Technische Hochschule Wildau
Technical University of Applied Sciences

Masterarbeit

zur Erlangung des akademischen Grades
Master

Technische Hochschule Wildau

Fachbereich Ingenieur- und Naturwissenschaften

Studiengang Telematik (M. Eng.)

Thema (deutsch): Entwicklung, Evaluation und Integration von Verfahren zur optimierten Nutzung von automatischer Spracherkennung in Robotikanwendungen

Thema (englisch): Development, evaluation and integration of methods to optimize the usage of automatic speech recognition for robotics applications

Autor/in: Tobias Lothar Erhardt Kannenberg

Seminargruppe: TM/17

Betreuer/in: Prof. Dr. rer. nat. Janett Mohnke

Zweitgutachter/in: Prof. Dr.-Ing. Anselm Fabig

Eingereicht am: 12.02.2020

Verlängerung des Bearbeitungszeitraumes

Auf Grund der Antragstellung des Studierenden wurde der Bearbeitungszeitraum für die Erstellung der Arbeit um 4 Wochen verlängert.

Das Abgabedatum nach Antragstellung ist der 12.02.2020.

Technische Hochschule Wildau
Fachbereich Ingenieur- und Naturwissenschaften

Masterarbeit

Antrag vom: 20.07.2019

Name:	Tobias Lothar Erhardt Kannenberg	Matrikel-Nr.:	147410919
Studiengang:	Telematik	Seminargruppe:	TM/17
		Beginn der Arbeit:	15.08.2019
Betreuende/r Hochschuldozent/in:	Prof. Dr. rer. nat. Janett Mohnke	Abgabetermin:	15.01.2020
Zweitgutachter/in:	Prof. Dr.-Ing. Anselm Fabig		
		Fachliche Betreuungsperson des Themenstellers:	Janett Mohnke
Themensteller (z.B. Betrieb, Institution, Behörde):	Technische Hochschule Wildau	Straße:	Hochschulring 1
		PLZ, Ort:	15745 Wildau

Kurzthema

Entwicklung, Evaluation und Integration von Verfahren zur optimierten Nutzung von automatischer Spracherkennung in Robotikanwendungen

Kurzthema in Englisch

Development, evaluation and integration of methods to optimize the usage of automatic speech recognition for robotics applications

Zielstellung

Es soll eine modulare Softwareplattform zur Verarbeitung von Audiodaten und Anbindung an Spracherkennungsdiensten für Robotikanwendungen geschaffen werden, mit deren Hilfe durch gezielte Vorverarbeitung die Erkennungsrate für Realbedingungen zu optimieren ist.

Inhaltliche Anforderungen / Teilaufgaben

- Ist-Analyse und Problemanalyse der Integrationen von Spracherkennungssoftware bestehender Robotikanwendungen
- Anforderungsanalyse, Entwurf und Implementierung der Softwareplattform
- Aufbau einer Testumgebung zur objektiven und realitätsnahen Evaluierung der Signalverarbeitung
- Implementierung von relevanten Signalverarbeitungsverfahren
- Evaluation und Anpassung der Signalverarbeitung für eine optimale Erkennungsrate von ausgewählter Spracherkennungssoftware unter Verwendung von ausgewählter Hardware

Sprache der Arbeit

Deutsch

Konsultationen erfolgen nach Vereinbarung mit dem betreuenden Hochschullehrer.

Prof. Dr. rer. nat. Janett Mohnke	Prof. Dr.-Ing. Anselm Fabig	Tobias Lothar Erhardt Kannenberg	*genehmigt*
(Hochschuldozent/in)	(Zweitgutachter/in)	(Student/in)	(Prüfungsausschuss)

Eingang der Abschlussarbeit _____ Sperrvermerk ☐ Ja ☒ Nein

Bibliographische Beschreibung und Referat

Entwicklung, Evaluation und Integration von Verfahren zur optimierten Nutzung von automatischer Spracherkennung in Robotikanwendungen

Masterarbeit, 2020, 101 Seiten, 58 Abbildungen, 9 Tabellen, 0 Anlagen, 1 Beilage

Technische Hochschule Wildau, Fachbereich Ingenieur- und Naturwissenschaften

Ziel Es soll eine modulare Softwareplattform zur Verarbeitung von Audiodaten und Anbindung an Spracherkennungsdiensten für Robotikanwendungen geschaffen werden, mit deren Hilfe durch gezielte Vorverarbeitung die Erkennungsrate für Realbedingungen zu optimieren ist.

Zusammenfassung

Die vorliegende Masterarbeit gibt einen Einblick in die Konzeption und Entwicklung einer modularen Experimentierplattform für Audioverarbeitung und automatischer Spracherkennung. Darüber hinaus wurde ein Testaufbau- und Ablauf erarbeitet mit dessen Aufnahmen objektive Bewertungen durch die Ergebnisse der Spracherkennungssoftware bezüglich der Aufnahmegüte vollzogen werden konnten. Dies wurde mit verschiedenen Audioeingabegeräten und Spracherkennungsprogrammen durchgeführt. Auf dieser Daten- und Bewertungsgrundlage kam es zur Implementierung von Signalverarbeitungsverfahren zur Aufbereitung der Audiodaten, mit dem Ziel die Erkennungsrate der automatischen Spracherkennung dadurch zu steigern.

Abstract

This master's thesis covers the process of design and development of a modular software framework for experimenting with audio processing and automatic speech recognition. Furthermore, a test setup and procedure was created to record speech with different devices and evaluate objectively the recordings by utilizing the outcome of various speech recognition software. Based on this data and evaluation basis, signal processing methods were implemented to increase the recognition rate using a preprocessing of the recording signal.

Schlagwörter Signalverarbeitung, Automatische Spracherkennung, Robot Operating System, Audioverarbeitung, Mikrofon-Array

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Abschlussarbeit eigenständig angefertigt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum, Unterschrift (Tobias Kannenberg)

Hinweise zum Lesen dieser Arbeit

Alle im Fließtext *kursiv* geschriebenen Wörter sind Eigennamen. Begriffe und Abkürzungen in *kursiver und serifenloser* Schrift sind im Glossar bzw. im Abkürzungsverzeichnis aufgeführt. Beinhaltet ein Text Wörter in `Festbreitenschrift`, handelt es sich um Ausschnitte aus einem Programmcode oder Symbolnamen. Diese können aufgrund der Syntaxhervorhebung in verschiedenen Farben auftreten. Wenn Zeilen mehrzeiliger Codeabschnitte zu breit sind, um sie vollständig darzustellen, werden sie mit einem vorangestellten ► in die nächste Zeile umgebrochen. Referenzen auf Quellen stehen in eckigen Klammern [ ]. Abbildungen, deren Beschriftungen keine Quellenangaben aufweisen, wurden vom Autor erstellt.

Inhaltsverzeichnis

1 Einleitung

Das *RoboticLab*[1] der Telematik an der *TH Wildau* setzt unter anderem Roboter vom Typ *NAO* und *Pepper* für Lehre, Forschung und reale Anwendungen ein. Im Mittelpunkt steht dabei die Interaktion mit den Menschen. Die Mensch-Maschine-Kommunikation kann dabei auf mehrere Arten erfolgen, wobei aber die verbale Sprache die natürlichste Kommunikationsart für den Nutzer ist. Vom Roboter zum Menschen kann mit Hilfe von Sprachsynthese kommuniziert werden. Für den entgegengesetzten Kanal wird eine automatische Spracherkennung (*Automatic-Speech-Recognition (ASR)*) benötigt, welche die Sprache in Textform umwandelt. Diese und weitere Verfahren sind schon vom Hersteller implementiert und für selbst entwickelte Robotikanwendungen über Schnittstellen nutzbar. Darauf aufbauend wird beim *RoboticLab* an einer linguistischen Analyse des gesprochen Textes gearbeitet, wodurch ein Roboter den Nutzer verstehen kann, indem die Bedeutung herausgefunden wird. Gepaart mit einem kontextbezogenen Wissensschatz, welcher dafür sorgt, dass sich der Roboter in seinem Einsatzgebiet wie z. B. die Hochschulbibliothek auskennt, ist er so nun in der Lage, gut mit Menschen zu kommunizieren und auf frei formulierte Fragen die passenden Antworten zu liefern. Jedoch lässt sich diese Funktionalität nur eingeschränkt anwenden, da die Spracherkennung nicht zufriedenstellende Ergebnisse liefert. Grund dafür ist ein von Störgeräuschen stark dominiertes Eingangssignal der Mikrofone, welche den Lüftern des Roboters entspringen. Des Weiteren sind die Programmierschnittstellen der vom Hersteller (*SoftBank Robotics*[2]) bereitgestellten Software zu unflexibel, wodurch unter anderem das Verwenden von digitalen Audiofilterverfahren und das Nutzen von alternativen Spracherkennungsprogrammen erschwert wird.

Ziel dieser Arbeit ist es, sich dieser Probleme anzunehmen und eine Softwarelösung zu entwickeln, welche das Arbeiten mit Audiosignalen und Spracherkennungssoftware erleichtert und darüber hinaus zu einer reduzierten Wortfehlerrate durch Vorverarbeitung des Audiosignals führt.

1.1 Zielbeschreibung

Diese Arbeit hat zwei aufeinander aufbauende Ziele. Zum einen soll eine modulare Softwareinfrastruktur geschaffen werden, welche die Nutzung von Spracherkennung und Handhabung von digitalen Audiosignalen im Kontext von Robotikanwendungen erleichtert. Auf dieser Grundlage sollen zum anderen Signalverarbeitungsverfahren implementiert, integriert, evaluiert und verbessert werden, damit Mängel bei der Aufnahmegüte kompensiert und die Möglichkeiten der vorhandenen Aufnahmetechnik (Mikrofon-Arrays) besser ausgenutzt werden können, sodass die Wortfehlerrate der anschließenden Spracherkennung signifikant verringert wird. Der *Pepper*-Roboter stellt die im Fokus stehende Hardware-

[1] siehe `https://icampus.th-wildau.de/icampus/home/de/roboticlab` (besucht am 23.01.2020)
[2] siehe `https://www.softbankrobotics.com/emea/en` (besucht am 23.01.2020)

und Softwareplattform dar, auf welcher die Audiodaten erhoben werden. Allerdings soll die Implementierung auf andere Plattformen übertragbar sein und somit weitestgehend unabhängig von den Eingabegeräten und der eingesetzten Spracherkennungssoftware funktionieren.

1.2 Abgrenzung

Die zu entwickelnde Softwarelösung ist kein alleinstehend nutzbares Programm für die Mensch-Maschine-Kommunikation. Es handelt sich um eine Umgebung, welche als Basis für die Implementierung von konkreten Anwendungen unter der Ausnutzung der neu gewonnenen Erkenntnisse dient und des Weiteren eine Experimentierplattform darstellt. Es findet keine Neuimplementierung oder Modifizierung der eigentlichen Spracherkennungssoftware statt. Auch der erkannte Text wird nur für weitere Analyseschritte zur Verfügung gestellt und nicht selbst weiter verarbeitet.

1.3 Aufbau der Arbeit

Zu Beginn leitet Kapitel 2 mit einer Vorbetrachtung ein, um den Kontext nahe zu bringen. Dort werden Grundlagen erklärt sowie eine genauere Analyse des Ist-Standes dargelegt. Nach weiteren Betrachtungen zu alternativer Aufnahmetechnik und Spracherkennungssoftware geht es in Kapitel 3 um das Aufstellen der Anforderungen an die im ersten Schritt zu entwickelnde Experimentierplattform, welche den Namen *Robot Sound and Speech System (RoSaSS)* trägt. Dessen Entwurf und Implementierung wird in den Kapiteln 4 und 5 beschrieben. In Vorbereitung auf die Umsetzung des zweiten Ziels wird in Kapitel 6 ein Testaufbau erarbeitet. Dies ist die Grundlage für das Evaluieren der Signalverarbeitungsverfahren, welche inklusive der Theorie und Implementierung in Kapitel 7 erläutert werden. Abschließend beinhaltet das Kapitel 8 ein Fazit sowie einen Ausblick für zukünftige Entwicklungen.

2 Vorbetrachtung

In diesem Kapitel werden einige Vorbetrachtungen behandelt, welche die nötige Basis für die Anforderungsanalyse bilden. Zunächst werden die Grundfunktionen von Sprachdialogsystemen erläutert und entsprechende Fachbegriffe eingeführt (siehe Abschnitt 2.1). Danach erfolgt in Abschnitt 2.2 eine Ist-Analyse am Beispiel des *Pepper*-Roboters und dessen *NAOqi API*, woraus unter anderem die Probleme hervorgehen, die später in den Anforderungen berücksichtigt werden müssen. Des Weiteren sind in den Abschnitten 2.4 und 2.3 grundlegende Betrachtungen zu den *ASR*-Engines und den Audioeingabegeräten geschildert.

2.1 Grundlagen eines Sprachdialogsystems

Damit Menschen mit Robotern als Sprachdialogsystem kommunizieren können, sind einige Schritte notwendig (siehe Abbildung 2.1 von (A) bis (I)). Da dies das übergeordnete Ziel ist, sollte der Gesamtüberblick mit beleuchtet werden, auch wenn sich diese Arbeit nur mit einem Teil davon genauer auseinandersetzt.

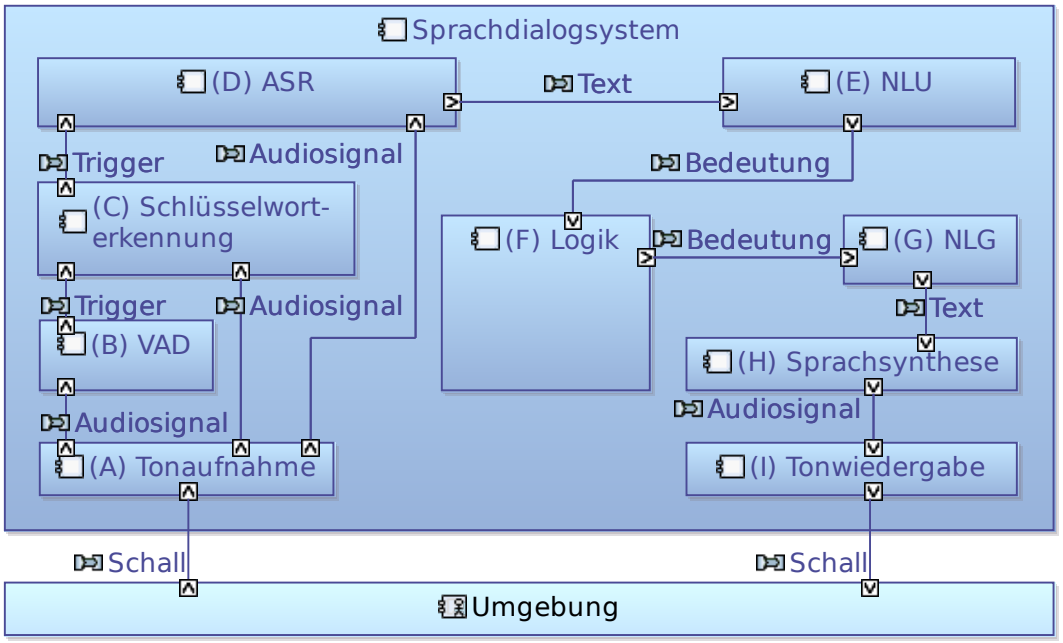

Abbildung 2.1: Bestandteile eines Sprachdialogsystems

Zunächst wird der Schall der Umgebung, welcher aus einer Mixtur aus der zu erkennenden Sprache und uninteressanten Hintergrundgeräuschen besteht, durch ein Mikrofon in

ein elektrisches und anschließend meist durch einen Analog-Digital-Umsetzer (Soundkarte) in ein digitales Audiosignal umgewandelt. In dieser Form der Tonaufnahme (A) kann es am besten weiter verarbeitet werden. Die *ASR*-Engine (D) versucht aus dem Audiosignal die Sprache in Textform zu extrahieren. Auf Grundlage dieser Transkription kann nun eine linguistische Auswertung (E) des Textes erfolgen (*Natural-Language-Understanding (NLU)*), um auf die Bedeutung zu schließen. Nun kann durch die implementierte Logik (F) entsprechend darauf reagiert werden, indem beispielsweise eine Aktion ausgelöst oder eine Datenbankabfrage zum Erhalten der gewünschten Information getätigt wird. Ist eine Rückmeldung zum Nutzer in Sprachform nötig, dann kann eine durch Schritt (G) entsprechend verfasste Antwort (*Natural-Language-Generation (NLG)*), unter Verwendung von Sprachsynthese (H) als Audiosignal ausgegeben (I) werden.

Je nach Ablauf der Kommunikation können weitere Zwischenschritte vor der Spracherkennung zum Einsatz kommen. Soll die Unterhaltung durch den Nutzer über Sprache initiiert werden, dann muss das System ständig das Audiosignal überwachen. In solchen Fällen ist es sinnvoll, die aufwendige Spracherkennung nur dann einzusetzen, wenn auch gesprochen wird. Dafür ist die sogenannte *Voice-Activity-Detection (VAD)* (B) zuständig, welche mit weniger rechenintensiven Methoden Aktivität im Audiosignal feststellt, die vom Nutzer als Geräuschquelle stammen könnte. Des Weiteren ist der Negativfall der *VAD* auch wichtig, weil damit das Ende einer Rede ermittelt werden kann, welches Einfluss auf den zeitlichen Ablauf des Dialogs hat.

Darüber hinaus ist eine Schlüsselworterkennung (C) zur Initiierung hilfreich. Somit werden erst weitergehende Analysen getätigt, wenn ein bestimmtes Schlüsselwort gefallen ist und somit davon ausgegangen werden kann, dass die Sprache an den Roboter adressiert ist. Dieses passive Zuhören ist auch als eine Form der Spracherkennung zu betrachten und kann durch die beim aktiven Zuhören verwendete *ASR*-Engine umgesetzt werden. Wird die Spracherkennung als Online-Dienst verwendet, hat eine getrennte Offline-Implementierung der Schlüsselworterkennung Vorteile, da einerseits der Netzwerkverkehr reduziert wird und andererseits nur zu definierten Zeitpunkten personenbezogene Audiodaten in die Hände des Betreibers fallen.

Aber auch das ist nicht in jedem Fall ein vollständiges Bild. Was Abbildung 2.1 nicht zeigt, sind z. B. weitere analytische Methoden, um mehr Informationen, die sich nicht immer in Text darstellen lassen, aus der Sprache zu gewinnen. Die Betonung bestimmter Wörter nimmt Einfluss auf die Bedeutung. Außerdem können aus der Art und Weise, wie gesprochen wird, Rückschlüsse auf die Emotionen des Sprechers gezogen werden. Auch bei der Sprachausgabe kann im Fall der Verwendung von humanoiden Robotern eine inhaltsbezogene Gestik und bei manchen Robotern sogar Mimik das Nutzererlebnis aufwerten.

2.2 Ist-Analyse

Der im Fokus stehende Roboter vom Typ *Pepper* wird in diesem Abschnitt genauer vorgestellt. Hier werden vor allem die Aspekte der Hardware und Software behandelt, welche direkt oder indirekt mit der Tonaufnahme, Verarbeitung, Tonwiedergabe und weiteren Schritten eines Sprachdialogsystems (siehe Abschnitt 2.1) zu tun haben. Dabei wird ein

besonderes Augenmerk darauf gelegt, die Gegebenheiten zu beschreiben, die in Verbindung mit der Umsetzung der groben Zielstellung in Unterabschnitt 1.1 stehen.

2.2.1 Hardware

Der *Pepper*-Roboter hat, wie es in Abbildung 2.2 zu sehen ist, sowohl die Mikrofone als auch die Lautsprecher am Kopf montiert. Ein Blick unter die Haube verrät die physische Mikrofonkonfiguration. Abbildung 2.3 zeigt, dass dazu mit Hilfe des dafür vorgesehen Schlüssels zunächst die Hinterkopfabdeckung abgenommen werden muss. Anschließend ist nach Lösen von zwei Schrauben nun auch die obere Abdeckung entfernbar.

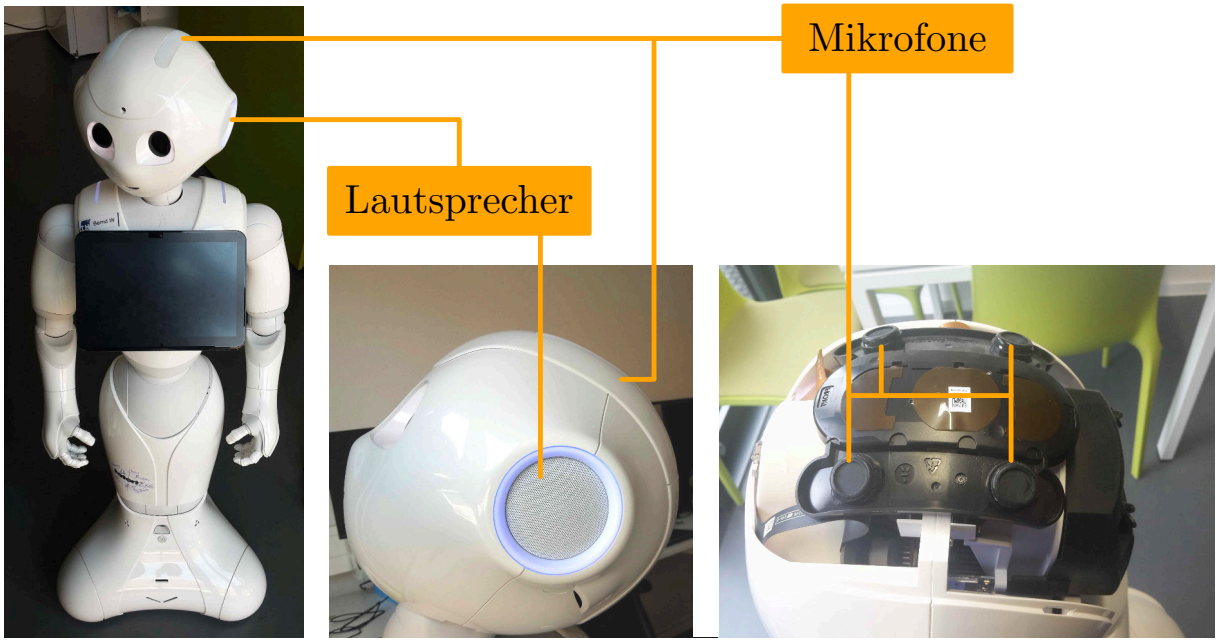

Abbildung 2.2: *Peppers* Mikrofone und Lautsprecher

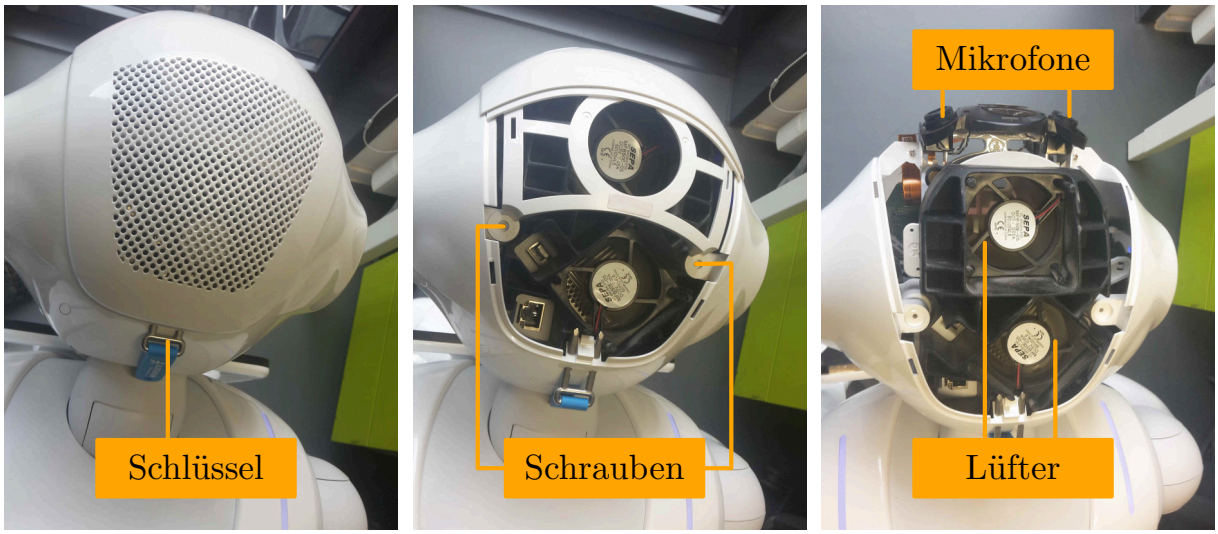

Abbildung 2.3: Schritte zum Freilegen der Mikrofone

Zum Vorschein kommt ein Mikrofon-Array mit vier einzelnen nach oben gerichteten und leicht nach außen geneigten Mikrofonen. Diese Position und Ausrichtung ist aufgrund der

Höhe des Roboters von 1,21 m ideal für die Kommunikation mit vor dem *Pepper* stehenden ausgewachsenen Menschen, da sie von schräg oben auf die Mikrofone herab sprechen. Im sitzenden Fall oder bei kleineren Menschen und Kindern sind Sprachquelle und Aufnahmegerät in etwa auf gleicher Höhe. Die gleichmäßige Anordnung und omnidirektionale Richtwirkung der Mikrofone sorgen für eine homogene Wahrnehmung der akustischen Umgebung in der horizontalen Ebene. Durch das steuerbare Halsgelenk ist eine Änderung der Kopfausrichtung und damit auch der der Mikrofone möglich, was die Audioaufnahme anpassbar an die jeweilige Situation macht.

Auch am Kopf befestigt sind die Lautsprecher, welche die Positionen der Ohren einnehmen und somit nach rechts und links gerichtet sind. Des Weiteren emittiert der Roboter im eingeschalteten Zustand ständig Schall durch den Betrieb seiner Lüfter. Zwei Ventilatoren befinden sich im Kopf und weitere besonders lautstarke im Torso (siehe Abbildung 2.4), welche für die Kühlung der Rechentechnik bzw. der Elektromotoren in den Gelenken zuständig sind. Zusätzlich sorgen auch die Gelenke und das Fahrwerk bei Bewegung für weitere unerwünschte Geräusche.

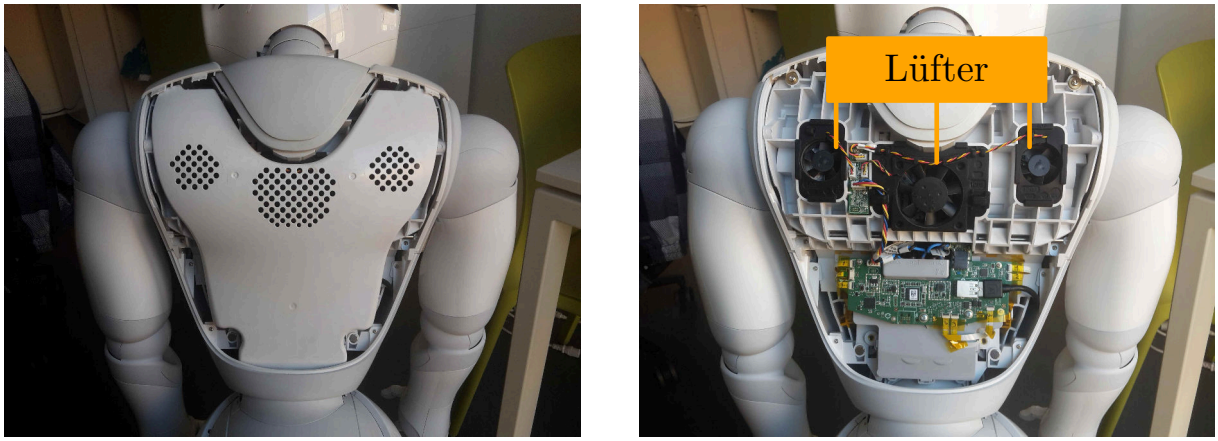

Abbildung 2.4: Lüfter im Torso nach Abnehmen von zwei Gehäuseschalen

2.2.2 Software

Auf dem Roboter vom Typ *Pepper* kommt als Betriebssystem ein *NAOqi OS* genanntes *Gentoo Linux*[3]-Derivat zum Einsatz, welches von *SoftBank Robotics* leider so eingerichtet und ausgeliefert wird, dass man als Entwickler keine Möglichkeiten hat, Systemanpassungen wie etwa das Aktualisieren von Softwarebibliotheken und weitere tiefgreifende Konfigurationen vorzunehmen. Die Steuerung des Roboters übernimmt ein Programm namens *NAOqi*. Diese proprietäre Software führt eine Vielzahl an vorinstallierten Modulen aus, welche auf die Robotik-Hardware zugreifen, Systemfunktionen verwalten und abstrahierte Grundfunktionen über Schnittstellen bereitstellen, die in ihrer Gesamtheit als *NAOqi API* bezeichnet werden. Die Kommunikation zwischen den einzelnen Modulen über die *NAOqi API* geschieht mittels Methodenaufrufe und Events, welche auch von eigenen Programmen genutzt werden können. Auch das Integrieren von selbst entwickelten Modulen zum Erweitern der Funktionalität ist möglich. Dabei wird die Kommunikation durch *NAOqi* so flexibel verwaltet, dass Programme und Module auf verschiedenen Rechnern ausgeführt

[3]weitere Informationen zu *Gentoo Linux* unter `https://gentoo.org/` (besucht am 23.01.2020)

werden können und über eine Netzwerkverbindung Informationen und Befehle mit dem Roboter austauschen.

NAOqi hat eine Reihe von Modulen integriert, mit deren Hilfe die Funktionalität eines Sprachdialogsystems abgebildet werden kann. Unter der Kategorie **`NAOqi Audio`** sind die Komponenten zusammengefasst, welche für die Audioeingabe, Verarbeitung und Ausgabe zuständig sind [Sof19c]. **`ALAudioDevice`** ist dabei das zentrale Modul, das die Eingabe und Ausgabe verwaltet. Mit Ausnahme von **`ALAudioPlayer`** wird es von den anderen Modulen als Quelle bzw. Senke verwendet. In der offiziellen Dokumentation wird beschrieben, dass **`ALAudioDevice`** die *Advanced Linux Sound Architecture (ALSA)* nutze [Sof19a]. Bei *ALSA* handelt es sich um ein Kernelmodul, welches mit Hilfe von Treibern auf die Audiohardware zugreift. Versuche[4] am Roboter ergaben jedoch die Erkenntnis, dass **`ALAudioDevice`** nicht direkt, sondern indirekt durch den Sound-Server *PulseAudio* mit *ALSA* kommuniziert. Das ist ein wichtiger Unterschied, wie sich in Abschnitt 4.3 zeigen wird. Abbildung 2.5 stellt den Audiozugriff zwischen den einzelnen Komponenten dar.

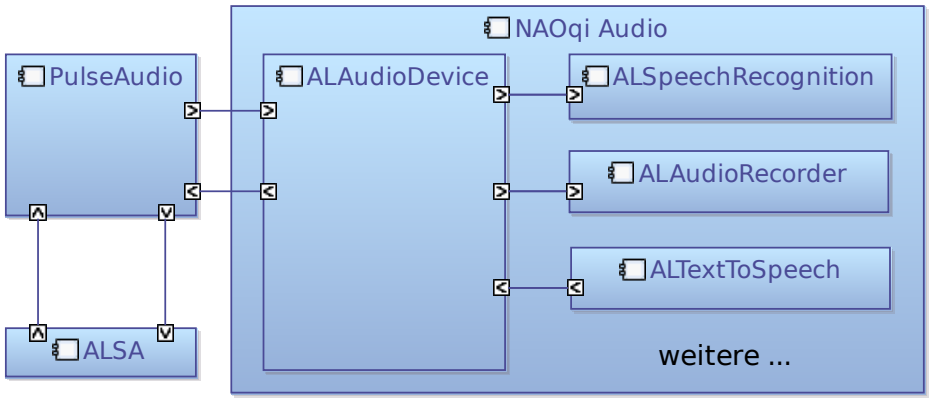

Abbildung 2.5: Audiozugriff von **`NAOqi Audio`**

Die Schnittstellen von **`ALAudioDevice`** ermöglichen Grundfunktionen wie das Einstellen der Ausgabelautstärke, Stummschalten, Starten und Stoppen der Tonaufnahme und einer Lautstärkeberechnung pro Mikrofon. Der Transport der Signale geschieht über den Austausch von Audiopuffern (Array aus abgetasteten Werten) mit einer festen Dauer von 170 ms. Die Abtastrate kann jedoch selbst definiert werden. Alternativ lässt sich anstelle des Eingangssignals der Mikrofone eine *WAVE*-Datei als Quelle verwenden.

Für das Speichern und Wiedergeben von Audiodateien sind die Module **`ALAudioRecorder`** und **`ALAudioPlayer`** zuständig. Im dem hier betrachteten Kontext haben sie aber eine eher unbedeutende Rolle. Mehr im Fokus der Betrachtung liegt **`ALSpeechRecognition`** als Implementierung der *ASR*. Dahinter verbirgt sich eine *VoCon Hybrid* genannte Spracherkennungssoftware. Der Name deutet auf die Eigenschaft, dass diese hybride Software sowohl offline operieren als auch einen Online-Dienst (Remote *ASR* Engine) zur Analyse des Signals hinzuziehen kann. Die große Anzahl an unterstützten Sprachen (darunter auch Deutsch) kann vom Online-Dienst leider nicht geboten werden [Sof19d], weshalb man sich bei der deutschen Sprache nur auf die Fähigkeiten des Offline-Teils berufen kann. Das Modul lässt sich so nutzen, dass es Text aus einem vorgegebenen Vokabular

[4] Ein Umstellen der standardmäßigen Quellen und Senken in *PulseAudio* nahm Einfluss auf die Audioeingabe und Ausgabe von *NAOqi*.

im Audiosignal erkennen kann. Als Vokabulareinträge sind einzelne Wörter oder Phrasen (Wortgruppen oder kurze Sätze) möglich. Um flexibler auf die Sprache reagieren zu können, kann das sogenannte Word-Spotting aktiviert werden. Somit muss das Gesagte nicht einem Vokabulareintrag entsprechen, sondern nur beinhalten, um erkannt zu werden. Eine weitere Möglichkeit ist das Definieren einer oder mehrerer Grammatiken, welche als Regelwerk für die zu erkennende Abfolge von Wörtern dienen, damit nicht jede gültige Variation einer Phrase als separater Eintrag angegeben werden muss. Es ist nicht dokumentiert mit welchen Verfahren oder ob die Audiodaten für die Spracherkennung überhaupt vorverarbeitet werden. Erste Untersuchungen[5] am laufenden *NAOqi* brachten z. B. zum Vorschein, dass es ein Modul namens **`AudioFilterLoader`** gibt. Dies wird jedoch weder in der Dokumentation erwähnt, noch hat es irgendwelche einsehbaren Schnittstellen.

Neben der beschriebenen Umwandlung der Sprache in Text, kann mit **`ALEmotionAnalysis`** auch unabhängig vom Inhalt die Gefühlslage der Stimme ausgewertet werden. Als Ergebnis wird die Intensität von Gelassenheit, Wut, Freude und Besorgtheit ermittelt. Aber auch Aufregung und Lachen kann registriert und ausgewertet werden. Wurde eine bestimmte Lautstärke überschritten, signalisiert es **`ALSoundDetection`** über ein Event, sofern es aktiviert wurde. Der Schwellwert ist konfigurierbar. Mit dem Modul **`ALSoundLocalization`** gibt es eine Komponente, die die Anordnung der Mikrofone ausnutzt, indem eine Richtungsortung (*Sound-Source-Localization (SSL)*) von Geräuschen durchgeführt wird. Bei mehreren gleichzeitigen Geräuschen wird aber nur die Richtung des lautesten bekanntgegeben.

Neben den beschriebenen Modulen aus der Kategorie **`NAOqi Audio`** gibt es weitere Module, welche die Nutzung auf einer höheren Abstraktionsebene ermöglichen. Erwähnenswert ist **`ALDialog`** mit einer mächtigen Verwaltung der Dialogführung [Sof19b]. Auch hier können Regeln für die Spracheingabe in Dateien definiert und nach Themen gruppiert werden. Dabei ist es sogar möglich, eine Grammatik (siehe Unterabschnitt 5.5.1) anzugeben, welche die Erkennung einer freien Formulierung zulässt. Im weiteren Unterschied zu **`ALSpeechRecognition`** sind die darauf passenden Antworten bei den Regeln mit hinterlegt und werden automatisch wiedergegeben. Nutzbar ist die Sprachsynthese auch über **`ALTextToSpeech`** bzw. **`ALAnimatedSpeech`** mit einer Gestikulierung.

2.2.3 Probleme

Bei der Nutzung von *Pepper* als Sprachdialogsystem existieren einige Probleme. Die wohl größte Schwachstelle ist die unzuverlässige Spracherkennung, was vermutlich nicht an der *ASR*-Engine selbst liegt, sondern an dem stark von Störsignalen behafteten Audiodaten. Sehr dominant sind die permanenten Geräusche der Lüfter des Roboters. So sorgt der Roboter für seine eigene Geräuschkulisse im unmittelbaren Umfeld. Das Montieren von Mikrofonen an anderen Stellen von *Pepper* würde nach der Einschätzung des Autors diesen Umstand nicht oder nur geringfügig verbessern.

Das zweite Problem erschwert die Kompensierung des ersten von Hardware verursachten Problems. Denn leider bietet **`ALAudioDevice`** nicht die Möglichkeit, Audiosignale umzuleiten, sodass z. B. keine selbst implementierte Signalverarbeitung vor **`ALSpeechRecognition`**

[5] Das Kommando `qicli` (ausgeführt in einer SSH-Login-Shell auf dem *Pepper*) listet alle aktuell geladenen Module auf.

zwischengeschaltet werden kann. Ein Austausch von Audiodaten zwischen den Modulen ist zwar prinzipiell möglich, jedoch müssten die Module dafür modifiziert werden, was aber ohne die Kenntnis des Quellcodes keine Option ist. Selbst, wenn der mögliche Umweg über das Speichern eines verbesserten Signals in Form von Dateien stattfindet, wird beim Einspeisen der Audiodaten in **`ALAudioDevice`** eine Rückkopplung erzeugt, da dadurch das Modul für die Verbesserung sein Ergebnis erneut als Eingangssignal erhält. Daher muss ein Wechselbetrieb zwischen der Vorverarbeitung des ursprünglichen Eingangssignals und der Nutzung des Signals stattfinden, was aber aufgrund der so entstehenden Phasen von Taubheit des Roboters nicht praktikabel ist. Innerhalb von *NAOqi* kann so also keine kontinuierliche Vorverarbeitung für bestehende Module zur Anwendung kommen.

Auch sind einzelne Bestandteile trotz des modularen Aufbaus von *NAOqi* fest in die Architektur integriert und nicht einzeln austauschbar, was die Verwendung von Alternativen behindert und kaum Flexibilität für den Ablauf der einzelnen Schritte bietet. Ein bedeutendes Beispiel ist das Modul **`ALSpeechRecognition`**, welches automatisch neben der Spracherkennung auch die *VAD* nutzt, um das Ende einer Rede festzustellen und den Aufnahmeprozess zu stoppen. Doch leider passiert dies oft zu früh, weshalb schon bei sehr kurzen Sprechpausen der Roboter mit einer erneuten Aufnahme anfängt, wodurch ein Verarbeiten von Reden, welche die Länge von kurzen Wortgruppen überschreiten, oft nicht möglich ist. Die im Hintergrund genutzte *ASR*-Engine ist auch nicht direkt austauschbar. Zudem ist das Dialogmanagement von **`ALDialog`** ebenfalls eng mit der *VoCon Hybrid*-Engine verknüpft.

Die offensichtliche Abwesenheit bzw. ungenügende Implementierung der Vorverarbeitung des Signals führt unter anderem dazu, dass während der Sprachausgabe keine Eingabe stattfinden kann, da sonst die Ausgabe der Sprachsynthese analysiert wird. Diese wird hier aber leider nicht herausgefiltert.

2.3 Eingabegeräte

Im Unterabschnitt 2.2.1 ist die Aufnahmetechnik von *Pepper* bereits grob vorgestellt worden. Das dortige Verwenden eines Mikrofon-Arrays anstelle eines einzelnen Mikrofons eröffnet mit Hilfe von den hier leider nicht bzw. nur zum Teil implementierten Signalverarbeitungsalgorithmen mehr Möglichkeiten zur Separierung von Nutz- und Störsignal, welche sich besonders im Anwendungsgebiet von Sprachdialogsystemen lohnen. Denn durch den Einsatz von mehreren Mikrofonen an verschiedenen Positionen trifft dort der Schall einer Quelle wie beispielsweise die Sprache des Nutzers zu unterschiedlichen[6] Zeiten ein, wodurch aus der Bestimmung des Zeitversatzes die Richtung der Schallquelle berechnet werden kann (*SSL*). Darüber hinaus ermöglichten Verfahren der *Sound-Source-Seperation (SSS)* das Fokussieren auf eine bestimmte Richtung bzw. das Ausblenden von Geräuschen aus anderen Richtungen. Diese Werkzeuge können in Kombination bei korrekter Anwendung für ein gutes Audiosignal sorgen, welches neben der Sprache kaum andere Geräusche enthält, was wiederum eine gute Voraussetzung für eine erfolgreiche Spracherkennung ist.

