The implementation of ImmACS - an Immersive Audio Communication System

Yannis Mastorakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes Campus, GR-70013 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Athanasios Mouchtaris*

University of Crete
Computer Science Department

The implementation of ImmACS - an Immersive Audio Communication System

Thesis submitted by
Yannis Mastorakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Yannis Mastorakis

Committee approvals: _____
Athanasios Mouchtaris
Associate Professor, Thesis Supervisor

_____
Panagiotis Tsakalides
Professor, Committee Member

_____
Xenofontas Dimitropoulos
Assistant Professor, Committee Member

Departmental approval: _____
Antonis Argiros
Professor, Director of Graduate Studies

Heraklion, November 2015

Abstract

In last decade there has been a lot of research on immersive audio capturing and reproduction. However, a complete communication system that utilizes immersive audio in real-time does not exist. In this work, we realize ImmACS, a complete communication system that allows the capturing and reproduction of high-quality immersive audio in real-time. ImmACS is based in a computationally efficient yet robust technique, that utilizes a circular microphone array for audio capturing, and headphones or loudspeakers for audio reproduction. Circular arrays overcome the inherent ambiguities of linear arrays and provide estimations in the full $[0°, 360°)$ range in the 2D space. The communication model supports multiple, concurrently active clients, that can simultaneously transmit and receive immersive audio. The model requires the mediation of a server for the distribution of the streams. The immersive audio stream consists of one audio channel accompanied by spatial meta-data, which we explicitly include in every packet as side information. Optional video streaming is also supported.

We implement Immacssip, the client of ImmACS, by modifying Baresip, an open-source VoIP client. We realize all the immersive audio functions in Libimmacs, a library we designed for optimal real-time performance and flexibility. We incorporate Libimmacs into Baresip and extend its configuration interface accordingly. Furthermore, we modify Baresip so that it is able to receive multiple streams per call, so as to allow the adjustment of the stream of each peer separately. Lastly, we utilize multiple threads for the decoding of the streams, so as to increase the throughput of the system in multi-core systems.

Bareserver is the server of ImmACS and is also based on Baresip. Each client connects to Bareserver directly, exploiting the fact that Baresip is by default a VoIP client. However, we have disabled the transmission of the local streams at Bareserver; it only relays the packets between clients connected to the same SIP account. The packets are relayed completely untouched, with no transcoding or down mix taking place, making the server's presence transparent to the overall communication. In addition, we utilize several worker threads that perform the relaying of the packets, so as to take advantage of the multi-core systems.

To facilitate the manipulation of the audio streams at the client, we provide Immacs Control, a flexible graphical interface. Immacs Control is an autonomous application that uses network sockets to communicate with the client. It allows the monitoring and adjustment, in real-time, of the direction and volume of the audio sources. All signal processing is performed at the client, while the interface only sends meta-data describing the audio filters.

Finally, to accommodate for system testing, we provide the ability to capture and reproduce audio from and to files at disk in real-time. In this way, a fully controlled input and output environment can be set easily, without the need for external hardware. In the same basis, we provide a special "echo" mode of function, where the server transmits the packets back to their senders.

Περίληψη

Τη τελευταία δεκαετία έχει γίνει αρκετή έρευνα σχετικά με την καταγραφή και την αναπαραγωγή εικονικού ήχου. Ωστόσο, δεν υπάρχει ένα πλήρες τηλεπικοινωνιακό σύστημα που να υποστηρίζει τη λειτουργία εικονικού ήχου σε πραγματικό χρόνο. Στην εργασία αυτή, υπολοποιούμε το ImmACS, ένα ολοκληρωμένο τηλεπικοινωνιακό σύστημα το οποίο επιτρέπει τη καταγραφή και την αναπαραγωγή εικονικού ήχου υψηλής ευκρίνειας σε πραγματικό χρόνο. Το σύστημά μας βασίζεται σε μια εύρωστη τεχνική με χαμηλό υπολογιστικό κόστος, που χρησιμοποιεί μία κυκλική συστοιχία μικροφώνων για την καταγραφή του ήχου, και ακουστικά ή ηχεία για την αναπαραγωγή. Οι κυκλικές συστοιχίες μικροφώνων δεν έχουν τις εγγενείς ασάφιες των γραμμικών συστοιχιών και προσφέρουν εκτιμήσεις στο πλήρες κυκλικό εύρος $[0^\circ, 360^\circ)$ στο δισδιάστατο χώρο. Το μοντέλο επικοινωνίας υποστηρίζει πολλαπλούς, ταυτόχρονα ενεργούς χρήστες, που μπορούν να στέλνουν και να λαμβάνουν εικονικό ήχο. Για την διανομή των ροών πληροφορίας χρειάζεται η μεσολάβηση ενός διακομιστή. Η ροή εικονικού ήχου αποτελείται από ένα κανάλι, συνοδευόμενο από χωρικά δεδομένα, τα οποία εισάγουμε στο κάθε πακέτο ως επιπρόσθετη πληροφορία. Υποστηρίζεται επίσης προαιρετικά και η επικοινωνία μέσω βίντεο.

