MASTER THESIS
INTELLIGENT SYSTEMS

A CONFIGURABLE SPEECH RECOGNITION PIPELINE FOR SOCIAL ROBOTS

CREATING A FUSION FRAMEWORK FOR ANALYZING SPEECH DATA

ROBERT FELDHANS

Bielefeld University,
Faculty of Technology,
Central Lab Facilities

REVIEWED BY
DR.-ING. SVEN WACHSMUTH,
M.SC. FLORIAN LIER

SUPERVISED BY
DR.-ING. SVEN WACHSMUTH,
M.SC. FLORIAN LIER,
M.SC. BIRTE RICHTER

OCTOBER 16, 2019

CONTENTS

1 INTRODUCTION & MOTIVATION

In order to be accepted by humans, social robots in general depend on Human Robot Interaction (HRI) [MHB14], which in turn is influenced and shaped by human-human interaction [RN96]. Since speech is one of the most important modalities of communication between humans, it is no surprise that it is also one of the most important parts of HRI. In HRI, verbal dialogue can be divided into speech generation and the main interest of this thesis: speech understanding.

Being able to perfectly recognize the words a person spoke does not completely cover speech understanding. Also needed is the perception and engagement of the speaker [Iva+17]. An exemplary dialogue between a robot and two humans is depicted in a provided video[1] and in stills in figure 1.1. The video depicts parts of a RoboCup@home Task at the RoboCup World Championship 2018 in Montreal. In three different instances two referees are giving a robot commands, but they take turns in speaking to the robot. The robot does not recognize two different speakers, and thus, cannot react appropriately. More so: when asking for the confirmation of a specific command, it does not even seem to recognize that a different person gives this confirmation, or that the original referee walked away and no longer partakes in the conversation.

Therefore, the shown interactions appear unnatural: the robot does not seem to perceive the human, instead it is quite clear that it just listens for a particular combination of words. A similar phenomenon could very publicly be observed in the recent past, when Amazons Alexa ordered a variety of objects online, after hearing commands from a TV commercial[2]. The solution employed by Amazon to stop this from happening with its own TV commercials seems to be a purposefully altered audio signal[3] instead of a more general approach.

In the context of social robotics, these kind of behaviors are not desirable since approaches to partially solve these problems can be found in current literature [AA09]. Voice recognition technologies which can be used to differentiate speakers are available [MBE10]. Additionally, computer vision is used to search for speakers, either standalone, looking for moving lips, or in combination with sound source localization (see chapter 2.3) [Cho+02; CD00; GPR00; CLH19].

Suitable robot behaviors need to be created, to take all this information into account. When creating these behaviors, one must consider how the corresponding percepts are processed. The behavior in question must either be able to combine the information, e.g. a spoken utterance and a distinct, detected voice or receive this information in a pre-combined manner. If the behavior handles the combination, both may not be fed into the behavior at the same time, resulting in

1 https://www.youtube.com/watch?v=iEMKZdwJPE8
2 http://archive.is/zXuJu
3 http://archive.is/d3uVu

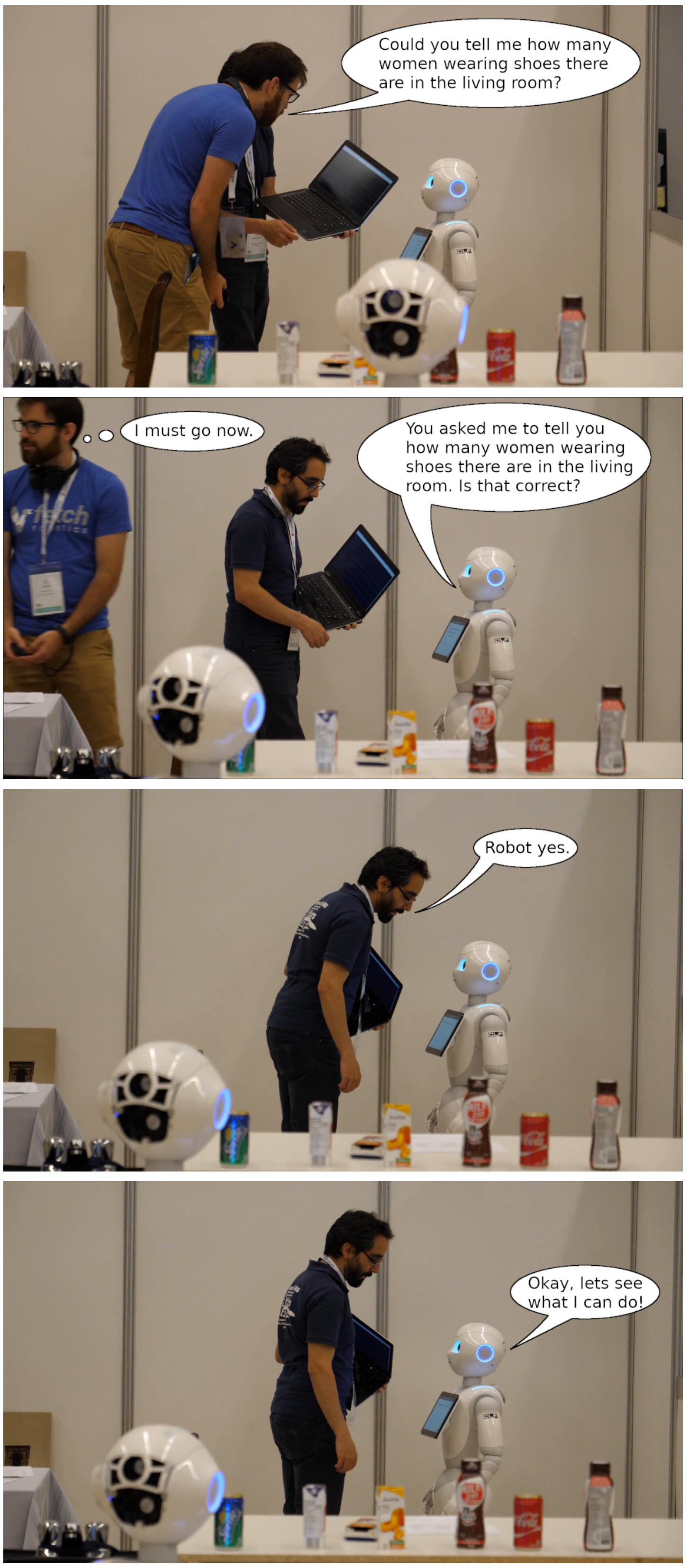
Could you tell me how many women wearing shoes there are in the living room?
I must go now.
You asked me to tell you how many women wearing shoes there are in the living room. Is that correct?
Robot yes.
Okay, lets see what I can do!

Figure 1.1: Interaction between two referees and a robot in the RoboCup@home league. The referee in blue abandons the robot mid interaction, which does not acknowledge this at all.

the problem to combine them. A number of factors have to be considered when combining these information, e.g. information may arrive asynchronous. For example: while just a single, long speech utterance may be received, a number of voices can be detected, or several results of the same voice. Results in that sense are singular perceptions of a component, e.g. a voice recognizer or a Sound Source Localization (SSL) detector. Thus results have to be combined in a manner that takes this asynchrony into account. Additionally, results will have different calculation times, based on what component created them and thus may be temporally unaligned. This must be considered as well, creating the need to synchronize results based on their occurrence in time.

The problems just presented are clearly out of scope for robot behaviors an thus should be handled separately, due to their complexity. I thus declare the **main goal of this thesis** to create a **framework for the automatic generation of these synchronized audio analysis results**.

Furthermore, a number of **secondary goals** can be declared. Naturally any approach to synchronize audio results shall not have a negative impact on these results. This can be specified in two more concrete terms. First and foremost, the **accuracy of the results shall not decrease** by being incorporated in the proposed solution. Second, synchronizing the **results shall not take considerably longer to compute** than not synchronizing them. To put this into context: waiting for a result more than twice as long can not be deemed acceptable.

Robotics is a field of intensive current research, and in the last years a number of leaps have been made. These include, but are no limited to OpenPose [Cao+18], YOLO [RF18] and DeepSpeech [Han+14]. Modularity and the ability to include such advances without the need to completely overhaul an existing system greatly decreases the time needed to incorporate such new technologies. In turn, research speed can benefit. I can thus declare an additional **secondary goal** in the form of **modularity**, to increase the usability of the proposed framework.

The rest of this thesis is structured as follows: I will first introduce concepts and frameworks in chapter 2. Here, first latency is introduced. Latency is of special importance since it plays a role in the computation times and is thus of particular interest with regards to the similar secondary goal. After this I will discuss and briefly explain the concepts of Automatic Speech Recognition (ASR) and SSL as these can yield results of interest and are incorporated in the later evaluation. Additionally, beamforming will be touched upon, as it is used to enhance audio signals and will later present a number of challenges for the design of the proposed framework. Finally I will explain the Robot Operating System (ROS), a middleware used in the later proposed framework.

Thereafter I will present related work in chapter 3.1, which will be divided in three parts. First, a brief overview of a framework which focuses on SSL and beamforming will be given. Then I will discuss several fusion and synchronization approaches in preparation for my approach. Lastly, an overview of current and former teams of the RoboCup@Home league will be given, with special focus on their speech understanding and data fusion approaches.

After this I will present my solution in chapter 4. Based on the goals formulated above and fusion approaches discussed in chapter 3.2, and in consideration of the concepts of latency and beamforming discussed in chapter 2.1, I will then present my approach for synchronization and fusion of speech analysis data. This solution is a framework comprised of a library and a master component, the Orchestrator. This chapter will close with an overview of the components developed over the course of this thesis.

I will then proceed to evaluate my proposed solution in chapter 5 based on two experiments. The first experiment is conducted with the help of a data set, and is intended to evaluate the performance of the developed framework with regards to computation time. The second experiment is heavily inspired by the ”Speech and Person Recognition” Task of RoboCup@Home [Mat+18]. It is intended to evaluate the accuracy of the proposed framework in comparison to a pre-existing solution. Both experiments will be able to show the main goal of this thesis to be achieved. The data set experiment in particular will show the requested modularity.

Lastly I will summarize my findings and point out possible future work in chapter 6. The central point of this chapter is that due to the findings of chapter 5, more and better components can be developed and the next step can be taken, i.e. behaviors equipped to handle situations such as those seen in figure 1.1 by utilizing synchronized data can be created.

2 CONCEPTS AND TECHNOLOGIES

In this chapter, I will provide an overview of and describe concepts and technologies used within this work. First, latency will be discussed, as it is important when designing the resulting framework. Then the concepts of ASR, SSL and beamforming are introduced. They are later incorporated into the resulting framework, or are at least considered during its design. Finally, ROS will be discussed, as it was chosen as middleware. The reasons for this will be discussed in chapter 4.

2.1 LATENCY

Regardless of the method used, when transmitting any kind of information, this transmission will not be instantaneous, but instead happen with a delay. This delay is a universal effect and can be found in macro levels, light from distant galaxies reaching us after thousands of years, over a more intermediate level, sending a letter, to a micro level, the sound of a spoken phrase traveling to the ear of the recipient. This delay is also known as *latency*.

However, while the laws of physics cannot be altered, one might be able to reduce latency when in control of the environment. To follow the previous example of sending a letter, one may opt for sending an intercontinental letter via plane instead of ship.

In this work, latency will be discussed in the context of transmitting audio signals. As audio signals are not singular events, akin to e.g. a photon, but rather a continuous stream and most of the computations presented in this work are done by computers which can work faster then audio signals typically occur, their latency behaves differently in a number of ways. One such special behavior for example can be observed when recording audio from a microphone. This is typically done in an iterative manner: the software will require a predetermined number of audio frames, i.e. singular data points of an audio stream. After these frames are acquired, they can be processed. Then, it will typically wait for the next batch of available data. Through this procedure, the amount of frames requested can have a impact on the overall latency measured. Consider requesting audio with a length equivalent to one second. Even when the software introduces virtually no latency, the audio frames at the beginning of the acquired audio chunk had to be recorded a second prior.

In chapter 5.1 I will show a particular speech recognizer to just spend 0.063 seconds processing audio, with an average signal length of about one and a half seconds. This particular speech recognizer needs audio to be recorded with a sample frequency of 16000Hz, which would enable it to record audio to the equivalent of around 1000 frames. This shows that finding an effective audio chunk size is important and not necessarily trivial.

2.2 AUTOMATIC SPEECH RECOGNITION

ASR is the general term for speech recognition typically performed by software. In general terms, an ASR program turns an audio signal into a written representation of the utterance encoded within this audio signal. A great variety of approaches are represented by ASR. Typical approaches calculate signal frequencies of the audio sample and then map them to the closest match of frequencies known to be produced by speech patterns via hidden Markov models [KKS89; Gal98]. Other approaches make use of neural nets and deep learning [DHK13].

2.3 SOUND SOURCE LOCALIZATION

SSL [FP70] is a technique used to determine the origin of a sound relative to an array of microphones. A great variety of algorithms has been developed in this field such as MUSIC [Ots+11] or FRIDA [Pan+17]; however they build upon a general principle, which I will outline here.

Consider a setup of a number of microphones and a sound source, seen for example in figure 2.1. The positions of the microphones must be known at all times and ideally be static. Due to the sound waves traveling with a constant speed, a single sound event can reach microphones at different points in time. By matching the events captured in the recording of each microphone with their respective selfs in the recording of the other microphones, one can compute the time-of-arrival differences of the event. In the given example, this time-of-arrival difference would be the smaller, blue arrow. Due to the speed of sound being constant and known, it is then possible to triangulate the general direction or even position of the source. Thus, by observing similar excitations detected at different points in time by different microphones a sound source can be localized.

SSL requires an array of at least two microphones. This will however yield two possible direction of arrival (DoA), which may be enough, depending on the application. To calculate an exact location in a two dimensional space, at least three microphones are needed, while a three dimensional space requires at least four. An excess of microphones is often used in practice to increase accuracy of the DoA.

Due to practical limitations caused by slight inaccuracies in positioning of the microphones, an exact sound source can often only be determined in the immediate vicinity or even just inside of the microphone array. Otherwise, only a general DoA can be calculated with respect to the microphone array. Most approaches do not handle the special case of finding the exact position of a source inside the microphone array and thus only provide a DoA.

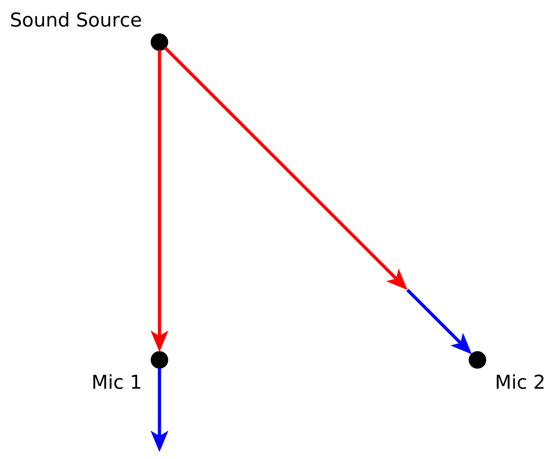

Figure 2.1: Runtime variances for SSL. A microphone array comprised of two microphones, Mic 1 and Mic 2, receive an audio signal from a sound source. When the signal of a particular event reaches Mic 1 (pictured in red), it is still on its way to Mic 2. When it reaches Mic 2 (pictured in blue), it has already passed Mic 1.

