instructions to configure each of them, but some general instructions on how to configure them depending on the type of protocol that is intended to be used will be given.

#### 3.2.8.1 Clients Using SIP and IAX Protocols

Clients that want to connect to the platform using SIP and IAX protocols must create an account of the type of the chosen protocol and configure it accordingly.

To configure the account, it's necessary to assign a username and password. These must match those that were specified in the *sip.conf* file of the Asterisk PBX. As an example, and to be consistent with the configuration previously used these will be, USERNAME and xxxxxx respectively.

In addition to these data, the IP address or DNS name of the domain where the server with Asterisk PBK is installed must be specified. In case the Asterisk server and the client are on the same local network, the local address of the Asterisk server will be used. If they were on a different network, the previously

registered DNS name would be used. In case the software requires us to enter a port number for the connection to the server, UDP port 5060 will be entered for the SIP protocol and UDP 4569 for the IAX protocol.

Another interesting configuration to do is to choose the audio CODEC to use. To select an appropriate CODEC, it's necessary to take into account the bandwidth that each one occupies against the quality that it offers. Asterisk supports the following audio compression CODECS, so one of these must be chosen:

- G.711 u-law (used in USA) – (64kbps)
- G.711 a-law (used in Europe and Brazil) – (64kbps)
- G.723.1 – Pass-through Mode
- G.726 – 16/24/32/40kbps
- G.729 – Requires license – (8kbps)
- GSM – (12-13kbps)
- iLBC – (15kbps)
- LPC10 – (2,5kbps)
- Speex – (2,15-44,2kbps)

### 3.2.8.2 Clients Using H.323 Protocol

In the case of using the H.323 protocol first an account of this type will be created. Then, for the IP address of the server to which we want to connect, we will use, as in the previous case, the local address of the Asterisk server in case both the server and the client are on the same network, or the previously registered DNS name in case of being in different networks.

No additional settings are required. Remember that H.323 protocol authenticates the client based on its IP address.

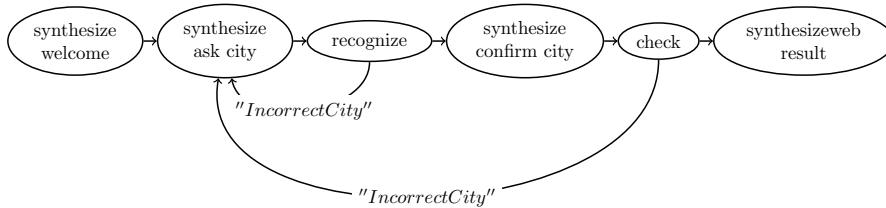
## 3.3 Weather Forecast Retrieval Application

Up to this point, the installation and configuration of the platform to support the different applications has been explained. In this section, we proceed to develop an example application that consists of an application to obtain the weather forecast for Spanish cities. The development will also serve as a guide for the creation of other types of applications.

To develop the applications, a language based on XML tags has been created. The RTMIVA platform will run the applications interpreting the corresponding XML file. Each time a call is made to the extension associated with an application, the applications interface will execute the instructions in the XML file associated to run the application. This section describes how to write these XML files.

### 3.3.1 Application File Definition

To write an application, the first thing to do is to define its operation. Applications are defined based on a set of states that have commands associated with them. These states and their corresponding commands are: *recognise*, *synthesise*, *check* and *synthesiseweb*. Later on, we will see their functionality and mode of use. Using these states for this example application, we have designed the operation shown in Figure 4.5.



**Figure 4.5:** Structure of the dialogues for the meteorological information retrieval system with RTMIVA architecture.

On the other hand, since this is a system of interaction through voice, we will have to define the type of sentences the system can recognise, this is, a LM. Each of the states in which the *recognise* command is involved has an associated LM. In this case, only a state of this type is used, and so only one LM has been defined. That is shown in Figure 4.6. In case of creating applications with more *recognise* states, you will have to define their corresponding LMs. Later we will see how these LMs are created. In addition to the *recognise* states, *check* states also have an associated LM, these states are used to ask the user if what the system has been understanding is correct and use a predefined LM consisting of the following words:

```

Eso_es
En_efecto
De_acuerdo
Adelante
Si
Claro
Vale
Perfecto
No
Te_repito
Has_reconocido_mal
Has_entendido_mal
Empieza_otra_vez
  
```

Once we have defined the application's behaviour and its LM we proceed to explain how to write the application's XML file. The final XML file is given in Appendix C.

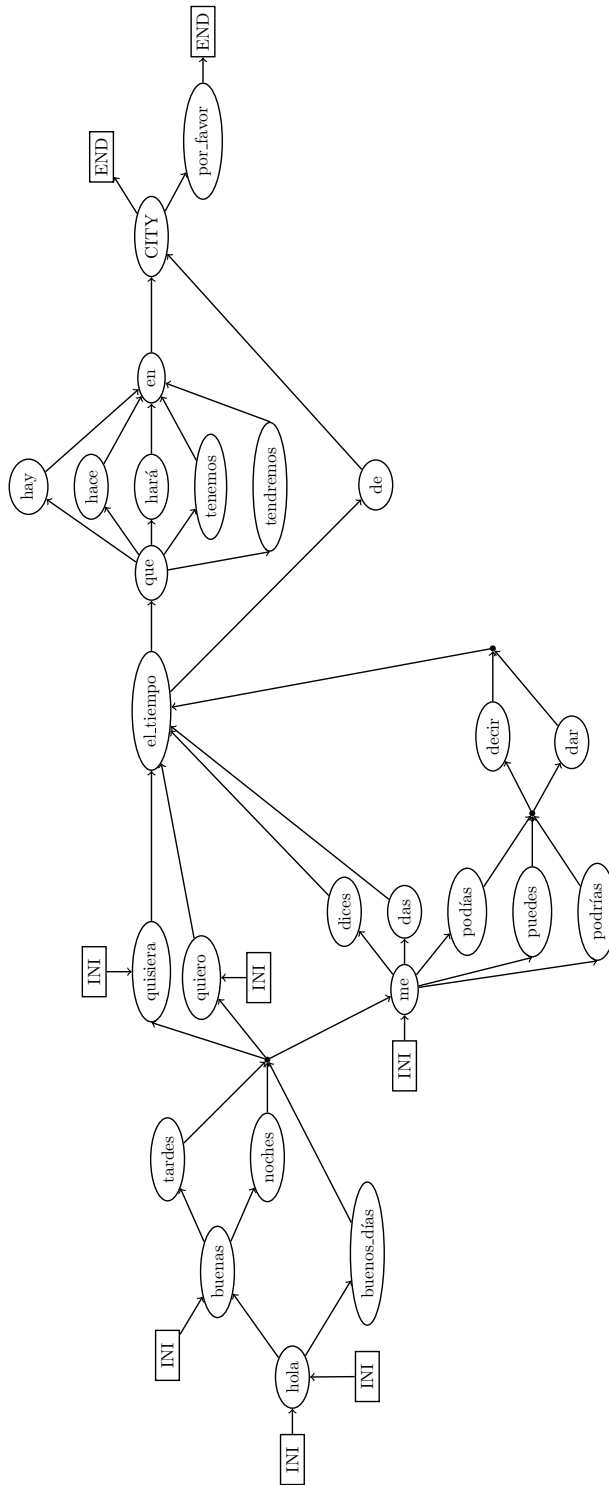


Figure 4.6: Language Model for the Weather Forecast retrieval application in RTMIVA.

The template for the definition of the application is as shown below:

```
<rtmiva>
  <name>Weather Forecast</name>
  <description>Application to retrieve weather forecast.</description>
  <extension>1</extension>
  <application>
    ...
    ...
    ...
  </application>
</rtmiva>
```

where:

- name: The name of the application. For this example *Weather Forecast*.
- description: A short description of the application.
- extension: Application's extension number. Every time a user dials this extension number, the application will be executed.
- application: Between these tags will go all the necessary commands that compose the application.

Once the general data of the application have been specified, the next step is to write the commands that make up the application, between the *<application>* tags. These commands follow the format shown below:

```
<command num="NUM" type="TYPE"></command>
```

with:

- num: Specifies the execution order inside the list of commands.
- type: The command to be executed.

The RTMIVA architecture has three commands predefined although it's also possible to define new ones. For this application, a *synthesiseweb* command has been defined. Following is shown how to use these commands:

### Synthesise

This command synthesises the phrase that is specified between the command tags. The insertion of variables is allowed as long as they have been defined and initialised, for example having executed previously a *recognise* command that has loaded the variable with one of the possible values.

```
<command num="1" type="synthesise">Welcome to weather forecast system.</
command>
```

### Recognise

This command is responsible for recording the user's speech for a number of seconds specified by a *seg* parameter. Then it sends the audio obtained to the ASR.

For each ASR recognition request, it's necessary to generate a LM so that it's necessary to specify the LM between the tags of the command. For this, the *lexicon* tag is available, which is used to determine the words that form the LM and their interconnection. Each word is included with a *lexicon* tag and the interconnection is specified with the parameters of the tag. These parameters are:

- num: The identifier of the word.
- pre: Identifiers of preceding words. If there are no preceding words this parameter will be "0". When there are several preceding words these are separated with a comma.
- suc: Identifiers of successor words. If there are no successor words this parameter will be "End". When there are several successor words these are separated with a comma.

In addition, it's also possible to use variables using the "%" character. These variables will be used to generate more complex LMs using them as containers of multiple values. The possible values that these variables can take must be defined. For this, a *variable* tag is available which has two parameters: *nom* that is used to specify the name of the variable for which we are defining the possible values and *numtotal* that defines the number of possible values that the variable can take. Between the starting and ending labels of *variable* tag the possible values are defined via a *value* tag.

In the next listing can be seen how the command would look like.

```
<command num="1" type="recognise" seg="6">
  <lexicon num="1" pre="0" suc="2,3,5,6,7">Hello</lexicon>
  ...
  ...
  ...
  <lexicon num="24" pre="17,23" suc="25,End">%city</lexicon>
  <lexicon num="25" pre="24" suc="End">please</lexicon>
  <variable numtotal="10" nom="city">
    <value num="1">Bilbao</value>
    ...
    ...
    ...
    <value num="10">Oviedo</value>
  </variable>
</command>
```

### Check

This command is used as a verification method that the user's speech input has been correctly understood by the system. It's similar to the *recognise*



command in the sense that it records the user's speech and sends it to the ASR to be recognised, although in this case, it's not necessary to specify a LM since it uses a prefixed one in the system. In addition, this command has an *iffalse* parameter that indicates a command number to jump to in case the user indicates to the system that what has been recognised is not correct.

Usually, this command must be preceded by a *synthesise* command to make clear to the user that the system is looking for confirmation. An example of using this command is shown below.

```
<application>
...
...
...
<command num="4" type="synthesise">Would you like to know the weather forecast
for \%city?</command>
<command num="5" type="check" iffalse="2"><command/>
</application>
```

### SynthesiseWeb

This command synthesises the phrase obtained from a URL passed to it as a parameter. The prototype of the command is as follows:

```
<command num="6" type="synthesiseweb">http://localhost/rtmiva/weather.php?city
=\%city</command>
```

For the implementation of the weather forecast application, a *weather.php* file has been developed and inserted in a web server. This file receives the name of a city via an HTTP GET request (given in the *synthesiseweb* command) and gets from Yahoo Weather the identification code of the city. For that a request with the following format is made:

*http://xoap.weather.com/weather/search/search?where=CITY\_NAME*

As an example, to know the identification code of Zaragoza a query to the following URL will be done:

*http://xoap.weather.com/weather/search/search?where=Zaragoza*

which will return the following result:

```
<search ver="3.0">
<loc id="MXSP1937" type="1">Zaragoza, Mexico</loc>
<loc id="MXVZ2292" type="1">Zaragoza, Mexico</loc>
<loc id="SPXX0086" type="1">Zaragoza, Spain</loc>
<loc id="MXCA0368" type="1">Ignacio Zaragoza, Mexico</loc>
<loc id="MXNL1054" type="1">General Zaragoza, Mexico</loc>
<loc id="MXCA0397" type="1">Valle De Zaragoza, Mexico</loc>
<loc id="MXMO0770" type="1">Atizapan De Zaragoza, Mexico</loc>
<loc id="MXOA1123" type="1">Juchitan De Zaragoza, Mexico</loc>
</search>
```

As can be seen in the XML that is received, there are many cities with the same name so, taking into account that the application we have developed is intended for Spanish cities, the code of the city containing *Spain* will be extracted. This code is given by the *id* parameter of the *loc* tag.

Once the code of the city has been extracted, we proceeded to request the weather to Yahoo Weather through the following url:

*http://weather.yahooapis.com/forecastrss?p=SPXX0086&u=c*

Sending the city code in the *p* parameter of the HTTP GET request and indicating that we want the temperatures of the response in Celsius degrees via the *u* parameter.

When making this request, an XML is obtained with the weather forecast which is then transformed into an appropriate format to be synthesised to the user.

### 3.3.2 Integrating the Application into Asterisk

For the integration of the application in the platform, on the one hand, the created file must be located in */var/www/applications/xml* directory. On the other hand, we will have to add its corresponding extension in the *extensions.conf* file. In this example we used "1" as the extension, so we add this value:

```
Exten => 1,1,Answer()
Exten => 1,2,AGI(central.php,{ $EXTEN },{ $CALLERID(name) })
Exten => 1,3,Hangup()
```

## 4 Concluding Remarks

This chapter has shown, on the one hand, a system aimed to give weather forecast information and news obtained from several digital publications. On the other hand, an architecture the author developed with the idea of having a framework in which SDSs can be implemented easily. The first system let us meet the needs imposed by the company. On the other hand, we got a fully functional architecture that allowed easy development of dialogue systems.

In any case, there was a limitation as far as the dialogue management is concerned. Being systems in which the dialogue management is carried out by the interoperation of different components, the adaptation to new tasks (for the first system) or the creation of more complex tasks become complicated and not flexible. Therefore, we saw the necessity to separate the dialogue management task into a single component to solve these problems. This strategy is the one followed in the systems shown in the following chapters.



# 5

## BUS INFORMATION SYSTEM BASED ON EDECAN FRAMEWORK

---

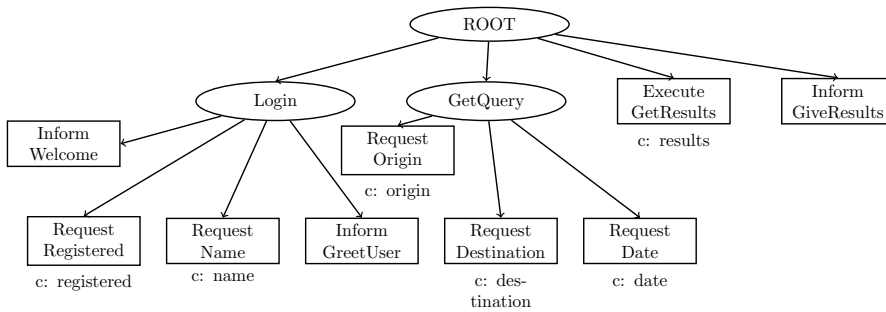
### **1** Introduction

---

As has been said, one of the author's objectives is to learn how to build dialogue systems. In the previous chapter, a system was implemented using the EDECAN framework. It was also showed the development of a new architecture for building dialogue systems. It was concluded that although the systems were fully functional, it's convenient to isolate the dialogue management task in successive systems.

On the other hand, given the interest in dialogue systems, the author is also interested in having a system that can serve as a basis for the subsequent adaptation to new tasks. Additionally, such a system will serve for the acquisition of dialogue data with real users that can then be used for the study of statistical dialogue managers. Thus, the system presented in this chapter was developed.

The system consists of a Phone Bus Information System in Spanish under EDECAN [86] framework. Regarding the DM, after an analysis of the different frameworks and toolkits found, it was concluded that the CMU RavenClaw DM was the most appropriate one because of being a well-tested and already used in real systems DM [58]. Section 2 describes the main characteristics of the RavenClaw DM. Section 3 describes the analysis performed on a corpus of telephone calls to an information system of bus schedules in Spanish. Through this analysis, the relevant information was extracted for later integration into the RavenClaw dialogue strategy. On the other hand, before the implementation of the system, the RavenClaw DM operation and programming was tested using its framework via a port of the CMU Let's Go system [58] to Spanish. That system was also used to develop and test the language dependent components that later were used in the final system. This is detailed in Section 4. Finally, Section 5 describes the developed system and the experiments carried out with real users.



**Figure 5.1:** RavenClaw dialogue tree example.

## 2

## RavenClaw Dialogue Manager

RavenClaw [75] is the DM used in the CMU's Olympus SDS development framework. It provides an advanced management structure with distributed software agents; i.e. dialogue flow can be modelled by software agents reasoning about their state and the next action. RavenClaw enforces a clear separation between the domain-dependent and the domain-independent aspects of the dialogue control logic. The domain-specific aspects are defined by a dialogue task specification, and a domain-independent dialogue engine executes the given dialogue task to manage the dialogue.

The dialogue task specification consists of a tree of dialogue agents. The tree is traversed recursively from left to right and from top to bottom. The execution of the dialogue task ends when the bottom-right node has been reached. During this process, preconditions, triggers and success/failure criteria may be added to the nodes to alter the normal exploration of the tree, allowing the definition of more complex dialogue structures. Figure 5.1 shows a dialogue tree example.

The dialogue agents can be classified into two categories:

- Fundamental dialogue agents (square nodes in Figure 5.1): Located at terminal positions in the tree. There are four types:
  - Inform: Generates an output.
  - Request: Request some information from the user.
  - Expect: Expects some information from the user without explicitly requesting it.
  - Execute: Performs some operation such as a database access.
- Dialogue-agencies (ellipse nodes in Figure 5.1): Located at non-terminal positions in the tree. Control the execution of the children nodes and encapsulate the higher level structure of the dialogue task.

On the other hand, the data that the system manipulates during the dialogue

is encapsulated in concepts. Concepts can be associated with various agents in the dialogue task tree, e.g. *registered* and *name* in Figure 5.1, can be accessed and manipulated by any agent in the tree. There are several basic concept types predefined in the RavenClaw dialogue management framework (Boolean, string, integer and float) although more complex developer-defined concept types can be defined, such as structures and arrays.

Regarding the domain-independent dialogue engine, to execute the dialogue task tree, it uses two data structures: a *dialogue stack* to capture the discourse structure at runtime, and an *expectation agenda* to capture what the system expects to hear from the user in any given turn.

To control the dialogue, the dialogue engine interleaves Execution and Input Phases. During the execution phases, dialogue agents from the task tree are placed on and executed from the dialogue stack, generating in the process the system behaviour. During the input phases, the system uses the expectation agenda to transfer information from the current user input into the concepts defined in the dialogue task tree.

During execution phases, first, the dialogue engine invokes the agent on top of the dialogue stack. When it has finished its task, the control is returned to the dialogue engine which tests the completion conditions for all the agents on the dialogue stack eliminating completed agents from the dialogue stack. Finally, the dialogue engine inspects the trigger (focus claims) conditions for all the agents in the dialogue task tree. If an agent in the task tree request focus, it's pushed on top of the dialogue stack.

On the other hand, during input phases (when executing Request agents), first, the system assembles the *expectation agenda*, a data structure organised into multiple levels each corresponding to one of the agents on the dialogue stack. The dialogue engine traverses the stack, from top to bottom, and constructs the corresponding levels in the expectation agenda. After the expectation agenda has been assembled, the dialogue engine waits for a user input. Finally, once an input arrives, the dialogue engine starts a concept binding stage in which the information available in the input is used to update system concepts. For that, The dialogue engine performs a top-down traversal of the agenda, looking for matching grammar slots in the user input. Wherever a match is found, the corresponding concept is updated accordingly. Once the input phase completes, the dialogue engine continues with another execution phase.

The second defining structure, the agenda, is an ordered list of agents used to dispatch inputs to appropriate agents in the task tree. It's recomputed for every turn, and the current agent is placed on top of the stack. Inputs are matched to successive items on the agenda. When a match occurs the corresponding agent is activated with the matching concepts as inputs of the dialogue. An agent may not consume all input concepts and thus remaining concepts are passed further down the agenda until agents can consume them.

### 3 Data Analysis

Given the choice of RavenClaw as DM for our system, we based our work on the implementation of the Let's Go system. Taking this into account and that the needs of the users of both systems do not have to be the same, an analysis of a corpus of customer service for bus schedules in Spanish for the city of Bilbao (Corpus 010) was done. We analysed, on the one hand, the necessary modifications in the dialogue strategy of the DM and, on the other hand, the necessary changes in other components of the Let's Go system (mainly the SP and NLG, which were also integrated in our system since it's closely linked to the RavenClaw DM).

The Bilbao city council has a telephone line to provide information about the city, including information about public transit. We were provided with such bus schedule information by the Bilbao city council. This information consisted of 353 dialogues of which 25 were in Basque and the remainder (328) in Spanish. 277 of these dialogues were held with female subjects and the remainder with male subjects. There were a total of 3976 user turns in the corpus.

To evaluate the user goals, the type of answers, the bus stops requested and other relevant data, the dialogues were analysed one by one. During the analysis the focus was especially on the first user turn following the idea that if we have an open question in the beginning of the dialogue (e.g.: How can I help you?), if the system is able to correctly understand the first user utterance, it would be able to conveniently drive the dialogue from that point on. The user goals in the 010 corpus were compared to those covered by the original Let's Go grammar distributed for the Spoken Dialogue Challenge [103]. Some of these goals were not present in the Let's Go grammar and were added to the grammar definition. This grammar was compiled and used to parse the user turns in the 010 corpus with an off-line version of the Phoenix parser [104], similar to the one used by the live system.

The result of the off-line parsing are presented in Table 5.1 and an example of a dialogue extracted from the 010 corpus is shown in Listing 5.1. The parsing results show that many parsing functions are not covered in the Let's Go grammar, and thus this requires modifications in the Let's Go dialogue strategy. The Let's Go grammar was not prepared to deal with frequency, last or first bus requests. Also, since the phone number used in Let's Go is the Port Authority number, the telephone requests were not necessary for the original system. Instead, for our case, the city council number is different from the bus company, and there were dialogues where users wanted the current bus company direct phone number.

These new goals required modifications in many dialogue modules that will be described in detail in the following sections.

Operator: Good morning 010 you are speaking to Ana  
 User: Hello, good morning  
 O: Tell me  
 U: Yes. Please, could you tell me at what time does the bus number 76 leaves from Recalde?  
 O: Yes, We look at it right now, just a moment. 76  
 U: Six  
 O: From Artazu Salvador at 8, 8:40, 9, 9:20, 9:40, 10, 10:20  
 U: So, at o'clock, at twenties and at forties  
 O: At forties, yes  
 U: OK  
 O: Good  
 U: So, it has the same that, uhm, the same minutes that from Federico Moyua upwards, that's it?  
 O: That is both back and forth  
 U: Back  
 O: It is the same time  
 U: OK, good, thank you very much  
 O: OK. Thank you for your call. Bye  
 U: Bye, bye

**Listing 5.1:** Example of a dialogue in the 010 corpus.

Function	Turns	Sub-functions	Sub-function turns
Time Information	992	Bus Frequency	33
		Last Bus	16
		First Bus	33
		Next Bus (*)	4
		Previous Bus (*)	5
		Time (*)	775
		Time Relative (*)	48
Line Information	71	Bus Number (*)	20
		Bus Direction (*)	51
Disambiguation	43	Departure (*)	29
		Arrival (*)	13
Generic	1814	Phone	3
		Suspend (*)	1
		Quit (*)	242
		Yes (*)	1023
		Good Enough (*)	1
		No (*)	563
		Don't Know (*)	61

**Table 5.1:** User goals detected in the 010 Corpus. Sub-functions observed in the Let's Go grammar are marked with (\*).

The final goal of the author is to implement the system presented in this chapter on a Unix system, although some tests were initially performed with



the Let's Go system. On the one hand, this system was used as a platform to test the programming and operation of the RavenClaw dialogue manager. On the other hand, it was also used as an initial platform on which make the necessary changes in the language-dependent components for their adaptation to the Spanish language.

Although the final system will be described in the next section, this section details the components that make up the Let's Go system. That could be of interest to the reader interested in building dialogue systems. Also, for the case of language-dependent components (mainly SP and NLG) the information provided is directly related to the final system since these components are the same in both systems.

## 4.1 Architecture

---

The Let's Go bus information system is built with the Olympus open-source framework for SDSs [74]. This framework has all the modules that compose an SDS: Speech Recognition (PocketSphinx [105]), Natural Language Understanding (Phoenix and Helios [106]), Dialogue Management (RavenClaw [75]), template-based Natural Language Generation (Rosetta [106]) and Synthesis Engine (Kalliope [74]). These modules communicate with socket-based protocols via a central HUB which was developed to follow the Galaxy-II protocol [32]. The Interaction Manager [107] will coordinate the modules. There are other modules such as the DateTime Parser and the Back-end Server that are also distributed with Olympus. The first one converts text dates into numerical dates, whereas the Back-end implements the communication between the DM and the database that the system might need to access.