Υλοποιούμε το Immacssip, το λογισμικό του χρήστη, τροποποιώντας το Baresip, μια εφαρμογή ανοιχτού κώδικα για VoIP κλήσεις. Υλοποιούμε όλες τις λειτουργίες για τον εικονικό ήχο στη βιβλιοθήκη Libimmacs, την οποία σχεδιάσαμε για ευελιξία και μέγιστη απόδοση σε πραγματικό χρόνο. Ενσωματώνουμε την βιβλιοθήκη στο Baresip και τροποποιούμε την διεπαφή του ανάλογα. Επιπρόσθετα, αλλάζουμε το Baresip ώστε να μπορεί να λαμβάνει πολλαπλές ροές δεδομένων ανά κλήση, ώστε να είναι δυνατή η διαχείρηση της ροής του κάθε χρήστη ξεχωριστά. Τέλος, υλοποιούμε την αποκωδικοποίηση των ροών παράλληλα, για να αυξήσουμε την απόδοση στα συστήματα πολλαπλών επεξεργαστών.

Ο Bareserver είναι ο διακομιστής του ImmACS και βασίζεται επίσης στο Baresip. Ο κάθε χρήστης συνδέεται στο διακομιστή άμεσα, εκμεταλευόμενος την ήδη υπάρχουσα υποστήριξη του Baresip για αυτό. Ωστόσο, απενεργοποιήσαμε την αποστολή των τοπικών ροών στο διακομιστή, και τον τροποποίησαμε ώστε μόνο να μεταβιβάζει τα πακέτα μεταξύ των χρηστών που είναι συνδεδεμένοι στον ίδιο λογαριασμό. Τα πακέτα αναμεταδίδονται απείραχτα, χωρίς να υλοποιείται κάποια μετατροπή ή μίξη, καθιστώντας τη παρουσία του διακομιστή ανεπαίσθητη στην όλη επικοινωνία. Επιπρόσθετα, παραλληλοποιούμε την αναμετάδοση των πακέτων για να πετύχουμε μεγαλύτερη απόδοση στα συστήματα πολλαπλών επεξεργαστών.

Διευκολύνουμε την διαχείρηση των ροών ήχου, υλοποιώντας το ImmACS Control, μια ευέλικτη γραφική διεπαφή. Η διεπαφή είναι μια αυτόνομη εφαρμογή και επικοινωνεί με τη κύρια εφαρμογή μέσω δικτύου. Επιτρέπει την παρακολούθηση και την προσαρμογή, σε πραγματικό χρόνο, της κατεύθυνσης και της έντασης των ηχητικών πηγών. Όλη η επεξεργασία σήματος πραγματοποιείται στη εφαρμογή χρήστη, ενώ η διεπαφή προσφέρει μόνο μετα-δεδομένα που περιγράφουν τα ηχητικά φίλτρα.

Τέλος, για να απλοποιήσουμε τις δοκιμές του συστήματος, επιτρέπουμε την φόρτωση και την αναπαραγωγή ήχου από και προς αρχεία στο δίσκο σε πραγματικό χρόνο.

Έτσι μπορεί εύκολα να στηθεί ένα πλήρως ελεγχόμενο πειραματικό περιβάλλον, χωρίς να χρειάζεται κάποια εξωτερική συσκευή. Στη ίδια βάση, προσφέρουμε μια ειδική λειτουργία όπου ο διακομιστής αναμεταδίδει τα πακέτα πίσω στους αποστολείς τους.

Acknowledgements

First of all I would like to thank my supervisor, Professor Athanasios Mouchtaris for giving me the opportunity to work with him and for his great support.

I would also like to thank the members of my dissertation committee, Professors Panagiotis Tsakalides and Xenofontas Dimitropoulos for their valuable suggestions, questions and advises for this work.

I would like to acknowledge the Institute of Computer Science (FORTH-ICS) for providing financial support and all the necessary equipment during this work.