2.4 BEAMFORMING

Beamforming [BCC17] could colloquially be described as a sort of inverse of SSL. Its goal is to, in a way, "listen" to a particular direction and provide an audio signal given the recording of a number of microphones and a direction. This new, artificial audio signal will greatly enhance audio events occurring in this particular direction. Similarly to SSL, a number of algorithms have been developed around a single principle, which will now be outlined. [Ger03] provides an in-depth overview with regards to established beamforming techniques.

While in SSL the source of the sound is not known and the time-of-arrival difference must be calculated from the signal, in beamforming the position of interest, or its angle relative to the microphone array is predetermined as seen in figure 2.2. Due to this, one can compute the time-of-arrival difference expected by this position or angle, as indicated by the blue arrow. By then combining the audio recorded from the microphones with respect to their positioning and expected time-of-arrival difference, one can create a new, virtual signal, which greatly improves the audio quality of events originating from this position or angle, while simultaneously greatly diminishing the events from distant positions or angles. Beamforming can be used either together with SSL, to improve sound quality from known sound sources, or independently from it, e.g. to focus on a person which had its position determined with the help of a camera.

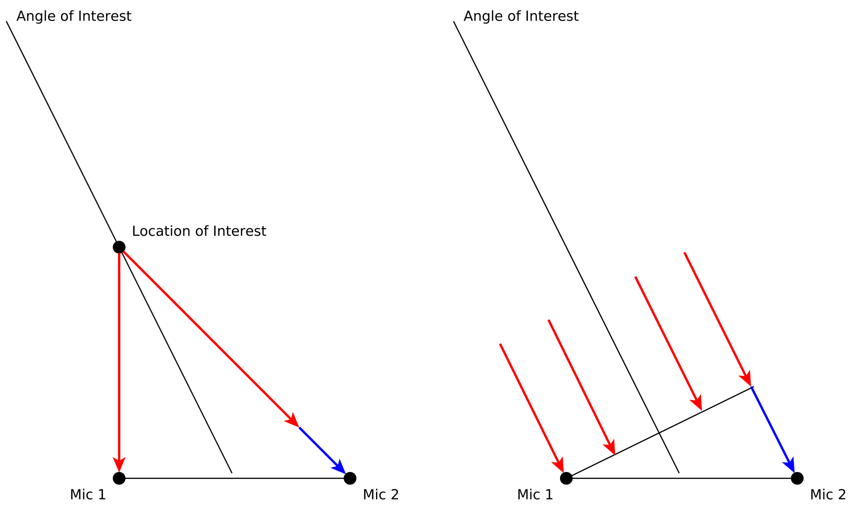

Figure 2.2: Beamforming with two microphones Mic 1 and Mic 2. On the left: By taking into account the additional time an audio event takes to reach Mic 2 (in blue), a new signal can be created, which greatly increases sound events originating from the location of interest. On the right: By the same principle, a location of interest infinitely far away can be targeted, which is then equivalent to improving sound events coming from a specific angle of interest.

2.5 ROBOT OPERATING SYSTEM

ROS [Qui+09] was developed explicitly for communication of programs in the context of robotics and offers a wide variety of standardized data types for transmission. These include equivalents of widely used data types such as integer, float, char and string. However, one is not limited to these data types, but can construct new types by combining already established ones.

Programs which make use of ROS, also called ROS-nodes, typically start by initializing ROS and thus announcing themselves to a dedicated ROS-master node, which orchestrates all communication within the ROS ecosystem and is always required. They will then announce all their chosen ways of communication with other ROS-nodes and either start their work or await data or requests from other ROS-nodes.

ROS offers three distinct methods of transmitting these data types with increasing complexity: the first is a simple one-way transmission of information[1], from a so called *Publisher*-node to a *Subscriber*-node. With it, data can be transmitted, also called published, to any number of *Subscriber*-nodes. Recipients are identified by the master-node by being subscribed to a specific communication channel, a so called topic, which must be the same as the one the *Publisher*-node publishes to.

The second method is based on providing data on an on-demand basis and is know as a ROS-*service*[2]. It features a *service*-server, which

1 http://archive.is/Tvsg7
2 http://archive.is/ab2sd

advertises a specific *service*. Any ROS-node may then send this *service*-server a request, which is, in essence, a ROS message. The *service*-server will then somehow react to this message with a single answer, another ROS message. Similar to the simpler publish-subscribe method, *services* are identified via a so called *service*-topic.

The third method will be only discussed briefly, as it is not used in this work. It is known as a ROS action[3] and identifiable, similar to the already discussed methods, by an action-topic. It is similar in nature to the ROS service, but features additional update messages send from the server to their respective client.

3 http://archive.is/ujkJK

3 RELATED WORK

In this chapter, a number of related projects and research findings will be presented and discussed. Several directions will be explored, i.e. an existing audio framework to transport audio & create audio processing pipelines and different approaches to fusion of information. Lastly RoboCup@Home, a competition between domestic robots with a strong research focus will be discussed, and several RoboCup@Home teams will be looked into with respect to their synchronization and fusion methods.

3.1 HARK

In this part Honda Research Institute Japan Audition for Robots with Kyoto University (HARK) [NOM17] will be discussed, a modular framework with explicit focus on integrating beamforming and speech recognition solutions, developed by a team of researchers centered around Kyoto University. Its goal is to provide an all in one solution for audition in robots specifically. HARK is modular as it provides several algorithms (so called nodes) for each processing step. These are centered around SSL, beamforming, filtering, and feature extraction nodes. HARK does not directly incorporate ASR solutions, but instead opts to extract audio features and send them via network to dedicated programs. These are thusly not under its direct control. It does not directly support other kinds of audio analysis, such as emotion-, voice-, or gender-recognition. Nevertheless, one could develop these kinds of nodes to work with audio received via network and thus "trick" HARK to incorporate these nodes. Naturally, it thus does not support the synchronization of results of any such components. All the more, it does not support the synchronization of SSL and ASR results, as results of ASR nodes lie outside its control.

HARK will take care of transporting audio data in between processing steps, but mostly transfers them in a frequency based representation, not as raw audio data. This results in quite stripped down nodes, as it reduces the number of fast Fourier transforms needed to flat one instead of one per node relying on them. Nodes which rely on raw audio however, must either manually restore the signal or use a specially developed node for this task. Both variants results in a degrading signal however.

3.2 FUSION

Moderns robots generally have a plethora of different sensors to perceive their environment, such as cameras, laser sensors and microphones. Oftentimes different sensors will provide information about identical characteristics, a picture provided by a camera may for example include a person, which could also be detected by laser detection of the legs of that person. To create an optimal model of the robots world, these information must be fused. In this section I will present a number of approaches for this fusion. Most of these approaches describe multisensor systems. Though the interest of this thesis lies in the field of speech understanding, which typically makes use of just one sensor -the microphone-, these can easily be applied by broadening the definition of sensor from physical ones to also include components as a sort of software based sensor.

[Tur14] and [Por+16] divide between feature and decision level fusion within multimodal systems. As such they only cover classification of a single characteristic with data of several sensors. Feature and decision level fusion differ in the level on which the fusion happens. Decision level fusion encompasses several unique classifiers, which are typically independent from each other. These classifiers can take the raw data of one or several sensors and produce distinct classifications of a shared characteristic. Afterwards, a special decision level fusion algorithm will take all these -potentially different- classifications and produce a definitive classification. In contrast to that, feature level fusion makes use of only a single classifier, which is fed with raw data of several sensors and produces a classification directly from that. Feature level fusion is thus only achievable with several sensors and specifically trained or prepared classifiers.

[AR94] provide an algorithm to fuse the asynchronous results of multiple sensors which focus on the same characteristic. They thus try to extract the most likely states of the observed system. [Bla+92] solves the issue of synchronizing results by batching all data within a specific time interval and then compressing them. In contrast to that, [YLZ07] uses a recursive algorithm in conjunction with a Kalman filter to fuse asynchronous multisensor data and thus better estimate the state of a given system. In [AGM05] asynchronous sensor data are fused without the need of synchronization. This however necessitates all sensors to perceive the same characteristic.

3.3 ROBOCUP@HOME

The RoboCup was founded in 1996 as an annual soccer competition for robots with the goal to beat the human world champions by 2050[1]. Over the years, RoboCup split into several leagues and expanded into other disciplines as well, such as the Rescue Robot league or the RoboCup@Work league.

RoboCup@Home[2] is a league of the RoboCup dedicated to service robots in a domestic environment. Therefore, human robot interaction is one of, if not the primary focus of this league. Naturally, speech recognition systems are used as the default way to interact and com-

1 http://archive.is/CGYqO
2 http://archive.is/TGhGg

municate with the robots. This results in the demand for robust speech recognition software.

One of the tasks tested in RoboCup@Home in 2017 and 2018, the ”Speech and Person Recognition” task [Mat+18], is of particular interest for this work, as it mainly tests the basic speech recognition ability of the robots. As it was used in several RoboCup events over two years, and on dozens of robots, it is an convenient test to evaluate any speech recognition system on a robot. Consequently, it will be used to evaluate the performance of the proposed pipeline later in chapter 5.2, where the test will also be illustrated in greater detail.

To gather information on the state of the art with regards to synchronization of sensor results and speech results in particular, I will provide an overview of established RoboCup@Home teams, which published information on that matter. However, this information is relatively sparse in team description papers and on the teams websites, so some additional information was gathered by contacting the teams directly.

3.3.1 *SPQReL*

The SPQReL team from Sapienza University of Rome and the University of Lincoln use several fusion approaches [Láz+18]. Visual person perceptions were matched with SSL results, based on their respective angular difference, and could thus be tagged as speaking. Spoken sentences could then be mapped to speaking persons (according to email). They employ several speech recognizers, i.e. Google Speech API and a Nuance speech recognizer, and fuse their results with a custom speech understanding system called *LU4R* [Bas+16], which is able to prioritize results based on expected domain specific terms. As such, this approach fits into the category of decision level techniques, as discussed in section 3.2.

3.3.2 *RoboFEI*

The RoboFEI team from Centro Universitário FEI in Sao Paolo [Per+19] uses a specific microphone configuration to handle problems caused by moving sound source localization microphones. Their robot’s head is attached to a rotating pipe, which enables it to spin around its axis, while the robot’s microphone array is attached to a static shaft running through that pipe which keeps it still. As such, the position and movement of the robot’s head are not relevant for sound source localization results and beamforming, and can as such be ignored. Nevertheless, the robots own position and movement needs to be taken into account when performing SSL, as is the case for all mobile robots.

3.3.3 *Tech United*

The Tech United team from the Eindhoven University of Technology provides some information about their speech recognition system. In 2018, they generated SSL information independently from their speech recognition [Bur+18]. They did not fuse these information however, instead opting to separately process them.

3.3.4 *Walking Machine*

The Walking Machine team from École de Technologie Supérieure in Montreal [CH19] uses an solution, ODAS [GM19], which would enable them to perform SSL and beamforming in one component, thus providing an enhanced audio signal. However, they appear to only use its SSL capabilities at this time.

3.3.5 *Kamerider*

The Kamerider team from Nankai University, China provides some information on the components they use for speech recognition and SSL [Tan19]. They use Gstreamer [Wal+01] to segment audio, which is then being processed by PocketSphinx (PS). Additionally, they use HARK for SSL. This way they appear to have modularized their components to a greater degree than other teams. However, they gave no information about the interfacing of thusly acquired ASR and SSL results.

3.3.6 *Hibikino Musashi*

The Hibikino-Musashi team of the Kyushu Institute of Technology also uses HARK, specifically its *MUSIC* algorithm for SSL [Tan+19]. Additionally, they employ an ASR solution in the form of *Google SpeechRec* via its web API on the Chrome web browser. However, I could find no information about how the results of those components were interfaced.

3.3.7 *ToBi*

The team of Bielefeld (ToBi) of Bielefeld University [Wac+19] can be discussed in greater detail, as I am a former member of this team. For SSL they used a proprietary solution which came with their Pepper robot[3] while they used a PS based component for ASR, the PocketSphinxAdapter (PSA).

ASR and SSL results are merged on the behavior layer, but only when SSL information is explicitly needed. The method with which

3 http://archive.is/2VKYS

they are merged is rather simplistic. SSL results are temporarily stored along a timestamp, which corresponds to the time these results were gathered. Whenever an ASR result needs to be enhanced with SSL results, an average is calculated over the SSL results which were recorded between two and half a second before said ASR result. This offset is employed to mitigate the time the PSA needs to recognize an utterance and the time needed by the behavior engine.

The PSA is an ASR component which is based on PS. It captures sound directly from a microphone via Advanced Linux Sound Architecture (ALSA), which is then segmented, to ensure only audio containing actual speech is evaluated. It segments audio based on a loudness threshold, which, if crossed long enough, will start a segmentation. This segmentation is ended either if a certain time limit is exceeded or if the loudness of received audio chunks drops under a second, typically slightly lower threshold. Audio which is therefore decided to contain speech will then be fed into PS. After a segmentation is ended, PS will produce a result, which is then published via a ROS message. This component will be used in later experiments (see chapters 5.1 and 5.2).

One of these experiments will show the PSA to require considerably less time for recognitions than these 500ms. However, a number of factors make this a good approximation still. First, during the RoboCup@Home event, all computations are done on the robot itself, which sports considerably weaker hardware than the machine used for the experiments later conducted. Additionally, the grammars typically used in RoboCup@Home Tasks are considerably bigger than the grammar used in the later experiments, which makes matching a given utterance to a phrase in the grammar computationally more expensive.

4 CONFIGURABLE SPEECH PIPELINE

In this chapter I will describe my solution to the problems presented in chapter 1. I will first give a broad overview how and where certain tasks are handled, before describing the two major components of the pipeline, the Orchestrator and library, in more detail. Lastly I will present a number of components developed for my solution as a proof of concept and to introduce them, as most of them are later used in the evaluation of this thesis (see chapter 5).

The main goal of this thesis is to provide synchronized results of audio analysis components, such as ASR and SSL results. To effectively provide fused results, these results are first needed in a standardized separated form. Additionally, to be able to synchronize separated results based on time, they need to be annotated with a timestamp. However, this timestamp needs to fulfill a number of requirements: as the first and most central requirement, the timestamp must not correspond to the time a given result was made available, but rather to the time when the corresponding audio signal was recorded. This is fundamental, as each result providing component can not be expected to take the exact same time as any other. Additionally, as the audio this timestamp corresponds to is not a singular event, but a continuous stream, it must contain a start- & an end-time to fully and accurately describe it. This is especially true since results may be generated on audio of vastly different length. Consider for example a simple "yes" or "no" in contrast to a longer question, such as "Hey robot, where can I find the orange juice?". This directly leads to another problem: as each results now requires an annotation in form of a timestamp, each component subsequently requires audio data annotated with timestamps.

This, in turn, leads to a number of cases with a high complexity in dependencies. Consider a setup of a SSL component, a beamformer and an ASR component. In this setup, the SSL component would have to acquire audio data and annotate it with timestamps as discussed above and then provide its results. An independent beamformer would in turn also have to acquire audio and annotate it, and then match the SSL results provided by the respective component to its audio data. Then it could calculate the beamformed audio signal, which would also required to be annotated with timestamps. Lastly, the ASR component would have to collect these timestamped audio signals, produce ASR results and then annotate them with timestamps. An additional dependency implicitly declared in this example is that the ASR component must be able to process the audio the beamformer produces.

