Fakultät Technik
Studiengang Informatik
Duale Hochschule Baden-Württemberg, Stuttgart

Faculty of Media Engineering and Technology
German University in Cairo

The Halef Book

An introduction to the first open-source, industry standards compliant Spoken Dialog System

Bachelor Thesis

Author: Ahmed Elmalatawy
Supervisor: Prof. Dr. David Sündermann

To my family: Azza and Laila
Thank you for your infinite support

Abstract

Spoken Dialog Systems and virtual voice assistants are of increasing importance in today's tech scene. Virtual voice assistants are constantly increasing the accessibility and efficiency of using technology and so, no smart phone ships without one now. Companies such as Google and Apple have taken specific interest in this field as they aim to develop more accurate and domain free virtual voice assistants. In the DHBW Spoken Dialog Systems Research Lab, Halef, a new Spoken Dialog System (SDS) was developed. It consists of three main parts; Asterisk, Cairo and JVoiceXML (JVXML). Those three parts are installed on different servers forming a distributed architecture, with the servers communicating using the industry standards compliant protocols; Session Initialization Protocol (SIP) and MRCP. This work describes how Halef functions, with its different components and distributed architecture, as well as the several changes that were made to each component to suit Halef's needs, improving its efficiency and widening its capabilities.

Contents

Chapter 1

Introduction

Over the past few years, conversational agents have grown in importance, conquering the world of technology by finally being available even in the smallest of devices that everyone has nowadays, smartphones. Conversational agents have been widely used for years in a variety of applications even before smartphones came by, like phone call routing and other simple, limited domain applications. But in the smartphones age, all of today's smartphones ship with a virtual voice assistant as big tech companies such as Google, Microsoft and Apple compete in providing the most functionality through such assistants, enabling their users to control their phones hands-free. However, most of these advancements are proprietary software and the open source community is yet to catch up.

This work describes Halef [3], an open source, industry standards compliant spoken dialog system [4] [5], which aims at bridging that gap between proprietary software and it's open source counterpart. Halef works in a unique distributed architecture, splitting it into three main components, with the ability to split those three main components even further giving a high degree of flexibility and easily allowing multiple different architectures to suit each use case. In addition to all that, all protocols that Halef runs on; SIP and MRCP/RTP, are industry standards compliant, making it suitable and trusted for use in industry as well as research. The Halef application development language, VXML, is also widely recognized in the community and many proprietary softwares out there already use it as well to define their applications. So, with all that in mind, we will jump into the nitty gritty details of Halef in the next few sections, describing how it works as well as how to tweak it for use in different areas of application with a variety of distributed architectures.

1.1 Background - General Halef Architecture

As mentioned earlier, Halef is composed of three main servers; Cairo, JVXML and Asterisk. Those three servers interact in different ways to provide the final functionality of Halef. In this section, a brief description of the role of each server will be given, in addition to how they all interact to provide Halef's functionality.

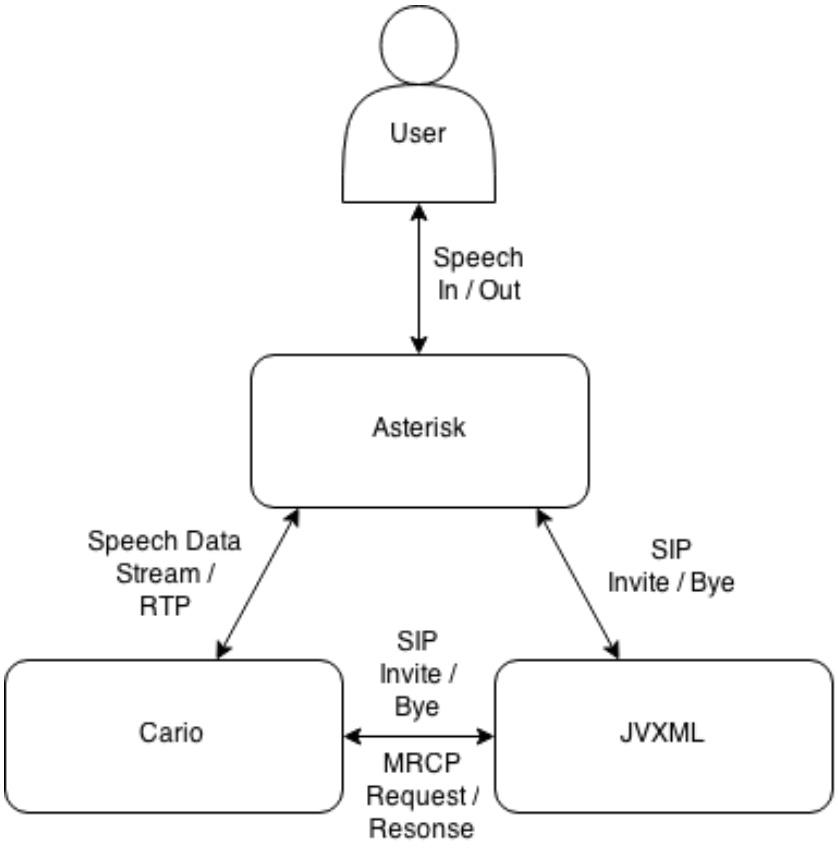

Figure 1.1: Halef's General Architecture

1.1.1 The Asterisk Server

Asterisk is a free, open source framework that provides the functionality of a telephony server servicing multiple types of connections like SIP, IP PBX and Public Switched Telephone Network (PSTN). The Asterisk project is widely used and recognized by industry, governments and the open source community. It has a large user base, providing a lot of support over the forums and the project itself is also documented very well. You can read more about the Asterisk project on its website[1] and the book "Asterisk: A Definitive Guide" [2]. In the case of Halef, regular phone calls over PSTN and VoIP calls over SIP are supported. Users can call in to Halef using their telephones or cellphones on the US number assigned to it, or they can call in free using a SIP softphone utilising the provided VoIP service. Either way, the Asterisk server serves as the entry point of a user call into Halef. When a user calls in over PSTN or SIP, the Asterisk telephony server picks up the call, welcomes the user then asks for an extension to route the user to. Each extension points to a different Halef instance running its own application, providing a robust way of running different Halef instances but connecting them all to the same telephony service that is set up on a single server. This reduces the work needed down to

[1] http://www.asterisk.org/

only integrating the needed JVXML and Cairo servers with the already running telephony server.