Many of these modules are either language or application dependent. The steps that were taken to have a fully-working system in Spanish will be described in the following subsections. The details will be given module by module. The final architecture is shown in Figure 5.2.

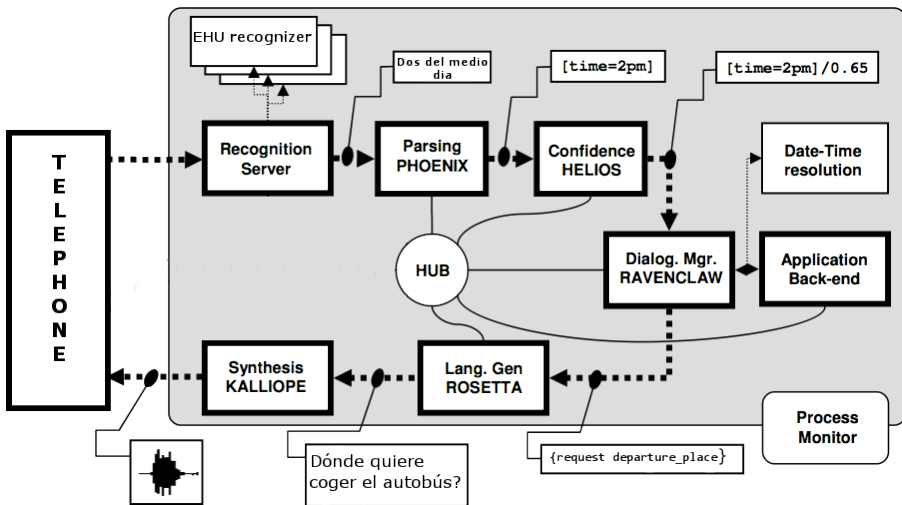
### 4.1.1 Automatic Speech Recognition

---

Olympus distribution comes with PocketSphinx [105], an open source large vocabulary, speaker-independent, continuous speech recognition engine developed by CMU.

It was decided to use the speech recogniser developed by the group *Pattern recognition and speech technology* of the Department of Electricity and Electronics of the Faculty of Science and Technology of the UPV/EHU. Taking into account the socket communication protocol implemented for Sphinx, this recogniser was integrated into the Olympus architecture.

We used acoustic models trained with the Dihana corpus [108], a corpus collected using the Wizard of Oz (WOZ) paradigm and consisting of a set of



**Figure 5.2:** Let's Go Spanish Port: System architecture, based on architecture proposed in [2].

900 dialogues in Spanish which task is about railway information for timetables, fares and services for long-distance trains in the Spanish territory. It was recorded at 8 KHz.

On the other hand, a language model had to be trained. For that, a script used to train the PocketSphinx language models, that automatically generates a corpus from the Phoenix grammar specification, was adapted. The generated corpus was then converted into a format that can be used by the tools that are used to generate new language models for the EHU recogniser. A few changes were introduced in the generation process to respect the gender and number agreements in Spanish. Five different language models were trained: Default, Place, Confirmation, Time and Next Query. Default was the one loaded at the beginning of the dialogue. It was trained with 300k sentences for all the contexts. Context-dependent models such as Place, Confirmation, Time and Next Query, are loaded at the corresponding dialogue state. Unlike the Default corpus, the corpora created to train these language models, only contains sentences from those states, that is, the Place model was trained with sentences typically used to ask for a Place information. They were trained with 200k, 5k, 300k and 50k sentences respectively. The number of words in the lexicon is 421 for Default, 360 for Place, 63 for Confirmation 311 for Time and 117 for Next Query.

#### 4.1.2 Natural Language Understanding

In Let's Go the Natural Language Understanding task is performed by two modules: Phoenix does the semantic parsing, and Helios performs confidence

```
FRAME: <frame name>
NETS:
  [<slot name>]
  [<slot name>]
;
```

**Listing 5.2:** Phoenix Semantic Parser frames file syntax.

annotation. The parsing is done using recursive transition networks that are compiled from the grammar definition. To have a network for the system in Spanish, a new grammar definition was created, according to the needs identified from the 010 Corpus. Needed steps to create this grammar are detailed below.

On the other hand, the `DateTime` module for parsing dates had to be modified. This module was translated to Spanish. Also, since the system deals with new time requests: frequency, first and last bus. The `DateTime` module was modified to parse these requests correctly.

Finally, for confidence annotation, the model distributed with Let's Go was used, although once enough data (around 500 dialogues) is collected from live users, an adapted model to the system in Spanish should be trained.

#### 4.1.2.1 Parser Grammar Definition

The Phoenix parser maps input word strings onto a sequence of semantic frames. A frame is a named set of slots, where the slots represent related pieces of information. Each slot has an associated Context-Free Grammar that specifies word string patterns that match the slot. The grammars are compiled into Recursive Transition Networks (RTNs). When filled in by the parser, each slot contains a semantic parse tree for the string of words it spans. The root of the parse tree is the slot name. For each application we want to develop, it's necessary to define the frames, slots and associated grammars. This is done by means of defining two files, a frames file and a grammar file.

#### Frames File

This file specifies the frames that are going to be used by the parser. A frame represents some basic type of action or object for the application. Slots in a frame represent information that is relevant to the action or object. Each slot name has an associated set of grammar rules (defined in the grammars file we will see below). The slot name will be the root of the corresponding semantic parse tree. For defining a frame, we must follow the syntax shown in Listing 5.2. An example of how it would look like, extracted from our system, is shown in Listing 5.3. In this example, the *TimeInformation* frame is used to parse information related to times and dates and *PlaceInformation* to parse information related to the places users can speak about.

```

FRAME: TimeInformation
NETS:
  [4_DepartureTime]
  [4_ArrivalTime]
  [4_DateTime]
;

FRAME: PlaceInformation
NETS:
  [1_SinglePlace]
  [2_DeparturePlace]
  [3_ArrivalPlace]
  [4_ComboPlace]
  [AnyStop]
;

```

**Listing 5.3:** Phoenix Semantic Parser frames file example.

```

[token_name]
(<pattern a>)
(<pattern b>)

MACRO1
(<pattern c>)

MACRO2
(<pattern d>)
;

```

**Listing 5.4:** Phoenix Semantic Parser grammar file syntax.

### Grammars File

The grammar file contains the grammar rules. The grammars are context-free rules that specify the word patterns corresponding to the token (network name). The syntax for a grammar for a token is shown in Listing 5.4.

The notation used in pattern specification is as follows:

- Lower case strings are terminals.
- Upper case strings are macros.
- Names enclosed in [] are non-terminals (calls to other token rules).
- Regular Expressions:
  - \*item indicates 0 or 1 repetitions of the item
  - + indicates 1 or more repetitions
  - +\* indicates 0 or more repetitions
- #include <filename> reads file at that point.

An example extracted from our system is shown in Listing 5.5.

```

[1..SinglePlace]
  ([stop_name])
;

[2..DeparturePlace]
  (estoy en [stop_name])
  (de [stop_name])
  (desde [stop_name])
  (*el NEXT *BUS *DE [stop_name])

NEXT
  (próximo)
  (siguiente)

BUS
  (bus)
  (autobús)

DE
  (de)
  (desde)
;

[stop_name]
  (san mamés)
  (abando)
  (universidad)
;

```

**Listing 5.5:** Phoenix Semantic Parser grammar file example.

### Compiling the Grammar

Once the frames and grammar files have been written, it's necessary to compile the grammars. This is done with an application distributed by CMU. The compilation of the files produces another set of files: *forms*, *base.dic*, and *Grammar.net*.

The *forms* file includes the definition of the defined frames and has the same content than the frames file. The *Grammar.net* includes the definition of the RTN grammars and the *base.dic* includes a list of all the words appearing in the grammars.

#### 4.1.3 Dialogue Manager

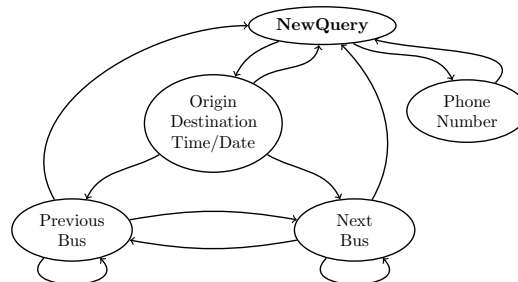
The analysis of the 010 Corpus made in Section 3 showed that user goals might be different from those originally found in Let's Go. Thus, the dialogue strategy had to be modified to deal with them.

For the new Time Information intentions (Frequency, First and Last bus), new strategies were implemented. This required changes in the backend as well. For frequency requests, first it was needed to collect all the results for a specific

route within the day of the request. Second, the difference between the first two departure schedules was computed and the frequency value was set. Third, the difference between two successive departures was successively computed until the value was different from the frequency set. Finally, the difference and the departing time of the previous trip will be returned. For first bus queries, the query was similar to a normal bus query. However the query returns the first time posterior to a time value indicating the bus functioning hours. For last bus, the same idea was followed, but in this case the query returns the last time prior to this time value. The value used to the time value indicating the functioning hours was 4:30 am. This time value represents the change of day time from the bus lines point of view that doesn't match with the normal change of day time, that is 12:00 am.

The Phone requests happen since the 010 line is the city council help service and sometimes they want to talk directly with the bus company. Whenever the system detects that the bus company phone number has been requested, the dialogue will immediately jump to a state where the phone number is provided. Once the number is given, the system goes to the next query state, where it asks if the user wants to make a bus request or leave the system.

To summarise, the dialogue strategy is shown schematically in Figure 5.3 and the RavenClaw task tree is shown in Figure 5.4.



**Figure 5.3:** Basic task specification for the Campus SDS system.

#### 4.1.4 Data Backend

The backend of this system is the bus schedule database. The database used in Let's Go uses the postgres format. Whenever a schedule modification takes place, the local authority generates the new schedules in the Google Transit Format Specification (GTFS). The schedules in this format are then converted into a postgres database to keep the compatibility with the previous backend architecture.

For development purposes the Bilbao Metro schedule from July 2011 in the GTFS format was obtained from GTFS data exchange [109]. This data was

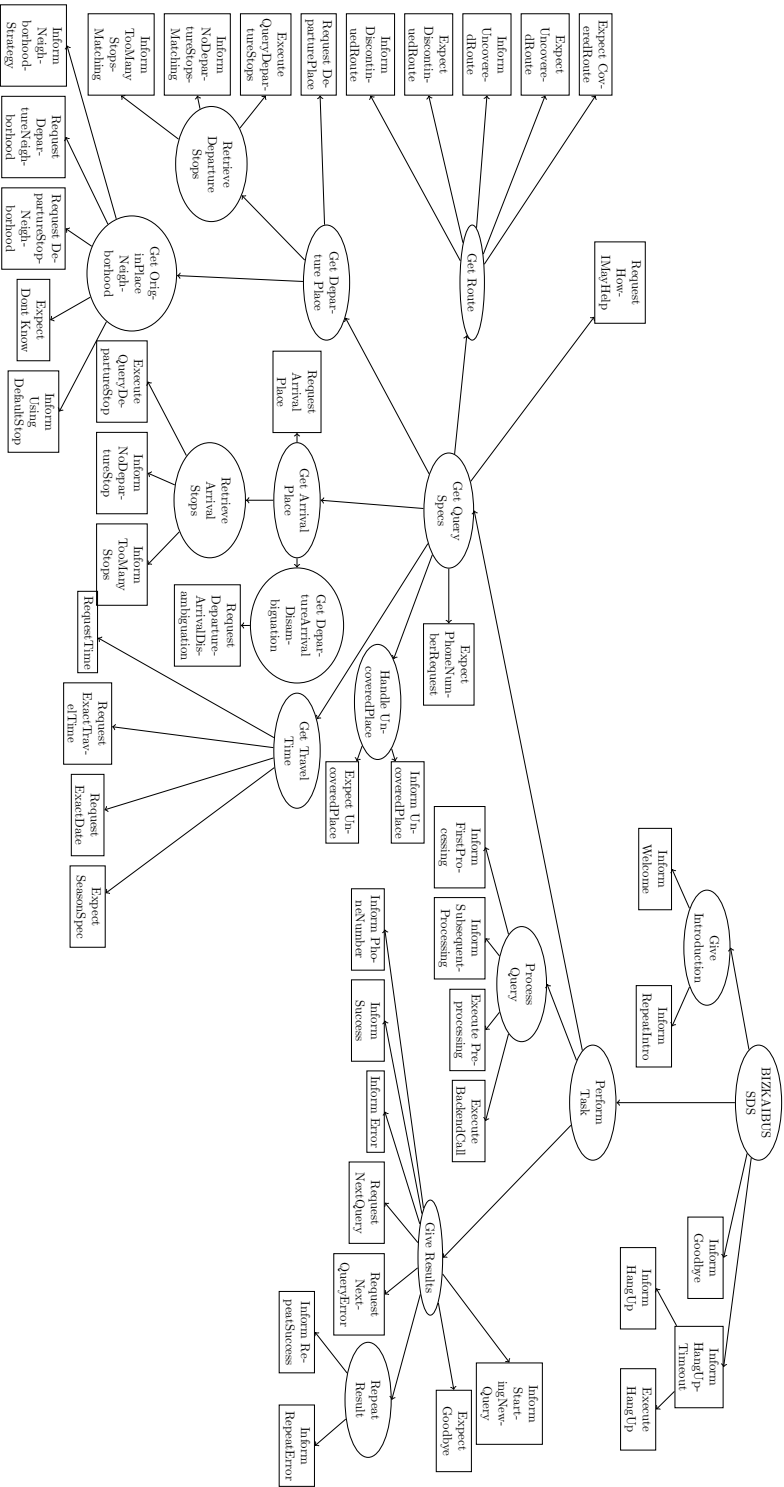


Figure 5.4: Task Tree for the Bus Information System implemented with Ravenclaw.

converted into a postgres database.

Minor changes in the backend processing were introduced to accommodate the new request types and other peculiarities of the Metro Bilbao schedule. Concerning the requests for first and last bus requests, the hour from when it starts searching for buses was adjusted to the change of day. The Bilbao Metro change of day is also different from day-to-day. Fridays it usually runs until 2 am (Saturday) and it does not stop from Saturday morning until Sunday 11 pm. This has to be taken into account whenever querying for first or last buses.

#### 4.1.5 Natural Language Generation

Rosetta is the Olympus module for language generation. It's a template-based NLG system in which generation of sentences is specified by a form, a sub-form, and attribute values that are used to fill a template. Also, it can generate variations based on input and context values. It separates the generic processing, that is, the interface with the dialogue system, from the domain-specific template processing.

The templates are composed of handwritten entries, which can also be programmatic functions (to deal with missing slots in the template), and are used to convert the semantic representation generated by the DM into a sentence in natural language that can be synthesised. The templates were adapted from the Let's Go original templates into Spanish. When converting them, gender and number variations were taken into account which increased the complexity of the templates. New sentences for the new types of requests supported by the system were also added to the templates.

##### 4.1.5.1 Templates Definition

As has been said, Rosetta separates the generic processing from the domain-specific template processing, and so it's necessary to write the domain-specific templates for the system. For writing these templates it's necessary to generate as many as request types from the DM which in this system are, *inform* and *request*. The *inform* request type is used when the DM just wants to give some information to the user and the *request* one to ask for some information to the user. We will have to write two files, *Inform.pm* and *Request.pm*. We will explain the procedure of writing the templates with an example of an *inform* request.

When the DM requests for a sentence to be generated it sends a message with the type of request and some additional information necessary for the NLG. An example of a such a message, and a template that will generate a sentence with it, is shown in Listing 5.6.

The NLG extracts the value of the *act* tag, *act* calls to the corresponding template, *Inform.pm* in this example, providing the rest of the information



<pre> start {   act    inform   object too_many_stops   place {     name  LAS ARENAS     type  neighborhood   } } end </pre>	<pre> \$Rosetta::NLG::act{"inform"} = {   "too_many_stops" =&gt; sub{     my %args = @_;     my \$place_name = getPlaceName(%args);     return "Sorry, \$place_name isn't specific enough.",   }, } </pre>
--	--

**Listing 5.6:** Rosetta Natural Language Generator: Input message and template example.

```

$Rosetta::NLG::act{"inform"} = {
  "welcome" => "Welcome to the bus information system",
  "bye" => ["Bye bye.",
           "Good bye.",
           "See you."],
  "too_many_stops" => sub{
    my %args = @_;
    my $place_name = getPlaceName(%args);
    return "Sorry, $place_name isn't specific enough.",
  },
}

```

**Listing 5.7:** Rosetta Natural Language Generator: Template example.

on the message as arguments. The invoked template will use the *object* tag to generate the required sentence. In the example, this means calling to the subroutine associated with *too\_many\_stops* to which the *place* structure will be passed as an argument. The sentence generated in this example will be: *Sorry, Las Arenas isn't specific enough.*

Listing 5.7 shows another example, as can be seen not only subroutines are used for sentences generation. Also individual sentences or sets of sentences from which one will be randomly chosen are used.

#### 4.1.6 Speech Synthesis

On the other hand, for speech synthesis, Olympus uses the Kalliope synthesis manager. Kalliope can be configured to use four different types of synthesisers: Festival [110], Flite [111], SAPI [112] and Cepstral Swift [113]. For the system, it was decided to use Loquendo TTS [100]. Loquendo provides good quality Spanish voices and support the SAPI interface.

#### 4.1.7 Telephone Interface

---

The original Let's Go telephone interface uses an analogue PBX to send the calls to the system. That solution relies on hardware that is nowadays obsolete and is not available in the market.

There is another solution for Olympus based on the PJSIP open source project [114] that allows the system to connect to a digital PBX using the SIP protocol. This system had to be configured to connect to the telephony services of the University of the Basque Country. On the other hand, to route the audio streams to the PBX, two virtual sound cards had to be emulated using Virtual Audio Cable 2 [115].

## 5 Bus Information System Based on EDECAN

---

The primary goal of this chapter is to create a Phone Bus Information SDS in Spanish similar to Let's Go [58].

The Olympus framework could be a good framework to get our goal, but taking into account that our main goal was to have a fully working system in an Unix-like system it was unreachable. The Olympus distribution is distributed as a Windows-only software and although some parts have been ported to Unix-like systems it was impossible to get a fully working system. So we decided to build the system under the EDECAN framework [86].

The developed system is providing bus schedules through the telephone to real users from the University of the Basque Country community that study or work in the Campus of Leioa, in the north of Spain. This section details the architecture of the system, its configuration, detailed information about how each of the system modules was implemented and finally the results of some experiments carried out related to the user experience and usability of the system.

### 5.1 Architecture

---

The architecture of the system is shown in Figure 5.5. As can be seen, it consists of the usual elements found in SDSs: ASR, TTS, NLG, NLU and DM. On the other hand, also has a Phone Interface that is in charge of receiving phone calls from users. A controller module has also been added. This module has a double functionality, on the one hand, has the task of distributing the messages that are exchanged within the system receiving the messages sent by the other modules and redirecting them to the corresponding module, on the other hand, acts as an error detection module detecting possible anomalies in the operation of the system and acting accordingly. EDECAN architecture, by its construction, is capable of carrying out the first of the functions of the controller, but it has been decided to add the controller, on the one hand,

to simplify the development process and, on the other hand, to integrate the error handling function.

Also, a DM Interface module is available. The proposed system has been developed under the Linux operating system. At the time of integrating the DM there was not a version of the RavenClaw DM that could be ported and integrated natively in the EDECAN framework so it was decided to use the existing version for the Windows operating system and look for a solution. To do so, it was decided to use the DM running on its framework, Olympus, and to develop a module that would serve as a connection between this framework and the EDECAN framework. This module acts as a client of both frameworks at the same time transforming the format of the messages of one architecture to the other and vice versa. Also, this module has been used to perform the functions of backend server for access to the database of the application.

Finally, a DateTime module is available, responsible for transforming the dates and times provided by users, in text format obtained after the ASR process, to the dates format used internally by the DM.

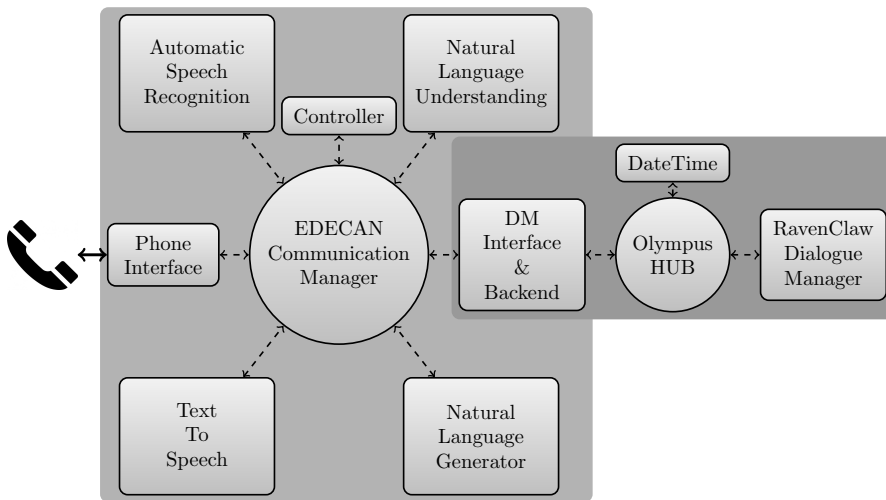


Figure 5.5: Architecture of the Campus SDS System.

On the other hand, the communication between the different modules is done by the exchange of messages that represent commands to be executed by the different modules. Table 5.2 shows the input and output commands for each of the modules that make up the system together with their arguments between parentheses.

The flow of these messages during the execution of the system would be as follows, remember that all messages go through the controller: Initially, when receiving a call from a user, the Phone Interface generates a *dm\_init\_session* message directed to the DM requesting to start a new dialogue.

	INPUT	OUTPUT
ASR	asr_run	asr_end (text)
SP	sp_run (text)	sp_end (text, parsed_text)
NLG	nlg_run (frame)	nlg_end (text)
TTS	tts_run (text)	tts_end (text)
DM Interface	dm_init_session dm_hangout dm_nonunderstanding_startover dm_nonunderstanding_giveup sp_end (text, parsed_text) tts_end	asr_run nlg_run (frame, Count) dm_end_session
Phone Interface	/ / / / /	dm_init_session dm_hangout
CONTROLLER	dm_init_session dm_hangout dm_end_session nlg_run (frame, Count) nlg_end (text) asr_run asr_end (text) tts_end (text) sp_end (text, parsed_text)	dm_init_session dm_hangout dm_nonunderstanding_startover dm_nonunderstanding_giveup nlg_run (frame) sp_run (text) asr_run tts_run (text)

**Table 5.2:** Set of commands (and arguments) for the different modules in the Campus SDS System.

The DM can respond with two types of messages, *inform* or *request*, indicating that either it wants the system to generate a sentence to inform the user or wants the ASR to be activated to obtain information from the user. Both messages imply the synthesis of some information destined to the user but they have associated a parameter that is used to know if after the synthesis it's necessary to activate the ASR or not. These messages arrive at the controller which generates an *nlg\_run* message destined to the NLG to generate the sentence that will later be synthesised and played to the user. The NLG, after generating the requested sentence, will send an *nlg\_end* message directed to the TTS which after synthesizing and playing the sentence generates a *tts\_end* message directed to the DM to inform that the audio synthesis has finished.

At this point depending on the value of the mentioned parameter the system will act in one way or another. In case the parameter indicates the activation of the ASR, the controller will generate an *asr\_run* message to activate the ASR. The ASR after receiving the message will begin to capture the intervention of the user to later carry out the recognition process. Once the recognition process has finished the ASR generates an *asr\_end* message with the recognition hypotheses directed to the SP which will extract the information and send it using an *sp\_end* message to the DM.

From here, whether the parameter indicates to activate the ASR or not, the DM will generate a new message (inform or request) and the same messages

flow will start again until either the end of the dialogue arrives, in which case a *dm\_end\_session* message will be generated, or the user terminates the call, in which case a *dm\_hangout* message will be generated. Both messages are intended to restart the system and put it on hold for a new call.

On the other hand, sometimes could occur that the controller detects that there are problems with the system to understand the users' interactions and in such cases the controller will send *dm\_nounderstanding\_startover* or *dm\_nounderstanding\_giveup* messages to the DM to inform of such situation and for the DM to act accordingly.

## 5.2 Configuring and Launching the System

### 5.2.1 Audio interface configuration

Audio interchange between the phone interface module with ASR and TTS respectively is carried out in a different way from that of message interchanging through the communication manager and controller module.

The surrounding idea of how they communicate is that the different modules read and write the audio signal from and to the audio interface and therefore having a module reading from the audio interface what another has written. The main problem with this approach is that the operating systems don't allow having an audio interface opened in both read and write modes at the same time. To solve this problem the concept of *loopback audio device* was used and specifically for the Linux operating system the *ALSA Loopback audio device*. An ALSA Loopback audio device is a virtual audio device created once *snd-aloop*, an ALSA kernel module, is loaded and that sends back the output signal back to itself. By default, a Loopback audio device consists of 2 devices, each composed of 8 subdevices, although the number of subdevices can be changed with a *pcm\_substreams* kernel module option.