Neben den in *Pepper* integrierten Mikrofonen ist die Benutzung von weiterer Hardware ein Gewinn für die Aussagekraft der Evaluation der zu entwickelnden Softwarelösung, welche

[6] Es sei denn, die Quelle hat zu allen Mikrofonen denselben Abstand.

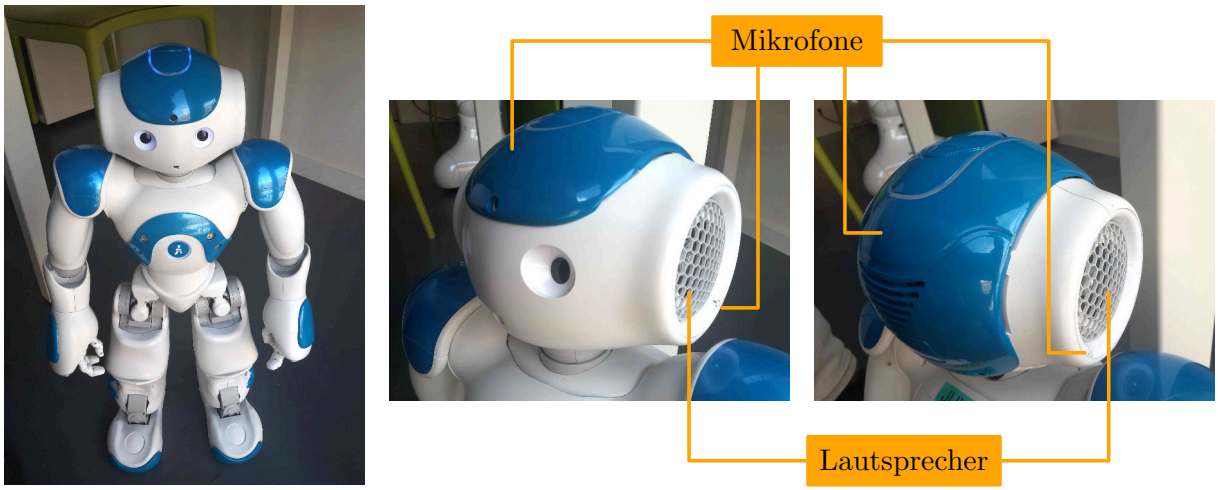

Abbildung 2.6: *NAOs* Mikrofone und Lautsprecher

nicht nur auf den *Pepper*-Roboter allein zugeschnitten, sondern prinzipiell in Verbindung mit anderen Hardwareplattformen und somit auch Eingabegeräten anwendbar sein soll. Ein Kandidat ist der kleinere *NAO*-Roboter mit einer anderen Mikrofonkonfiguration (siehe Abbildung 2.6), wobei hier die vier Mikrofone nicht auf einer Ebene liegen. Dieser ist dem *Pepper* von der Softwareinfrastruktur sehr ähnlich und bringt im Wesentlichen auch die gleichen Probleme, wie in Abschnitt 2.2.3 beschrieben, mit sich. Besser wäre aber das Benutzen von Hardware anderer Hersteller, wodurch auch die Unabhängigkeit von der Software *NAOqi* im Entwicklungsprozess praktisch getestet werden kann. Das muss nicht heißen, einen ganzen Roboter anzuschaffen. Allein alternative Audioeingabegeräte, welche sich am besten unkompliziert mit gängigen Rechnern verbinden lassen, reichen dafür völlig aus.

Folgende Kriterien müssen dabei erfüllt sein. Bei den Eingabegeräten muss es sich um Mikrofon-Arrays handeln, damit eben solche Signalverarbeitungsalgorithmen daran getestet werden können. Diese existieren in verschiedenen Bauformen. Bei linearen Arrays sind die Mikrofone in einer Reihe platziert. Für den Einsatz bei Robotikanwendungen sind diese eher weniger geeignet, da hier Nutzer aus allen Richtungen auf den Roboter einreden können. Mit dieser Form ist das Ergebnis der Winkelbestimmung immer zweideutig[7]. Würde man die Position der Schallquelle Q an der Mikrofonreihe spiegeln (Q'), dann käme derselbe Zeitversatz zustande, da sich die Entfernungen zu den Mikrofonen (M_1 bis M_4) nicht unterscheiden (siehe Abbildung 2.7). So ist nicht eindeutig klar, wo der Nutzer steht. Mit Hilfe von Mikrofonen mit Richtwirkung, die verschieden ausgerichtet sind, kann diese Information aber doch noch gewonnen werden. Allerdings wirkt hier die *SSS* auch symmetrisch, wodurch im Ergebnis die Geräusche aus der Spiegelrichtung kaum ausgeblendet werden.

Besser geeignet ist die Verwendung von zirkularen Arrays, wie sie beispielsweise beim *Pepper* eingesetzt werden. Durch die Anordnung auf einer Kreislinie ist nun bei einer Anzahl von drei oder mehr Mikrofonen die Eindeutigkeit der Winkelbestimmung zumindest innerhalb der Ebene gegeben. Dabei ist auch eine dreidimensionale *SSL* möglich, welche wiederum eine Zweideutigkeit aufweist, weil alle Mikrofone in einer Ebene liegen. Somit gibt es eine Symmetrie an dieser Ebene. Das ist jedoch weniger problematisch, da die

[7] Es sei denn, die Schallquelle liegt auf einer Linie mit den Mikrofonen.

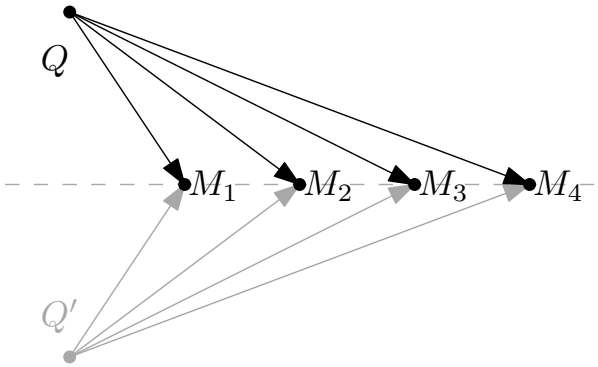

Abbildung 2.7: Symmetrie von linearen Mikrofon-Arrays

Schallquellen meist sowieso eher in der horizontalen Ebene angeordnet sind. Durch einige Tricks, welche die Direktivität der Mikrofone und die Abschattung durch die Form des Gerätes mit betrachten, können trotzdem die korrekten Richtungen bestimmt werden. Diese Bauform ist hier ausreichend. Darüber hinaus gibt es z. B. kugelförmige Bauformen, welche diese Limitierung nicht aufweisen, aber dann sperrig sind, da sie als separates Gerät mehr Volumen einnehmen bzw. sich nachträglich schwer in bestehende Roboter integrieren lassen, welche nicht dafür entworfen wurden. Weitere nicht unbedingt kategorisierbare Bauformen existieren daneben auch noch.

Da die einzelnen analogen Mikrofonsignale durch eine spezielle auf dem Gerät befindlichen Soundkarte umgesetzt werden müssen, ist ein digitaler Anschluss Voraussetzung, weil eine analoge Übertragung mittels eines Klinkensteckers nicht die nötige Anzahl an Kanälen bereitstellen kann. Die Schnittstelle für den Audiozugriff ist mit dem weit verbreiteten Standard der *Universal Serial Bus (USB)*-Geräteklasse eins nach [Kna06] aufgrund der guten Kompatibilität von Hardware und Software die erste Wahl. Denn durch die Verwendung von *USB*-Audio werden die Geräte nicht nur wegen der physischen *USB*-Steckverbindung unterstützt, sondern sie stellen die Audiodaten auch ohne den Einsatz gerätespezifischer Software bereit, die womöglich eigene Anforderungen an das zugrundeliegende Betriebssystem mitbringen würde. Welche Audiodaten das sind, ist auch nicht zu vernachlässigen. Um die volle Kontrolle über die Signalverarbeitung zu haben, ist es essenziell, dass alle Mikrofonsignale über *USB*-Audio einzeln ausgelesen werden können. Ein auf dem Gerät vorverarbeitetes und zusammengeführtes Signal eignet sich zwar für den Vergleich mit anderer Hardware, jedoch kann die eigene auf dem externen Rechner ausgeführte Softwarelösung hier nicht angewendet werden. Weitere Merkmale sind die Auflösung und die Abtastrate der integrierten Soundkarte. Bezüglich der Auflösung sind 16 bit pro abgetasteten Wert ausreichend, was die einer CD entspricht. Einige *ASR*-Engines unterstützen die Verarbeitung von sogar nur 8 bit Signalen. Wie stark die Auflösung im Endeffekt die Wirkung der Signalverarbeitung und das Ergebnis beeinflusst, wird sich erst bei der Umsetzung zeigen. Die Abtastrate definiert hingegen klarere Grenzen. Da die menschliche Sprache etwa im Frequenzbereich von 80 Hz bis 12 kHz liegt [Wik19a], muss nach dem *Nyquist-Shannon*-Abtasttheorem mindestens mit 24 kHz abgetastet werden, um diesen Frequenzbereich komplett analysieren zu können. Auch hier reicht mancher Spracherkennungssoftware ein geringerer Wert von 8 kHz, was bedeutet, dass das Frequenzband bis 4 kHz dafür eine ausreichende Informationsmenge enthält. In diesem Zusammenhang

spielt der Frequenzgang der Mikrofone eine ebenso wichtige Rolle. Denn wenn das Mikrofon nicht den gewünschten Frequenzbereich der akustischen Schwingung in ein elektrisches Signal umwandeln kann, dann hilft eine ausreichend hohe Abtastrate alleine auch nicht. Meist wird der Arbeitsfrequenzbereich so angegeben, dass dessen Grenzfrequenzen um 10 dB leiser aufgenommen werden als bei der Referenzfrequenz bei 1 kHz. Das heißt jedoch nicht unbedingt, dass Frequenzen außerhalb dieses Bereichs nicht im Signal vorhanden sind. Sie sind nur wesentlich leiser vertreten. Angesichts der geplanten Ausnutzung der Array-Konfiguration beeinflusst die höchst mögliche Abtastrate auch die Auflösung der Zeitmessung bei der *SSL*. Denn wenn der Schall innerhalb einer Abtastung die Strecke zwischen den Mikrofonen zurücklegen kann, dann ist davon im Audiosignal nur ein Versatz von maximal einer Zeiteinheit zu sehen. So kann höchstens bestimmt werden, welches Mikrofen den Schall zuerst empfangen hat. Eine genauere Richtungsbestimmung wird erst bei einer deutlich höheren Abtastrate möglich. Wie hoch diese sein muss, kann nicht pauschal festgelegt werden, da sie mit der Anzahl und Anordnung der Mikrofone zu tun hat. Darüber hinaus variiert die Schallgeschwindigkeit im Medium Luft bei verschiedenen Temperaturen.

Das Mikrofon-Array *16SoundsUSB* [Lét18] ist genau wie der Vorgänger *8SoundsUSB* [Lét17] ein Open-Hardware-Projekt von der *Université de Sherbrooke*[8]. Diese über *USB* ansteuerbaren Soundkarten können 16 bzw. 8 gleichzeitig angeschlossene und frei platzierbare Mikrofone verarbeiten. Doch leider werden diese nicht direkt verkauft. Eine eigene Fertigung basierend auf den frei verfügbaren Bauplänen würde den Rahmen dieser Arbeit sprengen.

Das Gerät *ReSpeaker Mic Array v2.0* [See19] (im Folgenden nur *ReSpeaker* genannt) besitzt mit seinen vier omnidirektionalen Mikrofonen eine ähnliche zirkulare Anordnung wie das Array beim *Pepper* (siehe Abbildung 2.8 links). Die Besonderheit ist, dass es sich hier nicht nur um eine reine Soundkarte handelt. Denn auf der Platine befindet sich eine Recheneinheit, welche bereits ohne das Zutun des externen Rechners Algorithmen der Signalverarbeitung für Mikrofon-Arrays anwendet. Somit kann dieses Array ohne weitere Software einfach für die Endanwendung mit einem vorverarbeiteten Signal benutzt werden. Für die Konfiguration und Kommunikation mit der Firmware können über ein *USB*-Protokoll Daten ausgetauscht werden, wofür der Hersteller bereits eine Software-Bibliothek bereitstellt. Darüber hinaus kann der *ReSpeaker* auch als Ausgabegerät verwendet werden. Mit einem an dem Audioausgang angeschlossenen Lautsprecher ist somit die Firmware in der Lage, das ihr bekannte Ausgangssignal, was sich auch in der Tonaufnahme wiederfindet, herauszufiltern. Diese Technik der *Acoustic-Echo-Cancellation (AEC)* ermöglicht beispielsweise eine Spracherkennung während einer Sprachausgabe wobei nur die Sprache des Nutzers erkannt wird. Um aber stattdessen die eigene Software zur Verarbeitung zu verwenden, gibt es die Wahl, verschiedene Firmware-Varianten über *USB* auf das Gerät zu laden, welche z. B. alle Rohsignale über *USB*-Audio verfügbar machen.

Ein weiteres Eingabegerät ist das Array *UMA-8 v2* (im Folgenden nur *UMA-8* genannt) von *miniDSP*[9] [min18]. Es weist ähnliche Eigenschaften wie der *ReSpeaker* auf. Die auf dem ersten Blick klar erkennbaren Unterschiede sind die größeren Ausmaße und eine zirkulare Anordnung von sechs Mikrofonen mit einem weiteren Mikrofon in der Mitte (siehe Abbildung 2.8). Auch dieses Array kann mit verschiedenen Firmware-Varianten bestückt

[8] siehe `https://www.usherbrooke.ca/` (besucht am 23.01.2020)
[9] siehe `https://www.minidsp.com/` (besucht am 23.01.2020)

werden. Jedoch ist es beim *UMA-8* im Gegensatz zum *ReSpeaker* nicht möglich, die Rohdaten und das von der integrierten Signalverarbeitung resultierende Signal gleichzeitig abzurufen. Es kann entweder nur der eine oder nur der andere Modus verwendet werden. Die in der Abbildung 2.8 gekennzeichnete digitale I^2S-Schnittstelle stellt einen Audioausgang und einen weiteren Audioeingang bereit. So ist weitere Hardware nötig, wenn analoge Lautsprecher in Verbindung mit der integrierten *AEC* genutzt werden sollen.

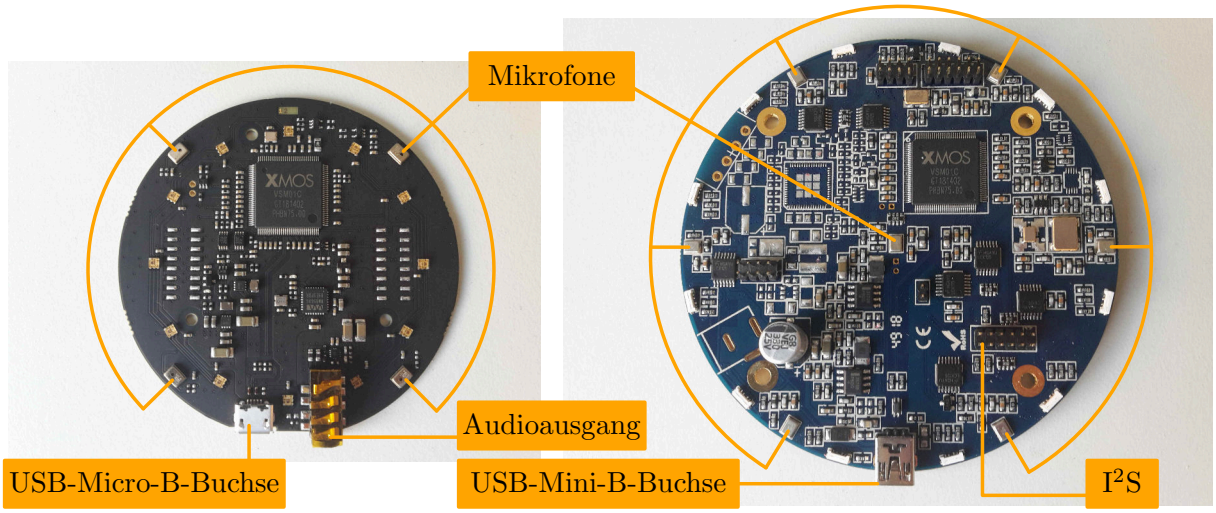

Abbildung 2.8: *ReSpeaker* (links) und *UMA-8* (rechts)

Beide Arrays können mit einer Auflösung von bis zu 24 bit und einer Abtastrate von 48 kHz über *USB*-Audio betrieben werden und genügen damit den Ansprüchen, weshalb sie für die Entwicklungsarbeit beschafft wurden. Unter der Annahme einer Ausbreitungsgeschwindigkeit[10] c von $343\,\mathrm{m\,s^{-1}}$ [Wol03, S. 97] bewegt sich der Schall innerhalb eines Abtastintervalls t_s etwa um 7 mm fort (s) (siehe Formel (2.1)). Verglichen mit den Durchmessern der Geräte von 7 cm beim *ReSpeaker* und 9 cm beim *UMA-8* erscheint diese Abtastrate f_s ausreichend für eine *SSL*. Die Tabelle 2.1 fasst die technischen Daten inklusive der Frequenzbereiche von den zu verwendenden Geräten zusammen.

$$s = c \cdot t_s = \frac{c}{f_s} = \frac{343\,\mathrm{m\,s^{-1}}}{48000\,\mathrm{Hz}} \approx 7{,}15\,\mathrm{mm} \tag{2.1}$$

Tabelle 2.1: Technische Daten der Eingabegeräte

	Mikrofonanzahl	max. Abtastrate	max. Auflösung	Frequenzbereich
Pepper	4	192 kHz	32 bit	100 Hz bis 10 kHz
ReSpeaker	4	48 kHz	24 bit	20 Hz bis 10 kHz
UMA-8	7	48 kHz	24 bit	100 Hz bis 10 kHz

2.4 ASR-Engines

Neben dem Eingabegerät befindet sich auf der anderen Seite der Systemgrenze die Spracherkennungssoftware. Im Falle des *Pepper*-Roboters ist dies die in Unterabschnitt 2.2.1

[10] Schallgeschwindigkeit in Luft bei 20 °C

bereits behandelte *VoCon Hybrid*-Engine, welche bei Benutzung der deutschen Sprache im Offline-Modus arbeitet. Auch hier soll die zu entwickelnde Software nicht nur an eine einzige *ASR*-Engine angepasst, sondern grundsätzlich leicht verknüpfbar mit Alternativen sein. Darüber hinaus zeichnet eine Evaluation mit mehren verschiedenen Spracherkennungsprogrammen ein erweitertes Bild.

Die Wahl für Alternativen muss nicht unbedingt auf die Software fallen, welche die beste Erkennungsrate oder eine sehr hohe Verbreitung und Etablierung aufweist. Vielmehr ist es wichtig sie flexibel für das Testen und die Auswertung einsetzen zu können. Dabei kann es durchaus von Vorteil sein, wenn die *ASR*-Engine nicht als ein Online-Dienst auf fremder Infrastruktur bereitgestellt wird. Zum einen ist es aus der Datenschutzsicht problematisch personenbezogene Daten des Nutzers oder anderer in der Nähe befindlicher Personen in Form von Audioaufnahmen ihrer Stimme an externe Unternehmen weiterzuleiten. Und zum anderen ist eine lokal ausführbare Implementierung nicht an eine Netzwerkverbindung gebunden und somit nicht so stark von zusätzlichen Latenzen bei der Audiodatenübertragung betroffen. Falls die Roboterplattform nicht die benötigten Leistungsmerkmale bzw. Speicherkapazität aufweist, kann die Spracherkennungssoftware immer noch auf einem externen Rechner im eigenen Intranet betrieben werden. Da das *RoboticLab* neben Englisch die Spracherkennung hauptsächlich auf Deutsch einsetzt und aufgrund dessen die Evaluierung der Software mit deutscher Sprache stattfindet, muss sie diese Sprache auch verarbeiten können bzw. kompatible Modelle aufweisen. Darüber hinaus ist eine Diktierfunktion Voraussetzung, wobei frei formulierte Sprache in Text übersetzt und für weitere Verarbeitungen zugänglich gemacht wird. Die Spracherkennung muss außerdem sprecherunabhängig funktionieren, wobei nicht erst die Stimmen der Nutzer eingelernt werden müssen. Es existiert eine große Vielfalt an *ASR*-Engines, die diese Kriterien erfüllen. Aus Gründen der Nachvollziehbarkeit und Reproduzierbarkeit für andere Entwickler liegt der Fokus hier auf der Nutzung quelloffener und frei verfügbarer Software zur Spracherkennung.

Im Gegensatz zu den meisten proprietären Produkten sind im Bereich der freien Software in der Regel ganze Toolkits für alle Schritte rund um die Entwicklung von *ASR* gegeben, was auch bedeutet, dass selten eine Plug-and-Play-Lösung zu finden ist. So sind oft mehrere Schritte und ein gutes Grundverständnis nötig, um eine funktionierende Spracherkennung aufzusetzen. Denn nicht nur die Software selbst, sondern auch passende Modelle für die zu verwendende Sprache sind Voraussetzung. Dazu zählt das akustische Modell, welches eine Zuordnung von Merkmalen im Audiomaterial auf Phoneme (ähnlich zu Laute) abbildet, ein Modell der Aussprache, welches für jedes mögliche Wort Folgen aus Phonemen speichert und ein Sprachmodell, das die Wahrscheinlichkeiten aufeinander folgender Wörter abbildet. Das akustische Modell wird auf Grundlage einer großen Menge an Trainingsdaten generiert. Ein sogenannter Korpus umfasst dabei unzählige Stunden an Audiomaterial von vielen unterschiedlichen Personen vorgelesenen und ebenfalls abgespeicherten Texten, woraus das entsprechende Toolkit ein Training durchführt. Dieser Prozess kann je nach Verfahren und Umfang des Korpus mehrere Tage dauern, weshalb die Verfügbarkeit von bereits vortrainierten akustischen Modellen zu begrüßen ist. Die hier im Interesse stehende Spracherkennung führt ein Decoder genannter Softwarebestandteil durch, welcher anhand von Merkmalen, die von kurzen aufeinanderfolgenden Zeitabschnitten des Audiosignals extrahiert wurden, in den Modellen nach der wahrscheinlichsten passenden Lösung (Wörter) sucht.

Julius[11] ist eine recht schlanke Spracherkennungssoftware. Es handelt es sich um einen reinen Decoder, welcher kompatibel mit Modellen ist, die mit dem *Hidden-Markov-Model-Toolkit* generiert wurden. Neben den offiziell bereitgestellten japanischen und englischen Modellen, wird bei *VoxForge*, einer Plattform für das Zusammentragen von freien Sprachkorpora, auch ein deutsches trainiertes Modell für *Julius* verfügbar gemacht[12], welches leider nur das akustische Modell beinhaltet und so allein nicht direkt nutzbar ist.

Kaldi auf der anderen Seite ist ein sehr umfangreiches Toolkit, welches sehr stark auf die Forschung ausgelegt ist. Ein deutsches Modell existiert auch, welches gleich auf Basis von drei Korpora trainiert wurde. Dazu zählen der *Tuda-De*-Korpus, der an der *Technischen Universität Darmstandt* in Kooperation mit der *Universität Hamburg* aufgenommen wurde, *Spoken Wikipedia Corpora*[13] und das *M-AILABS Speech Dataset*[14] [MK18].

Das *CMU Sphinx*-Projekt ist hingegen eher auf die praktische Anwendung bezogen [Car18]. Die verschiedenen Softwarebestandteile sind einzeln installierbar. Mit *PocketSphinx* ist der Decoder gegeben, welcher die offiziellen Modelle nutzen kann, worunter sich auch ein deutschsprachiges[15] auf Basis vom *VoxForge*-Korpus befindet.

Mozillas Projekt *Common Voice*[16] ist ähnlich wie *VoxForge* eine Plattform zum Zusammentragen von freien Korpora. Die Motivation ist unter anderem das auch von *Mozilla* stammende *DeepSpeech*-Toolkit mit einer guten Datengrundlage zu versorgen. Ein vortrainiertes Modell für die deutsche Sprache ist verfügbar [AZ19]. Dieses basiert neben dem *Common Voice*-Korpus auch auf den *VoxForge*- und *Tuda-De*-Korpora.

Ein weiteres *ASR*-Toolkit ist *wav2letter++* [Pra+18]. Die darauf aufbauende Software *speechless*[17] ist auf die Generierung und Verwendung eines deutschen Modells angepasst. Durch die Transfer-Learning-Technik kann ein in englischer Sprache trainiertes neuronales Netz, wofür es mehr umfangreichere Korpora[18] gibt, mit vergleichsweise geringem Aufwand auf deutsch umgelernt werden.

Tabelle 2.2 fasst die existierenden Schnittstellen und die unterstützten Eingabeformate (Auflösung, Abtastrate und Kanäle) zusammen. Möglicherweise existieren weitere inoffizielle Schnittstellen in hier nicht aufgeführten Programmiersprachen. Die Angabe von *Command-Line-Interface (CLI)* weist darauf hin, dass eine Nutzung über die Kommandozeile und somit auch durch Shell-Skripte möglich ist.

Die Nutzung aller dieser Programme und Toolkits für die Evaluation der Signalverarbeitung würde in Anbetracht der weiteren Konfigurationsmöglichkeiten der Eingabegeräte und akustischen Umgebungen den Rahmen dieser Arbeit sprengen. Daher werden neben der *VoCon Hybrid*-Engine nur wenige Alternativen eingebunden. Schnittstellen für weitere

[11] *Julius*: `https://github.com/julius-speech/julius` (besucht am 23.01.2020)
[12] deutsche Modelle auf *VoxForge*: `http://www.voxforge.org/de/Downloads` (besucht am 23.01.2020)
[13] *Spoken Wikipedia Corpora*: `https://nats.gitlab.io/swc/` (besucht am 23.01.2020)
[14] *M-AILABS Speech Dataset*: `https://www.caito.de/2019/01/the-m-ailabs-speech-dataset/` (besucht am 23.01.2020)
[15] deutsches Modell für *PocketSphinx*: `https://sourceforge.net/projects/cmusphinx/files/Acoustic%20and%20Language%20Models/German/` (besucht am 23.01.2020)
[16] *Common Voice*: `https://voice.mozilla.org/de` (besucht am 23.01.2020)
[17] *speechless*: `https://github.com/JuliusKunze/speechless` (besucht am 23.01.2020)
[18] Hier wurde der *LibriSpeech ASR Corpus* (`http://www.openslr.org/12` (besucht am 23.01.2020)) verwendet.

auch hier nicht erwähnte *ASR*-Engines können bei Bedarf später immer noch hinzugefügt werden.

Tabelle 2.2: Gegenüberstellung der *ASR*-Engines

	Schnittstellen	Eingabeformat
Julius	*CLI*, C, C++, *Python*	16 bit, 16 kHz oder weitere[19], Mono
Kaldi	C++, *Python*	nicht spezifiziert
PocketSphinx	*CLI*, *Python*, Java, Android	16 bit, 16 kHz oder 8 kHz[19], Mono
DeepSpeech	*CLI*, *Python*, Node.JS, Go, Rust, GStreamer	16 bit, 16 kHz, Mono
wav2letter++	C, C++, *Python*	16 bit, 16 kHz, Mono

[19]Die Abtastrate muss mit der der Trainingsdaten des verwendeten akustischen Modells übereinstimmen.

3 Anforderungsanalyse

Im Folgenden werden die Anforderungen an das zu entwickelnde Softwaresystem, welches die Grundlage für die spätere Implementierung der Signalverarbeitung ist, ermittelt. Um nun in der Formulierung präziser darauf zu referenzieren, wird an dieser Stelle der Arbeitstitel *RoSaSS* vergeben und im weiteren Verlauf der Arbeit benutzt. Im ersten Schritt werden die Anwendungsfälle dokumentiert, aus denen sich im nächsten Schritt die Anforderungen ergeben.

3.1 Anwendungsfälle

Als Anwendungsfälle oder Use-Cases werden funktionale Einheiten beschrieben. Sie bringen die zweckerfüllenden Grundfunktionen des Systems hervor, die mit der Außenwelt interagieren und durch Aktoren ausgelöst werden. Aktoren können z. B. Nutzer des Systems in verschiedenen Rollen oder auch Nachbarsysteme sein. Da es sich bei *RoSaSS* um keine Anwendung für den eigentlichen Endnutzer des Roboters handelt, sondern um ein Framework mit entsprechenden Strukturen und Schnittstellen, stellt der Aktor Nutzer eine Person dar, welche zum einen Zugriff auf den Roboter hat, um ihn zu konfigurieren und zum anderen Programmierkenntnisse sowie ein Grundverständnis von der Softwareinfrastruktur besitzt. Weitere Aktoren sind die Nachbarsysteme *NAOqi* sowie für *NAOqi* entwickelte Module, welche mit *RoSaSS* interagieren und hier *NAOqi*-Clients genannt werden und die *ASR*-Engines.

Jeder der folgenden Anwendungsfälle hat eine eindeutige Identifikation und eine Beschreibung. Darüber hinaus sind die Aktoren aufgelistet, welche an dieser Stelle mit dem System interagieren. Manche Anwendungsfälle lassen sich nur unter bestimmten Voraussetzungen bzw. Systemzuständen anwenden. Ein Hinweis dazu ist bei der jeweiligen Vorbedingung zu finden. Die Anmerkungen beinhalten eine erweiterte Beschreibung inklusive Ableitungen der Anforderungen.

UC-1 **Aktoren:** Nutzer **Anforderungen:** REQ-9, REQ-10, REQ-14

Vorbedingung: Es sind zu *RoSaSS* kompatible Softwareerweiterungen zur Signalverarbeitung vorhanden.

Beschreibung: Der Nutzer integriert Signalverarbeitungsverfahren durch das Einbinden von Softwareerweiterungen in *RoSaSS*.

Anmerkungen: Daraus ergibt sich, dass *RoSaSS* in der Lage sein muss, die Audiodaten zu verarbeiten (REQ-9) und Erweiterungen für die Signalverarbeitung zu verwenden (REQ-10). Die Erweiterungen können bereits bestehende oder vom Nutzer neu entwickelte Implementierungen sein. Da ein einziger Verarbeitungsschritt bei vielen Anwendungsszenarien voraussichtlich nicht ausreicht, ist eine Verknüpfung von mehreren Erweiterungen zu einem komplexeren Verarbeitungsprozess nötig (REQ-14).

UC-2 **Aktoren:** Nutzer, *NAOqi* **Anforderungen:** REQ-4, REQ-9, REQ-23

Vorbedingung: UC-1 wurde ausgelöst.

Beschreibung: Der Nutzer konfiguriert *RoSaSS* auf dem *Pepper* so, dass das Eingangssignal von *RoSaSS* vorverarbeitet und an *NAOqi* weitergeleitet wird.

Anmerkungen: Die Spracherkennung von *Pepper* wird ausschließlich über die unveränderte *NAOqi API* verwendet. Alle bisherigen und zukünftig für *Pepper* entwickelten Anwendungen, die die *VoCon Hybrid*-Engine nutzen, können somit ohne Änderungen am Quellcode bei einer gut implementierten bzw. konfigurierten Vorverarbeitung von einer geringeren Wortfehlerrate profitieren. Also muss *RoSaSS* die Audiodaten vom Mikrofon-Array abgreifen (REQ-4), verarbeiten (REQ-9) und an *NAOqi* weiterleiten (REQ-23) können.

UC-3 **Aktoren:** Nutzer **Anforderungen:** REQ-3, REQ-10, REQ-19, REQ-20, REQ-21, REQ-22

Vorbedingung: -

Beschreibung: Der Nutzer integriert eine Dialoglogik in *RoSaSS*.

Anmerkungen: Die Dialoglogik (Erweiterung (REQ-10)) legt auf Basis von analytischer Signalverarbeitung fest, wann verschiedene Zustände der Spracheingabe (z. B. kein Zuhören, passives Zuhören und aktives Zuhören) und Sprachausgabe (z. B. Stille oder Sprachsynthese) ablaufen. Das setzt voraus, dass steuerbare Komponenten in *RoSaSS* existieren, die Audio von einem Gerät aufnehmen (REQ-3) und wiedergeben (REQ-19) können und das gleichzeitig mit verschiedenen oder den selben Geräten (REQ-20, REQ-21 und REQ-22). Die Nutzung der Spracherkennung in *NAOqi* ist dabei schlecht möglich, da die dort integrierte Dialoglogik nicht umgangen werden kann.

UC-4 **Aktoren:** Nutzer, *ASR*-Engine **Anforderungen:** REQ-11, REQ-18

Vorbedingung: -

Beschreibung: Der Nutzer integriert Schnittstellen zu einer *ASR*-Engine, welche das Audiosignal von *RoSaSS* entgegen nimmt und den transkribierten Text an *RoSaSS* zurückliefert.

Anmerkungen: Die Strukturen in *RoSaSS* müssen daher eine Audioeinspeisung (REQ-11) zu und eine Textentgegennahme (REQ-11) von *ASR*-Engines vorsehen.

UC-5 **Aktoren:** *NAOqi*-Client **Anforderungen:** REQ-26

Vorbedingung: UC-4 wurde ausgelöst.

Beschreibung: Der erkannte Text der in *RoSaSS* eingebundenen Spracherkennung wird für *NAOqi*-Clients verfügbar gemacht.

Anmerkungen: Es muss eine Schnittstelle existieren, mit welcher *NAOqi*-Clients das Ergebnis der in *RoSaSS* eingebundenen Spracherkennung nutzen können (REQ-26). Anders als bei UC-2 müssen die *NAOqi*-Clients darauf angepasst sein.

UC-6 **Aktoren:** Nutzer **Anforderungen:** REQ-1, REQ-2, REQ-16, REQ-17

Vorbedingung: -

Beschreibung: Der Nutzer führt *RoSaSS* auf einem Rechner aus.

Anmerkungen: Beim Rechner kann es sich um den Entwicklungscomputer des Nutzers handeln oder um das Herzstück eines Roboters, wobei *Pepper* mit eingeschlossen wird (REQ-1). Es ist davon auszugehen, dass der Nutzer die nötigen Rechte besitzt Software wie etwa Abhängigkeiten von *RoSaSS* zu installieren. Besonders wichtig ist dabei die Unterstützung von *GNU/Linux*-Betriebssystemen, da diese häufig zur Steuerung von Robotern verwendet werden und sich mit Hilfe von Virtualisierungstechnologien (virtuelle Maschinen oder Container) auch gut unter anderen Plattformen nutzen lassen. Dabei ist *Ubuntu* bei Robotikanwendungen eine sehr verbreitete und somit auch hier im Fokus der Betrachtung stehende Distribution (REQ-2). Die native Verwendung unter aktueller Versionen anderer Betriebssysteme wie *macOS* (REQ-16) oder *Windows* (REQ-17) ist wünschenswert aber nicht zwingend notwendig.

UC-7 **Aktoren:** Nutzer **Anforderungen:** REQ-27, REQ-28

Vorbedingung: UC-6 wurde ausgelöst.

Beschreibung: Der Nutzer konfiguriert *RoSaSS* so, dass das Eingangssignal von *Pepper's* Mikrofon-Array über eine Netzwerkverbindung entgegengenommen wird.

Anmerkungen: Diese Funktionalität ist hilfreich für den Entwicklungsprozess, da so live mit den Audiodaten der Zielhardware getestet werden kann, ohne dabei die Flexibilität der auf dem Entwicklungsrechner anpassbaren Software aufgeben zu müssen. Damit das Ergebnis der Verarbeitung nicht verfälscht wird, muss die Übertragung (REQ-27) ohne eine verlustbehaftete Kompression stattfinden (REQ-28).

UC-8 **Aktoren:** Nutzer **Anforderungen:** REQ-3, REQ-4, REQ-5, REQ-6, REQ-29

Vorbedingung: UC-6 wurde ausgelöst.

Beschreibung: Der Nutzer konfiguriert *RoSaSS* so, dass mit dem Rechner verbundene Audio-Geräte als Audiodatenquelle für *RoSaSS* verwendet werden.

Anmerkungen: Zu den Audiogeräten (REQ-3) zählen mindestens *Pepper's* Mikrofon-Array (REQ-4) und der über USB (REQ-5) verbundene *ReSpeaker* bzw. *UMA-8*. Das beudetet auch, dass *RoSaSS* in der Lage sein muss, eine Anzahl von mindestens bis zu sieben Kanälen in Aufnahme (REQ-6) und Verarbeitung (REQ-29) zu unterstützen, damit die technischen Möglichkeiten vom *UMA-8* (sieben Mikrofone) voll ausgenutzt werden können.

UC-9 **Aktoren:** Nutzer **Anforderungen:** REQ-30, REQ-31

Vorbedingung: UC-7 oder UC-8 wurde ausgelöst.

Beschreibung: Der Nutzer löst eine persistente Audioaufnahme des Eingangssignals aus.

Anmerkungen: Dafür ist eine verlustfreie (REQ-31) Speicherung (REQ-30) nötig.

UC-10 **Aktoren:** Nutzer **Anforderungen:** REQ-13

Vorbedingung: UC-9 wurde ausgelöst.

Beschreibung: Der Nutzer konfiguriert eine Audiodatei als Eingangssignal.

Anmerkungen: Das Abspielen einer Audiodatei (REQ-13) ist hilfreich für den Entwicklungsprozess, da so die Software mit gleichen Ausgangsbedingungen getestet werden kann, ohne dabei die Schallemission in die akustische Umgebung nachstellen zu müssen. Außerdem kann dies auch ohne Roboter und Eingabegeräte getan werden.

UC-11 **Aktoren:** Nutzer **Anforderungen:** REQ-12, REQ-15

Vorbedingung: UC-1 wurde ausgelöst.

Beschreibung: Der Nutzer konfiguriert über *RoSaSS* Parameter der Erweiterungen.

Anmerkungen: Für eine Anpassung der Signalverarbeitung, ohne den Quellcode der Erweiterung ändern zu müssen, ist eine Parametrisierung hilfreich (REQ-15). Da es in dieser Arbeit darum geht, eine passende Vorverarbeitung zu entwickeln, soll es möglich sein, dies programmatisch während der Laufzeit zu tun (REQ-12). Das ist eine Voraussetzung für eine automatische Bestimmung der optimalen Parameterwerte durch systematisches Ausprobieren.

3.2 Anforderungen

Die in diesem Abschnitt aufgelisteten und aus den Anwendungsfällen hervorgegangenen Anforderungen an *RoSaSS* unterteilen sich in funktionale Anforderungen, welche darlegen, was das System können soll, und nicht funktionale Anforderungen (Qualitätsanforderungen und Rahmenbedingungen). Eine weitere Unterscheidung zwischen Muss- und Kann-Anforderungen legt fest, welche Anforderungen essenziell und welche zwar wünschenswert aber jedoch nicht erforderlich und daher optional sind.

3.2.1 Funktionale Anforderungen

REQ-3 *RoSaSS* **muss** das Audiosignal von einem Audioeingabegerät aufnehmen.

REQ-4 *RoSaSS* **muss** das Mikrofon-Array von *Pepper* als Audioeingabegerät unterstützen.

REQ-5 *RoSaSS* **muss** USB-Audioeingabegeräte unterstützen.

REQ-9 *RoSaSS* **muss** eine Vorverarbeitung des Eingangssignals durchführen können.

REQ-10 *RoSaSS* **muss** Erweiterungen zur Signalverarbeitung nutzen können.

REQ-11 *RoSaSS* **muss** das Eingangssignal an *ASR*-Engines weiterleiten können.

REQ-13 *RoSaSS* **muss** das Einspielen von persistierten Audiodaten als Eingangssignal unterstützen.

REQ-14 *RoSaSS* **muss** die Audioströme zwischen den Erweiterungen verknüpfen können.

REQ-15 *RoSaSS* **muss** Parameter der Erweiterungen konfigurieren können.

REQ-18 *RoSaSS* **muss** das Ergebnis der *ASR*-Engine entgegennehmen können.

REQ-19 *RoSaSS* **muss** Audio durch ein Wiedergabegerät ausgeben können.

REQ-20 *RoSaSS* **muss** das gleichzeitige Verwenden von Audioeingabe und Audioausgabe unterstützen.

REQ-21 *RoSaSS* **muss** Audioeingabe und Audioausgabe vom selben Gerät unterstützen.

REQ-22 *RoSaSS* **muss** Audioeingabe und Audioausgabe von verschiedenen Geräten unterstützen.

REQ-23 *RoSaSS* **muss** den vorverarbeiteten Audiostrom an *NAOqi* übergeben können.

REQ-26 *RoSaSS* **kann** den Ergebnistext der *ASR*-Engine für *NAOqi*-Clients verfügbar machen.

REQ-27 *RoSaSS* **kann** einen Netzwerkstrom vom Mikrofon-Array des *Peppers* als Eingangssignal verwenden.

REQ-30 *RoSaSS* **muss** das Eingangssignal in eine Datei persistieren können.

3.2.2 Qualitätsanforderungen

REQ-6 Die Audioaufnahme **muss** bis zu 7 Kanäle unterstützen.