1.1.2 The Cairo Server

The Cairo server is the core of Halef, being responsible for both speech recognition and synthesis. When the call is set up, the Cairo server is connected to the JVXML server over MRCP channels to receive recogntion and synthesis requests, and to the Asterisk Server over RTP to stream audio back and forth to and from the user. When the Cairo server receives an MRCP recognition request, it chooses the suitable recognizer type specified in the MRCP message through the recognition interface then segments the open RTP stream to detect the speech utterance to recognize, then returns the recognition result to JVXML. On the other hand, when it receives a synthesis request, it synthesizes the string specified in the MRCP message then streams the synthesized speech to the user through Asterisk, over RTP. The Cairo Server will be described in detail in Chapter 2.

1.1.3 The JVXML Server

JVXML is an open-source voice browser for VXML written in Java. It works as a VXML interpreter, parsing and executing the VXML applications written to Halef and specified in the JVXML Server's configuration file. The JVXML Server connects to Asterisk through SIP and to Cairo through both SIP and MRCP. Once a user call is received, Asterisk connects to the JVXML Server through SIP, which then forwards the SIP messages to the Cairo server to set up its MRCP and RTP channels. The JVXML server then goes on to parse and execute the specified application, sending recognition and syntesis requests to Cairo over MRCP when the application requests either action.

1.1.4 Interaction between the Servers

The three servers are interconnected over different channels that set up then provide Halef's functionality throughout the user's call. The first connection is between the Asterisk and the JVXML servers. Those two servers are connect via SIP, which is used to signal the start and end of a call, as well as provide the information needed to set up the RTP session. The SIP messages sent from the Asterisk server to the JVXML one are then forwarded by JVXML to the Cairo server. The Cairo server then uses the information in the messages to allocate and free resources as requested. The second connection is between the Cairo and Asterisk servers. This connection is over RTP, which is used to send the user's speech stream to Cairo and return the synthesized text prompt to Asterisk so it can play it to the user. The third essential conncection is between Cairo and JVXML. This connection is over MRCP which enables JVXML to send Cairo speech specific requests such as *RECOGNIZE*, *RECORD* and *SYNTHESIZE*. Cairo then returns the results of those operations, such as the recognized text for a *RECOGNIZE* request, to JVXML over MRCP as well.

1.2 Objectives

The main goal of this work is to describe Halef in as much detail as possible, from the conceptual and architectural level, to the classes distribution and actual code level. With three independent servers forming the backbone of Halef, each will be described in detail as to how it runs, its own architecture and finally, how it works to serve the requests it receives to contribute to the work of the whole system. Several practices and areas for change and improvement will be pointed out throughout the whole book and finally summarized in the final chapters. Some applications that have been integrated and tested with Halef will also be included to demonstrate how Halef can serve different applications as well as how to overcome some difficulties with needed features that are not part of the system yet.

1.3 Structure

This book starts the walk through Halef by describing the first server, the Cairo server, in chapter 2. This chapter will go into Cairo's architecture and how to start its different components. It then describes the life cycle of the five main requests this server has to handle, the SIP invite, SIP bye, MRCP recognize, MRCP record and MRCP synthesize requests.

The next chapter, 3, describes the JVXML server, starting from parsing the application file to communicating with the Cairo server to satisfy the recognition and synthesis requests.

Finally, some applications that have been already integrated into Halef will be described, in addition to the suggested future work on the system, before concluding the book.

Chapter 2

Cairo Server

The Cairo server serves as the main hub for managing Halef's resources for speech recognition and synthesis. It works as a stand-alone server waiting for any recognition or synthesis requests from the JVXML server. It then performs the desired operations and replies to the JVXML server with the operations' results. The Cairo server starts three different threads; a resource server thread, a receiver resource thread and a transmitter resource thread. The main architecture of the Cairo server is described in figure 2.1. In its communication with the JVXML server, three protocols are used; SIP for call initiation and ending, and MRCP over RTP for audio streaming using VoIP in addition to sending and receiving recognition/synthesis requests and responses. With this distinction, two different types of request flows arise; the SIP request life cycle and the MRCP request life cycle, each of which will be described in the following sections. But first, let's have a look at how to configure and start the Cairo server.

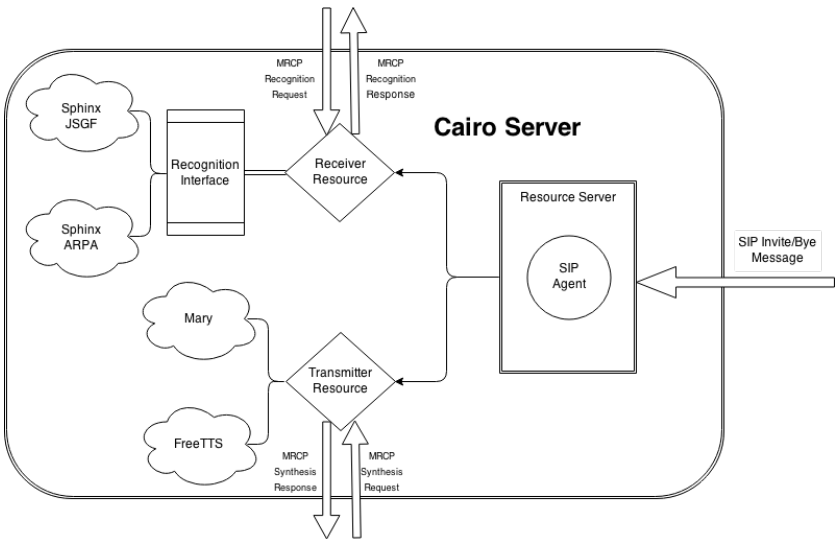

Figure 2.1: Cairo Server Architecture

2.1 Configuring the Cairo Server

Configuring the Cairo server is fairly simple and quick, if you understand the basic, needed components. The configuration is split into two main parts; receiver and transmitter, as you can see in listing 2.1.

Listing 2.1: Sample Cairo Config File

```
<cairo>
  <resources>
    <resource type="receiver">
      <name>receiver1</name>
      <mrcpPort>32416</mrcpPort>
      <rtpBasePort>42050</rtpBasePort>
      <maxConnects>50</maxConnects>
      <engines>5</engines>
      <recorderEngines>5</recorderEngines>
      <baseRecordingDir>/temp/cairo/recordingDir</
         baseRecordingDir>
      <baseGrammarDir>/temp/cairo/baseGrammarDir</baseGrammarDir>
    </resource>

  <resource type="transmitter">
      <name>transmitter1</name>
      <mrcpPort>32417</mrcpPort>
      <rtpBasePort>42150</rtpBasePort>
      <maxConnects>50</maxConnects>
      <engines>5</engines>
      <voiceName>kevin</voiceName>
      <basePromptDir>/temp/cairo/basePromptDir</basePromptDir>
    </resource>
  </resources>
</cairo>
```

