Towards a Distributed Open-Source Spoken Dialog System Following Industry Standards

Tim von Oldenburg[1,2,4,7], *Jonathan Grupp*[1,2,5], *David Suendermann*[1,3,6,8]

[1]DHBW, Stuttgart, Germany [2]IBM, Böblingen, Germany [3]SpeechCycle, New York, USA
Email: [4]`tim@voldenburg.com`, [5]`jonathan.grupp@gmail.com`, [6]`david@suendermann.com`
Web: [7]`www.tvooo.de`, [8]`www.suendermann.com`

Abstract

This paper outlines an architecture of a distributed spoken dialog system based exclusively on open-source components. These components comprise telephony, speech recognition, voice browser, and speech synthesis, all of which are to incorporate industrial standards as recommended by the W3C. Our work demonstrates that real-world spoken dialog systems can be built without use of proprietary components.

1 Introduction

Spoken dialog systems have been subject to academic research for quite some time. Beginning in 1990, substantial progress in the domains of spoken language understanding and dialog management has been made in the scope of the DARPA projects ATIS [1] and Communicator [2]. The evaluated systems focused on a specific domain of interest (flight scheduling), and, even though this domain sounds suitable for commercial use, the projects resulted in no considerable live deployments. One of the reasons was the high sophistication of the underlying models of speech recognition, understanding, generation, and synthesis as well as dialog management which made design and maintenance of these systems very difficult.

On the other hand, at the beginning of this century, a number of industrial players (AT&T, Sun, SpeechWorks, Nuance, among others) started to code spoken dialog systems adhering to fairly simple design paradigms (such as strongly limited vocabulary, predominantly yes/no questions, speech menus with only a few alternatives, rule-based dialog managers, prerecorded prompts, flat semantic structure) [3]. Even though these systems were not as "intelligent" and sophisticated as their academic counterparts, they had two main advantages: They were easy to build and worked surprisingly well. Even more so, when multiple industrial players sat together and agreed on common standards for speech grammars [4], semantic interpretation [5], language models [6], and dialog management by way of voice browsers [7].

While industry agreed on standards to produce components (theoretically) exchangeable among each other, the components themselves remained proprietary to each vendor. License fees for commercial software such as speech recognizers or voice browsers can be considerable, especially when multiple telephony ports are concerned. The goal of our current work is to build a multi-port spoken dialog system infrastructure adhering to the above mentioned industrial standards based exclusively on open-source software components. In doing so, we incorporate (where possible) available open-source software from diverse origins as described in the following sections.

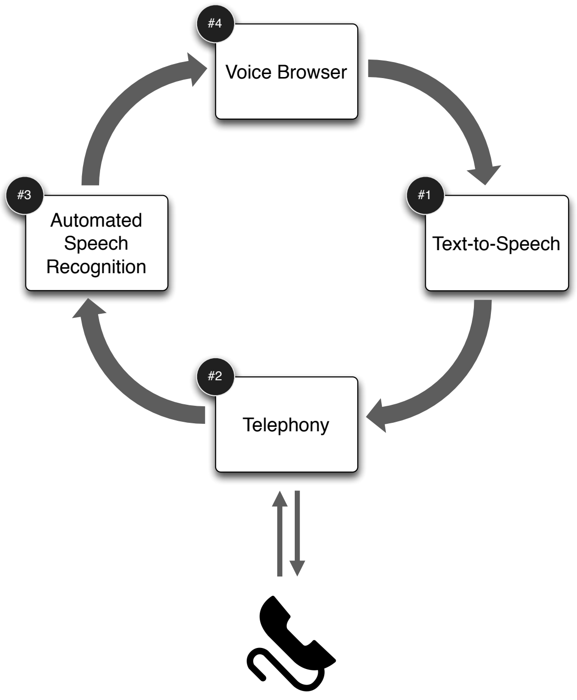

Figure 1: High-level architecture of a spoken dialog system

2 General Architecture

The spoken dialog system described in the following consists of four main components (see Figure 1) communicating with each other over the software stack illustrated in Figure 2:

1. a telephony server to handle calls,
2. a speech recognizer (ASR),
3. a voice browser, and
4. a text–to–speech synthesizer (TTS).

Even though some (or all) of these components may reside on a single server, in industrial settings, they are commonly distributed among several network nodes. This makes especially sense when more calls should be handled than a single server node can process. Sometimes, components are even distributed among facilities thousands of miles apart or hosted in a cloud computing environment. In order for our architecture to be able to work in the same fashion, we installed components on different network nodes in a virtualized environment.