REQ-12 Die Konfiguration der Parameter der Erweiterungen **muss** während des Betriebs von *RoSaSS* möglich sein.

REQ-28 Der Netzwerkstrom **muss** die Audiodaten verlustfrei übertragen.

REQ-29 Die Schnittstelle für die Erweiterungen zur Signalverarbeitung **muss** bis zu 7 Kanäle unterstützen.

REQ-31 Die Speicherung **muss** mit einem verlustfreien Audioformat erfolgen.

3.2.3 Randbedingungen

REQ-1 *RoSaSS* **muss** unter dem Betriebssystem *NAOqi OS* 2.5.8 oder höher ausführbar sein.

REQ-2 *RoSaSS* **muss** unter dem Betriebssystem *Ubuntu* 16.04 oder höher ausführbar sein.

REQ-16 *RoSaSS* **kann** unter dem Betriebsystem *macOS* 10.12 oder höher ausführbar sein.

REQ-17 *RoSaSS* **kann** unter dem Betriebsystem *Windows* 10 oder höher ausführbar sein.

4 Entwurf von RoSaSS

Dieses Kapitel legt die Entstehung des Konzeptes zum Aufbau von *RoSaSS* dar. Bevor es in Abschnitt 4.2 zur Technologieauswahl und anschließend daran in Abschnitt 4.3 zum eigentlichen Konzept kommt, wird für dieses System eine Selektion relevanter Technologien zunächst in Abschnitt 4.1 grob vorgestellt. Denn da es bereits eine Vielzahl an nutzbaren Technologien gibt, deren Verwendung gegenüber einer kompletten Eigenimplementierung den Aufwand erheblich senkt, ist es sinnvoll, darauf aufzubauen. Jedoch muss vorher eine gut überlegte Auswahl getroffen werden, bei der die Merkmale im Kontext des zu erreichenden Zielsystems betrachtet werden.

4.1 Technologieüberblick

Die folgenden Unterabschnitte führen in Modularisierungsframeworks als Alternativen zu *NAOqi*, in Audiosysteme zum Zugriff auf Audiogeräte und Signaltransport, in Audio-Plug-in-Standards zum Einbinden existierender Verarbeitungssoftware sowie in Persistierungsmöglichkeiten durch Audiodateiformate ein.

4.1.1 Modularisierungsframeworks

Auch wenn die Software *NAOqi* aufgrund ihres exklusiven Vorkommens auf bestimmten Robotern wie *Pepper* oder *NAO* nicht als Grundlage von *RoSaSS* in Frage kommt, so ist die Grundidee trotzdem interessant. Das Aufteilen von Softwarekomponenten (z. B. Verarbeitungsverfahren) in verschiedene voneinander abgekapselte Module, welche zur Laufzeit dazugeschaltet und auch noch entfernt miteinander kommunizieren können, führt bei korrekter Anwendung zu einer enormen Flexibilität, welche bei *RoSaSS* benötigt wird, da es auch als eine Experimentierplattform nutzbar sein soll. Mit einer solchen serviceorientierten bzw. ereignisgesteuerten Architektur lassen sich mit wenig Aufwand einzelne Module zu einem Gesamtsystem kombinieren, welches so für unterschiedliche Anwendungsszenarien auch verschiedene Gestalten annehmen kann. Die Tatsache einer netzwerkgestützten Kommunikation, welche die Aufteilung des Systems auf mehrere Rechner erlaubt, ist für die Erfüllung von REQ-27 zur entfernten Nutzung von *Pepper's* Mikrofonen hilfreich. Folgend werden die hier als Modularisierungsframeworks bezeichneten Alternativen zu *NAOqi* beschrieben.

- *Open Services Gateway initiative (OSGi)*[20] ist ein offener Standard für eine eben solche Modularisierung in der Programmiersprache Java, welcher von der *OSGi*-Allianz entwickelt wird und auch in der Industrie eine starke Verbreitung hat. Auch wenn *OSGi*

[20] *OSGi*: `https://www.osgi.org/` (besucht am 23.01.2020)

allgemein gehalten und nicht speziell für Audio- oder Robotikanwendungen gedacht ist, gib es Vorteile gegenüber *NAOqi*, wie beispielsweise eine hohe Robustheit und ein besser durchdachtes Design (Kommunikation zwischen den Komponenten unter Verwendung von klar definierten Schnittstellen, Handhabung von verschiedenen Zuständen von Komponenten, etc.).

- Das Framework *injected Plain Old Python Object (iPOPO)*[21] greift die Ideen von *OSGi* auf und macht sie für die Programmiersprache *Python* verfügbar ohne jedoch den Standard bedienen zu können, da kein Java verwendet wird.
- Im Kontext von Robotikanwendungen darf *Robot Operating System (ROS)*[22] in dieser Auflistung nicht fehlen. *ROS* ist eine lange etablierte Grundlage für unzählige existierende robotische Systeme. Der Fokus liegt auf der Übertragung von Sensormesswerten und Steuerbefehlen zwischen den einzelnen Softwarekomponenten. Darüber hinaus gibt es ein breit gefächertes Ökosystem mit Programmen zur Visualisierung und Simulation. Die hauptsächlich unterstützte Plattform ist *Ubuntu*. Eine Vielzahl an nutzbaren Modulen ist verfügbar. Allerdings ist für den Audiobereich recht wenig zu finden. Anders als bei *OSGi* und *iPOPO* herrscht hier eine prinzipielle Programmiersprachenunabhängigkeit. Die beste Unterstützung gibt es, neben Schnittstellen für einige andere Programmiersprachen, für *C++* und *Python*.
- *Robot Operating System 2 (ROS2)* ähnelt zwar *ROS(1)* in einigen Punkten, aber trotzdem sollte es hier als eine separate Alternative aufgelistet sein. Denn es ist eine parallel entstehende Neuentwicklung, wobei konzeptionelle Schwächen von *ROS*, welche sich über die Jahre herauskristallisiert haben, in einem neuen Designprozess eliminiert wurden. Die Implementierung dieses Designs ist zwar noch nicht vollständig abgeschlossen, jedoch ist der Stand nach Einschätzung des Autors bereits gut nutzbar. Zu den Vorteilen gegenüber *ROS* zählen unter anderem eine etwas einfachere Installation mit besserer Unterstützung der Betriebssysteme *macOS* und *Windows*, eine vollständig dezentrale Kommunikationsstruktur mit Möglichkeiten zur Verschlüsselung, Rechtevergabe und Definition von *Quality-of-Service (QoS)* (Garantien der Nachrichtenübermittlung). Momentan werden nur die Programmiersprachen *C++* und *Python* unterstützt. Über eine Brücke lassen sich auch für *ROS* entwickelte Module an *ROS2* mit ein paar Einschränkungen anschließen.

4.1.2 Audiosysteme

Für den Zugriff auf Audiodaten und/oder den Transport zwischen verschiedenen Softwarekomponenten gibt es Audiosysteme. Es ist zwingend notwendig, dass *RoSaSS* sich mit mindestens einem verbinden kann.

- Unter *GNU/Linux* kann das Vorhandensein von *ALSA* als vorausgesetzt angesehen werden. Dieses Kernelmodul verwaltet die Audiohardware und den Audiozugriff mit Hilfe von Gerätetreibern. Allerdings funktioniert es nur unter *GNU/Linux*. Andere Betriebssysteme haben ihre eigenen Schnittstellen zu den Audiogeräten.

[21] *iPOPO*: `https://ipopo.readthedocs.io/en/0.8.1/index.html` (besucht am 23.01.2020)
[22] *ROS*: `https://www.ros.org/` (besucht am 23.01.2020)

- *PortAudio* kann genau genommen nicht als Audiosystem gezählt werden, da es sich nur um eine Softwarebibliothek handelt. Erwähnenswert ist sie durch die Tatsache, dass sie den Zugriff auf die Audiodaten abstrahiert und kompatibel zu verschiedene Backends wie etwa *ALSA* ist. Somit kann bei Verwendung von *PortAudio* eine Plattformunabhängigkeit geschaffen werden.
- *PulseAudio* kommuniziert auch nicht direkt mit der Hardware. Dieser Sound-Server sorgt für eine sehr flexible Handhabung von Audiostreams sowie die gleichzeitige Nutzung der Audiogeräte von mehreren Programmen. Darüber hinaus lässt sich *PulseAudio* mittels einem *CLI* oder auch graphischer Anwendungen konfigurieren. *PulseAudio* ist primär für die Verwendung von *ALSA* unter *GNU/Linux* gedacht und ist dort auch sehr verbreitet (z. B. auch in *NAOqi OS*). Unter anderen Plattformen ist dieser Sound-Server wenn überhaupt nur eingeschränkt nutzbar.
- *JACK Audio Connection Kit (JACK)*[23] ist ein Soundsystem, welches besonders bei Software für Musikproduktion unter *GNU/Linux* zur Anwendung kommt. Herausragend ist die Möglichkeit zum Verknüpfen von Audiostreams zwischen *JACK*-kompatiblen Anwendungen und das bei sehr geringer Latenz. Durch eine Auswahl von verschiedenen Backends ist *JACK* plattformübergreifend nutzbar. Das gleichzeitige Verwenden von mehreren Audiogeräten ist zwar möglich, jedoch ist diese Software nicht dafür gedacht.

4.1.3 Audio-Plug-in-Standards

Um Anforderung REQ-10 (Einbinden von Erweiterungen zur Signalverarbeitung) zu erfüllen, bietet sich der Einsatz von Audio-Plug-in-Formaten an. So können zudem bereits existierende Implementierungen eingebunden werden, sofern *RoSaSS* eine solche Plug-in-Schnittstelle integriert. Es gibt jedoch eine Vielzahl an unterschiedlichen Formaten.

- *Virtual Studio Technology (VST)*[24] ist der Industriestandard im Bereich von Audio-Plug-ins. Auch wenn die Plug-ins unter verschiedenen Plattformen genutzt werden können, wenn sie entsprechend dafür kompiliert wurden, sind die meisten Plug-ins eher für *Windows* verfügbar. Denn unter *macOS* hat sich ein anderes Format durchgesetzt und durch die proprietäre Lizenzierung, welche erst in der Version 3 des Formates umgangen werden kann, wurde es unter *GNU/Linux* eher weniger verwendet.
- Das proprietäre Konkurrenzformat *Audio Unit (AU)*[25] kann nur unter *macOS* und *IOS* verwendet werden und scheidet somit aus der Betrachtung aus.
- *Linux Audio Developer's Simple Plugin API (LADSPA)*[26] ist eine offene Alternative, welche zwar primär für *GNU/Linux* gedacht aber auch unter anderen Betriebssystemen verwendbar ist, sofern das konkrete Plug-in kompatible Binaries aufweist.

[23] *JACK*: `https://jackaudio.org/` (besucht am 23.01.2020)
[24] *VST*: `https://www.steinberg.net/en/company/technologies/vst3.html` (besucht am 23.01.2020)
[25] *AU*: `https://developer.apple.com/documentation/audiounit` (besucht am 23.01.2020)
[26] *LADSPA*: `https://www.ladspa.org/` (besucht am 23.01.2020)

- Die Weiterentwicklung von *LADSPA* namens *LADSPA Version 2 (LV2)* ist heute das gängige Plug-in-Format unter *GNU/Linux*. Genau wie der Vorgänger *LADSPA* kann der offene Standard auch unter anderen Plattformen verwendet werden.

4.1.4 Audiodateiformate

Hinsichtlich der Anforderungen REQ-13 und REQ-30 muss *RoSaSS* Audiodaten einlesen und auch persistieren können. Es gibt eine riesige Auswahl an Audiodateiformaten, welche durch REQ-31 erheblich eingeschränkt wird. Denn bei den meisten Formaten wird mit verlustbehafteter Kompression gearbeitet. Hier sollen allerdings keine Informationen verloren gehen. Des Weiteren müssen mindestens sieben Audiokanäle unterstützt werden (REQ-6), damit alle Mikrofone vom *UMA-8* aufgenommen werden können.

- Eine Lösung ist, die Audiodaten, so wie sie im Speicher vorliegen, zu serialisieren und abzuspeichern. Somit kann das Format genau so gewählt werden, wie es gebraucht wird. Der große Nachteil ist die dadurch nicht vorhandene Kompatibilität mit anderer Software bzw. bereits vorliegenden Dateien.
- Das *WAVE*-Format genießt hingegen eine sehr hohe Verbreitung. Es wird von sehr vielen Audioverarbeitungs- oder Bearbeitungsprogrammen unterstützt und kann deshalb als der „gemeinsame Nenner“ betrachtet werden. Auch Aufnahmen in den meisten Sprachkorpora liegen in diesem Format vor. Die maximale Kanalanzahl ist mit 65535 praktisch unbegrenzt. In der gewöhnlichen Form liegen die Datem unkomprimiert vor, was in einer erheblichen Dateigröße resultiert. Andere Datenformate können in einer *WAVE*-Datei auch hinterlegt werden. Jedoch werden diese von vielen Programmen nicht unterstützt.
- Der *Free Lossless Audio Codec (FLAC)*[27] bietet eine verlustfreie Kompression. Verglichen mit *WAVE* ist *FLAC* weniger verbreitet. Es ist eine Anzahl von bis zu acht Kanälen möglich.

4.2 Technologieauswahl

Als erstes stellt sich die Frage, ob ein Modularisierungsframework als Unterbau zur Verwendung kommen soll oder ob stattdessen die Aufgabe des dynamischen Audiodatenaustauschs zwischen Softwarekomponenten schon von Audiosystemen wie *JACK* abgedeckt werden kann. Auch wenn *JACK* sehr gut für eine effiziente Audioübertragung geeignet ist und sich bestehende Software integrieren lässt, welche beispielsweise Plug-ins laden oder Visualisierungen anzeigen kann, hat ein solches System auch einen Nachteil. Denn die Nachrichtenübertragung ist alleine für die synchronisierte Übermittlung von Audiopuffern ausgelegt. Andere Daten wie z. B. Befehle oder Texte könnten über das *Musical Instrument Digital Interface (MIDI)*-Protokoll[28] innerhalb von *JACK* oder als Bytesequenz in Audiopuffern verpackt ausgetauscht werden. Dieses Vorgehen sorgt jedoch für eine starke Intransparenz und erfordert die Verwendung eines eigenen Protokolls. Da es bei *RoSaSS*

[27] *FLAC*: `https://xiph.org/flac/` (besucht am 23.01.2020)
[28] *MIDI*: `https://www.midi.org/specifications-old/item/the-midi-1-0-specification` (besucht am 23.01.2020)

nicht nur um Audiodaten, sondern auch um jegliche anderen Daten gehen kann, ist stattdessen der Einsatz von allgemeinen Modularisierungsframeworks die bessere Wahl. Wenn dadurch die extrem geringe Latenz der Audioübertragung etwas größer wird, ist das nicht von großem Nachteil. Denn hier geht es nicht um die Echtzeitverarbeitung und Wiedergabe von Musik, sondern eine Aufbereitung und Übermittlung von Sprachaufnahmen an eine *ASR*-Engine. Ein Hybridansatz, wobei nur die Audiodaten mit *JACK* übertragen werden und die restlichen Nachrichten innerhalb des Modularisierungsframeworks, macht das Gesamtsystem unübersichtlich und schwer zu durchschauen.

ROS hat gegenüber den anderen Frameworks *OSGi* und *iPOPO* den Vorteil, dass sich Module kombinieren lassen, welche in unterschiedlichen Programmiersprachen entwickelt wurden. Das ist zwar keine Voraussetzung, aber so kann z. B. ein vergleichsweise schneller Entwicklungsprozess in *Python* stattfinden, wobei die Möglichkeit besteht, einzelne Module später durch *C++*-Äquivalente auszutauschen, sofern sich bei der *Python*-Implementierung gravierende Geschwindigkeitsnachteile ergeben. Außerdem ist so die Auswahl an nutzbaren Softwarebibliotheken größer, wenn die Sprache nicht festgelegt sein muss. Bei vielen Robotern ist *ROS* häufig schon gegeben. Die *NAO*- und *Pepper*-Roboter gehören zu den Ausnahmen. Wenn sowohl *RoSaSS* als auch die restliche Programmierung eines Roboters auf *ROS* basiert, dann ist die Einbindung von und die Interaktion mit *RoSaSS* wesentlich einfacher und flexibler.

Die Auswahl auf den Nachfolger *ROS2* ist aufgrund des Entwicklungsstandes mit einem Risiko verbunden, da es sein kann, dass benötigte Funktionen noch nicht oder nur eingeschränkt unterstützt werden. Jedoch erweckt *ROS2* bereits einen recht soliden Eindruck. Da voraussichtlich keine Abhängigkeiten in Form von *ROS(1)*-Paketen benötigt werden, spielt die Tatsache, dass viele Pakete noch nicht nach *ROS2* portiert wurden, keine Rolle. Außerdem kommt die bessere Unterstützung für *macOS* und *Windows* den Anforderungen REQ-16 und REQ-16 entgegen.

Für den Audiozugriff muss *RoSaSS* sich nicht auf eine einzige Schnittstelle beschränken. Um eine bessere Plattformunabhängigkeit zu gewährleisten, ist *PortAudio* eine gute Wahl. Jedoch ist der Einsatz von *PulseAudio* flexibler und eine Möglichkeit, beim *Pepper*-Roboter das vorverarbeitete Signal an *NAOqi* weiterzuleiten (REQ-23).

Als Audiodateiformat erfüllt *FLAC* die Anforderungen. Aber mit einer maximalen Kanalanzahl von acht, ist nicht mehr viel Spielraum nach oben. Sollte zukünftig mit einem Mikrofon-Array experimentiert werden, welches beispielsweise zehn Kanäle aufweist, muss das Format gewechselt werden. Aus diesem Grund und auch vor allem wegen der höheren Kompatibilität, liegt die Konzentration auf dem *WAVE*-Format.

4.3 Konzept

Nachdem nun die Technologien grob ausgewählt wurden, folgt eine genauere Beschreibung zur Architektur von *RoSaSS*. Das Grundprinzip ist, dass sich jede Komponente auf eine Aufgabe beschränkt. Diese Komponenten werden bei *ROS2* Knoten genannt. Es gibt beispielsweise für den Audiozugriff Knoten, die als Signalquellen und auch welche, die als Senken agieren, um die Audiodaten für das restliche System verfügbar zu machen bzw. auszugeben. Abbildung 4.1 verdeutlicht schematisch die Zusammenhänge von allen

Varianten (unter *GNU/Linux*), welche während der Ausführung von *RoSaSS* gleichzeitig genutzt werden können aber nicht müssen. Die Pfeile symbolisieren den Datenfluss. *PulseAudio* ist die zu präferierende Schnittstelle für Audiogeräte, wobei *PortAudio* alternativ genutzt werden kann, wenn *PulseAudio* auf einem System nicht verfügbar ist.

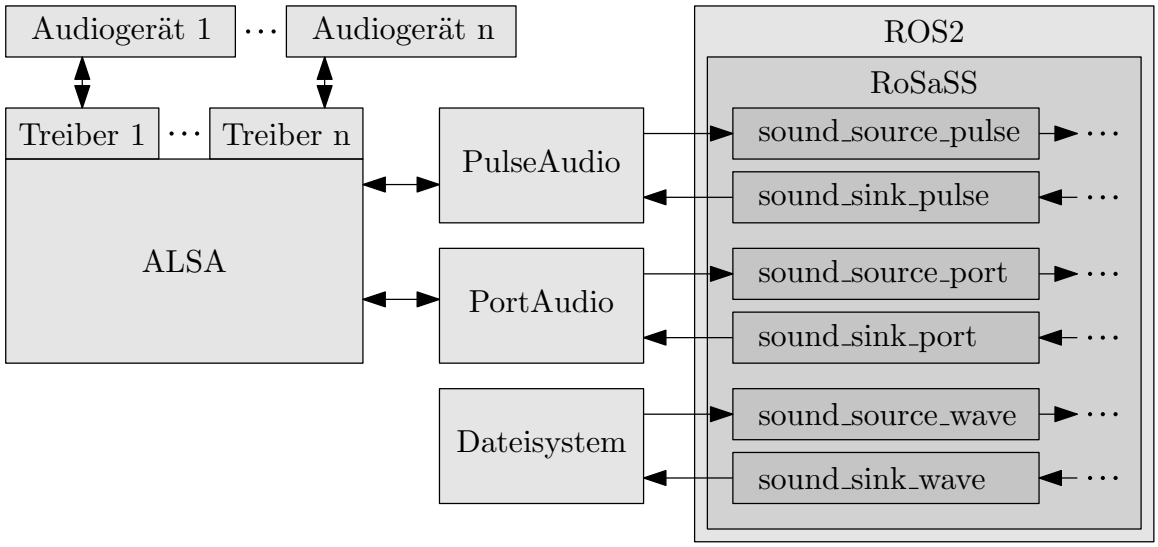

Abbildung 4.1: Schematische Darstellung des Audiozugriffs in *RoSaSS*

Die Stärken von *PulseAudio* lassen sich besonders beim folgenden Szenario demonstrieren. Um Anforderung REQ-23 zu erfüllen, kann auf dem *Pepper PulseAudio* so konfiguriert werden, dass *NAOqi* das Eingangssignal von *RoSaSS* bekommt (siehe Abbildung 4.2). Grundsätzlich macht *PulseAudio* jedes von *ALSA* erkannte Gerät als Senke (rechts) und / oder Quelle (links) den Anwendungen verfügbar, welche ihrerseits einen oder mehrere Streams mit diesen Geräten verknüpfen können. Diese Streams lassen sich in *PulseAudio* zu anderen Geräten unterbrechungslos umhängen, ohne, dass die betreffende Anwendung etwas davon merkt. Darüber hinaus wird für jede Senke ein Monitor als Quelle angeboten, welche das in die Senke eingegebene Signal wieder verfügbar macht. So lässt sich beispielsweise abhorchen, was der Lautsprecher ausgibt. Sind mehrere Streams von der selben oder von unterschiedlichen Anwendungen an einer Senke registriert, dann werden die Signale automatisch zusammengemischt.

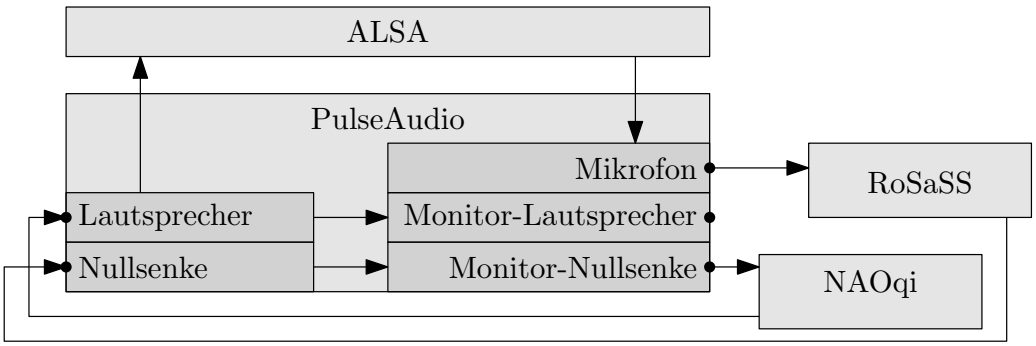

Abbildung 4.2: Schematische Darstellung der Konfiguration in *PulseAudio*

Normalerweise greift *NAOqi* auf das Eingangssignal vom Mikrofon-Array zu. Um dort *RoSaSS* zwischenschalten zu können, muss zunächst eine sogenannte Nullsenke erstellt werden. Dabei handelt es sich um ein virtuelles Ausgabegerät, welches nicht mit einer Hardware verknüpft ist [Gay17, Abschnitt „Object hierarchy“ / „Device“]. So kann *RoSaSS* das verarbeitete Signal in die Nullsenke geben, sodass ein Umstellen des Eingangsstreams von *NAOqi* auf den Monitor der Nullsenke zur gewünschten Konstellation führt.

Ähnlich wie beim Audiozugriff soll es je einen Knoten als Schnittstelle für jede eingebundene *ASR*-Engine geben. Dieses Prinzip gilt auch für Signalverarbeitungsverfahren, welche entweder direkt als Knoten implementiert oder aber von einem Knoten als *LV2*-Plug-in geladen werden. Da in *ROS2* beliebige Knoten auch zur Laufzeit dem System hinzugefügt werden können, wird eine sehr hohe Erweiterbarkeit erreicht (REQ-10). Der Austausch von Nachrichten zwischen Knoten geschieht über *Topics* (Nachrichtenkanäle), welche einen eindeutigen Namen tragen. Beim Starten von Knoten können diese Namen umdefiniert werden, ohne den Quellcode ändern zu müssen. Auf diese Weise lassen sich Knoten über einen gleichnamigen Kommunikationskanal miteinander verbinden, sofern das selbe Nachrichtenformat genutzt wird. Somit lassen sich erweiterte Verarbeitungsverfahren tatsächlich mit dem restlichen Graph verknüpfen (REQ-14). *ROS2*-Knoten können zudem Parameter deklarieren, welche zur Laufzeit von außen inspiziert und geändert werden können. So lassen sich verschiedene Parameterwerte ausprobieren bzw. dynamisch anpassen, ohne *RoSaSS* dabei neu starten zu müssen. Der Plan ist, dass auch alle Parameter von Audio-Plug-ins als änderbare *ROS2*-Parameter verfügbar gemacht werden. Damit wäre Anforderung REQ-15 erfüllt.

Nach REQ-18 muss *RoSaSS* die erkannten Texte von den eingebundenen *ASR*-Engines entgegen nehmen können. Bei der Verwendung der in *NAOqi* integrierten Software *VoCon Hybrid* ist demnach eine Schnittstelle von *RoSaSS* nach *NAOqi* erforderlich. Auch in der umgekehrten Richtung muss eine Interaktion von *NAOqi* aus möglich sein, um die in *RoSaSS* verwendete Spracherkennung dort nutzen zu können (REQ-26). Beides ließe sich als ein *ROS2*-Knoten realisieren, welcher gleichzeitig auch ein *NAOqi*-Modul ist bzw. auf *NAOqi* zugreift.

Die eigentliche Implementierung aller Knoten findet primär in der Programmiersprache *Python* statt, da diese Sprache gegenüber der anderen Auswahlmöglichkeit in *ROS2* (*C++*) den Vorteil eines leichteren Entwicklungsprozesses bietet. Bei dieser Interpretersprache ist es möglich, den Programmiercode interaktiv Zeile für Zeile auszuprobieren. Außerdem müssen *Python*-Programme nicht vorher kompiliert werden. Das Testen unter verschiedenen Betriebssystemen und Prozessorarchitekturen, wie sie bei Robotern vorkommen, kann somit ohne eine spezifische Neukompilierung geschehen. Der Nachteil ist zwar allgemein eine geringere Ausführungsgeschwindigkeit, jedoch gibt es für viele rechenintensive Aufgaben Softwarebibliotheken, welche sehr effizient arbeiten, da sie selbst in *C / C++* entwickelt wurden. Ein Beispiel dafür ist *NumPy* zur numerischen Verarbeitung von Arrays und mehrdimensionalen Matrizen. Generell ist die Verfügbarkeit von *Python*-Bibliotheken für die Nutzung spezifischer Technologien recht hoch. Dies zeigt auch ein Blick in die Tabelle 2.2 in Abschnitt 2.4. Nicht nur die dort in Betracht gezogenen *ASR*-Engines sondern auch *NAOqi* lassen sich mittels *Python* anbinden.

5 Implementierung von RoSaSS

Dieses Kapitel gibt einen Einblick in die Implementierungsdetails von *RoSaSS*. Dabei geht es zunächst um den Installationsprozess und wie er auf Robotern wie *Pepper* stattfinden kann. Anschließend wird nach einem kurzen Überblick themenweise auf die Implementierung und Funktionsweise eingegangen. Es werden hier allerdings nicht alle Bestandteile erklärt, da ein Teil der Komponenten in den folgenden Kapiteln genauer beschrieben werden.

5.1 Installation

Die Beilage 1 enthält eine detaillierte Installationsanleitung, welche sich auf *Ubuntu* 18.04 bezieht. *RoSaSS* liegt in Form von *ROS2* Paketen sowie in einzelnen *Python* Skripten vor, welche genauer in den späteren Abschnitten beschrieben werden. Bei der Installation von *ROS2* auf dem Roboter gibt es einiges zu beachten. Näheres wird im folgenden Unterabschnitt dargelegt. Auf Grundlage des dort geschilderten Vorgehens können auch die weiteren Abhängigkeiten installiert werden.

```
1 $ source /opt/ros/dashing/setup.bash
2 $ colcon build
3 Starting >>> rosass_msgs
4 Finished <<< rosass_msgs [26.6s]
5 Starting >>> rosass
6 Finished <<< rosass [0.77s]
7
8 Summary: 2 packages finished [27.5s]
```

Abbildung 5.1: Kommandos zum Bauen eines Workspaces

Für die Installation von *RoSaSS* selbst muss ein sogenannter Workspace angelegt werden. Ein *ROS2*-Workspace ist eine Verzeichnisstruktur, welche einerseits den änderbaren Quellcode von *ROS2*-Paketen enthält und andererseits das Bauen dieser Pakete ermöglicht, sodass sie in ausführbarer Form in weiteren Unterverzeichnissen abgelegt werden. Um einen Workspace mit *RoSaSS* in einem gewünschten Verzeichnis zu erzeugen, müssen die Pakete von *RoSaSS* in einem Unterverzeichnis namens `src` platziert werden. Befindet man sich mit der Konsole im Workspace-Verzeichnis, können nun mit dem Befehl `colcon build` die Pakete gebaut werden. Voraussetzung dafür ist, dass die *ROS2*-Installation in dieser Sitzung aktiviert wurde. Abbildung 5.1 zeigt in der Zeile 1 die Aktivierung, wobei im Hintergrund Umgebungsvariablen und Pfadangaben gesetzt werden, damit alle Hilfsprogramme von *ROS2* sowie die Kernpakete benutzbar sind. Die *ROS2*-Installation liegt

selbst auch in einer Workspace-Struktur vor. Ist dies geschehen, können nun Knoten von *RoSaSS* ausgeführt werden. Einzelheiten dazu sind im Abschnitt 5.6 zu finden.

5.1.1 Installation von ROS2

Durch den modularen Aufbau von *RoSaSS* müssen nicht zwangsläufig alle Abhängigkeiten von allen Knoten vorher installiert werden, wenn nur eine Teilmenge davon verwendet wird. Allen gemeinsam ist aber die Abhängigkeit *ROS2*. Hierfür wurde die *ROS2*-Distribution mit dem Codenamen *Dashing Diademata* oder einfach nur *Dashing*, welche bei Beginn der Implementierung von *RoSaSS* die neuste Version war, mittels der offiziellen Anleitung [Ope19] auf dem Entwicklungsrechner installiert. Um *ROS2* auf dem *Pepper*-Roboter zum Laufen zu bekommen, müssen notwendige Vorkehrungen getroffen werden. Denn dieser Installationsprozess kann nicht direkt unter *NAOqi OS* durchgeführt werden. Grund sind vom Hersteller gewollte Einschränkungen in den Konfigurationen des Betriebssystems wie beispielsweise die Deaktivierung des *root*-Nutzers, was die Installation von zusätzlicher Software stark erschwert.

Eine Lösung ist das Projekt *ros2_pepper* [Fer18]. Diese Sammlung an Konfigurationen und Installationsskripten ist dafür gedacht *ROS(1)* und auch *ROS2* auf einen Entwicklungsrechner so zu kompilieren, dass das Kompilat einfach nur auf den Roboter kopiert werden muss und danach ausgeführt werden kann. Möglich wird dies durch den vom Hersteller erhältlichen Cross-Compiler, welcher Programme erzeugt, die für das Betriebssystem und die Prozessorarchitektur des Roboters kompatibel sind. Für die Kompatibilität mit den in *NAOqi OS* installierten Bibliotheken wird beim Cross-Kompilieren eine Toolchain verwendet, welche die Softwarebibliotheken, deren Header-Dateien und weitere Konfigurationen beinhaltet, damit die erzeugten Programme hiermit verlinkt werden können. Das Projekt *ros2_pepper* erweitert diese Toolchain um viele Abhängigkeiten von *ROS* und den Kernpaketen von *ROS* selbst, damit auch eigene Pakete damit kompiliert werden können. Nach den eigenen Erfahrungen des Autors ist dies aber eine sehr labile Angelegenheit, wobei unter Umständen einige manuelle Eingriffe und langwierige Versuche nötig sind, um den Kompilierungsprozess erfolgreich zu beenden. Darüber hinaus muss dieses Setup bei jeder weiteren Abhängigkeit aufwendig mitgepflegt werden. Die Skripte von *ros2_pepper* installieren auch Pakete und deren Abhängigkeiten für das Abgreifen und Verarbeiten von Sensordaten, welche aus *NAOqi* stammen, sowie die Steuerung der Mechanik (z. B. Gelenke, Fahrwerk) aus *ROS* heraus. Für andere Projekte wie zur Ortung und Navigation ist dies eine hilfreiche Grundlage. Für *RoSaSS* jedoch findet sie in dieser Form keine Verwendung.

Aus diesen Gründen wurde sich für einen anderen Weg entschieden. In diesem Fall kann eine Virtualisierung weiterhelfen. So lässt sich durch Software ein System (Gastsystem) innerhalb eines anderen Betriebssystems (Wirtssystem) ausführen. Es gibt verschiedene Stufen der Virtualisierung. Auf der einen Seite wird mit einer sogenannten virtuellen Maschine die Hardware und Firmware eines Computers emuliert. Dadurch wird ein hoher Grad an Abkapselung zum Wirtssystem erreicht. Die Hardwareemulation ist jedoch recht rechenintensiv und sorgt für eine schlechte Performanz der dort ausgeführten Software. Auf der anderen Seite gibt es die Container-Virtualisierung. Hierbei liegen alle Dateien des Gastsystems in einem Verzeichnis oder Archiv. Beim Ausführen wird nur der Zugriff auf das Dateisystem virtualisiert, wodurch dieses Verzeichnis zum Wurzelverzeichnis wird.

Die gestarteten Programme werden direkt durch den Kernel des Wirtssystems ausgeführt. So entsteht nur ein extrem geringer Zuwachs an benötigter Rechenleistung. Ein weiterer Vorteil gegenüber virtuellen Maschinen ist die fehlende Notwendigkeit das Gastsystem erst booten zu müssen. Abhängig von der konkreten Implementierung müssen Gast- und Wirtssysteme aber eine gewisse Kompatibilität untereinander aufweisen.

Die meisten Container-Virtualisierungslösungen scheiden für den Gebrauch auf *Pepper* aus, da sie für die Installation und/oder Einrichtung *root*-Privilegien benötigen. Das betrifft aber nicht alle Implementierungen. Ein Gegenbeispiel ist *PRoot*. Dieses Programm kann in Binärform für die Plattform des Wirtssystems heruntergeladen und mit der Angabe des neuen Wurzelverzeichnisses ausgeführt werden, ohne dass dafür weitere Abhängigkeiten installiert oder Kernel-Einstellungen getätigt werden müssen. In der Installationsanleitung (siehe Beilage 1) sind genauere Instruktionen beschrieben, wie auf dem Roboter ein minimales *Ubuntu* 18.04 mit *ROS2* aufgesetzt werden kann. Innerhalb des Gastsystems kann man sich als *root*-Nutzer anmelden und beliebige Software mit Hilfe des dort zur Verfügung stehenden Paketmanagers installieren. Ein Problem ist dabei die Prozessorarchitektur *i386*. Denn *ROS2* wird in Binärform nur für die Architekturen *amd64*, *arm64* und *armhf* zur Installation angeboten[29]. Aus diesem Grund muss *ROS2* selbst kompiliert werden. Die offizielle Anleitung kann dafür aber nicht eins zu eins übernommen werden, da einige Abhängigkeiten auch erst manuell kompiliert und installiert werden müssen. Dieser ganze Installationsprozess muss nicht zwangsläufig auf dem Roboter ablaufen und kann auch mit *PRoot* auf einem 64-Bit-Rechner geschehen.

Der wohl größte Nachteil von *PRoot* ist das recht langsame Ausführen von Dateisystemzugriffen. Dadurch werden Prozesse, bei denen mit vielen Dateien interagiert wird oder Dateisuchen stattfinden, extrem verlangsamt. Spürbar wird dies bei Kompiliervorgängen und bei der Nutzung des *CLI* von *ROS2*. So dauert ein Starten und Initialisieren von *ROS2*-Knoten einige Sekunden, da sehr viele Suchprozesse zum Auflösen von Paketnamen, Pfaden und Import von *Python*-Bibliotheken involviert sind. Die Ausführungsgeschwindigkeit zur Laufzeit der Knoten ist davon aber nicht betroffen.

5.2 Überblick

Der Quellcode von *RoSaSS* ist in drei Verzeichnisse gegliedert. Das Verzeichnis **`bin`** enthält eigenständige Programme, welche nicht als *ROS2*-Knoten ausgeführt werden. Bei den restlichen zwei Verzeichnissen handelt es sich um *ROS2*-Pakete. Das Paket **`rosass`** enthält die Implementierungen der Knoten sowie sogenannte Launch-Skripte für verschiedene Anwendungsszenarien, die Vorschriften enthalten, wie welche Knoten mit welchen Parameterwerten und Verknüpfungen zueinander automatisch gestartet werden sollen. Denn bei der Verwendung von vielen Knoten wird das manuelle Starten und Konfigurieren von allen einzelnen Knoten für sich sehr mühsam und unübersichtlich. Zu guter Letzt sind im Paket **`rosass_msgs`** die Nachrichtenformate definiert, welche die Knoten zur Kommunikation nutzen. In *ROS2* wird das mit der *ROS Interface Definition Language (ROSIDL)* umgesetzt. Anhand dieser Beschreibungen mit fest definierten Typen für jedes Datenfeld wird beim Bauen eine Codegenerierung durchgeführt. Die daraus entstehenden *C++*- und

[29] siehe Paketquellen für *Ubuntu* 18.04 unter `http://packages.ros.org/ros2/ubuntu/dists/bionic/main/` (besucht am 30.01.2020)

Python-Klassen können dann von den Knotenimplementierungen programmiersprachenübergreifend verwendet werden. Darüber hinaus findet eine weitere Codegenerierung und ein anschließender Kompilierprozess statt, welcher dafür sorgt, dass die *ROS2*-Nachrichten zur Laufzeit so umgewandelt werden können, dass das darunterliegende Back-End, welches für die eigentliche Netzwerkkommunikation zuständig ist, diese verarbeiten kann. Ein solches Back-End wird in diesem Kontext auch *ROS Middleware (RMW)*-Implementierung genannt. *ROS2* selbst kann für verschiedene Back-Ends konfiguriert werden.

5.3 Audiozugriff

Die Implementierungen für den Audiozugriff sind in den Unterpaketen **`sound_source`** und **`sound_sink`** zu finden. Dabei ist die gemeinsame Funktionalität für jede spezielle Implementierung in einer Basisklasse zusammengefasst, welche diese weiter vererbt. Das Klassendiagramm 5.2 visualisiert die Vererbungshierarchie von den Knoten für den lesenden Audiozugriff, welche Daten von außerhalb in *RoSaSS* verfügbar machen.

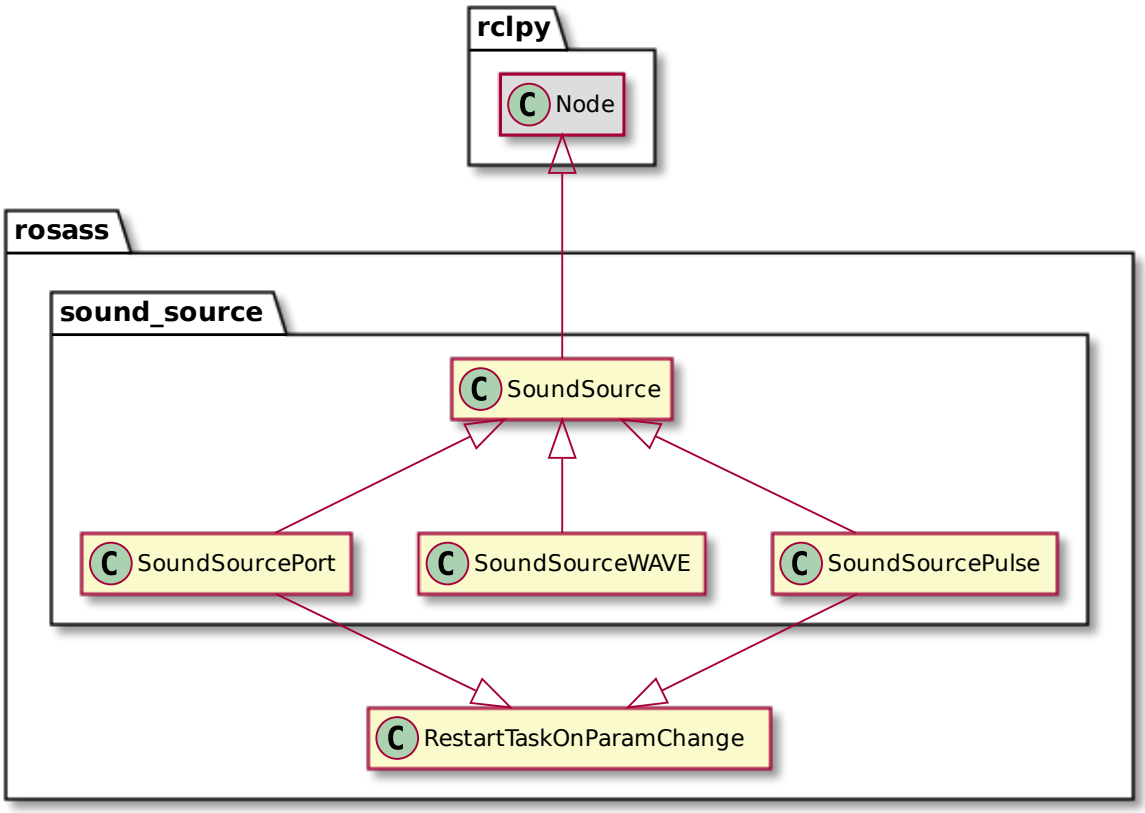

Abbildung 5.2: Vererbungshierarchie von **`SoundSource`**

Die Klasse **`Node`** der *ROS2*-Client-Library für *Python* (**`rclpy`**) beschreibt die Eigenschaften eines *ROS2*-Knotens. **`SoundSource`** erweitert diese, indem ein Nachrichtenkanal zum Senden von Audiodaten geöffnet wird und folgende Parameter deklariert werden:

- Der Parameter **`sample_rate`** in der Einheit Hertz ist zum Einstellen der gewünschten Abtastrate des Audiogerätes gedacht.