2.1.1 Configuring the Receiver Resource

The configuration of the receiver resource is specified in the `<resource type="receiver">` part of the Cairo configuration file. This is split into a few main parts:

baseGrammarDir This is where to store the grammar file for recognition if it is sent over the network to the Cairo server. This functionality is currently disabled as only the grammar file's URL is sent over the network.

baseRecordingDir This is where to save the speech utterances as wav files after the recording process is done by the *SphinxRecorder*.

engines This is the maximum number of recognition engines per pool.

ipAddress This is to specify the IP address of the resource. You generally don't need to specify this as the IP address is obtained automatically.

maxConnects This is the maximum number of allowed parallel sessions to connect to this receiver resource.

mrcpPort This is the MRCP port for the receiver resource. This should be the same as the MRCP transmitter port of the JVXML server.

name This is the identifier for the receiver being configured. The convention is to have the name as "receiver$index".

recorderEngines This is the maximum number of recorder engines per pool.

rtpBasePort This is the RTP port for the receiver resource. This should be the same as the RTP transmitter port of the JVXML server.

sphinxConfigURL This is the path to the Sphinx recognizer configuration file. You generally don't need to specify this if the file is compiled into the "resources/config/" folder in Cairo.

sphinxRecorderConfigURL This is the path to the Sphinx recorder configuration file. You generally don't need to specify this if the file is compiled into the "resources/-config/" folder in Cairo.

2.1.2 Configuring the Transmitter Resource

The configuration of the transmitter resource is specified in the `<resource type="transmitter">` part of the Cairo configuration file. This is also split into a few main parts:

basePromptDir This is where to save the synthesized speech wav file after the speech synthesizer is done.

engines This is the maximum number of synthesis engines per pool.

ipAddress This is to specify the IP address of the resource. You generally don't need to specify this as the IP address is obtained automatically.

maxConnects This is the maximum number of allowed parallel sessions to connect to this transmitter resource.

mrcpPort This is the MRCP port for the transmitter resource. This should be the same as the MRCP receiver port of the JVXML server.

name This is the identifier for the receiver being configured. The convention is to have the name as "transmitter$index".

rtpBasePort This is the RTP port for the transmitter resource. This should be the same as the RTP receiver port of the JVXML server.

voiceName This is the voice to be used by the FreeTTS synthesizer.

2.2 Starting the Cairo Server

Now, having understood all of the moving parts in the Cairo server, let's try to run it. To start the Cairo server, all you need to do is go to the scripts directory, then run start_cairo.sh. This script is going to start all three components through their respective starting scripts that can be found in Cairo's bin folder.

Listing 2.2: Cairo Starting Script

```
1 bash rserver.sh &
2 sleep 2
3 bash receiver1.sh &
4 sleep 4
5 bash transmitter1.sh &
```

Firstly, the resource server is started through its script *rserver.sh*. This scripts launches the server and passes some needed arguments like the SIP port to be used to communicate with the JVXML server. The resource server is given some time to initialize all of its components like setting up the SIP channel.

Listing 2.3: Cairo Resource Server Starting Script

```
1 #!/bin/bash
2
3 CLASS="org.speechforge.cairo.server.resource.ResourceServerImpl"
4 bash launch.sh $CLASS -sipPort 5050 -sipTransport udp
```

Secondly, the receiver resource thread is launched and passed the Cairo configuration file as well as the resource name to fetch from the config file.

Listing 2.4: Cairo Receiver Resource Starting Script

```
1 #!/bin/bash
2
3 CLASS=org.speechforge.cairo.server.resource.ReceiverResource
4 CAIRO_CONFIG=file:../config/cairo-config.xml
5 RES_NAME=receiver1
6 sh launch.sh $CLASS $CAIRO_CONFIG $RES_NAME
```

Lastly, the transmitter resource thread is started and passed the same kind of arguments as the receiver resource

Listing 2.5: Cairo Transmitter Resource Starting Script

```
1 #!/bin/bash
2
3 CLASS=org.speechforge.cairo.server.resource.TransmitterResource
4 CAIRO_CONFIG=file:../config/cairo-config.xml
5 RES_NAME=transmitter1
6 sh launch.sh $CLASS $CAIRO_CONFIG $RES_NAME
```

Now that the Cairo server is up and running, it is ready to receive call initiation requests over SIP, so, let's have a look at the SIP requests life cycle.

2.3 SIP Request Life Cycle

As the calls flow into Halef, there should be a way to communicate the call initialization and termination requests between all three servers. The SIP protocol is used for this exact purpose. SIP provides two main types of requests, namely, an invite and a bye request, each of which is self explanatory to the nature of its purpose. In the following subsections, we're going to look at how Cairo prepares itself for sending and receiving SIP requests, as well as, how it reacts to each type of SIP requests.

2.3.1 Setting Up SIP in Cairo

The SIP service is set up in Cairo's resource server thread as it acts as the main hub for receiving and handling the different types of SIP requests. The resource server, represented by the *org.speechforge.cairo.server.resource.ResourceServerImpl* class, implements a SIP session listener interface that allows it to work with the SIP protocol. When the resource server thread is first run, it receives its SIP parameters from the command line launching it, configures its own SIP agent accordingly and launches it, making the Cairo server ready for listening to SIP requests. The SIP agent is the main responsible for sending and receiving SIP requests and responses while the resource server is the one that determines how Halef uses and handles such requests. Once the SIP agent receives an invite request, it calls the resource server's *processInviteRequest* method that continues on to process and act upon such request. Similarly, when the SIP agent receive a bye request, it calls the resource server's *processByeRequest* method that processes and acts upon such request. Both types of requests initialize a sequence of operations to be done by the Cairo server, which are described in detail in the following sections.

2.3.2 SIP Invite Request

As mentioned earlier, once the SIP agent receives an invite request, it calls the resource server's *processInviteRequest* method which then initiates a series of operations through the *invite* method. The *invite* method iterates through the MRCP channels that are included in the Session Description Protocol (SDP) message the invite request is outlined in. It looks for three types of channels; receiver, recorder and transmitter channels. For each channel, its ID is extracted, formatted and included among other attributes in the media description for such channel. This step serves to identify the resources requested by the JVXML server for this SIP session by setting a receiver flag if a receiver or recorder channel is encountered and setting a transmitter flag if a transmitter channel is found. Once the processing of all sessions is done, the two flags are checked and the suitable resources are requested accordingly.