Moreover I would like to thank my colleagues at the Signal Processing Lab, Despoina, Nikos, Fillippos and Tasos, for their help and useful advises for this thesis and for the pleasant working environment.

I would also like to thank my close friends Alexandra and Francesca for their constant patience, encouragement and support during this important venture of my life.

Last but not least, I would like to thank my family for showing belief in me until the end.

To the memory of my beloved grandfather, Yannis
Στη μνήμη του πολυαγαπημένου μου παππού, Γιάννη

Contents

List of Figures

Chapter 1

Introduction

Humans are able to identify the direction of sounds in an acoustic environment. Spatial audio concerns the artificial reproduction of a sound field where all the directional information is preserved. Nowadays, spatial audio is utilized by many applications so as to enhance the overall user's experience. It is widely used at the movie, game, music and telecommunication industry. In some applications the spatial sound is created artificially, while in others there is the need to recreate a real acoustical environment, where multiple audio sources may simultaneously exist. For example, a live concert, a football match, a theatrical play, a teleconference call etc. There have been proposed several techniques for spatial audio capturing and reproduction [1, 2, 3, 4, 5], that aim at the best possible recreation of the true acoustical environment, so as to provide a realistic auditory sensation to the listener.

1.1 Motivation

A lot of research work has been made in spatial audio capturing and reproduction. However, according to our knowledge, a communication system that utilizes both capturing and reproduction of spatial audio in real-time does not exist. The goal of this thesis is to create a complete communication system, that will allow the capturing, streaming, reproduction and manipulation of the true acoustical environment of each user, in real-time, for use at a common personal computer.

1.2 Contribution

We implement a system for VoIP calls that allows each user to transmit and receive spatial audio. From now on we will refer to this system as ImmACS, which stands for Immersive Audio Communication System. Our system is based on a client-server communication model where all clients communicate through the server. It supports the communication of multiple, simultaneously active clients, i.e. teleconferencing. In contrast to the conventional VoIP teleconferencing systems, our

system allows each user to manipulate the spatial properties and the volume of each peer separately. We achieve this by an all-to-all communication model, where each client receives the streams of all the other clients. This permits the configuration of a different audio setup for each user, according to its own preferences. Our system also combines high quality audio with low latency streaming, features that make it ideal for teleconference and suitable for Network Musical Performance applications, where multiple musicians perform through the network.

To support the functions that concern spatial audio, we utilize the technique of Alexandridis et al. for immersive audio capturing, encoding and reproduction, which is suitable for real-time applications and outperforms other similar state-of-the-art methods [1]. This method utilizes a circular microphone array for capturing and loudspeakers or headphones for reproduction. Moreover, from its definition, it allows the separation of simultaneous active audio sources, based on their direction. This property will prove to be very useful for controlling the reproduction of each source separately. We implement the functionality of [1] in a shared library, which we have carefully designed for optimal real-time performance, modularity and portability. From now on, we will refer to this library as **Libimmacs**.

We do not implement the client and the server of our system from scratch. Instead, we base our implementation on Baresip, an minimalistic open-source VoIP client [6]. Baresip has a very modular architecture that allows for easy maintenance of the code and multiple extensions. It is portable to both Mac OS X and Linux, and supports a wide variety of audio and video codecs. To implement the client of our system, we incorporate the functions of Libimmacs to Baresip. From now on, we will refer to the ImmACS client as **Immacssip**.

To realize the server of our system, we modify Baresip accordingly. We considered faster and easier to convert Baresip from a VoIP client to the VoIP server of our system, than to modify a real VoIP server. We came to this decision by exploiting our knowledge on Baresip due to the modifications for the client, and the simplicity of the server's requirements that can be easily supported by Baresip. While this implementation approach is mostly ad-hoc, it proved to be the best possible for the purposes of this project. We will refer to the ImmACS server also as **Bareserver**.

Finally, we provide a flexible graphical interface to facilitate the manipulation of the audio streams. The interface works as an independent application and can be executed on the same or a different machine with Immacssip. It features two interactive graphical effects, to control the spatial audio of each stream. The Spatial Equalizer, that allows the adjustment of the volume of each source in relation its direction, and the Spatial Mapper that allows the modification of the direction of each source separately. We name this graphical user interface as **Immacs Control**. Figure 1.1 displays an overview of the ImmACS system, using an example where three simultaneously active clients communicate through Bareserver.