- Die Puffergröße kann mit dem Parameter **buffer_size** definiert werden. Damit eine Live-Verarbeitung stattfinden kann, müssen die Audiodaten segmentweise (Puffer) abgefragt und einzeln an die nächsten Knoten versendet werden. Bei großen Puffergrößen wird mehr Speicher benötigt und es entsteht eine höhere Latenz. Wählt man die Puffergröße jedoch zu gering, kann es zu Aussetzern kommen, weil dadurch ein erhöhter Kommunikationsaufwand entsteht. Außerdem ist das Zeitfenster bei der Verarbeitung eines Puffers geringer, wodurch es leicht passieren kann, dass eine kleine Verzögerung dafür sorgt, dass am Ausgabegerät Lücken im Datenstrom entstehen welche, hörbar sind. Ein Wert von 256 Samples pro Kanal (Standardeinstellung) funktioniert in vielen Fällen recht gut. Bei einem Mikrofon-Array mit vier Kanälen bedeutet dies z. B. dass ein Puffer $256 \cdot 4 = 1024$ Messwerte enthält.
- Für das Datenformat der Messwerte wurde der Parameter **sample_format** deklariert. Die Voreinstellung mit der Zeichenkette `'float32'` weist auf eine 32 bit Gleitkommazahl (einfache Genauigkeit) hin. Andere Formate werden zu diesem Zeitpunkt nicht von *RoSaSS* unterstützt. Denn bei manchen Verarbeitungsschritten sind aufgrund der verwendeten Technologien nur bestimmte Formate nutzbar. Das ständige Umwandeln zwischen den Formaten würde ein Informationsverlust bedeuten. Es wurde `'float32'` für die wohl ausreichende Präzision und Kompatibilität festgelegt.
- Abschließend existiert der Parameter **channel_selection** für die Kanalauswahl. So kann beispielsweise eine Untermenge der zur Verfügung stehenden Kanäle für das Weiterversenden selektiert werden. Als Wert muss eine Sequenz von Kanalindizes angegeben werden. `[1, 0]` bedeutet, dass an erster Stelle der zweite Kanal (Index 1) und danach der erste Kanal (Index 0) folgt. Auf diese Weise lässt sich eine beliebige Reihenfolge angeben.

Die Klassen **SoundSourcePort** und **SoundSourcePulse** erben zusätzlich auch von der Klasse **RestartTaskOnParamChange**, welche die Verwaltungslogik eines Hintergrundvorgangs abstrahiert, der nach Änderungen ausgewählter Parameter neu initialisiert werden muss. Nötig ist dies wegen der Interaktion mit dem Audiogerät.

Die gerade vorgestellten Parameter müssen nicht für jeden Knoten, welcher danach in der Verarbeitungskette folgt, einzeln konfiguriert werden. Denn alle notwendigen Informationen werden zusammen mit den Audiodaten übertragen und können bei den verarbeitenden Knoten berücksichtigt werden. Auf diese Weise verbreitet sich eine Parameteränderung an der Quelle (z. B. die Abtastrate) automatisch im restlichen System. Die konkrete Beschreibung des dafür verwendeten Nachrichtentyps **AudioBuffer** zeigt Abbildung 5.3. Das Feld **data** beinhaltet dabei die Audiodaten in einem Array mit Elementen des Typs **float32**, welcher nochmal mit der Konstanten **SAMPLE_FORMAT** dokumentiert wird. Mit dem Feld **sample_rate** wird die Abtastrate definiert und mit **channels** die Anzahl der Kanäle. Die Puffergröße ergibt sich aus der Länge von **data** dividiert durch **channels**. Denn **data** enthält die Messwerte aller Kanäle in abwechselnder Reihenfolge (siehe Abbildung 5.4 als Beispiel für vier Kanäle). Diese verschränkte Anordnung (Interleaving) ist üblich bei der Kommunikation mit Audiogräten und wird auch im *WAVE*-Dateiformat so umgesetzt. Zu guter Letzt eröffnet das Feld **is_last_buffer** mit dem standardmäßigen booleschen Wert **False** die Möglichkeit, einen Puffer als letzten in einer Abfolge zu markieren. Dadurch kann beispielsweise beim Abspielen einer Audiodatei am anderen Ende der Verarbeitungskette festgestellt werden, wann alle Daten dieser Datei übertragen wurden. Angewendet

rosass_msgs/msg/AudioBuffer.msg

```
# number of channels
uint8 channels

# sample rate in Hz
uint32 sample_rate

# sample format (32-bit float is currently supported)
string SAMPLE_FORMAT='float32'

# a flag to mark the last buffer in a sequence
bool is_last_buffer False

# audio data as interleaved array
float32[] data
```

Abbildung 5.3: Definition des Nachrichtentyps **AudioBuffer**

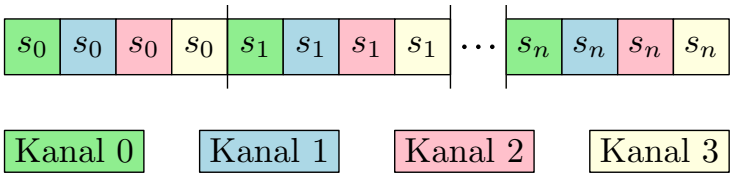

Abbildung 5.4: Anordnung der Samples s_0 bis s_n bei vier Kanälen

wird dieses Vorgehen bei der Evaluation mittels der *ASR*-Engines, welche nach Verarbeitung des jeweils letzten Puffers den Ergebnistext veröffentlichen (siehe Abschnitt 6.4).

Die Vererbungshierarchie bei **SoundSink** für den schreibenden Audiozugriff ist analog zu der in Abbildung 5.2. Als gemeinsamer Parameter existiert jedoch nur **channel_selection**.

5.3.1 WAVE

Zum Lesen und Schreiben von *WAVE*-Dateien wurde ursprünglich das in *Python* integrierte Modul *wave* vorgesehen. Jedoch stellte sich heraus, dass das Gleitkommaformat nicht unterstützt und stattdessen als eine Folge von Integer-Werten in 32 bit interpretiert wird. Mögliche Alternativen sind die Softwarebibliotheken *SciPy*[30] und *wavy*[31]. *SciPy* ist eine sehr umfangreiche Bibliothek, welche auf *NumPy* aufsetzt und viele Funktionen für wissenschaftliche Berechnungen bereitstellt, die zum Teil auch relevant für andere Schritte in *RoSaSS* sind. Trotzdem wird die Verwendung vorerst nicht geplant. Aus diesem Grund erscheint die Nutzung einer so umfangreichen Bibliothek übertrieben, da nur eine geringe Teilmenge der Funktionalität tatsächlich benötigt wird (Handhabung von *WAVE*-Dateien). Um den Fußabdruck von *RoSaSS* bezüglich der vorausgesetzten Abhängigkeiten nicht unnötig zu vergrößern, fiel die Entscheidung auf die schlankere *wavy*-Bibliothek.

[30] *SciPy*: `https://scipy.org/scipylib/` (besucht am 24.01.2020)
[31] *wavy*: `https://github.com/acelletti/wavy` (besucht am 24.01.2020)

Die Klasse **SoundSourceWAVE** verwendet diese zum Lesen der Audiodaten. Dabei ist zu beachten, dass die offizielle Version von *wavy* leider Fehler enthält, die das Lesen von den hier benutzten *WAVE*-Dateien mit einer anderen Kanalanzahl als zwei unmöglich macht. Als Umgehung dieses Problems ist stattdessen ein Fork[32] (Abzweig der Entwicklung vom Original) zu verwenden, bei welchem dieser Fehler behoben wurde. Als Parameter kennt **SoundSourceWAVE** die Abtastrate **sample_rate** nicht, da diese von der abzuspielenden Datei vorgegeben wird. Es werden zusätzlich zwei neue Parameter deklariert. Steht **all_channels** auf **True**, wird die manuelle Kanalauswahl **channel_selection** ignoriert und es werden alle Kanäle der Datei übernommen. Mit **directory** lässt sich das Verzeichnis definieren, in welchem die wiederzugebenden Audiodateien gesucht werden.

Das Abspielen also das Lesen des Signals vom Dateisystem und Entsenden in *RoSaSS* wird durch das Aufrufen des Services **StartPlayback** (Abbildung 5.5) ausgelöst. Ein *ROS2*-Service besteht aus einer Anfrage und einer Antwort und ist im Gegensatz zu einfachen Nachrichten nicht für die Übertragung von Nutzdaten, sondern die Übermittlung von Aufgaben oder Befehlen gedacht. Die Service-Beschreibung erfolgt mit der gleichen Syntax, wie die der Nachrichtentypen. Der einzige Unterschied besteht in der Trennung von Anfrage- und Antwortnachricht durch eine Zeile mit drei Strichen wie es in Zeile 10 zu sehen ist. Mit **file_name** wird der Dateiname bzw. Pfad relativ zum vorher gesetzten Parameter **directory**, welcher mit dem gleichnamigen Feld der Anfrage überschrieben werden kann, definiert. Das Abspielen von Dateien kann auf zwei Arten erfolgen. Entweder werden die Audiopuffer in einem Intervall veröffentlicht, welches der Originalwiedergabegeschwindigkeit (Realzeit) entspricht oder aber die Daten werden so schnell wie möglich versendet. Dieser Modus lässt sich mit dem Feld **realtime_mode** konfigurieren. Wenn die Audiodaten nicht live angehört werden müssen, empfiehlt es sich, diesen Wert auf **False** zu setzen, da so ein erheblicher Geschwindigkeitsgewinn bei der Verarbeitung entsteht. In diesem Fall wird der Parameter **buffer_size** ignoriert und die gesamten Audiodaten der Datei in einer einzigen Nachricht versendet. Auf diese Weise wird der Kommunikationsaufwand gesenkt, auch wenn im Hintergrund trotzdem eine Fragmentierung der Nachricht stattfindet. Besonders wichtig ist hierbei das Setzen der richtigen *QoS*-Einstellungen im Quellcode, damit keine dieser **AudioBuffer**-Nachrichten verloren geht. Denn mit den Standardeinstellungen kann es bei hoher Netzwerkauslastung bzw. zu geringem Sendeintervall (spielt bei einzelner Nachricht alleine keine Rolle), zu verworfenen Paketen kommen. Besonders gute Laufzeitverbesserungen lassen sich erreichen, indem eine andere *RMW*-Implementierung verwendet wird. Entgegen den in *ROS2* mitgelieferten *Fast RTPS*[33] kann beispielsweise *OpenSplice*[34] die Kommunikation zwischen auf dem selben Rechner ausgeführten Knoten beschleunigen, indem gar keine direkte Übertragung stattfindet. Möglich wird dies durch einen geteilten Speicherbereich, wo die Nachricht vom Sender geschrieben und von den Empfängern gelesen werden können. Sie muss zwar trotzdem noch von einer und in eine *ROS2*-Nachricht umgewandelt werden, wodurch jeder Knoten erst eine Kopie erzeugen muss, jedoch wird der Geschwindigkeitsgewinn bei bestimmten Anwendungsszenarien sehr deutlich. Um *OpenSplice* nutzen zu können, muss es zunächst für *ROS2* installiert werden. Anschließend muss der Workspace neu gebaut werden. Setzt man vor dem Starten der Knoten die Umgebungsvariable **RMW_IMPLEMENTATION**

[32] *wavy-Fork*: `https://github.com/GeneKong/wavy` (besucht am 24.01.2020)
[33] *Fast RTPS*: `https://github.com/eProsima/Fast-RTPS` (besucht am 25.01.2020)
[34] *OpenSplice*: `https://github.com/ADLINK-IST/opensplice` (besucht am 25.01.2020)

rosass_msgs/srv/StartPlayback.srv

```
# file name of the file to play
string file_name

# directory path
# leave it empty for default path
string directory ''

bool realtime_mode True

---

# True if playback started, False otherwise
# (due to an already running playback or read error)
bool success
```

Abbildung 5.5: Definition des Service-Typs **StartPlayback**

auf den Wert **rmw_opensplice_cpp**, ist die Umstellung vollzogen [Rob19]. Am Quellcode der Knoten muss nichts geändert werden.

Die Antwort dieses Services signalisiert mit dem Feld **success**, ob die Wiedergabe erfolgreich gestartet werden konnte. Soll diese frühzeitig abgebrochen werden, kann dies mit Aufruf des Services **StopPlayback** geschehen. Damit andere Knoten den Wiedergabestatus einsehen können, werden mit Nachrichten des Typs **TaskState** Zustandswechsel signalisiert.

Da *wavy* die Audiodateien nur lesen kann, wurde der Schreibvorgang von *WAVE*-Dateien für das hier benutzte Format in der Klasse **SoundSinkWAVE** selbst implementiert. Dabei wurde [Kab17] zur Hilfe genommen. Diese Dateien bestehen aus einem mehrstufigen Header, der am Anfang der Datei Informationen über den Inhalt, Format und Größe beschreibt. Nach dem generierten Header folgen einfach die Audiodaten Sample für Sample aneinandergereiht. Die so erzeugten Dateien lassen sich problemlos mit anderer Audiosoftware abspielen sowie auch von **SoundSourceWAVE** einlesen. Das erneute Abspeichern durch einen daran verbundenen **SoundSinkWAVE**-Knoten führt zu einer bitidentischen Datei, womit die verlustfreie Übertragung und Speicherung nachgewiesen ist. Starten lässt sich die Aufzeichnung mit dem Service-Typ **StartRecording**, welcher **StartPlayback** ähnelt. Wenn kein Dateiname angegeben ist, wird dieser automatisch aus dem Datum und der Uhrzeit generiert. Der verwendete Dateiname ist in der Antwort enthalten. Außerdem lässt sich eine Maximaldauer der Aufnahme definieren, falls diese nicht durch Aufruf des Services **StopRecording** beendet wird. Auch hier signalisiert eine *Topic* mit dem Nachrichtentyp **TaskState** die Zustandswechsel. Genau wie bei der Klasse **SoundSourceWAVE** hat **SoundSinkWAVE** den Parameter **all_channels** zum Aufzeichnen aller Kanäle. Während der Aufnahme darf sich die aufgenommene Kanalanzahl und Abtastrate nicht ändern, da diese im *WAVE*-Format statisch sind.

5.3.2 PortAudio

Für den Audiozugriff mittels der *PortAudio*-Schnittstelle wurde die *Python*-Bibliothek *SoundDevice* verwendet. Sowohl der Knoten für die Quelle als auch der Knoten für die Senke erweitert die Parameterliste um den Parameter **device**. Dieser gibt den Namen des Audiogerätes an, welches verwendet werden soll. Eine Liste der verfügbaren Geräte kann mit dem Befehl (Zeile 1) in Abbildung 5.6 abgerufen werden. Der Stern (*) markiert dabei die Standardauswahl.

```
1 $ python3 -m sounddevice
2   0 HDA Intel:ALC1220 Analog (hw:0,0), ALSA (2 in, 2 out)
3   1 HDA Intel:ALC1220 Digital (hw:0,1), ALSA (0 in, 2 out)
4   2 sysdefault, ALSA (128 in, 128 out)
5 * 3 default, ALSA (32 in, 32 out)
```

Abbildung 5.6: Beispielliste der verfügbaren Audiogeräte bei *PortAudio*

Immer wenn Parameter an einem der Knoten geändert werden, kommt es zu einer Validierung, ob diese kompatibel mit dem angegebenen Audiogerät sind. Ist dies nicht der Fall, wird die Änderung verworfen und der Parameter behält seinen alten Wert. Beispielsweise könnten die ausgewählten Kanäle nicht auf einem Gerät verfügbar sein. Oder die eingestellte Abtastrate wird nicht von der Hardware unterstützt. Bei **SoundSinkPort** lässt sich darüber hinaus noch eine Latenz bei der Wiedergabe auf der Hardware spezifizieren, um Aussetzern entgegen zu wirken.

5.3.3 PulseAudio

Genau wie bei den Knoten für den Audiozugriff mittels *PortAudio* sind die Knotenklassen **SoundSourcePulse** und **SoundSinkPulse** mit dem Gerätenamen (**device**) parametrisiert. Dieser lässt sich mit dem Programm *pactl* anzeigen. Abbildung 5.7 zeigt die entsprechenden Befehle zum Auflisten der Quellen (Zeile 1) und Senken (Zeile 4). Bei der ersten Zeichenfolge nach der Nummer handelt es sich um den zu verwendenden Namen.

```
1 $ pactl list sources short
2 0  alsa_output.pci-0000_00_1f.3.analog-stereo.monitor
    ▶ module-alsa-card.c  s32le 2ch 48000Hz  SUSPENDED
3 1  alsa_input.pci-0000_00_1f.3.analog-stereo  module-alsa-
    ▶ card.c  s32le 2ch 48000Hz  SUSPENDED
4 $ pactl list sinks short
5 0  alsa_output.pci-0000_00_1f.3.analog-stereo  module-alsa
    ▶ -card.c  s32le 2ch 48000Hz  SUSPENDED
```

Abbildung 5.7: Beispielliste der verfügbaren Audiogeräte bei *PulseAudio*

Für die Interaktion mit *PulseAudio* wurde eine *Python*-Bibliothek namens *libpulseaudio3*[35] benutzt. Das ist eine *ctypes*[36]-Schnittstelle, welche die Nutzung der in der Programmiersprache *C* entwickelten *PulseAudio*-Bibliothek von *Python* heraus ermöglicht. Diese bildet die mächtige *Asynchronous API* ab, mit der man ereignisgesteuert programmiert, um sich mit dem *PulseAudio*-Server zu verbinden und den Audiostream einzurichten [Pul19]. Anders als bei *PortAudio* ermöglicht *PulseAudio* das Konfigurieren von beliebigen Abtastraten, auch wenn diese nicht von der Hardware unterstützt werden. Dabei wird eine Abtastratenkonvertierung (Resampling) durchgeführt, bevor die Daten an den Client also in diesem Fall `SoundSourcePulse` oder `SoundSinkPulse` übertragen werden.

Um diese Knoten innerhalb einer Container-Virtualisierung erfolgreich ausführen zu können, müssen erst Vorkehrungen getroffen werden, damit diese sich direkt mit dem laufenden *PulseAudio*-Server des Wirtssystems verbinden [mvi19]. Der erste Schritt besteht darin, den Server dazu zu bringen, einen *Unix Domain Socket* für die Kommunikation zu öffnen. Der dafür notwendige Kommandozeilenbefehl (Abbildung 5.8 oben) muss von einer Shell im Wirtssystem ausgeführt werden. Als Resultat wird ein Modul im Server geladen, welches diesen Socket unter dem angegebenen Pfad erstellt. Außerdem muss im Container unter `/etc/pulse/client.conf` eine entsprechende Konfigurationsdatei vorliegen (Abbildung 5.8 unten). Sie verhindert das automatische Starten eines *PulseAudio*-Servers innerhalb des Containers und gibt den Pfad zum Socket an. Dieser muss allerdings zusätzlich durch die Umgebungsvariable `PULSE_SERVER` mitgegeben werden. Darüber hinaus ist der Pfad zu einer Cookie-Datei durch die Variable `PULSE_COOKIE` anzugeben, welche automatisch, wenn nicht anders spezifiziert, im selben Verzeichnis wie der Socket erstellt wird. Nach diesen Einstellungen können nun Programme des Gastsystems mit den *PulseAudio*-Server des Wirtssystems uneingeschränkt interagieren.

```
$ pactl load-module module-native-protocol-unix socket=/
   ► tmp/pulseaudio.socket
```

/etc/pulse/client.conf

```
default-server = unix:/tmp/pulseaudio.socket
# prevent a server running in the container
autospawn = no
daemon-binary = /bin/true
# prevent the use of shared memory
enable-shm = false
```

Abbildung 5.8: Einrichtung von *PulseAudio* für eine Container-Umgebung

Beim lesenden Audiozugriff auf dem *Pepper*-Roboter ist zu beachten, dass die Puffergröße `buffer_size` auf mindestens etwa 4000 gestellt werden sollte, da sonst der *ROS2*-Knoten der Klasse `SoundSourcePulse` nur Bruchstücke der Daten vom *PulseAudio*-Server bekommt. Vermutlich ist das Intervall, um die Audiopuffer bereitzustellen, zu kurz für den Server bei dieser Konstellation. Für die Kommunikation innerhalb von *RoSaSS* und auch bei der Audioausgabe tritt dieses Phänomen sogar bei einer wesentlich kleineren Puffergröße von 256 nicht auf.

[35] *libpulseaudio3*: `https://github.com/skakri/python3-pulseaudio/` (besucht am 25.01.2020)
[36] *ctypes*: `https://docs.python.org/3/library/ctypes.html` (besucht am 25.01.2020)

Damit die Konfiguration zum Vorschalten von *RoSaSS* vor *NAOqi* wie im Konzept in Abbildung 4.2 nicht manuell erbracht werden muss, wurde dies auch implementiert. Diese Funktionalität ist in einem gesonderten *ROS2*-Knoten ausgelagert, da die eben vorgestellten Knoten allein für den Transport der Audiodaten zuständig sind und sein sollen. Die Klasse `PulseVirtualSink` erledigt dies mit Hilfe der Bibliothek *pulsectl*[37] und handhabt selbst keine Audiodaten. Mit *libpulseaudio3* wäre es auch möglich gewesen, jedoch ist *pulsectl* genau für solche Aufgaben zugeschnitten und in dieser Hinsicht wesentlich einfacher zu benutzen. Beim Starten des Knotens wird die gewünschte Nullsenke am *PulseAudio*-Server erstellt und eingerichtet. Wird nun der Knoten beendet, dann werden diese Schritte wieder rückgängig gemacht. Das bedeutet, dass diese Nullsenke nur zur Laufzeit des Knotens existiert. So wird *PulseAudio* in dem Zustand verlassen, wie es vorgefunden wurde, um ein mehrfaches und unnötiges Erzeugen von Nullsenken zu verhindern. Der Knoten hat eine Reihe von Parametern. Mit `sink_name` und `channels` lässt sich der gewünschte Name des virtuellen Audiogerätes angeben und die Anzahl der Kanäle definieren. Der boolesche Parameter `take_over_streams` aktiviert bei Bedarf eine Funktion, welche automatisch alle existierenden Streams von einer bestimmten Quelle (`original_source_name`) an sich bindet. Auf diese Weise lässt sich z. B. ein Stream von *NAOqi*, welcher mit den Mikrofonen verbunden ist, umlenken auf den Monitor der Nullsenke. Bei der Nutzung der Spracherkennung in *NAOqi* wird allerdings ständig ein zusätzlicher Stream angelegt und wieder geschlossen, welcher dann sich dieser Übernahme entzieht. Aus diesem Grund kann mit dem Parameter `set_default` und `set_monitor_default` die Nullsenke bzw. dessen Monitor als Standardgerät eingestellt werden. Das führt dazu, dass neu erstellte Streams sich mit diesem verbinden, falls nicht explizit ein Gerätename angegeben wird, was bei *NAOqi* zutrifft.

5.4 Signalverarbeitung

Alle verarbeitenden Knoten erben von der Klasse `SoundProcessor`. Dadurch werden automatisch *Topics* für das Empfangen und Senden von Audiopuffern eingerichtet sowie zwei Parameter deklariert. Neben dem bereits bekannten Parameter `channel_selection` gibt es die boolesche Konfigurationsmöglichkeit `bypass`. Mit dem Wert `True` wird die in den Subklassen implementierte Verarbeitung übersprungen und der eingehende Audiopuffer direkt weiter gesendet. Auf diese Weise lassen sich so einzelne Verarbeitungsschritte (Knoten) bei Bedarf zur Laufzeit überbrücken und wieder einbinden, ohne dabei die Kommunikationstopologie in *RoSaSS* ändern zu müssen.

Die Signalverarbeitungsalgorithmen zur eigentlichen Aufbereitung der Audiodaten werden in Kapitel 7 detailliert beschrieben. Trotzdem gibt es darüber hinaus weitere damit verbundene Aspekte. Zum einen wurde ein Knoten zum Laden und Anwenden von Audio-Plug-ins geschaffen, damit *RoSaSS* von bereits existierenden Implementierungen profitieren kann. Zum anderen ist für die korrekte Nutzung der *ASR*-Engines eine Abtastratenkonvertierung notwendig. Denn die meisten Spracherkennungsprogramme unterstützen nur eine feste Abtastrate von 16 kHz. Sicherlich kann zu diesem Zweck beispielsweise von *PulseAudio* ein Stream abgerufen werden, welcher diese Anforderung erfüllt. Jedoch soll die Möglichkeit nicht ausgeschlossen werden, die Vorverarbeitung mit einer anderen Abtastrate durchzuführen und anschließend das Ergebnis an die *ASR*-Engine weiterzuleiten.

[37] *pulsectl*: `https://github.com/mk-fg/python-pulse-control/` (besucht am 26.01.2020)

5.4.1 Plug-in-Schnittstelle

Um Audio-Plug-ins nutzen zu können, muss ein sogenannter Plug-in-Host eingebunden werden. Zu diesem Zweck gibt es die Softwarebibliothek *lilv*[38] für den Plug-in-Standard *LV2*, welche auch in *Python* genutzt werden kann. Sind Plug-ins auf einem Rechner installiert, liegen sie abhängig vom Betriebssystem in bestimmten Verzeichnissen (z. B. in `/usr/lib/lv2/`). Der Knoten der Klasse `SoundProcessorLV2` scannt durch die Nutzung von *lilv* diese Verzeichnisse ab und verschafft sich einen Überblick über die zur Verfügung stehenden Plug-ins. Dabei trägt jedes Plug-in einen *Uniform Resource Identifier (URI)* als identifizierendes Merkmal, welcher beispielsweise den eindeutigen Wert `http://lv2plug.in/plugins/eg-amp` tragen kann. In diesem Fall handelt es sich um ein sehr einfaches Beispiel-Plug-in (Example Amplifier) zum Verändern der Lautstärke, welches für gewöhnlich vorinstalliert ist. Idealerweise entspricht der *URI*-Wert einer gültigen Web-Adresse, wo weitere Informationen über dieses Plug-in zu finden sind. Diese Konvention ist jedoch nicht zwingend, wie dieses Beispiel zeigt. Über den *ROS2*-Parameter `plugin_uri` kann das gewünschte Plug-in durch eine Zeichenkette angegeben werden.

Nach dem Laden folgt die Initialisierung der Schnittstellen. Sie werden auch als Ports bezeichnet. Ein Plug-in kann eine beliebige Anzahl an eingehenden und ausgehenden Ports haben, welche zusätzlich in Audiodaten und Steuerungsinformationen (Parameter) zu unterscheiden sind. Bei der Initialisierung wird zunächst für jeden Port ein Speicherbereich allokiert, welcher dann dem Plug-in mitgeteilt wird. So können diese Bereiche mit Daten befüllt werden, die beim Anwenden des Plug-ins als Eingangsdaten vorliegen. Anschließend sind die Ergebnisse im Speicher der ausgehenden Ports zu finden. Mit *lilv* funktioniert dies in *Python* so, dass für jeden Port ein *NumPy*-Array angelegt wird, dessen Elemente mit neuen Daten überschrieben werden können. Nach diesem Schema werden die `AudioBuffer`-Nachrichten zu und von den Audio-Ports übertragen. Der *ROS2*-Knoten legt darüber hinaus für jeden eingehenden Port für Steuerungsinformationen einen Parameter an, damit diese zur Laufzeit beeinflussbar sind. Genau genommen werden auf Basis der Metadaten noch weitere aber schreibgeschützte Parameter definiert, um diese Informationen dem restlichen System verfügbar zu machen. Es geht hierbei um den Standardwert sowie um den Wertebereich in Form von Minimum und Maximum. Das Namensschema wird in Abbildung 5.9 deutlich. Mit dem Befehl `ros2 param list` lassen sich alle Parameternamen oder nur die des spezifizierten Knotennamens auflisten. Dabei werden alle zum Plug-in gehörigen Parameter mit dem Prefix `plugin.` versehen. Die Metadatennamen reichen durch den Postfix `.default`, `.minimum` und `.maximum` eine Hierarchieebene weiter. Um beim oben genannten Beispiel-Plug-in zu bleiben, wurde hier der Name `gain` des einzigen Steuerungs-Ports gefunden und als *ROS2*-Parameter verfügbar gemacht. Der in Abbildung 5.9 gelistete Parameter `use_sim_time` kann hierbei ignoriert werden, da er momentan in *RoSaSS* nicht verwendet wird. Er existiert automatisch in jeder Knotenklasse und ist dafür gedacht den Knoten mitzuteilen, ob die Systemzeit als Zeitreferenz genutzt werden soll oder aber eine alternative Uhr. Benötigt wird dies vor allem in Simulationen, welche in verschiedenen Geschwindigkeiten ablaufen bzw. sogar pausiert werden können, ohne dabei Zeitmessungen innerhalb der Knotenimplementierungen zu verfälschen.

[38] *lilv*: `http://drobilla.net/software/lilv` (besucht am 27.01.2020)

```
$ ros2 param list /sound_processor_lv2
  bypass
  channel_selection
  plugin.gain
  plugin.gain.default
  plugin.gain.maximum
  plugin.gain.minimum
  plugin_uri
  use_sim_time
```

Abbildung 5.9: Liste der Parameternamen des Knotens **/sound_processor_lv2** mit initialisiertem Plug-in des *URI* **http://lv2plug.in/plugins/eg-amp**

5.4.2 Abtastratenkonvertierung

Für das Resampling wurde in der Klasse **SoundProcessorResample** die *Python*-Bibliothek *samplerate*[39] genutzt. Sie verwendet im Hintergrund die auch als *Secret Rabbit Code* bezeichnete Bibliothek *libsamplerate*[40] und zeichnet sich besonders durch eine hochqualitative und verzerrungsarme Abtastratenkonvertierung aus. Parametrisiert ist dieser Knoten durch **sample_rate**, womit sich die Abtastrate des ausgehenden Signals angeben lässt, nach welcher die eingehenden Daten konvertiert werden sollen.

5.5 ASR

Die Anbindungen von verschiedenen *ASR*-Engines sind in je einer Knotenklasse implementiert. Sie erben alle von der Klasse **ASR**, welche die Start- und Stopp-Logik sowie die Handhabung von Service-Anfragen bereitstellt. Ähnlich wie schon beim Abspielen oder Aufnehmen von Audiodateien (siehe Unterabschnitt 5.3.1) gibt es auch hier Services zum Starten und Stoppen der Spracherkennung. Der Service **StartSTT** startet den *Speech-To-Text (STT)*-Vorgang. Das optionale boolesche Feld **finish_on_last** sorgt für ein automatisches Stoppen, sobald ein mit **is_last_buffer** markierter Audiopuffer verarbeitet wurde. Während des *STT*-Vorgangs werden Zwischenergebnisse in Textform mit Nachrichten vom Typ **Text** in einem Intervall veröffentlicht, welches durch den Wert des optionalen Feldes **progress_report_interval** definiert ist. Diese Funktion kann für die Interaktion mit dem Nutzer hilfreich sein, wenn der bereits erkannte Text während des Sprechens beispielsweise auf einem Bildschirm präsentiert wird. Bei einem Intervall von Null findet das Senden von Zwischenergebnissen nicht statt. Der Service **StopSTT** beendet die Spracherkennung manuell und liefert das fertige Ergebnis zurück. Damit der Text auch bei einem automatischen Beenden verfügbar ist, wurde zu diesem Zweck der Service **GetHypothesis** zum direkten Abfragen dieser von der *ASR*-Engine aufgestellten Hypothese geschaffen. Auch hier werden mit dem Nachrichtentyp **TaskState** Zustandswechsel beim Starten bzw. Stoppen bekannt gegeben.

[39] *samplerate*: `https://github.com/tuxu/python-samplerate/` (besucht am 27.01.2020)
[40] *libsamplerate*: `http://www.mega-nerd.com/libsamplerate/index.html` (besucht am 27.01.2020)

Implementiert wurde all dies unter häufiger Verwendung von Schablonenmethoden. Das ist ein Verhaltensmuster, welches auch bei einigen anderen bereits beschriebenen Knotenimplementierungen zur Anwendung kommt. Es wird dabei in der Elternklasse das Grundgerüst programmiert, das Methoden verwendet, die ihre konkrete Implementierung erst in den Unterklassen erhalten [Gam+04, S. 366]. So müssen für jede *ASR*-Engine nur einzelne Methoden überschrieben werden, ohne dabei die Ablauflogik in jeder Knotenklasse programmiert zu haben. In diesen Methoden findet die Initialisierung des Decoders einschließlich dem Laden der Modelle, das Einfüttern der Audiodaten, das Starten sowie Stoppen eines Redeabschnittes und das Abfragen der Hypothese statt. Nach diesem Muster wurden die *ASR*-Engines *PocketSphinx*, *DeepSpeech* und *Kaldi* mit Hilfe der dazugehörigen *Python*-Bibliotheken eingebunden. Neben dem bereits in der Elternklasse **`ASR`** deklarierten und genutzten Parameter **`channel_selection`** erweitern die Unterklassen die Parameterliste um Dateipfade zu den zu verwendenden Modellen.

5.5.1 VoCon Hybrid

Besonderheiten gibt es bei der Einbindung der *VoCon Hybrid*-Engine, welche auf dem *Pepper*-Roboter nur über *NAOqi* nutzbar ist. Im Unterschied zu den anderen Knoten konsumiert **`ASRVoCon`** die eingehenden Audiodaten nicht, sondern sorgt nur für die Steuerung der Spracherkennung. Das Einspeisen der Audiodaten erfolgt über einen **`SoundSinkPulse`**-Knoten, welcher den Stream an die mit **`PulseVirtualSink`** eingerichtete Nullsenke weitergibt (siehe Abbildung 4.2).

Die Implementierung des *ROS2*-Knotens, der gleichzeitig auch Kontakt mit *NAOqi* aufbaut, erwies sich auf den ersten Blick als problematisch. Denn es liegt ein Versionskonflikt mit *Python* vor. Die Bibliotheken für die Kommunikation mit *NAOqi* sind nur kompatibel mit *Python* in der Version 2. *ROS2* baut wiederum ausschließlich auf *Python* 3, was auch zu begrüßen ist. *Python* 2 gilt als veraltet und sollte nicht mehr verwendet werden. Die letzte Aktualisierung war ursprünglich für 2015 geplant. Um mehr Zeit für den Umstieg auf *Python* 3 zu geben, wurde das Support-Ende auf April 2020 verschoben [Gre19]. Doch leider ist dieser Umstieg vom Hersteller des Roboters nicht durchgeführt worden.

Gelöst wurde das Problem durch eine Aufspaltung in zwei Komponenten, wobei der *ROS2*-Teil in einem *Python* 3-Interpreter läuft und der *NAOqi*-Teil in einem anderen *Python* 2-Prozess ausgeführt wird. Diese Komponenten brauchen dann einen eigenen Kommunikationskanal untereinander. Um nicht dafür den Verbindungsprozess, ein Protokoll und Serialisierungsvorgänge selbst entwickeln zu müssen, wurde auf *Python Remote Objects 4 (Pyro4)*[41] gesetzt. Bei dieser Technologie wird ein *Python*-Objekt mit ausgewählten Attributen und Methoden über eine Netzwerkverbindung verfügbar gemacht. Am anderen Ende kann dieses Objekt durch einen Proxy verwendet werden. Dabei handelt es sich um ein Stellvertreterobjekt, dessen Methodenaufrufe in Wirklichkeit auf der anderen Seite des Kanals stattfinden. Der dafür benötigte Datenaustausch wird automatisch im Hintergrund erledigt und muss nicht selbst explizit ausgelöst werden. Somit können diese Proxy-Objekte nach der Verbindungsherstellung einfach wie normale Objekte verwendet werden. Gegenüber vergleichbaren Frameworks hat *Pyro4* den Vorteil, kompatibel zwischen *Python* 2 und 3 zu sein. Für die Steuerung von *VoCon Hybrid* muss zunächst auf dem Roboter das Programm **`bin/naoqi_vocon.py`** aufgerufen werden (siehe Abbildung 5.10). Dieses

[41] *Pyro4*: `https://github.com/irmen/Pyro4` (besucht am 29.01.2020)

verbindet sich mit *NAOqi* und stellt Methoden zur Steuerung der Spracherkennung mit Hilfe von *Pyro4* bereit. Dabei fungiert dieses Programm als Server und legt in diesem Fall einen *Unix Domain Socket* an. Der *ROS2*-Knoten der Klasse **`ASRVoCon`** verbindet sich damit, erzeugt ein Proxy-Objekt und kann dies letztendlich zur Steuerung von *VoCon Hybrid* nutzen.

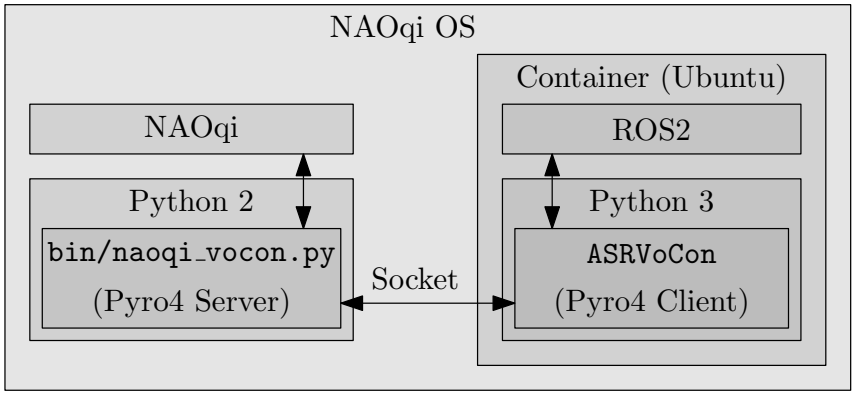

Abbildung 5.10: Steuerung der *VoCon Hybrid*-Engine

Parametrisiert ist dieser Knoten durch **`language_code`**, welcher zum Einstellen der Sprache mittels eines Sprachcodes als Zeichenkette geschaffen wurde. Beispielsweise sorgt der Wert `'de_DE'` für eine auf Deutsch stattfindende Spracherkennung. Voraussetzung ist ein passendes auf dem Roboter installiertes Sprachpaket. Für die Durchführung der Spracherkennung wird auf das Modul **`ALDialog`** zurückgegriffen. Denn dort kann eine spezielle Topic-Beschreibung (nicht zu verwechseln mit der *ROS2-Topic*) angegeben werden, wodurch ein freies Diktat möglich ist. Abbildung 5.11 zeigt deren Inhalt. In Zeile 1 wird der Name der Topic festgelegt, welcher frei gewählt werden kann. Die Sprache wird in Zeile 2 anhand des konfigurierten **`language_code`** gesetzt. Interessant wird es in der letzten Zeile. Wenn die Spracherkennung den Text in den runden Klammern versteht, wird der Text, welcher sich danach befindet, als Antwort mittels Sprachsynthese ausgegeben. In diesem Fall steht **`_*`** für einen beliebigen Text, welcher **`^empty`**, also Nichts, als Antwort bekommt. Leider kommt es immer wieder zu Unterbrechungen der Spracherkennung, die von der nicht umgehbaren *VAD* verursacht wird.

```
1 topic: ~dictation()
2 language: de_DE
3 u: (_*) ^empty
```

Abbildung 5.11: Topic-Definition beim *NAOqi*-Modul **`ALDialog`** für ein freies Diktat

5.5.2 Nutzung durch NAOqi

Um die in *RoSaSS* eingebundenen *ASR*-Engines auch von *NAOqi* heraus steuern zu können, wurde das Vorgehen mit *Pyro4* in umgekehrter Richtung angewendet. Diesmal übernimmt ein *ROS2*-Knoten (Klasse **`RoSaSSProxy`**) die Rolle des Servers. Das Programm **`bin/naoqi_rosass.py`** registriert sich als *NAOqi*-Modul, dessen Services (Methoden) von anderen *NAOqi*-kompatiblen Programmen aufgerufen werden können und so über *Pyro4* in *ROS2* zur Ausführung kommen.