Requesting a Resource

The Cairo server functions in a kind of service based approach, having its different resources running on separate threads or maybe even servers, taking advantage of Java's Remote Method Invocation (RMI) API to glue all the parts together. As mentioned earlier, the Cairo server is composed of three main threads; a resource server, a receiver resource and a transmitter resource. The SIP invite message is received on the resource server thread, but it then needs to request the different resources that are needed for the call, although they are running on other threads. So how is this done? The resource server is always started first as seen earlier. As it starts, it initializes a resource registry through the *org.speechforge.cairo.server.resource.ResourceRegistryImpl* class. This resource registry serves as the connective of all three threads. The way it operates is that the registry is hosted on a server, the resource server, then as every resource is started, it is provided the address of this registry to which it should register its resource object through Java's RMI API. Whenever the resource server then needs a resource of some type, it goes to this resource registry and asks to get a resource object of that type. If a free resource of that type is found, it is given to the resource server which then binds it to the SIP session it wants to initiate by calling the resource's *invite* method, supplying it with the invite request's SDP message in addition to the SIP session's ID.

Requesting a Receiver Resource

Before getting into what happens when a receiver resource is tied to a certain SIP session, let's first see how a receiver resource is structured. The receiver resource has three different object pools; a recognition one, a recorder one and a RTP stream replicator one. However, the recognition one is replaced by a recognition interface that will handle all different types of recognition engines to provide flexibility. It also has a MRCP server socket to communicate with the JVXML server over MRCP once the channels are set up. When the *invite* method of a receiver resource is called, a series of operations is triggered to set up the MRCP channels and borrow the necessary objects from their respective object pools. First, all of the receiver and recorder channels are extracted from the SDP message then the suitable resources are allocated according to each channel's media descriptor. For both types of channels, a RTP stream replicator object is borrowed from its respective pool and the MRCP channel is then registered with the MRCP server socket. If the channel is a recognition channel, it is allocated an instance of the recognition interface to be passed to the *RTPRecogChannel* along with the borrowed replicator instance. On the other hand, if the channel is a recording one, a recorder engine is borrowed from its respective object pool then passed to the *RTPRecorderChannel* along with the borrowed replicator object, the recording directory extracted from the configuration file and a content descriptor describing the media file type in which to save the recorded audio stream (Wave in our case). The way the channel resources are mapped to the session is through the *org.speechforge.cairo.server.resource.ResourceSession* class which maintains a *java.util.Map* mapping the SIP session ID to the MRCP channel ID which in turn maps to the resources bound to each channel. This setup provides access to all channels in a SIP session in addition to all the resources that the Cairo server allocated to each session

even after it's done processing the SIP invite request.

2.3.3 SIP Bye Request

As mentioned earlier, once the SIP agent receives a bye request, it calls the resource server's *processByeRequest* method. This method aims at releasing all of the resources that are bound to the session being closed. It iterates through all of the resources in that session calling the *bye* method of each resource. As we have only two types of resources, transmitter and receiver, we will examine the bye methods for both resources.

Releasing a Receiver Resource

The receiver resource has a few bound resources that should be released once the call is terminated. The SIP session's ID is passed to the *bye* method, enabling it to retrieve the session's resources from the *ResourceSession* class. It then goes through all of the resources bound to the session, signaling the MRCP server to close the opened channels and shutting down the *RTPStreamReplicator*s and returning them to their pool. It finally removes the session from the *ResourceSession* map.

2.4 MRCP Recognition Request Life Cycle

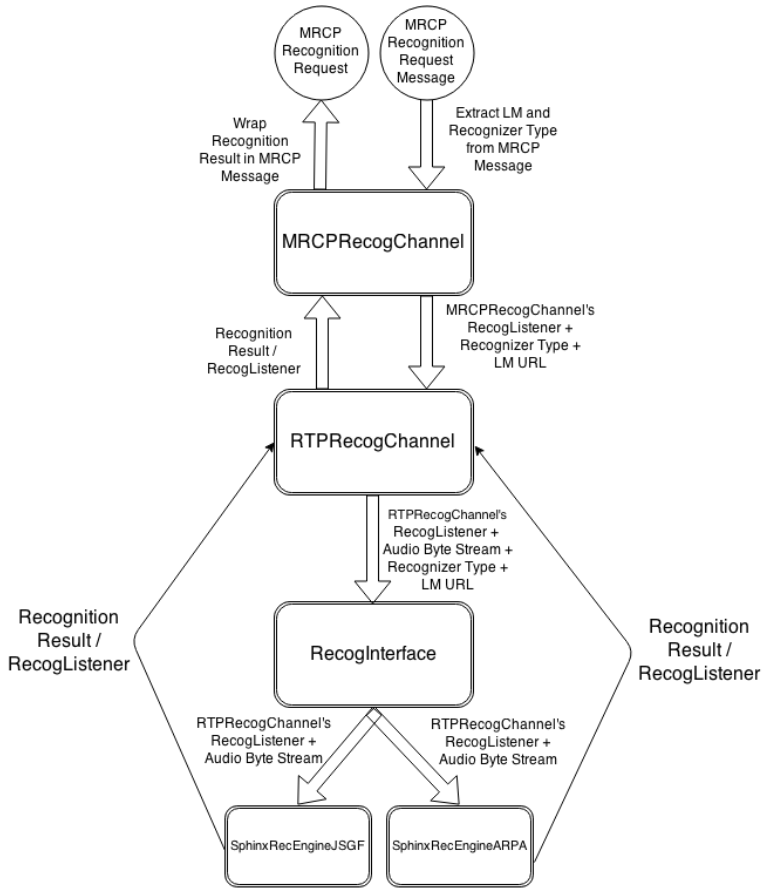

Figure 2.2: MRCP Recognition Request Life Cycle

2.4.1 MRCP Message Parsing and Recognition Preparation

When the MRCP recognition channels are setup and the necessary resources allocated in the SIP invite request processing, the bound receiver resource is then ready to receive speech recognition requests over MRCP. During the channels setup, an object of type *org.speechforge.cairo.server.recog.MrcpRecogChannel* is instantiated and added to the MRCP server of the receiver resource as the manager of the recognition part for such session. When a recognition request is sent by the JVXML server over a certain MRCP channel, it fires the *recognize* method in the respective *MrcpRecogChannel*. This method receives the MRCP message, does some preprocessing accordingly, then delegates the actual recognition part to the allocated *org.speechforge.cairo.server.recog.RTPRecogChannel*. After doing some checks on the current state of this channel and whether this message has any contents, the *invite* method proceeds to parse the recognition request, extracting first the requested recognizer type. The currently supported recognizer types are Sphinx JSGF and Sphinx ARPA recognizers specified in the message as "application/jsgf" and

"application/arpa" respectively. It then proceeds to extract the URL of the language model file to be used in satisfying this recognition request.