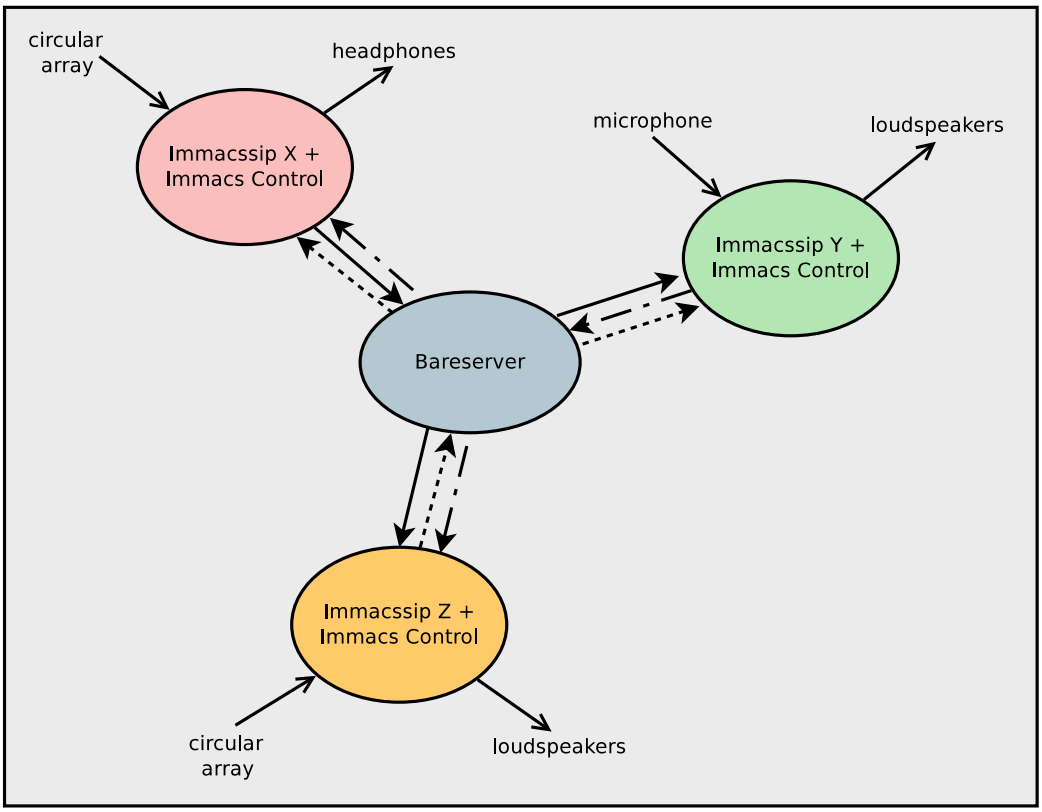

Figure 1.1: Overview of the ImmACS system. In this example, three clients communicate simultaneously through the server. The stream of each client is depicted using a different arrow, to facilitate the visualization of the route of the streams. Notice that the communication is possible among clients that utilize different input and output equipment.

1.3 Organization of this thesis

This thesis is organized as follows. In Chapter 2 we describe the basic theoretical framework for developing real-time audio applications. In Chapter 3 we analyze the structure of Libimmacs and explain our design choices. Chapter 4 concerns all the required modifications that converted Baresip to Immacssip. We provide all the aspects of the implementation of Immacs Control in Chapter 5. The details for the conversion of Baresip to Bareserver are given in Chapter 6. In Chapter 7 we provide all the necessary instructions for the operation of the system from a user's perspective. Finally, we conclude in Chapter 8, where we also express our plans for future work.

Chapter 2

Real-Time Audio Programming Background

In this chapter, we provide the fundamentals of real-time audio programming. We assume that the reader has some basic theoretical knowledge on digital signal processing and some experience on generic programming and the use of system calls. We discuss about some real-time unsafe programming habits and oppose some basic, more robust practices that can increase real-time performance. This chapter refers to general purpose operating systems such as Linux, Mac OS X, Mac iOS, Android, Windows, where the robust performance of real-time applications is not trivial. The suggested guidelines are quite generic and can be applied to any of these systems. Overall, the proposed techniques can be applied to any real-time application, not just audio.

2.1 The audio interface

An audio interface, also known as a sound card, is a device that allows a computer to receive and send audio data from and to the outside world. Its main function is to transcode a digital audio signal to analog or other digital forms. An audio interface that allows the simultaneous input and output of audio data is a **full-duplex** device. When both input and output can be performed but not on the same time, it is a **half-duplex** device. It is not important for an audio developer to know how audio interfaces work in detail. However, having a basic idea can really help to develop better audio applications. The following two examples provide an intuitive explanation of the input and output operation on a sound card.