3 Implementation

3.1 Software Stack

We use the Linux operating system on all servers which is open-source according to the GNU General Public License (GPL) and other licenses. Ubuntu Server is one of the most popular Linux distributions and is commercially supported by Canonical making it an excellent choice for professional environments. Ubuntu comes with the Advanced Linux Sound Architecture (ALSA) to process audio. In our scenario of virtual machines where no physical sound hardware exists, ALSA uses a dummy sound driver to emulate a hardware device. Depending on the component's task, one of the audio frameworks *PulseAudio* or *JACK Audio Connection Kit* is used in order to complement functionality on top of *ALSA*. Both frameworks are distributed under the GNU Lesser Public License (LGPL).

Textual data are transferred using TCP, whereas audio is streamed via UDP. Encoding, decoding, and streaming are done with the *GStreamer* framework. A diagram of the software stack is given in Figure 2.

Software Versions

In order to make it easier to build upon and recreate our work, we listed all used software and their versions in Table 1.

Software	Version	Purpose
Ubuntu	11.10	operating system
Asterisk	1.8.4.4	PBX server
PocketSphinx	0.7	speech recognizer
Festival	2.1	speech synthesis
GStreamer	0.10	audio streaming
PulseAudio	1.0	interface GStreamer/Sphinx
JACK	1.9.6	interface GStreamer/Asterisk
ALSA	1.0.23	interface GStreamer/Festival

Table 1: Software versions used in our setup

3.2 Audio Coding and Transportation

Conventional narrow-band telephony uses a sampling rate of 8kHz. Festival comes with voices of either 8kHz or 16kHz (wideband). To allow for a simple setup, we chose 8kHz as the sampling rate of audio in the whole environment. The bit depth was chosen to be 16 bits per sample. Audio is not being compressed, but transferred as RAW PCM data. As we are transporting voice, a single channel (mono) is sufficient.

For network transportation, the Real-time Transport Protocol (RTP) is used. RTP is an industry standard, usually employed in VoIP systems. It is often used in combination with the Session Initiation Protocol (SIP, see Section 3.3), for example in the Asterisk PBX server. To ensure the quality of service (QoS) required by a spoken dialog system using VoIP, RTP powers different features, such as padding and timestamps. The RTP plugin for GStreamer comes with a jitter buffer, but its use may not be recommended as it leads to a longer runtime of the audio transport.

The GStreamer multimedia framework is used for all encoding, decoding, transformation and, streaming tasks. GStreamer is open-source according to the LGPL, but the plug-ins that contain audio and video codecs may be licensed differently. GStreamer uses a pipeline concept; it takes an input and then forwards this input to every step in the pipeline, one after the other. The steps are, for example, encoding, decoding, multiplexing, demultiplexing, or format conversion.

3.3 Components

Telephony Server

The widely used *Asterisk* is employed as a telephony server. It is open-source according to the GPL and can handle a wide range of telephony protocols for private branch exchange (PBX) services. In order to do so, Asterisk translates all protocols into the same, internally used protocol. This protocol is called IAX2 and can also be used to connect separate Asterisk systems with each other. Asterisk is a multi-threaded server which could enable a spoken dialog system to handle multiple calls simultaneously. Asterisk's command line interface, configuration files (*sip.conf* and *extensions.conf*) and its powerful dialplan script allow to take full control of the server. Our implementation relies on SIP and RTP to establish and conduct phone calls.

In order to interface Asterisk with other components, we use the Asterisk JACK dialplan application which then allows us to stream the caller's audio to the ASR node and receive the generated audio from the TTS component. Unfortunately, at the time of writing these lines, Asterisk's JACK application was not yet functioning properly. Options how to address this issue are discussed in Section 3.4.

Automated Speech Recognition

Speech recognition is handled by *Sphinx* [8], an open-source toolkit by the Speech Group at Carnegie Mellon University in Pittsburgh, USA. The GStreamer framework provides a plugin for Sphinx that allows to transcribe the recognized text directly from any audio source GStreamer can handle.

We developed a Python program that constructs a permanent GStreamer pipeline listening for an incoming audio stream. When audio from the telephony component arrives, it gets transcribed. Sphinx comes along with a voice activity detector (VAD). The recognized text chunks are sent via TCP connection to the voice browsing component.

Voice Browser

We are currently exploring several alternatives for open-source voice browsers including OpenVXI (a library originally developed by SpeechWorks and now maintained by Vocalocity) or building a browser from scratch.