Most of these requirements could, in theory, be handled by the components themselves. However, I decided to outsource most of the work done by the components into a library, to allow them to focus on their respective goals. This provides a number of advantages:

First, by outsourcing this work, standards must inherently be defined. As they can easily be heeded by the components, they provide benefits for the individual components and for the final synchronization of the results. More so, by defining these standards and providing a library to heed them, the fragmentation of speech recognition components is encouraged. Take for example the PSA described in chapter 3.3.7. Not only does it contain an ASR component, but also a Voice Activity Detection (VAD) component. This makes sense from a developmental perspective, but impedes further development and replacing of its parts. Even when done correctly and iteratively, after a few exchanges new technologies may need different interfaces. By encouraging this melding of components to not occur, the resulting framework becomes highly modular and enables fast prototyping. It also makes benchmarking individual components of such a pipeline easier, as I will later show with a number of experiments for evaluation (see chapter 5.1).

Furthermore, by providing a way to easily send properly timestamped audio between components, a massive amount of work is lifted off the individual components, which in turn ensures the correct transmission of audio timestamps. Additionally, more technical benefits include the prevention of code duplication and the reduction of bugs, which provide more time for actual research.

By embedding each component into the framework provided by the library, the proposed solution for finally synchronizing the results is also provided with standardized interfaces. I named this solution the Orchestrator, as it not only synchronizes the results, but also keeps track of all components within the proposed pipeline.

Both library and Orchestrator work in tandem and need to communicate with each other on several occasions. ROS was chosen as an underlying middleware, see chapter 2.5. There are several reasons for this. First, due to ROS relying on TCP, all communication can be expected to arrive, none will be lost. Additionally, pre-existing infrastructure, such as the behavior controller used in one of the experiments, already depends on ROS. As such choosing ROS will prove beneficial for future work.

I will now continue to discuss the Orchestrator and the proposed pipelines library in more detail in designated sections.

4.1 LIBRARY

In this part of this thesis the library of the proposed framework will be described, along with its tasks. As discussed earlier, the library of the proposed framework solves a number of problems, which are either directly needed for synchronization of results or make it considerably easier. First the reasoning for some major design decision will be laid down. Afterwards more technical details and explanations will follow.

Audio Formats

One of the central tasks of the library involves transmitting audio from component to component. This includes some aspects that are not immediately obvious. One of those is the audio format each component requires. An audio format determines the structure of the audio data on a lower level. Typically, the structure of any audio data can be described by four factors:

1. **Samplerate**: determines the resolution of an audio signal in time domain.

2. **Bitrate**: determines the resolution of the singular data points within an audio signal.

3. **Endianness**: determines the order of bytes of the singular data points within a audio signal. Litte-endian encoding puts the least significant bytes first, while big-endian encoding puts the most significant bytes first.

4. **Channel Count**: determines the number of distinct audio signals within the signal. When recording audio, the *Channel Count* is generally equivalent to the number of microphones used to record the audio signal.

Working on audio data which is present in a different format then expected will in most cases lead to unusable results, maybe even to critical segmentation faults.

Under normal circumstances, a component will request audio in the format it requires directly, e.g. via ALSA. However, as the library now handles all audio transmission, this is no longer feasible. With respect to one of the secondary goals of this thesis, to provide a modular framework, every components algorithm can also not be assumed to work with a specific audio format. Thus each component is required to either use a single, pre-determined audio format or the library must handle resampling and converting audio signals for each component. By handling resampling and converting of audio signals within the library, components of the resulting framework can theoretically employ any kind of algorithm they desire, without having to handle resampling and converting audio themselves. Furthermore, by abstracting resampling and converting from the components themselves, the library can be equipped to change audio formats adaptively when new components are introduced or old components vanish from a specific configuration, in a standardized way.

Consequently, the library needs to be able to resample and convert audio between arbitrary audio formats. The Sound eXchange (SoX) Resampler library (SOXR) [rob07; Sor04] was chosen to resample the audio and a custom implementation was used to change its *Bitrate*. The library does not handle different *Channel Counts*. The reason for this is the absence of a sufficient default behavior. While creating a two channel audio signal from a single channel is easily done by doubling it, how to convert a two channel signal with non-identical tracks into a three channel signal? What about the other way round? Should a two channel signal be interpolated from a three channel signal, or should one of the tracks be discarded? The answers to these questions differ from case to case. As such, it seems more wise to separate the channel mechanics into dedicated components and entrust the user with finding the best choice. The library was prepared to handle different *Endianness*, but ultimately such a feature was not implemented, as different byte order is mostly a historic phenomenon and most modern operating systems, such as Windows, MacOs and most Unix systems, use little-endian encoding.

From the perspective of the library, it can then be divided between an externally used format, which is the format in which the component provides or requests audio data to the library; and an internally used format, which is the format in which the audio is transmitted. Both of these can be identical, but must not be. Resampling and converting of audio signals should be minimized, so careful consideration is required when choosing the format with which to transmit audio between components.

Audio Format choosing

It is however equally important where these formats are chosen, as it could be done in two ways: either in a distributed manner, where all existing components must communicate with each other and come to a single conclusion. Alternatively they could be determined in a centralized manner, where a single master program will collect information about each component and then based on this information will make a decision. The later was the chosen course of action, which was motivated as follows: the main advantage of a distributed approach is that it does not rely on an additional program, as does the centralized approach. It does so however at a cost. Due to the nature of the distributed approach each component taking part must have the same information as every other. See for example figure 4.1. In this scenario of average complexity one can clearly see the centralized approach to introduce less overhead. While the communication between `c_1` and `c_2` is straight forward within the distributed approach, the communication between `c_2`, `c_3`, `c_4` and `c_5` becomes quite convoluted. Additionally, the way in which each format is determined must be deterministic as each component must come to the same conclusion. All this results in a overhead in both computational load and inter-component communication.

General Requirements

However, as there is already always an additional program employed to synchronize the results, namely the Orchestrator, the benefit of the distributed approach becomes negligible. I thus chose to implement the centralized approach with the Orchestrator as the master component, which determines each components internally used audio format. I

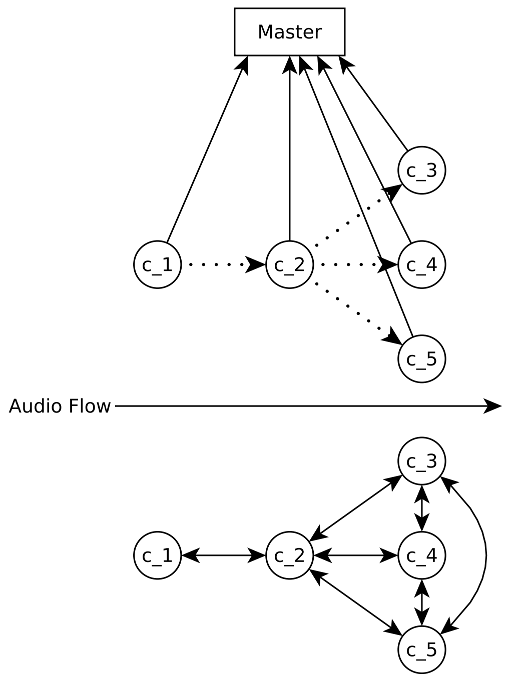

Figure 4.1: Centralized (above) versus distributed (below) approach of determining audio formats between the components `c_1` to `c_5`. Information sending about each nodes preferred audio topic format are indicated by arrows. The direction of audio transmission is indicated by pointed lines above, and from left to right generally speaking.

will go into detail how the Orchestrator chooses these formats in the following section 4.2.

The library was developed in C++, due to it having access to a number of very widely used and tested libraries for audio processing, such as SoX as well as its speed. However, a number of interesting software was developed for Python, see for example libraries on which I based most of the developed components, discussed below in chapter 4.3. This was the motivation to develop Python bindings for the library, using Boost Python [AG03]. It proved fruitful, as most the then developed components actually make use of these bindings rather then the original C++ library. Developing the library in Python in the first place was considered, but ultimately not chosen because of C++'s superior support with regards to sound libraries, such as SoX and SOXR.

Result Types

Another important task of the library is to provide standardized ROS messages of common results generated from raw audio. This is realized by a number of ROS messages, dedicated to speech-, emotion-, gender-, voice-, SSL-, and VAD- information. These messages can be inspected in figure 4.2. They generally consist of a start & end timestamp, a probability and a string containing their actual result. Special cases are: the *VADInfo* message, which lacks such a string, as it only needs

Duration	
start	finish

GenderInfo		
gender*	probability	Duration

EmotionInfo		
emotion*	probability	Duration

VoiceIdInfo		
voiceId*	probability	Duration

VADInfo	
probability	Duration

SpeechInfo		
hypotheses[]		Duration
recognizedSpeech*	probability	

SSLInfo			
directions[]			Duration
sourceId*	angleVertical	angleHorizontal	

EsiafRosMsg		
GenderInfo[]	EmotionInfo[]	VoiceIdInfo[]
VADInfo[]	SpeechInfo[]	SSLInfo[]

Figure 4.2: A list of all available result messages and their composition. All entries indicated with a star are of type string and generally used for result transmission. All other entries are either of a type shown here (e.g. Duration), or are floats. Square brackets indicate this entry to be a list of this type with unspecified length. The EsiafRosMsg is a special type reserved for usage by the Orchestrator, as it represents the message of synchronized results.

to capture a timeframe; the *SpeechInfo* message, which consists of a list of result strings and their respective probabilities, along with an duration; and the *SSLInfo* message, which in addition to a duration encompasses a list of SSL outcomes, determined by an vertical as well as horizontal angle and a distinct source ID. In any case, these must be published by the components to similarly standardized ROS-topics, consisting of the name of the component and the type of the message.

I will now illustrate the tasks handled by the library by means of typical actions taken by an example component in its life-cycle. My example will be that of a typical VAD component. As ROS is used as an underlying middleware, components within the proposed framework must also use ROS for a number of tasks. Thus, it is necessary for the component to first initialize ROS. Following this, the proposed library will need to be initialized, which is done by first creating a handle. This handle keeps track of all relevant information the proposed library needs.

The example component may then declare its intention to output or request audio. In the example, the VAD component will first declare its intent to output audio and then to require audio. However, the library does not put a limit as to how many in- & outputs a component can request, so an identifier is needed. The same identifier will later be internally used to generate the ROS topics with which the actual audio will be transmitted. Thus, it also serves to map the audio between components. If the VAD component would use the input topic ''`vad_input`'', then a microphone component would be needed to output to the same topic, so that the audio could be correctly transmitted. Furthermore, to request an audio in- or output it is needed to specify the format in which the audio should arrive or will be given to the library respectively.

When declaring the output of audio, there are no further requirements. When requesting audio however, the component must present the library with a callback function. This function will later be invoked when audio was send to the component.

Raw Audio Transmission

Once the component is initialized and is processing data, the component will require to actually receive and output audio. To output audio, it will need to provide the raw audio signal in the format that it chose previously and its timestamps along with the topic on which this audio shall be transmitted. The library will then convert this raw audio signal and its corresponding timestamps to an *Augmented Audio Message*, as seen in figure 4.3. The information contained in these messages can be divided into three categories:

1. Transmission information
2. Synchronization information
3. Meta information

The transmission information consists of the audio signal that is to be transmitted as well as a message ID. This ID is generated by the library on the transmitting side and is consecutive. ROS as a middleware does not ensure messages to arrive at a node in the same order they were sent. Thus, by adding an ID to each message, the receiving side of the library can later rearrange the order of arrival, should messages arrive out-of-order. The library will also resample and convert the raw audio signal it received in the externally used format into the internally used format, if these are not identical. Depending on the size of audio chunk this part of the message should normally be the largest. It must be noted however, that because picking the right size of audio chunking is a non-trivial exercise (see chapter 2.1), the library does not give any restrictions on the size of the audio signal to be transmitted. Even varying the size of the audio chunks between messages is allowed.

The synchronization information is the part of the message which carries the timestamps corresponding to the audio signal. It therefore enables components to enhance their results with accurate timestamps which in turn are later synchronized by the Orchestrator.

The last part of the message consists of optional meta information. These meta information can be SSL or VAD information. As previously discussed, this information must be published by their respective nodes to the Orchestrator for synchronization purposes. However, as a special case for SSL and VAD components, they should also enhance the audio information with them. The reasoning behind this is rather straight forward: most beamforming components will need SSL information to work correctly. If these information is not transmitted with the audio signal, the beamformer would need to acquire these information by itself. After this it would then need to synchronize the SSL information with the audio signal. Thus a component of the proposed framework would be tasked with the very goal of this thesis. How absurd this would be. If these information are however included in the audio signal themselves, they are automatically fused and the beamformer can begin its work outright. This argument works analogous with ASR components and the VAD information.

To enhance an *Augmented Audio Message* with these meta information, a given component must explicitly send them to the library before outputting said *Augmented Audio Message*. To receive these information, the component must provide the library with a callback function for each. The receiving component will have all of its callback functions be called in the opposite order. This process can also be seen in figure 4.4. To come back to the example component: the VAD component, upon receiving audio, first generates its VAD information, which it then publishes to the Orchestrator and simultaneously gives this information to the library as meta information. Afterwards it sends the audio signal it received via the library, which in turn enhances this audio with said VAD information. The *Augmented Audio Message* then arrives in the library of e.g. an ASR component. Before anything else, the library first checks the ID of the message. If it was higher then expected, it stores this message and waits for more messages. When the expected message arrives, the audio signal is resampled and converted from the internally used format to the externally used format. Then the input callback function of the component is called with the timestamps and audio signal of the message. This way the component can use the timestamps directly when publishing results, or just propagate them while outputting audio. Subsequently the library calls possible SSL meta information callback functions first and then their VAD equivalents. Thus it is guaranteed that all audio was processed before information about an ending utterance is given to the component. After all callback functions for a given message are called, the library will check for previously but out-of-order received messages and processes them in the same way.

Coordination between Library and Orchestrator

After the declarations of audio in- & and outputs are done, the component will signal the library to start. Up until this point, no actual

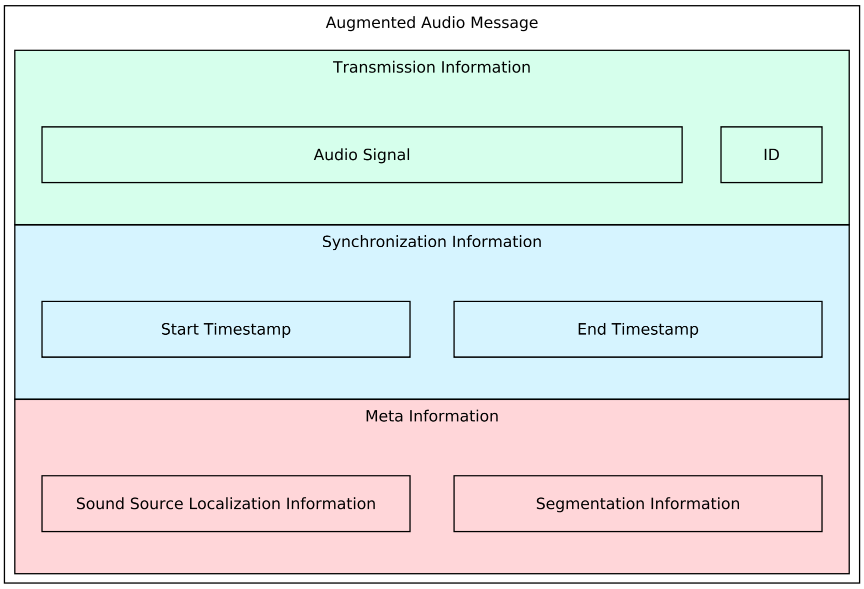

Figure 4.3: Composition of Augmented Audio Messages. The message can be divided into three parts: transmission information in green, synchronization information in blue, and Meta information in red.