In a formal way, this virtual sound card consists of 2 devices:

$$\begin{aligned} &hw:Loopback,0 \\ &hw:Loopback,1 \end{aligned}$$

If an application writes its output audio signal to one of the subdevices, let's say *hw:Loopback,0,0*, the audio will be available as input in the corresponding subdevice *hw:Loopback,1,0*.

So, the generic principle is that an output signal to subdevice *hw:Loopback,i,n* becomes an input signal from *hw:Loopback,j,n* with

$$\begin{aligned} i &= [0 .. 1] \\ j &= i \text{ (meaning if } i = 0, j = 1 \text{ and vice-versa)} \\ n &= [0 .. (s-1)] \text{ with } s = \text{number of subdevices} \end{aligned}$$

```
alias snd-card-0 snd-aloop
options snd-aloop enable=1 index=0 pcm_substreams=2
```

**Listing 5.8:** `/etc/modprobe.d/sound.conf` file for the campus SDS system.

### ALSA Loopback configuration

For the system under consideration a loopback audio device with two devices and two subdevices each is needed, one device for the PHONE-ASR audio interchange and another for the PHONE-TTS interchange.

The first step is to make sure that the `snd-aloop` kernel module is loaded and therefore the following command must be executed:

```
modprobe snd-aloop
```

or, alternatively, added to the `/etc/rc.local` file in order for the module to be loaded at boottime.

As a second step, and to specify the number of devices and subdevices, the `/etc/modprobe.d/sound.conf` file must be edited. Listing 5.8 shows the corresponding file used for the system. The first line specifies the creation of a Loopback device, consisting of two devices and the second line establishes some options for the Loopback device. The `enable` option indicates that the Loopback device is active and the `pcm_substreams` indicates the number of subdevices for each of the devices conforming the Loopback device. On the other hand, the `index` option associates an index to the Loopback device. This is useful to differentiate the Loopback device between other possibly present audio devices in the system, both virtual and physical, and is also necessary to specify to an audio software which device to use.

Finally, as the last step and so that the applications that use the ALSA API have knowledge of the Loopback device and can access it a new file, `.asoundrc`, must be created, with the content shown in Listing 5.9. This file must be located in the user's home directory or renamed to `asound.conf` and located in the `/etc` directory.

#### 5.2.2 Configuring and Launching EDECAN Services

To make the components belonging to the EDECAN framework work the procedure is similar to that described in Section 2.2 of Chapter 4. On the one hand, we must run the services that make the system, and on the other hand, the communication manager.

For running the services we can do:

```

pcm.loop000 {
    type plug
    slave.pcm "hw:0,0,0"
}

pcm.loop010 {
    type plug
    slave.pcm "hw:0,1,0"
}

pcm.loop001 {
    type plug
    slave.pcm "hw:0,0,1"
}

pcm.loop011 {
    type plug
    slave.pcm "hw:0,1,1"
}

```

**Listing 5.9:** .asoundrc file for the Campus SDS system.

```
super_server 6611 1 6611
```

In the same way the *config\_files/services/services.xml* file that lists the configuration files corresponding to each of the services to be launched, and such services configuration files must be previously defined. These files are included in Appendix B.

For running the communication manager:

```
comm_manager 6633 6644 localhost config_files/router/manager.xml
```

The corresponding configuration file is given in Appendix B.

### 5.2.3 Configuring and Launching Olympus Services

Olympus framework is similar to EDECAN framework in the sense that it consists of a central node that connects with different services and acts as a message router between them. Similarly, the communication manager (called HUB) must be run on the one hand, and services on the other hand.

To run the HUB:

```
CMU_HUB -pgm_file hubcfg.pgm
```

Listing 5.10 shows the *hubcfg.pgm* configuration file. This file specifies the services that will be connected to the HUB by specifying its ports and ips, and also specifies the commands that can be sent to the respective services. On the other hand, it also specifies some rules to execute depending on the type of messages that the HUB receives. As an example, and looking at the listing,

```

SERVICE_TYPE: dm_interface
SERVICE_PROVIDER: dm_interface
PROVIDER_ID: [dm_interface]
OPERATIONS: reinitialize systemutterance be_launch_query close_session
LOCATION: localhost:18000

SERVICE_TYPE: DialogManager
SERVICE_PROVIDER: DialogManager
PROVIDER_ID: [RavenClaw]
OPERATIONS: reinitialize begin_session handle_event start_inactivity_timeout
              cancel_inactivity_timeout end_session
LOCATION: localhost:17000

SERVICE_TYPE: datetime
SERVICE_PROVIDER: datetime
PROVIDER_ID: [datetime]
OPERATIONS: reinitialize fr_ParseDateTime fr_ParseTimeRange fr_ParseInt SetRefDate
              CompactDate GetDateTime ParseDateTime ParseInt begin_session
LOCATION: localhost:15000

SERVER: Builtin
OPERATIONS: new_session end_session increment_utterance nop
MODE: singlethread

RULE: :startutt --> Builtin.increment_utterance

RULE: :close_session --> dm_interface.close_session

RULE: :action_type == "system_utterance" --> dm_interface.systemutterance
IN: :properties

```

**Listing 5.10:** Campus SDS System Olympus HUB configuration file.

when the HUB receives a message with the *:close\_session* field, a *close\_session* command will be sent to the *dm\_interface* service.

Finally and to have the system completely functional, it will be necessary to launch the DialogManager and DateTime services. To do this, we execute the following commands:

*wine DM.exe*

and

*CMU\_DateTime*

## 5.3 Modules Implementation Details

### 5.3.1 Automatic Speech Recognition

This module is responsible for acquiring the user's audio signal and once it's obtained sending it to the speech recognizer and retrieving the recognition



```
POST /speech-api/v2/recognize?lang=LANG&key=KEY HTTP/1.1
Host: www.google.com
Content-Type: audio/x-flac; rate=16000
Content-Length: LENGTH

AUDIO_DATA
```

**Listing 5.11:** HTTP POST request example for the Google Speech API.

hypothesis. When it receives an *asr-run* message it starts recording the user's audio signal, capturing it directly from the audio device provided in the service configuration file, that is connected to the phone interface as explained in section 5.2.1. A Voice Activity Detector (VAD) detects when the user has stopped speaking and, at this point, the module sends the audio signal to Google's servers and gets the recognition hypotheses. Finally it sends a *asr-end* message with the recognition hypothesis embedded in it.

The VAD is based on a states machine in which an energy threshold and a silence tail can be tuned to adjust the performance. On the other hand, the audio acquisition was implemented using the Linux ALSA API [116] and for the recognition hypotheses retrieval, the Google Speech API was used.

Google's servers accept HTTP POST requests with the audio signal embedded in the body of the request message and return a text string in JSON format with the recognition hypotheses. In Listing 5.11 is shown the format of the HTTP POST request required to use the API. It consists of a call to an URL to which some parameters are added and a message consisting of a header specifying the format of the audio signal to be sent and a body in which the audio signal is embedded. As for the parameters, two of them must be supplied, on the one hand, a *lang* parameter specifying the language in which the audio signal is recorded which can be any valid locale (es-ES, fr-FR, en-US, ...), and on the other hand a *key* parameter, consisting of a key provided by Google and that can be obtained via Google Developer Console [117]. These parameters are provided to the module in the corresponding service configuration file. Regarding the audio format, the Google Speech API accepts several ones but in the system a 16000 kHz sample rate, 1 channel and 16 bit precision FLAC audio format was used.

On the other hand Listing 5.12 shows an example of a response provided by the Google Speech API. It consists of a set of recognition hypotheses labelled with a confidence score. The module is responsible for extracting the best hypothesis and once it's obtained sending it to the system's controller module.

```

{
  "result": [
    {
      "alternative": [
        {
          "transcript": "hello",
          "confidence": 0.9717918
        },
        {
          "transcript": "LOL",
          "confidence": 0.95454544
        }
      ]
    },
    {
      "final": true
    }
  ],
  "result_index": 0
}

```

**Listing 5.12:** Google Speech API JSON response example.

### 5.3.2 Semantic Parser

For the SP, CMU's Phoenix [9] was used. It was ported to the EDECAN framework directly from the code provided by CMU. Just the necessary changes to integrate it as a module of EDECAN were made.

When the module receives a *sp\_run* message it extracts from it the text utterance to parse, extracts the relevant information and sends a *sp\_end* message with the extracted information embedded in it.

As mentioned above the purpose of the system is to provide bus schedules to the Leioa Campus at the University of the Basque Country in Spain. The service includes 23 different routes connecting the campus to 46 municipalities in the region and includes a total of 452 stops. Some semantic grammars were defined to guarantee coverage of all municipalities, routes and stops. Moreover a large variety of spontaneous and time/dates expressions were also included in Phoenix grammars.

### 5.3.3 Natural Language Generator

We used CMU's Rosetta [118] for NLG. It's distributed as a set of Perl language scripts. For integrating it in the EDECAN architecture a wrapper was written to run these scripts. Following the steps detailed in Section 4.1.5 the required templates were created.

When the module receives a *nlg\_run*, it extracts the information necessary to generate a sentence from the message, and when it has been generated sends a *nlg\_end* message with the generated sentence embedded on it.

**5.3.4** Text to Speech

---

For the TTS module Nuance's Loquendo TTS [100] was used. The module developed for the INMA system was reused with a slight modification consisting in changing the name of the *asr\_run* output command to *tts\_end*.

When the module receives a *tts\_run* it extracts the sentence to be synthesised from the message and synthesises it. When the synthesis has finished the module sends a *tts\_end* message. The synthesised audio is written to the audio device indicated in the parameters provided in the service configuration file and, taking into account that this device is connected to the device from which the phone interface is reading audio, as explained in Section 5.2.1, played to the user.

**5.3.5** Phone Interface

---

This module was developed porting PJSUA, a command line SIP User Agent (UA) written with PJSIP [114] Open source SIP stack, to the EDECAN framework. It's connected as a client to an Asterisk server provided by the University of the Basque Country that allows receiving analogue calls.

The phone interface is waiting for client's calls and when one of them arrives it sends a *dm\_init\_session* message directed to the DM to start the dialogue.

To configure it as a client the parameters in Listing 5.13 were used. From these parameters the most important ones are the first set of them which specify the address of the server to connect to and the username and password to use as a client. Other important parameters are *-capture-dev* and *-playback-dev* which specify the audio devices to use for audio capture and playback respectively. The capture device will be the one that is connected to the device from which the ASR captures the audio and the playback device will be the one that is connected to the device in which the TTS writes the audio.

**5.3.6** Dialogue Manager

---

As for the DM CMU's RavenClaw was used. For this module no extra work was necessary taking into account that it was already developed when testing it in the Olympus framework (Section 4.1.3). It just was necessary to take the DM executable file and run it.

**5.3.7** Dialogue Manager Interface and Data Backend

---

As has been said, the DM used in the system was CMU's RavenClaw [75]. RavenClaw is only distributed as a Windows operating system application. The portability of this DM to a Unix-based development environment was unachievable. Thus it was not possible to port the DM to the EDECAN framework. As a consequence we decided to keep the RavenClaw DM in the

```
--registrar=sip:10.0.14.102
--realm=softphone@ehu
--id=sip:34946017481@10.0.14.102
--username=123456
--password=123456

--reg-timeout=3600

--log-file=pjsua.log
--app-log-level=5

--capture-dev=6
--playback-dev=8

--snd-clock-rate=8000
--clock-rate=8000
--no-vad
--ec-tail=0
--quality=10

--max-calls=1
--auto-answer=200
```

**Listing 5.13:** SIP client configuration file for the Campus SDS System.

Olympus framework, running through the Wine [119] Linux Windows emulator, although it could be running in any windows based machine or windows virtual machine.

To communicate the DM with the rest of the system this module was developed to extract the relevant information from the Olympus messages and translate them into the format used by EDECAN, and vice versa. For reading and writing the Olympus messages we used the API provided with the Linux Olympus distribution.

Listing 5.14 shows an example of how a message transformed from the Olympus framework format to the EDECAN framework format looks.

### Data Backend

This module was also used for retrieving data from the backend database providing the bus lines information. This database is accessed when the DM wants to confirm the existence of the stop names requested by the user and when the DM wants to obtain the schedules of the routes requested by the user. The backend database differs from the one used in Section 4.1.4.

The database requests are as follows: As for the case of looking for confirmation of the existence of a stop the query to the database reduces to look for the stop name in a static local database. This database consists of plain text files containing the routes, stops and timetables that are provided to the module

```

{c systemutterance
:session_id "Default"
:properties { c properties
:dialog_act "inform"
:dialog_state "turn_number = 0
notify_prompts =
dialog_state = /LetsGoPublic
nonu_threshold = 49467.3203
stack = {
/LetsGoPublic
}
input_line_config = {
set_dtmf_len = 1,
set_lm = first_query
}"
:dialog_state_index "0"
:final_floor_status "free"
:id "DialogManager-0:000"
:frame "start
{
act inform
object how_to_get_help
_repeat_counter 0
Count = 0
}
end
"
:utt_count "0"
}
}

```

```

<package>
<command>nlg_run</command>
<frame>
start
{
act request
object how_may_i_help_you_directed
_repeat_counter 0
Count 0
}
end
</frame>
<Count>0</Count>
</package>

```

**Listing 5.14:** Example of a message in Olympus framework format (up) and of a message in EDECAN framework format (down) for the Campus SDS System.

in its service configuration file.

On the other hand, when requesting for the bus schedules, when an immediate date is required by the user the module makes a call to a Web Server provided

by the company responsible for the bus lines. This Web Server accepts HTTP GET or POST requests and provides the remaining time for a bus to pass through the origin place provided by the user and which destination is the one provided by the user. Listing 5.15 shows an example of the HTTP GET and POST format to use. Finally, when a non-immediate date is required the information is retrieved from the static local database.

<pre> REQUEST: GET /apps/danok/tqws/tq. asmx/GetPasoParada? strLinea=string&amp; strParada=string HTTP /1.1 Host: apli.bizkaia.net  RESPONSE: HTTP/1.1 200 OK Content-Type: text/xml; charset=utf-8 Content-Length: length  &lt;?xml version="1.0" encoding="utf-8"?&gt; &lt;string xmlns="http:// tempuri.org/ LANTIK_TQWS/TQ"&gt; string &lt;/string&gt; </pre>	<pre> REQUEST: POST /apps/danok/tqws/tq.asmx/GetPasoParada HTTP/1.1 Host: apli.bizkaia.net Content-Type: application/x-www-form- urlencoded Content-Length: length  strLinea=string&amp;strParada=string  RESPONSE: HTTP/1.1 200 OK Content-Type: text/xml; charset=utf-8 Content-Length: length  &lt;?xml version="1.0" encoding="utf-8"?&gt; &lt;string xmlns="http://tempuri.org/LANTIK_TQWS /TQ"&gt; string &lt;/string&gt; </pre>
---	--

**Listing 5.15:** Campus SDS System: Data Backend HTTP GET and POST example.

### 5.3.8 Controller

The main task of the controller is to control the flow of messages within the system receiving the output messages from all other modules and redirecting them to the appropriate module. As an example, when it receives a *tts.-end* message from the TTS module it redirects it to the DM interface to be sent to the DM to follow the dialogue flow. This module was developed from scratch.

The controller is also responsible for avoiding infinite dialogue loops caused by errors in the ASR process. In the messages received from the DM there is a parameter that indicates the number of consecutive executions of an action (for example a *request* for some information). Depending on the value of this parameter the controller will act as follows: If the value of this parameter is equal to a predefined number (three in our system) it will send a *dm.-nonunderstanding\_startover* message to the DM to make it know there have been too many recognition errors. The DM will react by asking the user to restart the dialogue or continue with it. On the other hand, in case of this

parameter being another predefined value higher than the previous one (six in our system) the controller will send a *dm\_nonunderstanding\_giveup* message. In this case the DM will react by finishing the dialogue established with the user after having informed the user of such a situation.

On the other hand, as has been said, the DM sends *inform* and *request* messages which are both associated with an audio synthesis to be played to the user. In both cases the DM is informed about the end of the synthesis, but in the case of the *request* message after the synthesis the ASR must be activated. This situation is managed by the controller by maintaining a variable with the value of the last message from the DM. In the case of this value being *request* the controller after sending a *tts\_end* message to the DM will send an *asr\_run* message to the ASR module, while the DM will be waiting for a *sp\_end* message. Instead, in the case of the value being *inform* the controller will send the *tts\_end* message to the DM which will continue with the next dialogue step.

---

#### 5.3.9 DateTime

DateTime interprets temporal expressions in user input and resolves them to absolute dates and times used internally by the DM. The module operates on fragments of the input parse and makes use of context information, such as the current time and date, to resolve relative expressions such as "tomorrow". The module also maps numeric expressions.

This module was developed by CMU and covers expressions in English, thus to integrate it into the system it was necessary to translate all language dependant strings to Spanish.

---

### 5.4 Experimental Evaluation

The primary goal of the experimental evaluation was to evaluate the performance, user experience and usability of the system. To this end, 30 users were selected to take part in the experiments. Users were not given any specific instructions on how to use the system; they were only informed that the University has a new automatic telephone service providing bus schedules of the lines connecting the Campus. On average each user made 3.37 calls. Table 5.3 shows the data related to the users when it comes to gender, age, the frequency of use of the bus lines from which the system offers information as well as the frequency of use of automatic and computer systems in general.

We first provide a preliminary evaluation of the system regarding the performance (Section 5.4.1) achieved by these users when interacting with the system. To this end we used the Task Completion (TC) and Average Dialogue Length (ADL) metrics. TC evaluates the success rate of the system in providing the user with the requested information, based on the total number of dialogues carried out and the number of successful dialogues achieved. ADL

Gender		Age		User of buses	
Male	60%	18-25	26.7%	Regular	40%
Female	40%	25-35	13.3%	Occasional	23.3%
		35-50	43.3%	Not user	36.7%
		50-65	13.3%		
		65+	3.3%		

Computer system user	
Regular	60%
Occasional	30%
Not user	10%

**Table 5.3:** Description of the main characteristics and profiles of the set of users taking part in the experiments

is the average number of turns in a successful task understanding turn as an intervention of either the system or the user.

Then the main User Experience evaluation is provided. The user experience was measured via a set of standardised questionnaires. To this end the System Usability Scale (SUS) [120] (Section 5.4.2), the Subjective Assessment of Speech System Interfaces (SASSI) [121] (Section 5.4.3) and the AttrakDiff 2 [122] (Section 5.4.4) questionnaires were used. A description of these questionnaires and their evaluation method can be found in Appendix D.

#### 5.4.1 System Performance

Table 5.4 shows the system performance evaluation in terms of TC and ADL values, *Full Queries* referring to the queries in which the user has to provide an origin, a destination and a time/date and *Previous-Next Queries* referring to the queries of the user asking for information relative to the previous query, according to the task description in Section 4.1.3. Listing 5.16 shows examples of both types of dialogues.

	TC	ADL
Full Queries	85.71%	12.52
Previous-Next Queries	100%	2.78
Total Queries	87.42%	NAN

**Table 5.4:** Task Completion and Average Dialogue Length of the user-system interaction metrics measured during the User Experience evaluation for the Campus SDS System.



Although Table 5.4 reveals good TC rates they could be improved mainly by improving the ASR and the SP. For the ASR, Google allow modifying neither acoustic nor language models so the only solution could be to use another ASR system that allows the customisation of these models, i.e. Kaldi [123], and train better models. Regarding the SP one possible improvement could be to add common recognition errors as valid parsing values. A clear example could be the word "UPV", that usually is recognised by the ASR as "ITV". It could be possible to add "ITV" as a synonymous for "UPV".

**FULL QUERY DIALOGUE:**

S: Welcome to the Bizkaibus line information system. Tell me where you are or where you want to go.

U: I'm at the university.

S: What is your destination?

U: I would like to go to Termibus.

S: What time?

U: As soon as possible, please.

S: One moment, please. There is an A2318 that leaves the university at 15:00. For more information on buses related to this trip you can say, when is the next bus or, when is the previous bus. To ask for a different trip, you can say, start a new search. When you are done, you can say goodbye.

**PREVIOUS\_NEXT QUERY DIALOGUE:**

U: What time is next?

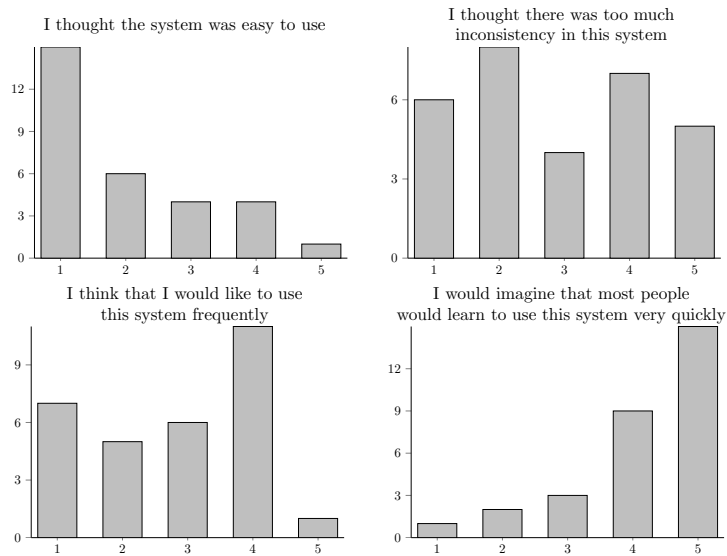
S: Just a moment. The next bus that leaves from the university towards Termibus is at 15:30.

**Listing 5.16:** Dialogue example for the Campus SDS System.

### 5.4.2 Usability

To get insights regarding the Quality of Experience of the system, i.e. the perception of the system on a general level, selected aspects of the taxonomy by Wechsung [124] were investigated. In detail, the usability, learnability, and intuitivity were measured with the SUS questionnaire. The questionnaire consists of a set of ten statements which users must grade their agreement on a scale from 1 to 5, being 5 the user to strongly agree with the sentence and 1 to strongly disagree.

From the answers to these questions the SUS score is obtained. SUS scores fall between 0 and 100, the higher the score the better the overall usability. Scores below 50 are typically considered as not acceptable [125]. Table 5.5 shows results obtained for a subset of the questionnaire. The SUS score was 67.75 (sd = 17.56). Hence, even though the perceived ease of use of the system is rather good, the overall usability may be improved.



**Table 5.5:** Results for a subset of the SUS questionnaire.

### 5.4.3 Speech Assessment

The SASSI questionnaire, developed to measure users' subjective experiences with speech recognition systems, was employed to examine the interaction quality. Users are given a series of statements, which they grade according to their agreement with them on a scale from 1 to 6. The analysis of the questionnaires provides developers with an assessment of the system along several axes such as easiness, friendliness, speed, etc. Scores are summarised in Figure 5.6, Tables 5.6 and 5.7 shows results obtained for "Habitability" and "Speed" respectively. The higher the value on a subscale the more participants agreed that the system performance was good. Results indicate that "Response Accuracy" (3.62), "Annoyance" (3.33) and "Habitability" (3.7) were judged as neutral, "Likeability" (4.02) as fair and "Cognitive Demand" (4.68) and "Speed" (4.47) good.

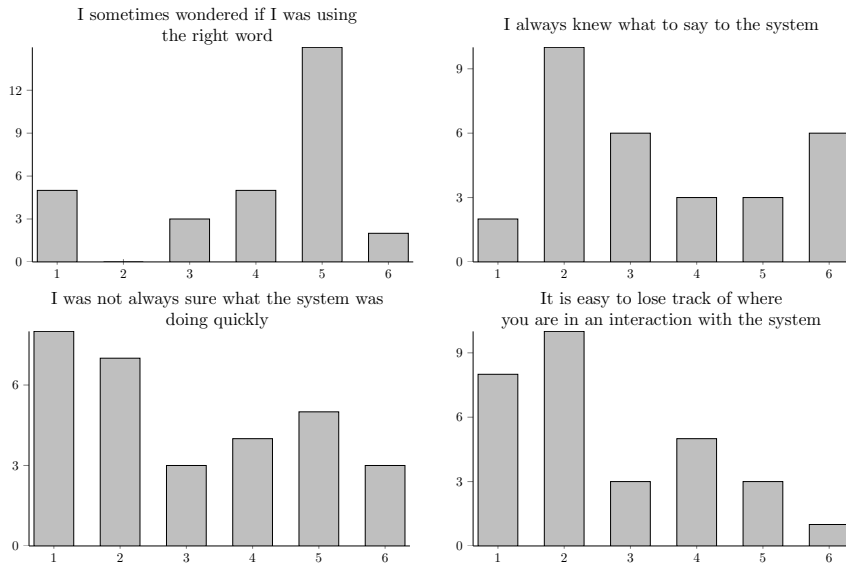


Table 5.6: SASSI questionnaire results: Habitability subset.