Listing 2.6: Extract Application Type and Language Model's URL from MRCP Message

```
1  // check if the MRCP message has any content. If so, extract the
       URL of the language model.
2  if (request.hasContent()) {
3    contentType = request.getContentType(); // recognizer type to
       use
4    if (contentType.equalsIgnoreCase("application/jsgf") ||
       contentType.equalsIgnoreCase("application/arpa") ) {
5      // if it's one of the supported recognizer types, fetch the
         language model's URL.
6      try {
7        grammarLocation = new GrammarLocation(new URL(request.
           getContent()));
8      } catch (IOException e) {
9          statusCode = MrcpResponse.STATUS_SERVER_INTERNAL_ERROR;
10     }
11   }
12 }
```

At this point, the recognition process is delegated to the *RTPRecogChannel* through passing the extracted information from the MRCP message along with a callback listener to the *RTPRecogChannel.recognize* method.
The *RTPRecogChannel* handles all the dirty work involved in the recognition process, setting up data streams and needed recognizers, then calling the suitable methods through the recognition interface to complete the recognition process. It starts by extracting the data stream from the *RTPStreamReplicator* into the format that will be used in all later stages which is *javax.media.protocol.PushBufferDataSource*. It then starts the recognition process through the recognition interface, implemented through the class *org.speechforge.cairo.server.recog.RecogInterface*, by first activating the needed recognizer through calling the *RecogInterface.activateRecEngine* method with the specified application type and the language model's URL. The actual recognition process is started then through the *RecogInterface.startRecognition* method passing to it the previously extracted *PushBufferDataSource* and a callback listener. From this point onwards, the recognition interface will assume leadership of the process, picking the suitable recognizer that the application has asked for, starting recognition through this recognizer, then handing the result back to the *RTPRecogChannel* through the callback listener.

2.4.2 Recognition Interface

In the original implementation of the Cairo server, the Sphinx recognizer was integrated within the whole system which imposed a lot of flexibility restrictions. As the need arose for using other types of recognizers which proved to provide better results, a solution

providing generality and an easy plug-in-plug-out nature had to be implemented, and this is when the recognition interface was born. The recognition interface is basically an added layer of abstraction that separates the management of the actual recognition engines from the rest of the Cairo server. With this addition, all the Cairo server has to do is call the generic methods provided by the recognition interface and then let it handle all the mess of managing which recognizer type to pick, which specific recognition engine to launch and how to pass the data source to this specific recognizer instance. In addition to managing the recognition engines, the recognition interface can also implement several new functionalities like caching the recognition engines that already have a certain language model loaded in them. Let's dive now into the details of how the recognition interface works.

Managing the Different Recognition Engines

The recognition engine needs to have a flexible way of managing the different recognition engines which assumes nothing about which recognizers are in use and which functionality they have to offer. In order to satisfy those purposes, a *java.util.HashMap* is used, mapping the unique recognizer type to recognition engines pool of that type. This *String* to *ObjectPool* mapping is generic enough to represent all keys and recognizer types making adding a new recognition engine to Halef a really easy task. When a recognition request is placed, the recognition interface will go and fetch the required engine from the relative *ObjectPool* fetched from the map.

Listing 2.7: Recognition Interface Application Type to *ObjectPool* Map

```
1 private HashMap<String, ObjectPool> _recPools;
2
3 public RecogInterface(ReceiverConfig config) {
4   _recPools = new HashMap<String, ObjectPool>();
5   initialize();
6 }
```

Initializing the Recognition Interface

The initialization process is pretty straight forward, creating new *ObjectPool*s for the predefined recognizer types. During the current setup, the predefined types are hard coded into the recognition interface with the application type being explicitly specified as *application/jsgf* and *application/arpa*. Going forwards, such predefinition can be changed from hard coding certain types to fetching them from a certain section in the Cairo configuration file, providing more flexibility and saving on memory especially as the number of different recognizers to use becomes bigger. However, if the recognition pool for a certain application type isn't created during the initialization phase, all is not lost. If a certain mapping from application type to a certain object pool is specified in the Cairo configuration file, this pool can then be automatically added to the map and the recognition can start with this new type. This feature as well is not yet implemented, but

it can easily be added in the future, and the way the recognition interface is implemented fully supports that.

Listing 2.8: Recognition Interface Initialization

```
1  private ReceiverConfig _config;
2
3  public void initialize(){
4    try {
5      _recPools.put(
6          "application/jsgf",
7          SphinxRecEngineFactoryJsgf.createObjectPool(
8              _config.getSphinxConfigURL(),
9              _config.getEngines()));
10     _recPools.put(
11         "application/arpa",
12         SphinxRecEngineFactoryArpa.createObjectPool(
13             _config.getSphinxArpaConfigURL(),
14             _config.getEngines()));
15   } catch (InstantiationException e) {
16     e.printStackTrace();
17   }
18 }
```

Activating a certain Recognition Engine and Starting Recognition

When a recognition request is placed by the JVXML server and it reaches the recognition interface as part of the request's life cycle, the recognition interface performs the task of fetching the suitable recognition engine, loading the language model into it, then starting the recognition thread. First of all, the *RecognitionInterface.activateRecEngine* method is called, providing it with the application type and the location of the language model. This method fetches the correct recognition pool, using the application type as the key, from the interface's map. It then requests an available recognition engine instance from that pool to carry out this recognition request. It is now assumed, as forced by Halef's current capabilities, that only a single recognition request will be taking place at a certain time instance and so, the resources management here is done by saving the currently requested recognition engine as well as its pool into a single instance of *org.speechforge.cairo.server.recog.ActiveRecognizer*.

Listing 2.9: Recognition Interface Recognizer Handling

```
1  private ActiveRecognizer _activeRecog;
2
3  public void startRecognition(PushBufferDataSource dataSource,
4      RecogListener recogListener) throws
           UnsupportedEncodingException {
5    _activeRecog.startRecognition(dataSource, recogListener);
```

```
6  }
7
8  public void activateRecEngine(String appType,
9      GrammarLocation grammarLocation, boolean hotword) throws
           Exception {
10   if (_recPools.containsKey(appType) == false) {
11     _logger.error("App type unsupported!");
12     return;
13   }
14   ObjectPool recPool = _recPools.get(appType);
15   _activeRecog = new ActiveRecognizer(recPool, recPool.
        borrowObject(),
16       appType);
17   _logger.debug("Loading grammar...");
18   _activeRecog.loadLM(grammarLocation);
19   _activeRecog.setHotword(hotword);
20 }
21
22 public void returnRecEngine() {
23   if (_activeRecog != null) {
24     _logger.debug("Returning recengine to pool...");
25     try {
26       _activeRecog.returnRecEngine();
27     } catch (Exception e) {
28       _logger.debug(e, e);
29     }
30     _activeRecog = null;
31   } else {
32     _logger.warn("No recengine to return to pool!");
33   }
34 }
```

The *ActiveRecognizer* class handles the specifics of managing the current recognition engine according to its type, explicitly loading the language model in it, starting its recognition thread and finally deallocating its language model and returning it to its respective object pool once the recognition is finalized. From that point till the end of the recognition request cycle, any functionality called to the recognition interface will be passed to the active recognizer to explicitly call the methods to perform such functionality. With the current implementation, the *ActiveRecognizer* assumes knowing the details of the recognizer types that Halef supports and hard codes test cases for each type to work with it differently. However, for actual generality purposes, it would be better to have an interface that specifies how a recognition engine should act in general, then have every recognition engine type implement it. This then would be the way of handling how the *ActiveRecognizer* deals with the required recognition functionality, reaching a fully generic implementation that assumes nothing about the recognizer type.