Sound waves produced by a source are captured by a microphone and converted to continuous alternate current. This signal can be first enhanced by an amplifier or given directly as input to the audio interface. The analog signal is then sampled and quantized by using an **Analog-to-Digital Converter (ADC)**. Audio information is stored in small chunks of data in the hardware buffer of the interface. At regular time intervals the audio interface provides these data to the computer for further

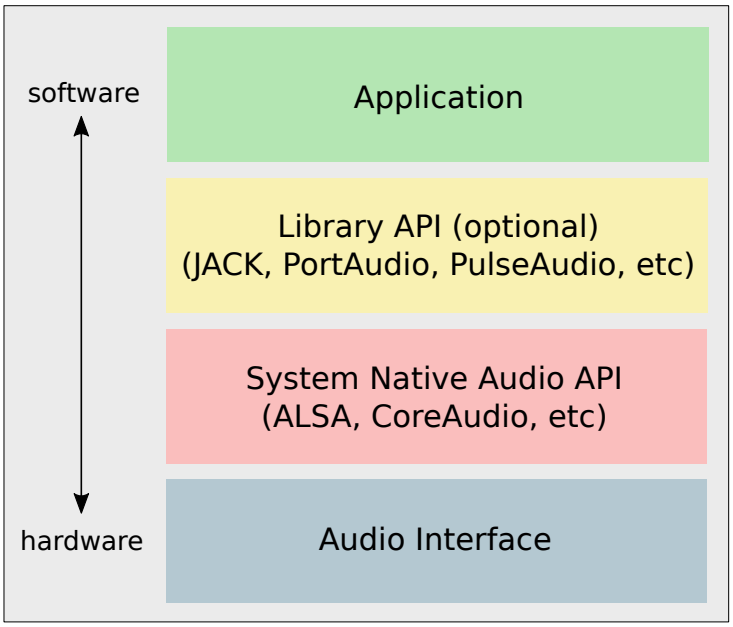

Figure 2.1: Audio data flow from hardware to software.

processing.

The reproduction works in a very similar way. The audio interface requests new audio data from the computer when it considers that it is necessary. The computer then transfers audio data into the hardware buffer of the sound card. The digital samples are converted to an analog signal by a **Digital-to-Analog Converter (DAC)** and exported by the output of the interface. Then, the analog signal can be given as input to an amplifier and converted to sound waves using a loudspeaker.

2.2 Audio APIs

The audio interface interrupts the system when audio data is required for input or output. After the interrupt the operating system takes over. Operating systems provide an abstraction layer for using the audio interfaces. Hardware-specific details are usually hidden and a common functionality is provided by a unified **audio Application Programming Interface (API)**. Each operating system provides its own API for audio. Linux audio API is provided by the Advanced Linux Sound Architecture (ALSA) software framework [7, 8]. Core Audio is the audio API for Mac iOS and OS X [9]. Microsoft Windows provides the Windows Audio Session API (WASAPI) [10].

System-native audio APIs provide an abstract yet flexible way for developing audio applications. However, the application will be bounded to the API's operating system and different implementations will be required for different operating systems. Furthermore, those APIs may be too complicated for very simple audio

applications or deficient and impractical for more complex ones. For these reasons, plenty of **audio libraries** exist, each one of them suitable for a different use. These libraries usually rely on a lower level audio API and provide a higher level interface. Some of them follow. PortAudio is an open source, cross-platform audio API [11]. Rt-Audio is a cross-platform C++ class for real-time audio input and output [12]. JACK Audio Connection Kit (recursively JACK) is a cross-platform sound server, that provides real-time, low latency audio communication between applications [13]. Similar to JACK, PulseAudio is a cross-platform audio server also capable of network streaming of audio [14]. An example of the audio data flow from hardware to software level can be found in Figure 2.1.

Most of the audio APIs provide DSP interface through a **callback** function, *e.g. JACK, CoreAudio, PortAudio etc.* The callback function is indicated to the API by the developer and must be compatible with the callback prototype the API provides. The callback function is called by the API every time audio data needs to be read (output) or written (input) to the application, similarly to the interrupt of the audio interface. Several audio APIs provide the option to read and write audio data at the same callback, simplifying the programming interface even more. The inverse procedure where the developer explicitly calls an API function so as to read or write audio data is the blocking method, *e.g. ALSA*. We refer to it as blocking because the caller usually blocks its execution by waiting the API to be ready for reading or writing data.