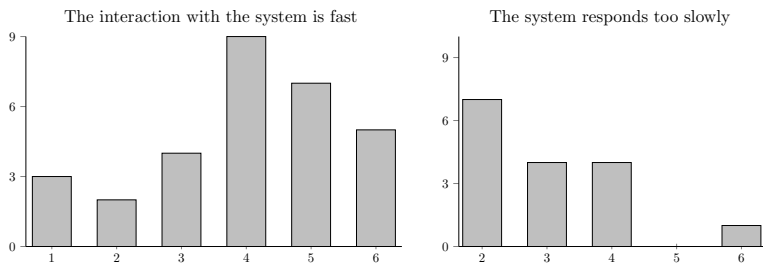


Table 5.7: SASSI questionnaire results: Speed subset.

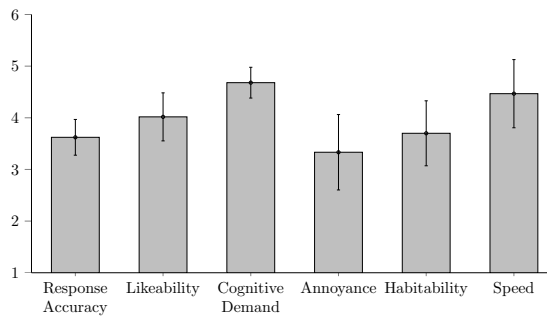


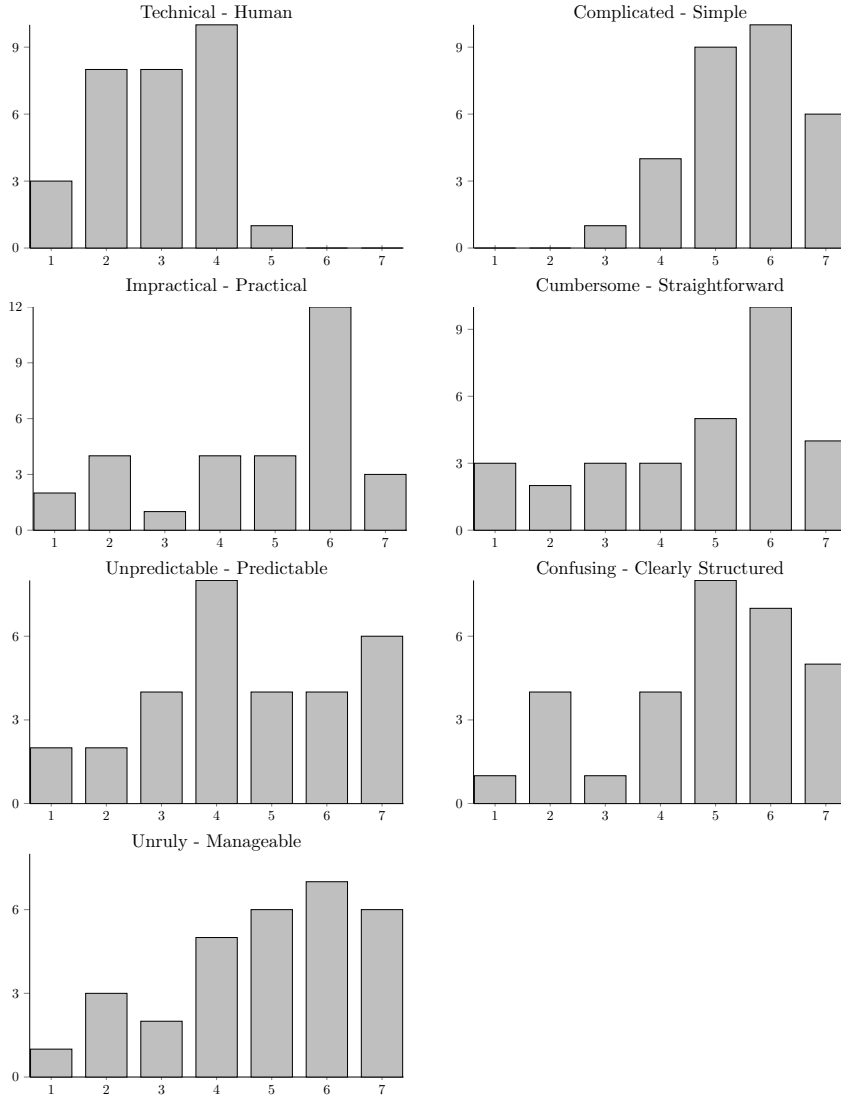
Figure 5.6: SASSI. Mean values and standard errors.

#### 5.4.4 Attractiveness

AttrakDiff 2 is a questionnaire aimed at measuring the user experience based on a model consisting of hedonic and pragmatic qualities of software products. Users are given a set of 28 questions which they grade on a scale from 1 to 7. Each of the aspects attractiveness, pragmatic quality, hedonic quality-stimulation, and hedonic quality-identity is represented by seven items. Some examples are: unpleasant-pleasant or ugly-pretty (attractiveness), complicated-simple or confusing-clear (pragmatic quality), conventional-original or conservative-innovative (hedonic quality-stimulation), lacking style-stylish or poor quality-high quality (hedonic quality-identity).

An aspect to take into account about the AttrakDiff 2 questionnaire is that it was designed to evaluate the attractiveness of software products in which graphical user interfaces are used. But given the aspects it measures (a description can be found in Appendix D), the author of this thesis understands that these are also found in other types of systems not using graphical interfaces, as is the case of the system being evaluated. So the author proposes the use of AttrakDiff 2 questionnaire to evaluate speech-based systems.

Figure 5.7 shows the mean scores for each of the four subscales. Tables 5.8 and 5.9 shows results for pragmatic quality and hedonic quality-identity respectively. This test shows that the users find in general the system pleasant and easy to use as expected. What surprises most of an automatic telephone information system is that the user does not give it very low scores in hedonic quality-stimulation and hedonic quality-identity subscales. This fact is probably due to the lack of quality in the automatic information systems the participants know and use (rerouting, ticketing, weather information, etc.)



**Table 5.8:** Attrakdiff 2 (Pragmatic Quality) results.

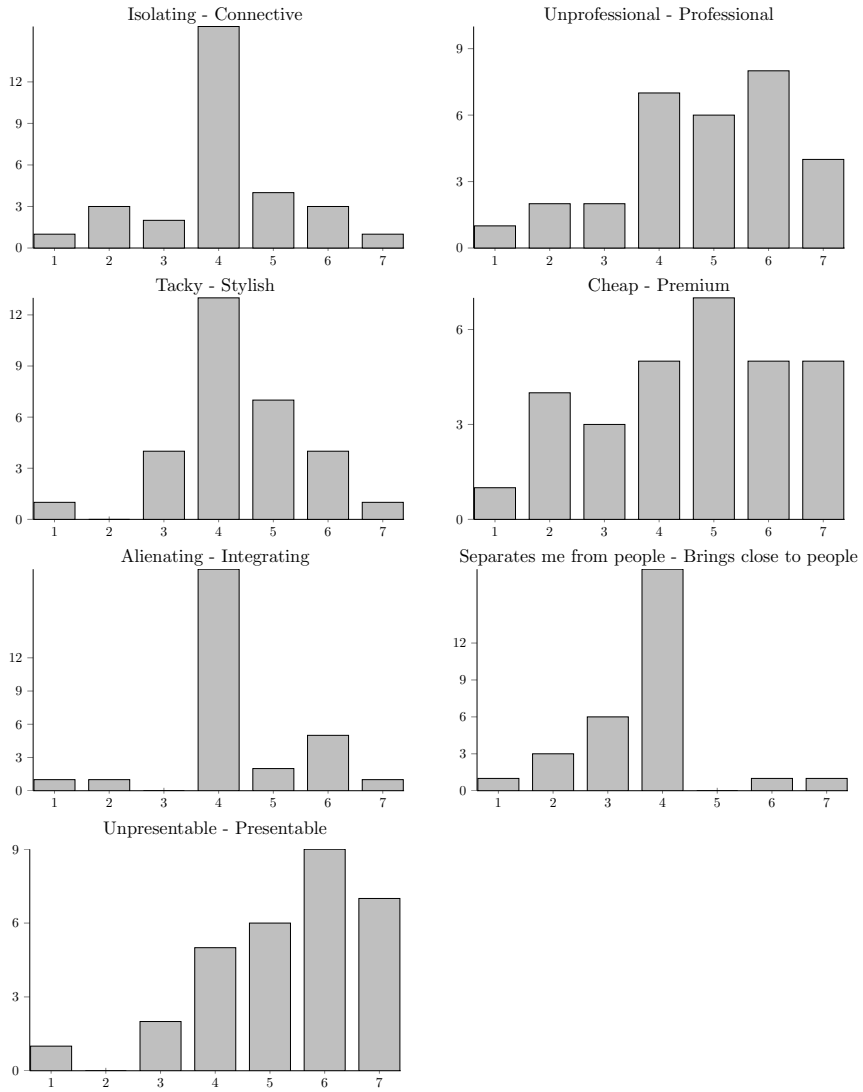
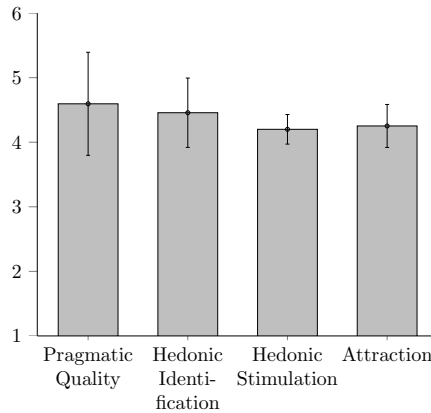


Table 5.9: Attrakdiff 2 (Hedonic Quality-Identity) results.



**Figure 5.7:** AttrakDiff 2. Mean values and standard errors.

## 6 Concluding Remarks

We have created in this chapter a Phone Bus Information SDS in Spanish similar to Let's Go under the Unix-based EDECAN framework. To this end, we have integrated the RavenClaw DM in EDECAN framework by a communication module that makes this DM available to work in the Unix environment. The system provides schedules and stops information about some regional municipalities connecting the Campus. A preliminary system performance evaluation involving a reduced set of initial users showed good task completion rates. The user experience has been evaluated through a set of standardised questionnaires including SUS, SASSI and AttrakDiff 2, resulting in a quite good user experience regarding usability and good scores in some SASSI and attractiveness aspects. Neutral scores will be improved with some work on Language Understanding modules.

# 6

## SYSTEMS BASED ON ACTIVEMQ

---

### 1 Introduction

---

In the previous chapter it has been built a system that allows being adapted to different tasks. In the system the RavenClaw DM has been used. From the knowledge acquired about this DM, it was found to be adequate for its adaptation to different tasks and at the same time for the implementation of complex tasks. To confirm these two hypotheses two additional works including said DM have been carried out. For them, the ActiveMQ [84] messaging broker has been used. ActiveMQ is a messaging broker that allows the interconnection of different services and applications via a message interchanging mechanism that consists of a set of queues in which the different services can read from and write to. When a service wants to send a message to another service, the first writes the message in a queue and the second reads it from the same queue.

In Section 2 the RavenClaw DM has been tested within the vAssist SDS system [126]. For that, the DISCO-based DM used in the vAssist system has been replaced with the RavenClaw DM. Additionally, the vAssist system has been translated to the Spanish language. To compare the performance of both DMs an experimental evaluation has been carried out. Section 3 describes the preliminary steps on the development of a personal assistant humanoid robot. More precisely, the dialogue system design for an anthropomorphic assistant robot, Roberta, in which the RavenClaw DM has been integrated.

### 2 vAssist

---

This section presents some works carried out with the vAssist SDS system, a system developed under the vAssist project [127].

The project aims to provide voice-controlled Home Care and Communication Services for seniors suffering from chronic diseases and persons suffering from motor skills impairments. The primary goal is the development of simplified



and adapted interface variants for telemedical and communication applications using multilingual natural speech and voice interaction. vAssist aims to enhance the perceived quality of healthcare services and to enable a reduction in the costs related to their production and delivery by achieving channel independence in the delivery of vAssist services, so that existing hardware and interfaces in the home of the users can be used such as PC, TV, mobile phone or tablets.

The vAssist SDS system is based on an open and adaptive software architecture that allows for an easy configuration of SDSs according to a given target scenario, user requirements and context. It's based on the Disco plan-based DM [128] and the Linked Form Filling (LFF) task model. The works carried out consisted in, on the one hand, extending the system to integrate the Spanish language (initially English, French, German and Italian languages were supported), and, on the other hand, integrating an alternative DM (CMU's RavenClaw) for comparison purposes.

A multilingual lab evaluation of the final vAssist assistive home care and communication service applications running on a smart-phone was carried out. Such was carried out with real users in Austria, France and Spain. The evaluation was carried out in terms of system performance. On the other hand the experimental comparison of the Disco-LFF DM and the RavenClaw DM working within the same SDS architecture, dealing with the same task and language (i.e. Spanish), and interacting with the same users, was carried out.

This section first will describe the vAssist SDS system architecture, then the implementation details for the new language integration and DM replacement will be given and finally the results of the experimental evaluation will be given.

## 2.1 Architecture

The vAssist SDS extends the usual chained design (i.e. ASR + NLU + DM + NLG + TTS). Components were split into sub-modules and new processes were integrated into a state-of-the-art workflow chain. Figure 6.1 shows the resulting SDS architecture.

### 2.1.1 Speech Recognition

The system uses the Google Speech API where an HTTP POST request transmits the signal segment to be recognised. The API returns the n-best hypotheses, being n a parameter of the request, as well as the confidence score for the best one. An empty result is returned when the segment cannot be recognised with enough confidence, i.e. when it does not contain speech.

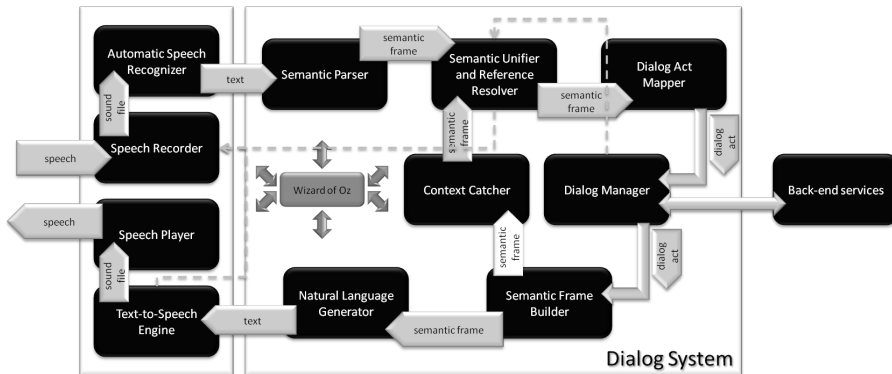


Figure 6.1: vAssist: System architecture.

### 2.1.2 Semantic Parsing

The SP, which gets inputs from the ASR, associates semantic labels to text utterances (or parts of them). The most commonly used parsing techniques are based on context-free grammars or probabilistic context-free grammars, which are either hand-coded, based on the analysis of collected dialogue data, or designed by experts. It integrates the algorithm proposed by [129], which is the application of the work from [130]. Instead of matching whole sentences with parse structures, the algorithm looks for patterns in chunks of the text-level utterance and the temporary (i.e. currently assigned) Semantic Frame (SF). The module applies an ordered set of conditional rules, which is learned from data.

### 2.1.3 Natural Language Generation

A simple but effective solution to produce natural language utterances conveying the DM's messages was targeted. Input messages are SFs. The engine is fed with a set of templates that consist of a title (identical to an SF's goal) associated with an utterance, and whose parts may be replaced by slot names or slot name-value pairs. The result is a natural language utterance to be synthesised or displayed on a screen.

### 2.1.4 Text to Speech

MaryTTS [131], an open-source speech synthesis framework maintained by the Cluster of Excellence MMCI and the DFKI, is used for synthesis. It offers pre-built voice models for different languages as well as tools to create and manipulate them. The MaryTTS module is a client connected to a generating server (hosted local or remote). A request containing the text to be synthesised with additional prosodic information is sent to the central server, which returns

the speech stream. The text-to-speech module of the present platform is a basic client program embedded into an ActiveMQ wrapper.

### **2.1.5** Semantic Unification and Resolution

---

The Semantic Unifier and Reference Resolver (SURR) holds a rather simplistic forest of nodes which is used to mine the dialogue history, incorporate external information sources and add local turn context. It's the meeting point of the user's semantic frames, the external environment sensors and functions, the dialogue history, and the links generated by the context catcher.

At its core the SURR embeds a forest structure. Trees consist of hierarchies of fully or partially defined SFs (some nodes are calls to external systems or services). When requested, the SURR may dynamically modify (remove/add) branches of the forest. The top node of a hierarchy defines the root.

The SURR algorithm tries to find a unique path from an input SF, i.e. from the parsed user input, to nodes of the forest, to a root node. Going up the trees, the algorithm applies the optional operations held on branches.

Reaching a root node equals the user input being contextualized [132]. In case the algorithm cannot find such a path, i.e. the SURR fails to produce a suitable SF (given the current context and available knowledge), a "NoMap" SF is generated to signal a 'non-understanding' to consecutive components.

### **2.1.6** Dialogue Act Mapping

---

As a last stage of the NLU processing, the dialogue act mapping is performed. Once an input has been parsed, external and local references have been resolved, and the semantic level has been unified, the ultimate step is to convert the SF into a Dialogue Act (DA). Following an input, the mapper retrieves a set of available DAs. Then it looks for a unique match between the SF and the set of DAs.

### **2.1.7** Dialogue Management

---

The core of the implemented DM is based on Disco [133], an open-source dialogue management library, whose algorithm processes task hierarchy models. A dialogue model is a constrained XML tree of tasks. The plan recogniser uses the recipes defined in the dialogue models and this dialogue state to select the best available plans for the tasks in the stack. Then the reasoning engine selects the most appropriate next step.

In an attempt to overcome the hurdles inherent to the specification of task models, the dialogue modelling paradigm was shifted to a LFF one. Form-filling dialogues are based on structures containing sets of fields which the user needs to provide a value for to trigger a terminal action. The order in

which the DM asks for the values is not predefined. The user may define multiple field values within a single utterance/turn.

The LFF language offers to combine these properties with the ability to trigger actions at any point of the dialogue and the inclusion of subforms. Furthermore, fields and subforms can be optional, i.e. either be ignored when unset or proposed to the user. Here, the unlimited depth of a task model to circle tasks while keeping a sequencing order was used; i.e. the link between two task nodes is a reference, hence a node can point to its "parent" node.

The aim of the LFF language is to offer a somehow simpler design method to a powerful standard dialogue modelling specification. Since it's also an XML based language we opted for XSLT to convert an LFF document into a compliant dialogue model.

A number of rules were defined to create a well-formed LFF document. Doing this, the relative reduction in terms of code size and task hierarchy depth was 76% and 77%, respectively.

---

## 2.2 Adding Languages and RavenClaw DM Integration

---

### 2.2.1 Adding Languages

The addition of a new language (Spanish in this case) consisted of two steps. On the one hand, the creation of the NLG templates, and on the other hand the creation of the SP rules.

#### NLG Templates

The creation of the NLG templates is very straightforward. The NLG generates the sentences from two parameters: A dialogue act and a topic associated with such dialogue act. When the NLG receives a message it extracts these two parameters and searches in the templates file for a sentence matching the two parameters. If there are several sentences for the same parameters a random one is chosen.

As an example, when receiving the following message:

*Ask.What : slot = drug;*

the NLG will search for a sentence matching the *Ask.What* dialogue act and the *drug* topic generating the following sentence:

*¿Cómo se llama el medicamento que desea añadir?  
(What is the name of the drug you want to add?)*

Listing 6.1 shows a reduced example of the final file.

```

Ask.What:Qué prescripción desea cambiar? [/slot=prescription_id/]
Ask.What:Qué prescripción desea modificar? [/slot=prescription_id/]
Ask.What:Desea suprimir? o modificar la prescripción [/slot=type/]
Ask.What:Cómo se llama el medicamento que desea añadir? [/slot=drug/]
Ask.What:A partir de cuándo debe tomar el medicamento? [/slot=from/]
Ask.What:Durante cuánto tiempo tiene que tomar el medicamento? [/slot=
prescription_period/]
Ask.What:Qué periodo de análisis desea? [/slot=sleep_analysis_period/]
Ask.What:Quiere usted confirmar? [/slot=confirmation/]
Ask.What:Son correctos los datos? [/slot=confirmation/]
Ask.What:Qué información quiere cambiar? [/slot=field/]
Ask.What:Qué información quiere modificar? [/slot=field/]
NoMap:Lo siento, Le importaría repetir?
NoMap:Entendí /reco/ pero no puedo relacionarlo con el contexto actual
NoMap:Entendí /reco/ pero esa acción no está disponible
NoMap2:Su petición no está disponible. Se puede decir nuevo medicamento o enviar un
mensaje. [/goal=undefined/]
NoMap2:Puede decir buenos días o presión sanguínea [/goal=undefined/]
Feedback:Su mensaje ha sido enviado [/dialog=message/]
Feedback:El mensaje ha sido enviado [/dialog=message/]
Feedback:Se ha enviado su mensaje [/dialog=message/]
Feedback:Su tensión ha sido registrada [/dialog=bloodpressure/]
Feedback:Se ha registrado su tensión [/dialog=bloodpressure/]

```

**Listing 6.1:** vAssist SDS: NLG template file example.

## SP Rules

The process of creating the rules used by the SP consisted of the creation of a database consisting of sentence/sp-label pairs. Subsequently, through an automatic process, this database was analysed and, from the analysis, the rules were extracted. Listing 6.2 shows a reduced example of the file used.

### 2.2.2 RavenClaw DM Integration

To integrate RavenClaw in the architecture shown in Figure 6.1, the original Disco-LFF DM was substituted. As in the case of previous systems using RavenClaw, it was used as a service of its native architecture, Olympus. A module responsible for translating the message format defined by RavenClaw to the message format defined by the Disco-based component and vice versa was developed. On the other hand, the dialogue model trees used by the Disco-LFF DM were replicated within RavenClaw.

## 2.3 Experimental Evaluation

### 2.3.1 Task and Experimental Scenarios

To empirically evaluate the operation of the developed voice-controlled application running on a smartphone under standardised condition, several sce-

```

3 meses <=> input(number=3,period=meses)
1 año <=> input(number=1,period=año)
desayuno en casa <=> input(intake_time=desayuno)
comprimidos <=> input(dosage_form=comprimidos)
a partir del 2 de noviembre <=> input(month_name=noviembre,day_number=2)
lunes 17 noviembre <=> input(month_name=noviembre,day_number=17,day=lunes)
martes 18 noviembre <=> input(month_name=noviembre,day_number=18,day=martes)
a las 9h 25 <=> input(wake_up_time=9h 25)
dos horas <=> input(hour=dos)
mañana <=> input(relative_date=mañana)
dos pastillas <=> input(dosage_form=pastillas)

```

**Listing 6.2:** vAssist SDS: SP training file example.

narios were defined and implemented. In detail, the following scenarios and associated tasks were applied for the experimental study:

- The Prescription Management enables to monitor medical prescriptions and individual intake times. To evaluate this scenario, participants were asked to add a new predefined prescription to the application database and to set a reminder for it (AP). The application requests information regarding name of medication, quantity, dosage form, frequency, and time of intake.
- The Health Report (HR) provides an overview of physiological data. Participants filled in predefined glycaemia and blood pressure data.
- The Sleep Report (SR) monitors sleep quality. The following data was provided by the users: the time he/she went to bed, the time he/she fell asleep, and their wake-up times. Participants also reported awake periods at night and the total number of hours slept. Finally, users were asked to rate their well-being on a six-point scale. Furthermore, the evaluation included setting a reminder to remember completing the sleep report (SRR).
- Fitness Data Management consists in reporting daily activities (FD) and setting reminders for the reports. Within the evaluation, participants were asked to enter a new report including the duration of their fitness activity.
- The Communication Services include sending messages (SM) and initiating phone calls (PC). Participants were asked to test both functions.

### 2.3.2 Experimental Evaluation

Two series of experiments were carried out: We evaluated the vAssist system including the Disco-LFF engine in three languages: French, German and Spanish. Further, we compared the RavenClaw and Disco-LFF DMs built into the vAssist system with Spanish users.

Sixteen users took part in the experiments in each of the trial sites. In France,

14 male and 2 female persons between 65 and 90 years (mn=77.0) participated in the study. In Austria, 8 male and 8 female participants between 60 and 76 (Mn=68.0) years old took part. The Spanish trial site included 12 males and 4 females between 26 and 69 (Mn=39.6) years.

Users were first shown the smartphone application, followed by a short demonstration and usage advices. The experimental scenarios were then carried out without any other introduction than the simple description of the goal. It was up to the user to figure out how to perform each task.

The system’s performance was measured in terms of TC, i.e. success rate, and ADL, i.e. efficiency. TC evaluates the success rate of the system in providing the user with the requested information, based on the total number of dialogues carried out and the number of successful dialogues achieved for a specific task. ADL is the average number of turns in a successful task.

