Architecture of a Distributed Open Source Spoken Dialog System

Tim von Oldenburg, Jonathan Grupp

June 2012

Contents

Abbreviations

ALSA Advanced Linux Sound Architecture

ASR Automated Speech Recognition

CLI Command Line Interface

D-Bus Desktop Bus

GPL GNU General Public License

IAX2 Inter-Asterisk eXchange

IPC Inter-Process Communication

IVR Interactive Voice Response

JACK Jack Audio Connection Kit

LAN Local Area Network

LGPL GNU Lesser General Public License

PA PulseAudio

PBX Private Branch Exchange

PCM Pulse-Code Modulation

RTCP RTP Control Protocol

RTP Real-time Transport Protocol

SDS Spoken Dialog System

SIP Session Initiation Protocol

TTS Text–to–Speech

VM Virtual Machine

VoIP Voice over IP

WIMP Window–Icon–Menu–Pointer

Chapter 1

Motivation & Introduction

The purpose of this study thesis is to develop an architecture, evaluate components and implement a prototype for an open source Spoken Dialog System (SDS).

Dialog systems have already been around for a long time to serve human–computer interaction. Having started with textual command-line interpreters, they nowadays have different forms, for example Window–Icon–Menu–Pointer (WIMP) or hypertext. In telephony, recorded speech has found its application in Interactive Voice Response (IVR) systems. IVR systems allow callers to navigate through a menu using the telephone's keypad or when more advanced with their voice.

With the development of Spoken Dialog Systems the time of pre-recorded messages and navigation using the keypad was over. A Spoken Dialog System features both speech output and input. Using speech synthesizers, language does no longer have to be recorded, and using speech recognition techniques, a caller can talk to the system naturally. In an optimal case, a human would talk to a system as he would talk to another human thus greatly improving human–computer interaction.

1.1 Open Source Software

Open Source software gives us the possibility to build a SDS with highest flexibility. Although it requires knowledge of low-level software, networking and audio codecs, an open source SDS allows both scientific and commercial uses without being tied to a proprietary solution. The easy access to source code makes a system adaptable to specific needs, and as long as the open source license does not restrict the use, the software can be used for commercial software, too.

1.2 Scope of this Study Thesis

This study thesis is a common work of two students, Jonathan Grupp and Tim von Oldenburg. We concentrated on the network and audio infrastructure, speech synthesis and telephony, whereas Ramin Safarpour, a third student, concentrated on the voice browser and speech recognition to complement our work. To assure comprehensibility, Safarpour's elements are also partly presented in this study thesis, but not in great detail. For implementation details and other detailed information please refer to Ramin Safarpour's study thesis.

The initially desired outcome — building a prototypical SDS — could not be achieved in the designated period of time, since many unforeseen problems arose which kept us from meeting our schedule. The prototypical implementation in chapter 4 therefore is not a complete SDS however it introduces possible components, problems with the implementation and entry points and further fields of investigation for future research on building an open source based SDS.

The outcome is, in general, a showcase on how to implement such a system in the real world. Both the planned goal and the implementation do not consider features like *multiple callers with separate channels* or *signaling to ensure a correct call flow*. Those features would be required in a real business case but are not covered by this study thesis and require further research.

Chapter 2

General Architecture of Spoken Dialog Systems

A SDS usually consists of four components:

1. A Text–to–Speech (TTS) engine
   This synthesizer transforms textual data, e.g. a sentence, to speech. In the case of a SDS, this could be an answer to a caller's question or a prompt to provide additional information. Using a speech synthesizer instead of recorded speech allows individual responses to be formulated instead of only having standardized pre-recorded answers.

2. An interface to the caller (telephony server)
   A telephony server is used to handle the incoming calls and redirect the calls to the appropriate SDS components. It might be a Voice over IP (VoIP) server, but it could also be hooked up to an ISDN line.

3. An Automated Speech Recognition (ASR) engine
   As the caller's input is no longer given via a telephone's keys but using his speech, a software is needed to "recognize" it, ie. transform it into textual data that can then be processed by the voice browser.

4. A voice browser
   The voice browser is the component that controls the call flow. It decides what action to do and what answer (in text form) to give based on the textual representation of the caller's input.

These components work together to form a SDS. To make them easily distinguishable, we give every component a number (1–4).

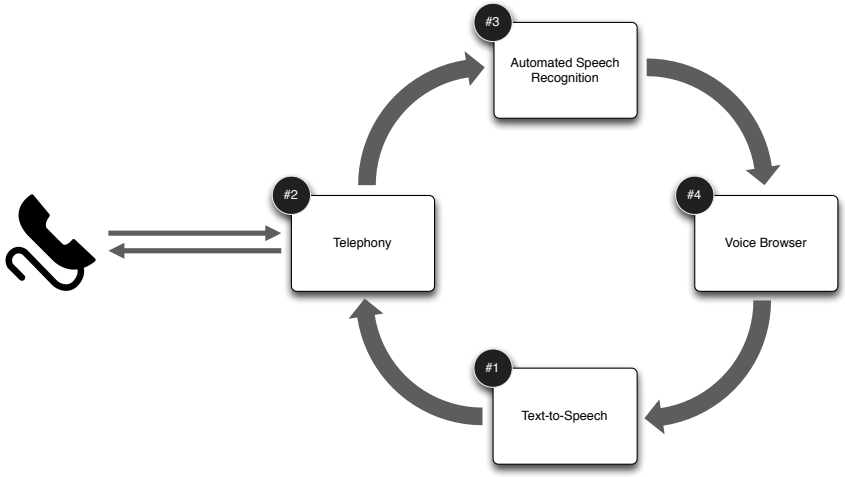

Figure 2.1: General SDS Architecture

The arrows in Figure 2.1 represent interfaces between those components. We define an interface as a point where data – be it textual or audio – are transferred from one component to another.

2.1 Call Cycle

From a technical point of view, the call process of a SDS is a cycle. The following steps are taken with every cycle:

1. Receiving of the caller's speech by the telephony server
2. Speech recognition (transformation into textual data)
3. Selection of a pre-defined reaction (textual data) by the Voice Browser
4. Synthesizing of the reaction text to speech
5. Sending the synthesized reaction back to the caller's phone

Chapter 3

Open Source Components

For every component of the SDS, we set up a Virtual Machine (VM) with the same operating system and similar network and audio stacks. This distribution allows for high flexibility, as the locations of the components are independent of each other and may even be spread over different continents. Also, a component can easily be swapped for a different solution, as long as the interfaces to the other components persist.

This chapter presents reasons for choosing the software we chose and explains the functionality of all software components in detail. We give every VM the same number as the SDS component it employs.