5.6 Ausführung

Um die bereits vorgestellten Knoten ausführen zu können, muss der Workspace, in dem sie gebaut wurden, für die Sitzung aktiviert werden (Abbildung 5.12 Zeile 1). Das entsprechende Skript befindet sich dort im generierten Unterverzeichnis **`install`**. Es ist zu beachten, dass diese Schritte in einer anderen Konsole durchgeführt werden sollten als in der, wo der Bauprozess stattfand. Nun kann mit dem Befehl **`ros2 run`** und der Angabe des Paketnamens sowie des Programmnamens ein Knoten ausgeführt werden. In *RoSaSS* wurden die erzeugten Programme nach der entsprechenden Knotenklasse benannt. Zeile 2 zeigt dies am Beispiel von **`SoundSourcePulse`**. Mit diesem Befehl können beim Starten weitere Einstellungen getroffen werden, wie etwa das Umbenennen von *Topics* oder das Setzen von Parametern. Um nicht jeden Knoten einzeln in einer neuen Konsole starten zu müssen, können alternativ die Launch-Skripte genutzt werden. In Zeile 3 ist ein Beispiel zu sehen, welches zwei Knoten konfiguriert, miteinander verknüpft und startet. In diesem Fall wird das Audiosignal von einem Eingabegerät in das Standard-Ausgabegerät mit Hilfe von *PulseAudio* gegeben. Wichtig ist hierbei, dass das Ausgabegerät nicht zu laut eingestellt bzw. isoliert vom Eingabegerät sein sollte, da sonst eine unangenehme Rückkopplung entsteht.

```
1 $ source install/setup.bash
2 $ ros2 run rosass sound_source_pulse
3 $ ros2 launch rosass basic_wire.launch.py
```

Abbildung 5.12: Befehle zum Ausführen von Knoten

Der Code dieses Skriptes ist in Abbildung 5.13 zu sehen. Darin wird eine Beschreibung des Startvorgangs erzeugt, welche von **`ros2 launch`** abgearbeitet wird. Konkret werden hier die Knoten beschrieben. Durch das Feld **`parameters`** können die gewünschten Werte von Parametern aufgelistet werden. Bei diesem Beispiel werden beide Knoten für die Handhabung von Stereodaten eingerichtet. Alle weggelassenen Parameter tragen den im Quellcode der Knoten hinterlegten Standardwert. Mit **`remappings`** werden Namen von Nachrichtenkanälen oder Services umbenannt. Beispielsweise wird die im Quellcode festgelegte *Topic* `'/output/audio'` des Knotens **`sound_source_pulse`** nach `'microphone'` abgebildet. Geschieht dasselbe mit der eingehenden *Topic* bei der Senke, können diese Knoten miteinander kommunizieren. Abbildung 5.14 visualisiert die Knotenkonstellation in einem Komponentendiagramm. Die Rechtecke stellen die Knoten dar, welche einen Namen tragen und einen Typ (≪Klasse≫). Die Namen sind ebenfalls im Quellcode definiert und gelten als identifizierendes Merkmal. Es ist jedoch möglich, den Namen bei der Ausführung zu ändern. Sonst könnten nicht mehrere Knoten des gleichen Typs aktiv sein. Schnittstellen, in diesem Fall *Topics*, werden als Kreise dargestellt, welche auch einen Namen und Typ tragen. Die Pfeile geben die Übertragungsrichtung der Nachrichten an.

Andere Launch-Skripte sind weitaus umfangreicher gestrickt. Nicht nur, weil sie eine größere Anzahl an Knoten beschreiben, sondern auch, weil diese Skripte selbst parametrisiert sein können und auf Basis darauf verschiedene Konstellationen erzeugen und auch auf andere Skripte verweisen. Ein Beispiel ist **`live_asr.launch.py`**, welches anhand der Argumente **`device`** und **`asr_engine`** verschiedene Konfigurationen für Eingabegräte bzw. *ASR*-Engines definiert, die auf mehrere Skripte verteilt sind. Mit der Option **`-s`** (siehe

rosass/launch/examples/basic_wire.launch.py

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='rosass',
            node_executable='sound_source_pulse',
            parameters=[
                {'channel_selection': [0, 1]},
            ],
            remappings=[
                ('/output/audio', 'microphone')
            ]
        ),
        Node(
            package='rosass',
            node_executable='sound_sink_pulse',
            parameters=[
                {'channel_selection': [0, 1]},
            ],
            remappings=[
                ('/input/audio', 'microphone')
            ]
        ),
    ])
```

Abbildung 5.13: Launch-Skript **basic_wire.launch.py**

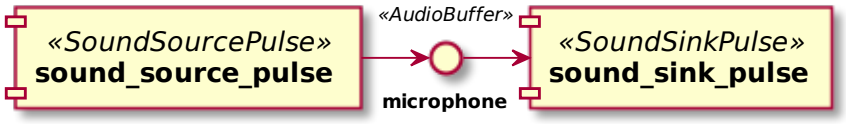

Abbildung 5.14: Komponentendiagramm von `basic_wire.launch.py`

```
$ ros2 launch -s rosass live_asr.launch.py
Arguments (pass arguments as '<name>:=<value>'):

    'device':
        the input device to use. choose from ('default' |
            ▶ 'pepper' | 'respeaker' | 'uma8')
        (default: 'default')

    'asr_engine':
        the asr engine to use. choose from ('kaldi' | '
            ▶ sphinx' | 'deepspeech' | 'vocon')
        (default: 'kaldi')
```

Abbildung 5.15: Befehl zum Inspizieren eines Launch-Skripts

Abbildung 5.15) wird eine Beschreibung der möglichen Argumente ausgegeben, ohne den Startprozess durchzuführen. Abbildung 5.16 zeigt das Resultat bei Standardwerten. Bereitgestellte Services werden hierbei als Schnittstellen ohne Pfeile dargestellt. In dieser Konstellation sind die Services nicht in Benutzung. Sie können jedoch manuell ausgeführt werden. Dies geht entweder mit dem Befehl `ros2 service` oder aber mit Hilfe des Programms *rqt*. Das ist eine graphische Software, mit welcher man unter anderem *ROS2*-Koten inspizieren und auch Services aufrufen kann.

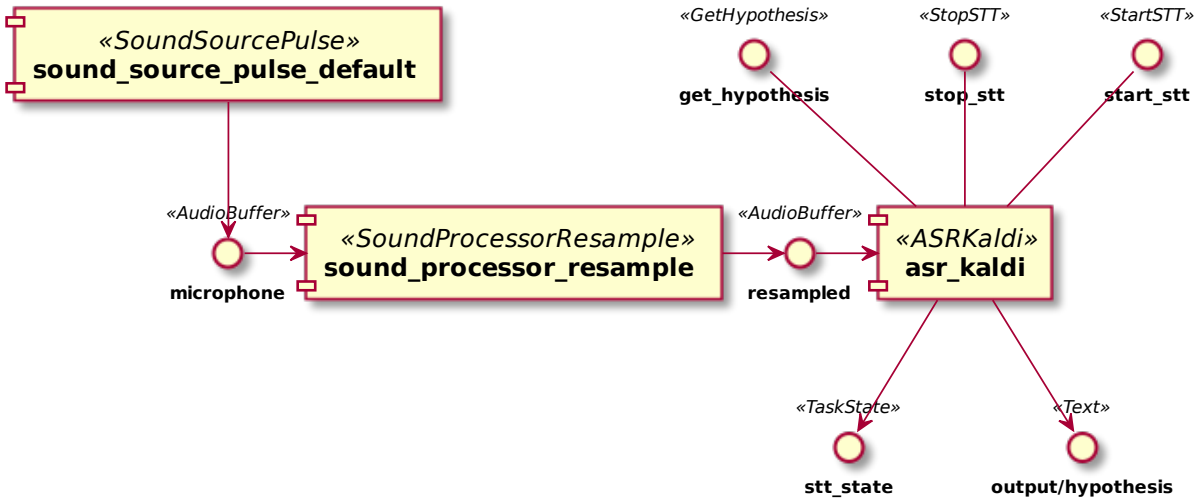

Abbildung 5.16: Komponentendiagramm von `live_asr.launch.py`

6 Testaufbau

Bevor genauer die Signalverarbeitung detailliert beschrieben wird, sollte eine Möglichkeit geschaffen werden, diese objektiv bewerten zu können, damit während der Entwicklung auf dieses Feedback reagiert werden kann. Das Ziel ist, dass durch diese Vorverarbeitung die *ASR*-Engines den gesprochenen Text besser verstehen, also bei der Transkription weniger Fehler machen. Um das nicht immer live am Roboter oder anderer Hardware ausprobieren zu müssen, wurde die nun folgende Vorgehensweise durchgeführt. Die persistenten Aufzeichnungen der Audiodaten von gesprochenen Texten lassen sich immer wieder bei der Bewertung heranziehen. Damit sie unter reproduzierbaren Bedingungen und in mehreren Durchläufen mit unterschiedlicher Hardware vergleichbare Ergebnisse liefern, scheidet ein direktes Vorsprechen durch Menschen aus, da so ein exakt wiederholbarer Sprechvorgang nicht möglich ist. Aus diesem Grund sollen bei der Aufzeichnung vorher aufgenommene Audiodaten durch einen Lautsprecher mit fest eingestellter Lautstärke wiedergegeben werden. Diese aufgezeichneten Testdaten können nun von den *ASR*-Engines transkribiert werden. Durch den Vergleich mit den Originaltexten ist so eine Bewertung möglich. Dieser Test verschafft einen Überblick, wie gut die Spracherkennung mit den aktuellen Bedingungen zurecht kommt. Durch das vorherige Anwenden der Signalverarbeitung auf jene Daten kann dadurch auch eine Verbesserung messbar gemacht werden.

6.1 Rohdatenbeschaffung

Als Ausgangsdatenquelle mit aufgenommenen Sprachabschnitten, bekannten Originaltexten und einer Vielfalt an verschiedenen Stimmen eignen sich besonders gut Sprachkorpora, welche auch für die Erstellung von Sprachmodellen für die *ASR*-Engines herangezogen werden. Die Wahl fiel auf die Daten des *Tuda-De*-Korpus, da dieser entgegen den meisten anderen frei verfügbaren deutschsprachigen Korpora unter kontrollierten und gleichbleibenden Bedingungen aufgezeichnet wurde. Nach dem Download[42] und dem Entpacken der Daten wird man mit Verzeichnissen von einigen tausend Dateien konfrontiert. Jede Aufnahme liegt in einer *WAVE*-Datei in 16 bit-Auflösung, mit 16 kHz-Abtastung und einem Kanal (mono) vor. Und das in mehrfacher Ausführung, da mit mehreren Geräten gleichzeitig aufgezeichnet wurde. Die Dateien sind nach Datum, Uhrzeit und Gerät benannt. Zu jeder aufgenommenen Sprachsequenz gibt es eine *eXtensible Markup Language (XML)*-Datei, welche genauere Informationen darüber enthält. Dazu zählen der Text, die Herkunft des Textes, die pseudonymisierte Identifikationsnummer des Sprechers, Altersklasse, Geschlecht, Bundesland und weitere. Da diese Metadaten in der vorliegenden Form sehr umständlich zu durchsuchen sind, wurde hierfür ein *Python*-Programm entwickelt,

[42] Die Daten des *Tuda-De*-Korpus sind unter `http://ltdata1.informatik.uni-hamburg.de/kaldi_tuda_de/german-speechdata-package-v3.tar.gz` verfügbar. (besucht am 31.01.2020)

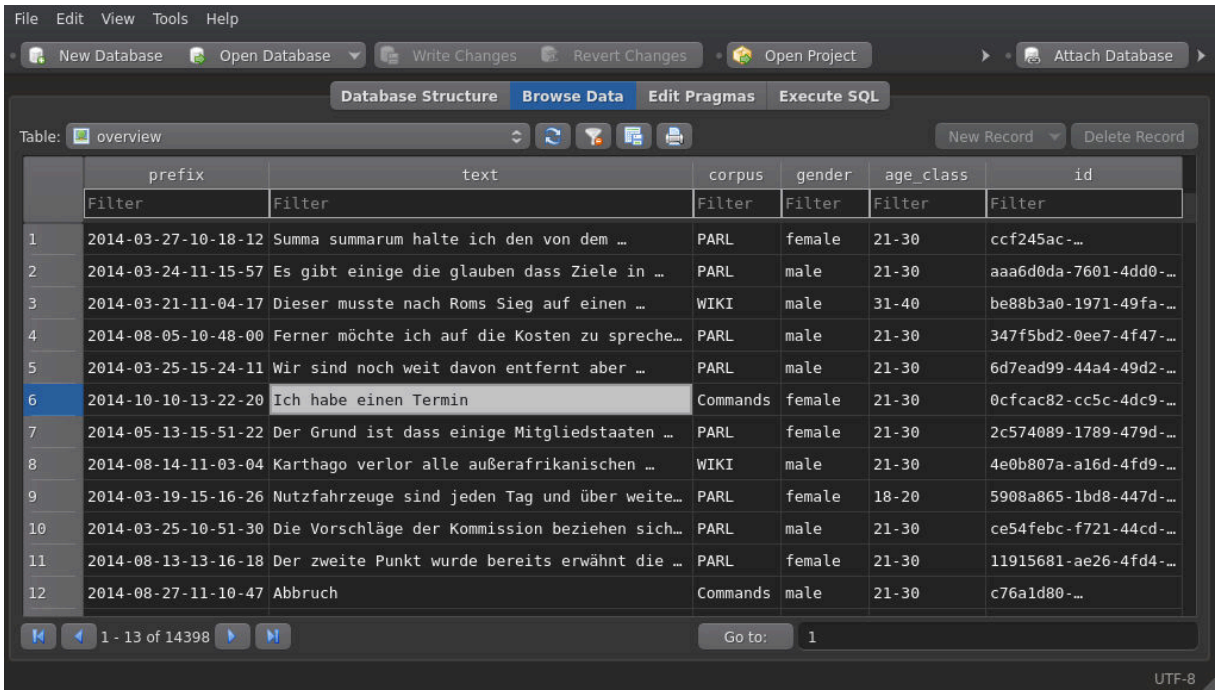

Abbildung 6.1: Ansicht der Daten im *DB Browser for SQLite*

welches unter Angabe des Verzeichnispfades die gesamten *XML*-Dateien einliest und deren Daten strukturiert in einer relationalen Datenbank ablegt. Das Programm und die resultierenden Datenbanken sind in der Beilage 1 enthalten. Als Datenbanksystem wurde *SQLite*[43] gewählt. Im Gegensatz zu anderen wird hier kein Datenbankserver benötigt, da *SQLite* genau genommen nur ein eigenständiges Dateiformat ist, welches von vielen Programmen unterstützt wird.

Dazu mussten keine externen Softwarebibliotheken installiert werden, da sowohl für das Parsen von *XML* als auch die Handhabung von *SQLite* Module in der Standard-*Python*-Installation existieren. Die Daten werden in den Tabellen namens **`sentences`**, **`speakers`** sowie **`recordings`** abgelegt und miteinander verknüpft. Darüber hinaus wird eine Ansicht **`overview`** erzeugt, welche die wichtigsten Daten zusammen in einer Tabelle darstellt. Nach dieser Datentransformation kann die resultierende Datenbank beispielsweise in *DB Browser for SQLite*[44] betrachtet und gezielt durchsucht werden (siehe Abbildung 6.1), um die Audiodateien von Interesse zu finden.

Für das Durchführen der Aufnahmen ist es zu bevorzugen, dass die Rohdaten selbst wenig von Störgeräuschen, Hall und sonstigen Einwirkungen betroffen sind, da solche durch den Versuchsaufbau nochmal dazukommen. Von den verschiedenen Aufnahmevarianten des Korpus klingen die mit dem Namenszusatz *Yamha* nach subjektiver Einschätzung des Autors am natürlichsten. Diese Aufnahmen sind auch in allen Teilen des Korpus vertreten. Darüber hinaus liegen z. B. auch verarbeitete Daten eines Mikrofon-Arrays vor, welche allerdings Verzerrungen und Probleme bei der adaptiven Ausrichtung zu beinhalten scheinen. Der *Tuda-De*-Korpus ist aufgeteilt in Testdaten und Trainingsdaten. Mit den 1028 unterschiedlichen Aufnahmen pro Gerät im Test-Set sollen *ASR*-Engines bewertet werden. Dabei kommen sowohl die Texte als auch die Sprecher nicht im Trainings-Set

[43] *SQLite*: `https://www.sqlite.org/` (besucht am 31.01.2020)
[44] *DB Browser for SQLite*: `https://sqlitebrowser.org/` (besucht am 31.01.2020)

vor. Das wesentlich größere Trainings-Set (14398 Aufnahmen pro Gerät) beinhaltet im Gegensatz dazu auch viele Sätze, die von verschiedenen Stimmen gesprochen werden. Dadurch können am Ende Schlussfolgerungen über die Unterschiede zwischen den Stimmen gezogen werden, da die Texte aufgrund der Gleichheit keinen Einfluss auf das Ergebnis nehmen.

6.2 Implementierung des Aufnahmeablaufs

Die Ablauflogik des Aufnahmevorgangs der Testdaten ist in dem Knoten der Klasse **`SchedulerRecord`** implementiert. Eine textbasierte *Tab-separated values (TSV)*-Datei, welche einfach manuell mit einem Texteditor erstellt werden kann, dient dabei als Testvorschrift, die definiert, welche Dateien abgespielt werden sollen. Der Inhalt muss so formatiert sein, dass in der ersten Spalte die Dateinamen stehen und in der zweiten Spalte die dazugehörigen Texte. Alle Spalten sind mit Tabulatoren voneinander getrennt. Für diesen Aufnahmeablauf werden die Texte hier zwar nicht benötigt, jedoch kann so die selbe Datei für den späteren Bewertungsvorgang wiederverwendet werden. Der Pfad zu dieser Datei muss als Wert des Parameters **`test_rule_file_path`** übergeben werden.

Der Knoten nutzt die Services von weiteren Knoten, welche *WAVE*-Dateien abspielen, den Ton über *PulseAudio* ausgeben und das Mikrofonsignal eines anderen Gerätes entgegennehmen und letztendlich wieder abspeichern. Das Sequenzdiagramm in Abbildung 6.2 stellt dabei die wichtigsten Kommunikationsabläufe dar. Nicht gezeigt wird hier der Transport der Audiodaten und mancher Nachrichten, welche bei diesem Vorgang keine Rolle spielen. Dabei wird für jede Audiodatei in der Liste zunächst die Aufnahme gestartet, dann das Abspielen der Rohdaten begonnen und nach Abschluss der Wiedergabe die Aufnahme gestoppt.

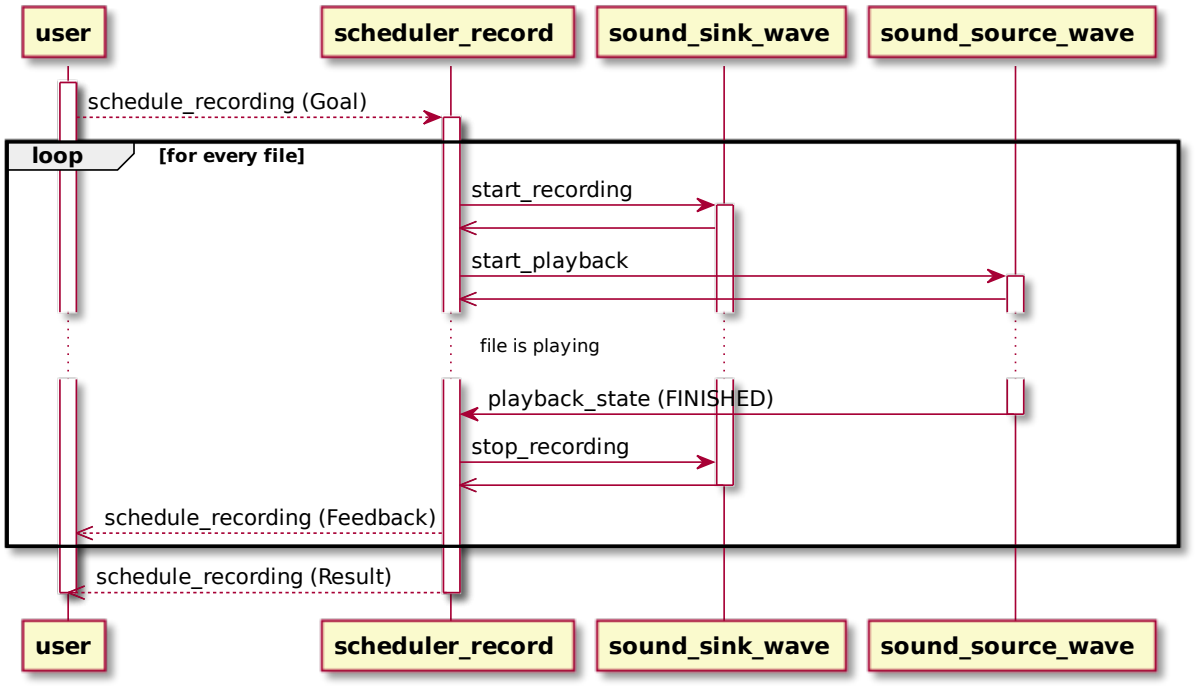

Abbildung 6.2: Vereinfachtes Sequenzdiagramm des Aufnahmeablaufs

Entgegen von einfachen Nachrichten, sind Service-Aufrufe mit einem zusätzlichen nicht ausgefüllten Pfeil in die Gegenrichtung gekennzeichnet, da immer eine Antwort zurück kommt. Bei den gestrichelten Pfeillinien handelt es sich um *ROS2*-Actions. Man kann sie sich als asynchrone Services vorstellen. Während bei einem Service-Aufruf eine direkte Antwort erwartet wird, kann ein Client bei einer Action ein Ziel (Goal) setzen, in der Zwischenzeit andere Aufgaben abarbeiten und danach auf die Antwort reagieren. Darüber hinaus können dem Aufrufer Zwischenergebnisse (Feedback) gesendet und die Möglichkeit gegeben werden, eine Action abzubrechen. Beschrieben wird ein Action-Typ in *ROSIDL* durch die Angabe von nicht nur zwei sondern drei Teilnachrichten (Ziel, Ergebnis und Feedback). Mit Services und einem selbst programmierten Thread-Handling ist dies auch möglich. Die Kapselung dieser und weiterer Mechanismen in einer Action macht die Programmierung jedoch einfacher und eleganter.

Der Teilnehmer **user** in der Abbildung 6.2 steht stellvertretend für den Auslöser der Aufnahmeprozedur. Das kann ein weiterer Knoten oder aber auch der Kommandozeilenbefehl **ros2 action send_goal** sein (siehe Abbildung 6.2 Zeile 2), welcher im Hintergrund auch als ein Knoten ausgeführt wird, wobei der Name, der Typ sowie die Argumente der Action (in diesem Fall keine) dahinter angegeben werden müssen. Die Option **--feedback** sorgt für die sofortige Ausgabe der Feedback-Nachrichten, welche die Gesamtanzahl der aufzunehmenden Dateien und die Anzahl der bereits abgearbeiteten Dateien beinhalten und nach jedem Schleifendurchlauf gesendet werden. In der Ergebnisnachricht befinden sich keine weiteren Daten.

Zum vorherigen Initialisieren aller für diese Aufgabe benötigten Knoten wurde das Launch-Skript **record_test_data** erstellt. Es benötigt Informationen in Form von Argumenten, wo sich die Testvorschrift befindet, in welchem Verzeichnis die Audiodateien liegen, wo die Aufnahmen abgespeichert werden sollen und welches Eingabegerät zu verwenden ist (Zeile 1). Gespeichert werden die Aufnahmen mit dem selben Dateinamen wie die Rohdaten.

```
1 $ ros2 launch rosass record_test_data.launch.py rule_file
    ▶ :=list.tsv audio_data_dir:=tuda-de recording_dir:=rec
    ▶ -respeaker device:=respeaker
2 $ ros2 action send_goal --feedback /schedule_recording
    ▶ rosass_msgs/action/ScheduleRecording '{}'
```

Abbildung 6.3: Beispielbefehle zum Aufnehmen der Testdaten vom *ReSpeaker*

Ein Spezialfall ist die Verwendung des *Pepper*-Roboters. Denn dessen Mikrofon-Array kann von außen nur über eine Netzwerkverbindung genutzt werden. Aus diesem Grund wurde die Launch-Konfiguration in zwei Teile separiert, welche abhängig von bestimmten Argumenten aktiviert werden oder nicht. Hat **play** den Wert `True`, dann werden alle Knoten bezüglich der Wiedergabe inklusive **scheduler_record** gestartet. Alle anderen Knoten sind mit dem Argument **record** verbunden. Beide Argumente tragen standardmäßig den Wert `True` und müssen im Normalfall nicht explizit angegeben werden. Auf diese Weise muss nur eine Launch-Datei für alle Varianten gepflegt werden. Das Komponentendiagramm in Abbildung 6.4 zeigt diese Aufteilung. Auch hier ist die Bereitstellung der Action-Schnittstelle mit einer gestrichelten Linie gekennzeichnet. Die Halbkreise stellen die Nutzung eines Services dar. Wenn auf dem Entwicklungsrechner, welcher mit dem

Lautsprecher verbunden ist, nur der `play`-Teil gestartet wird (`record` ist `False`) und auf dem *Pepper* nur der `record`-Teil (`play` ist `False`), kann der Aufnahmeprozess durchgeführt werden. Das Speichern der aufgezeichneten Dateien findet auf dem Roboter statt. Dieser flexible Ansatz ist deshalb möglich, weil sich *ROS2*-Knoten automatisch gegenseitig finden, wenn sie im selben Netzwerk agieren. Denn sie senden ständig Informationen über sich aus. Damit mehrere *ROS2*-Netzwerke nicht ungewollt miteinander interferieren, gibt es die Umgebungsvariable `ROS_DOMAIN_ID`. Die hier anzugebende Ganzzahl zwischen 0 und 100, wird intern auf IP-Port-Nummern umgerechnet, auf denen das gegenseitige Entdecken (Discovery) stattfindet. Somit interagieren nur Knoten der Rechner miteinander, welche die selben Identifikationsnummern konfiguriert haben.

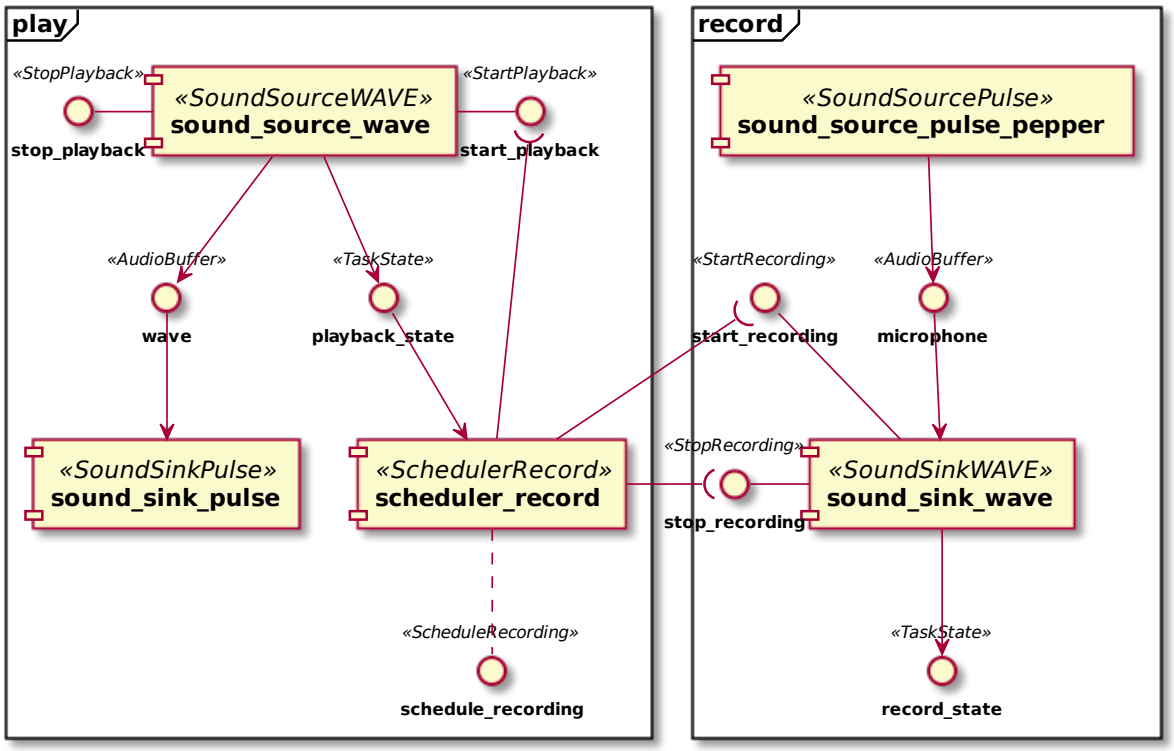

Abbildung 6.4: Komponenten des Aufnahmevorgangs

6.3 Durchführung der Aufnahmen

Der Aufnahmeprozess wurde in einem kleinen Büroraum durchgeführt. Die Ansteuerung des Lautsprechers und der externen Mikrofon-Arrays geschah über die *USB*-Schnittstellen des Entwicklungsrechners, wodurch diese Geräte in *PulseAudio* zur Verfügung standen. Abbildung 6.5 zeigt Fotos vom Testaufbau.

Die genauen Maße sind in der nicht maßstabsgetreuen Skizze in Abbildung 6.6 hinterlegt. Dabei ist das Mikrofon-Array vom *Pepper* etwa auf einer Höhe von 1,15 m in waagerechter Ausrichtung platziert. Etwa 1,55 m davon entfernt und um 15 cm angehoben befindet sich

Abbildung 6.5: Fotos des Testaufbaus

ein hochkant montierter länglicher Lautsprecher[45]. Der Hintergedanke bei dieser Ausrichtung ist, dass dadurch die Ausdehnung der schallabgebenden Fläche im Horizontalbereich verringert wird und somit einem Menschen stärker ähnelt. Die Vertikalausdehnung dieses Stereo-Lautsprechers konnte auch durch die Deaktivierung der unteren Hälfte verkleinert werden, indem die Lautstärke des rechten Kanals auf Null gesetzt wurde. Dazu können Programme wie *pavucontrol*[46] oder *pulsemixer*[47] genutzt werden, um die Lautstärken und weitere Einstellungen in *PulseAudio* durchzuführen. Der linke Kanal stand auf der Lautstärkestufe 120. Die Aufnahmen mit den anderen Eingabegeräten wurden jeweils einzeln abgewickelt, wobei sie auf dem Kopf vom *Pepper*-Roboter fixiert wurden, um ähnliche Ausgangsbedingungen zu schaffen (siehe Abbildung 6.7).

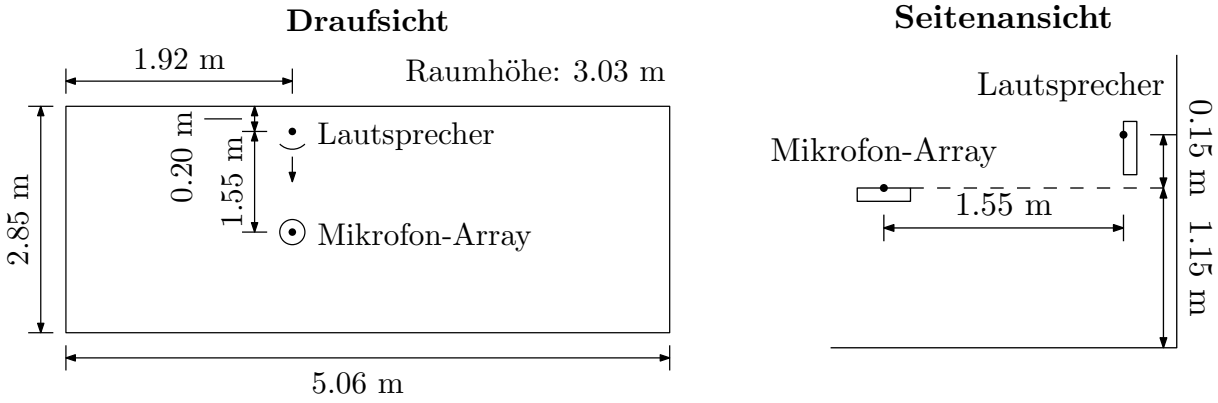

Abbildung 6.6: Skizze des Testaufbaus

6.4 Implementierung der Bewertung

Zum Zweck der Evaluation der Audiodaten bezüglich der Erkennbarkeit durch *ASR*-Engines, wurde in *RoSaSS* ein *ROS2*-Knoten der Klasse `EvaluatorText` erstellt. Dieser liest die Testvorschrift ein, welche auch beim Aufnahmeprozess verwendet wurde, und arbeitet sie Datei für Datei ab. Dabei wird erst die Spracherkennung aktiviert, dann die

[45] Als Lautsprecher wurde das Modell *SPA5210B/10* verwendet: `https://www.philips.de/c-p/SPA5210B_10/` (besucht am 03.02.2020)

[46] *pavucontrol*: `https://freedesktop.org/software/pulseaudio/pavucontrol/` (besucht am 02.02.2020)

[47] *pulsemixer*: `https://github.com/GeorgeFilipkin/pulsemixer` (besucht am 02.02.2020)

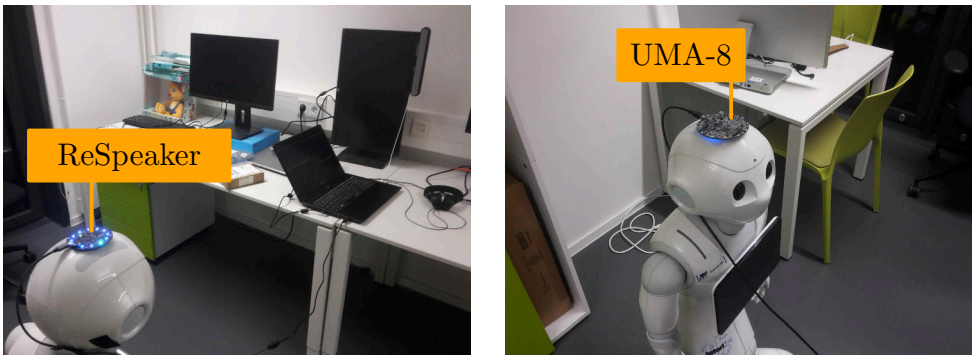

Abbildung 6.7: Fotos des Testaufbaus mit *ReSpeaker* und *UMA-8*

entsprechende Datei abgespielt und nach Ablauf dieser die Hypothese von der *ASR*-Engine eingeholt sowie ganz am Ende durch einen Vergleich mit dem Originaltext ausgewertet. Das Sequenzdiagramm in Abbildung 6.8 zeigt die wesentlichen Schritte des Ablaufs.

Im Unterschied zum Aufnahmeprozess wird das Stoppen des Audiodatenkonsumenten (hier die jeweilige *ASR*-Engine) nicht explizit durch einen Service-Aufruf herbeigeführt, sondern selbst durch den Empfang und die Verarbeitung eines mit **`is_last_buffer`** markierten Audiopuffers ausgelöst. Das ist notwendig für den korrekten zeitlichen Ablauf, da die Verarbeitungsdauer der Spracherkennung nicht bekannt ist und darum das Stoppen von ihr selbst ausgehen muss. Besonders deutlich wird es bei der hierfür vorgesehenen Nutzung der schnellen Verarbeitung, wobei alle Schritte so schnell wie möglich ausgeführt werden (siehe Parameterbeschreibung zu **`realtime_mode`** in Abschnitt 5.3.1). Da in diesem Modus die Audiodaten einer Datei am Stück in einer Nachricht versendet werden, was zwar die Gesamtlaufzeit verringert, aber die Latenz pro Puffer vergrößert, ist ohne eine solche Synchronisation oder eine großzügige aber verschwenderische Wartezeit eine korrekte Ausführung nicht möglich. Wird zur Bewertung die *VoCon Hybrid*-Engine herangezogen, welche indirekt die Audiodaten über *PulseAudio* bekommt und nicht darauf reagieren kann, bleibt nichts anderes übrig, als die Variante mit der Wartezeit anzuwenden. Ohnehin muss hier der Realzeitmodus verwendet werden, weshalb eine großzügige Wartezeit von 4 s zu verschmerzen ist, bevor die Hypothese abgefragt wird.

Die Signalverarbeitungskette zwischen **`sound_source_wave`** und **`asr`** ist in der Abbildung nicht modelliert, wobei die Audiopuffer durch die *Topic* **`source`** gesendet werden und nach einer Abtastratenkonvertierung unter dem Namen **`resampled`** letztendlich zur *ASR*-Engine finden. Dazwischen können beliebige Knoten zur Verarbeitung eingefügt werden. Welche das sind, kann beim Aufruf des Launch-Skripts **`evaluation_text.launch.py`** spezifiziert werden. Abbildung 6.9 zeigt die entsprechenden Befehle zum Starten der Knoten und zum Auslösen des Bewertungsvorgangs. Zu den Argumenten zählen die Angabe der Testvorschrift, das Verzeichnis der zu lesenden *WAVE*-Dateien sowie zu verwendende Spracherkennungssoftware. In diesem Beispiel werden die Aufnahmen vom *ReSpeaker* mit *Kaldi* bewertet.

Es gibt darüber hinaus noch den Parameter **`dsp_name`**. Dort kann angegeben werden, welches Signalverarbeitungsverfahren (*Digital-Signal-Processing (DSP)*), das die Audiodaten verändert, verwendet werden soll. Anhand des übergebenen Wertes wird ein danach benanntes Launch-Skript unter **`rosass/launch/dsp/`** zusätzlich angewendet. Die darin beschriebenen Knoten nehmen die Daten der *Topic* **`source`** auf und stellen das

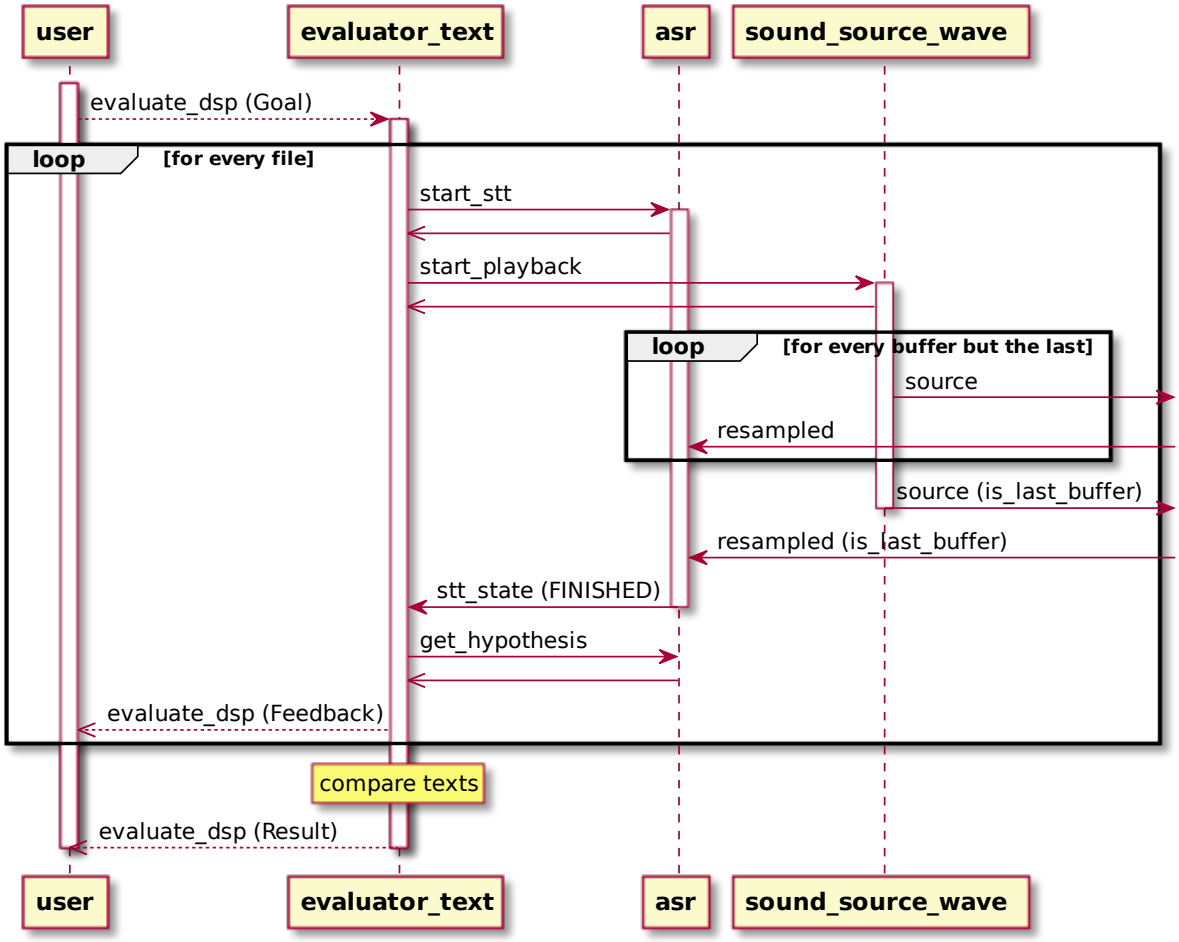

Abbildung 6.8: Vereinfachtes Sequenzdiagramm des Evaluationsablaufs

Ergebnis durch **`processed`** bereit. Beim Weglassen des Argumentes entsteht an dieser Stelle ein Knoten, welcher einfach die **`AudioBuffer`**-Nachrichten unverändert weitergibt (erzeugt aus **`dsp_bypass.launch.py`**). Die resultierende Knotenkonstellation ist in Abbildung 6.10 zu sehen.

Als Feedback zur auslösenden Action wird auch hier wieder der Fortschritt als Dateianzahl übermittelt. Nach dem Vergleich der Texte steht die Wortfehlerrate in der Ergebnisnachricht. Wie diese berechnet wird, klären die folgenden Unterabschnitte.

6.4.1 Die Wortfehlerrate

Die Wortfehlerrate, welche auch unter dem Namen *Word-Error-Rate (WER)* zu finden ist, gehört zu den wichtigsten Größen, wenn es darum geht, die Leistung von *ASR*-Engines zu bewerten. Hier wird dieses Maß dazu genutzt, um Verbesserungen durch eine Vorverarbeitung festzustellen. Sie entsteht aus dem Vergleich von Originaltext und der Hypothese der Spracherkennung, indem der Quotient aus der Editierdistanz und der Anzahl der Wörter im Originaltext gebildet wird. Als Editierdistanz (auch *Levenshtein*-Distanz genannt) ist die minimale Anzahl an nötigen Operationen definiert, die den einen Text in den anderen überführen [Wik18a]. Dabei sind Einfüge-, Lösch- und Ersetzungsoperationen (zusammengesetzt aus Löschen und Einfügen) erlaubt. Diese Betrachtung arbeitet auf Sequenzen