### 2.3.2.1 System Performance

The first series of experiments was carried out in France, Austria and Spain, evaluating the vAssist system with the Disco-LFF DM. Table 6.1 shows the system performance evaluation in terms of TC and ADL values.

	French		German		Spanish	
	TC	ADL	TC	ADL	TC	ADL
AP	93.33%	8.00	88.88%	8.18	84.00%	13.62
HR	100.00%	3.15	93.33%	3.78	100.00%	4.41
SR	91.66%	7.81	100.00%	7.25	100.00%	10.18
SRR	83.33%	3.40	100.00%	3.50	87.50%	5.78
FD	100.00%	3.00	66.66%	3.00	93.75%	4.53
SM	100.00%	3.86	100.00%	4.62	100.00%	6.21
PC	100.00%	1.92	100.00%	1.82	100.00%	2.00
Average	97.12%	4.44	95.18%	4.73	92.19%	6.21

**Table 6.1:** TC and ADL of the vAssist system using the Disco-LFF DM.

Table 6.1 reveals good TC rates, with the French version being the one generating the highest system performance and the Spanish version the one producing the lowest. Surprisingly, the results show that the vAssist system performance is not better for younger users (Spain: mn=39.6 years) than for older ones (France: mn=77 years). Language dependent modules, i.e. the ASR and, more importantly, the NLU, were more robust in French and German. Spanish results suffered from a less robust semantic parser and the missing mobile UI, leading to a higher number of turns to achieve the task goals.

### 2.3.2.2 Disco-LFF and RavenClaw DM Comparison

The second series of experiments was carried out in Spanish only. Note that both DMs were integrated in the same architecture (Figure 6.1), i.e. only the task planning and the agent execution differed. Each user carried out the

	Disco-LFF DM		RavenClaw DM	
	TC	ADL	TC	ADL
AP	84.00%	13.62	94.40%	15.64
HR	100.00%	4.41	100.00%	4.90
SR	100.00%	10.18	83.30%	11.90
SRR	87.50%	5.78	75.00%	6.08
FD	93.75%	4.53	92.80%	4.30
SM	100.00%	6.21	100.00%	6.64
PC	100.00%	2.00	100.00%	2.42
Average	92.19%	6.21	89.90%	6.60

**Table 6.2:** Comparing the Disco-LFF and RavenClaw DMs.

scenarios defined in Section 2.3.1 with either of the DMs. Table 6.2 shows the system performance achieved by both systems in terms of TC and ADL, for each of the defined subscenarios. Both metrics show similar behaviour for the Disco-LFF and the RavenClaw DM. A Z-test comparing the average TC proportions and the ADL means showed no statistically significant difference between the two DMs ( $p$ -value = 0.05). A detailed scenario-based analysis showed, however, differences between TC values in the AP and the SR scenarios, which correspond to longer dialogues in terms of the ADL metric. A previous series of experiments has furthermore highlighted a certain lack of robustness exhibited by the language dependent modules of the Spanish vAssist version. This issue was more evident in longer dialogues (AP and SR).

### 3 Personal Assistant Robot Dialogue System

This section describes work on dialogue system design for personal assistant humanoid robots. It was carried out during eNTERFACE'16 [134] workshop, a workshop organized around several research projects dealing with multimodal human-machine interfaces design. During the project, the dialogue system design for an anthropomorphic assistant robot, Roberta, was developed. The focus is on the first stage of LifeLine Dialogues, meaning conversational creation of stories about a user's life. The goal of the project was to help the elderly record semi-autobiographical narratives while combating the deterioration of memory and speech abilities. Modelling of dialogue scenarios for first-time users were completed. This allows Roberta to personalize future conversations based on each user's place of origin, work and education history, and hobbies, which are all information gathered during a user's first conversation with Roberta. This was accomplished through, on the one hand, an adaptable dialogue system with topic administration and multi-modal functionalities, specifically face recognition, by extending RavenClaw-type management framework, and on the other hand, the customization of user relationship with Roberta by introducing contextual data, conversation history, and emotion



awareness.

The long-term plan is for both users and Roberta to initiate topics for conversations, fight memory lapse and speech deterioration of users, and ultimately for Roberta to become a companion, not solely an assistant. The initial focus and the topic of this section is on creating a dialogue management system for intuitive interaction possibilities with Roberta for first-time users. During the first meeting, Roberta is able to set up a conversation history per user, ask for critical information about the user's life to begin a user's LifeLine in a conversational manner, and gather visual data from the environment, a face to be precise, to detect the user after his/her first encounter.

### 3.1 Architecture

The system developed consists of an input perception layer, an understanding module, an interaction module that includes the dialogue manager which also manages a user blackboard, a dialogue history blackboard and an open domain conversational agent that gets information from a digital repository and a final output layer. Figure 6.2 shows the architecture of the system.

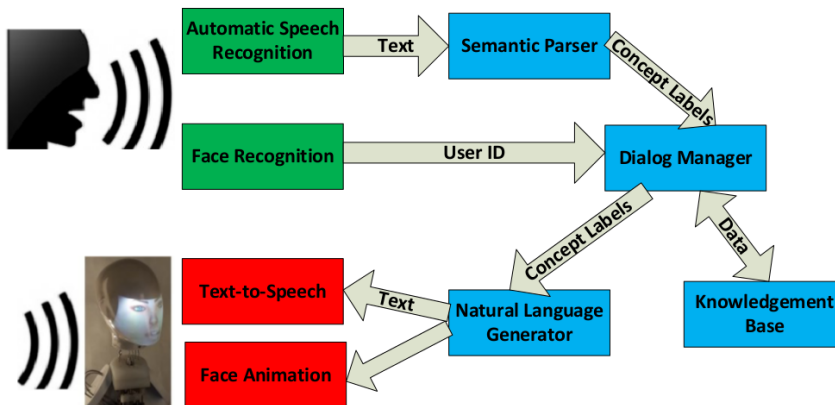


Figure 6.2: Roberta: System architecture.

As has been said, this architecture was implemented using ActiveMQ which makes possible the interconnection of different services and applications via a message interchanging mechanism that consists of a set of queues in which the different services can read from and write to. Figure 6.3 shows a schematic of the interconnection of the different modules.

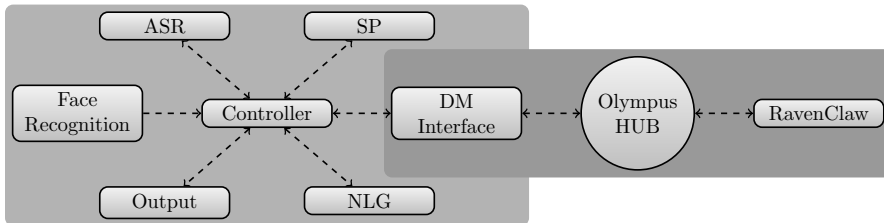
As can be seen, besides having the different system modules interconnected via ActiveMQ queues, for the DM Interface we have that it's also connected to the CMU Olympus HUB which at the same time is connected to the RavenClaw DM. As for the case of the Bus Information System presented in Chapter 5, the

```
wget http://www.apache.org/dyn/closer.cgi?filename=/activemq/5.14.5/apache-
activemq-5.14.5-bin.tar.gz
tar -xvzf apache-activemq-5.14.5-bin.tar.gz

./apache-activemq-5.14.5/bin/linux-x86-64/activemq
```

**Listing 6.3:** Installing and running ActiveMQ.

communication with the RavenClaw DM had to be done via its communication manager, and so, it was necessary to have a DM Interface connected to both ActiveMQ framework and Olympus framework at the same time acting as a message translator between both frameworks.



**Figure 6.3:** Roberta: Architecture implementation with ActiveMQ.

On the other hand, it has been chosen to add a Controller module to act as a message routing manager. It receives the output messages from all the other modules and depending on the input message decides the order of execution of the rest of the modules. As an example, when it receives an input message from the ASR it extracts the content of the message (the recognition hypotheses) and sends it to the SP.

## 3.2 Configuring and Launching the System

This section explains how to configure and launch all the system modules. On the one hand, it will be necessary to launch the services connected to the ActiveMQ broker, and on the other hand, the RavenClaw DM that is connected to the Olympus framework.

### 3.2.1 Configuring and Launching ActiveMQ Services

Before launching the ActiveMQ services it will be necessary to install and run the ActiveMQ broker itself. Listing 6.3 shows how to do that.

Once the ActiveMQ broker is installed and running, the next step is to run the different modules. It's done like shown in Listing 6.4. For each of the

```

java -jar DMINTERFACE.jar failover://tcp://localhost:61616 DMIN CONTROLIN
java -jar ASR.jar 16000 1 16 256 plughw:0,0 en-US GOOGLE_SPEECH_API_KEY
  failover://tcp://localhost:61616 ASRIN CONTROLIN
java -jar SP.jar data/Grammar roberta.net failover://tcp://localhost:61616 SPIN
  CONTROLIN
java -jar NLG.jar data/rosetta/perl/main.pm failover://tcp://localhost:61616 NLGIN
  CONTROLIN
java -jar jar/CONTROL.jar failover://tcp://localhost:61616 CONTROLIN
java -jar jar/FACERECOGNIZER.jar failover://tcp://localhost:61616 CONTROLIN

```

**Listing 6.4:** Roberta SDS: Commands to run the ActiveMQ modules.

```

{
  "activemq-queue_in": "TTSIN",
  "activemq-queue_out": "CONTROLIN",
  "uri": "http://localhost:59125/process",
}

```

**Listing 6.5:** Roberta SDS: Output module configuration file.

modules the last three arguments are, the URL where the ActiveMQ broker is installed, the name of the queue from which the module reads the input messages and the name of the queue to which the module writes the output messages. Alternatively, other arguments are provided. For example, in the case of the SP, it's provided with the directory where the grammar files are located and the name of the grammar net to use.

Of all the modules, the Output module is still not running. It's started with the following command:

```
./roberta
```

This executable file reads a JSON configuration file where the location of the ActiveMQ broker, the input queue to read messages from and the output queue to write messages to are specified. Listing 6.5 shows this file.

### 3.2.2 Configuring and Launching Olympus Services

Olympus framework consists of a communication manager (called HUB) that connects with different services and acts as a message router between them. To make all the components work, the communication manager on the one hand, and the services on the other hand must be launched.

To run the communication manager:

```
CMU_HUB -pgm_file hubcfg.pgm
```

Listing 6.6 shows the hubcfg.pgm configuration file. This file specifies the services that will be connected to the HUB by specifying its ports and IPs,

```

SERVICE_TYPE: DialogManager
SERVICE_PROVIDER: DialogManager
PROVIDER_ID: [RavenClaw]
OPERATIONS: reinitialize begin_session handle_event start_inactivity_timeout
             cancel_inactivity_timeout end_session
LOCATION: localhost:17000

SERVICE_TYPE: VAssistRavenclawServer
SERVICE_PROVIDER: VAssistRavenclawServer
PROVIDER_ID: [VAssistRavenclawServer]
OPERATIONS: reinitialize systemutterance
LOCATION: localhost:1201

SERVER: Builtin
OPERATIONS: new_session end_session increment_utterance nop
MODE: singlethread

RULE: :startutt --> Builtin.increment_utterance

RULE: :action_type == "system_utterance" --> VAssistRavenclawServer.
      systemutterance

IN: :properties

```

**Listing 6.6:** Roberta SDS: Olympus HUB configuration file.

also specifies the commands that can be sent to the respective services. On the other it also specifies some rules to execute depending on the type of messages that the HUB receives.

On the other hand, for running the services (in this case, just the RavenClaw DM):

*wine DM.exe*

### 3.3 Modules Implementation Details

#### 3.3.1 ActiveMQ Services Programming

The system modules were written in Java language code. ActiveMQ provides a Java API for programming the services. Listing 6.7 shows a template of how an ActiveMQ client should be written. In the constructor class all the initializations are done. Mainly, the input and output queues are created.

On the other hand, in the *run()* method is where all the logic of the service occurs. It consists of a loop in which the service loads the input messages, processes them, and generates the output messages.

**3.3.2** Automatic Speech Recognition

For the ASR, again, Google ASR was used. An external application written in C language code was written responsible for acquiring the user's speech input, sending it to Google's servers, and retrieving the recognition hypotheses. This application was included in the ActiveMQ service via a system call. Regarding the procedure of sending the audio signal to Google servers and recognition hypotheses retrieval, what was said in Section 5.3.1 of Chapter 5 is applicable here as well.

```
public class EXAMPLECLASS implements Runnable {
    static String URL, queue_in, queue_out;
    static Session session;
    static Destination destination_in, destination_out;
    static MessageConsumer consumer;
    static MessageProducer producer;
    static ConnectionFactory connectionFactory;
    static Connection connection;

    EXAMPLECLASS(String[] args) throws JMSEException {
        URL = args[2];
        queue_in = args[3];
        queue_out = args[4];

        connectionFactory = new ActiveMQConnectionFactory(URL);
        connection = connectionFactory.createConnection();
        connection.start();

        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

        destination_in = session.createQueue(queue_in);
        destination_out = session.createQueue(queue_out);

        consumer = session.createConsumer(destination_in);
        producer = session.createProducer(destination_out);
    }

    public void run() {
        TextMessage out;

        while (true) {
            try {
                Message message = consumer.receive();
                String text = ((TextMessage) message).getText();

                ... DO SOMETHING ...

                out = session.createTextMessage(output_message);
                producer.send(out);
            }
            catch (Exception ioe) {
            }
        }
    }
}
```

**Listing 6.7:** ActiveMQ service structure.

### 3.3.3 Semantic Parser

---

As for the SP, CMU's Phoenix was used. Again, like in the case of the ASR an external application was developed and then integrated into the ActiveMQ service via a system call. This external application was a slightly modified version of the one distributed by CMU.

On the other hand, regarding the Phoenix grammars, a dataset of sentences was collected, analysed and transformed in grammar rules and concepts. This dataset set was obtained from 15 participants and gave some initial constructs and vocabulary to work with. Functions used in the parser are "name", "place", "yes", "no", "age", "ask feeling", "ask age", "ask name", "ask place", "and you", "ask occupation", "occupation", "occupation academic", "no occupation", "work topic", "work place", "work passion love", and "work passion hate".

### 3.3.4 Natural Language Generator

---

CMU's Rosetta [118] was used for NLG. As already has been said it consists of a set of Perl language files. Its integration into the service was, again, done via a system call. On the other hand, the corresponding sentence generation templates had to be written.

### 3.3.5 Output

---

The goal of this service is to provide an output consisting of a facial animation in sync with the TTS. For the facial animation Unity game engine was used and for the TTS Cerevoice [135] was used.

GRFL's Base Liza 3D model with meshed viseme animations [136] were used. This saved time since it included animation clips for facial expressions and pre-generated viseme articulations. The phonemes served by the Cerevoice text-to-speech instance are mapped to their matching visemes for the English language and synchronized with the 3D animations and the synthesized audio.

### 3.3.6 Face recognition

---

The purpose of this module is to initialize a new dialogue each time it recognizes a user is present. Additionally, it can recognize the identity of the user. At the time of having detected a user it will send a message directed to the DM to start a new dialogue. This message will include the user's identity with which the DM will start a different kind of dialogue. These dialogues can be questions about the user's life, work and hobbies in the case of being a new user of the system, or, questions about these same data for a known user.

To identify the user, 2D face images were extracted from a video. To this end, the SudFrog (Telecom SudParis Face Recognition Open-source using Gabor

Filters) software [137] was adapted and integrated into the system.

The scenario and data flow is the following: a set of known subjects were enrolled with their faces. The enrolment phase consists of face and landmarks detection to calculate the eyes and mouth positions to create the biometric template of the person. This will represent the subject for further comparisons, to perform the identification phase. Therefore, first a face detection algorithm is running. When it detects a face, the module will extract an image of the face, and next step is to perform the geometric and illumination normalizations. From this normalized image, the Gabor features are extracted. This features and next compared to the existing templates recorded during previous enrolments, and the identifier of the most similar subject is passed to the DM. The DM continues the process by asking if it's the person or not. Regarding the answer the DM will choose the dialogue strategy.

### 3.3.7 Dialogue Manager Interface

---

The DM used in the system was CMU's RavenClaw [75]. As for the case of the Bus Information System in Chapter 5 it was decided to integrate RavenClaw running in its framework, Olympus, and thus it was necessary to implement a module in order to extract the relevant information and to translate the format of the messages used within the Olympus framework to the format used within this system, and vice versa.

Listing 6.8 shows an example of how a message transformed from the Olympus framework format to the format used within this system looks.

### 3.3.8 Controller

---

The purpose of this module is to receive the output messages from all the other modules in the system and redistribute them to the appropriate module. As an example, when it receives a message from the ASR with the recognition hypotheses it will redirect this message to the SP.

On the other hand, as was explained in Chapter 5, the DM sends *inform* and *request* messages which are both associated with an audio synthesis to be played to the user. In both cases the DM is informed about the end of the synthesis, but in the case of the *request* message after the synthesis the ASR must be activated. This situation is managed by the controller by maintaining a variable with the value of the last message from the DM. In the case of this value being *request* the controller after receiving a message informing about the end of the audio synthesis will send a message to the ASR to activate it. Instead, in the case of the value being *inform* the controller will just send a message to the DM, informing about the end of the audio synthesis, which will continue with the next dialogue step.

```

{c systemutterance
:session_id "Default"
:properties { c properties
:dialog_act "inform"
:dialog_state "turn_number = 0
notify_prompts =
dialog_state = /LetsGoPublic
nonu_threshold = 49467.3203
stack = {
/LetsGoPublic
}
input_line_config = {
set_dtmf_len = 1,
set_lm = first_query
}"
:dialog_state_index "0"
:final_floor_status "free"
:id "DialogManager-0:000"
:inframe "start
{
act inform
object how_to_get_help
_repeat_counter 0
Count = 0
}
end
"
:utt_count "0"
}
}

```

```

inform#/start
{
act request
object how_may_i_help_you_directed
_repeat_counter 0
Count 0
}
end

```

**Listing 6.8:** Example of a message in Olympus framework format (up) and of a message in ActiveMQ framework format (down) for the Roberta SDS System.

### 3.3.9 RavenClaw Dialogue Manager

As for the DM CMU's RavenClaw was used. The design of the dialogue strategy to implement in RavenClaw was extracted from the analysis of a previously created dialogues corpus acquired via the WOZ technique. The WOZ prototyping platform used can be found on <sup>1</sup>, it's specifically for designing interactions based on natural language processing [138]. The data collection method consisted of the wizard managing all dialogues. Beforehand, possible

<sup>1</sup><https://github.com/stephanschloegl/WebWOZ>



utterances that the wizard can make the system say to a participant were created. For unexpected participant responses, the wizard provided utterances ad hoc, and these ad hoc instances were considered and incorporated as needed in building the DM. Even though the greater goal for Roberta is to offer open domain dialogues, the first stage of development was directed towards limited scenarios that mimic initial conversations between two strangers. The corpus was created with 15 participants to collect interaction data and user strategies covering four initial preidentified topics i.e. (1) introduction, (2) professional life, (3) hobbies, and (4) conversation closing. Below are two sample conversations from our experiments:

### **Conversation 1**

Roberta: Hi I am Roberta. Do you want to register and talk with me?

User: Yes I would like that.

R: What is your name?

U: John.

R: Where are you from?

U: Israel. And you?

R: I'm from Enschede. Do you know it?

U: Yes I do, but I still prefer Israel.

### **Conversation 2**

R: What's your job?

U: I'm now working on an airport security project.

R: What tasks do you do?

U: That's very specific. Right now I am working on voice activation.

R: Tell me about your typical workday.

U: Well I wake up around eight and come to work.

R: Wonderful. What did you study?

U: I studied electrical engineering in France.

Figure 6.4 displays the dialogue structure extracted from the WOZ conversations. An agency was defined for each topic based on the WOZ scenarios. The interaction dialogues gathered served as a guide for designing the agencies. The flexible dialogue structure relies on concepts that act as triggers that control agency or subtree execution, which results in topic change as often as needed. For example, if a user brings up a hobby, this triggers the system's interest in this hobby. It will search in the knowledge repository to check for any relevant information about this hobby, or if the hobby is recognised in the user blackboard or the dialogue blackboard history as an attribute value, then the system can ask the user about his/her hobby via a request agent. On the other hand, WOZ experiment participants expressed a tendency to ask questions to the system. To accommodate this, an "and-you" agency was added, which used some expectation agents to make the dialogue more natural, and also information agents that give the system possible answers (cf. Figure 6.5).

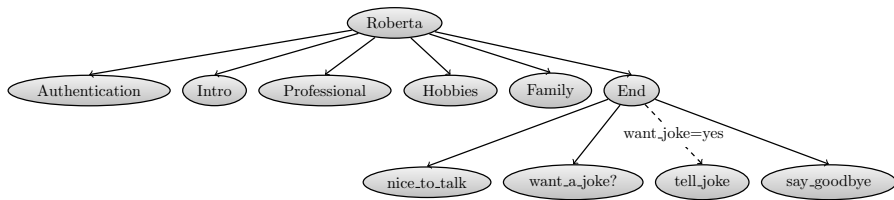


Figure 6.4: Roberta SDS System: Dialogue structure.

### 3.4 Demonstrator

At the end of the eNTERFACE workshop it was possible to test the dialogue system developed via a system demonstrator. Figure 6.6 shows a screenshot of such demonstrator.



Figure 6.6: Roberta: Screenshot of the Roberta SDS demonstrator.

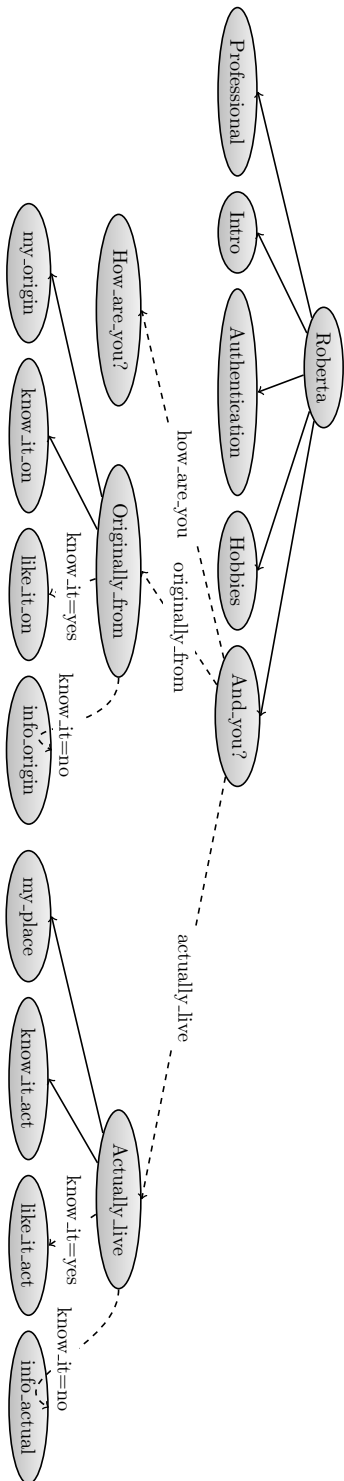


Figure 6.5: Roberta SDS System: Dialogue structure for when users ask questions.

# 7

## CONCLUSIONS

---

In a world in which technology and technological devices have become habitual and omnipresent, humans need to learn to communicate with all kind of devices. Until recently, humans needed to learn how the devices express themselves to communicate with them. But in recent times the tendency has become to make communication with these devices in more intuitive ways. Taking intuitiveness into consideration, the ideal way to communicate with devices would be the natural way of communication between humans, the speech. In this context, humans have long been investigating and designing systems that use this type of communication, giving rise to the so-called SDSs.

In the described context, based on different contacts carried out with the enterprise world, the research group to which the author of this thesis is assigned detected a need for such dialogue systems, which awoke in the group and the author of this thesis the interest for this type of systems. The research group focused on engineering and in building functional and reliable systems. Several systems were built.

With the above in mind, the primary goal of this thesis has been to describe the developed systems and to show how they were implemented. Also, to show how the difficulties encountered were solved. Additionally, the thesis serves as a review of the state-of-the-art regarding architectures and toolkits focused on the development of SDSs. Finally, it has also been written in the hope that it will serve future developers of SDSs as a guide for their construction.

During the development of the systems several conclusions and contributions were obtained. They are summarised next.