3.1 Software Versions

In order to make it easier for future researchers to build upon and recreate our work, we listed all used software and their versions in the subsequent tables:

Software	Version	Purpose	Source
Ubuntu	11.10	Operating System	Ubuntu Website
Asterisk	1.8.4.4	PBX Server	Ubuntu Repositories
Festival	2.1	Speech Synthesis	Ubuntu Repositories
gstreamer	0.10	Audio Streaming	Ubuntu Repositories
pulseaudio	1.0	Interface gstreamer/Sphinx	Ubuntu Repositories
jackd	1.9.6	Interface gstreamer/Asterisk	Ubuntu Repositories
ALSA	1.0.23	Interface gstreamer/Festival	Ubuntu Installation

Software	Get version from shell
Ubuntu	`cat /etc/*-release`
Asterisk	`sudo asterisk -version`
Festival	`festival --version`
gstreamer	`gst-launch --version`
pulseaudio	`pulseaudio --version`
jackd	`jackd --version`
ALSA	`cat /proc/asound/version`

3.2 Installation

The open source software used can be installed using Ubuntu's standard package manager APT.

Listing 3.1: Installing packages on Ubuntu using apt-get

```
1 $ sudo apt-get install asterisk festival festvox-16k dbus
    pulseaudio jackd gstreamer0.10-tools gstreamer0.10-plugins-base
     gstreamer0.10-plugins-ugly gstreamer0.10-plugins-ugly-
    multiverse gstreamer0.10-plugins-good gstreamer0.10-pulseaudio
```

This command will install all the packages needed, including dependencies. Look at the table below for details.

Package	Software included
asterisk	Asterisk
festival	Festival
festvox-16k	16kHz voices for Festival
dbus	D-Bus
pulseaudio	PulseAudio
jackd	JACK
gstreamer0.10-tools	GStreamer tools, e.g. `gst-launch-0.10`
gstreamer0.10-plugins-base	GStreamer base plugins
gstreamer0.10-plugins-ugly	GStreamer MAD plugin (MP3 decoding)
gstreamer0.10-plugins-ugly-multiverse	GStreamer LAME plugin (MP3 encoding)
gstreamer0.10-plugins-good	GStreamer RTP plugin
gstreamer0.10-pulseaudio	GStreamer PulseAudio plugin

3.3 Operating System

A Linux-based operating system provides everything we need in an operating system for the SDS. The Linux Kernel [1] is Open Source under the GNU General Public License (GPL) v2 and can thus be used for commercial purposes. The wide availability of Linux-based software and the high spread in industrial applications favor Linux as the operating system for an Open Source SDS.

A Linux-based operating system is called a "distribution". We choose the Ubuntu Server [2] distribution because of its stability and great community support.

3.4 Network and Audio Stack

The network and audio stacks are designed to process the audio and network traffic of the dialog system. The components can be categorized in (virtualized) hardware, the Linux Kernel (drivers) and user-space software. Network and audio stacks are consistent across the different VMs, whereas not all elements are required on every VM.

Figure 3.1 shows an overview on the network and audio stack and how the

different components interact with each other.

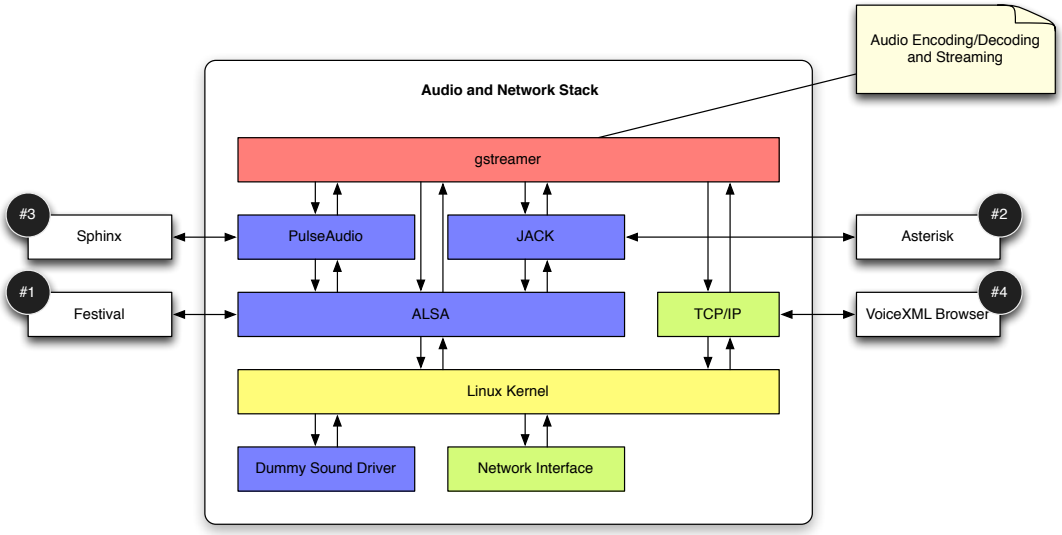

Figure 3.1: Network and Audio Stack

3.4.1 Audio Stack

ALSA

All components except the voice browser need to process audio. A commonly used framework for audio processing within Linux is Advanced Linux Sound Architecture (ALSA) [3], which is licensed as open source according to the GPL and GNU Lesser General Public License (LGPL).

ALSA capable devices are supported as Linux Kernel modules. As there is no actual audio hardware device available in a virtual machine, the dummy driver `snd-dummy` is employed. It can be loaded as superuser[1] using:

Listing 3.2: Loading the dummy sound driver

```
1 $ modprobe snd-dummy
```

Once loaded, ALSA settings – such as volume for the various channels – can be controlled using tools like `amixer` or `alsamixer` and the audio can be played and recorded through the device. Also, frameworks that work on top of ALSA now can make use of the device.

[1] A superuser (also called "root") is a Linux user that has full administrative rights.

PulseAudio

The PulseAudio framework sits on top of ALSA and adds a lot of functionality and features to it. One of the features we make use of is a so-called monitor device. Monitor devices enable us to send the output of a playback channel to the input of a record channel. Thus we can further process the signal. This is especially important for VM #1, as Festival is only able to output synthesized speech to a playback channel or to record it to disk. Using a PulseAudio monitor device, we can loop back Festival's output and send it over the network to the telephony server.

PulseAudio is open source software according to the LGPL.

JACK

The Jack Audio Connection Kit (JACK) can use both ALSA and PulseAudio as backends. JACK is a powerful framework for realtime audio processing and is often used in Digital Audio Workstations (professional recording and mixing software) and synthesizers. JACK is the only framework Asterisk has a connector to.

JACK is open source software according to the LGPL and GPL and can be run using:

Listing 3.3: Starting the Jack Daemon

```
1 $ jackd -d alsa
```

GStreamer

All encoding, decoding and transform steps as well as streaming is done by the GStreamer framework [4].