```
1 $ ros2 launch rosass evaluation_text.launch.py rule_file:=
    ▶ list.tsv audio_data_dir:=rec-respeaker asr_engine:=
    ▶ kaldi
2 $ ros2 action send_goal --feedback /evaluate_dsp
    ▶ rosass_msgs/action/EvaluateDSP '{}'
```

Abbildung 6.9: Beispielbefehle zum Durchführen der Evaluation

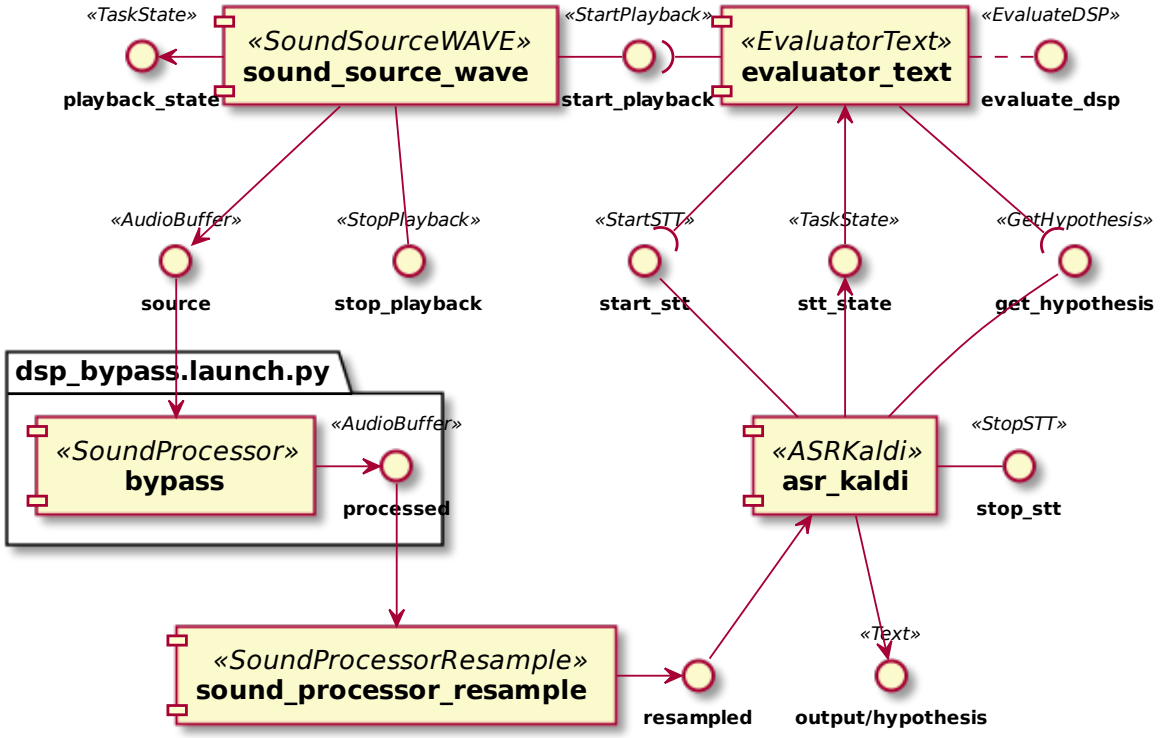

Abbildung 6.10: Komponenten des Evaluationsprozesses

von miteinander vergleichbaren Elementen. Im Fall der Wortfehlerrate sind es Sequenzen aus Wörtern, wobei alle Satzzeichen ignoriert werden. Denn diese werden von den *ASR*-Engines nicht erkannt. Dies gilt in der Regel auch für die Groß- und Kleinschreibung. Daher muss für einen fairen Vergleich der Originaltext von Satzzeichen bereinigt sein. Nach dem Einlesen der Texte als Zeichenketten wird zunächst alles auf Kleinschreibung umgestellt und anschließend werden die Buchstabensequenzen zwischen den Leerzeichen als Wörter gruppiert. Die so entstandene Wortsequenz eignet sich nun für die Distanzberechnung. Trotzdem kann es bezüglich der *ASR*-Engine noch Sonderfälle geben, da sich die Schreibweise von gleichen Wörtern unterscheiden kann. Bemerkbar macht sich das bei der Nutzung von *VoCon Hybrid*. In den *Tuda-De*-Daten sind, wie bei den anderen Spracherkennungsprogrammen auch, gesprochene Zahlen als Zahlenwörter ausgeschrieben. *VoCon Hybrid* hingegen liefert diese als umgewandelte Ziffern zurück, wodurch die Wörter fälschlicherweise als ungleich betrachtet werden. Für diesen Bewertungsprozess muss daher der Originaltext entsprechend angepasst vorliegen.

Ein Textbeispiel zur Distanzbestimmung ist in der Abbildung 6.11 zu sehen. Um von der ersten Zeile zur zweiten Zeile zu gelangen sind drei Einfügungen (+), eine Löschung (−) und eine Ersetzung (∓) nötig. Die Editierdistanz ist daher die Summe fünf. An

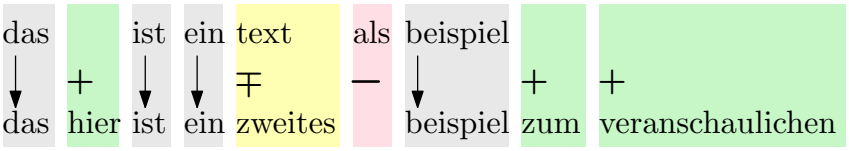

Abbildung 6.11: Beispieltexte mit einer Editierdistanz von fünf Operationen

diesem Beispiel ist auch erkennbar, dass es mehrere Lösungen geben kann. Anstelle der Ersetzung von „text“ und Löschung von „als“ hätte auch eine Löschung von „text“ und eine Ersetzung von „als“ stattfinden können. Das ist aber nicht von Bedeutung, da sich die minimale Gesamtanzahl an Operationen dadurch nicht ändert. Zur Berechnung der *WER* wird die Distanz durch die Wortanzahl der ersten Zeile geteilt ($\frac{5}{6} \approx 83\,\%$). Ein Wert von 0 % deutet auf Gleichheit. Bei 100 % können keinerlei Übereinstimmungen festgestellt werden. Der Wertebereich ist nach oben jedoch nicht begrenzt. Denn durch das Einfügen von Elementen in einem der beiden Texte, kann die *WER* theoretisch bis ins Unendliche getrieben werden.

6.4.2 Berechnung der Wortfehlerrate

Ein Mensch sieht bei solchen kurzen Texten relativ einfach die Muster und kann intuitiv die notwendigen Operationen nennen, ohne einen konkreten Algorithmus zu folgen. Dieser wird aber für die automatisierte *WER*-Berechnung benötigt. Implementiert wurde der *Wagner-Fischer*-Algorithmus, welcher eine Distanzmatrix berechnet [Wik19b], die nun an dem Beispiel in Abbildung 6.12 erklärt wird. Gegeben sind die zu vergleichenden Sequenzen s_1 und s_2 mit den Elementen (e_a, e_b, e_c, e_d) und $(e_a, e_e, e_c, e_d, e_f)$. Nun wird eine Matrix erstellt mit $|s_2| + 1$ Zeilen und $|s_1| + 1$ Spalten, dessen Indizes aufsteigend die Elementindizes repräsentieren, wenn die Sequenzen von vorn um ein Element erweitert werden würden. An dieser Stelle ist in der Abbildung das leere Element ϵ gezeigt. Die erste Zeile und Spalte wird mit den Werten der Indizes initialisiert (nullbasierte Indizierung). Für die Berechnung jedes anderen Feldes, müssen die Werte der drei benachbarten Felder (blau und grün markiert), deren Zeilen und/oder Spaltenindizes kleiner sind, herangezogen werden. Aus diesem Grund wird mit dem Feld $(1, 1)$ begonnen (orangener Rahmen, links). Die kleinen Zahlen auf den Nachbarfeldern stellen Zwischenergebnisse dar. Diese werden durch die Addition mit 1 gebildet, die beim diagonalen Feld nur stattfindet, wenn sich die Elemente unterscheiden. In diesem Fall sind e_a und e_a identisch. Aus den Zwischenergebnissen wird das Minimum (grün) übernommen. Die Berechnung wird zeilenweise oder spaltenweise fortgeführt, bis das letzte Feld der Distanzmatrix seinen Wert erhalten hat. Dieser Wert entspricht der Editierdistanz. In dem Beispiel sind s_1 und s_2 um zwei Editieroperationen voneinander entfernt.

Bei der Bewertung wird hier nicht nur ein Textpaar verglichen, sondern je ein Paar pro Aufnahme, die durch die Testvorschrift definiert sind. Es gibt mehrere Möglichkeiten die Gesamtwortfehlerrate zu bestimmen. In der einen Variante werden alle Originaltexte und alle Hypothesen jeweils konkateniert, sodass ein einzelnes Textpaar entsteht. Alternativ kann bei der anderen Variante für jedes Paar zunächst nur die Editierdistanz berechnet werden. Die *WER* ist dann der Quotient aus der Summe aller Distanzen und der Gesamtzahl aller Wörter in den Originaltexten. In der Regel führen beide Varianten zum

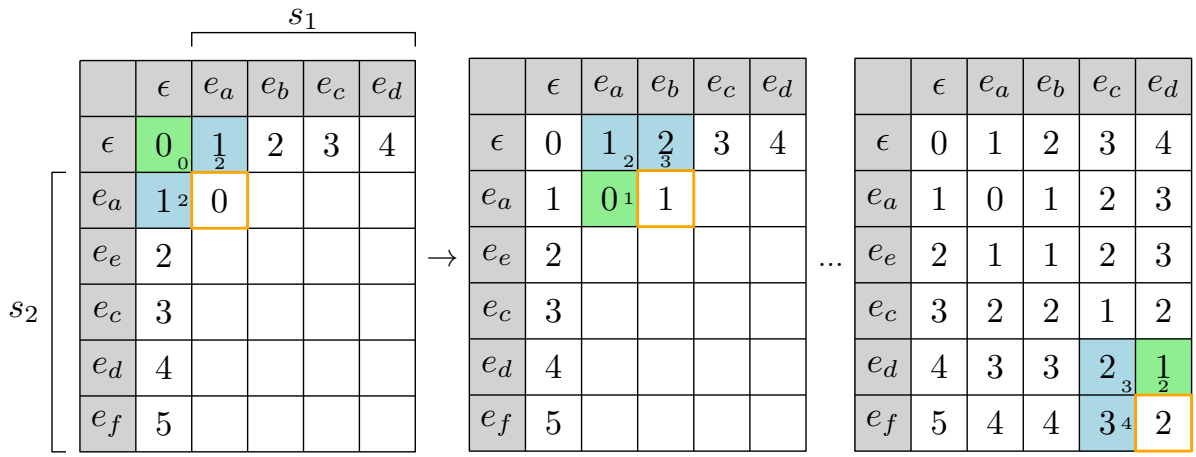

Abbildung 6.12: Schritte zur Berechnung der Distanzmatrix

gleichen Ergebnis. Sonderfälle gibt es aber trotzdem, wenn es paarübergreifend zu ungewollten Übereinstimmungen kommt. Denn bei der Konkatenation geht die eindeutig durch den Bewertungsablauf bestimmte Zuordnung der einzelnen erkannten Wörter zur Aufnahme und damit zum Textabschnitt verloren. So kann beispielsweise ein zu viel erkanntes Wort und ein nicht erkanntes Wort bei der nächsten Aufnahme identisch sein, was die Abbildung 6.13 an einem Beispiel zeigt. Die Einzelberechnung erkennt hingegen diese Art von Fehler. Darüber hinaus wächst der Rechenaufwand bei steigender Anzahl an Textpaaren nur linear. Bei der Konkatenation entsteht dabei jedoch eine große Matrix, welche einen quadratisch steigenden Aufwand bewirkt. Aus diesen Gründen wird eine Einzelberechnung durchgeführt.

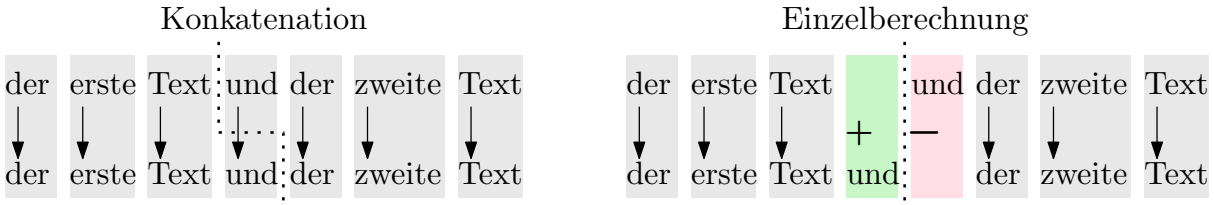

Abbildung 6.13: Gegenüberstellung von Konkatenation und Einzelberechnung

6.5 Ergebnisse

Um einen ersten Überblick über die Ausgangssituation zu haben, wurde der Bewertungsprozess ohne zusätzliche Verarbeitungsschritte für die verschiedenen Eingabegeräte und *ASR*-Engines durchgeführt. Aus dem Test-Set des *Tuda-De*-Korpus sind zehn Aufnahmen mit einer Gesamtspiellänge von 52 s für die Aufnahme und Bewertung selektiert worden. Die Auswahl ging mehr oder weniger willkürlich von statten. Es wurde jedoch darauf geachtet, keine einseitige Liste der Aufnahmen bezüglich Altersklassen und Geschlecht zusammenzustellen. Alle verwendeten Originalaufnahmen, Testvorschriften und eigene Aufnahmen sind in der Beilage 1 zu finden. Tabelle 6.1 beinhaltet die berechneten Bewertungsergebnisse, welche darüber hinaus in Abbildung 6.14 visualisiert sind. Mit Ausnahme von *Kaldi*, werden die in Abschnitt 2.4 aufgeführten Modelle benutzt. Denn

bei der verwendeten *Python*-Schnittstelle musste aus Kompatibilitätsgründen des Formates auf ein anderes Modell ausgewichen werden[48].

Tabelle 6.1: Wortfehlerrate Test-Set in %

Eingabegerät	PocketSphinx	DeepSpeech	Kaldi	VoCon Hybrid
Original	48.5	71.2	5.9	50.5
ReSpeaker (DSP)	84.2	97.0	41.6	92.1
ReSpeaker	91.1	100.0	84.2	100.0
UMA-8	92.1	100.0	83.2	97.0
Pepper	96.7	100.0	97.9	100.0

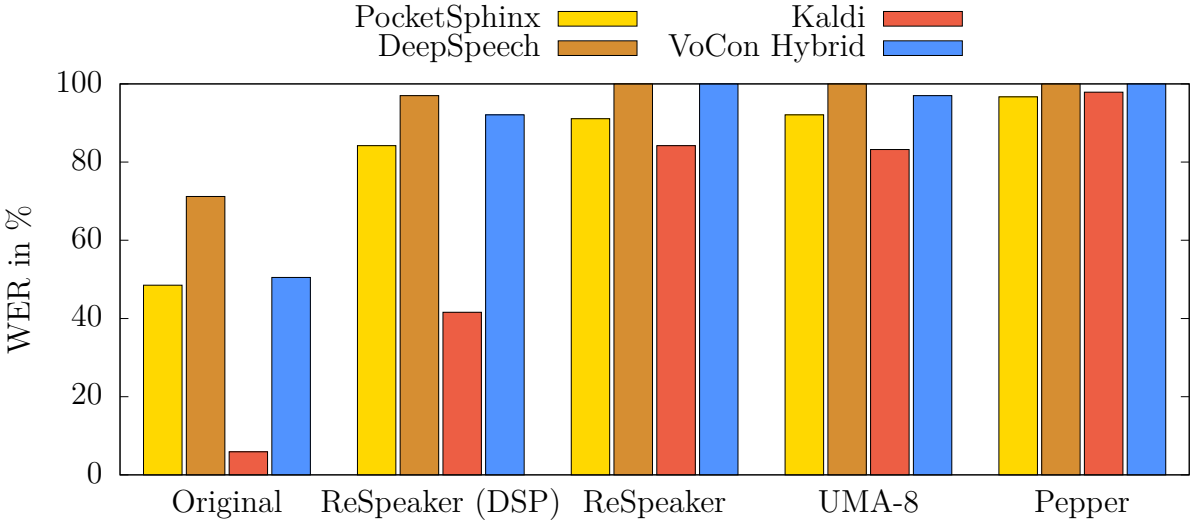

Abbildung 6.14: Wortfehlerrate Test-Set

Das Eingabegerät „Original“ steht für die direkte Bewertung der Rohdaten, ohne dass sie neu aufgenommen wurden. Dort sind sehr deutliche Unterschiede zwischen den Spracherkennungsprogrammen zu erkennen. Während *Kaldi* eine sehr gute Leistung demonstriert, macht *PocketSphinx* mit einer *WER* von fast 50 % vergleichsweise viele Fehler. Ein ähnlicher Wert wurde bei *VoCon Hybrid* bestimmt. Erwähnenswert ist, dass die Spracherkennung dort immer wieder von der vorgeschalteten *VAD* unterbrochen wurde. Bei der Durchführung der Bewertung auf dem Roboter musste außerdem in *PulseAudio* manuell die Lautstärke erhöht werden, damit überhaupt Text erkannt wurde. Das geht vermutlich ebenfalls auf das Konto der *VAD*, welche erst ab einen bestimmten Lautstärkepegel die Daten an den Decoder weiterzureichen scheint. In der Variante, in der die Spracherkennung genutzt wird, hat der Autor aber keine Einstellmöglichkeit zu diesem Schwellwert gefunden. Die eigentliche Leistung dürfte daher besser sein, als es diese Werte suggerieren. Während bei den anderen *ASR*-Engines immer das selbe Ergebnis entstand[49], traten bei *VoCon Hybrid* starke Schwankungen in wiederholten Versuchen auf, weshalb immer der beste von drei Durchgängen ausgewählt wurde. Eine gewisse dynamische Adaptivität konnte bei allen hier getesteten Engines festgestellt werden. Jedoch werden beim

[48] Modell für *Kaldi*: `https://goofy.zamia.org/zamia-speech/asr-models/kaldi-generic-de-tdnn_f-r20190328.tar.xz` (besucht am 10.02.2020)

[49] bei Neuinitialisierung der Modelle

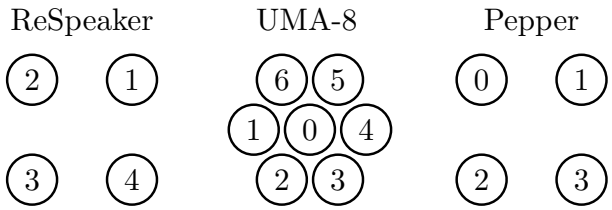

Abbildung 6.15: Kanalzuweisung bei den Mikrofonen der Eingabegeräte

Neustarten der Knoten die Modelle erneut geladen und die *ASR*-Engines auf den Startzustand zurückgesetzt, weshalb sie die Ergebnisse im ersten Durchlauf reproduzieren. Weitere Bewertungsvorgänge ohne einen Neustart, ergeben hier auch Veränderungen. Jedoch sind diese gegenüber *VoCon Hybrid* sehr gering. Um *VoCon Hybrid* zurückzusetzen müsste vermutlich jedes mal *NAOqi* neugestartet werden, was aber recht viel Zeit kostet. Warum *DeepSpeech* in diesen Tests so schlecht ausfällt, konnte der Autor nicht herausfinden. Es besteht die Möglichkeit, dass *DeepSpeech* suboptimal durch *RoSaSS* verwendet wird.

Bei den weiteren Eingabegeräten wurden zunächst alle Mikrofonkanäle mit einer Abtastrate von 48 kHz aufgezeichnet und jeweils den Kanal mit der kleinsten Nummer für die Bewertung herangezogen. Eine Übersicht der Kanalzuweisung zu den Mikrofonen ist in Abbildung 6.15 zu sehen. Dabei ist oben vorne. Bei diesen Aufnahmen ist die Erkennbarkeit der Sprache erwartungsgemäß gesunken und das teilweise so drastisch, dass *DeepSpeech* und *VoCon Hybrid* mitunter leere Hypothesen zurückgeliefert haben. Denn die Spracherkennung steht dabei vor sehr schwierigen Bedingungen. Viele Faktoren spielen dabei eine Rolle. Dazu zählen die nicht perfekte Wiedergabe durch den Lautsprecher, der Hall im Büroraum, die relativ große Entfernung zwischen Quelle und Senkte sowie die ebenfalls nicht perfekte Aufnahme beim Eingabegerät. Im Falle des Mikrofon-Arrays beim *Pepper* kommen noch laute Lüftergeräusche hinzu. Darum ist es nicht verwunderlich, dass hier allgemein die höchsten Fehlerraten erreicht wurden. Für die anderen Aufnahmen wurde der Roboter ausgeschaltet. Der *ReSpeaker* und der *UMA-8* schneiden in etwa gleich schlecht ab. Hinter den Namen „ReSpeaker (DSP)“ verbirgt sich der Kanal mit dem Index null. Dabei handelt es sich um ein auf dem Gerät selbst vorverarbeitetes Signal, welches bei allen *ASR*-Engines Verbesserungen hervorruft. Bei *Kaldi* konnte die *WER* sogar halbiert werden. Daraus ergibt sich die Erkenntnis, dass es möglich ist, mit entsprechenden Verfahren eine signifikante Aufwertung der Spracherkennung zu erreichen. Der *UMA-8* besitzt auch eine solche Funktionalität, die aber nicht gleichzeitig mit der Aufzeichnung der Mikrofonsignale möglich ist. Dazu hätte die Firmware häufig gewechselt werden müssen.

Tabelle 6.2: Wortfehlerrate Trainings-Set in %

Eingabegerät	PocketSphinx	DeepSpeech	Kaldi	VoCon Hybrid
Original	8.6	71.9	0.7	80.5
ReSpeaker (DSP)	68.0	99.2	55.5	100.0
ReSpeaker	90.6	99.2	95.3	100.0
UMA-8	85.2	100.0	94.5	100.0
Pepper	94.5	100.0	98.4	100.0

Darüber hinaus wurde eine weitere Testreihe aus dem Trainings-Set des Korpus vollzogen. Es wurden 18 Aufnahmen zusammengestellt, welche in der Summe eine Laufzeit von

68 s ergeben und deren Ergebnisse in Tabelle 6.2 sowie in Abbildung 6.16 präsentiert werden. Auffällig ist, dass *Kaldi* bei den Rohdaten eine fast perfekte Hypothese (*WER* < 1 %) aufstellt. Grund hierfür ist, dass das verwendete Modell mit genau diesen Daten trainiert wurde und darum damit besonders gut klar kommt. Diese Effekte relativieren sich aber bei der Auswertung der erneuten Aufnahmen durch die anderen Geräte. Da ist die Fehlerrate sogar etwas höher. Die Erkennungsleistung von *DeepSpeech* ist davon entgegen den Erwartungen fast unberührt, obwohl das Modell ebenfalls auf Basis des *Tuda-De*-Korpus entstand. *PocketSphinx* steht in dieser Testreihe auch besser dar und profitiert stärker von der Aufbereitung durch den *ReSpeaker*. Das verwendete Modell stützt sich zwar auf einen anderen Korpus, aber es kommen teilweise die selben Formulierungen vor. Für *VoCon Hybrid* ist diese Auswahl schwieriger zu verarbeiten. Das Gesamtbild bleibt jedoch ähnlich. Man darf die Aussagekraft dieser und der vorherigen Ergebnisse nicht überbewerten, da es sich nur um eine recht kleine Stichprobe handelt, wo die Verständlichkeit von einzelnen Stimmen eine große Rolle spielen kann. Ziel ist jedoch nicht eine möglichst präzise Bestimmung zu haben, sondern eine Möglichkeit am Ende Verbesserungen messbar zu machen. Dafür sollte die Stichprobengröße ausreichend sein.

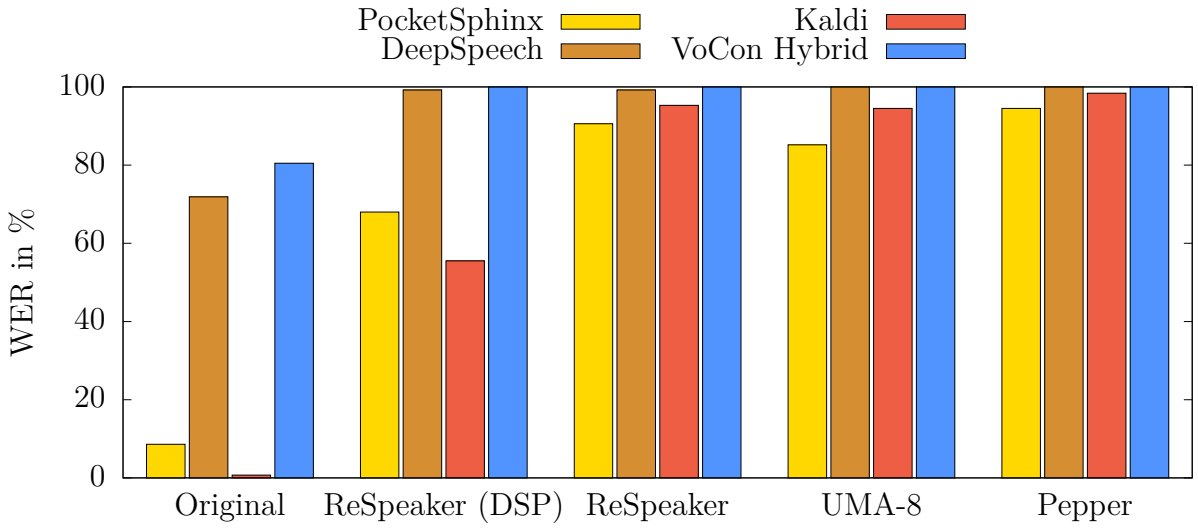

Abbildung 6.16: Wortfehlerrate Trainings-Set

Was durch das Test-Set so bisher nicht gezeigt werden konnte, ist ein Geschlechtervergleich, da im Trainings-Set die selben Texte von verschiedenen Menschen vorgelesen wurden. Eventuell lassen sich daraus Aussagen treffen, ob und warum Stimmen in verschiedenen Tonhöhen in diesem Testfall die Erkennbarkeit beeinflussen. Die oben ausgewerteten Aufnahmen aus dem Trainings-Set beinhalten jeden Satz mit Audiodaten von einer männlichen und einer weiblichen Stimme. Bei getrennter Betrachtung (siehe Tabelle 6.3 und Abbildung 6.17), welche aufgrund der nicht ganz so hohen Fehlerraten nur mit *PocketSphinx* und *Kaldi* durchgeführt wurden, ist keine eindeutige Tendenz erkennbar. Bei den Rohdaten hat *Kaldi* in der weiblichen Variante (f) sogar alles komplett richtig bestimmt, während bei den anderen Aufnahmen die männliche Variante (m) eher zu etwas weniger Fehlern führt. Das ist jedoch bei jedem Gerät und jeder Spracherkennungssoftware etwas anders. Interessant ist, dass die Ergebnisse des zusammengefassten Bewertungsprozesses nicht immer genau zwischen denen der getrennten Bewertung liegen, was sich wieder auf die adaptive Anpassung der *ASR*-Engines zurückführen lässt. Denn schon beim Ändern

der Reihenfolge kommen andere Hypothesen zurück. Bei einer zeitlich sehr aufwendigen Testreihe mit einer wesentlich größeren Auswahl an Aufnahmen würden sich diese Auswirkungen im Gesamtergebnis minimieren.

Tabelle 6.3: Wortfehlerrate Trainings-Set nach Geschlechtern getrennt in %

Eingabegerät	PocketSphinx (m)	PocketSphinx (f)	Kaldi (m)	Kaldi (f)
Original	9.4	4.7	1.6	0.0
ReSpeaker (DSP)	70.3	68.8	53.1	61.0
ReSpeaker	75.0	84.4	92.2	98.4
UMA-8	90.6	84.4	95.3	98.4
Pepper	98.4	98.4	100.0	96.9

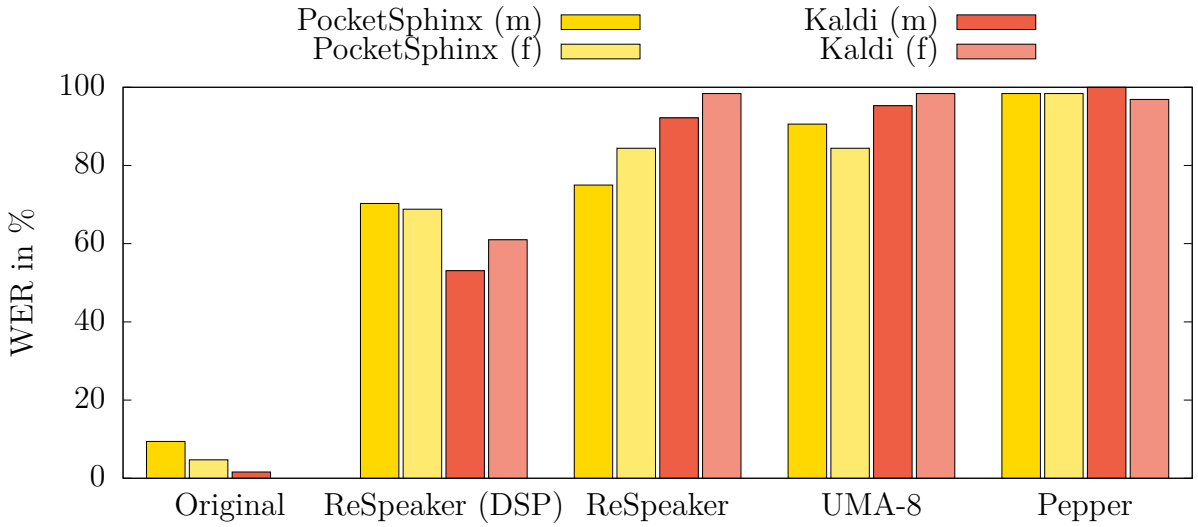

Abbildung 6.17: Wortfehlerrate Trainings-Set nach Geschlechtern getrennt

Vielleicht interessanter ist die nun folgende Betrachtung. Bei den bis hierher bewerteten Aufnahmen der anderen Geräte wurde der Roboter mit samt seiner Lüfter ausgeschaltet, um ihn mit üblichen Bedingungen (ohne Roboter) zu vergleichen. Nach dem Einschalten des Roboters verhalten sich die nun so bestimmten Fehlerraten beim *ReSpeaker* und *UMA-8* ähnlich wie die beim Mikrofon-Array des *Peppers*, obwohl diese außerhalb des Gehäuses angebracht waren. Darüber hinaus fallen die Verbesserungen durch den *ReSpeaker* wesentlich geringer aus (siehe Tabelle 6.4 und Abbildung 6.18).

Tabelle 6.4: Wortfehlerrate Trainings-Set mit Lüftergeräuschen in %

Eingabegerät	PocketSphinx	Kaldi
ReSpeaker (DSP)	80.5	97.7
ReSpeaker	93.0	99.2
UMA-8	96.1	99.2

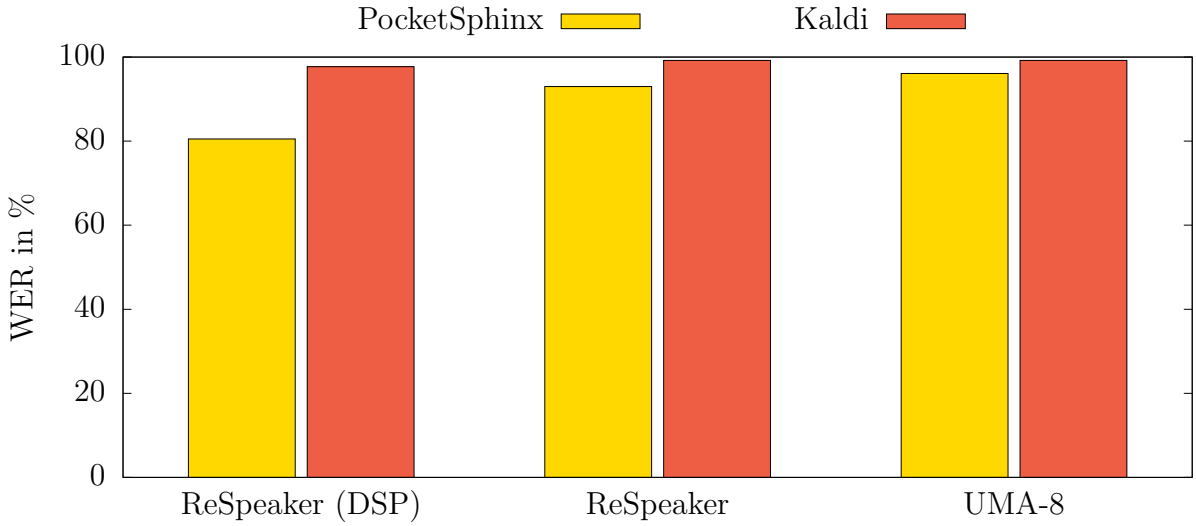

Abbildung 6.18: Wortfehlerrate Trainings-Set mit Lüftergeräuschen

7 Signalverarbeitung

Dieses Kapitel beschäftigt sich mit der Aufbereitung der Audiodaten mit Fokus auf dem Mikrofon-Array des *Pepper*-Roboters. Um das Problem besser zu verstehen, lohnt sich eine Visualisierung in Form von Spektrogrammen. Diese zeigen die Frequenzanteile eines Signals von kleinen Abschnitten über den Zeitverlauf. In Abbildung 7.1 ist das Spektrogramm einer Audiodatei aus dem Trainings-Set des *Tuda-De*-Korpus zu sehen, welches mit dem Programm *SoX*[50] erstellt wurde.

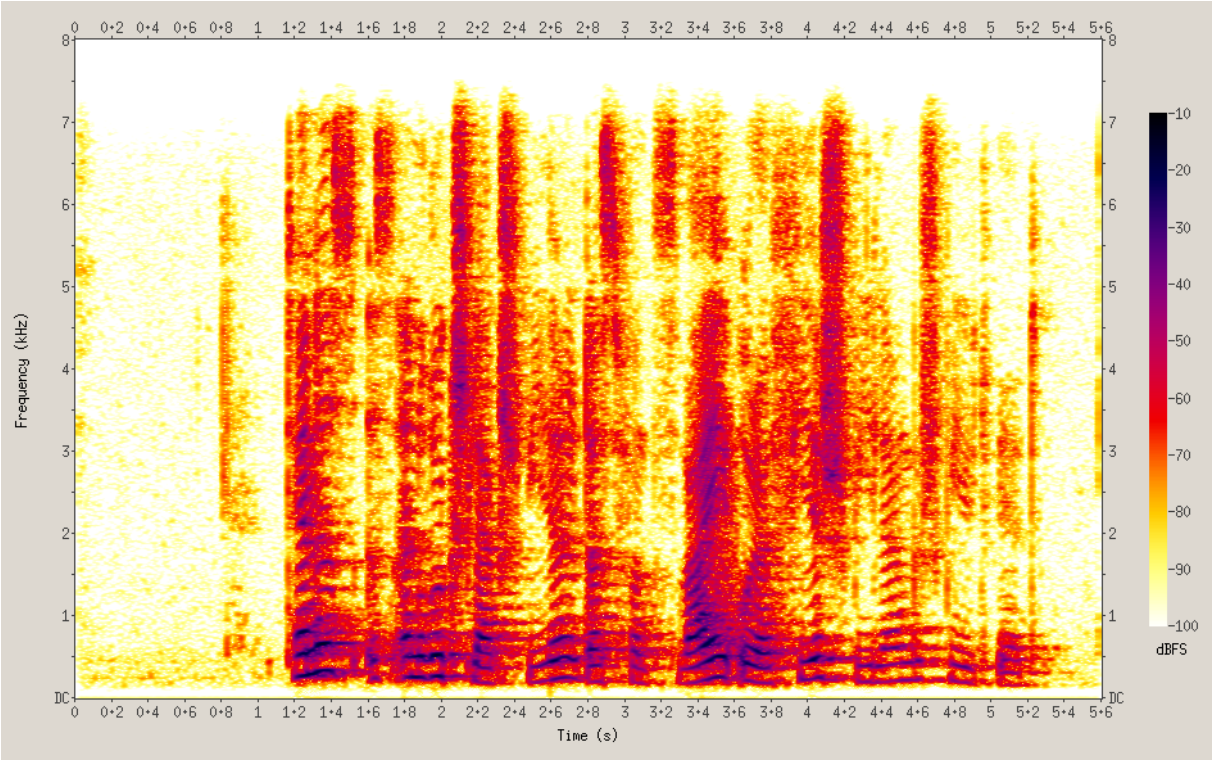

Abbildung 7.1: Spektrogramm der Datei `2014-08-11-13-34-16_Yamaha.wav`

Dort ertönt der Satz „Tennis ist ein Rückschlagspiel, das von zwei oder vier Spielern gespielt wird.“. Auf der x-Achse verläuft die Zeit in Sekunden und auf der y-Achse die Frequenz mit einer oberen Schranke von 8 kHz, da die Aufnahme eine Abtastrate von 16 kHz hat (Abtasttheorem [Smi97, S. 39–44]). Die Farbskala gibt an, wie stark eine Frequenz vertreten ist. Das wird hier in der Einheit *Decibels relative to full scale (dBFS)* angegeben, wobei Dezibel eine logarithmische Verhältnisgröße ist. In diesem Fall ist der Referenzwert (0 dBFS) die maximal mögliche Amplitude, welche in dem verwendeten Format abgespeichert werden kann. Da ein lauteres Signal nicht möglich ist, kann die Skala nur in den negativen Bereich ragen. Zur besseren visuellen Erkennbarkeit beginnt die Farbskala auf diesen Spektrogrammen erst bei −10 dBFS und endet bei −100 dBFS. Das Nutzsignal

[50] *SoX*: `http://sox.sourceforge.net/` (besucht am 06.02.2020)

(Sprache) hebt sich sehr deutlich von dem sehr leisen und daher hell dargestellten Hintergrundrauschen ab. Bei den kurzen horizontalen Linien, welche in regelmäßigen Abständen im Frequenzbereich wiederkehren, handelt es sich um tonale Laute mit einer Grundfrequenz und Obertönen (vielfache der Grundfrequenz). Harte Konsonanten sind als scharfe vertikale Kanten sichtbar, da sie kurzzeitig einen großen Frequenzbereich belegen.

Durch die erneute Aufzeichnung mittels *Peppers* Mikrofonen entsteht dagegen das Spektrogramm in Abbildung 7.2 (Kanal null). Abgesehen von der Tatsache, dass es eine kleine zeitliche Verschiebung gibt, die nicht weiter stören soll, sind die einst klaren Sprachmuster deutlich schwieriger zu erkennen. Die Lautstärke ist allgemein angehoben, was zum einen an der Aufnahmelautstärke und zum anderen am dazukommenden Rauschpegel liegt. Der obere helle Streifen ist die Spur eines Tiefpassfilters, welcher dafür sorgt, dass höhere Frequnenzanteile entfernt werden. Dieser wird vor der Abtastung oder auch einer Abtastratenkonvertierung angewendet, um Aliasing-Effekte ([Smi97, S. 39–44]) zu vermeiden. Neben der Überlagerung mit dem Rauschen ist noch ein zweiter Effekt zu beobachten. Die Merkmale des Sprachsignals wirken in Laufrichtung der Zeit horizontal verwischt. Erklären lässt sich das durch die Reflexionen bzw. Mehrfachreflexionen des Schalls an den Wänden und sonstigen Gegenständen, welche zusätzlich zum auf direktem Weg empfangenen Signal verzögert am Mikrofon eintreffen.

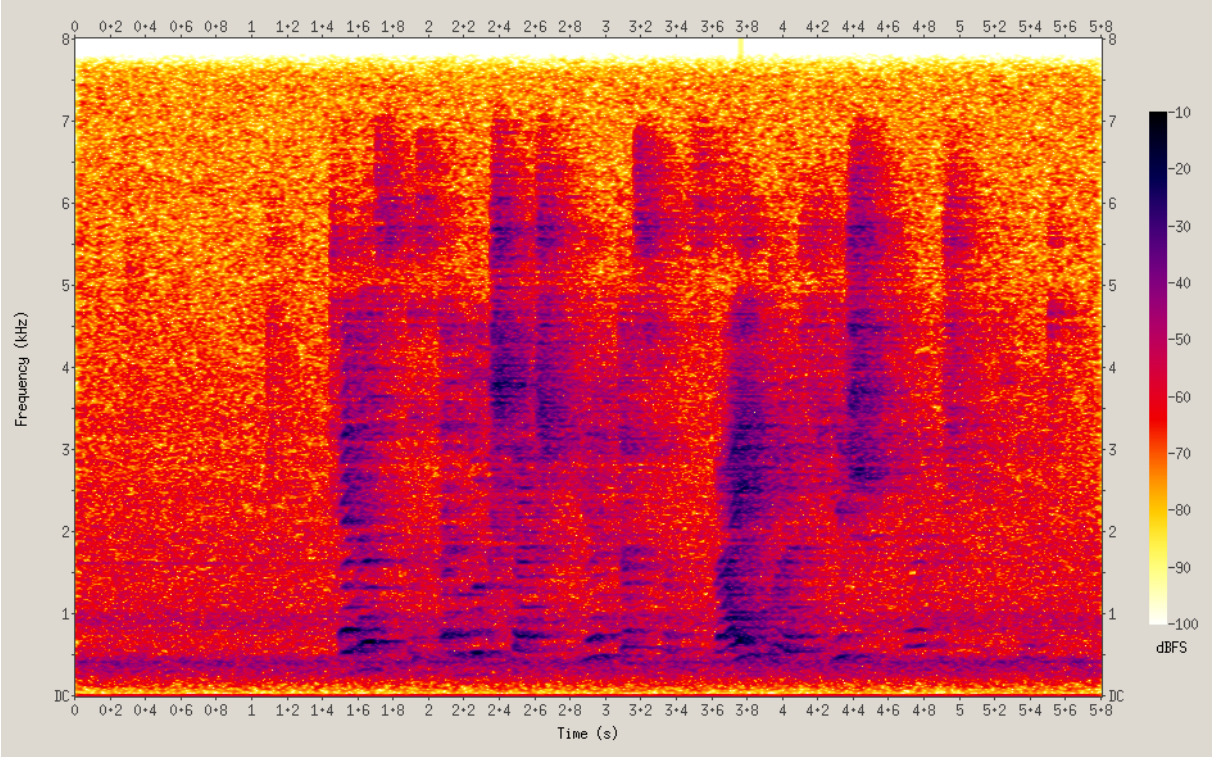

Abbildung 7.2: Spektrogramm bei *Peppers* Mikrofonen

In den folgenden Abschnitten wird ein Einblick in die Implementierung und Evaluierung von mehreren relevanten Signalverarbeitungsverfahren gegeben, die diese akustische Situation verbessern sollen. Doch zunächst geht es um eine automatische Parameteroptimierung, welche zwar kein Signalverarbeitungsverfahren in diesem Sinne ist, aber für das Bestimmen von guten Parameterwerten der Verfahren Anwendung findet.

7.1 Parameteroptimierung

Mit der Bestimmung der Wortfehlerrate gibt es bereits eine Messmethode für die Güte des resultierenden Signals. Da die meisten Verarbeitungsverfahren sich oft mit mehreren numerischen Parameterwerten anpassen lassen, muss die richtige Besetzung gefunden werden, um ein optimales Ergebnis zu liefern. Dies kann beispielsweise durch manuelles Probieren geschehen, was aber im Fall von vielen Parametern ein zielgerichtetes Vorgehen erschwert. In Anbetracht des jedes Mal neu durchzuführenden Bewertungsprozesses, um die Beeinflussung der Parameteranpassung zu erfahren, ist es eine recht langwierige Angelegenheit, die automatisiert werden sollte. Genau dies tut der Knoten **`OptimizerES`**, dessen Namen auf die Verwendung der Evolutionsstrategie deutet, die im nächsten Unterabschnitt näher erklärt wird. Dieser Knoten ist selbst parametrisiert durch **`parameter_definition_path`**. Der dort anzugebende Pfad muss auf eine Beschreibungsdatei im *YAML Ain't Markup Language (YAML)*-Format zeigen, welche die einzustellenden Parameter mit ihren Ausgangswerten definiert. Sie könnte z. B. so aussehen wie in Abbildung 7.3. Dieses Format wurde deswegen gewählt, da es genau in dieser Form auch von *ROS2*-Werkzeugen zum Initialisieren von Parametern angewendet werden kann. Die hierarchische Datenstruktur beginnt mit einem Knotennamen als Wurzelelement, dessen Parameter eingestellt werden sollen. Unter dem Element **`ros__parameters`** kann eine Liste aus Parameternamen sowie ihren Werten angelegt werden. In dieser Datei ist es möglich, mehrere solcher Konstrukte zu definieren, um Einstellungen bei vielen Knoten auf einmal vornehmen zu können.