The interfaces to both ASR and TTS on the other hand, are already defined and kept simple. The interface between voice browser and ASR acts as a TCP server, waiting for connections. If a client connects and sends a text to be evaluated, this text gets forwarded to the actual voice browser. The interface to TTS acts as a TCP client. A response coming from the voice browser is sent to the TTS. Hence, the TCP server and client together act as a wrapper around the voice browser.

Speech Synthesizer

As speech synthesizer, the open-source software *Festival* [9] is used. This synthesizer is developed at the *Center*

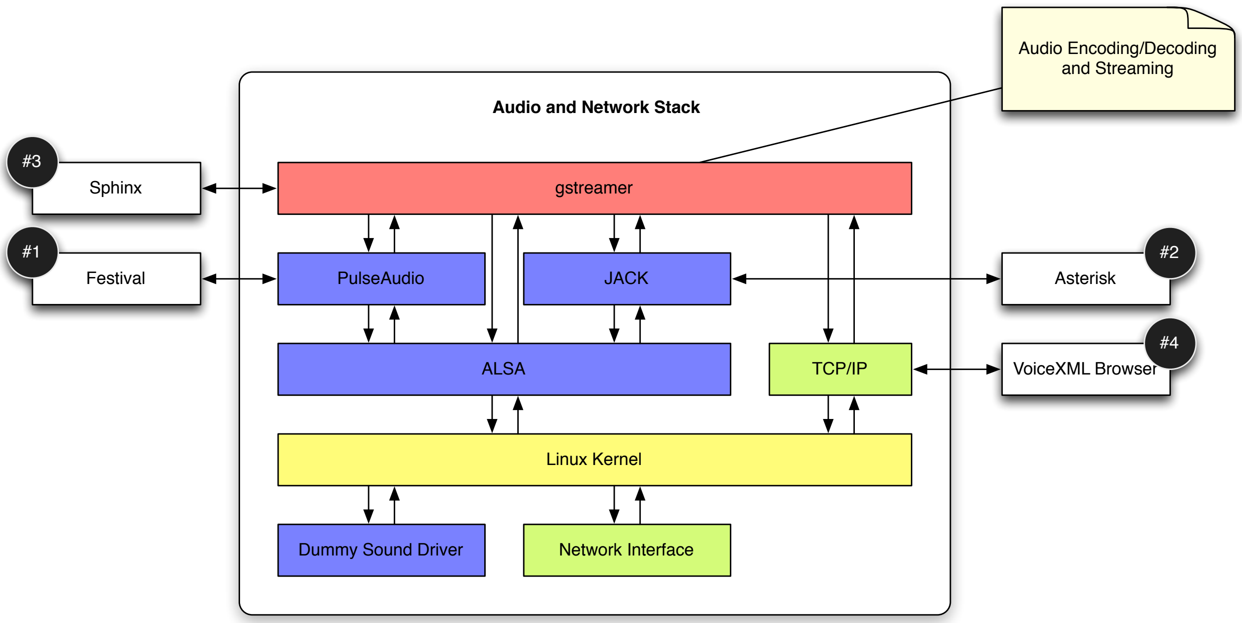

Figure 2: Diagram of the underlying software stack

for Speech Technology Research at the University of Edinburgh, UK, and is distributed under a free X11-type license which grants unrestricted commercial and non-commercial use.

Festival's speech engine is invokable using an interactive command-line interface (which supports scripting via the Scheme language). It can also process text from the operating system's standard input (STDIN), and it can run as a TCP server that synthesizes text sent by connected clients. Under the first impression, the TCP variant seems to be a good choice, as the synthesizer could be invoked by a foreign host—such as a voice browser. The downside to this approach is that it would be more difficult to implement further logic into the process, such as controlling multiple streams and ensuring security. Thus, we decided to implement a TCP server on our own using Python, which then invokes Festival. This way, further features can more easily be implemented.

Festival can only output the synthesized speech to the system's standard audio output (an ALSA playback channel) or write it into a .WAV file. As we need to stream the audio back to Asterisk, this is quite problematic. Writing a file and reading the file back from a hard disk is not an option, as it would cause too much lag. But it is also not directly possible to make use of the signal on the system's audio output, as it is treated by ALSA as a playback channel.

There are two solutions to this problem. The first one includes setting up the PulseAudio framework, which works on top of ALSA and provides a so-called "monitor device" that loops playback back to a virtual recording device (input device). The audio input can then be further processed, for example by GStreamer. This solution would set up a permanent GStreamer pipeline, streaming all audio that is played on the system to the telephony server.