GStreamer can handle various audio and video codecs and devices. Devices are either "sinks" or "sources". A source for instance may be a media file (read), an incoming network connection streaming audio or video, a recording device or any other representation of media *input*. A sink on the other hand can be a media file too (write), an outgoing network connection, a playback device or any other form of media *output*. The sequence of coding and converting steps that happen from a source to a sink is called the "pipeline". Figure 3.2 shows the diagram of a pipeline that could be used to play an audio file.

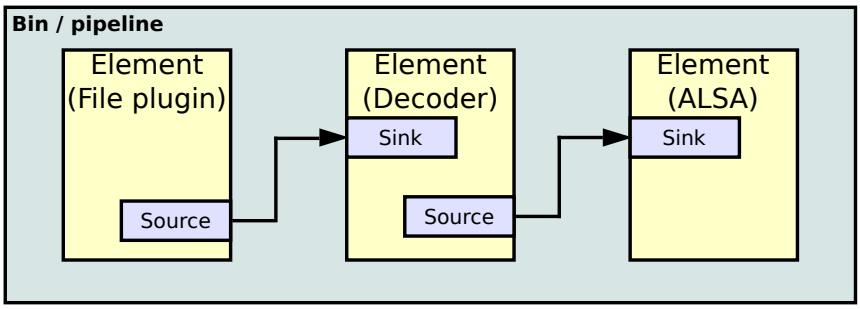

Figure 3.2: GStreamer Pipeline[11]

The first pipeline element is always a source, whereas the last one is always a sink. Every element in between has to include both a source and a sink — it takes the input from the previous element, processes it and passes the output to the next element.

A GStreamer pipeline can be constructed and launched via the command-line program `gst-launch` (on our Ubuntu VMs this program is called `gst-launch-0.10`). It takes the pipeline as an argument. The different elements of the pipeline are connected using an exclamation mark.

The example pipeline from Figure 3.2 can be constructed the following way:

Listing 3.4: Example GStreamer Pipeline

```
1 $ gst-launch filesrc location=/path/to/file.mp3 ! mad !
    audioconvert ! audioresample ! alsasink
```

The `audioconvert` and `audioresample` pipeline elements are used to automatically transform the input and output formats of different sinks and sources. In this example, the output of the `mad` element is transformed into the input format for the `alsasink`.

The pipeline concept of GStreamer is really powerful, allowing you to mix different sources (e.g. a webcam video and a radio stream). However, as we do not need video in this project, the pipelines we use are kept quite simple.

The GStreamer framework is open source according to the LGPL, but the plug-ins that contain audio and video codecs may be licensed differently.

3.4.2 Network Stack

The Linux virtual machines employ virtual network devices which make them capable to support generic TCP/IP traffic. In this spoken dialog system, we need both TCP and UDP traffic.

TCP is used for textual data:

- The caller's speech after being recognized by the ASR
- The voice browser's generated answer before being synthesized by the TTS

UDP is used for streaming audio:

- Synthesized speech being streamed to the telephony server
- Human speech being streamed to the ASR

3.4.3 Audio and Packet Format

Media that is processed by a computer is often encoded using either a lossy or lossless codec. This means, raw audio or video data (or both) are compressed using a certain format. When it comes to realtime streaming over the network, usually a format is chosen which supports loss of packets during the transfer.

There are several codecs that are a good fit for a SDS, for example the patent-free Speex codec which is especially developed for compressing speech. However, we chose not to deploy a codec but to used the raw Pulse-Code Modulation (PCM) [6] audio format instead. This eliminates the need for a codec to be installed on the source and sink server and nowadays bandwidths can easily stream it.

The voices used by Festival usually have a sampling rate of 8kHz (telephone quality, narrowband), some 16kHz (wideband), which is small enough to transport over an ethernet Local Area Network (LAN) in real time. As the quality equals — or is even better than — telephone quality, it is also sufficient for transporting the caller's speech.

For the packet format we also looked for industry standards. The format and protocol usually employed in VoIP systems is called Real-time Transport Protocol (RTP) [5]. RTP is often used in combination with SIP, for example in the VoIP server Asterisk, which is also used in our scenario. Both protocols will be discussed in section 4.2.

GStreamer can handle raw PCM audio, convert between different sampling rates and bit depths, and provides several tools for handling RTP [7], for example a jitter buffer.

The audio format we decided to use is RTP with a payload of 1-channel raw PCM, a sampling rate of 16kHz and a bit depth of 16 bits per sample (see Table 3.1.

3.5 Speech Synthesizer

For speech synthesis we employed the open-source software **Festival** [10]. This synthesizer is being developed and maintained at the Center for Speech Tech-

Codec	Pulse-Code Modulation
Channels	1 (mono)
Sampling Rate	16kHz
Bits per Sample	16
Packet Format	RTP

Table 3.1: Proposed audio format

nology 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 comes with an interactive Scheme interpreter and with a non-interactive command-line interface. Unfortunately, Festival can only output its generated audio to the systems standard audio output or to an audio file. Since we want to send the synthesized speech to the caller over Asterisk, we needed to redirect the output from the standard audio output. This is done using a pulseaudio monitor device (see 3.4.1 and 4.1.4).

Festival comes with British and American English male voices (amongst others) in 8 and 16kHz formats.

To use the synthesizer, you can invoke the command-line interface using either `festival --tts filename` with a text file as input, or by piping the input into the Festival process using `echo "Hello, World!" | festival --tts`. Invoking Festival without the `--tts` argument opens the interactive Scheme interpreter, where you can synthesize text using the function `(SayText "Hello, World!")`.

3.6 Telephony Server

The widely used **Asterisk** [8] is employed as a telephony server. It is open source according to the GNU General Public License (GPL) and can handle a wide range of telephony protocols for Private Branch Exchange (PBX) services. Asterisk is supported by the Digium company, which is situated in Huntsville, Alabama.

The handling of Asterisk will be discussed in section 4.2.

3.7 Automated Speech Recognition

Automated Speech Recognition is handled by **Sphinx** [9], a toolkit by the Speech Group at Carnegie Mellon University in Pittsburgh, USA. Sphinx is

open source according to a BSD-style license. For further information Ramin Safarpour's study thesis should be consuluted.

3.8 Voice Browser

The voice browsing functionality is planned to be implemented using the W3C recommended web-standard *VoiceXML*. For further information Ramin Safarpour's study thesis should be consuluted.

Chapter 4

Prototypical Implementation

During the project phase we were confronted with many unpredicted problems, as nobody else had ever tried to build a distributed open–source source, spoken dialog system before. This chapter is about documenting those problems as well as presenting the results of our research and tests.