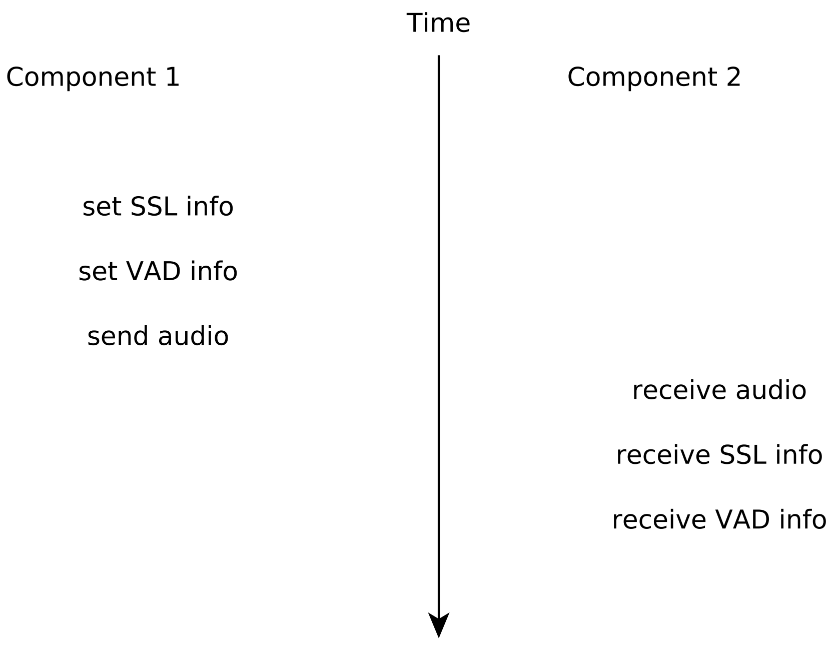

Figure 4.4: Timeline of two components sending and receiving *Augmented Audio Messages*. One can see the meta information of SSL and VAD to be set before sending the corresponding audio, but being received after it.

communication between the component and the Orchestrator has happened. Now however, the library will prepare all information necessary for the Orchestrator to register the component. The name of the component is needed, so the Orchestrator can differentiate components from one another, as well the designation of the node, which corresponds to the task the component will perform. I will go into more detail regarding this designation in the Orchestrator's chapter 4.2, as it serves to ease synchronization of results.

Further information is needed regarding in- & output topics of audio that were previously declared. The information about meta-information is not needed for the registration process, as they are handled internally by the library itself. Registering the component is then done via a dedicated ROS-service, which has the library send these information. Using a ROS-service in this instance instead of a simple ROS-message has the advantage, that it can be ensured that the Orchestrator is started and ready, which may not necessarily be the case when starting the Orchestrator and components of the framework in rapid succession.

After this registration has finished, the components work may begin. When the component has finished its work it may signal this to the library, or simply exit. If the library was informed in this way, it will shut down all audio transmission. As components cannot be expected to exit in a regulated manner in all cases, the Orchestrator must be able to handle a sudden disappearance of components. Thus there is no dedicated way to de-register a component within the library, and instead the Orchestrator handles all component shut downs. I will focus on the Orchestrator's approach for this in the following chapter 4.2.

4.2 ORCHESTRATOR

In this section the Orchestrator as part of the proposed framework will be discussed. The Orchestrator has to fulfill two major tasks, synchronizing results and determining the formats used to transmit audio, as discussed in the previous chapter 4.1.

Determining Audio Formats

First the determination of audio in- & output formats used internally by the library will be discussed. To achieve this, a number of factors have to be considered. To reduce network traffic and improve transmission speed it is advantageous to transmit audio with the lowest possible bandwidth. Consider two nodes, where one node transmits and one node receives audio which differs only in that one node internally works with 16kHz and the other with 48kHz. In this scenario, resampling should happen at the node which works with 48kHz, in order to transmit only 16kHz audio.

However, it may be the case that network capacity is abundant while CPU capacity is not, e.g. in embedded systems. Consider a node producing an audio signal at 16kHz, and four nodes requiring this audio signal at 48kHz, e.g. a VAD, gender-, emotion-, and voice recognition. In this scenario, one may opt to not resample four times, but rather once and transmit the higher bandwidth audio, or, more generally speaking, minimize the number of resamplings happening in the pipeline.

To tackle this issue, I developed two algorithms which could each handle one of these cases and added a simple way to add new algorithms, should the need arise. The first algorithm introduces a cost function which calculates the bandwidth of each topics formats. This algorithm is formalized in figure 4.5. It applies the aforementioned cost function $cost_{format}$ on each in- & output format for a given topic and then chooses the minimum as the optimal.

The second algorithm is a bit more complex, as it prioritizes the amount of resamplings over the used bandwidth when choosing the right format. It is formalized in figure 4.6. As such, it calculates for each format on a given topic the number of resamplings that would occur, if that format was chosen for this topic. Then the format which has the fewest resamplings is chosen as the optimal format. However,

$$\begin{aligned} cost_{format} &= channel\_count \cdot bitrate_{signal} \cdot frequency \\ format_{optimal} &= min(bitrate_i \mid i \in formats_{input} \cup formats_{output}) \end{aligned}$$

Figure 4.5: Formula to calculate the lowest bandwidth format out of a given set of in- & output formats. The $bitrate_{signal}$ is the amount of bits a single sample requires, while the $bitrate_i$ is the $cost_{format}$ of the format i.

$$\alpha(i,k) = \begin{cases} 1 & ,i = k \\ 0 & ,i \neq k \end{cases}$$

$$resamplings_{format} = \sum_{f=1}^{F} \alpha(format, f),\ f \in formats$$

$$format_{optimal} = argmin_f(resamplings_f),\ f \in formats$$

Figure 4.6: Formula to calculate the lowest bandwidth format out of a given set of in- & output formats. The $bitrate_{signal}$ is the amount of bits a single sample requires, while the $bitrate_{format}$ indicates, how many bits a second of this signal require.

there may be several formats which fulfill this condition. If this is the case, it comes naturally to mind to choose the format which requires the least amount of bandwidth, so the algorithm formalized in figure 4.5 is used to choose the optimal format out of the formats which require the least amount of resampling.

After the Orchestrator determined the transmission format for each topic, it will inform all registered components about their respective internally used audio formats via a ROS-message. The library part of each component will listen to these messages and then store and use these formats as discussed in chapter 4.1. It is important to note at which points in time the Orchestrator will determine the audio formats. Since it cannot conclusively determine if all components that are going to be started have already registered, as it lacks a list of such components, whenever a component registers, the formats of all topics are determined anew. Additionally, a component may die or shut down at any point in time. When this is detected, all formats are determined as well, since the Orchestrator cannot foretell how long the pipeline will still run in the current configuration. More convenient audio formats may save a lot of bandwidth or computation time, or improve the quality of the transmission, by not unnecessarily resampling and converting audio.

Determining whether a component has shut down must be conducted by the Orchestrator. For this reason it will routinely ping all components via ROS's inbuild ROS-node pinging functionality. Should ROS not find a node in question, or should the node be non-responding, the Orchestrator will remove it from its internally kept list of active nodes and terminate all communication with it. Such nodes will furthermore not be considered for fusion of results anymore.

Result Fusion

When synchronizing results, a few key circumstances need to be kept in mind. The first and probably most important one is that because of the library and the efforts to annotate the audio and in turn results

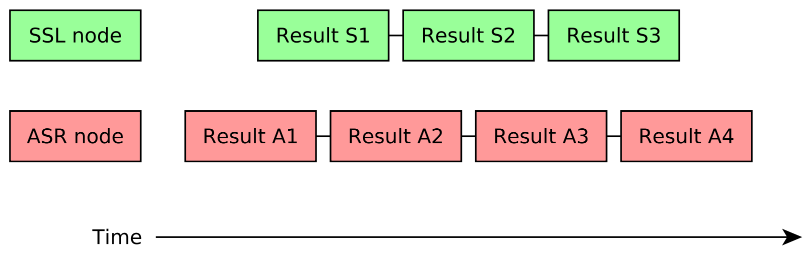

Figure 4.7: Overlap of results from two distinct components, an SSL component in green and an ASR component in red. Trying to synchronize as much results as possible may result in a macro result containing all presented data.

with timestamps, the synchronization of the results itself is rather trivial. However, there are still a few possible pitfalls. Most importantly, the Orchestrator can not know with certainty when results are arriving, or -more precisely- when all of the results that need to be synchronized will have arrived. This is rather unfortunate, as it is of utmost importance to relay all information as soon as possible. This is to keep further programs reliant on the synchronized results, such as natural language processors or robot behavior controllers, not waiting and thus keep latency down to a minimum. There is another important question, which has been put aside up until now. When results are synchronized, with respect to what are they synchronized? After all, results of different types may overlap within the time frame of their respective occurrence. An example of this phenomenon can be observed in figure 4.7. The first result of an ASR node, A1, may be fused with the first result of an SSL node, S1. This however occurred partially with A2, which may be also fused for the result. This could iteratively go on until all shown results may be fused together. Naturally this is not desirable, as the latency of the first results would needlessly increase and no meaningful information could be gained by this synchronization.

A few approaches to deal with this problem may come to mind. It could for example be chosen to synchronize with respect to each recorded audio message. But due to the library not enforcing a singular, constant size for the audio transmitted via these messages (see chapter 4.1 and 2.1), this could quickly explode in complexity. Especially so since a beamformer may create a new artificial audio signal which may transport smaller audio signals with each message. Alternatively it may slightly shift the timestamps of the messages, due to factoring in the time-of-arrival differences for the different microphones. This way, all -or worse, just some- of the results may have been computed on timestamps differing from the originally recorded ones.

The approach ultimately chosen was to determine one so called anchor type, which would then be the basis of all synchronization. Each result received of this anchor type would by their timestamps determine the timeframe with which results of all other types would

VAD	VADInfo
SpeechRec	SpeechInfo
SSL	SSLInfo
Gender	GenderInfo
Emotion	EmotionInfo
VoiceId	VoiceIdInfo
Other	None

Figure 4.8: Possible designations of components along with result messages (see figure 4.2) expected from them.

then be synchronized with. Any result occurring at least partially in this timeframe would be included in the fused result. This approach is similar to [Bla+92], in that my approach also batches results. Unlike [Bla+92] however, in my approach batching happens dynamically, based on the anchor result and not on a pre-determined timeframe.

Initially it was planned to only use speech utterances as anchor. However, as time moved on, I realized that this may not be the only interesting type for synchronization. Programs for tracking persons for example could be very interested in obtaining data on which particular person was speaking at a given moment, synchronized with their position provided by SSL information. Therefore the algorithms of the Orchestrator would quickly be adapted to not only be able to use every result type as anchor type, but to do so concurrently. Thus any result type of particular interest can be provided without giving up synchronization with other results.

In chapter 4.1, the fact that each component will register itself giving a designation was discussed. A list of these designations can be seen in figure 4.8. This designation corresponds to a result type, seen on the right side of the figure. Through this connection the Orchestrator can know to expect certain result types from each component of a pipeline. A special case in this is the `Other` designation, which should be assigned to utility components, such as an audio recorder, or a playback node. The Orchestrator will not expect any results of these components, but still take them into account when determining audio formats.

The Orchestrator may receive results in any order. However, fused results should not be published, until all results of interest were received. When the Orchestrator receives a result of one of his anchor types, results of interest may already have been received, or are still to be received. Thus a way to gather all these results is needed. To take care of all results the Orchestrator has already received, it was chosen to create a database, in which all incoming results are saved. This may seem a little over-engineered, but will later be of additional service.

When the Orchestrator receives a result of its anchor type, it will then create a so called `Fusion` object. This object will store all relevant results and additional information used to fuse the results. It starts by querying the database for all results which occurred in the timeframe predefined by the anchor result. It will then continue to collect incom-

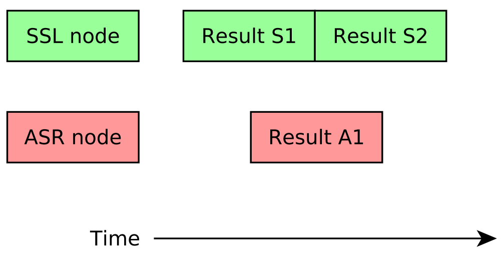

Figure 4.9: Complete overlap of results. If the ASR component publishes the anchor type, and these need only be synchronized with SSL results, it can be determined that all relevant information was gathered as soon as *Result S2* was received.

ing results, given that they are eligible for fusion until it determines that no new results will be received.

Fusion Conditions

This can happen in either of two ways. The first, straight forward way is the case if all possible results are already present. As the timestamps of the audio is continuous, the results can be as well. Such is the case in figure 4.9, where the results produced by the SSL component are continuous and do not allow for another result of the same type to occur. If this is detected, the results may be fused and published as an synchronized result immediately.

The second method to determine if no more results will be received is more complex. Components should not be expected to produce results for every bit of sound they receive. More explicitly, components should be expected to sometimes not produce results, e.g. if they are unsure of their result or if they could not detect something. As an example, consider an SSL component to produce a result based on a shattering glass. Other components, especially components requiring an actual utterance or at least a voice to be present in the signal should not detect anything during this time and thus not produce any results. This would make synchronizing the generated SSL results with anything other impossible. Additionally, it must be kept in mind that components may unexpectedly shut down. Synchronizing their then not produced results would be impossible. A component may also struggle to calculate one specific result, due to it being either very computationally expensive or due to having a temporary shortage of computational power, because e.g. other programs consuming much computational power.

Fusion Timeouts

To counter this problem a timeout is needed that will tell the Orchestrator to abandon any results not yet received. This is somewhat contrary to the previously discussed goal of ensuring the synchronization of all results, but is crucial to ensure all results to be actually relayed to further components and arrive there with minimum latency. Finding a feasible value for this timeout is not trivial, as very broad circumstances can have an impact on it. Consider for example a setup running over a slow, but very stable network connection, or a number of components running on very weak hardware. In both cases the timeout should be bigger than normal.

An optimal timeout may also be dependent on the components used in a specific pipeline configuration. Consider a setup with a component which is very fast in computing results and one which takes a long time for this. In the case the fast component did not produce a result, a long timeout would lead to a higher latency than a small one. A large timeout would however be needed for the component which takes a long time to produce results.

The method I developed to find the optimal timeout takes into account all of these concerns. Making use of the database included to save results, not only received results are saved, but also their time to compute. This can be calculated easily, as this is just the difference between the time the result is received by the Orchestrator and the end time of the audio timestamp. This gives the absolute time needed from recording the audio to the point in time at which results are available to the Orchestrator. Because this time is generated for each result, the average for each type of result can then be calculated. Thus a timeout for each type of result can be generated, which is equivalent to a timeout for each component. To avoid ignoring results which take shortly longer to arrive at the Orchestrator than the average, the average value is multiplied with a constant factor to gain the final timeout. I found a factor of 1.5 to work quite well in this regards.