```
1 /evaluator_dummy:
2     ros__parameters:
3         param1: 0.0
4         param2: 0.0
```

Abbildung 7.3: Beispiel einer Parameterbeschreibungsdatei

`OptimizerES` liest diese Datei mit Hilfe von *PyYAML*[51] ein und nimmt die darin gefundenen Werte als Ausgangszustand. Während des Optimierungsprozesses muss immer wieder eine Bewertung herangezogen werden. Darum benutzt **`OptimizerES`** die Action **`EvaluateDSP`**, wodurch z. B. der Bewertungsprozess von **`EvaluatorText`** ausgelöst wird, welcher im Abschnitt 6.4 bereits vorgestellt wurde. Um die Funktionsweise einfach testen zu können, ohne eine Vielzahl an Knoten starten zu müssen, wurde der Knoten **`EvaluatorDummy`** geschaffen, dessen Name in Abbildung 7.3 vorkommt. Auch dieser stellt die Action **`EvaluateDSP`** bereit und berechnet selbst anhand seiner eigenen Parameter **`param1`** und **`param2`** einen Ergebniswert. Dabei ist das Optimum bei null mit den willkürlich gewählten Parameterwerten 3,14 und 42. Der Ergebniswert berechnet sich aus der Summe der Abstände aller Parameter zu ihren Optimalwerten. Auf diese einfache Weise wird ein Minimierungsproblem emuliert. Wie auch bei der *WER* ist der kleinste mögliche Wert anzustreben. Gestartet wird der Optimierungsprozess mit der Action **`Optimize`**, deren Definition Abbildung 7.4 zeigt. Dort kann mit dem booleschen Feld **`maximize`** festgelegt werden, ob der Ergebniswert maximiert oder minimiert werden soll. Ist das Limit **`limit`** erreicht, wird die Optimierung abgebrochen. Darüber hinaus lässt sich noch eine

[51] *PyYAML*: `https://pyyaml.org/` (besucht am 06.02.2020)

Schrittweite einstellen, deren Erklärung später folgt. In der Antwort wird das gefundene Optimum zurückgegeben. Die Feedback-Nachrichten informieren über den aktuellen Zustand der Parameter (**values**) mit ihren Namen (**param_names**) sowie der Wert des bisher bestimmten Optimums (**quality**). Die Bedeutung der Generationsnummer wird später erklärt. Wichtig ist zu verstehen, dass der Knoten **OptimizerES** nur Parameter einstellt und die Bewertung des Zustands in Form einer Gleitkommazahl abfragt, ohne dabei die Zusammenhänge des restlichen Systems zu kennen.

rosass_msgs/action/Optimize.action