4.1 VM #1 – Speech Synthesis

The component #1 implements speech synthesis. As outlined in Chapter 3, this component has to take a simple string of text from VM #4 (voice browser), to synthesize it and stream it to VM #2 (telephony server). This requires a rather complex setup which we created using several startup scripts that will be explained in the following subsections. This component — as it is discussed here — is complete and can be used in a fully functional Spoken Dialog System with respective parallelization and flow control. In other words: no unsolved problems exist.
The scripts have to be executed in the order implicated by their names:

1. Clean the environment using `00clean.sh` as root
2. Start D-BUS using `01dbus.sh` as root
3. Start PulseAudio using `02pa.sh` as user
4. Start streaming using `03stream.sh` as user
5. Open the TCP server listening for text using `04listen.py` as user

Except one, all of these scripts are Bash[1] scripts; the last one is a Python program. Let's have a look at the different scripts and their tasks.

4.1.1 Cleaning the environment

This step is very important so ensure a clean environment. The script makes sure that no process of D-BUS, PulseAudio and no GStreamer pipeline is running. Mixing up multiple instances of D-BUS and PulseAudio can cause unpredictable behavior, including GStreamer simply failing to start.

Having multiple instances of a GStreamer pipeline running is also undesirable; if something is streamed twice or more times, the receiver will record cluttered audio files, recording from both sources in parallel. Usually, this will not be interfering audio, but interleaving samples instead. Because of this, you may want to have only one stream at a time for each source and destination, never more.

Listing 4.1: 00clean.sh — Cleaning the environment

```
1  #!/bin/bash
2
3  echo "* Terminating all gstreamer pipelines... "
4  killall gst-launch-0.10
5  echo "* Terminating all per-user pulseaudio services... "
6  killall pulseaudio
7  echo "* Stopping the system-wide DBUS daemon... "
8  stop dbus
9  echo "* Cleaning up DBUS process information... "
10 rm /var/run/dbus/*
```

4.1.2 Starting D-BUS

D-BUS is a messaging system used for Inter-Process Communication (IPC). It is necessary for PulseAudio (PA) to run. This scripts starts the system-wide D-BUS daemon.

Please note that the `start` and `stop` commands used here are part of the *upstart* system of Ubuntu. They may not be available on other Linux distributions. For compatible System V commands, use `/etc/init.d/dbus start` and `/etc/init.d/dbus stop`.

Listing 4.2: 01dbus.sh — Starting D-BUS

```
1  #!/bin/bash
```

[1] Bash is the Bourne Again Shell, a command line interpreter for UNIX-like systems. It is the most commonly used shell for Linux and supports several programmatic expressions.

```
2
3 start dbus
```

4.1.3 Starting PulseAudio

As PulseAudio is difficult to run in a headless environment (means, an environment without a graphical user interface), we have to make sure it is set up correctly. PulseAudio needs Desktop Bus (D-Bus) to be run as a so-called session bus, i.e. one D-Bus process per system user. This can be ensured by sticking to the following startup procedure:

1. Start a D-Bus session using `dbus-launch`
2. Store the relevant session address and process ID in environment variables
3. Export them so they can be used in a sub-shell, if necessary
4. Start PulseAudio as a daemon

Listing 4.3: 02pa.sh — Starting PulseAudio

```
1  #!/bin/bash
2
3  dbusinfo=( $(dbus-launch) )
4  DBUS_SESSION_BUS_ADDRESS=${dbusinfo[0]#DBUS_SESSION_BUS_ADDRESS=}
5  DBUS_SESSION_BUS_PID=${dbusinfo[1]#DBUS_SESSION_BUS_PID=}
6
7  export DBUS_SESSION_BUS_ADDRESS
8  export DBUS_SESSION_BUS_PID
9
10 export -p | grep DBUS
11
12 pulseaudio --start
```

4.1.4 Streaming all Audio Output

To stream the synthesized speech to component #2 (telephony server), a GStreamer pipeline is set up that makes use of the previously started PulseAudio. PulseAudio provides a virtual so-called "monitor" device. This device redirects all audio output that is generated on the system (e.g. music or system sounds) to the sound card's input, where usually audio is recorded. Thus, we can use the monitor as a GStreamer source. The terms "source" and "sink" were already explained in chapter 3.

On our reference system, the monitor device was labeled

`alsa_output.platform-snd_dummy.0.analog-stereo.monitor`

One can identify the device on every system running PulseAudio using the command `pactl list | grep monitor`.

The audio signal taken from the monitor device is further sent into a queue, converted into the audio format proposed in Table 3.1, and finally sent to the server that hosts component 2 (VM #2 - telephony server) via UDP.

Listing 4.4: 03stream.sh — Streaming audio playback to VM #2

```
#!/bin/bash

gst-launch-0.10 pulsesrc device=alsa_output.platform-snd_dummy.0.
    analog-stereo.monitor ! queue ! audioconvert ! audio/x-raw-int,
    channels=1,depth=16,width=16,rate=44100 ! rtpL16pay  ! udpsink
    host=141.31.8.112 port=51000
```

4.1.5 Receiving and Synthesizing Text

So far, no audio has been produced, so the stream will only transfer silence. The fifth and last script for this component is not a Bash script, but a Python program. It will open a multi-threaded TCP server on a specified port. As soon as a client connects and a string is received, the program opens a new Festival process and passes the string to it, which then gets synthesized. The client connection is closed immediately[2].

This Python program only demonstrates the simplest solution possible. It does not care about timing, which means it cannot guarantee that more than one Festival process is synthesizing text at the same time (which would lead to overlapping speech). A mechanism to prevent this would have been out of scope for this study work.

The application prints out every string to be synthesized.

Listing 4.5: 04listen.py — TCP server and TTS

```
#! /usr/bin/env python
import SocketServer, subprocess, sys

HOST = ''
PORT = 50008

class SingleTCPHandler(SocketServer.BaseRequestHandler):
  def setup(self):
    print '+ ', self.client_address, 'connected'
    self.request.send('hi ' + self.client_address[0] + '\n')
```

[2]Having a persistent connection is not necessary here.

```
11
12    def handle(self):
13      # self.request is the client connection
14      data = self.request.recv(1024)  # clip input at 1Kb
15      print '>', 'Synthesizing', data.strip()
16      p = subprocess.Popen("echo \"%s\" | festival --tts" % data.
            strip(),stdout=subprocess.PIPE,stderr=subprocess.PIPE,shell
            =True)
17      self.request.close()
18
19    def finish(self):
20      print '- ', self.client_address, 'disconnected'
21
22  class SimpleServer(SocketServer.ThreadingMixIn, SocketServer.
        TCPServer):
23    # Ctrl-C will cleanly kill all spawned threads
24    daemon_threads = True
25    # much faster rebinding
26    allow_reuse_address = True
27
28    def __init__(self, server_address, RequestHandlerClass):
29      SocketServer.TCPServer.__init__(self, server_address,
            RequestHandlerClass)
30
31  if __name__ == "__main__":
32    server = SimpleServer((HOST, PORT), SingleTCPHandler)
33    # terminate with Ctrl-C
34    try:
35      server.serve_forever()
36    except KeyboardInterrupt:
37      sys.exit(0)
```

4.1.6 Alternative Solution without Monitor Device

Immediately before finishing the work on this report, we found out that there is a GStreamer plugin for Festival. Unfortunately, the time was too short to test out this approach, but it may be a good point to evaluate for future students or researchers.