Listing 2.10: Active Recognizer Class

```
1  public class ActiveRecognizer {
2
3    private ObjectPool _recPool;
4    private Object _recEngine;
5    private String _appType;
6
7    public ActiveRecognizer(ObjectPool recPool, Object recEngine,
        String appType) {
8      _recPool = recPool;
9      _recEngine = recEngine;
10     _appType = appType;
11   }
12
13   public void startRecognition(PushBufferDataSource dataSource,
14       RecogListener recogListener) throws
            UnsupportedEncodingException {
15     if (_appType.equals("application/jsgf")) {
16       ((SphinxRecEngineJSGF) _recEngine).startRecognition(
            dataSource,
17           recogListener);
18       ((SphinxRecEngineJSGF) _recEngine).startRecogThread();
19     } else if (_appType.equals("application/arpa")) {
20       ((SphinxRecEngineARPA) _recEngine).startRecognition(
            dataSource,
21           recogListener);
22       ((SphinxRecEngineARPA) _recEngine).startRecogThread();
23     }
24   }
25
26   public void loadLM(GrammarLocation grammarLocation)
27       throws GrammarException, IOException {
28     if (_appType.equals("application/jsgf")) {
29       ((SphinxRecEngineJSGF) _recEngine).load(grammarLocation);
30     } else if (_appType.equals("application/arpa")) {
31       ((SphinxRecEngineARPA) _recEngine).load(grammarLocation);
32     }
33   }
34
35   public void setHotword(boolean hotword) {
36     if (_appType.equals("application/jsgf")) {
37       ((SphinxRecEngineJSGF) _recEngine).setHotword(hotword);
38     }
39   }
40
41   public void returnRecEngine() throws Exception {
```

```
42      _recPool.returnObject(_recEngine);
43    }
44
45  }
```

Once the recognition engine is loaded into the *ActiveRecognizer* instance, it can be used to initiate the recognition process through the *startRecognition* method. The results are not returned directly by this method, instead, they are returned through the recognition listeners that are passed from the *MRCPRecogChannel* to the *RTPRecogChannel* and then to the recognition engine through the interface. The result is then passed up from each listener to the layer above it, till it finally reaches the *MRCPRecogChannel* which then encapsulates it in a MRCP message and sends it to the JVXML server.

2.4.3 Sphinx Recognition Engine

In this section, a speech recognizer's, namely Sphinx4, support package will be described, also giving an insight on how to integrate any speech recognizer into Halef. Sphinx4 has the advantage of being written in Java and thus, it's easy to use its API to integrate it into Halef whose main development language is also Java. For other recognizers that aren't written in Java, a Java API has to be provided in order to be able to integrate it into Halef, for example using SWIG for recognizers implemented in C/C++, or by providing recognition as a service. The latter however, may need some smart design to create a Java API that simulates an integrated recognition engine to Halef while communicating with the recognition service to obtain the recognition results. Any recognizer to be added to Halef has to have certain properties, the first is being a poolable object and the second is implementing the *SpeechEventListener* interface. The first property guarantees that we can create a pool of several recognition engine objects of that particular recognizer through its factory class, implementing the factory design pattern. The second property is one that gives us a way to handle the different speech signals and also propagate the results to the higher levels of abstraction, namely the *RTPRecogChannel* and subsequently the *MRCPRecogChannel*.

Speech Data Handling

The speech data stream is passed to the recognizer through the *startRecognition* method in the form of a *javax.media.protocol.PushBufferDataSource*. This stream is passed as is from the *RTPRecogChannel* and should be accommodated to fit the needs of the recognizer in use. In the case of Sphinx4, this *PushBufferDataSource* is transformed into a *org.speechforge.cairo.rtp.server.sphinx.RawAudioProcessor* in the *startRecognition* method through the *RawAudioTransferHandler* class. The *RawAudioProcessor* class is then responsible for taking the speech data's byte stream and passing it to the Sphinx4 front end to do the actual recognition.

Listing 2.11: *startRecognition* method

```
1  public synchronized void startRecognition(PushBufferDataSource
       dataSource, RecogListener listener) throws
       UnsupportedEncodingException {
2      if (_rawAudioTransferHandler != null) {
3          throw new IllegalStateException("Recognition already in
               progress!");
4      }
5    // extract speech data byte stream
6      PushBufferStream[] streams = dataSource.getStreams();
7      if (streams.length != 1) {
8          throw new IllegalArgumentException("Rec engine can handle
               only single stream datasources, # of streams: " +
               streams);
9      }
10   // pass the data byte stream into the RawAudioProcesser through
         the RawAudioTransferHandler
11     try {
12         _rawAudioTransferHandler = new RawAudioTransferHandler(
               _rawAudioProcessor);
13         _rawAudioTransferHandler.startProcessing(streams[0]);
14     } catch (UnsupportedEncodingException e) {
15         _rawAudioTransferHandler = null;
16         throw e;
17     }
18     _recogListener = listener;
19 }
```

Speech Event Listeners

As the speech signal is passed as a continuous one over the RTP channel, the Sphinx front-end needs a way to figure out when the actual speech utterance, not emptiness or noise, to be recognized starts and ends. This is done through an instance of the *org.speechforge.cairo.rtp.server.sphinx.SpeechDataMonitor* class. This object's configuration is loaded from the Sphinx configuration file, started in the recognition engine's constructor and passed to it is the recognition engine instance as the *SpeechEventListener*. When a speech start signal is encountered, the *SpeechDataMonitor* broadcasts a speech start signal to all of its associated *SpeechDataListeners* through the *broadcastSpeechStartSignal()* method. Similarly, when a speech end signal is encountered, the *broadcastSpeechEndSignal()* method is used to send a speech end signal to all listeners. In the *speechStarted()* method, the abstract class *SphinxRecEngine* does a very simple function which is setting the call back listener to the *RTPRecogChannel*'s *RecogListener* passed to the recognition engine earlier through the *startRecognition()* method. At this point, we have an abstract *SphinxRecEngine* class, implementing all the common behaviors of any Sphinx recognition engine disregarding its type. We have two different types of recognition engines built over this abstract *SphinxRecEngine* class, one for recognition

using JSGF grammars, namely *SphinxRecEngineJSGF*, and another for recognition using ARPA n-gram language models, namely *SphinxRecEngineArpa*.