```
1  # problem type
2  # True: maximize, False: minimize
3  bool maximize True
4  # step size of optimization
5  # use 0.0 for automatic step size (dynamic)
6  float32 step_size 0.0
7  # finish if limit is reached
8  float32 limit
9  ---
10 float32 optimum
11 ---
12 uint16 generation
13 float32 quality
14 float32[] values
15 string[] param_names
```

Abbildung 7.4: Action-Definition von **Optimize.action**

7.1.1 Die Evolutionsstrategie

Die Evolutionsstrategie ist ein nichtlineares Optimierungsverfahren. Das bedeutet, dass es im Stande ist, mit Systemen umzugehen, bei welchen das Optimum nicht durch das Lösen eines linearen Gleichungssystems bestimmt werden kann. Daher sind die Eingabewerte nicht proportional zum Ausgabewert des Systems. Bei der hier bestimmten *WER* ist sogar noch nicht einmal ein mathematischer Zusammenhang in Form einer Formel bekannt. Aus diesem Grund bleibt nur ein Ausprobieren von verschiedenen Parameterwerten übrig. Das muss jedoch nicht planlos erfolgen. Für solche Fälle ist eben die sogenannte Evolutionsstrategie gut geeignet. Sie orientiert sich mit ihren Grundprinzip an der biologischen Evolution [Sch76, S. 123].

Der Startzustand (Zustand der Parameter) wird dabei als Gensatz eines Elter bezeichnet, welcher mutierte Nachkommen erzeugt. Ein Nachkomme trägt also einen veränderten Satz von Genen mit sich, wobei die Mutation durch eine Addition einer Zufallszahl zu jedem einzelnen Gen herbeigeführt wird. Die Zufallszahl wird für jedes Gen neu bestimmt und liegt in einem Bereich, der die maximal mögliche Erhöhung und Reduktion durch eine einstellbare Schrittweite begrenzt. Dadurch unterscheiden sich die Individuen der Population in ihren Genen (Parameterwerte) und darum auch in ihrer Vitalität (Qualität bzw. Bewertungsergebnis). Im Anschluss darauf erfolgt ein Selektionsvorgang, wobei

das beste Individuum zum neuen Elter ernannt wird und alle anderen verworfen werden. Die Auswahl kann auch erneut auf denselben Elter treffen, wenn keine seiner Nachkommen besser sind. Nach dem gleichen Schema geht es mit der zweiten und den folgenden Generationen weiter [Sch76, S. 124].

Dabei gibt es unzählige Varianten der Evolutionsstrategie, die unterschiedliche Vorgehensweisen anwenden in Bezug auf die Menge der Nachkommen eines Elters pro Generation, wie viele Eltern im Spiel sind und wie das Selektionsverfahren abläuft. Hier wurde die einfachste Variante (1+1)-ES implementiert, wobei es nur einen Elter mit einem Nachkommen pro Generation gibt. Darüber hinaus ist eine adaptive Schrittweiteneinstellung zur Erhöhung der Fortschrittsgeschwindigkeit nach der $\frac{1}{5}$-Erfolgsregel [Sch76, S. 128 ff.] im Algorithmus integriert, welcher dafür sorgt, dass das Optimum schneller also in weniger Generationen erreicht wird. Angewendet wird diese Anpassung, wenn das Feld `step_size` den Wert `0.0` trägt. Ansonsten bleibt die Schrittweite konstant auf ihren angegebenen Wert.

Voraussetzung für die erfolgreiche Anwendung der Evolutionsstrategie ist, dass das System nicht chaotisch auf Änderungen der Eingabewerte reagiert und der Zusammenhang zum Ausgabewert stückweise stetig ist. Es muss einen Verlauf in Richtung des Optimums geben, sodass sich Generation für Generation herangetastet werden kann. Einzelne Sprungstellen oder Verläufe in eine andere Richtung können aber durchaus überwunden werden. Das hängt jedoch unter anderem von den generierten Zufallszahlen ab. Bei der Evolutionsstrategie kann nicht versichert werden, ob der gefundene Wert tatsächlich das globale Optimum eines Systems ist oder sich in einem lokalen Optimum „festgefahren“ hat. Das kann passieren, wenn keine der angewendeten Mutationen zu einem besseren Ergebnis führt.

Da die *WER* aufgrund der endlichen Wortanzahl nur diskrete Werte annimmt, kann sich der Verlauf als eine Art Treppenfunktion vorgestellt werden. Dort ist es recht wahrscheinlich, dass einige Mutationen keine Änderung der Wortfehlerrate herbeiführen, wodurch sich Elter und Nachkomme häufig auf einem Plateau befinden und nicht eindeutig bestimmt werden kann, wer besser bzw. in die richtige Richtung mutiert ist. Darum sollten nicht zu wenige Aufnahmen in den Bewertungsprozess einfließen. Genau genommen kann allgemein die digitale Zahlendarstellung eines Computers auch nur diskrete Werte annehmen. Diese Abstufungen sind aber vernachlässigbar klein bei den verwendeten Gleitkommazahlen. Ein weiteres Problem ist, dass die *ASR*-Engines bei gleichen Eingabewerten nicht immer dieselben Ergebnisse liefern, was den Selektionsprozess stört, da ein leicht schlechterer Gensatz zu einer besseren Bewertung führen kann.

7.1.2 Visualisierung

Um den aktuellen Zustand und den Verlauf des Optimierungsprozesses komfortabler einschätzen zu können, wurde mit dem Knoten der Klasse `VisualizerES` eine Live-Visualisierung in Form von Diagrammen implementiert.

Diese zeigen die Parameterwerte und den Ergebniswert über den Verlauf der Generationen (siehe Abbildung 7.5). Verwendet wurde dazu die *Python*-Bibliothek *gnuplotlib*[52], welche

[52] *gnuplotlib*: `https://github.com/dkogan/gnuplotlib` (besucht am 06.02.2020)

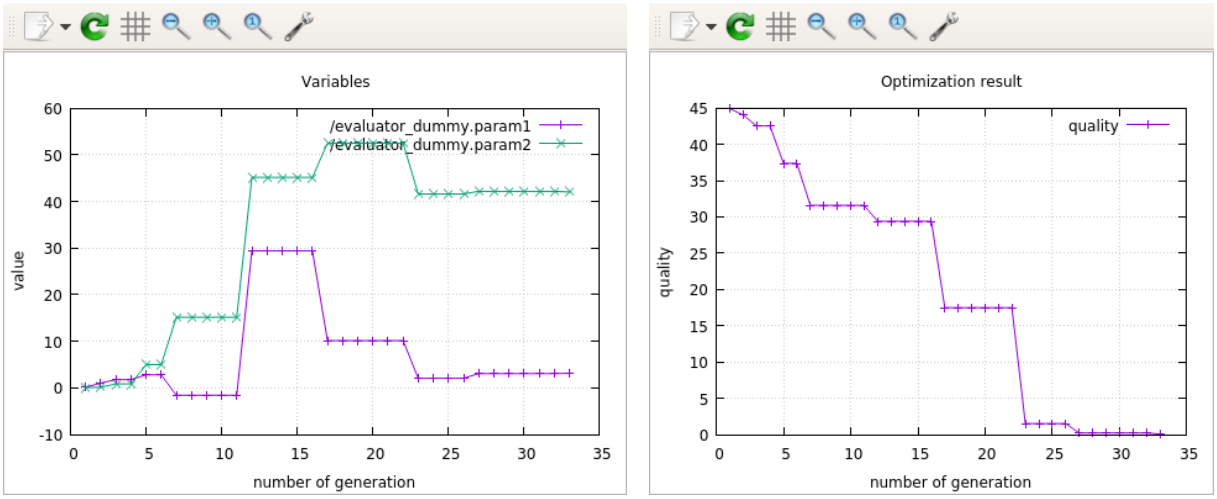

Abbildung 7.5: Bildschirmfotos von **`VisualizerES`**

im Hintergrund das Diagrammwerkzeug *Gnuplot*[53] nutzt. Die Visualisierung speist sich allein aus den Daten der Feedback-Nachrichten von der Action **`Optimize`**. Da normalerweise das Feedback nur an den Auslöser gesendet wird, veröffentlicht **`EvaluatorES`** dieselben Nachrichten zusätzlich über einen weiteren Kanal, sodass nebenbei **`VisualizerES`** auch an die Daten kommt. Erfolglose Mutanten werden nicht dargestellt. Darum sind Abschnitte ohne Änderung zu sehen, weil über mehrere Generationen keine besseren Nachkommen erzeugt wurden. In diesem Beispiel wurde der bereits vorgestellte **`EvaluatorDummy`** als zu optimierendes System herangezogen. Die Knotenkonstellation kann mit dem Launch-Skript **`optimization_dummy.launch.py`** initialisiert werden.

7.2 Rauschreduzierung

Die nun folgenden Verfahren widmen sich der Rauschreduzierung. Um das störende Rauschen besser zu verstehen, wurde zunächst ein Rauschprofil erstellt, welches auch bei den implementierten Verfahren als Grundlage dient.

7.2.1 Erstellung eines Rauschprofils

Damit das Rauschen gut durch ein statisches Rauschprofil charakterisiert werden kann, darf es keine starken Veränderungen über den Zeitbereich geben. Zur Überprüfung und auch als Grundlage für die Profilerstellung wurde über eine Minute lang nur das Rauschen aufgenommen. Das daraus generierte Spektrogramm (Kanal null) in Abbildung 7.6 zeigt über den langen Zeitbereich gesehen keine sichtbaren Schwankungen. Betrachtet man jedoch kurze Zeitabschnitte, sind diese Lüftergeräusche deutlich verrauscht. Leider ist die Belastung im unteren Frequenzbereich am stärksten, wo auch vorrangig die Sprache angesiedelt ist. Diese Geräusche haben zwar einzelne Hochpunkte, die als horizontale Linien zu erkennen sind, jedoch sind sie allgemein leider breitbandig. Wären es nur einzelne sehr

[53] *Gnuplot*: `http://www.gnuplot.info/` (besucht am 06.02.2020)

schmale Frequenzbänder, könnten sie unkompliziert herausgefiltert werden und hätten wenig Überlagerungen mit dem Nutzsignal.

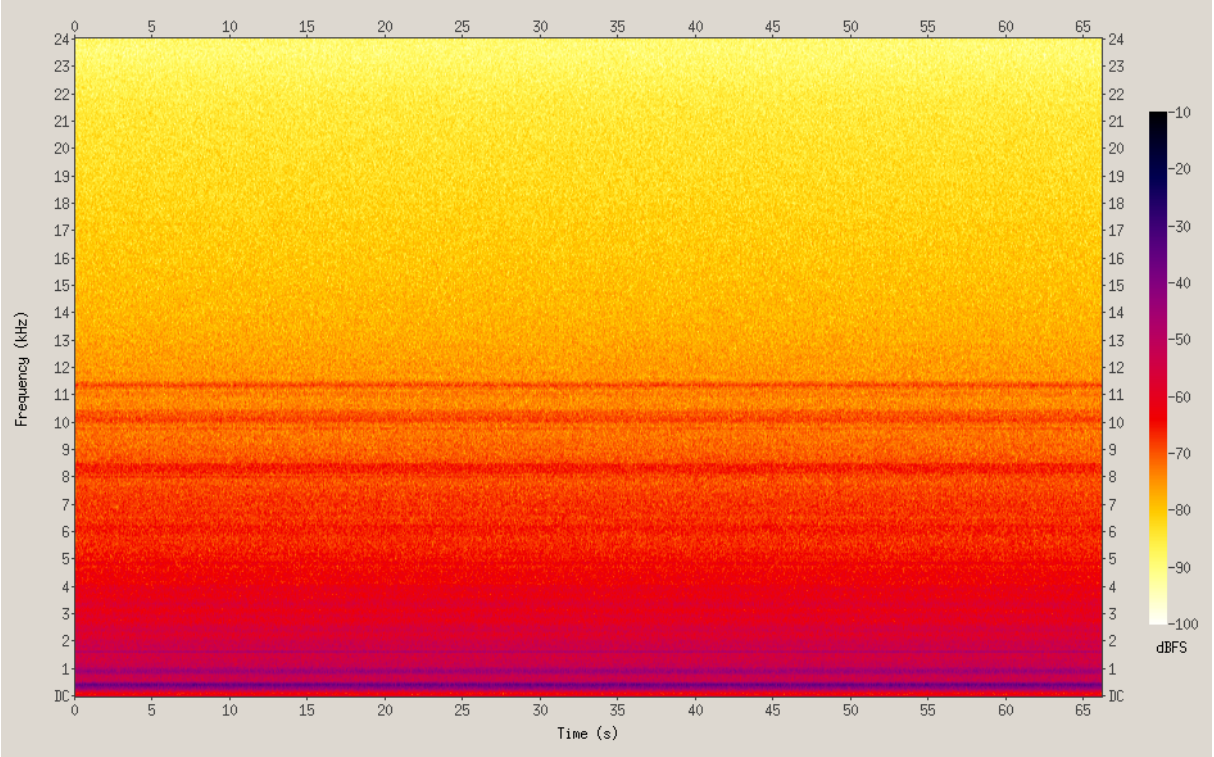

Abbildung 7.6: Spektrogramm vom Lüfterrauschen

Das hier zu generierende Rauschprofil enthält Informationen darüber, welche Frequenzen wie stark im Rauschen vertreten sind. Dazu wird das Spektrum über die Zeit gemittelt. Dieser Vorgang erledigt das *Python*-Programm **`extract_noise_profile.py`**, das sich im **`bin`**-Verzeichnis befindet. Im ersten Schritt werden die Audiodaten der eingelesenen Datei in Segmente mit einer Größe von 2048 Samples unterteilt. Von denen wird dann jeweils das Spektrum berechnet. Möglich wird dies mit der Anwendung der sogenannten *Fast-Fourier-Transform (FFT)*, welche durch *NumPy* bereitgestellt wird. Dabei handelt es sich um eine effiziente Berechnungsmethode der *Discrete-Fourier-Transform (DFT)*, die ein Signal in ihre Frequenzbestandteile zerlegt. Die Länge des zu analysierenden Signals bzw. Segments sollte bei der *FFT* eine Potenz von zwei sein, da nur dann bestimmte Eigenschaften für die schnelle Berechnung ausgenutzt werden können [Mey17, S. 174 ff.]. Ist dies nicht der Fall, kann der Rest bis zur nächsten Zweierpotenz mit Nullen aufgefüllt werden [Mey17, S. 202]. Bei den hier gewählten 2048 Samples pro Segment besteht dazu aber keine Notwendigkeit. Das Ergebnis dieser *DFT* repräsentiert 2049 komplexe Zahlen, welche je für ein Frequenzband stehen. Mit der vorliegenden Abtastrate von 48 kHz resultiert die Transformation in einem Band bei 0 Hz und 1024 weiteren Bändern bis einschließlich 24 kHz. Übrig bleiben 1024 Frequenzbänder im negativen Bereich, die die Werte der positiven Bänder spiegeln und darum nur redundante Informationen enthalten. Diese können hier ignoriert werden. Mit den 1025 übriggebliebenen komplexen Zahlen lässt sich in diesem Format nicht viel erkennen. Durch die Umformung in Polarkoordinaten aus den Real- und Imaginäranteilen, welche man sich zusammen als zweidimensionale Vektoren vorstellen kann, entstehen so interpretierbare Zusammenhänge. Polarkoordinaten haben einen Betrag (Länge) und einen Winkel, die mit der Amplitude und der Phase des Signals

pro Frequenzband korrespondieren [Smi97, S. 161–164]. Für das Rauschprofil sind nur die Amplituden interessant. Diese werden für jedes Segment berechnet. Das arithmetische Mittel pro Frequenzband über alle Segmente ergibt das Rauschprofil.

Doch dabei existiert ein Problem. Denn die *DFT* ist für periodische Signale geeignet. Die hier unterteilten Segmente sind jedoch nicht periodisch. Darum kommt es zum sogenannten *Leakage*-Effekt, bei dem die Frequenzen nicht ganz sauber zerlegt werden können [Mey17, S. 188]. Denn die *DFT* betrachtet die Eingangsdaten als zyklisch, wobei der letzte Sample der Vorgänger des ersten ist. Bei Signalen wie diesen haben die beiden Samples nichts miteinander zu tun und es kommt an der Stelle bei der zyklischen Betrachtung zu einer Diskontinuität (Sprung), welche weitere Frequenzen im Spektrum verursacht, die eigentlich nicht enthalten sind. Die Lösung ist die Anwendung einer Fensterung (siehe Abbildung 7.7). Dabei wird von der Mitte des Segments in Richtung Anfang und Ende ein Ausblenden vollzogen, sodass der erste und letzte Sample an den Wert null angeglichen werden. Dadurch ist dieser Sprung eliminiert. Das Ausblenden darf aber auch nicht zu abrupt geschehen, da sonst ebenfalls zusätzliche Frequenzanteile entstehen. Für einen schonenden Übergang gibt es eine Vielzahl an Fensterfunktionen zur Auswahl. Verwendet wird das *Hanning*-Fenster, welches ebenfalls einfach mit *NumPy* generiert werden kann. Mit der elementweisen Multiplikation des Segmentes mit dem Fenster der selben Länge, ist die Fensterung durchgeführt [Mey17, S. 188 f.]. Darauf wird nun das Rauschprofil erstellt, welches als *NumPy*-Array für die weitere Nutzung in einer Datei abgespeichert wird.

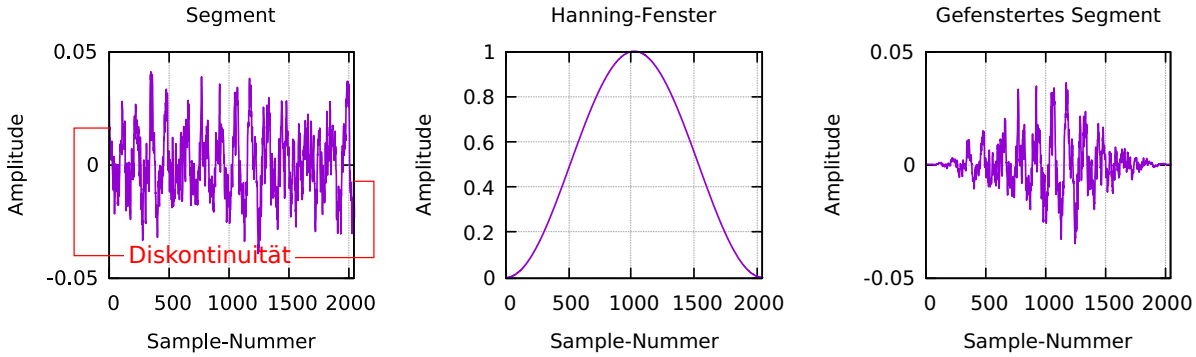

Abbildung 7.7: Fensterung eines Segments

Darüber hinaus wird das Profil in einem Diagramm angezeigt. Ein Ausschnitt daraus ist in Abbildung 7.8 zu sehen. Die Betrachtung aller einzelnen Kanäle führt zur Erkenntnis, dass das Mikrofon vorne rechts (Kanal eins) bis etwa 500 Hz extrem mit Störgeräuschen belastet wird. Bei den anderen Kanälen gibt es an dieser Stelle auch einen Peak, welcher jedoch wesentlich schwächer ist, obwohl die Mikrofone zwei und drei sich näher am Lüfter befinden. Die Schallausbreitung von einem Lüfter scheint an der Position besonders ungünstig zu sein. Der bisher für die Bewertung verwendete Kanal Nummer null ist daher eine gute Wahl.

7.2.2 Spektrale Subtraktion

Unter der Annahme, dass sich das Signal aus einer Addition von Nutzsignal und Rauschen zusammensetzt, kann das Nutzsignal extrahiert werden, indem das Rauschen vom Signal

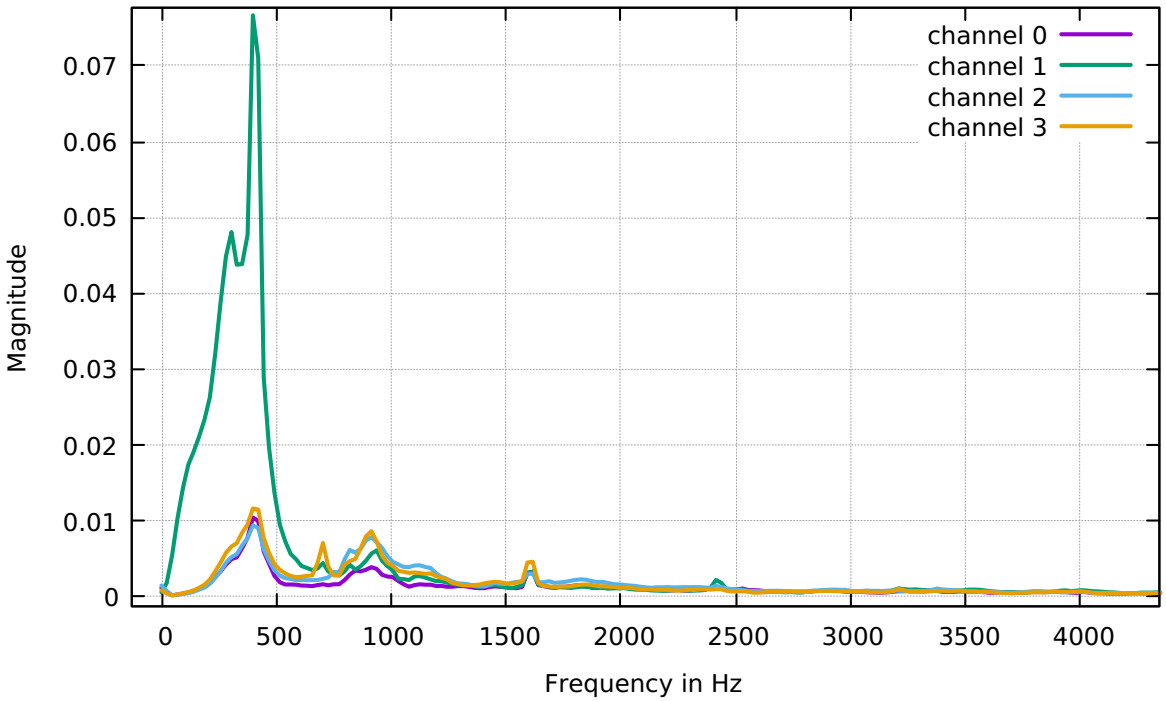

Abbildung 7.8: Ausschnitt des Rauschprofils

abgezogen wird. Bei der spektralen Subtraktion wird diese Operation im Frequenzbereich durchgeführt. Die Grundidee ist dabei, das Signal mittels der *DFT* in den Frequenzbereich zu transformieren, das Rauschprofil vom Spektrum zu subtrahieren und anschließend mittels der *Inverse-Discrete-Fourier-Transform (IDFT)* wieder in ein Signal im Zeitbereich zu synthetisieren. Wird die bei der Erstellung des Rauschprofils angewendete Vorgehensweise mit der Fensterung aus bekannten Gründen übernommen, entsteht ein neues Problem. Denn hier wird das Spektrum nicht allein für Analyse-Zwecke benutzt, sondern muss nach der Veränderung wieder zurücktransformiert werden, wodurch die Fensterform im Ergebnis erhalten bleibt. Der resultierende Effekt kann praktisch als Amplitudenmodulation bezeichnet werden und ist hier nicht gewollt. Eine Fensterung kann, obwohl sie nur eine Multiplikation ist, nicht rückgängig gemacht werden, da die Fensterfunktion Nullwerte enthält, die aber so erwünscht sind.

Durch das Verarbeiten von überlappenden Bereichen kann dieses Problem umgangen werden. Wie viele andere Fensterfunktionen auch, hat das *Hanning*-Fenster die Eigenschaft, dass sich überlappende Bereiche von Nachbarfenstern bei einem richtig gewählten Versatz zu eins aufaddieren. Beim *Hanning*-Fenster ist das bei einem Überlappungsgrad von 50 % der Fall (siehe Abbildung 7.9).

Angewendet auf die Verarbeitung von aufeinander folgenden Segmenten, müssen diese vor einer versetzten Fensterung in halbierten Einheiten neu zusammengesetzt werden. Abbildung 7.10 visualisiert diesen Vorgang. Die blauen Bereiche repräsentieren das Signal in voller Amplitude. Eine Fensterung wird durch einen Farbverlauf nach Weiß angedeutet, da Weiß für Samples mit Nullwerten stehen. Zusammengehörige Segmente (Rechtecke) werden an einem Stück verarbeitet. Durch die angewendete Fensterung kann das auch problemlos im Spektralbereich geschehen. Dabei muss bei der ersten Überlappung ein Halbsegment mit Nullwerten angefügt werden. Nach der Addition zum Schluss ist das

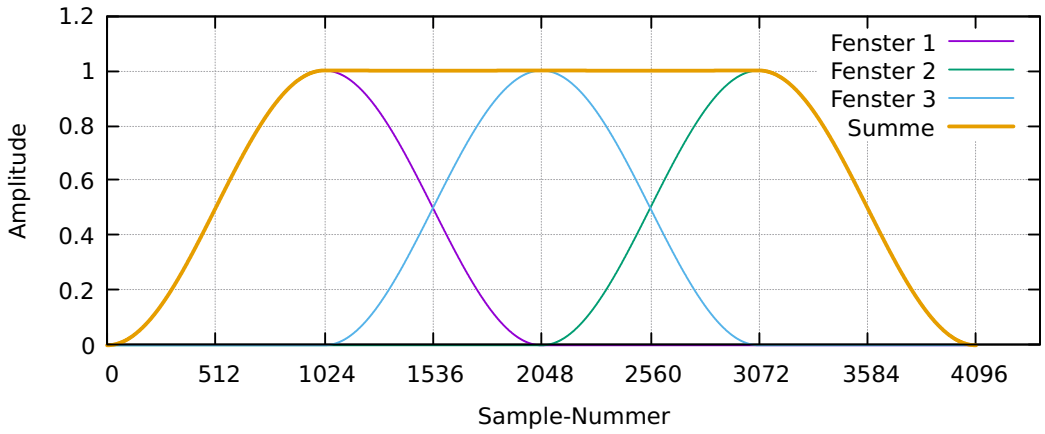

Abbildung 7.9: Summe von überlappenden *Hanning*-Fenstern

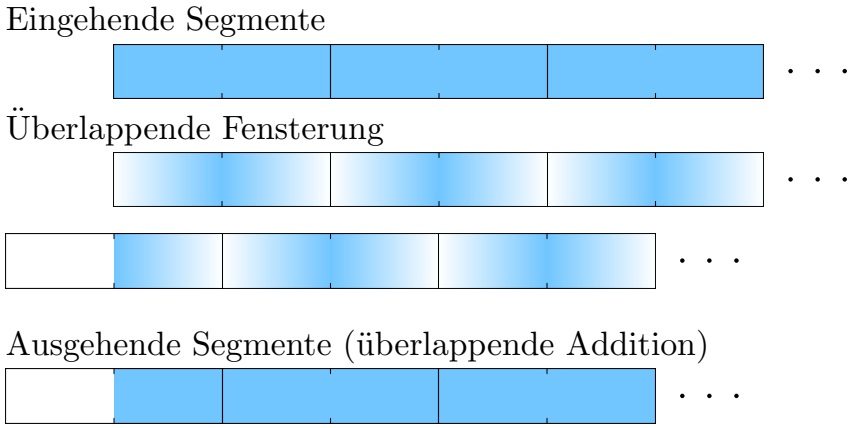

Abbildung 7.10: Überlappende Fensterung

Signal bzw. das verarbeitete Signal wieder in voller Amplitude zusammengefügt. Auf diese Weise haben die Daten der resultierenden Segmente einen Versatz von einer halben Segmentlänge. Dies könnte auch anders gelöst werden mit einem Versatz nach vorn. Jedoch müsste immer auf das Eintreffen des Segmentes gewartet werden, bis das aktuelle verarbeitet und weitergesendet werden kann. Ein Nachteil dieser ganzen Methodik ist, dass sich die zu verarbeitende Datenmenge fast verdoppelt.

Implementiert wurden diese und weitere Funktionalitäten einschließlich des Einlesens des Rauschprofils in der Klasse **`SoundProcessorSpectral`**, die von **`SoundProcessor`** erbt. So müssen Unterklassen, die eine spektrale Verarbeitung implementieren, lediglich eine Methode zur Veränderung der Amplituden überschreiben. Die Phasenwerte werden vom eingehenden Signal unverändert übernommen, sodass am Ende eine Umformung der Polarkoordinaten in kartesische Koordinaten stattfinden kann, die als Eingabedaten der *IDFT* dienen, welche mit der *Inverse-Fast-Fourier-Transform (IFFT)* berechnet wird. Das Ganze funktioniert aber nur, wenn die Segmentlänge passend zum genutzten Rauschprofil ist. Da sich die Puffergröße in *RoSaSS* variabel einstellen lässt und lassen soll, muss darauf reagiert werden können. Ist die Puffergröße kleiner, wird nach der Fensterung der Rest mit Nullen aufgefüllt, die nach der Verarbeitung wieder entfernt werden. So geht die Rechnung zwar auf, aber das Resultat ist unter Umständen schlechter, da weniger Informationen in die Transformation zum Spektrum einfließen. Bei größerer Puffergröße wird

der Puffer in gleichgroße Segmente zerlegt, die einzeln verarbeitet und am Ende wieder zu einem Puffer der ursprünglichen Größe zusammengesetzt werden. Falls das die Puffergröße nicht zulässt oder zu kleine Segmente dabei entstehen, wird ein Fehler geworfen. Eine Größenänderung kann auch zur Laufzeit stattfinden, wodurch der Zustand neu initialisiert wird und wieder mit einer halben Segmentlänge aus Nullwerten begonnen wird. Durch diese Mechanismen bleibt die ursprüngliche Puffergröße erhalten und jede Puffernachricht kann einzeln verarbeitet werden, ohne auf weitere Nachrichten warten zu müssen. Dabei funktioniert eine Puffergröße, die mit der Profilgröße (hier 2048) oder dem Vielfachen davon übereinstimmt am besten. Bezüglich der Kanalauswahl können nicht mehr Kanäle verarbeitet werden, als das Rauschprofil enthält. Eingeschränkt ist auch die Abtastrate, die genau mit der des Rauschprofils übereinstimmen muss. Sollen andere Abtastraten verwendet werden, kann entweder ein entsprechendes Rauschprofil generiert werden[54] oder aber der Knoten wird von **`SoundProcessorResample`**-Knoten umrahmt.

Der Knoten der Klasse **`SoundProcessorSpectralSubstraction`** führt die spektrale Subtraktion des Rauschprofils vom Eingangssignal aus und orientiert sich dabei zum Teil an [Vih+16, S. 667–670]. Da manche Frequenzanteile auch leiser sein können als im Rauschprofil, entstehen bei der Subtraktion negative Amplituden. Diese werden auf null gesetzt. Darüber hinaus wird die Gesamtlautstärke verringert, wenn sich das Signal nicht deutlich vom Rauschen abhebt. Diese Schwelle lässt sich mit dem Parameter **`silence_threshold`** einstellen. Wie leise das ist, bestimmt der Faktor **`silence_factor`**, welcher die Amplitudenwerte multipliziert. Ein abruptes Leisestellen führt je nach Konfiguration der Schwelle zum „Verschlucken“ von leisen Lauten in der Sprache. Aus diesem Grund wird stattdessen die Lautstärke langsam ausgeblendet. Wie schnell das von statten geht, spezifiziert der Parameter **`fade_speed`**.

Hört man sich das Resultat an, ist das Rauschen tatsächlich leiser. Doch leider handelt man sich dabei Artefakte ein, welche die Aufgabe der *ASR*-Engines nicht vereinfachen. Mit Unterstützung der Parameteroptimierung von **`silence_threshold`** und **`fade_speed`** konnten für *PocketSphinx* und *Kaldi* die Werte in Tabelle 7.1 bestimmt werden. Gegenüber dem unverarbeiteten Signal hat sich die *WER* bei *PocketSphinx* um 0,7 Prozentpunkte verbessert. Bei *Kaldi* hat sich die Bewertung dagegen nicht messbar verändert. Das Vorgehen sah so aus, dass zunächst manuell die Parameter so konfiguriert wurden, dass sich das Ergebnis subjektiv am besten anhört. Diese Werte wurden als Startzustand der Optimierung übernommen, welche dann die Feinjustierung durchführte. Nach mehreren Durchläufen, die zu verschiedenen Konfigurationen führten, wurden am Ende die Parameterwerte nach einem Kaltstart der *ASR*-Engines noch einmal bewertet. Die besten Ergebnisse sind dann ausgewählt worden.

Tabelle 7.1: Parameterwerte und *WER* bei Anwendung der spektralen Subtraktion

ASR-Engine	`silence_threshold`	`silence_factor`	`fade_speed`	*WER* in %
PocketSphinx	-11,84	0,05	1,35	93,8 (-0,7)
Kaldi	-10,53	0,05	0,21	98,4 (-0,0)

Ein Blick auf das resultierende Spektrogramm in Abbildung 7.11 mit den Einstellungen bei *PocketSphinx* zeigt im Vergleich zu Abbildung 7.2 deutliche Veränderungen. Es ist klar zu erkennen, dass unter dem Schwellwert liegende Zeitabschnitte stark in ihrer Amplitude

[54] Die Abtastrate wird durch die einzulesende Audiodatei vorgegeben.

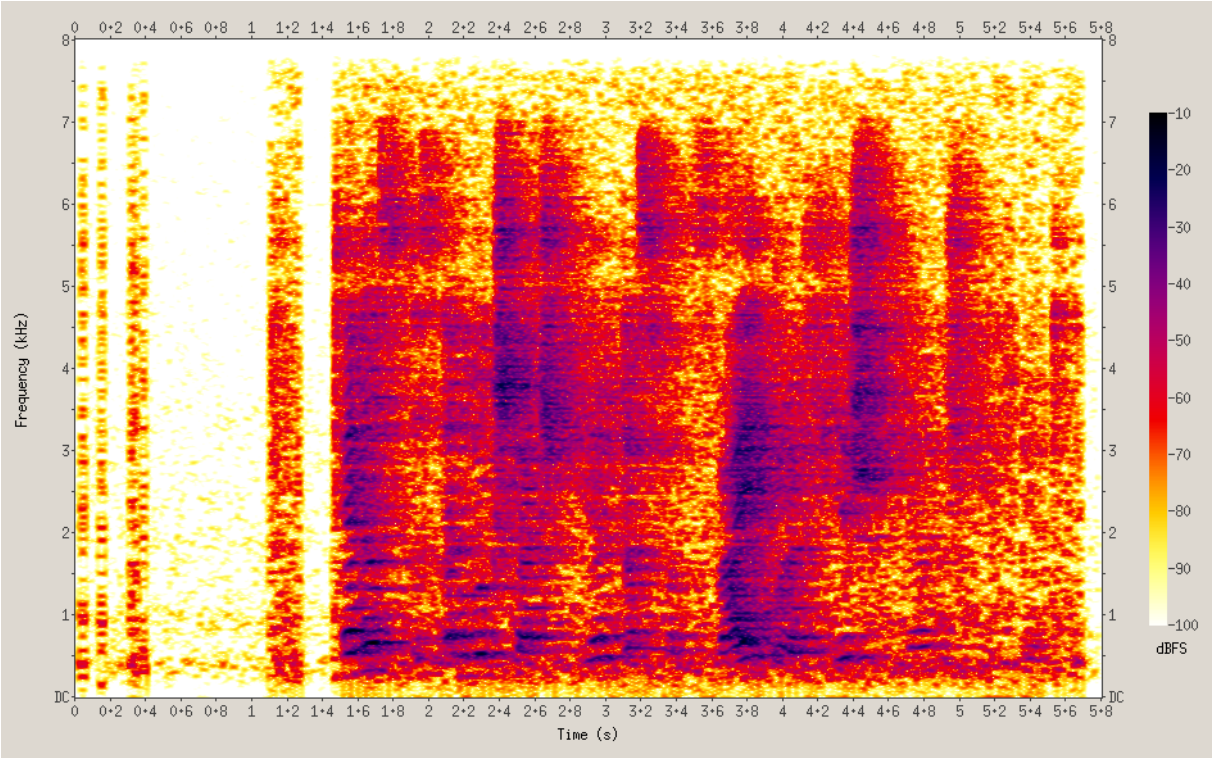

Abbildung 7.11: Spektrogramm nach der Anwendung der spektralen Subtraktion

verringert werden. Dabei kommt es am Anfang häufig zu der Situation, dass Hintergrundgeräsche über der Schwelle liegen und daher nicht unterdrückt werden. Im Allgemeinen hat das Rauschen zwar abgenommen, aber es bleiben Flocken übrig. Es ist über den Zeit- und Frequenzdimensionen grobkörniger geworden, was sich unnatürlich anhört. Die Details der Sprache scheinen aber im Wesentlichen erhalten zu bleiben.

7.2.3 Spektrales Gate

In der Audioverarbeitung wendet die Methode des Gatings eine Unterdrückung von leisen Abschnitten an, ähnlich zu der implementierten Zusatzfunktionalität bei der spektralen Subtraktion. Beim spektralen Gate wird hingegen jeder Frequenzbestandteil für sich betrachtet. Erst wenn dieser eine höhere Amplitude als im Rauschprofil aufweist, bleibt er unverändert erhalten [Sam+15]. Im anderen Fall wird die Amplitude reduziert.

Dieses Verfahren konnte mit wenig Aufwand in `SoundProcessorSpectralGate` implementiert werden, da durch ein Erben von `SoundProcessorSpectral` die gesamte Funktionalität für eine spektrale Verarbeitung zur Verfügung steht. Als Einstellmöglichkeiten existieren die Parameter `threshold_factor` und `reduction_factor`. Letzterer gibt an, wie stark die Amplituden reduziert werden sollen. Dabei wirkt das Rauschprofil als Schwelle, die mit `threshold_factor` durch Multiplikation verändert werden kann.

Tabelle 7.2 zeigt die durch die Optimierung bestimmten Parameterwerte und die Bewertungsergebnisse. Das spektrale Gate ist verglichen mit der spektralen Subtraktion etwas effektiver. Auch hier zeigt *PocketSphinx* eine stärkere Verbesserung. Im Spektrogramm (Abbildung 7.12) ist eine deutliche Rauschreduzierung zu sehen, während die lautesten

Tabelle 7.2: Parameterwerte und *WER* bei Anwendung des spektralen Gates

ASR-Engine	`threshold_factor`	`reduction_factor`	*WER* in %
PocketSphinx	2,20	-0,60	90,6 (-3,9)
Kaldi	2,66	-0,08	97,7 (-0,7)

Anteile der Sprache unangetastet bleiben. Die Übergänge zwischen Nutzsignal und Hintergrund sind sowohl in der Zeitdimension als auch in der Frequenzdimension sehr hart. Viele vereinzelte übriggebliebene Blöcke sind als kurze und störende Töne hörbar.

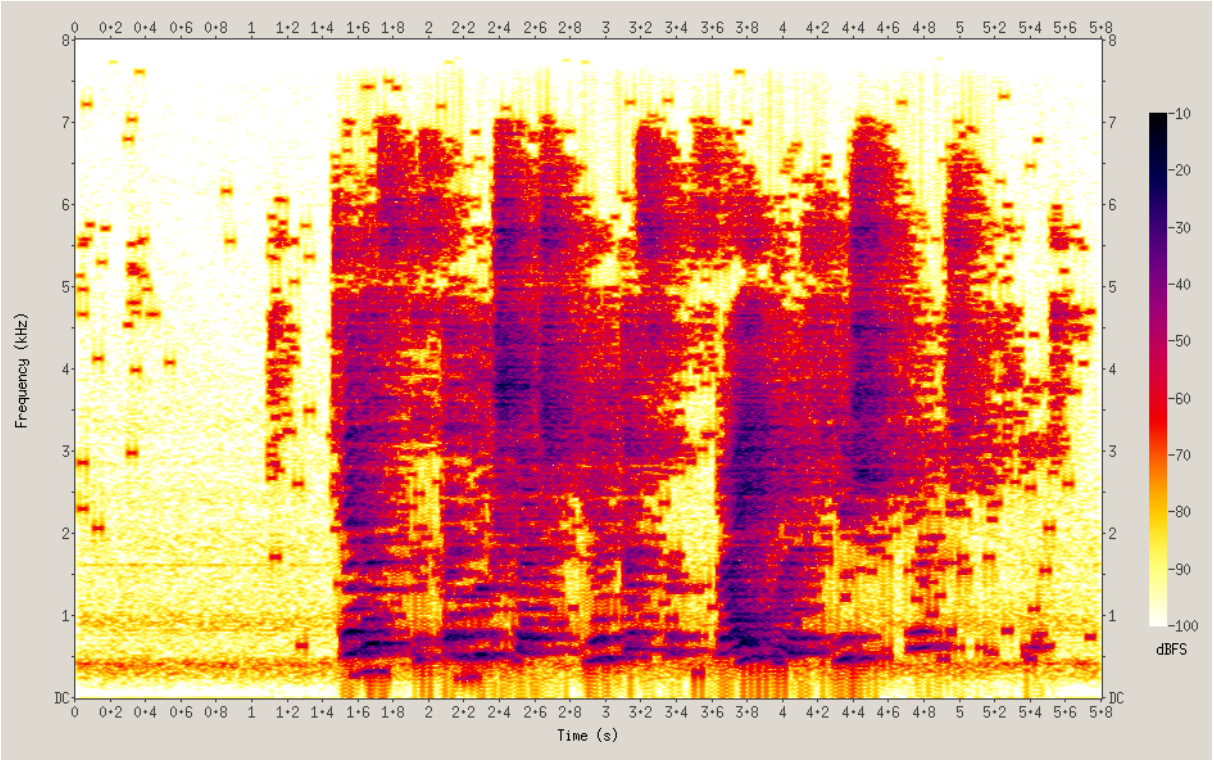

Abbildung 7.12: Spektrogramm nach der Anwendung des spektralen Gates

7.3 Beamforming

Während bei den einfachen Rauschreduzierungsverfahren die Daten einzelner Kanäle verarbeitet werden, lassen sogenannte Beamforming-Verfahren alle Kanäle eines Mikrofon-Arrays in die Berechnung einfließen. Als Resultat entsteht ein einkanäliges Audiosignal. Ein solches Verfahren kommt auch auf der internen Verarbeitung des *ReSpeaker* zur Anwendung. Die primitivste Form ist der *Delay-and-Sum*-Beamformer (siehe Abbildung 7.13). Dieser richtet die einzelnen Kanäle zeitlich zueinander aus und bildet eine Summe oder das arithmetische Mittel aller übereinanderliegenden Samples. Der zeitliche Versatz wird dabei aus den Positionen der Mikrofone und der gewünschten „Blickrichtung“ bestimmt. Im Endeffekt werden Signalanteile aus der richtigen Richtung durch die passgenaue Überlagerung verstärkt, wobei Signale aus anderen Richtungen sich versetzt überlagern und dadurch das Endergebnis weniger beeinflussen.

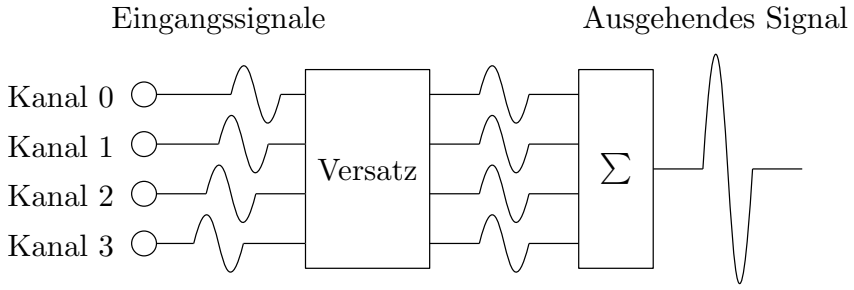

Abbildung 7.13: Grundprinzip des *Delay-and-Sum*-Beamformers

Implementiert ist das Verfahren in der Klasse **`SoundProcessorBeamform`**. Neben der Kanalauswahl hat der Knoten den einzigen Parameter **`time_shift`**. Mit der Angabe eines Arrays aus Gleitkommazahlen wird der Versatz (Verzögerung) pro Kanal in Millisekunden eingestellt, welcher auf Grundlage der Abtastrate in ganzzahlige Sample-Breiten umgerechnet wird. Auf diese Weise ist das Beamforming (*SSS*) von der automatischen Winkelbestimmung zur Schallquelle (*SSL*) separiert, da dieser Parameter nicht nur manuell sondern auch von anderen Knoten gesetzt werden kann. Diese Knoten existieren in *RoSaSS* jedoch noch nicht.

Für das Anwenden auf die Aufnahmen von *Peppers* Mikrofon-Array wurde ein Versatz von `[0.169, 0.169, 0.0, 0.0]` gewählt. Das entspricht einer Richtwirkung nach vorn, wo sich der Lautsprecher befand. Genau genommen war dieser gegenüber den Mikrofonen noch leicht erhöht. Aber für diesen Test ist die vereinfachte Annahme ausreichend. Die vorderen Mikrofone (Index null und eins) haben einen Abstand von 5,8 cm zu den hinteren Mikrofonen. Bei einer Schallgeschwindigkeit von $343\,\mathrm{m\,s^{-1}}$ ergibt das nach Formel (7.1) einen Versatz von 0,169 ms. Um diese zeitliche Länge müssen die Signale der ersten beiden Kanäle verzögert werden, damit die gewünschten Überlagerungen mit den Signalen der hinteren Mikrofone entstehen. Negative Verzögerungen sind nicht möglich, da dafür Daten aus nachfolgenden Audiopuffern bekannt sein müssten.

$$t = \frac{s}{c} = \frac{0{,}058\,\mathrm{cm}}{343\,\mathrm{m\,s^{-1}}} \approx 0{,}000169\,\mathrm{s} = 0{,}169\,\mathrm{ms} \tag{7.1}$$

Beim Anhören des Ergebnisses sind kaum Unterschiede zu erkennen. Vergleicht man jedoch das Spektrogramm (Abbildung 7.14) mit Abbildung 7.2, können visuell Veränderungen festgestellt werden. Das Rauschen ist etwa ab 1,5 kHz ein wenig leiser geworden und das besonders laute Lüftergeräusch von Kanal eins hat seine Spuren im unteren Frequenzbereich hinterlassen. Im Allgemeinen scheinen die Echo-Effekte leicht abgenommen zu haben. Leider sind aber manche Details der Sprache schwieriger auf dem Bild zu erkennen. Das trifft besonders den Bereich zwischen 1 und 2,5 kHz. Nach dem Bewertungsprozess konnte *PocketSphinx* den Text mit einer *WER* von 93,8 % und *Kaldi* mit 94,5 % transkribieren. Das entspricht einer Verbesserung um 0,7 bzw. 3,9 Prozentpunkte. Dieses Mal spricht *Kaldi* besser auf die Vorverarbeitung an.

Neben dem *Delay-and-Sum*-Beamformer gibt es eine Vielzahl an weiteren wesentlich effektiveren Varianten, welche eine sehr starke Richtwirkung über einen großen Frequenzbereich erreichen. Für deren Implementierung bzw. Einbindung von bestehenden Implementierungen war die restliche Bearbeitungszeit zu kurz. Auch bei einem gut funktionierenden

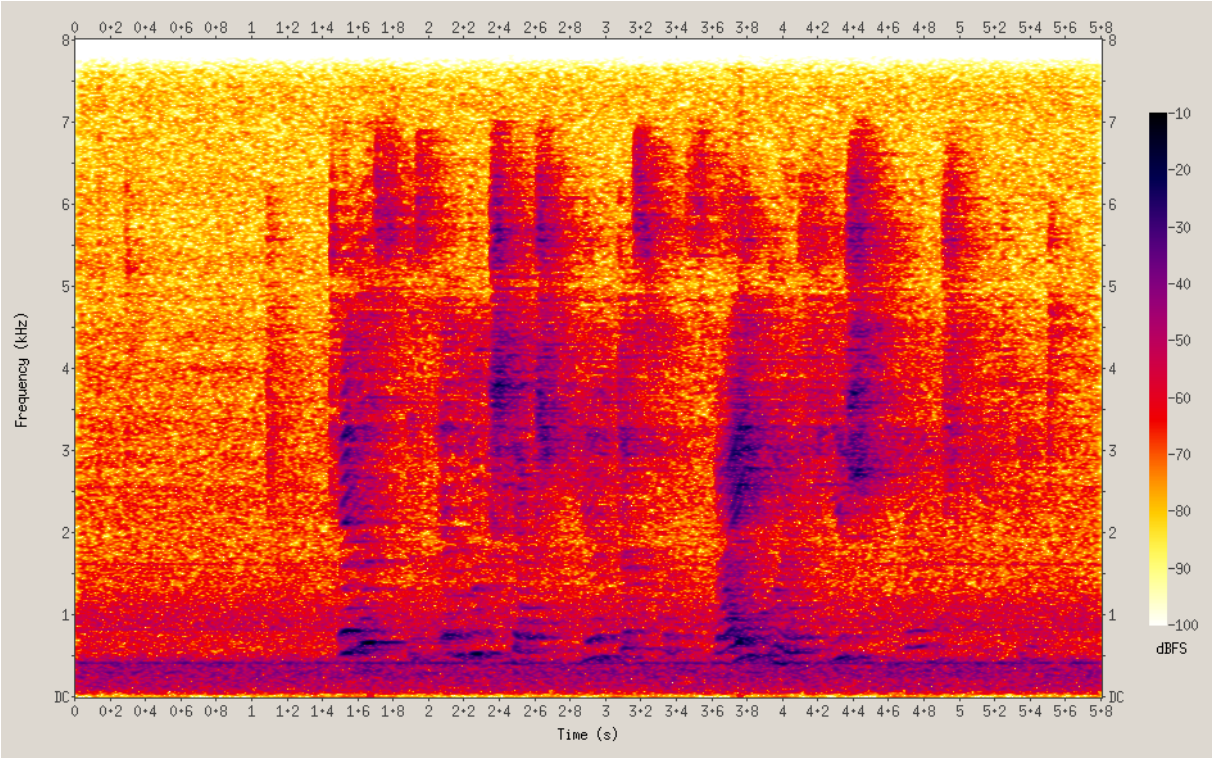

Abbildung 7.14: Spektrogramm nach der Anwendung des *Delay-and-Sum*-Beamformers

Beamformer könnte die Tatsache zum Problem werden, dass die Lüftergeräusche durch die Ausbreitung unter der Gehäuseschale aus vielen Richtungen eintreffen.

7.4 Zusammenfassung

Alles im allen können die hier implementierten und getesteten Verfahren zur Aufbereitung des Audiosignals beim *Pepper*-Roboter nicht viel ausrichten. Immerhin sind leichte Senkungen der Wortfehlerrate messbar (siehe Tabelle 7.3), was aber nicht unbedingt heißt, dass die Situation verbessert wurde.

Tabelle 7.3: Zusammenfassung der Bewertungsergebnisse bei allen Verfahren

Verfahren	*WER* in % bei *PocketSphinx*	*WER* in % bei *Kaldi*
kein Verfahren	94,5	98,4
spektrale Subtraktion	93,8	98,4
spektrales Gate	90,6	97.7
Delay-and-Sum-Beamformer	93,8	94,5

Besonders bei den Rauschreduzierungsverfahren ist aufgefallen, dass viele falsche Wörter erkannt wurden, wo sonst die *ASR*-Engines nichts zurück geliefert hätten. Dadurch wird im Endeffekt bei den nachgelagerten Schritten zum *NLU* mehr Schaden verursacht, als wenn einige Wörter nicht erkannt werden würden. Die Bewertung könnte mehr in die richtige Richtung weisen, wenn bei der Editierdistanzberechnung Ersetzungen und Einfügungen stärker gewichtet werden würden als Löschungen.

Neben den hier verhältnismäßig einfachen und ausbaufähigen Varianten der Verfahren gibt es reifere Implementierungen, von denen signifikante Verbesserungen zu erwarten sind. Deren Einbindung und Bewertung in *RoSaSS* konnte im zur Verfügung stehenden Zeitrahmen leider nicht mehr realisiert werden.

8 Fazit und Ausblick

Die Ziele dieser Arbeit waren die Erschaffung einer Softwareplattform zur Handhabung von digitalen Audiosignalen, automatischer Spracherkennung und Signalverarbeitung sowie das Integrieren von Verfahren zur Verbesserung der Spracherkennung des *Pepper*-Roboters. Das erste Ziel kann als erreicht eingestuft werden. Denn mit *RoSaSS* wurde eine anforderungsgerechte Implementierung geschaffen, welche die Situation stark aufwertet. Auch wenn noch Kleinigkeiten verbessert oder erweitert werden können, ist *RoSaSS* eine gute Grundlage für Experimente oder auch Live-Anwendungen.

Für das Erreichen des zweiten Ziels wurden Fortschritte gemacht. Aufgrund des beschränkten Zeitrahmens konnten aber nur wenige Verfahren in einer einfachen Form implementiert und getestet werden. Zu einer signifikanten Verbesserung der Wortfehlerrate kam es damit nicht. Angedacht war und ist noch die Nutzung und Evaluierung von *noise-repellent*[55] und *speech-denoiser*[56]. Bei beiden handelt es sich um Rauschreduzierungsprogramme im *LV2*-Plug-in-Format. Eine solche Schnittstelle ist in *RoSaSS* bereits implementiert, weshalb eine zukünftige Integration ohne Quellcodeänderungen zu erwarten ist. Diese kam jedoch noch nicht zur Anwendung. Darüber hinaus ist die Verwendung von der *Open embeddeD Audition System (ODAS)*[57]-Bibliothek geplant, welche Algorithmen rund um das Thema Mikrofon-Arrays bereitstellt.

Weitere Experimente mit besseren Lautsprechern, einer höheren Anzahl an Aufnahmen, zusätzlichen künstlichen Störquellen und das in verschiedenen Räumlichkeiten sowie Abständen sind denkbar. Für eine fertige Anwendung beim Alltagsbetrieb des *Pepper* Roboters fehlen noch Komponenten wie die *VAD*, Schlüsselworterkennung und das Herausfiltern der eigenen Lautsprecherausgaben. Durch die Verwendung von *ROS2* als Unterbau stehen viele Möglichkeiten bei robotischen Systemen offen, weitere Datenquellen zur Aufbereitung der Audiosignale einzubeziehen. Denn bei der Unterstützung mit einer Datenauswertung von Kamerabildern können Nutzer bezüglich ihrer Lage und Gesprächsbereitschaft auch optisch observiert werden. Außerdem können mit Kenntnis der Steuerbefehle für Gelenkmotoren und Fahrwerk zusätzliche Maßnahmen zum Unterdrücken dieser Geräuschquellen auf den Aufnahmen getroffen werden. Diese und viele andere Visionen sind zum Teil aber eher in einer etwas ferneren Zukunft angesiedelt.

Abschließend kann angemerkt werden, dass in der Summe viel erreicht wurde und der Autor dabei eine Menge dazu gelernt hat.

[55] *noise-repellent*: `https://github.com/lucianodato/noise-repellent` (besucht am 09.02.2020)
[56] *speech-denoiser*: `https://github.com/lucianodato/speech-denoiser` (besucht am 09.02.2020)
[57] *ODAS*: `https://github.com/introlab/odas`

Abkürzungen

AEC Acoustic-Echo-Cancellation 12, 13

ALSA *Advanced Linux Sound Architecture* 7, 24, 25, 28, 84

ASR Automatic-Speech-Recognition 1, 3, 4, 7–9, 11, 14–17, 19, 21, 22, 27, 29, 35, 40, 42–45, 48, 49, 53–56, 58–60, 62, 68, 74, 76, 78, 92

AU Audio Unit 25

CLI Command-Line-Interface 15, 16, 25, 32

dBFS Decibels relative to full scale 64

DFT Discrete-Fourier-Transform 70–72

DSP Digital-Signal-Processing 54

FFT Fast-Fourier-Transform 70

FLAC Free Lossless Audio Codec 26, 27

GNU GNU's Not Unix 83

IDFT Inverse-Discrete-Fourier-Transform 72, 73

IFFT Inverse-Fast-Fourier-Transform 73

iPOPO injected Plain Old *Python* Object 24, 27

JACK JACK Audio Connection Kit 25–27

LADSPA Linux Audio Developer's Simple Plugin API 25, 26

LV2 *LADSPA Version 2* 26, 29, 41, 80

MIDI Musical Instrument Digital Interface 26

NLG Natural-Language-Generation 4

NLU Natural-Language-Understanding 4, 78

ODAS Open embeddeD Audition System 80

OSGi Open Services Gateway initiative 23, 24, 27

Pyro4 *Python* Remote Objects 4 43, 44

QoS Quality-of-Service 24, 36

RMW *ROS* Middleware 33, 36

ROS Robot Operating System 24, 27, 31–33, 82

ROS2 *Robot Operating System 2* 24, 27, 29–33, 36, 37, 39–41, 43, 44, 47, 51–53, 66, 80, 85

RoSaSS Robot Sound and Speech System 2, 17–36, 39–41, 44, 45, 53, 60, 73, 77, 79, 80, 90

ROSIDL *ROS* Interface Definition Language 32, 51

SSL Sound-Source-Localization 8–10, 12, 13, 77

SSS Sound-Source-Seperation 9, 10, 77

STT Speech-To-Text 42

TSV Tab-separated values 50

URI Uniform Resource Identifier 41, 42, 90

USB Universal Serial Bus 11–13, 52, 85

VAD Voice-Activity-Detection 4, 9, 44, 59, 80

VST Virtual Studio Technology 25

WER Word-Error-Rate 55, 57, 59–61, 66–68, 74, 76–78, 92

XML eXtensible Markup Language 48, 49

YAML YAML Ain't Markup Language 66

Glossar

Advanced Linux Sound Architecture

Dieses Kernelmodul (kurz *ALSA*[58]) sorgt unter *GNU/Linux* mit Hilfe von Treibern für die Steuerung von Audiogeräten und den Zugriff auf Audiodaten. 7, 81, 85

DeepSpeech

DeepSpeech[59] ist ein Toolkit für automatische Spracherkennung. 15, 16, 43, 60, 61

GNU/Linux

GNU/Linux bezeichnet eine Familie von unixartigen Betriebssystemen, welche auf dem *Linux*-Kernel[60] und Softwarepaketen des *GNU's Not Unix (GNU)*-Projekts[61] basieren. 19, 24–26, 28, 83–85

Kaldi

Kaldi[62] ist ein Toolkit für automatische Spracherkennung. 15, 16, 43, 54, 58–61, 74, 76–78

LADSPA Version 2

LADSPA Version 2[63] (kurz *LV2*) ist ein plattformübergreifendes Audio-Plug-in-Format, welches aber hauptsächlich unter *GNU/Linux* zur Anwendung kommt. 26, 81

macOS

macOS ist ein unixbasiertes Betriebssystem von *Apple*. 19, 22, 24, 25, 27

NAO

NAO ist die Typenbezeichnung eines humanoiden Roboters vom Hersteller *SoftBank Robotics* [65]. 1, 10, 23, 27, 84

[58] *ALSA*: `https://www.alsa-project.org` (besucht am 23.01.2020)
[59] *DeepSpeech*: `https://github.com/mozilla/DeepSpeech` (besucht am 23.01.2020)
[60] *Linux*: `https://www.kernel.org/linux.html` (besucht am 23.01.2020)
[61] *GNU*: `https://www.gnu.org/` (besucht am 23.01.2020)
[62] *Kaldi*: `http://kaldi-asr.org/` (besucht am 23.01.2020)
[63] *LV2*: `https://lv2plug.in/` (besucht am 23.01.2020)
[63] *macOS*: `https://www.apple.com/de/macos` (besucht am 23.01.2020)

NAOqi

NAOqi ist eine properitäre Software zur Steuerung von *NAO*- und *Pepper*-Robotern. Durch das modulare Design ist eine Erweiterung durch selbst entwickelte Software möglich. 6–10, 17–19, 22–24, 27–29, 31, 40, 43, 44, 60, 84–86, 90

NAOqi API

Die *NAOqi API* umfasst die Programmierschnittstellen aller vorinstallierten Module in *NAOqi*. 3, 6, 18

NAOqi OS

Das auf *Gentoo* basierende *GNU/Linux* namens *NAOqi OS* ist das Betriebsystem für *NAO* und *Pepper*. 6, 22, 25, 31

NumPy

Die *Python*-Bibliothek *NumPy*[64] stellt effiziente Implementierungen zur numerischen Verarbeitung von Arrays und mehrdimensionalen Matrizen bereit. 29, 35, 41, 70, 71

Pepper

Pepper ist die Typenbezeichnung eines humanoiden Roboters vom Hersteller *SoftBank Robotics*[65]. Im Unterschied zum *NAO* ist dieser größer, besitzt ein Tablet zur Bedienung und bewegt sich anstelle von Beinen mit einem omnidirektionalen Fahrwerk fort. 1, 3–6, 8–10, 12, 13, 18–23, 27, 28, 30–32, 39, 43, 51–53, 60, 62, 64, 65, 77, 78, 80, 84, 91

PocketSphinx

PocketSphinx[66] ist eine Spracherkennungssoftware. 15, 16, 43, 59, 61, 74–78

PortAudio

Die Softwarebibliothek *PortAudio*[67] abstrahiert den Audiozugriff über verschiedene Schnittstellen (darunter auch *ALSA*), sodass sie plattformübergreifend genutzt werden kann. 25, 27, 28, 38, 39, 90

PRoot

Die Container-Virtualisierungslösung *PRoot*[68] erlaubt eine Ausführung ohne *root*-Privilegien. 32

PulseAudio

Der der unter *GNU/Linux* weit verbreitete Sound-Server *PulseAudio*[69] ermöglicht Programmen einen flexiblen Zugriff auf die Audioaufnahme und Wiedergabe und

[64] *NumPy*: `https://numpy.org/` (besucht am 23.01.2020)
[65] *SoftBank Robotics*: `https://www.softbankrobotics.com/emea/en` (besucht am 23.01.2020)
[66] *PocketSphinx*: `https://github.com/cmusphinx/pocketsphinx` (besucht am 23.01.2020)
[67] *PortAudio*: `http://www.portaudio.com/` (besucht am 23.01.2020)
[68] *PRoot*: `https://proot-me.github.io/` (besucht am 23.01.2020)
[69] *PulseAudio*: `https://www.freedesktop.org/wiki/Software/PulseAudio/About/` (besucht am 23.01.2020)

baut dabei auf verschiedene Backends, wie etwa *Advanced Linux Sound Architecture*, auf. 7, 25, 27, 28, 38–40, 45, 50, 52–54, 59, 90

Python

Bei *Python*[70] handelt es sich im eine dynamisch typisierte Interpretersprache. 16, 24, 27, 29, 30, 32, 33, 35, 38, 39, 41–43, 48, 49, 59, 68, 70, 81, 82, 84

ReSpeaker

Das *ReSpeaker Mic Array v2.0* ist ein Mikrofon-Array mit vier Mikrofonen, welches sich über *USB*-Audio nutzen lässt und auch eine integrierte Signalverarbeitung bereitstellt [See19]. 12, 13, 20, 51, 54, 60–62, 76, 90

Robot Operating System 2

Bei *Robot Operating System 2*[71] (kurz *ROS2*) handelt es sich nicht um ein Betriebssystem im eigentlichen Sinne sondern um eine Menge an quelloffenen Softwarebibliotheken und Werkzeugen, welche die Programmierung von Robotern vereinfachen. Es ist vergleichbar mit *NAOqi* und lässt sich auch durch Module erweitern. 24, 82

Topic

Eine *Topic* bezeichnet in *ROS2* einen Nachrichtenkanal über den *ROS2*-Knoten miteinander kommunizieren können. 29, 37, 40, 44, 45, 54

Tuda-De

Tuda-De[72] ist der Name eines Sprachkorpus, welcher an der *Technischen Universität Darmstandt* in Kooperation mit der *Universität Hamburg* aufgenommen wurde. 15, 48, 49, 56, 58, 61, 64

Ubuntu

Ubuntu[73] ist der Name einer weit verbreiteten *GNU/Linux*-Distribution. 19, 22, 24, 30, 32

UMA-8

Das Mikrofon-Array *UMA-8 v2* vom Hersteller *miniDSP* ist ein *USB*-Audio-Gerät, welches sieben Mikrofone sowie eine integrierte Signalverarbeitung besitzt [min18]. 12, 13, 20, 26, 54, 60, 62, 90

Unix Domain Socket

Ein *Unix Domain Socket* ist ein bidirektionaler Kommunikationskanal, welcher auf einem unixartigen Betriebsystem für die Interprozesskommunikation verwendet werden kann. Die Datenübertragung findet durch den Kernel statt. Adressiert wird dieser Socket in Form eines Pfades im Dateisystem [Wik18b]. 39, 44

[70] *Python*: `https://www.python.org/` (besucht am 23.01.2020)
[71] *Robot Operating System 2*: `https://index.ros.org/doc/ros2/` (besucht am 23.01.2020)
[72] *Tuda-De*-Korpus: `https://www.inf.uni-hamburg.de/en/inst/ab/lt/resources/data/acoustic-models.html` (besucht am 23.01.2020)
[73] *Ubuntu*: `https://ubuntu.com/` (besucht am 23.01.2020)

VoCon Hybrid

VoCon Hybrid[74] ist eine Spracherkennungssoftware, welche in *NAOqi* zum Einsatz kommt. 7, 9, 14, 15, 18, 29, 43, 44, 54, 56, 59–61, 90

WAVE

WAVE[75] ist ein etabliertes und weit verbreitetes Audiodateiformat. 7, 26, 27, 34–37, 48, 50, 54

Windows

Windows[76] ist der Name eines Betriebssystems von *Microsoft*. 19, 22, 24, 25, 27

[74] *VoCon Hybrid*: `https://www.nuance.com/mobile/speech-recognition-solutions/vocon-hybrid.html` (besucht am 23.01.2020)
[75] *WAVE*: `https://de.wikipedia.org/wiki/RIFF_WAVE` (besucht am 23.01.2020)
[76] *Windows*: `https://www.microsoft.com/en-us/windows` (besucht am 23.01.2020)

Quellenverzeichnis

[AZ19] Aashish Agarwal und Torsten Zesch. „German End-to-end Speech Recognition based on DeepSpeech“. In: *Preliminary proceedings of the 15th Conference on Natural Language Processing (KONVENS 2019): Long Papers*. Erlangen, Germany: German Society for Computational Linguistics & Language Technology, 2019, S. 111–119.

[Car18] Carnegie Mellon University. *About CMUSphinx — CMUSphinx Open Source Speech Recognition*. 2018. URL: `https://cmusphinx.github.io/wiki/about/` (besucht am 16. 10. 2019).

[Fer18] Esteve Fernandez. *Scripts for cross-compiling ROS and ROS2 for Softbank's Pepper*. 24. Apr. 2018. URL: `https://github.com/esteve/ros2_pepper` (besucht am 20. 01. 2020).

[Gam+04] Erich Gamma et al. *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. Addison Wesley Verlag, 2004. ISBN: 9783827321992.

[Gay17] Victor Gaydov. *PulseAudio under the hood*. 21. Sep. 2017. URL: `https://gavv.github.io/articles/pulseaudio-under-the-hood/` (besucht am 03. 10. 2019).

[Gre19] Greg H. *Python 2 series to be retired by April 2020*. 20. Dez. 2019. URL: `http://pyfound.blogspot.com/2019/12/python-2-sunset.html` (besucht am 29. 01. 2020).

[Kab17] Peter Kabal, Hrsg. *Audio File Format Specifications*. 2. Mai 2017. URL: `http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html` (besucht am 24. 01. 2020).

[Kna06] Geert Knapen, Hrsg. *Universal Serial Bus Device Class Definition for Audio Devices*. 31. Mai 2006. URL: `https://www.usb.org/sites/default/files/Audio2.0_final.zip` (besucht am 07. 10. 2019).

[Lét17] Dominic Létourneau. *8 Sounds USB*. IntRoLab of Université de Sherbrooke. 22. März 2017. URL: `https://sourceforge.net/p/eightsoundsusb/wiki/Main_Page/` (besucht am 09. 10. 2019).

[Lét18] Dominic Létourneau. *16SoundsUSB - 16 Synchronized Inputs USB (UAC2) Sound Card*. IntRoLab of Université de Sherbrooke. 16. Juli 2018. URL: `https://github.com/introlab/16SoundsUSB` (besucht am 09. 10. 2019).

[Mey17] Martin Meyer. *Signalverarbeitung*. Springer-Verlag GmbH, 24. Sep. 2017. ISBN: 3658183209.

[min18] miniDSP. *UMA-8 v2 — User Manual. USB MICROPHONE ARRAY WITH EMBEDDED DSP*. 14. Sep. 2018. URL: `https://www.minidsp.com/images/documents/UMA-8%20v2%20User%20manual.pdf` (besucht am 09. 10. 2019).

[MK18] Benjamin Milde und Arne Köhn. „Open Source Automatic Speech Recognition for German“. In: *CoRR* abs/1807.10311 (2018). arXiv: 1807.10311. URL: http://arxiv.org/abs/1807.10311.

[mvi19] mviereck. *Container sound: ALSA or Pulseaudio*. 16. Aug. 2019. URL: https://github.com/mviereck/x11docker/wiki/Container-sound:-ALSA-or-Pulseaudio (besucht am 26. 01. 2020).

[Ope19] Open Robotics, Hrsg. *Installing ROS 2 Dashing Diademata*. 1. Aug. 2019. URL: https://index.ros.org/doc/ros2/Installation/Dashing/ (besucht am 20. 01. 2020).

[Pra+18] Vineel Pratap et al. „wav2letter++: The Fastest Open-source Speech Recognition System“. In: *CoRR* abs/1812.07625 (2018). URL: https://arxiv.org/abs/1812.07625.

[Pul19] PulseAudio, Hrsg. *PulseAudio: Asynchronous API*. 13. Sep. 2019. URL: https://freedesktop.org/software/pulseaudio/doxygen/async.html (besucht am 20. 01. 2020).

[Rob19] Open Robotics, Hrsg. *Working with multiple ROS 2 middleware implementations*. 14. Nov. 2019. URL: https://index.ros.org/doc/ros2/Tutorials/Working-with-multiple-RMW-implementations/ (besucht am 25. 01. 2020).

[Sam+15] Peter Sampson et al. *How Audacity Noise Reduction Works*. 2015. URL: https://wiki.audacityteam.org/wiki/How_Audacity_Noise_Reduction_Works (besucht am 08. 02. 2020).

[Sch76] Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*. Birkhäuser Basel, 1. Jan. 1976. 400 S. ISBN: 3764308761.

[See19] Seed Studio. *ReSpeaker Mic Array v2.0 — Seed Wiki*. 25. Juni 2019. URL: http://wiki.seeedstudio.com/ReSpeaker_Mic_Array_v2.0/ (besucht am 09. 10. 2019).

[Smi97] Steven W. Smith. *The Scientist & Engineer's Guide to Digital Signal Processing*. California Technical Pub, 1997. ISBN: 0966017633.

[Sof19a] SoftBank Robotics Europe, Hrsg. *ALAudioDevice - Aldebaran 2.5.11.14a documentation*. 2019. URL: http://doc.aldebaran.com/2-5/naoqi/audio/alaudiodevice.html (besucht am 28. 09. 2019).

[Sof19b] SoftBank Robotics Europe, Hrsg. *ALDialog - Aldebaran 2.5.11.14a documentation*. 2019. URL: http://doc.aldebaran.com/2-5/naoqi/interaction/dialog/aldialog.html (besucht am 28. 09. 2019).

[Sof19c] SoftBank Robotics Europe, Hrsg. *NAOqi Audio - Aldebaran 2.5.11.14a documentation*. 2019. URL: http://doc.aldebaran.com/2-5/naoqi/audio/index.html (besucht am 28. 09. 2019).

[Sof19d] SoftBank Robotics Europe, Hrsg. *Supported languages - Aldebaran 2.5.11.14a documentation*. 2019. URL: http://doc.aldebaran.com/2-5/family/pepper_technical/languages_pep.html (besucht am 03. 10. 2019).

[Vih+16] Siddala Vihari et al. „Comparison of Speech Enhancement Algorithms“. In: *Procedia Computer Science* 89 (2016), S. 666–676. DOI: 10.1016/j.procs.2016.06.032.

[Wik18a] Wikipedia. *Levenshtein-Distanz — Wikipedia, Die freie Enzyklopädie*. 2018. URL: https://de.wikipedia.org/w/index.php?title=Levenshtein-Distanz&oldid=176864487 (besucht am 03. 02. 2020).

[Wik18b] Wikipedia. *POSIX local inter-process communication socket — Wikipedia, Die freie Enzyklopädie*. 2018. URL: https://de.wikipedia.org/w/index.php?title=POSIX_local_inter-process_communication_socket&oldid=179063453 (besucht am 29. 01. 2020).

[Wik19a] Wikipedia. *Menschliche Stimme — Wikipedia, Die freie Enzyklopädie*. 2019. URL: https://de.wikipedia.org/w/index.php?title=Menschliche_Stimme&oldid=192666130 (besucht am 08. 10. 2019).

[Wik19b] Wikipedia contributors. *Wagner–Fischer algorithm — Wikipedia, The Free Encyclopedia*. 2019. URL: https://en.wikipedia.org/w/index.php?title=Wagner%E2%80%93Fischer_algorithm&oldid=904154368 (besucht am 04. 02. 2020).

[Wol03] Klaus-Peter Wolf. *Tafelwerk interaktiv, Neue Bundesländer*. Volk und Wissen, 2003. ISBN: 9783464571477.

Abbildungsverzeichnis

Tabellenverzeichnis

Beilagenverzeichnis

Beilage 1

Diese Beilage ist eine DVD-R mit folgendem Inhalt:

- PDF-Dokument dieser Masterarbeit
- Quellcode der erarbeiteten Softwarelösung
- Quellcodedokumentation
- Installationsanleitung
- Datenbanken mit Metadaten des *Tuda-De*-Korpus
- Programm zum Erstellen dieser Datenbanken
- ausgewählte Audioaufnahmen aus dem *Tuda-De*-Korpus
- verwendete Testvorschriften
- eigene Audioaufnahmen
- verarbeitete Audioaufnahmen