The GStreamer plugin for Festival can be found in the package `gstreamer0.10-plugins-bad`. This means, it is not as stable as it could be, maybe lacking a code review.

For the plugin to work, Festival needs to run in server mode.

Listing 4.6: Start Festival in server mode

```
1  $ festival --server
```

The Festival process listens for incoming connections, usually on port 1314. Using a telnet session, one can send Scheme commands to the server, just as if he would access the command-line interpreter of Festival.

Listing 4.7: Sample telnet session on Festival

```
$ telnet localhost 1314
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
(SayText "Hello")
LP
#<Utterance 0xb690c9d8>
ft_StUfF_keyOK
Connection closed by foreign host.
```

This works just like our own Python-based TCP server, but of course allows for less flexibility. Using a Python program, one can add handling of different channels, error handling and much more. Nevertheless, for a simple test setup the server mode of Festival would be sufficient.

Using the GStreamer plugin, one can also directly pipe text into the Festival process. From the plugin's source code comes an exemplary pipeline:

Listing 4.8: Exemplary GStreamer–Festival pipeline

```
$ echo 'Hello G-Streamer!' | gst-launch fdsrc fd=0 ! festival !
    wavparse ! audioconvert ! alsasink
```

As you can see, you can directly process synthesized speech using GStreamer. For example, it could directly be streamed to the telephony server. Nevertheless, there are a few questions still to be answered:

- Can I open a pipeline listening on TCP for text? Does this pipeline stay open, or does it terminate after receiving the first connection?
- Can I use the festival element as a source only and directly use telnet (or any other TCP client) to control it?
- Do I need to open a new pipeline everytime I want to synthesize text?

4.2 VM #2 – Telephony

As the telephony service could not be set up and configured during the course of this study thesis - we will get into this later, we settled for a short cut on this component: instead of allowing a user to give input by speaking, we would use

a recorded audio file of speech, and instead sending a user the system's answer, we would just record the answer to an audio file.

Thus we need two pipelines to set up, one for recording the incoming audio, and one for streaming the speech of a virtual caller to the ASR component. Nevertheless we invested an extensible amount of time trying to figure out how to set up Asterisk for use with JACK or ALSA. In this section we want to give an introduction to how Asterisk works, how it is configured and which problems we faced with it.

Having briefly introduced Asterisk in section 3.6, this section is dedicated to a more in-depth introduction to the software.

Asterisk is written in C and C++, it is a deamon process which allows the handling of several threads (e.g. phone calls). This is really important because, telephony servers usually handle a lot more than just one call. The main strength of Asterisk is its broad protocol support which enables the owners of the Asterisk PBX to connect analogous as well as digital telephones to the server. Asterisk supports among others the following telephony protocols: Session Initiation Protocol (SIP), Inter-Asterisk eXchange (IAX2) - used by Asterisk internally, H.323.

Therefore through Asterisk it does not matter what kind of telephone (hardphone, softphone [3], landline, cellphone) you call or have. Asterisk can make it all happen. In order to do that Asterisk translates all protocols into the same protocol internally used Asterisk protocol. This protocol is called IAX2 and also can be used to connect separate Asterisk systems with each other. Having several PBX systems can be helpful for workload-balancing, latency, or other functional purposes.

4.2.1 Installation

Asterisk comes with the standard repositories of many linux distributions like Ubuntu [2] and therefore can be installed using the standard package manager of your system.

Listing 4.9: Installing Asterisk on Ubuntu using apt-get

```
$ sudo apt-get install asterisk
```

[3]Softphones are telephones which do not physically exist. For instance a Skype *phone* would be considered a softphone since itis only a piece of software running on your computer, as opposed to your landline which is an actual piece of plastic and circuits.

However if you want to use Asterisk to forward incoming audio to Jack, you need to make sure that you install at least Asterisk 1.8.X. Older versions like 1.6.2.X do not support Jack audio hooks [12]. If you are unsure which version of asterisk you are running, please see section 4.2.3 Command Line Interface (CLI). After installation, asterisk needs to be started. This is done using the asterisk executable. The installation path depends on your linux distribution.

Listing 4.10: Starting/Stopping Asterisk on Debian-based Systems

```
$ sudo /etc/init.d/asterisk start
  Starting Asterisk PBX: asterisk.
$ sudo /etc/init.d/asterisk restart
  Stopping Asterisk PBX: asterisk.
  Starting Asterisk PBX: asterisk.
$ sudo /etc/init.d/asterisk stop
  Stopping Asterisk PBX: asterisk.
```

4.2.2 Resources

Unfortunately documentation, tutorials and user contributions for Asterisk are a little sparse for unusual features (like the Jack Application) but yet there is a couple of helpful sites, which we want to list here:

- Official Asterisk Website [8]
  `http://www.asterisk.org/`
  Especially helpful are the introductory video, the forums and the documentation

- Asterisk Guru Website [13]
  `http://asteriskguru.com/`
  Provides tutorials and guides for standard tasks like installation, troubleshooting, configuration and Private Branch Exchange (PBX) scenarios. This proved to be the most extensive and detailed resource for information on Asterisk on the web.

- VoIP Info Website [14]
  `http://www.voip-info.org/`
  Not only contains a couple of helpful articles about Asterisk but also a lot articles which help understand internet telephony technologies like VoIP and SIP.

- Asterisk Book [15]
  `http://www.asteriskdocs.org/`
  A book on Asterisk!

4.2.3 Command Line Interface (CLI)

Asterisk has a Command Line Interface (CLI) which allows us to directly interact with it. After having started the Asterisk server (4.2.1) we can start the CLI by running the following command in the linux shell:

Listing 4.11: Starting the Asterisk Command Line Interface (CLI)

```
$ sudo asterisk -r
```

The CLI offers a wide range of commands and queries, most of which are explained in reference [16]. In this section we want to introduce those CLI commands which proved to be the most helpful or the most frequently used in this project.