These timeouts are generated for each anchor result anew. Thusly, the Orchestrator can react to possible changes in the execution times of components and adjust the timestamps on the fly. However, the first anchor result the Orchestrator receives will not be able to generate timeouts this way, thus an alternative is needed for it. I found a hard coded high timeout and the potentially higher latency resulting on this first result to be acceptable. However, at this point another feature of the database approach from before becomes relevant. As results are stored permanently, it is possible to simply load an already existing database. If the configuration of the used pipelines have not changed too much with respect to components and network or compute environment, the Orchestrator can make use of the previous runs and thus eliminate the drawback of the first anchor result.

Latency caused by Fusion

Lastly the latency introduced by synchronizing results is worth a look at. In a best case scenario, where the Orchestrator can make use of overlapping results (as shown in figure 4.7), the latency is simply the time of the slowest computing component, along with a calculation time the Orchestrator introduces. The worst case scenario can be seen in figure 4.10. This latency l_f is mostly determined by the exact factors discussed previously: the maximum of the average time to

$$l_f = V_A \cdot max(\{\frac{1}{n_i} \cdot \sum_{k=1}^{n_i} l_{i,k} \mid i \in components\})$$

Figure 4.10: Maximum latency of a single fusion. V_A is a constant. n_i is the amount of results the orchestrator already received of a component i. $l_{i,k}$ is the latency of the kth result of the component i.

compute of all components and the factor V_A. I consider this to be a worthwhile trade off, due to the most important anchor message types only being generated when an utterance or at least voice is present in the corresponding audio. I believe these important anchor message types to be that of the ASR-, emotion-, gender-, and voice-results.

4.3 DEVELOPED & INCORPORATED COMPONENTS

In this section, components that were developed for the proposed pipeline will be discussed. These components cover different aspects of speech recognition.

One good example would be a DeepSpeech component I developed[1]. It makes use of a TensorFlow [Mar+15] implementation of Baidus Deepspeech [Han+14].

Some initially planned and partially developed components were left out of this list, because it became clear they could not be included into this thesis in a meaningful way. Thus development for these components was stopped or they were left out in favor of diverting time to more important parts of this thesis.

A handful of components are included in the library's repository, as they provide utility functions or are commonly used. None of these components provide any information to the *Orchestrator*, apart from registering. These components are:

Audio Grabber The *Audio Grabber* is the most fundamental component, as it is the basis of most pipeline configurations. It is able to grab audio from a microphone via ALSA and feed it into the pipeline.

Audio Player This component is the *Audio Grabbers* counterpart, as it receives audio data from the pipeline and outputs it via ALSA through a speaker. As such, it is mostly used for debugging purposes and to enable quick auditory microphone checks.

Channel Splitter As discussed in section 4.1, the proposed pipeline's library does not support mixing of channels. The *Channel Splitter* is used to mitigate this absence by receiving a multichannel audio signal and producing a corresponding number of single channel outputs. One possible use case is to split a multichannel audio needed for SSL into single channel audio usable for speech recognition, when no beamforming is desired.

Recorder The *Recorder* is able to write audio it receives via the proposed pipeline to a file. Its foremost usage is to save audio it receives for later inspection or analysis. This can either be done for documentation purposes or, as was mostly done during development, to check if the pipeline transmitted and resampled audio correctly.

Additional components not included in the libraries repository cover the topics of:

Wav Player

The *Wav Player*[2] fulfills the same role as the *Audio Grabber*, as it feeds audio into the pipeline. However, it will read a given set of wav files

1 https://github.com/Slothologist/DeepSpeech4Ros
2 https://github.com/Slothologist/esiaf_wav_player

in and output them into the pipeline. It is capable of either producing silence in between each wav file or waiting for a configurable amount of time before playing the next file. As such, it's predominant use case is to enable evaluations of other components with the help of data sets.

Sound Source Localization

I developed a component[3] which performs SSL. It uses the python library `pyroomacoustics` [SBD17], and specifically its SRP algorithm, though usage of all other SSL algorithms provided by `pyroomacoustics` can be configured. For each sound chunk the component receives via the proposed framework, it creates SSL results and then sends them to the Orchestrator.

A slight variation of this component[4] will be used in one of the experiments of the evaluation chapter (see 5.2). This variant will, instead of publishing results for each sound chunk it receives, store these results in a queue. To communicate the results it provides a ROS service which allows any component to ask for results which occurred in a specified time frame. Additionally, this variant will not receive audio from the proposed framework, but instead capture it from a microphone via ALSA.

Voice Activity Detection

The VAD I developed[5] is a reimplementation of a pre-existing algorithm present in the PSA (see chapter 3.3.7). Its segmentation is therefore virtually identical to the PSA's.

If it detects the end of a segmentation, it will enhance the corresponding audio chunk with a `segmentation_ended` annotation, as described in chapter 4.1, before sending it. Audio will not be transmitted to the next component(s) if it were not found to include speech.

Automatic Speech Recognition

The ASR component I developed[6] is a very simple wrapper around the python wrapper of PS. It will feed all audio it receives into PS, which will then dynamically produce results. However, these results are only used if a VAD signals the end of a segment. As such, this PS component requires a VAD to work. Whenever a result is produced, it is sent to the Orchestrator via published ROS message (see chapter 4.1).

Emotion Recognition

The emotion recognition component I developed[7] makes use of `Speech Emotion Recognition` [har19]. `Speech Emotion Recognition` uses a

3 `https://github.com/Slothologist/esiaf_doa`
4 `https://github.com/Slothologist/ma_baseline_doa`
5 `https://github.com/Slothologist/AudioSegmenter`
6 `https://github.com/Slothologist/esiaf_pocketsphinx`
7 `https://github.com/Slothologist/esiaf_speech_emotion_recognition`

neural net to extract the emotional state of a person based on their speech. As such, it is capable to categorize speech into three emotions: angry, happy, and sad. Additionally, speech can be categorized as neutral. Small changes were required for it to meet all the proposed framework's expectations, so I changed it to include a probability for each result. Each sound chunk this component acquires will be fed into `Speech Emotion Recognition` which will return an emotion and probability for each of these sound chunks. These results are then sent to the Orchestrator for further synchronization.

Gender Recognition

The gender recognition component[8] I created is actually a somewhat new development. It is based upon `Speech Emotion Recognition`, as it uses a nearly identical neural net and dataset. However, I adjusted the output dimension of the neural net and reorganized its training data to classify either to male or to female, instead of the four emotions. It also includes my changes, so its results include a probability. I retrained the net and achieved a training accuracy of 99.63% while maintaining a test accuracy on unseen data of 91.20%. Training and test data was randomly split 80:20. Similar to the described emotion recognition component, it will produce a gender result for each sound chunk it receives along with a probability for it and send them to the Orchestrator.

It should be noted that the data set was quite small with only 339 audio samples, 188 of which by 5 female speakers , 151 from 5 male. I suspect this being the cause for it to not achieve the level of accuracy on real world data that it achieved on the test data. This observation is however anecdotally and was not evaluated formally. However, I was satisfied with this solution, as it produces somewhat reasonable results.

It should be noted for all these developed components that it is not a part of this thesis to develop new approaches to recognition of gender, emotion or speech. I chose this course of action with this specific component however, because an almost feasible component in the form of the described emotion recognizer was already in place and retraining of its neural net seemed to be a quick and easy way to produce a new component. The alternative would have been to restructure and adjust a different approach, which may not have worked at all.

8 https://github.com/Slothologist/esiaf_gender_rec

5 EVALUATION

In this chapter, two experiments will be presented which were used to evaluate the proposed speech recognition pipeline. First, different configurations of the proposed speech recognition pipeline will be compared against each other and against a baseline in the form of an already existing solution. This test will focus on performance and computational cost. Second, a configuration of the proposed pipeline will be compared against an existing solution in a slightly altered RoboCup@Home Speech- & Person Recognition task with focus on sound source localization results and their fusion with speech recognition results.

5.1 ASSESSING PERFORMANCE ON A COMPREHENSIVE DATA SET

In this section, an evaluation of the proposed pipeline and an pre-existing solution on a pre-existing data set will be presented. The goal of this evaluation is to take a closer look at the performance of the proposed pipeline with the explicit focus on computational cost as well as time needed for recognizing speech. Recognition percentage will be used as control variable, to ensure equivalence between pipelines.

Based on the additional features the proposed pipeline introduces, most prominently synchronization of speech results and the modular nature of the pipeline (see chapters 1 and 4.1), the proposed pipeline is expected to consume more CPU power as well as taking longer to compute results in comparison to the pre-existing solution. For further examination, the cost different additional components will add to the pipeline will be inspected to determine the scaling of the proposed pipeline. Due to the pre-existing solution being based on PS and the increase in available CPU power since its first release (which was in 2012), I consider a steep increase in CPU power to about five to ten times as a success.

I will lastly discuss if this additional cost renders the pipeline unsuitable for real world applications or if the benefits outweigh these costs.

5.1.1 *Tested Pipeline Configurations*

Pre-Existing Solution

The pre-existing solution only consists of the PSA, see figure 5.1. The PSA grabs audio from a microphone using ALSA, filters the audio using an integrated VAD and finally uses, as its name suggests, PS to recognize speech. Results are then published via ROS.

The PSA is and was used extensively for several years in the RoboCup@Home setup of Team ToBi [09]. In this evaluation, it is

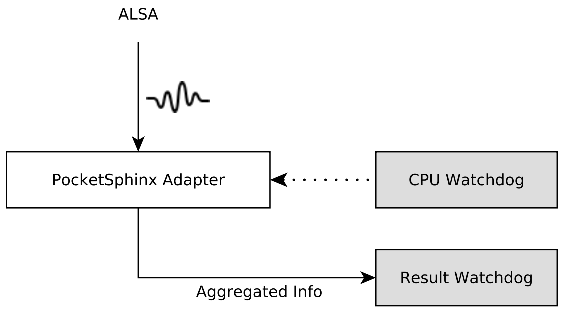

Figure 5.1: Test scenario for existing pipeline. The PSA acquires audio data from ALSA and provides speech recognitions. A CPU and result watchdog monitor and log its CPU usage and results respectively.

primarily used to provide a baseline for the proposed pipeline and context for the acquired results.

Basic Pipeline

This configuration of the proposed pipeline is intended to reassemble the PSA as closely as possible. As seen in figure 5.2, it consists of a VAD, a speech recognizer (using PS), the Orchestrator and a Wav file player to feed audio into the pipeline. Recognition results are gathered by the Orchestrator and then communicated via ROS.

Elongated Pipeline

This configuration of the proposed pipeline is nearly identical to its baseline, but incorporates a dummy component to evaluate how much overhead in time and CPU cost an additional processing step produces. As indicated in figure 5.3 and by its name, the dummy node does neither alter nor compute information on the audio data it receives, but instead just relays it from the WAV player to the VAD.

Widened Pipeline

This configuration of the proposed pipeline is nearly identical to its baseline, but incorporates a dummy component to evaluate how much overhead in time and CPU cost an additional information provider produces. As indicated in figure 5.4 and by its name, the dummy node receives audio data from the VAD and works basically in parallel to the speech recognizer. Upon receiving audio data, the dummy node provides hard coded information to the Orchestrator. This way the Orchestrator does not have to wait on receiving the dummy node's information during its synchronization steps, and can theoretically provide synchronized data as soon as speech recognition results are provided to it.

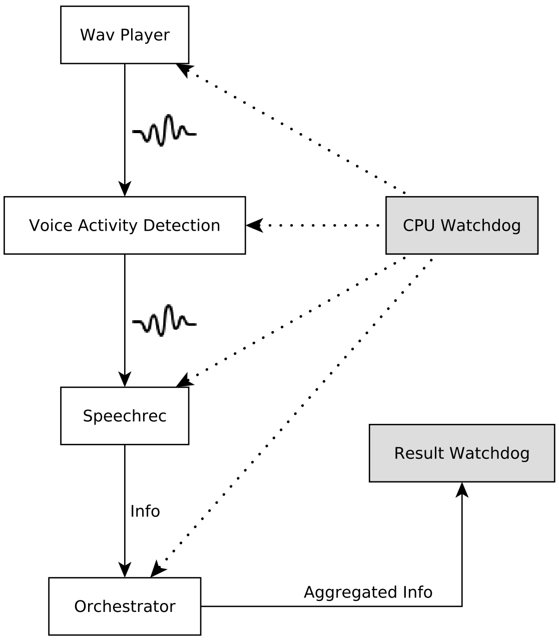

Figure 5.2: Test scenario for proposed pipelines baseline. It consists of a WAV player, which provides audio data. The audio is then processed by a VAD and a speech recognizer. The Orchestrator gathers the speech recognizers data and provides it to a result watchdog, which then saves the results. A CPU watchdog monitors and logs CPU usage of all components.

Realistic Pipeline

This configuration of the proposed pipeline resembles its baseline, but includes two additional information provider. One provides information on the gender of the speaking person while the other provides information about their emotional state. As shown in figure 5.5, both additional components run in parallel to the speech recognition and provide the Orchestrator with their information.

This configuration of the pipeline is intended to resemble a realistic use case of the proposed pipeline, in that several audio analysis results need to be synchronized.

Data set

The data set (see figure 5.6) used for this evaluation consists of 1723 samples ranging between a half and fourteen seconds. It incorporates twelve speakers (male and female) speaking 24 phrases. Individual samples were recorded using two microphone types, omni directional and cardioid. They were recorded in two rooms, some of them with noise, others without.

To more easily feed all samples into the pre-existing solution, the WAV files containing the samples were concatenated into one big WAV file, with three seconds of silence added in between the individual samples, to easily distinguish the samples from one another. This

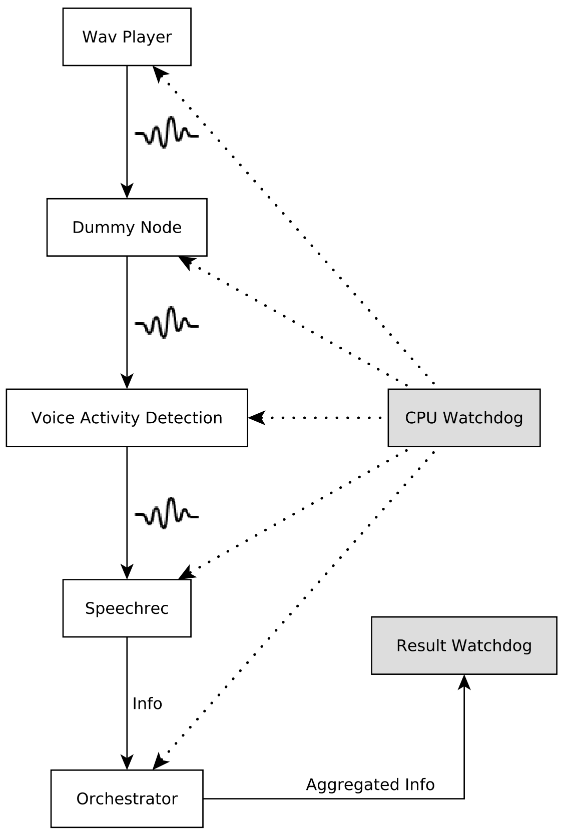

Figure 5.3: Test scenario for an elongated configuration of the proposed pipeline. In comparison to the baseline of the proposed pipeline, it encompasses an additional dummy node which is placed in between the WAV player and the VAD. It does not alter the audio in any way, but is monitored by the CPU watchdog as well.

also simplifies matching the recorded speech recognition results with the actual utterances. The proposed pipeline was fed each sample individually, but three seconds of silence were similarly inserted into the audio. This difference was necessary, as the pre-existing solution had to be fed with a single WAV file for reasons later discussed in detail. Feeding each sample individually into the proposed pipeline enabled us to keep track of which file was played at any given moment by publishing information about each WAV file after it played, which in turn increased each results assignability to said WAV file. The complete data sets playtime including the silence amounted to nearly two hours and fifty minutes.