The second solution makes use of the Festival plugin for GStreamer. With every chunk of text that needs to be synthesized, a new GStreamer pipeline is invoked that uses Festival to synthesize the text and then directly forwards it to the telephony component. This solution is preferable, as PulseAudio is not needed, and there is no permanent pipeline running. Not needing PulseAudio simplifies the software stack significantly, as it is rather complex to set up PulseAudio and its dependencies in a headless (i.e. virtual, lacking a graphical user interface) environment. This solution could not be tested yet, although it is promising.

3.4 Issues

The only issue with the proposed architecture is the JACK dialplan application used to interface Asterisk and JACK. This application is provided by Asterisk, and its documentation can be found in Asterisk's user guide. However, when properly using the JACK dialplan application, it silently fails, producing neither errors, warnings nor debugging information in both the Asterisk command line interface and the JACK daemon process. To cope with this issue, a newer version (see Table 1) of Asterisk could be deployed hoping that this issue will then be resolved. In case this will not result in the desired outcome, different private branch exchange (PBX) servers could be evaluated in respect to their connectability to JACK or other audio frameworks and libraries which can be set up to work with GStreamer (see Figure 2).

4 Conclusion and Future Work

This paper shows that it is possible to build a distributed spoken dialog system based solely on open-source components. There are still some issues open with the current setup, in particular the interface between Asterisk and GStreamer and the choice of a voice browser, but both are issues that can be adressed and solved.

This implementation assumes a simplistic scenario with only one call at a time. In a real-world scenario, this will rarely be the case. Instead, spoken dialog systems are required to be capable of handling multiple calls at once (multi-port spoken dialog systems).

To meet the requirements of a real-world scenario, a fifth component must be developed that acts as a supervisor to the other ones. It has to work closely together with the telephony server and must handle different calls separately. It would do so by adding a control layer on each network communication so that every instance of streamed, sent, or processed text or audio is assigned to a specific communication.

This is where the advantage of a distributed and open implementation gets obvious: The system has a high degree of scalability. If one of the servers is overloaded, the supervisor can route traffic over a different server with the same functionality. To achieve this, the supervisor component would also act as a load balancer.

Another addition to the system would be a security infrastructure to ensure encrypted and safe network communication. In a professional environment, this feature would be inevitable. The great advantage of using open-source software is that existing software can easily be integrated with the current setup. The load balancer BalanceNG, for instance, offers paid technical support and thus would be a good choice in productive environments. Security, on the other hand, could be handled via the OpenSSL library.

Fixing the existing issues and developing a supervisor component are the next steps towards a distributed, open-source spoken dialog system that can compete with commercial solutions.

References

[1] C. Hemphill, J. Godfrey, and G. Doddington, "The ATIS Spoken Language Systems Pilot Corpus," in *Proc. of the Workshop on Speech and Natural Language*, (Hidden Valley, USA), 1990.

[2] M. Walker, J. Aberdeen, and G. Sanders, *2001 Communicator Evaluation*. Philadelphia, USA: Linguistic Data Consortium, 2003.

[3] D. Suendermann, *Advances in Commercial Deployment of Spoken Dialog Systems*. New York, USA: Springer, 2011.

[4] A. Hunt and S. McGlashan, "Speech Recognition Grammar Specification Version 1.0. W3C Recommendation,"

[5] L. Tichelen and D. Burke, "Semantic Interpretation for Speech Recognition (SISR) Version 1.0. W3C Recommendation,"

[6] M. Brown, A. Kellner, and D. Raggett, "Stochastic Language Models (N-Gram) Specification. W3C Working Draft,"

[7] S. McGlashan, D. Burnett, J. Carter, P. Danielsen, J. Ferrans, A. Hunt, B. Lucas, B. Porter, K. Rehor, and S. Tryphonas, "VoiceXML 2.0. W3C Recommendation,"

[8] K. Seymore, S. Chen, S. Doh, M. Eskenazi, E. Gouvea, B. Raj, M. Ravishankar, R. Rosenfeld, M. Siegler, R. Stern, and E. Thayer, "The 1997 CMU Sphinx-3 English Broadcast News Transcription System," in *Proc. of the DARPA Broadcast News Transcription and Understanding Workshop*, (Lansdowne, USA), 1998.

[9] P. Taylor, A. Black, and R. Caley, "The Architecture of the Festival Speech Synthesis System," in *Proc. of the ESCA Workshop on Speech Synthesis*, (Jenolan Caves, Australia), 1998.