Listing 4.12: Reload Asterisk configuration

```
*CLI> reload
```

Listing 4.13: Show Asterisk version

```
*CLI> core show version
```

Listing 4.14: Stop Asterisk

```
*CLI> core stop ... # You can use TAB for auto-completion or
    suggestions
```

Listing 4.15: Show all available applications

```
*CLI> core show applications
```

Listing 4.16: Show application documentation

```
*CLI> core show application ... (e.g. Jack)
```

Listing 4.17: Turn on/off SIP debugging

```
*CLI> sip set debug on/off
```

Generally speaking it is wise to have the CLI running while you are testing your system. Since errors and warnings are printed in the CLI in real-time and can be helpful tracking down your bugs.

4.2.4 Session Initiation Protocol (SIP)

In order to be able to call the SDS with a SIP-phone, Asterisk needs to be set up properly. Of course it is also possible to connect hardphones to Asterisk but for demonstration purposes we decided to only set up SIP-telephony and target our effort at other areas.

SIP is used for creating, terminating or modifying media sessions (for instance voice or video) between two or more parties. In Asterisk's case, SIP is used to create two way RTP session. The RTP session is then used to transfer the audio from the Asterisk server to the caller and vice versa. Both protocols are core technologies of VoIP services and will be explained here:

Session Initiation Protocol (SIP) - is an application-layer protocol which is used for signaling media sessions (remember: creating, terminating or modifying). It usually runs over port 5060.
The simplest use-case contains two user-agents (e.g. softphones) and a registrar (e.g. Asterisk). In reality there are usually a couple of registrars involved that need to exchange the users they know before a call can be properly redirected and established. Each SIP-registrar keeps track of all available participants by using a name:ip pair. Every participant who would like to make itself available to others first has to register with the registrar. In a second step one of the participants invites another participant to open a session between them. In our case there will be an RTP Media Session. Figure 4.1 shows the registration sequence with the two exemplary users *User1* and *User2*. Figure 4.2 shows a sequence diagram of a basic SIP dialog. We will explain the individual steps of both in the following paragraphs.

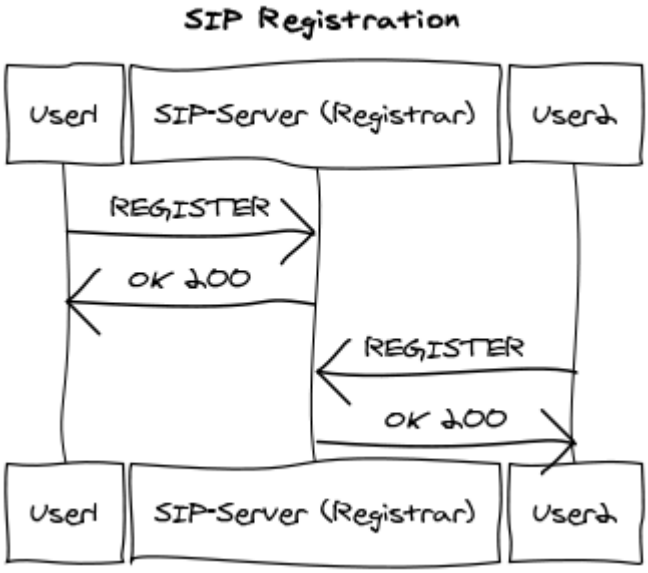

Figure 4.1: SIP Registration

The registration process is a very simple one. Every client who wants to register with the registrar (SIP-Server) sends a *REGISTER* message. The SIP protocol allows to specify a couple of extra headers for this message which can be used to provide additional information like the user-agent which was used to formulate the REGISTER request. After the registrar receives the *REGISTER* message it confirms the registration with an *OK 200* message, if no conflicts arise. Here is an example:

Listing 4.18: User sends REGISTER request to registrar

```
REGISTER sip:141.31.8.112 SIP/2.0
Via: SIP/2.0/UDP 192.168.178.36:5059;rport;branch=z9hG4bKcvkxlfvj
Max-Forwards: 70
To: "billy" <sip:billy@141.31.8.112>
From: "billy" <sip:billy@141.31.8.112>;tag=ceuyw
Call-ID: cspjhwxzlagqqrr@bfg-laptop
CSeq: 405 REGISTER
Contact: <sip:billy@192.168.178.36:5059>;expires=3600
Allow: INVITE,ACK,BYE,CANCEL,OPTIONS,PRACK,REFER,NOTIFY,SUBSCRIBE,
    INFO,MESSAGE
User-Agent: Twinkle/1.4.2
Content-Length: 0
```