Setup

Two slightly different setups were used for the various configurations of the proposed pipeline and the existing solution, due to differences in acquiring audio data. What they have in common is that all pipelines were tested on the same computer, a Thinkpad Carbon X1 with a Intel(R) Core(TM) i7-7500U, to ensure equal CPU performance.

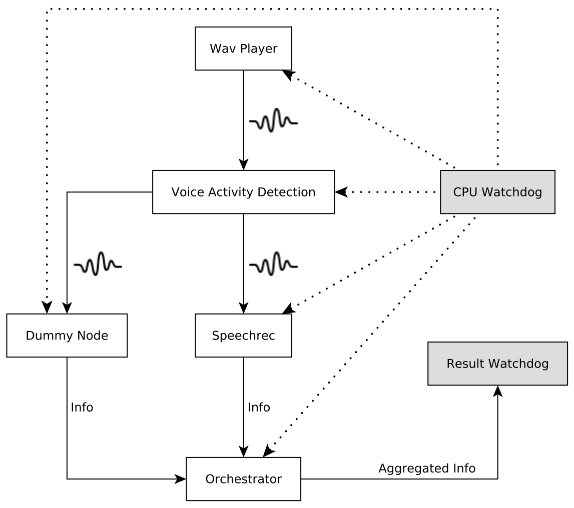

Figure 5.4: Test scenario for a widened configuration of the proposed pipeline. In comparison to the baseline of the proposed pipeline, it encompasses an additional dummy node which is set in parallel to the speech recognizer and provides hard coded information to the Orchestrator, which then synchronizes these information with those it received from the speech recognizer.

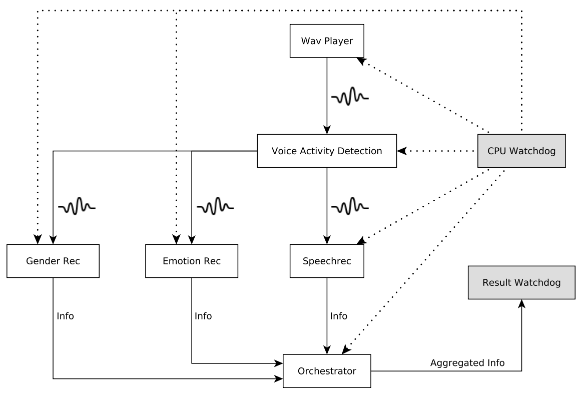

Figure 5.5: Test scenario for a realistic configuration of the proposed pipeline. In comparison to the baseline of the proposed pipeline, it encompasses two additional components which are set in parallel to the speech recognizer, namely a gender and emotion recognizer. Both provide additional information to the Orchestrator, which then synchronizes these information with those it received from the speech recognizer.

Samples	1723
Phrases in Kitchen	1149
Phrases in Wardrobe	574
Noisy Phrases	575
Clean Phrases	1148
Individual Phrases	24
Speakers	12
Female	5
Male	7
Runtime (w/ silence)	2:49:23
Runtime (w/o silence)	1:23:01
Language	German

Figure 5.6: Detailed Information about the Data Set.

All setups were comprised of equivalent components. If not otherwise stated, all variations of the proposed pipeline used the exact same components with identical configurations (safe for some minor details, e.g. a modified path for the audio between the WAV player, dummy node and VAD in the elongated version of the proposed pipeline, see figure 5.2 and 5.3).

The existing pipeline's PSA internally incorporates a VAD, which was reimplemented and used as the VAD of the proposed pipelines. It uses PS for speech recognition, as does the speech recognizer used in the proposed pipeline. Both PS instances use the same speech models, dictionary, and grammar.

In both cases two small scripts were used to record the components CPU usage as well as speech recognition results and timings. CPU usage was universally recorded using the python library `psutil` [Rod+13] and its `cpu_times` functionality. However, `psutil` reliability can be called into question, as it seems to not take modern CPU's capability to increase or decrease their clock speed, depending on load, into account. As such, the recorded CPU costs should be taken as rough estimates and not as exact measurements. Nonetheless, these data points are included despite their supposed inaccuracy because apart from one tested pipeline, all other pipelines did not produce enough load on the CPU to cause it to increase its clock speed. This is supported by the fact that identical components across various pipelines produce very similar CPU loads, as seen in figure 5.8. Incidentally, a process producing the computational load of one CPU second over any timespan is equal to the process running on one core of a CPU under full load for one second. Speech recognition results were recorded by a ROS node which collected ROS messages published by the proposed pipeline's Orchestrator and the PSA respectively.

Recording

The times needed to calculate results were recorded in two different ways. Due to the fact that the proposed pipeline annotates its results with the time the audio was recorded, it is rather simple to calculate the time needed for recognition. One can simply subtract the time when the analyzed audio signal ended from the time the synchronized result was received by the recording component and thus, get the total time needed for recognition. To be precise, this way the time from when the segmentation ends to the point where the recording node received the recognized speech result is measured.

The method of feeding audio into the proposed pipeline does not matter as long as the timestamps of the audio are correct. So instead of grabbing the audio via a microphone a WAV file player was used to feed the data set directly into the pipeline.

The existing solution however was in need of a small workaround to work on audio files rather than audio directly grabbed by a real microphone. A virtual microphone was needed to feed the data set into the PSA. This virtual microphone provided access to a WAV file containing the data set. As information about singular utterances inside the data set could not be propagated through ALSA and the PSA, a timestamp is recorded with the start of feeding the WAV file into the virtual microphone and thus in turn into the PSA. Speech recognition results were then recorded along timestamps acquired when these results were received in the dedicated ROS node. Supplementary, while concatenating the data set into the singular WAV file, annotations were created to keep track of when each sample ended and which phrase it contained. To achieve comparable results, slight modifications to the internal segmenter of the PSA were made, i.e. timestamps on each successful segmentation would be recorded.

By combining the information provided by 1. the timestamp created during the start of feeding the WAV file, 2. the timestamps recorded when recording the speech recognition results and 3. the timestamps after each successful segmentation, the absolute time needed by this solution could be computed. This is mostly done by subtracting the timestamps, collected after successful segmentation, from the corresponding timestamps, recorded when the results were received. Thus, the time from the segmentations end to receiving the results is recorded, as previously with the proposed pipeline.

In any case, speech recognition results and timings were recorded as raw data and written to files. Afterwards, these raw results could be processed and evaluated with the help of further scripts.

5.1.2 *Discussion*

If one compares the recognition rate of all the pipelines in figure 5.7, the various configurations of the proposed pipeline all share virtually identical results, which however differ from the PSA's result. This is somewhat unexpected, as special care was taken to ensure both speech recognizers use not only PS, but also equal configuration files, including speech model, dictionary and grammar. The audio segmentation for the proposed pipeline is a reimplementation of the one used internally in the PSA, thus they should provide equal results. Furthermore,

Pipeline	Recognition percentage	Absolute time till result	Sum CPU time
Pre-Existing	77.02%	0.063 sec	228.99 sec
Proposed	99.77%	1.014 sec	1209.30 sec
Elongated	99.77%	1.039 sec	1399.63 sec
Widened	99.65%	1.090 sec	1665.66 sec
Realistic	99.48%	1.780 sec	37517.13 sec

Figure 5.7: Results of the different pipelines in comparison. Shown are recognition rate, average time for recognition and consumed CPU time.

Component	Pre-Existing	Proposed	Elongated	Widened	Realistic
Speech Rec	247.44	546.65	545.73	574.82	285.77
Orchestrator	-	107.66	114.69	415.73	962.56
Wav Player	-	380.11	378.71	405.40	138.22
VAD	-	174.88	188.56	191.97	51.26
Dummy	-	-	171.94	77.74	-
Gender Rec	-	-	-	-	5959.50
Emotion Rec	-	-	-	-	30119.82

Figure 5.8: Detailed CPU cost by pipeline and component.

they were basically bypassed in that the silence in between samples enabled the usage of very lenient VAD configurations.

The proposed pipeline's PS component is written in Python and uses its PS wrapper, which in turn makes use of the same C++ libraries as the PSA, so it would be difficult to claim this as the determining difference responsible for this discrepancy. Clearly more testing is needed to produce substantiated assumptions about the difference in recognition results.

If one compares the results of the proposed and elongated pipeline in figure 5.7, a slight increase in absolute time per recognition can be seen, as well as a moderate increase in CPU cost. The increase in CPU cost is virtually completely due to the additional dummy component present in this pipeline, as can be extracted from figure 5.8.

The slight increase of 15ms in absolute recognition time was, as discussed before, expected. Considering a very elaborate configuration, where audio would be transmitted through audio grabbing, sound source localization, beamforming, audio enhancement, VAD and speech recognition, the latency introduced by using the proposed pipeline could be estimated to have an upper bound of $15 \cdot 6 = 90$ms. Depending on the time taken by the components themselves probably be just a fraction of the absolute recognition time needed.

The widened configuration of the proposed pipeline is the first discussed configuration in which the Orchestrator has to actually synchronize data. In the baseline and elongated configuration it could take advantage of not having any other information provider and just relay all incoming data. The additional computational power required by the Orchestrator can easily be observed in figure 5.8. Though, this configuration of the pipeline seems to generally have higher computational load, probably due to inconsistencies caused by `psutil`, as discussed in section 5.1.1. The Orchestrator is responsible for around a quarter of the computational load of the whole pipeline compared to roughly a tenth in the baseline and elongated configuration.

The moderate uptick in absolute recognition time may also be explained by the additional synchronization work the Orchestrator has to contribute. This increase of 86ms in itself should not be particularly noticeable as it increases the absolute recognition time by just around 8 percent, which could be seen as a valuable trade-off for gaining synchronized speech results. However, further tests could be carried out to investigate the scaling of this particular part of the proposed pipeline.

Regarding the realistic pipeline, the absolute time for recognition saw a increase in comparison to the baseline of the proposed pipeline, as can be seen in figure 5.7. Thanks to the Orchestrator saving all received information inside its database, as described in chapter 4.2, it was possible to inspect the latencies of each component individually. The emotion and gender recognizers latency proved to be three and thirteen times as large as the speech recognizers, which can be supported by inspecting their CPU cost in figure 5.8. This leads to the assumption that the defining factor for the increase in time needed for recognition is actually the result of these components needing more time to compute their results. As this is an expected feature of the proposed pipeline, the longer time for recognition is a trade-off for having synchronized results.

The baseline of the proposed pipeline consuming at least six times as much CPU power as the existing solution, as can be seen in figure 5.7 may seem like an tremendous increase. But in comparison to the playtime of nearly 2 hours and 50 minutes the CPU time cost of circa 20 minutes and 4 minutes for the proposed pipeline and the existing solution respectively are both more than acceptable, especially considering that most modern CPUs have more than one core and this load is shared between them.

Additionally, one should keep in mind that PS was designed years ago to work on mobile devices of that time, so it is one of the computational most inexpensive speech recognizers still used. Despite this, it is the computationally most demanding component of all the proposed pipelines configurations apart from the realistic one.

More modern speech recognizers can generally be divided into offline and cloud services. Cloud based speech recognizers, such as Google [Goo13] or Microsoft [Mic18] speech to text services produce varying degrees of latency, as audio must first be streamed to distant data centers before it can be processed and the results can be sent back. Depending on the available Internet connection and the time spend analyzing the audio, these approaches typically take at least as long as fast offline approaches.

Modern offline approaches mostly focus on deep learning at the time of writing. As such they often use elaborate neural networks which can only be run on graphics cards in a timely manner and are computationally exceptionally costly.

The results of the realistic version of the proposed pipeline reflect this. Both the gender and emotion recognizer used in the pipeline use rather simple, but nevertheless computationally costly neural networks. Still, they exceed the cost of all other pipelines by more than a factor of 20.

Based on this and the fact that all components of the proposed pipeline can easily be swapped out, the actual resource cost the components themselves introduce may not even matter that much, as each component could easily be replaced, either by one performing better but being less resource efficient or vice versa.

As can clearly be seen in figure 5.7, each configuration of the proposed pipeline takes over a second to compute speech recognition results, which seems extremely high in comparison to the existing solution. Considering the insights procured with the elongated version of the proposed pipeline, recognition time should in the presented configurations never increase by over 100ms due to audio transferring.

The argument could be made that due to the recognition results being different between the PSA and proposed pipelines' PS, both implementations could behave fundamentally different from one another. This could explain the proposed pipelines' PS component to perform better while also consuming considerably more CPU time. But even when scaling the PSA's recognition time to match the six times higher CPU cost of the proposed pipeline, the PS component's part of the recognition time would only amount to around 320ms.

In a worst case scenario, summing up these upper bounds on added recognition time would result in an average time of about 420ms. Due to the actually measured time being more than twice that, I suspect some sort of bug to be responsible for this behavior, which sadly could not be eliminated because of time constraints and would be material for future work.

Regardless, as the results of the widened and elongated configurations of the proposed pipeline indicate, this bug appears to only add a fixed amount of time needed for recognition and to not scale with the number of its components.

To sum up, a number of key insights could be gained by this experiment. The recognition rate as control variable showed that the pipeline can not be compared to the pre-existing solution without restriction. As a number of factors, i.e. all components of the pipeline apart from the PS speech recognizer, were independent from this control variable, I felt confident to compare them nevertheless. That being said, the proposed pipeline was shown to have an overhead in CPU cost and computation time. Future work may determine if this overhead has a negative impact on HRI, and if so, how much.

5.2 robocup speech recognition test

In this section, an evaluation of the proposed pipeline in a real world scenario will be presented. The goal of this evaluation is to assess if using the proposed pipeline leads to improved recognition results, in this particular case to increase the accuracy of sound source localization results. These could be achieved through the result synchronization the proposed pipeline provides.

This evaluation will be based on the Speech & Person Recognition task of RoboCup@home 2018 [Mat+18], although some minor modifications will be made to the setup. The evaluation will be conducted on the Pepper robot[1].

Software Setup

Two distinct solutions will be benchmarked: the existing RoboCup setup and a reimplementation of it within the proposed pipeline. Both will mainly be comprised of an SSL, a speech recognizer and a layer to combine the results of both these components.

The proposed pipeline (see figure 5.9) will consist mainly of the basic speech recognition setup established as the baseline for the data set evaluation (see chapter 5.1.1), i.e. an audio grabber to record audio from a microphone as well as the previously (see chapter 5.1.1) established VAD, PS speech recognizer and Orchestrator. As

For the existing solution (see figure 5.10), the already established PSA will take the role of the speech recognizer. For SSL, a slightly modified variant of the proposed pipelines component will be employed, as elaborated earlier (see chapter 4.3). To combine both results, a behavior controller called Bonsai [Fre13] will be used. An channel splitter similar to the proposed pipeline is not necessary, as ALSA will assume this task for the PSA. Since result synchronization is not a feature of primary concern for Bonsai, its default behavior regarding this will be employed, which is to aggregate the SSL results from two seconds to half a second before a speech recognition result was received.