- A review of frameworks for SDSs development has been carried out. Along with the implementation of the different systems, has made it possible to see that the choice of an appropriate framework is an important task. Depending on the specifications of the system we want to develop we can choose one framework or another. Among these specifications the platform (operating system) on which the framework works and the set of components it provides. Regarding the first, not all frameworks are designed to be used on all platforms, and regarding the second, depending on the set of components the framework provides we can save more or less development time. It has also shown that it's important to choose a framework which allows as much modularity as possible, as this

- will facilitate the future replacement of components.
- We have developed several components to implement SDSs under EDECAN framework. It's an important achievement as we provide a new set of components to this framework that will facilitate the implementation of future systems.
  - We have made it possible to use the RavenClaw DM in other frameworks apart from its native one, Olympus. This is of interest since RavenClaw is a state-of-the-art DM and it's interesting that it can be integrated into other frameworks. Also, in the specific case of the EDECAN framework, it's of particular interest since this framework does not currently have any dialogue management module.
  - The development of a new framework to implement SDSs focused on mobile devices has been carried out. This is interesting because nowadays almost everyone has one of these devices and it's useful to have some platform that allows us the development of systems focused on them. On the other hand, this platform is also attractive because it enables the creation of SDSs and speech-based applications in a straightforward way by reducing its implementation to the edition of a single XML-based file.
  - We have developed a set of SDSs aimed to deal with a variety of applications, e.g. weather forecast retrieval, bus schedules information retrieval, personal and elderly assistance, under a variety of frameworks, namely, EDECAN, Olympus, RTMIVA and ActiveMQ. We think this task could be useful because we contribute to the development and evolution of the SDSs and, as well, because these systems will help future developers learn how the SDSs can be built.
  - We analysed the users' experience in a telephone information system for buses through a set of standardised questionnaires including SUS, SASSI and AttrakDiff 2. These questionnaires are useful for studying the operation of systems. They give an idea of what users think about the systems and we can analyse what aspects can be improved. The questionnaires resulted in a quite good user's experience regarding usability. Also, good scores in some SASSI and attractiveness aspects were obtained. We have seen that some aspects could be improved by improving mainly the language dependent components of the system.

After all this work, we have verified that the implementation of SDSs is a complex task due to the amount of components necessary for its operation. One of the primary problems encountered when developing these systems has to do with the technology underlying each of the different elements that integrate the systems. The current technology is not free of errors and makes these components not to perform always optimally. A clear example of this are the mistakes in the ASR modules, they not always are able to correctly recognize what the users say. Therefore, methods to handle these errors have to be designed.

Finally, this type of systems are evolving by leaps and bounds and very likely will become a daily reality. Hence the interest of this thesis.

# A

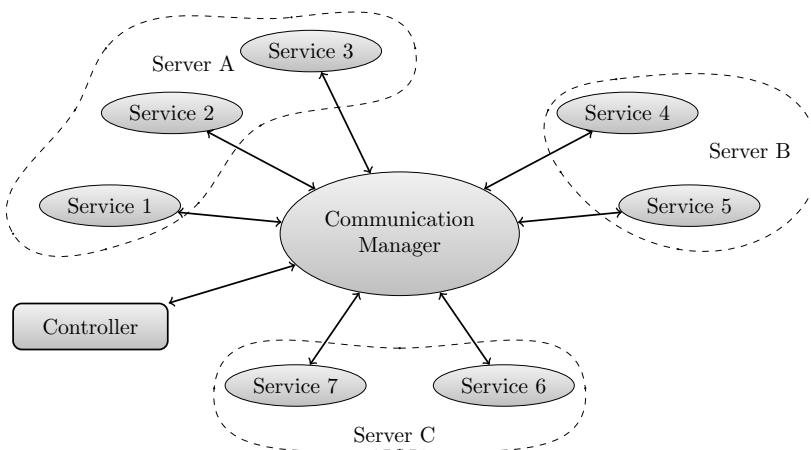
## EDECAN ARCHITECTURE SPECIFICATION

---

### 1 Introduction

---

EDECAN is a communication interface for development of distributed applications, based on TCP/IP protocols and XML messages interchange. It's based on the client-server paradigm and is built using three main components: a *communication manager*, a set of *servers* and a set of *services* running inside the servers. Additionally, it's possible to have a *controller* connected to the communication manager in order to manage a system. Figure A.1 shows such an architecture.



**Figure A.1:** EDECAN architecture block diagram.

#### **Communication Manager**

The communication manager is the central node of any system. It acts as a set of clients of several servers and has three main purposes:

- Routing messages from and to services in the servers. By default, it's done based on a static routing table defined by the developer although it's possible to do dynamic routing from information obtained from the message itself.
- Making modifications on system services such as connecting, disconnecting, subscribing or unsubscribing them.
- Acting as server, wait for a controller connection in order to initializing a system or modifying an existing one.

### Server

A server consists on an application that is waiting for incoming service requests from a communication manager. When a service request arrives it launches the requested service in a new system process and continue listening for new requests. The responsibility of the subsequent interaction with the communication manager falls in the service created.

### Service

A service is an application launched by a server under a communication manager request. When a service request arrives to a server the associated message has the information of the service requested, this information is passed to the service created by the server and said service loads the configuration files for the service requested and launch it.

## 2

## Communication Protocol and Routing Modes

### 2.1

### Communication Protocol

EDECAN architecture is built using `SOCK_STREAM` sockets under TCP protocol. The communication between the services on a system is done via XML messages over these sockets. The basic form of a message is shown in Listing A.1.

```
<?xml version="1.0"?>
<package>
  <command>NAME</command>
  <argument>VALUE</argument>
  ...
  ...
</package>
```

**Listing A.1:** EDECAN basic message example.

The communication manager and services are responsible of sending and receiving the messages. When a message arrives they parse it and obtain the

values of the relevant fields. There are two possible behaviours depending on which one is receiving a message.

### Service

When a service receives a message there are two possibilities:

- First Reception: The first message received by a service is always a connection package. In this case the content of the field "comando" should be "conexion" and additionally there should be another field specifying the service requested, for example  $service_jTTS_i/service_j$ . Extra fields are also allowed.
- General case: The content of the field "comando" is extracted, the service configuration file is accessed and the function associated with the command called, passing to it the full XML package in order to parse the function arguments if needed.

### Communication Manager

In this case there are also two possibilities depending on the routing mode (see Section 2.2).

## 2.2 Routing Modes

### Forced Routing

When a service generates a message it's possible to include "from" and "to" attributes in the root of the message. In this case the communication manager will route the message to service indicated in "to" attribute in case it exists. An example of such a message is shown in Listing A.2.

```
<?xml version="1.0"?>
<package from="service1" to="service2" >
...
...
</package>
```

**Listing A.2:** EDECAN forced routing message example.

### Static Routing

When a message doesn't include "from" and "to" attributes will be routed following the static routing table provided in the system configuration file (see Section 3.1). An example of such a message is shown in Listing A.3.



```
<?xml version="1.0"?>
<package>
  ...
  ...
</package>
```

**Listing A.3:** EDECAN static routing message example.

## 3 System Configuration

### 3.1 Communication Manager

#### Launching Application

In order to launch a communication manager there are two options.

When using the first one, it must be invoked with three arguments like shown below.

```
comm_manager controller_port DNS_port DNS_ip
```

On the other hand, when using the second one it must be invoked with four arguments as follows:

```
comm_manager controller_port DNS_port DNS_ip config_file
```

#### Configuration File

When invoking the communication manager with a configuration file, the system is built reading this file. The configuration file specifies the services to launch and the routing table between them. Listing A.4 shows an example of a configuration file with two services.

It can be seen that the configuration file consist of two main sections. On the one hand, a section defined in "servicios" node that specifies the services that will be part of the system. Each service is defined in a "servicio" node and it must specify "puerto", "direccion" and "tipo\_servicio" indicating communication port and address, and service type respectively. Additionally "configuracion" and "configuracion\_file" can be added. This fields are used to pass arguments to the service at launching time is needed. When using the "configuracion" field the arguments are specified inside the field and in case of using "configuracion\_file" field it specifies the path of a file including the arguments.

On the other hand, a section defined in "tabla\_routing" note. This section specifies the routing table between the services of the system.

```
<?xml version="1.0"?>
<manager>
  <services>
    <service name="Service1">
      <port>6611</port>
      <address>localhost</address>
      <location>LAB</location>
      <configuration>
        <arg1>VALUE<arg1>
        <arg2>VALUE<arg2>
      </configuration>
      <service_type>service1</service_type>
    </service>
    <service name="Service2">
      <port>6611</port>
      <address>localhost</address>
      <location>LAB</location>
      <configuration_file>PATH_TO_ARGS_FILE</configuration_file>
      <service_type>service2</service_type>
    </service>
  </services>

  <routing_table>
    <Service1>
      <Service1>0</Service1>
      <Service2>1</Service2>
    </Service1>
    <Service2>
      <Service1>0</Service1>
      <Service2>1</Service2>
    </Service2>
  </routing_table>
</manager>
```

**Listing A.4:** EDECAN communication manager example configuration file

## 3.2 Server

### Launching Application

In order to launch a server following command must be invoked:

```
server puerto_ini incremento puerto_fin
```

### Configuration File

When a server is launched it loads a file defining the services available. An example of such a file is shown in Listing A.5. It specifies the path of the files defining the available services on the server.

```

<?xml version="1.0"?>
<services>
  <service1>path_to.file/service1.xml</service1>
  <service2>path_to.file/service2.xml</service2>
  <service3>path_to.file/service3.xml</service3>
  <service4>path_to.file/service4.xml</service4>
  <service5>path_to.file/service5.xml</service5>
</services>

```

**Listing A.5:** EDECAN server example configuration file.

### 3.3 Service

#### Defining a Service

In order to define a service two elements are needed. On the one hand, a dynamic library implementing the service's functions and on the other hand a XML configuration file specifying the commands the service responds to. Listing A.6 shows an example of a service configuration file. This configuration file must include the name of the service, the name of the dynamic library associated with the service and the set of commands the service responds to.

```

<?xml version="1.0"?>
<service name="NAME" >
  <so>libNAME.so</so>
  <dll>libNAME.dll</dll>
  <command nombre="conexion" >
    <function>entrada</function>
  </command>
  <command nombre="desconexion" >
    <function>salida</function>
  </command>
  <command nombre="desconexion_caidagestor" >
    <function>salida_caidagestor</function>
  </command>
  <command nombre="command1" >
    <function>command1 associated function</function>
  </command>
  <command nombre="command2" >
    <function>command2 associated function</function>
  </command>
  <command nombre="command3" >
    <function>command3 associated function</function>
  </command>
</service>

```

**Listing A.6:** EDECAN service example configuration file.

Regarding the dynamic library it must meet two requirements:

- All functions associated with a command in the service's configuration file must be implemented in the dynamic library.
- These functions must have the following format:

```
int function_name(char *xmlIN, char **xmlOUT, void **usr);
```

where *xmlIN* and *xmlOUT* parameters are used by the service to send and receive data to the communication manager and *usr* parameter is used as a pointer to a memory address in which the dynamic library can allocate and free memory for data structures.

On the other hand, the dynamic library can implement functions not following the EDECAN function format, but these functions can not be associated to service command and can be only used as auxiliary functions in order for the service to achieve its task.

## 4 Controller Commands

The controller can send system configuration orders to the communication manager in order to be executed.

In general, the format of the commands sent is a text string with a determined number of arguments space separated and being the first parameter the type of order to execute, and the rest the auxiliary arguments.

### Mount System

This order is aimed to mount a new system. The text string sent is as follows:

```
montar_sistema config_file
```

The first argument identifies the order as "montar\_sistema" is a text string containing a XML text string with the system configuration file. The configuration file must specify the services to be connected by the communication manager, specified by IP, port number and service type, and also must contain a static routing table. An example of a system configuration file is shown in Listing A.4.

### Umount System

The communication manager can handle only one system at a time. So, if a new system is desired to be mounted without having to restart the communication manager, it's possible to send an umount order that will disconnect all the connected services and resources will be freed.

The text string sent is as follows:

```
desmontar_sistema
```

### Add New Service

It is possible to add a new service to an already mounted system. The order sent to the communication manager is as follows:

```
alta_servicio config_file
```

The second argument corresponds to a configuration file specifying service's data such as name, IP address, port, service type and routing table. An example of a configuration file for adding a new service is shown in Listing A.7.

```
<service name="New_Service">
  <port>6520</port>
  <address>localhost</address>
  <location>LAB</location>
  <config_file>NONE</config_file>
  <service_type>New_Service</service_type>

  <routing_table_send>
    <Service1>0</Service1>
    <Service2>1</Service2>
    <Service3>1</Service3>
  </routing_table_send>
  <routing_table_rec>
    <Service1>1</Service1>
    <Service2>1</Service2>
    <Service3>0</service3>
  </routing_table_rec>
</service>
```

**Listing A.7:** EDECAN communication manager example add new service configuration file

### Remove Service

If desired to remove a service from a mounted system, it's possible to do it with an order as follows:

```
baja_servicio service_name
```

where *service\_name* is the name of the service to remove from the system.

### Connect Service

It could be possible that a service is part of a system but it's disconnected (it has just been inserted in the system, it's down because of a failure, ...) and we want to connect it again. Such a situation can be solved sending an order as follows:

```
conectar_servicio service_name
```

where *service\_name* is the name of the service to connect.

### Disconnect Service

If for any reason is desired to disconnect a service from the system without unloading it, it can be done sending the order:

```
desconectar_servicio service_name
```

where *service\_name* is the name of the service to disconnect.

### Modify System's Routing Table

If it's desired to modify the full routing table of a system it can be done with the command:

```
config_rutas_sistema config_file
```

The second argument of the command is a configuration file specifying the new routing table. An example of a configuration file is shown in Listing A.8.

```
<routing_table>
  <Service1>
    <Service1>0</Service1>
    <Service2>1</Service2>
  </Service1>
  <Service2>
    <Service1>0</Service1>
    <Service2>1</Service2>
  </Service2>
</routing_table>
```

**Listing A.8:** EDECAN communication manager example routing table

### Modify Service's Routing Table

Instead of modifying the routing table of the entire system, it's also possible to modify the routing table of an unique service. The format of the order to send to the communication manager is as follows:

```
config_rutas_servicio config_file
```

where the first argument is the name of the command to be executed by the communication manager and the second one is a XML configuration file specifying the new routing table for the service. Listing A.9 shows an example of a configuration file.

```
<Service1>
  <Service1>0</Service1>
  <Service2>1</Service2>
  <Service3>1</Service3>
</Service1>
```

**Listing A.9:** EDECAN communication manager example service routing table

**Request System's Services Information**

It's possible to obtain information for each of the services of a system such as name, IP address, port, configuration file and connection state (that is, if a service is connected or disconnected). It's done via the string:

*info\_sistema*

When the communication manager receives the command replies with a text string containing the information.

**Request System's Routing Table Information**

For obtaining the routing table of a system the controller can send a command as follows:

*info\_sistema\_rutas*

When the communication manager receives the command replies with a text string containing the system's routing table.

# B

## SYSTEMS BASED ON EDECAN: CONFIGURATION FILES

---

1

### Campus SDS System

---

```
<?xml version="1.0"?>
<manager>
  <services>
    <service name="controller">
      <port>6611</port>
      <address>localhost</address>
      <location>LAB</location>
      <configuration>
        <log_dir>log.dialogos</log_dir>
      </configuration>
      <service_type>controller</service_type>
    </service>

    <service name="dm">
      <port>6611</port>
      <address>localhost</address>
      <location>LAB</location>
      <configuration>
        <routes_file>../data/backend/rutas.txt</routes_file>
        <stops_file>../data/backend/paradas.txt</stops_file>
        <timetables_file>../data/backend/horarios.txt</timetables_file>
      </configuration>
      <service_type>dm</service_type>
    </service>

    <service name="nlg">
      <port>6611</port>
      <address>localhost</address>
      <location>LAB</location>
      <configuration>NONE</configuration>
      <service_type>nlg</service_type>
    </service>

    <service name="tts">
      <port>6611</port>
```



```

<address>localhost</address>
<location>LAB</location>
<configuration>
  <audiodevice>loop010</audiodevice>
  <fsamp>8000</fsamp>
  <voice>Jorge</voice>
</configuration>
<service_type>tts</service_type>
</service>

<service name="VOIP" >
  <port>6611</port>
  <address>localhost</address>
  <location>LAB</location>
  <configuration>
    <cfg_file>../data/voip.cfg</cfg_file>
  </configuration>
  <service_type>VOIP</service_type>
</service>

<service name="asr">
  <port>6611</port>
  <address>localhost</address>
  <location>LAB</location>
  <configuration>
    <audiodevice>loop011</audiodevice>
    <fsamp>16000</fsamp>
    <channels>1</channels>
    <bitsample>16</bitsample>
    <nsamples_frame>256</nsamples_frame>
    <language>es-ES</language>
    <api_key>XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX</api_key>
  </configuration>
  <service_type>asr</service_type>
</service>

<service name="sp" >
  <port>6611</port>
  <address>localhost</address>
  <location>LAB</location>
  <configuration>
    <parser_grammar_dir>../data/phoenix/Grammar</parser_grammar_dir>
    <parser_grammar_name>DigaTask.net</parser_grammar_name>
  </configuration>
  <service_type>sp</service_type>
</service>
</services>

<routing_table>
  <controller>
    <controller>0</controller>
    <VOIP>1</VOIP>
    <dm>1</dm>
    <nlg>1</nlg>
    <tts>1</tts>
    <asr>1</asr>
    <sp>1</sp>
  </controller>

```

```
<VOIP>
  <controller>1</controller>
  <VOIP>0</VOIP>
  <dm>0</dm>
  <nlg>0</nlg>
  <tts>0</tts>
  <asr>0</asr>
  <sp>0</sp>
</VOIP>

<dm>
  <controller>1</controller>
  <VOIP>0</VOIP>
  <dm>0</dm>
  <nlg>0</nlg>
  <tts>0</tts>
  <asr>0</asr>
  <sp>0</sp>
</dm>

<nlg>
  <controller>1</controller>
  <VOIP>0</VOIP>
  <dm>0</dm>
  <nlg>0</nlg>
  <tts>0</tts>
  <asr>0</asr>
  <sp>0</sp>
</nlg>

<tts>
  <controller>1</controller>
  <VOIP>0</VOIP>
  <dm>0</dm>
  <nlg>0</nlg>
  <tts>0</tts>
  <asr>0</asr>
  <sp>0</sp>
</tts>

<asr>
  <controller>1</controller>
  <VOIP>0</VOIP>
  <dm>0</dm>
  <nlg>0</nlg>
  <tts>0</tts>
  <asr>0</asr>
  <sp>0</sp>
</asr>

<sp>
  <controller>1</controller>
  <VOIP>0</VOIP>
  <dm>0</dm>
  <nlg>0</nlg>
  <tts>0</tts>
  <asr>0</asr>
  <sp>0</sp>
</sp>
```

```
</routing_table>
</manager>
```

**Listing B.1:** Campus SDS System communication manager configuration file.

```
<?xml version="1.0"?>
<services>
  <controller>config_files/services/controller.xml</controller>
  <VOIP>config_files/services/VOIP.xml</VOIP>
  <dm>config_files/services/dm.xml</dm>
  <nlg>config_files/services/nlg.xml</nlg>
  <tts>config_files/services/tts.xml</tts>
  <asr>config_files/services/asr.xml</asr>
  <sp>config_files/services/sp.xml</sp>
</services>
```

**Listing B.2:** Campus SDS System services configuration file.

```
<?xml version="1.0"?>
<service name="asr">
  <so>libASR.so</so>
  <command name="conexion">
    <function>entrada</function>
  </command>
  <command name="desconexion">
    <function>salida</function>
  </command>
  <command name="desconexion_caidagestor">
    <function>salida_caidagestor</function>
  </command>
  <command name="asr_run">
    <function>asr_run</function>
  </command>
  <command name="asr_stop">
    <function>asr_stop</function>
  </command>
</service>
```

**Listing B.3:** Campus SDS System ASR EDECAN module configuration file.

```
<?xml version="1.0"?>
<service name="sp">
  <so>libSP.so</so>
  <command name="conexion">
    <function>entrada</function>
  </command>
  <command name="desconexion">
    <function>salida</function>
  </command>
  <command name="desconexion_caidagestor">
    <function>salida_caidagestor</function>
  </command>
  <command name="sp_run">
    <function>sp_run</function>
  </command>
```

```
</service>
```

**Listing B.4:** Campus SDS System SP EDECAN module configuration file.

```
<?xml version="1.0"?>
<service name="nlg">
  <so>libNLG.so</so>
  <command name="conexion">
    <function>entrada</function>
  </command>
  <command name="desconexion">
    <function>salida</function>
  </command>
  <command name="desconexion_caidagestor">
    <function>salida_caidagestor</function>
  </command>
  <command name="nlg_run">
    <function>nlg_run</function>
  </command>
</service>
```

**Listing B.5:** Campus SDS System NLG EDECAN module configuration file.

```
<?xml version="1.0"?>
<service name="tts">
  <so>libTTS.so</so>
  <command name="conexion">
    <function>entrada</function>
  </command>
  <command name="desconexion">
    <function>salida</function>
  </command>
  <command name="desconexion_caidagestor">
    <function>salida_caidagestor</function>
  </command>
  <command name="tts_run">
    <function>tts_run</function>
  </command>
  <command name="tts_stop">
    <function>tts_stop</function>
  </command>
</service>
```

**Listing B.6:** Campus SDS System TTS EDECAN module configuration file.

```
<?xml version="1.0"?>
<service name="dm">
  <so>libDM.so</so>
  <command name="conexion">
    <function>entrada</function>
  </command>
  <command name="desconexion">
    <function>salida</function>
  </command>
  <command name="desconexion_caidagestor">
```

```

    <function>salida_caidagestor</function>
  </command>
  <command name="dm_init_session">
    <function>dm_init_session</function>
  </command>
  <command name="dm_end_session">
    <function>dm_end_session</function>
  </command>
  <command name="dm_nonunderstanding_start_over">
    <function>dm_nonunderstanding_start_over</function>
  </command>
  <command name="dm_nonunderstanding_giveup">
    <function>dm_nonunderstanding_giveup</function>
  </command>
  <command name="sp_end">
    <function>sp_end</function>
  </command>
  <command name="tts_end">
    <function>tts_end</function>
  </command>
  <command name="dm_hangout">
    <function>dm_hangout</function>
  </command>
</service>

```

**Listing B.7:** Campus SDS System DM EDECAN module configuration file.

```

<?xml version="1.0"?>
<service name="VOIP">
  <so>libVOIP.so</so>
  <command name="conexion">
    <function>entrada</function>
  </command>
  <command name="desconexion">
    <function>salida</function>
  </command>
  <command name="desconexion_caidagestor">
    <function>salida_caidagestor</function>
  </command>
</service>

```

**Listing B.8:** Campus SDS System Phone Interface EDECAN module configuration file.

```

<?xml version="1.0"?>
<service name="controller">
  <so>libcontroller.so</so>
  <command name="conexion">
    <function>entrada</function>
  </command>
  <command name="desconexion">
    <function>salida</function>
  </command>
  <command name="desconexion_caidagestor">
    <function>salida_caidagestor</function>
  </command>

```

```

<command name="dm_init_session">
  <function>dm_init_session</function>
</command>
<command name="controller_end_session">
  <function>controller_end_session</function>
</command>
<command name="controller_hangout">
  <function>controller_hangout</function>
</command>
<command name="nlg_run">
  <function>nlg_run</function>
</command>
<command name="nlg_end">
  <function>nlg_end</function>
</command>
<command name="asr_run">
  <function>asr_run</function>
</command>
<command name="asr_end">
  <function>asr_end</function>
</command>
<command name="tts_end">
  <function>tts_end</function>
</command>
<command name="sp_end">
  <function>sp_end</function>
</command>
</service>

```

**Listing B.9:** Campus SDS System EDECAN controller module configuration file.

## 2 INMA

```

<?xml version="1.0"?>
<manager>
  <services>
    <service name="asr">
      <port>6611</port>
      <address>localhost</address>
      <location>LAB</location>
      <configuration>NONE</configuration>
      <service_type>asr</service_type>
    </service>

    <service name="comprehension">
      <port>6611</port>
      <address>localhost</address>
      <location>LAB</location>
      <configuration>NONE</configuration>
      <service_type>comprehension</service_type>
    </service>

    <service name="tts">
      <port>6611</port>

```

```

    <address>localhost</address>
    <location>LAB</location>
    <configuration>NONE</configuration>
    <service_type>tts</service_type>
  </service>

  <service name="databackend">
    <port>6611</port>
    <address>localhost</address>
    <location>LAB</location>
    <configuration>NONE</configuration>
    <service_type>databackend</service_type>
  </service>

  <service name="ui">
    <port>6611</port>
    <address>localhost</address>
    <location>LAB</location>
    <configuration>NONE</configuration>
    <service_type>ui</service_type>
  </service>
</services>

<routing_table>
  <tts>
    <tts>0</tts>
    <asr>1</asr>
    <comprehension>0</comprehension>
    <databackend>0</databackend>
    <ui>1</ui>
  </tts>

  <asr>
    <tts>0</tts>
    <asr>0</asr>
    <comprehension>1</comprehension>
    <databackend>0</databackend>
    <ui>1</ui>
  </asr>

  <comprehension>
    <tts>1</tts>
    <asr>1</asr>
    <comprehension>0</comprehension>
    <databackend>1</databackend>
    <ui>1</ui>
  </comprehension>

  <databackend>
    <tts>0</tts>
    <asr>0</asr>
    <comprehension>0</comprehension>
    <cliente>0</cliente>
    <ui>1</ui>
  </databackend>

  <ui>
    <tts>1</tts>
    <asr>0</asr>