Listing 4.19: User receive OK from registrar

```
SIP/2.0 200 OK
```

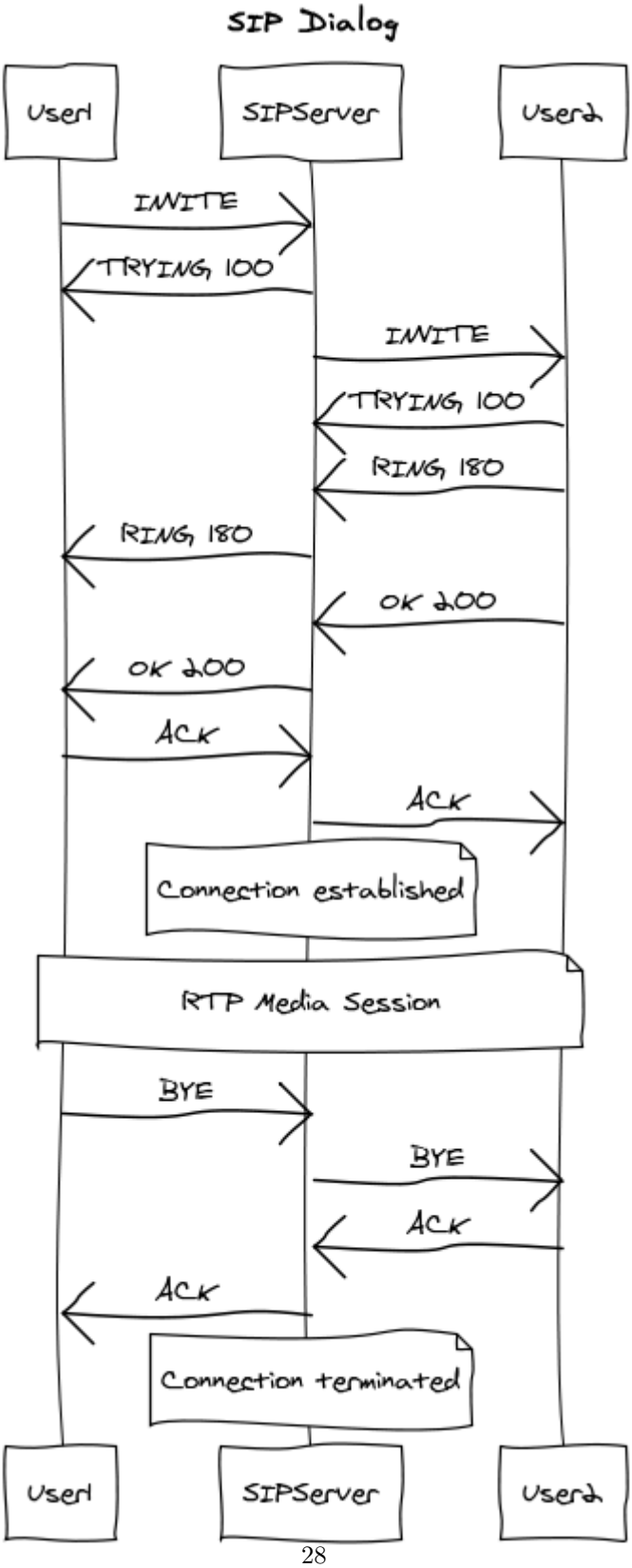

Figure 4.2: A basic SIP dialog

```
Via: SIP/2.0/UDP 192.168.178.36:5059;branch=z9hG4bKcvkxlfvj;
    received=88.72.49.2;rport=5059
From: "billy" <sip:billy@141.31.8.112>;tag=ceuyw
To: "billy" <sip:billy@141.31.8.112>;tag=as2f738d86
Call-ID: cspjhwxzlagqqrr@bfg-laptop
CSeq: 405 REGISTER
Server: Asterisk PBX 1.8.4.4~dfsg-2ubuntu1
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, SUBSCRIBE, NOTIFY,
     INFO, PUBLISH
Supported: replaces, timer
Expires: 3600
Contact: <sip:billy@192.168.178.36:5059>;expires=3600
Date: Tue, 05 Jun 2012 13:39:42 GMT
Content-Length: 0
```

After having registered with the registrar, participants can call other registered participants by sending an INVITE to the registrar with the other participants name. Let us assume we (User1) want to call somebody else (User2) who is registered with the registrar found at 141.31.8.112.

Listing 4.20: Sending INVITE to registrar

```
INVITE sip:User2@141.31.8.112 SIP/2.0 #name@registrarIP
...
Contact: <sip:User1@192.168.178.36:5060> # our own ip
```

Usually additional headers are required to identify who is calling who, but since we are only trying to understand what SIP does, not how it actually works in every detail, we dispense with a lengthy description of the possible headers and stay on a functional level.
The registrar immediately responds with *TRYING* to let User1 (us) know that our *INVITE* was received.

Listing 4.21: Registrar responds with TRYING

```
SIP/2.0 100 Trying
```

The registrar forwards our invite to User2:

Listing 4.22: Registrar sends an INVITE to User2

```
INVITE sip:User2@141.31.8.112 SIP/2.0 #name@registrarIP
...
Contact: <sip:User1@192.168.178.36:5060> # our own ip
```

User2, before doing anything else, confirms that the *INVITE* was received:

Listing 4.23: User2 responds with TRYING

```
SIP/2.0 100 Trying
```

After the request was processed internally the user agent of User2 starts ringing and notifies the registrar and thus User1 about it. As soon as User2 picks up, their user agent sends an *OK 200* which needs to get acknowledged. Then the media session can start:

The Real-time Transport Protocol (RTP) protocol is used to transfer audio/video streams over ip-based networks. It is used in conjunction with RTP Control Protocol (RTCP) protocol which implements and enforces quality of service for the streams. In most cases the registrar/SIP-Server is not involved in the RTP session, most servers can be configured to proxy the RTP session (so can Asterisk), but it is very unusual. That means that the streams are transferred directly between the users, no other parties involved. The SIP-Server is only needed to set up and tear down the connection.

The session is terminated as soon as one of the users sends a *BYE* message using SIP. This message gets acknowledged and the session is terminated (see bottom figure 4.2).

4.2.5 Setting up Asterisk to work with SIP

In order to make calls with Asterisk using SIP there are two Asterisk configuration files which we need to inspect.

First there is the **sip.conf** where general settings regarding the SIP protocol and the devices (phones) can be specified. A proper documentation of how to configure the sip.conf to suit your needs can be found at the voip-info.org website [17] or in chapter 5 of the Asterisk book [15].

Listing 4.24: Path to sip.conf under debian-based systems

```
/etc/asterisk/sip.conf
```

Listing 4.25: Example of the sip.conf

```
[general]
bindport=5060

[user1]
type=friend

[user2]
```

```
type=friend
```

Secondly there is the **extensions.conf**. Within this file the behavior of all calls going through Asterisk is controlled. This is done using the Asterisk script language Dialplan. This is were the magic happens: you can set up an IVR system for your callers, redirect and connect calls and heaps of other scenarios. The callers IDs are called *extensions* in this context. These extensions can consists of digits and characters. Dialplan script is very powerful and we recommend reading the introductory guide at asteriskguru.com [18].

Listing 4.26: Path to extensions.conf under debian-based systems

```
/etc/asterisk/extensions.conf
```

The syntax of Dialplan script is pretty straight forward and always follows this pattern:

```
exten => <extension>,<priority>,<application(arg1, arg2, ...)>
#fill the placeholders <...> according to the needs of your
    application
```

A small example which allows users to call user1 and user2:

```
exten => user1,1,Dial(SIP/user1)
exten => user2,1,Dial(SIP/user2)
```

4.2.6 Connecting Asterisk to gstreamer

This is necessary to send the audio stream which comes from the caller to the ASR component. On the other side of the call cycle (Figure 2.1), it is necessary to input the synthesized answer of the voice browser back into Asterisk so it can be streamed to the caller. Simply put: a two way communication between Asterisk and gstreamer is required. During the course of this project we evaluated and tested several approaches to connect Asterisk to gstreamer, unfortunately we still have not found a solution that delivers the desired result. In this section we will introduce these different approaches and describe the problems we ran into by doing so.
Figure 3.1 shows that gstreamer can interact with Pulseaudio, ALSA or JACK.

Connect using Jack In 2008 Russel Bryant, an open source software engineer, developed a Dialplan application which enable Asterisk to connect to Jack [19]. The documentation for this application can be found by typing:

```
CLI*> core show application JACK
CLI*> core show application JACK_HOOK
```

in the Asterisk CLI or online in the asterisk documentation [20]. Since this is a Dialplan application it can be easily invoked by inserting the appropriate script in the *extensions.conf* of Asterisk.

```
exten => user2,1,Jack(o(jack_server), i(playback_1), o(capture_1),
    c(asterisk_channel))
```

Unfortunately the documentation is very sparse and no tutorials or guides or even threads in forums exist, so when we ran into troubles we contacted the author of the application for help, but we have not received any response.