Naturally, similar components within both solutions will be configured equivalently. The output of both solutions will be logged and contain SSL angles and recognized speech. Both solutions will be run concurrently, to eliminate run-to-run variances between the solutions. To reduce reciprocal influence, the existing RoboCup setup will run on the robot, while the proposed pipeline will run on a secondary machine, except for the audio grabber node, which must run on the robot.

Description of the RoboCup@home Task

The RoboCup task would normally start with a small visual person recognition part. Since visual perception is not part of this thesis, this

1 http://archive.is/2VKYS

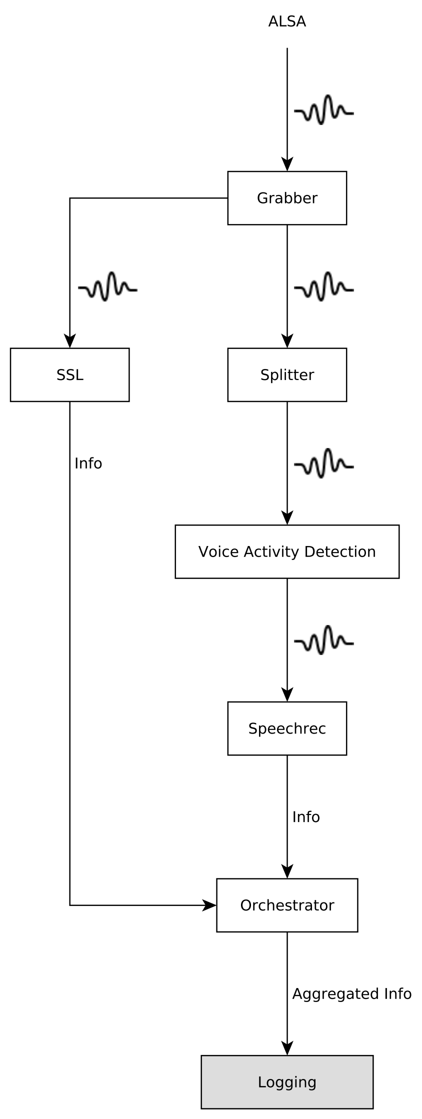

Figure 5.9: Setup of of proposed pipeline. Audio is only acquired from ALSA at one point, then fed into SSL and speech recognizer separately. Generated Information is gathered by the Orchestrator and provided after aggregation to a logging component.

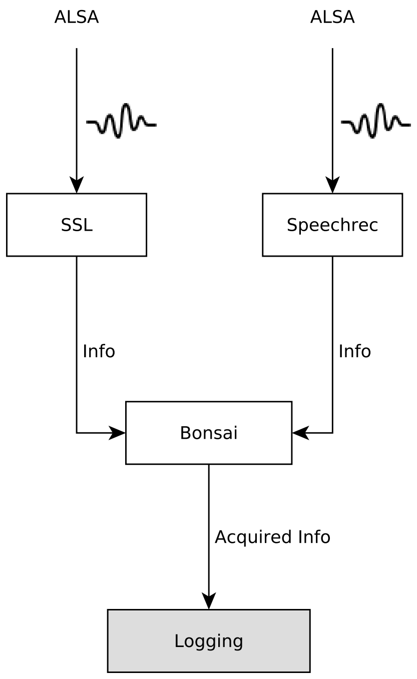

Figure 5.10: Setup of existing solution. Audio is acquired from ALSA by both the speech recognizer and the SSL component independently. Information from both is then collected and combined by Bonsai. Bonsai's output is then logged.

part of the task will be left out. The task will instead start with the *Riddles Game*, in which an operator standing in front of the robot will ask the robot five questions which it should answer.

Afterwards, between five and ten persons would surround the robot in the *Blind Mans Bluff Game*. Five random persons would then ask the robot each an additional questions. The robot should turn to the speakers and answer them. To ease the evaluation of SSL results, only four persons will partake in the *Blind Mans Bluff Game* of this evaluation. One person each will stand directly in front of the robot, behind the robot and to either side of the robot, with circa a $90°$ shift in angle with respect to the initial orientation of the robot (see figure 5.11). All questions for the task will be generated from the publicly available RoboCup 2018 command generator [Mat+15].

Simultaneous execution of both solutions requires small changes to the task. Since the robot can not turn to two different angles at once, instead of turning the robot to the speaker, logged angle information was used to evaluate SSL results. For similar reasons and since producing answers to the questions can be done rather easily, recognizing the correct question was counted as correctly answered. Both these information can easily be acquired by inspecting the logged output of both solutions.

To ease scoring of the task and provide documentation, the task was recorded by a video camera. The task was conducted eight times,

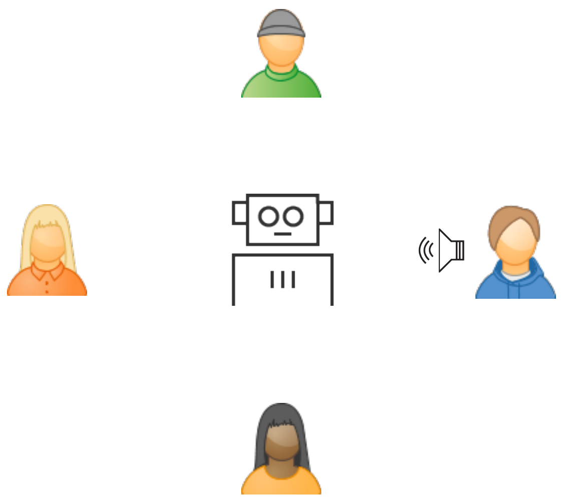

Figure 5.11: Test scenario for the RoboCup task, specifically the *Blind Mans Bluff Game*. One person each is locate in front, behind and to either side of the robot. Here, the person on the right is asking the robot a question.

but two of these runs were excluded from the evaluation, due to camera malfunctions. The recordings of the included runs were made available[2].

To simulate the environment preset on most RoboCup events, which are typically conducted in a crowded convention center and to provide non optimal conditions for the SSL, extensive noise was generated in front of the robot by two speakers. This noise is intended to increase the difficulty of SSL, to provide more meaningful results. A still image of the video recordings can be seen in figure 5.12. The two speakers seen in the foreground were playing restaurant noises, which consisted mostly of human chatter.

Pinpoint accuracy of SSL results is generally not required. For the purposes of locating and differentiating speakers from one another, especially in a household environment, where generally only a handful of people are present, any SSL result within a margin of error of 15° can be regarded as correct. Rather erroneous detections caused by noise before or after the spoken question should not falsely be included in an analysis on where a speaking person is located. As such, for the purpose of scoring both pipelines' results, any SSL result would be regarded as correct, if it was below the proposed 15° margin of error. Results were manually evaluated by combining information from the recorded videos, spoken questions and logged SSL recordings.

Discussion

The performance of the existing solution and the proposed framework can be seen in figure 5.13. When comparing results of both solutions, it

2 https://youtu.be/TrodN1UkQPM

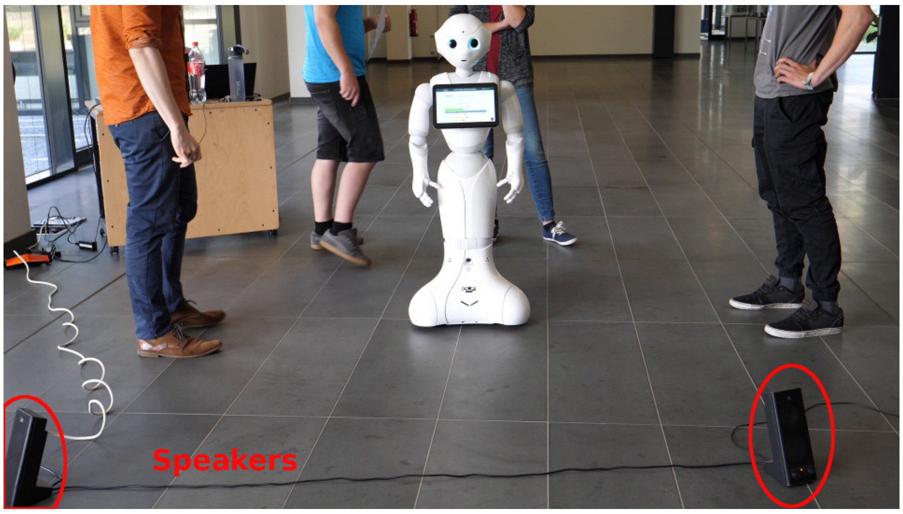

Figure 5.12: Task evaluation setup. Speakers can be seen in the foreground, while the test crowd shuffles around in between runs.

Run	Correctly recognized sentences		Points		Points by SSL		Avg SSL Error	
	Old	New	Old	New	Old	New	Old	New
1	50%	65%	45	50	0	0	83.4	86.5
2	50%	55%	40	70	10	20	92.1	46.6
4	35%	55%	25	35	0	0	116.0	87.9
5	40%	55%	30	50	10	20	98.4	54.3
6	45%	60%	40	60	10	0	97.6	82.3
7	60%	60%	50	60	10	10	105.2	51.9
Avg	46.7%	58.3%	38.3	54.2	6.7	8.3	98.78	68.25

Figure 5.13: Results of the pre-existing (Old) and proposed pipeline (New) in the RoboCup task by run. Run 3 and 8 were omitted because of camera malfunctions.

can easily be seen that similarly to the data set evaluation (see chapter 5.1.2), speech recognition results of the proposed pipeline outperform those of the existing solution.

The SSL results of both solutions may appear quite weak, but one should keep the noise in mind which was specifically generated to increase the difficulty of SSL. In a real world scenario, similar noise may come from a loud dishwasher or TV. A robot should however still be able to locate a person giving it commands when confronted by such challenging environments.

The proposed pipeline provided slightly more correct SSL results, which is to say it produced five opposed to the four of the existing solution where 30 were possible. Both factor in when comparing the actual scores of the RoboCup task, where the proposed pipeline outperforms the existing solution. However, I consider this score to not be very meaningful, as most of it is caused by the proposed pipeline's speech recognizer performing better.

Additionally, information about the average error of the SSL results were generated, as I consider these in this particular case to be more conclusive to the evaluation due to the score results being very close to one another. When comparing them, the proposed pipeline outperforms the existing solution in all runs but one. In run seven and two it (nearly) halves the error. Overall, the proposed pipeline reduces the error in angle by around 30° in comparison to the existing solution. I believe this to be the case due to the proposed pipelines enhanced synchronization abilities. Within the existing solution, shorter questions' SSL results may be random as Bonsai's SSL lookup time frame can completely miss the actual time in which the question was asked.

6 CONCLUSION & FUTURE WORK

The goal of this thesis was to improve HRI, by providing robot behaviors with synchronized results and thus enable them to focus on perceiving humans better. In chapter 3.1 I presented a similar framework to the proposed one, although focused on signal enhancement rather than result fusion, as well as fusion approaches, which were considered in the resulting system. Furthermore I showed the absence of speech result fusion with other audio analysis data in leading robotics teams. For this I created a framework which synchronizes and combines results of components of various disciplines, such as emotion and gender recognition. Smaller, secondary goals were also formulated: the solution was supposed to be modular, to increase its value for research. Naturally, the solution should also perform as good as pre-existing solutions with regards to computational speed and accuracy.

The RoboCup@Home experiment could show my fusion approach to not only work, but to improve the accuracy of the used SSL component in comparison to previously employed methods. In this particular area, this thesis can be considered fully successful. The data set experiment however is more of a mixed bag. It inherently showed the proposed framework to be quite modular by its ability to incorporate additional components and reuse nearly all components from the RoboCup experiment. On the other hand, it showed the proposed framework to be considerably slower then a comparable pre-existing solution. I briefly investigated this problem and could determine the long computation time to be not be explainable by just the components and framework alone. As such, I suspect an undiscovered bug to be he cause of this discrepancy. Future work may thus begin by exploring this inconsistency and -if possible- eliminate it.

Another path for future work could start with creating more components for the proposed framework. A number of the components I developed (see chapter 4.3) can not considered to be state of the art, but rather rapidly developed proofs of concept. For productive work and actual research in the fields of speech recognition and robotics, especially multi-modal sensor fusion, as outlined in chapter 3.2 it may prove fruitful to incorporate better performing algorithms into the proposed framework.

Another, rather unintentional feature of the proposed framework became clear when conducting the data set experiment. The proposed framework provides a very easy way to benchmark different components against each other by providing distinct interfaces and as such controls the environment of components to be benchmarked very tightly. This way algorithms to be benchmarked can be controllably fed audio and their impact upon other components inspected. It would for example be easily conceivable to test the word error rate of different combinations of VAD and ASR components.

Apart from this, the logical next step would be to use the proposed framework to feed information into a robot behavior. This way the work I presented can be properly utilized and thus improve HRI, the very motivation for this thesis. Due to this thesis, it is now possible to easily create behaviors in such a way that humans can be better perceived. To come back to the initial example of the interaction between the two referees and the robot in the RoboCup@Home, as shown in figure 1.1, it is now easily possible for the robot to map a spoken command via SSL to a visually perceived person. Provided the robot can visually or otherwise detect the referee disengaging from the conversation, it can act accordingly.

ACRONYMS

ALSA	Advanced Linux Sound Architecture. 12, 17, 31, 32, 46–48
ASR	Automatic Speech Recognition. 5, 6, 9, 12, 15, 16, 21, 22, 27, 29, 30, 32, 53
DoA	direction of arrival. 7
HARK	Honda Research Institute Japan Audition for Robots with Kyoto University. 9, 12
HRI	Human Robot Interaction. 1, 53
PS	PocketSphinx. 12, 13, 32, 35, 36, 40, 42–44, 46
PSA	PocketSphinxAdapter. 12, 13, 16, 32, 35, 36, 40–42, 44, 46
ROS	Robot Operating System. 8, 13, 16, 18–21, 23, 26, 32
SoX	Sound eXchange. 17, 18, 57
SOXR	SoX Resampler library. 17, 18
SSL	Sound Source Localization. 6–9, 11, 12, 15, 18, 19, 21–23, 27–29, 31, 32, 46–51, 53
VAD	Voice Activity Detection. 16, 18–23, 25, 28, 32, 36–38, 40, 42, 46, 53

BIBLIOGRAPHY

[09] *Team of Bielefeld*. RoboCup@Home Participants. https://www.cit-ec.de/en/tobi. 2009.

[AA09] Harm op den Akker and Rieks op den Akker. "Are You Being Addressed?: Real-time Addressee Detection to Support Remote Participants in Hybrid Meetings". In: *Proceedings of the SIGDIAL 2009 Conference: The 10th Annual Meeting of the Special Interest Group on Discourse and Dialogue*. SIGDIAL '09. London, United Kingdom: Association for Computational Linguistics, 2009, pp. 21–28. ISBN: 978-1-932432-64-0. URL: http://dl.acm.org/citation.cfm?id=1708376.1708379.

[AG03] David Abrahams and Ralf W. Grosse-Kunstleve. "Building hybrid systems with Boost.Python". In: 2003.