SphinxJSGF Recognition

The *SphinxRecEngineJSGF* class is a child of the *SphinxRecEngine* class, implementing the functionality of the Sphinx recognizer for JSGF grammars. The class has an instance of *edu.cmu.sphinx.jsapi.JSGFGrammar* which acts as the grammar to plug in and out of the recognizer for each different recognition request. Before each recognition request is fulfilled, the grammar specified in the MRCP message, described before in section 2.4.1, has to be loaded through the *load(GrammarLocation)* method. This method takes a *GrammarLocation* object as a parameter and loads the specified grammar into the *JSGFGrammar* object.

Listing 2.12: *load()* JSGF method

```
1 protected final JSGFGrammar _jsgfGrammar;
2
3 public synchronized void load(GrammarLocation grammarLocation)
     throws IOException, GrammarException {
4     _jsgfGrammar.setBaseURL(grammarLocation.getBaseURL());
5     _jsgfGrammar.loadJSGF(grammarLocation.getGrammarName());
6 }
```

Once the grammar is loaded, the recognizer is ready to do the actual recognition through the *startRecognitionThread* method. This method will dispatch a separate thread that will initiate and follow up with the recognition process. The *run* method first starts by calling the *waitForResult* method that will ask the recognition engine to do the recognition then get the recognition result.

Listing 2.13: *run()* JSGF method

```
1  public void run() {
2      RecognitionResult result = null;
3      result = SphinxRecEngineJSGF.this.waitForResult(hotword);
4      RecogListener recogListener = null;
5      synchronized (SphinxRecEngineJSGF.this) {
6          recogListener = _recogListener;
7      }
8    // pass on the result upwards through the recognition listener
9      recogListener.recognitionComplete(result);
10 }
```

The *waitForResult* method gives the command to do the recognition through calling the *recognize* method of the *edu.cmu.sphinx.recognizer.Recognizer* instance that this *SphinxRecEngine* has. There are two modes for obtaining the final result, if the *hotword* flag is set, then it keeps asking for recognition results until it reaches a grammatical result, otherwise, only one result is obtained and returned disregarding its grammaticality.

Listing 2.14: *waitForResult()* JSGF method

```
1  private RecognitionResult waitForResult(boolean hotword) {
2      Result result = null;
3
4      //if hotword mode, run recognize until a match occurs
5      if (hotword) {
6          RecognitionResult rr = new RecognitionResult();
7          boolean inGrammarResult = false;
8          while (!inGrammarResult) {
9              result = _recognizer.recognize();
10             rr.setNewResult(result, _jsgfGrammar.getRuleGrammar()
                   );
11             if ((!rr.getRuleMatches().isEmpty()) && (!rr.
                   isOutOfGrammar())) {
12                 inGrammarResult = true;
13             }
14         }
15     } else { //if not hotword, just run recognize once
16         while(result==null || result.getBestFinalResultNoFiller()
               == null || result.getBestFinalResultNoFiller().trim()
              .isEmpty())
17             result = _recognizer.recognize();
18     }
19     stopProcessing();
20     if (result != null) {
21         Result result2clear = _recognizer.recognize();
22     } else {
23     // recognizer returned a null result
24         return null;
25     }
26     return new RecognitionResult(result, _jsgfGrammar.
          getRuleGrammar());
27 }
```

After the recognition result is obtained, it is passed upwards to the *RTPRecogChannel* through the provided *RecogListener* that was passed downwards during the recognition initiation phase by calling the *recognitionComplete* method.

SphinxARPA Recognition

The *SphinxRecEngineARPA* class is a child of the *SphinxRecEngine* class, implementing the functionality of the Sphinx recognizer for ARPA statistic language models. The class has an instance of *edu.cmu.sphinx.linguist.language.ngram.SimpleNGramModel* which acts as the ARPA n-gram model to plug in and out of the recognizer for each different recognition request. Before each recognition request is fulfilled, the language model specified in the MRCP message, described before in section 2.4.1, has to be loaded through the

load(GrammarLocation) method. This method takes a *GrammarLocation* object as a parameter and loads the specified language model into the *SimpleNGramModel* object.

Listing 2.15: *load()* ARPA method

```
1  protected final SimpleNGramModel _nGramModel;
2  protected final LexTreeLinguist _linguist;
3  private PropertySheet _ps;
4
5  public void load(GrammarLocation grammarLocation) throws
       IOException, GrammarException {
6      String file = grammarLocation.getBaseURL().toString() + "/" +
           grammarLocation.getGrammarName() + ".lm";
7      _nGramModel.deallocate();
8      _ps.setString(LanguageModel.PROP_LOCATION, file);
9      _nGramModel.newProperties(_ps);
10     _nGramModel.allocate();
11     _linguist.allocate();
12 }
```

Once the n-gram language model is loaded, the recognizer is ready to do the actual recognition through the *startRecognitionThread* method. This method will dispatch a separate thread that will initiate and follow up with the recognition process. The *run* method first starts by calling the *waitForResult* method that will ask the recognition engine to do the recognition then get the recognition result.

Listing 2.16: *run()* ARPA method

```
1  public void run() {
2      RecognitionResult result = SphinxRecEngineARPA.this.
           waitForResult();
3    RecogListener recogListener = null;
4      synchronized (SphinxRecEngineARPA.this) {
5          recogListener = _recogListener;
6      }
7    // pass on the result upwards through the recognition listener
8      recogListener.recognitionComplete(result);
9  }
```

The *waitForResult* method gives the command to do the recognition through calling the *recognize* method of the *edu.cmu.sphinx.recognizer.Recognizer* instance that this *SphinxRecEngine* has.

Listing 2.17: *waitForResult()* ARPA method

```
1  private RecognitionResult waitForResult() {
2      Result result = null;
3      do {
4          result = _recognizer.recognize();
```

```
5       } while (result != null && result.getBestResultNoFiller().
            trim().isEmpty());
6       stopProcessing();
7       if (result != null) {
8           Result result2clear = _recognizer.recognize();
9       } else {
10      // recognizer returned a null result
11          return null;
12      }
13      RecognitionResult recognitionResult = null;
14      try {
15          recognitionResult = RecognitionResult.
                constructResultFromString(result.getBestResultNoFiller
                ());
16      } catch (InvalidRecognitionResultException ex) {
17        // invalid recognition result
18      }
19      return recognitionResult;
20  }
```

After the recognition result is obtained, it is passed upwards to the *RTPRecogChannel* through the provided *RecogListener* that was passed downwards during the recognition initiation phase by calling the *recognitionComplete* method.