Calling user2 on our system resulted in Asterisk to stop upon calling JACK, without any errors or warnings. Using the JACK shell tools (on linux insert jack into shell and press tab to view full list) we found out that the application stops before it connects to the specified JACK server. But without any errors or warnings on either side of the connection there was no getting further. Possible entry points for future developers could be:

- playing around with different combinations of versions for Asterisk and JACK
- debugging the application - would require expert knowledge of Asterisk, JACK, audio programming and probably also a lot of time
- possibly asterisk mailing lists

Connect using ALSA Asterisk can directly send a caller's audio stream to ALSA. However ALSA does not support audio routing which would be needed to somehow capture the audio with gstreamer. ALSA could be used to directly play the caller's audio on a speaker connected to the server running Asterisk. This is useful for SIP telephony on mobile devices but not for the voice browsing use case.
Asterisk can be configured for the use with ALSA in the *alsa.conf* [21].

Listing 4.27: Path to alsa.conf under debian-based systems

```
/etc/asterisk/alsa.conf
```

Listing 4.28: Calling to ALSA from Dialplan

```
exten => user2,1,Dial(Console/alsa)
```

Listing 4.29: Documentatino

```
CLI*> core show application Dial
```

Connect using PulseAudio Unlike for JACK there is no Dialplan application to connect calls with PulseAudio. Naturally the next thing we thought of was sending the caller's audio stream to ALSA, retrieve it with PulseAudio and then connect to gstreamer. Due to the nature of ALSA this is not possible either.

Using another PBX server When we first started working on this research project, we quickly decided to go with Asterisk as a telephony server, since it is the most popular and flexible open source PBX project for standard PBX tasks. However the application we are trying to build requires access to low level audio processing functionality (namely JACK or Pulseaudio) and integration with Asterisk is not very stable yet, therefore it might be useful to look for other open source PBX solutions with better access to JACK.

4.2.7 Dummy Setup

As described in the sections before, we did not manage to get Asterisk connected to gstreamer in the available amount of time. To proof that the rest of our setup works as expected, we faked the telephony component.

The incoming speech that in a real environment comes from a caller is faked by playing an audio file. The outgoing speech is not being sent to the caller, but recorded to an audio file instead. These two tasks are done using scripts that call gstreamer pipelines.

Basically, the setup here is a lot leaner as in section 4.1. There is no need for PA, as no monitor device is being used. Thus, we do not have to initialize D-Bus and do not have to clean anything beforehand. The streaming pipeline is slightly edited and the according script is renamed to `02stream.sh`. Also, an additional script `01record.sh` is employed.

Execute the following steps using the scripts presented in the next two subsections:

1. Start recording incoming audio using `01record.sh recording.mp3` as user
2. Start streaming using `02stream.sh sourcefile.mp3 441000` as user (change the second parameter according to your MP3's clock rate)

It may be a good idea to invoke these two scripts in different SSH sessions to have easy control over both.

Recording

Listing 4.30: 01record.sh — Record audio coming from the TTS

```
#!/bin/bash

gst-launch-0.10 -vvv udpsrc port=51000 ! "application/x-rtp, media=
    (string)audio, clock-rate=44100, width=16, height=16, encoding-
    name=(string)L16,encoding-params=(string)1, channels=(int)1,
    channel-position=(int)1, payload=(int)96" ! gstrtpjitterbuffer
    do-lost=true ! rtpL16depay ! audioconvert ! lame ! filesink
    location=$1
```

This script takes the name of the audio file to record to as a parameter.

Streaming

The pipeline for streaming differs slightly from the one used for streaming from VM #1 to VM #2: a file source is used instead of the monitor device, and an decoding step is taken in between.

Listing 4.31: 02stream.sh — Streaming an MP3 file to the ASR

```
#!/bin/bash

gst-launch-0.10 filesrc location=$1 ! mad ! audioconvert ! audio/x-
    raw-int,channels=1,depth=16,width=16,rate=$2 ! rtpL16pay !
    udpsink host=141.31.8.113 port=51000 &
```

This script takes the name of the audio file being streamed as the first parameter. The second parameter is the MP3 file's clock rate in Hz. It may be, for example, 44100 or 8000. Please note that the IP mentioned by the *udpsink* points to VM #3.

4.3 VM #3 – Automated Speech Recognition & VM #4 – Voice Browser

The topics Automated Speech Recognition (ASR) and Voice Browsing are covered in Ramin Safarpour's concurrent study work and will not be discussed here. For detailed information please refer to Ramin Safarpour's paper.

Bibliography

[1] The Linux Kernel Archives
http://kernel.org/

[2] Ubuntu Server 12.04 LTS
http://www.ubuntu.com/download/server

[3] Advanced Linux Sound Architecture (ALSA) project homepage
http://www.alsa-project.org/

[4] GStreamer: open source multimedia framework
http://gstreamer.freedesktop.org/

[5] RFC 3550 — RTP: A Transport Protocol for Real-Time Applications
http://tools.ietf.org/html/rfc3550

[6] The Philosophy of PCM
http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?reload=
true&arnumber=1697556

[7] rtp — Real-time protocol plugins
http://gstreamer.freedesktop.org/data/doc/
gstreamer/head/gst-plugins-good-plugins/html/
gst-plugins-good-plugins-plugin-rtp.html

[8] Asterisk — Private Branch Exchange
http://www.asterisk.org/

[9] Sphinx — Speech Recognition Toolkit
http://cmusphinx.sourceforge.net/

[10] The Festival Speech Synthesis System
http://www.cstr.ed.ac.uk/projects/festival/

[11] Wikipedia — GStreamer Pipeline Diagram
http://en.wikipedia.org/wiki/File:GStreamer_Technical_Overview.svg

[12] Asterisk — Versions https://wiki.asterisk.org/wiki/display/AST/Asterisk+Versions

[13] Asterisk Guru Website
http://asteriskguru.com/

[14] VoIP Info Website
http://www.voip-info.org/

[15] Asterisk — Book http://www.asteriskdocs.org/

[16] voi-info.org — Asterisk Command Line Interface (CLI)
http://www.voip-info.org/wiki/view/Asterisk+CLI

[17] voip-info.org — Asterisk sip.conf http://www.voip-info.org/wiki/view/Asterisk+config+sip.conf

[18] asteriskguru.com — Asterisk extensions.conf http://asteriskguru.com/tutorials/extensions_conf.html

[19] Russel Bryant — JACK Interfaces for Asterisk http://russellbryantnet.wordpress.com/2008/01/13/jack-interfaces-for-asterisk/

[20] Asterisk Documentation — JACK Application http://www.asterisk.org/docs/asterisk/trunk/applications/jack

[21] voip-info.org — alsa.conf http://www.voip-info.org/wiki/view/Asterisk+config+alsa.conf