```

```

    <comprehension>1</comprehension>
    <databackend>0</databackend>
    <ui>0</ui>
  </ui>
</routing_table>
</manager>

```

**Listing B.10:** INMA communication manager configuration file.

```

<?xml version="1.0"?>
<services>
  <tts>config_files/services/tts.xml</tts>
  <asr>config_files/services/asr.xml</asr>
  <ui>config_files/services/ui.xml</ui>
  <comprehension>config_files/services/comprehension.xml</comprehension>
  <databackend>config_files/services/databackend.xml</databackend>
</services>

```

**Listing B.11:** INMA services configuration file.

```

<?xml version="1.0"?>
<service name="asr">
  <so>libASR.so</so>
  <command name="conexion">
    <function>entrada</function>
  </command>
  <command name="desconexion">
    <function>salida</function>
  </command>
  <command name="desconexion_caidagestor">
    <function>salida_caidagestor</function>
  </command>
  <command name="asr_run">
    <function>asr_run</function>
  </command>
</service>

```

**Listing B.12:** INMA ASR EDECAN module configuration file.

```

<?xml version="1.0"?>
<service name="tts">
  <so>libTTS.so</so>
  <command name="conexion">
    <function>entrada</function>
  </command>
  <command name="desconexion">
    <function>salida</function>
  </command>
  <command name="desconexion_caidagestor">
    <function>salida_caidagestor</function>
  </command>
  <command name="tts_run">
    <function>tts_run</function>
  </command>
</service>

```



**Listing B.13:** INMA TTS EDECAN module configuration file.

```

<?xml version="1.0"?>
<service name=" databackend">
  <so>libDATABACKEND.so</so>
  <command name="conexion">
    <function>entrada</function>
  </command>
  <command name="desconexion">
    <function>salida</function>
  </command>
  <command name="desconexion_caidagestor">
    <function>salida_caidagestor</function>
  </command>
  <command name="action_meteo">
    <function>action_meteo</function>
  </command>
  <command name="action_news">
    <function>action_news</function>
  </command>
</service>

```

**Listing B.14:** INMA DataBackend EDECAN module configuration file.

```

<?xml version="1.0"?>
<service name="comprehension">
  <so>libCOMP.so</so>
  <command name="conexion">
    <function>entrada</function>
  </command>
  <command name="desconexion">
    <function>salida</function>
  </command>
  <command name="desconexion_caidagestor">
    <function>salida_caidagestor</function>
  </command>
  <command name="comp_run">
    <function>comp_run</function>
  </command>
  <command name="comp_reinit">
    <function>comp_reinit</function>
  </command>
</service>

```

**Listing B.15:** INMA Comprehension EDECAN module configuration file.

```

<?xml version="1.0"?>
<service name="ui">
  <so>libUI.so</so>
  <command name="conexion">
    <function>entrada</function>
  </command>
  <command name="desconexion">
    <function>salida</function>
  </command>

```

```
</command>
<command name="desconexion_caidagestor">
  <function>salida_caidagestor</function>
</command>
<command nombre="view_meteo">
  <function>view_meteo</function>
</command>
<command nombre="view_news">
  <function>view_news</function>
</command>
<command nombre="activate_voice">
  <function>activate_voice</function>
</command>
<command nombre="deactivate_voice">
  <function>deactivate_voice</function>
</command>
<command nombre="show_ini_page">
  <function>show_ini_page</function>
</command>
<command nombre="set_avatar_speak">
  <function>set_avatar_hablar</function>
</command>
<command nombre="set_avatar_not_speak">
  <function>set_avatar_no_hablar</function>
</command>
</service>
```

**Listing B.16:** INMA User Interface EDECAN module configuration file.



# C

## RTMIVA APPLICATIONS CONFIGURATION FILES

---

### 0.1 Applications Interface Code

---

```
#!/usr/bin/php5 -q
<?php
set_time_limit(30);
require('phpagi/phpagi.php');
error_reporting(E_ALL);

\sagi = new AGI();
\sagi->answer();
\sext=\$argv[1];
\susu=\$argv[2];
\bus=1;

\directorio=opendir("/var/www/aplicaciones/xml");
\fich="";
while ((\sarchivo = readdir(\sdirectorio))&&(\sbus==1)){
    if ((\sarchivo!=".")&&(\sarchivo!="..")){
        if (\sxt==leerextensionxml("/var/www/aplicaciones/xml/\sarchivo")){
            \fich=\sarchivo;
            \bus=0;
        }
    }
}
closedir(\sdirectorio);
if (\sbus==0){
    procesarPeticon(\sfich,\sagi,1);
} else {
    \sfrase="Hola \susu, la extension \sargv[1], a la que deseas llamar, no dispone de
ninguna aplicacion vinculada. Intentelo con otra extension.";
exec("sudo /opt/Loquendo/LTTS/./txt2audio Jorge /var/lib/asterisk/sounds/es/
temp.wav \sfrase",\svalida);
\sagi->stream_file('temp');
}

\sagi->hangup();

function leerextensionxml(\$xml)
{
    \sarchivo=file_get_contents(\$xml);
    \sobjXML=simplexml_load_string(\sarchivo);
}
```

```

return \ObjXML->extension;
}

function procesarpeticion(\$xml,\$agi,\$nume)
{
    \archivo=file_get_contents("/var/www/aplicaciones/xml/\$xml");
    \ObjXML=simplexml_load_string(\archivo);
    \contador=1;
    \acabar=false;
    foreach (\ObjXML->aplicacion[0]->comando as \$comando){
        if (((\contador==\$nume)||(\contador>\$nume))&&(\acabar==false)){
            if (\comando['tipo']=="sintetizar"){
                sintetizar(\comando,\$vari,\$valorvari,\$agi);
            } else if (\comando['tipo']=="sintetizarif"){
                \numi=0;
                for (\$ji=0;\$ji<count(\$vari);\$ji++){
                    if(\$valorvari[\$ji]!="null"){
                        \$varcarg[\$numi]=\$vari[\$ji];
                        \numi++;
                    }
                }
            }
            foreach (\comando->ifff as \$ifff){
                for (\$ji=0;\$ji<count(\$varcarg);\$ji++){
                    if((string)\$ifff['cond']==(string)\$varcarg[\$ji]){
                        sintetizar(\$ifff,\$vari,\$valorvari,\$agi);
                    }
                }
            }
        } else if (\comando['tipo']=="reconocer"){
            \stiempo="".((int)\comando[seg])*1000;
            \n=0;
            foreach(\$comando->variable as \$variable){
                \$vari[\$n]=\$variable['nom'];
                \n++;
            }
            \$mes=date("n");
            \$dia=date("d");
            \$anio=date("y");
            \$hora=date("G");
            \$min=date("i");
            \$seg=date("s");

            \$nofich="\$anio\$mes\$dia\$hora\$min\$seg";
            \$agi->record_file("temp/\$nofich",'wav','#',\$stiempo,'0','0','2');
            exec("../bin/param08 /var/lib/asterisk/sounds/temp/\$nofich.wav ../temp/
parametros/\$nofich.xml",\$salida);
            \$fi= explode(".",\$xml);
            if (!file_exists("../temp/lexico/" .\$fi[0].".txt")){
                exec("php ../bin/xmllexico.php \$xml",\$salida);
                exec("../bin/crear_lexico ../temp/lexico/" .\$fi[0]."_tmp.txt ../temp/lexico/
.\$fi[0].".txt");
                exec("rm ../temp/lexico/" .\$fi[0]."_tmp.txt");
            }
            copy("../temp/lexico/" .\$fi[0].".txt","../data/lexico.txt");
            exec("java -classpath ../commons-jxpath-1.2.jar crearBin ../temp/
parametros/\$nofich.xml");
            exec("../bin/reconocedor ../temp/parametros/\$nofich.bin",\$salida);
            \$reconocido="";
            foreach(\$salida as \$line){

```

```

    \Sreconocido=\$line;
}

\Sreconocido=str_replace('_', '\Sreconocido);
\Sreconocido=str_replace(' ', '\Sreconocido);
\Sreconocido=str_replace('ñ', 'ni', \Sreconocido);
\Sreconocido=str_replace('á', 'a', \Sreconocido);
\Sreconocido=str_replace('é', 'e', \Sreconocido);
\Sreconocido=str_replace('í', 'i', \Sreconocido);
\Sreconocido=str_replace('ó', 'o', \Sreconocido);
\Sreconocido=str_replace('ú', 'u', \Sreconocido);

for (\$y=0; \$y<count(\$vari); \$y++){
    \$valorvari[\$y]=getvariable(\$comando, \$vari[\$y], \Sreconocido);
}

exec("sudo rm /var/lib/asterisk/sounds/temp/\$nofich.wav", \$salida);
exec("sudo rm ../temp/parametros/\$nofich.xml", \$salida);
exec("sudo rm ../temp/parametros/\$nofich.bin", \$salida);
} else if (\$comando['tipo']=="sintetizarweb"){
    \$url=\$comando;
    \$trozos=explode('%', \$url);
    for(\$ii=0; \$ii<count(\$trozos); \$ii++){
        for(\$jj=0; \$jj<count(\$vari); \$jj++){
            \$len=strlen(\$vari[\$jj]);
            if (\$len>strlen(\$trozos[\$ii])){
                \$len=strlen(\$trozos[\$ii]);
            }
            if(substr(\$trozos[\$ii], 0, \$len)==\$vari[\$jj]){
                \$url=str_replace('%'.\$vari[\$jj], \$valorvari[\$jj], \$url);
            }
        }
    }
}

\$data = file_get_contents( \$url );
exec("sudo /opt/Loquendo/LTTS/./txt2audio Jorge /var/lib/asterisk/sounds
/es/temp.wav \$data", \$salida);
\$agi->stream_file('temp');
} else if (\$comando['tipo']=="comprobar"){
    \$mes=date("n");
    \$dia=date("d");
    \$anio=date("y");
    \$hora=date("G");
    \$min=date("i");
    \$seg=date("s");

    \$nofich="\$anio\$mes\$dia\$hora\$min\$seg";
    \$agi->record_file("temp/\$nofich", 'wav', '#', '2000', '0', '0', '2');
    exec("../bin/param08 /var/lib/asterisk/sounds/temp/\$nofich.wav ../temp/
parametros/\$nofich.xml", \$salida);
    \$agi->stream_file("temp/\$nofich");
    copy("../temp/lexico/comprobar.txt", "../data/lexico.txt");
    exec("java -classpath :commons-jxpath-1.2.jar crearBin ../temp/
parametros/\$nofich.xml");
    exec("../bin/reconocedor ../temp/parametros/\$nofich.bin", \$salida);
    \$recono="";
    foreach(\$salida as \$line){
        \$recono=\$line;
    }
}

```

```

    \${recono}=str_replace('.',',',\${recono});

    exec("sudo rm /var/lib/asterisk/sounds/temp/\${nofich.wav}",\${salida});
    exec("sudo rm ../temp/parametros/\${nofich.xml}",\${salida});
    exec("sudo rm ../temp/parametros/\${nofich.bin}",\${salida});

    if((\${recono}=="No")||(\${recono}=="Te repito")||(\${recono}=="Has
reconocido mal")){
        procesarpeticion(\${xml},\${agi},(int)\${comando['iffalse']});
        \${acabar}=true;
    }
    exec("sudo /opt/Loquendo/LTTS/./txt2audio Jorge /var/lib/asterisk/sounds/es/
temp.wav Comprobacion: \${recono}.\",\${salida});
    \${agi}->stream_file('temp');
}
}
\${contador}++;
}
}

function getvariable(\${comando},\${variable},\${reconocido})
{
    \${resp}="null";
    foreach(\${comando}->variable as \${vari}){
        if (\${vari['nom']}==\${variable}){
            foreach(\${vari}->valor as \${svalor}){
                if ((strpos(\${reconocido},(string)\${svalor})||(\${reconocido}==(string)\${svalor})){
                    return \${svalor};
                }
            }
        }
    }
    return \${resp};
}

function sintetizar(\${sin},\${vari},\${svalorvari},\${agi})
{
    \${trozos}=explode('%',\${sin});
    for(\${ii}=0;\${ii}<count(\${trozos});\${ii}++){
        for (\${jj}=0;\${jj}<count(\${vari});\${jj}++){
            \${len}=strlen(\${vari}[\${jj}]);
            if (\${len}>strlen(\${trozos}[\${ii}])){
                \${len}=strlen(\${trozos}[\${ii}]);
            }
            if(substr(\${trozos}[\${ii}],0,\${len})==\${vari}[\${jj}]){
                \${sin}=str_replace('%'.\${vari}[\${jj}],\${svalorvari}[\${jj}],\${sin});
            }
        }
    }
    exec("sudo /opt/Loquendo/LTTS/./txt2audio Jorge /var/lib/asterisk/sounds/es/
temp.wav \${sin}",\${salida});
    \${agi}->stream_file('temp');
}
}
?>

```

Listing C.1: RT-MIVA: Weather forecast application XML file

## 0.2 TTS: text2audio source code

```

#include <stdio.h>
#include "/opt/Loquendo/LTTS/include/loqtts.h" /* Loquendo TTS include file */
#include <string.h>

#define CODING "l" /* "l" (linear), "a" (A-law), "u" (U-law) */
#define SAMPLE_RATE 8000 /* Use 8000 for A-law or U-law coding */

int main(int argc, char *argv[])
{
    ttsHandleType hInstance; /* Instance handle */
    ttsHandleType hVoice; /* Voice handle */
    ttsResultType err; /* Error code returned by TTS APIs */

    char* p = "Jorge";
    char* fich = "sample.wav";
    char frase[5000]="";
    int i;

    fprintf(stderr,"Number: %i \n",argc);
    if(argc > 3) {
        p = argv[1];
        fich= argv[2];
        for(i=3;i<argc;i++)
        {
            strcat(frase,argv[i]);
            strcat(frase," ");
        }
        fprintf(stderr,"Sentence: %s\n",frase);
        fprintf(stderr,"File : %s\n",fich);
    } else {
        fprintf(stderr, "Exec '%s <speaker> <file> <text> '\n", argv[0]);
        return -1;
    }
    fprintf(stderr, "Using voice %s\n", p);

    /* Initializes the LoquendoTTS Instance */
    err = ttsNewInstance(&hInstance, NULL, NULL);
    if (err != tts_OK)
    {
        fprintf(stderr, "%s", ttsGetError(NULL));
        return err;
    }

    /* Sets the voice parameters */
    err = ttsNewVoice(&hVoice, hInstance, p, SAMPLE_RATE, CODING);
    if (err != tts_OK)
    {
        fprintf(stderr, "%s", ttsGetError(hInstance));
        (void)ttsDeleteInstance(hInstance);
        return err;
    }

    /* Sends samples directly to the audio board */
    err = ttsSetAudio(hInstance, "LoqAudioFile",argv[2], CODING, 0);

```



```

if (err != tts.OK)
{ fprintf(stderr, "\%s", ttsGetError(hInstance));
  (void)ttsDeleteInstance(hInstance);
  return err;
}

/* Converts text to speech */
err = ttsRead(
    hInstance, /* Instance handle */
    frase, /* Input */
    TTSBUFFER, /* "Input" is a text buffer */
    TTSANSI, /* Input text is in ANSI */
    TTSDEFAULT, /* Default ReadingMode */
    TTSBLOCKING); /* ttsRead keeps control until the end */

if (err != tts.OK)
{
    fprintf(stderr, "\%s", ttsGetError(hInstance));
    (void)ttsDeleteInstance(hInstance);
    return err;
}

/* Closes the Loquendo TTS instance; the voice will be automatically closed */
(void)ttsDeleteInstance(hInstance);
return 0;
}

```

**Listing C.2:** RT-MIVA: source code for the TTS interface.

### 0.3

### Weather Information Retrieval XML File

```

<rtmiva>
  <name>Weather Forecast</name>
  <description>Aplicacion a la que se le puede solicitar el tiempo de una city y te
  devuelve la prediccion de esta.</description>
  <extension>1</extension>

  <application>
    <command num="1" type="synthesize">Bienvenido al servicio de prediccion
    meteorologico.</command>
    <command num="2" type="synthesize">Que puedo hacer por usted?</command>
    <command num="3" type="recognize" seg="6">
      <lexicon num="1" pre="0" suc="2,3,5,6,7">Hola</lexicon>
      <lexicon num="2" pre="0,1" suc="4,5,6,7">buenas</lexicon>
      <lexicon num="3" pre="0,1" suc="5,6,7">buenos dias</lexicon>
      <lexicon num="4" pre="2" suc="5,6,7">tardes</lexicon>
      <lexicon num="5" pre="0,1,2,3,4" suc="15">quisiera</lexicon>
      <lexicon num="6" pre="0,1,2,3,4" suc="15">quiero</lexicon>
      <lexicon num="7" pre="0,1,2,3,4" suc="8,9,10,11,12">me</lexicon>
      <lexicon num="8" pre="7" suc="13,14">podias</lexicon>
      <lexicon num="9" pre="7" suc="13,14">puedes</lexicon>
      <lexicon num="10" pre="7" suc="13,14">podrias</lexicon>
      <lexicon num="11" pre="7" suc="15">das</lexicon>
      <lexicon num="12" pre="7" suc="15">dices</lexicon>
    </command>
  </application>
</rtmiva>

```

```

<lexicon num="13" pre="8,9,10" suc="15">dar</lexicon>
<lexicon num="14" pre="8,9,10" suc="15">decir</lexicon>
<lexicon num="15" pre="5,6,11,12,13,14" suc="16,17">el tiempo</lexicon>
<lexicon num="16" pre="15" suc="18,19,20,21,22">que</lexicon>
<lexicon num="17" pre="15" suc="24">de</lexicon>
<lexicon num="18" pre="16" suc="23">hara</lexicon>
<lexicon num="19" pre="16" suc="23">hace</lexicon>
<lexicon num="20" pre="16" suc="23">hay</lexicon>
<lexicon num="21" pre="16" suc="23">tenemos</lexicon>
<lexicon num="22" pre="16" suc="23">tendremos</lexicon>
<lexicon num="23" pre="18,19,20,21,22" suc="24">en</lexicon>
<lexicon num="24" pre="17,23" suc="25,End">\%city</lexicon>
<lexicon num="25" pre="24" suc="End">por favor</lexicon>

<variable numtotal="10" nom="city">
  <value num="1">Bilbao</value>
  <value num="2">San Sebastian</value>
  <value num="3">Barcelona</value>
  <value num="4">Madrid</value>
  <value num="5">Valencia</value>
  <value num="6">Benidorm</value>
  <value num="7">Santander</value>
  <value num="8">Zaragoza</value>
  <value num="9">Salamanca</value>
  <value num="10">Oviedo</value>
</variable>
</command>
<command num="4" type="synthesize">Desea saber la prediccion de \%city?</
command>
<command num="5" type="check" iffalse="2" />
<command num="6" type="synthesizeweb">http://localhost/rtmivap/tiempo.php?
city=\%city</command>
</application>
</rtmiva>

```

**Listing C.3:** RT-MIVA: Weather forecast application XML file



# D

## USABILITY QUESTIONNAIRES

---

### 1 System Usability Scale

---

The System Usability Scale (SUS) provides a reliable tool for measuring the usability, giving a global view of subjective assessments of usability. SUS consists of a 10 item questionnaire (shown in Table D.1) with five response options, from Strongly disagree (1) to Strongly agree (5). It allows to evaluate a wide variety of products and services, including hardware, software, mobile devices, websites and applications. It has become an industry standard. Some benefits of using SUS include:

- Is a very easy scale to administer to system users.
- Can be used on small sample sizes with reliable results.
- Is valid, i.e. it can effectively differentiate between usable and unusable systems.

#### **Using SUS**

SUS questionnaire is usually used after users have interacted with the system being evaluated at least once, but before any discussion takes place. Users should be asked to record an immediate response to each item, rather than thinking about items for a long time. All items should be checked. If a user feels that cannot respond to a particular item, he or she should mark the centre point of the scale.

#### **Scoring SUS**

SUS yields a single number representing a composite measure of the overall usability of the system being studied. Scores for individual items are not meaningful on their own.

To calculate the SUS score:

- For odd items: subtract one from the user response.
- For even items: subtract the user responses from 5.
- Add up the converted responses for each user and multiply that total by 2.5.

SUS scores have a range of 0 to 100. Based on research [139], SUS scores above

68 would be considered above average and anything below 68, below average. Also, scores below 50 are typically considered as not acceptable [125].

For the example questionnaire shown below:

- Total score = 22
- SUS Score =  $22 * 2.5 = 55$

	1	2	3	4	5	
I think that I would like touse this system frequently					x	4
I found the system unnecessarily complex				x		1
I thought the system was easy to use		x				1
I think that I would need the support of a technical person to be able to use this system	x					4
I found the various functions in this system were well integrated		x				1
I thought there was too much inconsistency in this system			x			2
I would imagine that most people would learn to use this system very quickly		x				1
I found the system very cumbersome to use				x		1
I felt very confident using the system					x	4
I needed to learn a lot of things before I could get going with this system		x				3

- 
- I think that I would like to use this system frequently
  - I found the system unnecessarily complex
  - I thought the system was easy to use
  - I think that I would need the support of a technical person to be able to use this system
  - I found the various functions in this system were well integrated
  - I thought there was too much inconsistency in this system
  - I would imagine that most people would learn to use this system very quickly
  - I found the system very cumbersome to use
  - I felt very confident using the system
  - I needed to learn a lot of things before I could get going with this system
- 

**Table D.1:** SUS questionnaire.

## 2 Subjective Assessment Speech System Interfaces

The Subjective Assessment of Speech System Interfaces (SASSI) questionnaire (Table D.2) is aimed to measure user satisfaction with speech systems. SASSI consists of a set of statements (shown in Table D.1) with six response options, from Strongly disagree (1) to Strongly agree (6). It measures several aspects:

- **Response Accuracy:** Related to whether the system recognises users input correctly and hence does what the user intended and expected.
- **Likeability:** Related to opinion and feeling users have about the system.
- **Cognitive Demand:** Summarises both the perceived level of effort needed to use the system and user feelings arising from this effort.
- **Annoyance:** Evaluates the feeling or state of being annoyed/frustrated of users when using the system.
- **Habitability:** Related to whether the user knows what to say and knows what the system is doing. A habitable system may be defined as one in which there is a good match between the user's conceptual model of the system and the actual system.
- **Speed:** Evaluates the response speed of the system.

For each aspect, the score is obtained calculating the mean value and standard deviation for the values of all the answers. A thing to take into account is that for each aspect evaluated there are "positive" and "negative" statements. It is necessary to invert the "sign" of one of them by rescaling the scores (preferably the "negative" ones). As an example, the sub-questionnaire for Cognitive Demand shown below has a score of 2.2 ( $sd = 0.98$ ).

	1	2	3	4	5	6		1	2	3	4	5	6
I felt confident using the system		x						x					
I felt tense using the system					-x			x					
I felt calm using the system	x							x					
A high level of concentration is required when using the system				-x							x		
The system is easy to use		x						x					

In general, the higher the score of an aspect the better is perceived such aspect by the user.

<b>Response Accuracy</b>
The system is accurate
The system is unreliable
The interaction with the system is unpredictable
The system didn't always do what I wanted
The system didn't always do what I expected
The system is dependable
The system makes few errors
The interaction with the system is consistent
The interaction with the system is efficient
<b>Likeability</b>
The system is useful
The system is pleasant
The system is friendly
I was able to recover easily from errors
I enjoyed using the system
It is clear how to speak to the system
It is easy to learn to use the system
I would use this system
I felt in control of the interaction with the system
<b>Cognitive Demand</b>
I felt confident using the system
I felt tense using the system
I felt calm using the system
A high level of concentration is required when using the system
The system is easy to use
<b>Annoyance</b>
The interaction with the system is repetitive
The interaction with the system is boring
The interaction with the system is irritating
The interaction with the system is frustrating
The system is too inflexible
<b>Habitability</b>
I sometimes wondered if I was using the right word
I always knew what to say to the system
I was not always sure what the system was doing
It is easy to lose track of where you are in an interaction with the system
<b>Speed</b>
The interaction with the system is fast
The system responds too slowly

**Table D.2:** SASSI questionnaire.

### 3 AttrakDiff 2

AttrakDiff 2 questionnaire (Table D.3) is intended for measuring the attractiveness of interactive products. Using 28 pairs of opposite adjectives, users evaluate their perception of the product being evaluated. Each pair of adjectives is rated within a seven anchor scale.

The questionnaire evaluates the attractiveness of the product based on four aspects:

- Pragmatic Quality (PQ): Describes the usability of a product and indicates how successfully users are in achieving their goals using the product.
- Hedonic Quality - Stimulation: Mankind has an inherent need to personal growth, i.e. wants to develop their skills and knowledge further. The HQ-S aspect indicates to what extent the product can support those needs.
- Hedonic Quality - Identity (HQ-I): Indicates to what extent the product allows the user to identify with it.
- Attractiveness (ATT): Describes a global value of the product based on the quality perception.