[AGM05] A. T. Alouani, J. E. Gray, and D. H. McCabe. "Theory of distributed estimation using multiple asynchronous sensors". In: *IEEE Transactions on Aerospace and Electronic Systems* 41.2 (Apr. 2005), pp. 717–722. DOI: 10.1109/TAES.2005.1468761.

[AR94] A. T. Alouani and T. R. Rice. "On asynchronous data fusion". In: *Proceedings of 26th Southeastern Symposium on System Theory*. Mar. 1994, pp. 143–146. DOI: 10.1109/SSST.1994.287895.

[Bas+16] Emanuele Bastianelli et al. "A Discriminative Approach to Grounded Spoken Language Understanding in Interactive Robotics". In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. IJCAI'16. New York, New York, USA: AAAI Press, 2016, pp. 2747–2753. ISBN: 978-1-57735-770-4. URL: http://dl.acm.org/citation.cfm?id=3060832.3061005.

[BCC17] Jacob Benesty, Israel Cohen, and Jingdong Chen. "Adaptive Beamforming". In: Nov. 2017, pp. 283–320. DOI: 10.1002/9781119293132.ch8.

[Bla+92] W. Dale Blair et al. "Least-squares approach to asynchronous data fusion". In: *Acquisition, Tracking, and Pointing VI*. Ed. by Michael K. Masten and Larry A. Stockum. Vol. 1697. International Society for Optics and Photonics. SPIE, 1992, pp. 130–141. DOI: 10.1117/12.138164. URL: https://doi.org/10.1117/12.138164.

[Bur+18] M.F.B. van der Burgh et al. *Tech United Team description paper*. Participation in RoboCup@Home, https://github.com/RoboCupAtHome/AtHomeCommunityWiki/wiki/Team-Description-Papers. https://github.com/RoboCupAtHome/

AtHomeCommunityWiki/wiki/files/tdp/2018-opl-techunited_eindhoven.pdf. 2018.

[Cao+18] Zhe Cao et al. "OpenPose: realtime multi-person 2D pose estimation using Part Affinity Fields". In: *arXiv preprint arXiv:1812.08008*. 2018.

[CD00] Ross Cutler and L. Davis. "Look who's talking: speaker detection using video and audio correlation". In: Feb. 2000, 1589–1592 vol.3. ISBN: 0-7803-6536-4. DOI: 10.1109/ICME.2000.871073.

[CH19] Jeffrey Cousineau and et al. Huynh-Anh Le. *Walking Machine Team description paper*. Participation in RoboCup@Home, https://github.com/RoboCupAtHome/AtHomeCommunityWiki/wiki/Team-Description-Papers. https://github.com/RoboCupAtHome/AtHomeCommunityWiki/wiki/files/tdp/2019-opl-walkingmachine.pdf. 2019.

[Cho+02] T. Choudhury et al. "Boosting and structure learning in dynamic Bayesian networks for audio-visual speaker detection". In: *Object recognition supported by user interaction for service robots*. Vol. 3. Aug. 2002, 789–794 vol.3. DOI: 10.1109/ICPR.2002.1048137.

[CLH19] Joon Son Chung, Bong-Jin Lee, and Icksang Han. "Who said that?: Audio-visual speaker diarisation of real-world meetings". In: (June 2019).

[DHK13] L. Deng, G. Hinton, and B. Kingsbury. "New types of deep neural network learning for speech recognition and related applications: an overview". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. May 2013, pp. 8599–8603. DOI: 10.1109/ICASSP.2013.6639344.

[FP70] Richard Fay and Arthur Popper. "Introduction to Sound Source Localization". In: vol. 25. Jan. 1970, pp. 1–5. DOI: 10.1007/0-387-28863-5_1.

[Fre13] Siepmann Frederic. "Behavior coordination for reusable system design in interactive robotics". dissertation. Bielefeld University, 2013.

[Gal98] M.J.F. Gales. "Maximum likelihood linear transformations for HMM-based speech recognition". In: *Computer Speech & Language* 12.2 (1998), pp. 75–98. ISSN: 0885-2308. DOI: https://doi.org/10.1006/csla.1998.0043. URL: http://www.sciencedirect.com/science/article/pii/S0885230898900432.

[Ger03] A. B. Gershman. "Robust adaptive beamforming: an overview of recent trends and advances in the field". In: *4th International Conference on Antenna Theory and Techniques (Cat. No.03EX699)*. Vol. 1. Sept. 2003, 30–35 vol.1. DOI: 10.1109/ICATT.2003.1239145.

[GM19] François Grondin and François Michaud. "Lightweight and optimized sound source localization and tracking methods for open and closed microphone array configurations". In: *Robotics and Autonomous Systems* 113 (2019), pp. 63–80. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2019.01.002. URL: http://www.sciencedirect.com/science/article/pii/S0921889017309399.

[Goo13] Google. *Google Cloud Speech-to-Text*. Software Product. https://cloud.google.com/speech-to-text/. Nov. 2013.

[GPR00] A. Garg, V. Pavlovic, and J. M. Rehg. "Audio-visual speaker detection using dynamic Bayesian networks". In: *Proceedings Fourth IEEE International Conference on Automatic Face and Gesture Recognition (Cat. No. PR00580)*. Mar. 2000, pp. 384–390. DOI: 10.1109/AFGR.2000.840663.

[Han+14] Awni Y. Hannun et al. "Deep Speech: Scaling up end-to-end speech recognition". In: *CoRR* abs/1412.5567 (2014). arXiv: 1412.5567. URL: http://arxiv.org/abs/1412.5567.

[har19] harry-7. *Speech Emotion Recognition*. Open Source Project. https://github.com/harry-7/speech-emotion-recognition. 2019.

[Iva+17] Serena Ivaldi et al. "Towards Engagement Models that Consider Individual Factors in HRI: On the Relation of Extroversion and Negative Attitude Towards Robots to Gaze and Speech During a Human–Robot Assembly Task". In: *International Journal of Social Robotics* 9.1 (Jan. 2017), pp. 63–86. ISSN: 1875-4805. DOI: 10.1007/s12369-016-0357-8. URL: https://doi.org/10.1007/s12369-016-0357-8.

[KKS89] K. Kita, T. Kawabata, and H. Saito. "HMM continuous speech recognition using predictive LR parsing". In: *International Conference on Acoustics, Speech, and Signal Processing,* May 1989, 703–706 vol.2. DOI: 10.1109/ICASSP.1989.266524.

[Láz+18] M.T. Lázaro et al. *SPQReL Team description paper*. Participation in RoboCup@Home, https://github.com/RoboCupAtHome/AtHomeCommunityWiki/wiki/Team-Description-Papers. https://github.com/RoboCupAtHome/AtHomeCommunityWiki/wiki/files/tdp/2018-sspl-spqrel.pdf. 2018.

[Lei18] David Leins. *Hyperion*. Open Source Project. https://github.com/hyperion-start/hyperion-core. 2018.

[Lie+16] F. Lier et al. "Towards automated system and experiment reproduction in robotics". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Oct. 2016, pp. 3298–3305. DOI: 10.1109/IROS.2016.7759508.

[Mar+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[Mat+15] Mauricio Matamoros et al. *RoboCup@Home Command Generator*. Open Source Project. https://github.com/kyordhel/GPSRCmdGen. 2015.

[Mat+18] Mauricio Matamoros et al. *RoboCup@Home 2018: Rules and Regulations*. http://www.robocupathome.org/rules/2018_rulebook.pdf. 2018.

[MBE10] Lindasalwa Muda, Mumtaj Begam, and I. Elamvazuthi. "Voice Recognition Algorithms using Mel Frequency Cepstral Coefficient (MFCC) and Dynamic Time Warping (DTW) Techniques". In: *CoRR* abs/1003.4083 (2010). arXiv: 1003.4083. URL: http://arxiv.org/abs/1003.4083.

[MHB14] O. Mubin, J. Henderson, and C. Bartneck. "You just do not understand me! Speech Recognition in Human Robot Interaction". In: *The 23rd IEEE International Symposium on Robot and Human Interactive Communication*. Aug. 2014, pp. 637–642. DOI: 10.1109/ROMAN.2014.6926324.

[Mic18] Microsoft. *Microsoft Speech to Text*. Software Product. https://azure.microsoft.com/en-us/services/cognitive-services/speech-to-text/. Sept. 2018.

[NOM17] Kazuhiro Nakadai, Hiroshi G. Okuno, and Takeshi Mizumoto. "Development, Deployment and Applications of Robot Audition Open Source Software HARK". In: *Journal of Robotics and Mechatronics* 29.1 (2017), pp. 16–25. DOI: 10.20965/jrm.2017.p0016.

[Ots+11] Takuma Otsuka et al. "Bayesian Extension of MUSIC for Sound Source Localization and Tracking." In: Jan. 2011, pp. 3109–3112.

[Pan+17] H. Pan et al. "FRIDA: FRI-based DOA estimation for arbitrary array layouts". In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Mar. 2017, pp. 3186–3190. DOI: 10.1109/ICASSP.2017.7952744.

[Per+19] Bruno F. V. Perez et al. *RoboFei Team description paper*. Participation in RoboCup@Home, https://github.com/RoboCupAtHome/AtHomeCommunityWiki/wiki/Team-Description-Papers. https://github.com/RoboCupAtHome/AtHomeCommunityWiki/wiki/files/tdp/2019-opl-robofeiathome.pdf. 2019.

[Por+16] Soujanya Poria et al. "Fusing audio, visual and textual clues for sentiment analysis from multimodal content". In: *Neurocomputing* 174 (2016), pp. 50–59. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2015.01.095. URL: http://www.sciencedirect.com/science/article/pii/S0925231215011297.

[Qui+09] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*. 2009.

[RF18] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *arXiv* (2018).

[RN96] Byron Reeves and Clifford Nass. "The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places". In: *Bibliovault OAI Repository, the University of Chicago Press* (Jan. 1996).

[rob07] robs@users.sourceforge.net. *SoX Resampler Library*. Open Source Project. https://sourceforge.net/projects/soxr/. 2007.

[Rod+13] Giampaolo Rodola et al. *psutil*. Open Source Project. https://psutil.readthedocs.io. 2013.

[SBD17] Robin Scheibler, Eric Bezzam, and Ivan Dokmanić. "Pyroomacoustics: A Python package for audio room simulations and array processing algorithms". In: (2017). DOI: 10.1109/ICASSP.2018.8461310. eprint: arXiv:1710.04196.

[Sor04] Laurent Soras. "The Quest For The Perfect Resampler". In: (Feb. 2004).

[Tan+19] Yuichiro Tanaka et al. *Hibikino Musashi Team description paper*. Participation in RoboCup@Home, https://github.com/RoboCupAtHome/AtHomeCommunityWiki/wiki/Team-Description-Papers. https://github.com/RoboCupAtHome/AtHomeCommunityWiki/wiki/files/tdp/2019-dspl-hibikino-musashiathome.pdf. 2019.

[Tan19] Jeffrey Too Chuan Tan. *Kamerider Team description paper*. Participation in RoboCup@Home, https://github.com/RoboCupAtHome/AtHomeCommunityWiki/wiki/Team-Description-Papers. https://github.com/RoboCupAtHome/AtHomeCommunityWiki/wiki/files/tdp/2019-opl-kamerider_opl.pdf. 2019.

[Tur14] Matthew Turk. "Multimodal interaction: A review". In: *Pattern Recognition Letters* 36 (2014), pp. 189–195. ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2013.07.003. URL: http://www.sciencedirect.com/science/article/pii/S0167865513002584.

[Wac+19] Sven Wachsmuth et al. *Team of Bielefeld Team description paper*. Participation in RoboCup@Home, https://github.com/RoboCupAtHome/AtHomeCommunityWiki/wiki/Team-Description-Papers. https://github.com/RoboCupAtHome/AtHomeCommunityWiki/wiki/files/tdp/2019-opl-kamerider_opl.pdf. 2019.

[Wal+01] Erik Walthinsen et al. *Gstreamer*. Open Source Project. https://gstreamer.freedesktop.org/. Jan. 2001.

[YLZ07] L. P. Yan, B. S. Liu, and D. H. Zhou. “Asynchronous multirate multisensor information fusion algorithm”. In: *IEEE Transactions on Aerospace and Electronic Systems* 43.3 (July 2007), pp. 1135–1146. DOI: 10.1109/TAES.2007.4383603.

STATEMENT OF AUTHORSHIP

I hereby certify that this thesis has been composed by me and is based on my own work unless stated otherwise. No other person's work has been used without due acknowledgment in this thesis. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged. This thesis has not been presented to an examination office in the same or a similar form yet.

Bielefeld, October 16, 2019

Robert Feldhans

REPRODUCIBILITY

SOURCE CODE

The source code for this thesis lies in a rather vast number of git repositories. All of these repositories are open source. Following is a list of all repositories used in this thesis along a short description:

Repository	Description
MasterThesis	Contains the latex code of this thesis
esiaf_orchestrator	Contains the source code of the Orchestrator described in chapter 4.2
esiaf_ros	Contains the source code of the library described in chapter 4.1
ma_eval_cost	Contains the results of the data set experiment, the scripts used to save them and scripts to automate analysis
esiaf_hyperion_configs	Contains configuration files for system startup for both experiments
esiaf_gender_rec	Gender recognition component
esiaf_eval_dummy_nodes	Test nodes used in the data set experiment
AudioSegmenter	VAD component
esiaf_wav_player	Component to feed wav files into the framework
rfeldhans-MA-master	Contains general information such as User stories and raw data of RoboCup@Home experiment as well as analysis script and processed information (`eval_raw_data` branch)
esiaf_doa	SSL component
ma_baseline_doa	SSL component for the pre-existing pipeline used in the RoboCup@Home experiment
esiaf_pocketsphinx	ASR component
esiaf_channel_utils	Components for channel handling, used in the RoboCup@Home experiment
esiaf_speech_emotion_recognition	Emotion recognition component
speech-emotion-recognition	Underlying library for the emotion and gender recognition components
Presentation	Contains the presentation for this masters thesis (under the `master_thesis` tag)

All these Repositories can be found under `https://github.com/Slothologist/MasterThesis` where MasterThesis can be any of the above repository names.

BUILD INSTRUCTIONS

Due to the number of components used in this thesis and the fact that each of them is embedded in their own git repository, building this source code can be a bit challenging. To alleviate this, I created a distribution in the CITK [Lie+16]. First, to bootstrap the CITK, follow their tutorial[1]. Afterwards you can install the distribution named ”speechrec-pipeline”, again following their tutorial[2].

Building is supported for Ubuntu 18.04 and 16.04, though under 16.04 an additional library has to be build from hand[3].

START INSTRUCTIONS

Similar to building such a great number of components, it is equally challenging to start them. I thus created configuration files for the Hyperion component launch engine [Lei18]. These and Hyperion itself are included in the CITK distribution. Several Hyperion configuration files are available, including those used for both experiments. After starting Hyperion itself, components can then be started at will through its GUI. I recommend starting the ROS core and the Orchestrator first, then the desired intermediate components and lastly the `Audio Grabber` or `Wav Player`.

1 `https://toolkit.cit-ec.uni-bielefeld.de/tutorials/bootstrapping`
2 `https://toolkit.cit-ec.uni-bielefeld.de/tutorials/installing`
3 This is documented within the README file of the relevant project under `https://github.com/Slothologist/esiaf_ros/`