Chapter 3

JVoiceXML Server

JVXML

Chapter 4

Applications

In this chapter, several applications that have been already integrated into Halef will be discussed.

4.1 Voice Enabled Question Answering using OpenEphyra

This was the first application that has been built for Halef's demoing purposes. The application's idea is to integrate OpenEphyra, a question answering system inspired by IBM's Watson, into Halef, thus creating a voice enabled service for question answering. Of course this application had one major challenge, being how to train a generic enough language model to enable the recognizer to effectively recognize questions that have no specific domain. For our demo, we created a small grammar consisting of a few questions and then the user calls in, asks a question out of those, the recognition result is then passed to OpenEphyra to get a suitable answer, then the answer is synthesized and played out to the user.

4.2 Stuttgart's VVS Transportation System

The second application is one that serves the city of Stuttgart, Germany through providing a service to get the next available Verkehrsverbund Stuttgart (VVS) transportation route from one station to another. The user calls in and is asked to provide the name of the route's start station, then is prompted to provide the name of the end station and finally, the route is queried through the VVS API to get the next available route which is then prompted out to the user. The user is prompted the number of the transportation line to take, the timing it leaves from the start station and the timing it arrives at the end station. If the route isn't a direct one, the user is prompted the details of each and every connection in the format explained previously. The API is queried through HTTP GET requests, and it then provides the results in XML form which then needs to be parsed to extract the details of the route, then put it in a suitable format to be passed

as voice output to the user. Since the applications for Halef are written in JVXML, the functionality to deal with the API needed to be written in another language, then the JVXML server will deal with it as a service to query and get the results. The service to deal with the VVS API was made in Java and it actively listens to requests sent to it by the JVXML server as they communicate through JSON objects.

Chapter 5

Conclusion

In the end, here is a wrap up of what Halef is, and how it operates. Halef, the open source, industry standards compliant spoken dialog system, consists of three main servers, namely, the Asterisk call management server, the Cairo server for recognizing, recoding and synthesizing speech and the JVXML server for parsing the applications and issuing the recognition, recoding and synthesis requests to the Cairo server. The user call starts at the Asterisk server, which receives the user call over SIP or PSTN, then routes the user to the correct application according to the entered extension. This call is then connected to the suitable JVXML server which will fetch the correct application file, parse it and execute it. During the application's execution, the required recognition and synthesis requests are then sent over to the Cairo server to carry them out and return their respective results. Speech data is passed between the servers using MRCP over RTP which follows the industry's standards and the applications are developed using JVXML, which is also widely used in speech applications, also following industry standards.

Chapter 6

Ongoing and Future Work

As the development is ongoing in Halef, several new features are being added and a bunch of others are being improved.

6.1 Areas of improvement

Throughout the book, a few points have been pointed out as possible areas of addition or improvement. In this section, those points will be summarized.

6.1.1 Plug in - Plug out Nature for Speech Recognizers and Synthesizers

The Cairo server was developed as a separate project and it shipped with Sphinx4 and FreeTTS integrated into it. This integration goes into the heart of the Cairo server that it makes the job of adding other speech recognizers and synthesizers more difficult than it should be. Ideally, Halef should provide an easy interface that allows to plug in and plug out any speech recognizer or synthesizer easily. This would give Halef maximum flexibility and would allow it to cope with the new technologies as they roll out. The recognition interface described earlier took a step forwards in this direction, achieving some sort of flexibility and allowing the use of two recognizer types instead of one by adding Sphinx4 ARPA to the originally present Sphinx4 JSGF recognizer. However, the job of adding a totally different recognizer other than Sphinx4 is still an ongoing work as will be explained later. During this process, it is expected to meet more areas where the architecture of the Cairo server will have to be changed to provide more separability between it and the Sphinx4 recognizer.

6.1.2 Multiple Recognition Threads

The Cairo server's architecture is great for parallelization and handling multiple threads of recognition, synthesis or recording. However, those features aren't being used yet in the current version of Halef. This is enforced by the implementation of the *ActiveRecognizer* class, for example, which operates based on the assumption that there is only

one recognition thread currently occurring on this server. As the need to scale up grows, multiple recognition threads will definitely be needed and an alternate implementation for the *ActiveRecognizer* class, one that supports multiple concurring recognitions, has to be sought after.

6.1.3 Supporting more VXML tags in JVXML

The VXML Standard 2.1 [1] provides a lot of interesting functionality that makes developing speech-based applications using it even more powerful. However, the JVXML parser doesn't fully support all of the tags and functionality provided by the VXML Standard 2.1. This limits the developers' capabilities when making applications for Halef and may be considered a step back to other commercial softwares.

6.2 New Areas of Development

6.2.1 Integrating the Kaldi Recognizer

There is an ongoing effort to integrate the Kaldi recognizer into Halef, adding the power and effectiveness of DNNs speech processing to Halef. As the effort continues, new limitation points are discovered as Sphinx4 is being decoupled from the Cairo server to make way to different types of recognizers in there. This puts to Halef's ability to accommodate different recognizers to test, being a great measure to how far Halef has gone on the way to reaching the desired plug in - plug out nature mentioned earlier in 6.1.1.

Appendix

Appendix A

Lists

SDS	Spoken Dialog System
JVXML	JVoiceXML
SDP	Session Description Protocol
SIP	Session Initialization Protocol
RMI	Remote Method Invocation
VXML	Voice Extensible Markup Language
PSTN	Public Switched Telephone Network

List of Figures

Listings

Bibliography

[1] W3C Recommendation. Voice Extensible Markup Language (VoiceXML) 2.1. *http://www.w3.org/TR/voicexml21/*, 2007.

[2] Russell Bryant, Leif Madsen, and Jim Van Meggelen. *Asterisk: The Definitive Guide*. O'Reilly Media, 4th edition, 2013.

[3] David Suendermann-Oeft. Modern conversational agents. 2013.

[4] Jonathan Grupp Tim von Oldenburg and David Suendermann-Oeft. Towards a Distributed Open-Source Spoken Dialog System Following Industry Standards. In *Speech Communication; 10. ITG Symposium; Proceedings of*, pages 1–4. VDE, 2012.

[5] Tim von Oldenburg and Jonathan Grupp. Architecture of a Distributed Open Source Spoken Dialog System. Technical report, DHBW Stuttgart, Germany, 2012.