For each aspect, the score is obtained calculating the mean value and standard deviation for the values of all the answers. As an example, the sub-questionnaire for PQ shown below has a score of 4.29 (sd = 0.88).

	1	2	3	4	5	6	7	
Technical					x			Human
Complicated					x			Simple
Impractical				x				Practical
Cumbersome			x					Straightforward
Unpredictable			x					Predictable
Confusing					x			Clearly Structured
Unruly					x			Manageable

In general, the higher the score of an aspect the better is perceived such aspect by the user.



<b>Pragmatic Quality</b>	
Technical	- Human
Complicated	- Simple
Impractical	- Practical
Cumbersome	- Straightforward
Unpredictable	- Predictable
Confusing	- Clearly Structured
Unruly	- Manageable
<b>Hedonic Quality - Identification</b>	
Isolating	- Connective
Unprofessional	- Professional
Tacky	- Stylish
Cheap	- Premium
Alienating	- Integrating
Separates Me from People	- Brings Close to People
Unpresentable	- Presentable
<b>Hedonic Quality - Stimulation</b>	
Conventional	- Inventive
Unimaginative	- Creative
Cautious	- Bold
Conservative	- Innovative
Dull	- Captivating
Undemanding	- Challenging
Ordinary	- Novel
<b>Attraction</b>	
Unpleasant	- Pleasant
Ugly	- Attractive
Disagreeable	- Likeable
Rejecting	- Inviting
Bad	- Good
Repelling	- Appealing
Discouraging	- Motivating

**Table D.3:** ATTRAKDIFF 2 questionnaire.

# BIBLIOGRAPHY

---

- [1] S. Young, "Probabilistic methods in spoken dialogue systems," *Philosophical Trans of the Royal Society*, vol. 1769(358), pp. 1389–1402, 2000.
- [2] "RavenClaw/Olympus Web Page." [Online]. Available: [http://www.cs.cmu.edu/~dbohus/ravenclaw-olympus/what\\_is\\_olympus.html](http://www.cs.cmu.edu/~dbohus/ravenclaw-olympus/what_is_olympus.html)
- [3] S. J. Young, "The htk hidden markov model toolkit: Design and philosophy," vol. 2, 1994, pp. 2–44.
- [4] K. F. Lee, H. W. Hon, and R. Reddy, "An overview of the SPHINX speech recognition system," in *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 38, no. 1, 1990, pp. 35–45.
- [5] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, and K. Vesely, "The kaldi speech recognition toolkit," in *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*, 2011.
- [6] T. Winograd, "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language," *Cognitive Psychology*, vol. 3, no. 1, 1972.
- [7] W. A. Woods, R. M. Kaplan, and B. Nash-Webber, "The Lunar Sciences Natural Language Information System: Final Report," *BBN Rep.*, no. 2378, 1972.
- [8] R. Schank, N. Goldman, C. Rieger, and C. Riesbeck, "MARGIE: Memory Analysis, Response Generation and Inference on English," in *3IJCAI*, 1973.
- [9] W. Ward, "Understanding spontaneous speech: the phoenix system," in *ICASSP 91: 1991 International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, Apr 1991, pp. 365–367.
- [10] Y. Wang, L. Deng, and A. Acero, "Spoken language understanding - an introduction to the statistical framework," in *IEEE Signal Processing Magazine*, vol. 5, 2005, pp. 16–31.
- [11] S. Miller, R. Bobrow, R. Ingria, and R. Schwartz, "Hidden understanding models of natural language," in *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, 1994, pp. 25–32.
- [12] S. Della-Pietra, M. Epstein, S. Roukos, and T. Ward, "Fertility Models for Statistical Natural Language Understanding," in *Proceedings of the Association for Computational Linguistics*, 1997, pp. 168–173.
- [13] Y. He and S. Young, "Semantic processing using the hidden vector state model," *Computer Speech and Language*, vol. 19, pp. 85–106, 2005.
- [14] C. Wong and H. Meng, "Improvements on a semi-automatic grammar induction framework," in *IEEE Workshop Automatic Speech Recognition and Understanding*, Madonna di Campiglio, Italy, 2001.
- [15] S. Busemann and H. Horacek, "A flexible shallow approach to text generation," in *9th International Workshop on Natural Language Generation*, 1998, pp. 238–247.
- [16] K. van Deemter and J. Odiijk, "Context modelling and the generation of spoken discourse," in *Speech Communication*, vol. 21(1/2), 1997, pp. 101–121.
- [17] S. C. McRoy, Susan W. and S. S. Ali, "An augmented template-based approach to text realization," in *Natural Language Engineering*, vol. 9(4), 2003, pp. 381–420.
- [18] F. Mairesse, M. Gašić, F. Jurčićek, S. Keizer, B. Thomson, K. Yu, and S. Young, "Phrase-based statistical language generation using graphical models and active learn-

- ing,” in *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ser. ACL '10, 2010, pp. 1552–1561.
- [19] C. Gardent, B. Gottesman, and L. Perez-beltrachini, “Using regular tree grammars to enhance sentence realisation,” *Nat. Lang. Eng.*, vol. 17, no. 2, pp. 185–201, 2011.
- [20] Y. W. Wong, “Learning for semantic parsing and natural language generation using statistical machine translation techniques,” 2007.
- [21] V. Rieser and O. Lemon, “Natural language generation as planning under uncertainty for spoken dialogue systems,” in *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*, 2009, pp. 683–691.
- [22] —, “Reinforcement Learning for Adaptive Dialogue Systems: A Data-driven Methodology for Dialogue Management and Natural Language Generation,” in *Theory and Applications of Natural Language Processing*, 2011.
- [23] I. K., G. L.M., W. D.H., T. M.K., and R. P.E., “CASY: The Haskins configurable articulatory synthesizer,” in *Proceedings of the 15th International Congress on Phonetic Sciences*, 2003, pp. 185–188.
- [24] K. Yu, H. Zen, F. Mairesse, and S. Young, “Context adaptive training with factorized decision trees for hmm-based statistical parametric speech synthesis,” in *Speech communication*, vol. 53, 2011, pp. 914–923.
- [25] D. Goddeau, H. Meng, J. Polifroni, S. Seneff, and S. Busayapongchai, “A form-based dialogue manager for spoken language applications,” in *ICSLP*, 1996.
- [26] V. Zue, S. Seneff, J. Glass, J. Polifroni, C. Pao, T. J. Hazen, and L. Hetherington, “Jupiter: A telephone-based conversational interface for weather information,” vol. 8, 2000, pp. 85–96.
- [27] S. Seneff and J. Polifroni, “Dialogue management in the mercury flight reservation system,” in *Proceedings of ANLP-NAACL Workshop on Conversational Systems*, 2000, pp. 1–6.
- [28] A. Rudnicky and W. Xu, “An agenda-based dialog management architecture for spoken language systems,” in *IEEE Automatic Speech Recognition and Understanding Workshop*, 1999.
- [29] W. Xu and A. Rudnicky, “Task-based dialog management using an agenda,” in *ANLP/NAACL Workshop on Conversational Systems*, 2000, pp. 42–47.
- [30] H. Aust and O. Schroer, “An overview of the philips dialog system,” in *Proceedings of the DARPA Broadcast News Transcription and Understanding Workshop*, 1998.
- [31] H. Strik, A. Russel, H. van den Heuvel, L. Boves, and C. Cucchiari, “A spoken dialogue system for public transport information,” in *Proceedings of the Department of Language and Speech*, vol. 19, 1996, pp. 129–142.
- [32] S. Seneff, E. Hurley, R. Lau, C. Pao, P. Schmid, and V. Zue, “Galaxy-ii: A reference architecture for conversational system development,” in *ICSLP*, 1998, pp. 931–934.
- [33] C. Rich and A. L. Sidner, “COLLAGEN: a collaboration manager for software interface agents,” in *User Model User-Adapt Interact*, vol. 8, 1998, pp. 315–350.
- [34] J. F. Allen and C. R. Perrault, “Analyzing intention in utterances,” in *Artificial Intelligence*, vol. 15(3), 1980, pp. 143–178.
- [35] “Olympus.” [Online]. Available: <http://www.ravenclaw-olympus.org/>
- [36] “Disco - Collaborative Discourse Manager.” [Online]. Available: <https://github.com/charlesrich/Disco>
- [37] D. Traum, J. Bos, R. Cooper, S. Larsson, I. Lewin, C. Matheson, and M. Poesio, “A model of dialogue moves and information state revision,” in *Trindi deliverable D2.1*,

- 1999.
- [38] P. Fodor and J. M. Huerta, “Planning and logic programming for dialog management,” in *Proceedings of SLT*, 2006, pp. 214–217.
- [39] S. Young, “Probabilistic Methods in Spoken Dialogue Systems,” *Philosophical Transactions of the Royal Society of London*, 2000.
- [40] J. D. Williams and S. Young, “Partially observable markov decision processes for spoken dialog systems,” in *Computer Speech & Language*, vol. 21(2), 2007, pp. 393–422.
- [41] F. J. Ciccek, B. Thomson, and S. Young, “Reinforcement learning for parameter estimation in statistical spoken dialogue systems,” *Computer, Speech and Language*, vol. 26, no. 3, pp. 168–192, 2012.
- [42] S. Young, M. Gašić, B. Thomson, and J. D. Williams, “POMDP-based Statistical Spoken Dialog Systems: A review,” *Proceedings of the IEEE*, vol. 101, no. 5, pp. 1160–1179, 2013.
- [43] M. I. Torres, “Stochastic Bi-Languages to model Dialogs,” in *International Conference on Finite State Methods and Natural Language Processing*, 2013, pp. 9–17.
- [44] P.-H. Su, D. Vandyke, M. Gasic, D. Kim, N. Mrksic, T.-H. Wen, and S. Young, “Learning from Real Users: Rating Dialogue Success with Neural Networks for Reinforcement Learning in Spoken Dialogue Systems,” in *InterSpeech*, 2015, pp. 2007–2011.
- [45] I. V. Serban, A. Sordoni, Y. Bengio, A. Courville, and J. Pineau, “Building End-to-end Dialogue Systems Using Generative Hierarchical Neural Network Models,” in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI’16. AAAI Press, 2016, pp. 3776–3783.
- [46] R. P. E. Levin and W. Eckert, “A stochastic model of human-machine interaction for learning dialog strategies,” in *IEEE TSAP*, vol. 8, no. 1, 2000, pp. 11–23.
- [47] E. V. Alejandro H. Toselli and F. Casacuberta, “Multimodal Interactive Pattern Recognition and Applications.” Springer, 2011.
- [48] M. I. Torres, J. M. Benedí, R. Justo, and F. Ghigi, “Modeling spoken dialog systems under the interactive pattern recognition framework,” in *Lecture Notes on Computer Science*, 2012, pp. 519–528.
- [49] M. I. Torres and F. Casacuberta, “Stochastic k-tss bi-languages for machine translation,” in *9th International Workshop on Finite State Models for Natural Language Processing (FSMNLP)*, 2011, pp. 98–106.
- [50] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to Sequence Learning with Neural Networks,” *CoRR*, vol. abs/1409.3215, 2014.
- [51] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” *CoRR*, vol. abs/1409.0473, 2014.
- [52] O. Vinyals and Q. V. Le, “A neural conversational model.” *CoRR*, vol. abs/1506.05869, 2015.
- [53] J. D. Williams.
- [54] J. D. Williams and G. Zweig, “End-to-end LSTM-based dialog control optimized with supervised and reinforcement learning,” in *abs/1606.01269v1*, 2016.
- [55] M. Rickert, M. E. Foster, M. Giuliani, T. By, G. Panin, and A. Knoll, *Integrating Language, Vision and Action for Human Robot Dialog Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 987–995.
- [56] D. Suendermann, J. Liscombe, J. Bloom, and R. Pieraccini, “Topic and Emotion Classification of Customer Surveys.”

- [57] M. Serras, N. Perez, M. I. Torres, A. Del Pozo, and R. Justo, *Topic Classifier for Customer Service Dialog Systems*. Springer International Publishing, 2015, pp. 140–148.
- [58] A. Raux, B. Langner, and D. Bohus, “Let’s Go Public! Taking a spoken dialog system to the real world,” in *InterSpeech*, 2005.
- [59] F. Morbini, D. DeVault, K. Georgila, R. Artstein, D. Traum, and L. Morency, “A Demonstration of Dialogue Processing in SimSensei Kiosk,” in *15th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, 2014, p. 254.
- [60] H. Aust, M. Oerder, F. Seide, and V. Steinbiss, “The philips automatic train timetable information system,” vol. 17, no. 3-4, 1995, pp. 249–262.
- [61] “PARLANCE Project.” [Online]. Available: <https://sites.google.com/site/parlanceprojectofficial>
- [62] H. Hastie, M. Aufaure, P. Alexopoulos, H. Bouchard, H. Cuayahuitl, N. Dethlefs, M. Gasic, J. Henderson, O. Lemon, X. Liu, P. Mika, N. B. Mustapha, T. Potter, V. Rieser, B. Thomson, P. Tsiakoulis, Y. Vanrompay, B. Villazon-Terrazas, M. Yazdani, S. Young, and Y. Yu, “The PARLANCE mobile app for interactive search in english and mandarin,” in *Proceedings of SIGDIAL 2014*, 2014.
- [63] O. Dušek, O. Plátek, L. Žilka, and F. Jurčíček, “Alex: Bootstrapping a spoken dialogue system for a new domain by real users,” in *Proceedings of the 15th Annual Meeting of the Special Interest Group on Discourse and Dialogue (SIGDIAL)*, 2014, pp. 79–83.
- [64] M. Swerts, D. Litman, and J. Hirschberg, 2000.
- [65] T. Paek and E. Horvitz, “Conversation as action under uncertainty,” in *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, ser. UAI ’00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 455–464.
- [66] O. Lemon, A. Gruenstein, L. Cavedon, and S. Peters, “Multi-tasking and collaborative activities in dialogue systems,” in *Proceedings of SigDIAL’02*, 2002.
- [67] J. Gustafson, L. Bell, J. Beskow, J. Boye, R. Carlson, J. Edlund, B. Granstorm, D. House, and M. Wiren, “Adapt - a multimodal conversational dialogue system in an apartment domain,” in *Proceedings of ICSLP’00*, 2000.
- [68] G. Ferguson and J. Allen, “Trips: an intelligent integrated problem-solving assistant,” in *Proceedings of AAAI’98*, 1998.
- [69] “Hound.” [Online]. Available: <http://www.soundhound.com/hound>
- [70] “Siri.” [Online]. Available: <http://www.apple.com/es/ios/siri>
- [71] “Cortana.” [Online]. Available: <http://windows.microsoft.com/es-es/windows-10/getstarted-what-is-cortana>
- [72] “Alexa.” [Online]. Available: <https://developer.amazon.com/public/solutions/alexa>
- [73] “Google Now.” [Online]. Available: <https://www.google.com/landing/now>
- [74] D. Bohus, A. Raux, T. K. Harris, M. Eskenazi, and A. Rudnicky, “Olympus: an open-source framework for conversational spoken language interface research,” in *Workshop on Bridging the Gap: Academic and Industrial Research in Dialog Technologies*, 2007, pp. 32–39.
- [75] D. Bohus and A. Rudnicky, “The RavenClaw dialog management framework: Architecture and systems,” in *Computer Speech and Language*, vol. 23, no. 3, 2009, pp. 332–361.
- [76] “Sphinx: Open Source Speech Recognition Toolkit.” [Online]. Available: <https://cmusphinx.github.io/wiki>

- [77] “Galaxy Communicator.” [Online]. Available: <http://communicator.sourceforge.net/index.shtml>
- [78] “Flipper Multi Modal Dialogue System.” [Online]. Available: <https://hmi-utwente.github.io/FlipperMMDS>
- [79] G. Skantze and S. A. Moubayed, “IrisTK: a statechart-based toolkit for multi-party face-to-face interaction,” in *ICMI*, 2013.
- [80] “CSLU Toolkit.” [Online]. Available: <http://www.cslu.ogi.edu/toolkit/>
- [81] “AT&T Statistical Dialog Toolkit.” [Online]. Available: <http://www2.research.att.com/sw/tools/asdt/>
- [82] J. D. Williams, “Incremental partition recombination for efficient tracking of multiple dialog states,” in *ICASSP*, 2010.
- [83] P. Milhorat, “An open-source framework for supporting the design and implementation of natural-language spoken dialog systems,” Ph.D. dissertation, Télécom ParisTech, 12 2014.
- [84] “ActiveMQ.” [Online]. Available: <http://activemq.apache.org>
- [85] “Alex Dialogue Systems Framework.” [Online]. Available: <https://github.com/UFAL-DSG/alex>
- [86] J. E. García, A. Ortega, A. Miguel, and E. Lleida, “Arquitectura distribuida para el desarrollo de sistemas de diálogo hablado, EDECÁN,” 2008.
- [87] “OpenDial.” [Online]. Available: <http://www.opendial-toolkit.net>
- [88] “TRINDIKIT.” [Online]. Available: <http://www.ling.gu.se/projekt/trindi/trindikit/>
- [89] “TRINDI.” [Online]. Available: <http://www.ling.gu.se/projekt/trindi/>
- [90] “SIRIDUS.” [Online]. Available: <http://www.ling.gu.se/projekt/siridus/>
- [91] “TALK.” [Online]. Available: <http://www.talk-project.org/>
- [92] J. Bos, E. Klein, O. Lemon, and T. Oka, “Dipper: description and formalisation of an information-state update dialogue system architecture,” in *Proceedings of SIGDIAL*, 2003, pp. 115–124.
- [93] “Midiki.” [Online]. Available: <http://midiki.sourceforge.net/>
- [94] “Mitre.” [Online]. Available: <http://www.mitre.org/>
- [95] M. Turunen and J. Hakulinen, “Jaspis - an architecture for supporting distributed spoken dialogues,” in *INTERSPEECH*, 2003.
- [96] R. J. Ross and J. Bateman, “Daisie: Information state dialogues for situated systems,” in *TSD2009. LNAI5729*, pp. 379–386.
- [97] “Houndify.” [Online]. Available: <http://www.soundhound.com/houndify>
- [98] “Agilingua, LLC.” [Online]. Available: <http://www.agilingua.com/en/index.html>
- [99] “Alexa Skills Kit.” [Online]. Available: <https://developer.amazon.com/public/solutions/alexa/alexa-skills-kit>
- [100] “Loquendo Text To Speech.” [Online]. Available: <http://www.nuance.com/for-business/text-to-speech/index.htm>
- [101] “Asterisk.” [Online]. Available: <http://www.asterisk.org>
- [102] “PHPAGI.” [Online]. Available: <http://phpagi.sourceforge.net>
- [103] A. W. Black, S. Burger, B. Langner, G. Parent, and M. Eskenazi, “Spoken dialog

- challenge 2010,” in *SLT*, 2010, pp. 448–453.
- [104] W. Ward and S. Issar, “Recent improvements in the cmu spoken language understanding system.” in *Proceedings of the workshop on Human Language Technology*, 1994, pp. 213–216.
- [105] D. Huggins-daines, M. Kumar, A. Chan, A. W. Black, M. Ravishankar, and A. I. Rudnicky, “Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices.” in *Proceedings of ICASSP 2006*, 2006.
- [106] D. Bohus and A. Rudnicky, “Integrating multiple knowledge sources for utterance-level confidence annotation in the cmu communicator spoken dialog system.” in *2nd International Workshop on Community Networking.*, 2002.
- [107] A. Raux and M. Eskenazi, “A multi-layer architecture for semi-synchronous event-driven dialogue management,” in *ASRU*, 2007, pp. 514–519.
- [108] J. M. Benedí, E. Lleida, A. Varona, M. J. Castro-Bleda, I. Galiano, R. Justo, I. L. de Letona, and A. Miguel, “Design and acquisition of a telephone spontaneous speech dialogue corpus in spanish: Dihana,” in *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC 2006)*, Genoa (Italy), 2006, pp. 1636–1639.
- [109] “GTFS Data Exchange.” [Online]. Available: <http://www.gtfs-data-exchange.com>
- [110] A. Black, P. Taylor, and R. Caley, “The festival synthesis system.” December 2002.
- [111] A. W. Black and K. A. Lenzo, “Flite: A small fast runtime synthesis engine.” in *4Th ISCA Tutorial and Research workshop on Speech Synthesis*, 2001, pp. 20–24.
- [112] “Microsoft Speech API (SAPI),” 1996. [Online]. Available: [https://msdn.microsoft.com/en-us/library/ee125663\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ee125663(v=vs.85).aspx)
- [113] “Cepstral Swift,” 2005. [Online]. Available: <http://www.cepstral.com/es/downloads>
- [114] “PJSIP Project.” [Online]. Available: <http://www.pjsip.org/>
- [115] “Virtual Audio Cable.” [Online]. Available: <http://www.ntonyx.com/vac.htm>
- [116] “ALSA Library API.” [Online]. Available: [https://www.alsa-project.org/main/index.php/ALSA.Library\\_API](https://www.alsa-project.org/main/index.php/ALSA.Library_API)
- [117] “Google Developer Console.” [Online]. Available: <https://console.developers.google.com>
- [118] “Rosetta Natural Language Generator.” [Online]. Available: <http://wiki.speech.cs.cmu.edu/olympus/index.php/Rosetta>
- [119] “WINE.” [Online]. Available: <https://www.winehq.org>
- [120] J. Brooke, “SUS - A quick and dirty usability scale,” *Usability evaluation in industry*, vol. 189, no. 194, pp. 4–7, 1996.
- [121] K. S. Hone and R. Graham, “Towards a Tool for the Subjective Assessment of Speech System Interfaces (SASSI),” *Natural Language Engineering*, vol. 6, no. 3-4, pp. 287–303, 2000.
- [122] “Attrakdiff 2.” [Online]. Available: <http://attrakdiff.de/index-en.html>
- [123] “KALDI ASR.” [Online]. Available: <http://kaldi-asr.org>
- [124] I. Wechsung, *An Evaluation Framework for Multimodal Interaction: Determining Quality Aspects and Modality Choice*. Springer Publishing Company, Incorporated, 2014.
- [125] S. Sauro, “L2 performance in text-chat and spoken discourse,” *System*, vol. 40, no. 3, pp. 335–348, 2012.

- [126] P. Milhorat, S. Schlögl, G. Chollet, and J. Boudy, “Un système de dialogue vocal pour les seniors: études et spécifications.” in *JetSan 2013 : Journées d’étude sur la TéléSanté*, 2013.
- [127] “vAssist: Voice Controlled Assistive Care and Communication Services for the Home.” [Online]. Available: [http://vassist.tech-experience.at/project\\_overview/index.html](http://vassist.tech-experience.at/project_overview/index.html)
- [128] G. Neumann, “Design principles of the disco system,” in *Proceedings of the Twente Workshop on Language Technology (TWLT 5). Natural Language Interfaces, June 3-4*, F. de Jong and A. Nijholt, Eds., University of Twente, Enschede, The Netherlands, 1993.
- [129] F. Jurčiček, F. Mairesse, M. Gašić, S. Keizer, B. Thomson, K. Yu, and S. Young, “Transformation-based Learning for semantic parsing,” in *InterSpeech*, 2009, pp. 2719–2722.
- [130] E. Brill, “Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging,” *Computational linguistics*, vol. 21, no. 4, pp. 543–565, 1995.
- [131] M. Schröder and J. Trouvain, “The German text-to-speech synthesis system MARY: A tool for research, development and teaching,” in *International Journal of Speech Technology*, vol. 6, no. 4, 2003, pp. 365–377.
- [132] P. Milhorat, “An open-source framework for supporting the design and implementation of natural-language spoken dialog systems,” Ph.D. dissertation, Télécom ParisTech – 46, rue Barrault – 75013 Paris, 2015.
- [133] C. Rich, “Building task-based user interfaces with ANSI/CEA-2018,” *IEEE Computer*, no. 8, pp. 20–27, 2009.
- [134] “eINTERFACE 2016.” [Online]. Available: <http://hmi.ewi.utwente.nl/enterface16>
- [135] “Cerevoice Engine Text-to-Speech SDK.” [Online]. Available: <https://www.cereproc.com/en/products/sdk>
- [136] “Unity Asset Store.” [Online]. Available: <https://www.assetstore.unity3d.com/en/\#!/content/52234>
- [137] “Telecom SudParis Face Recognition Open-source using Gabor Filters.” [Online]. Available: <http://share.int-evry.fr/svnview-eph>
- [138] S. Schögl, G. Doherty, and S. Luz, “Wizard of Oz experimentation for language technology applications: Challenges and tools.” in *Interacting with Computers*, vol. 6, no. 27, 2015, pp. 592–615.
- [139] “System Usability Scale (SUS).” [Online]. Available: <https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html>



eman la zabal zazu



Universidad Euskal Herriko  
del País Vasco Unibertsitatea