2.3 Circular buffering

In audio applications there is often the need to store audio data for future use. That is where audio buffers come in handy. An audio buffer is practically a First In First Out (FIFO) queue, where one writes, i.e. produces, and another reads, i.e. consumes the data. The situation where new data cannot be written to a buffer because there is not enough space left is called a buffer **overflow**. The opposite case where data need to be consumed but the buffer does not have enough data, is called a buffer **underflow**. A special kind of buffer is very often used for audio applications, the circular buffer. A circular buffer, also known as a **ring buffer**, is a data buffer that its end is connected to its beginning [15]. However, this connection is only intuitive, as the computer memory cannot be circular in practice.

For the manipulation of the ring buffer, two memory pointers are needed. One pointer indicates the head, i.e. start, of the buffer and the other the tail, i.e. end, of the buffer. At a write operation the new data are written where the tail pointer indicates and the pointer is modified accordingly. Similarly at a read operation the data are read from where the head pointer indicates and the pointer is appropriately changed. When the head/tail pointer reaches the end/start boundaries of the storage memory then it wraps around the start/end. In both operations, no transposition of any of the data is needed, just the modification of the corresponding pointer.

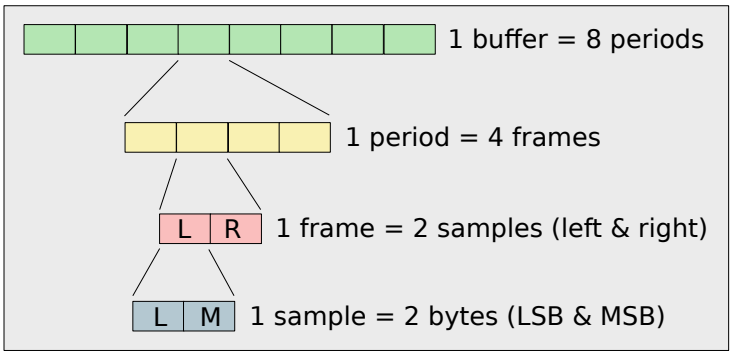

Figure 2.2: Subdivisions of an audio data structure example that consists of a buffer of eight periods, with a period of four frames, with a frame of two samples, with a sample of two bytes.

A ring buffer has a great advantage. Assuming that there are enough data for read and enough space for write, read and write operations use **independent** memory regions. That allows a lock-free asynchronous operation, with the restriction that only one writes and only one reads data. As it will be explained later, this property can make ring buffers a very useful tool for generic communication among the audio and other threads.

2.4 Audio data format and structure

Raw digital audio data is represented using the Linear Pulse-Code Modulation (LPCM) method, often referred simply as PCM. LPCM is an uncompressed format where the amplitude of an analog signal is sampled at uniform time intervals and each sample is quantized uniformly to its respective equivalent in a range of discrete values. The **sample rate** determines the frequency range of the digital signal while the number of discrete quantization values, also known as **bit-depth**, determines the quantization resolution. Audio interfaces support bit-depths of 8, 16, 20, 24, 32 bits per sample, while sample rates vary from 8000 to 192000 Hz.

A disadvantage of using integers for sample representation is that the values are dependent of the integer limits. A common practice to overcome this problem is to convert samples from integer to floating point numbers that range from -1 to 1. Floating point numbers can be single precision (32 bits) or double precision (64 bits).

In generic purpose computers, audio data are processed in groups of samples. A **sample** is the smallest audio data structure. On the other hand, the highest is the **buffer** of the application. A buffer consists of several fragments or **periods**, which are provided to the audio algorithm for processing. A graphical representation of the different audio structures is presented at Figure 2.2.

The samples of multiple channels can be stored in a period in two ways. When

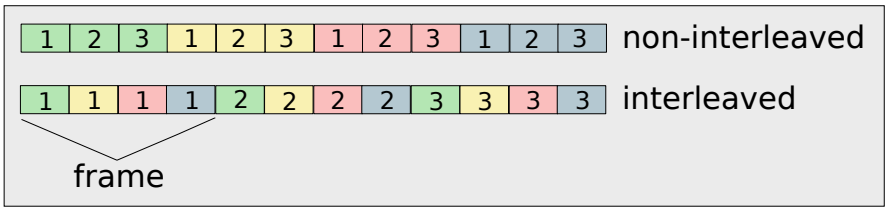

Figure 2.3: Sequence of audio samples in interleaved and non-interleaved format, for an example of four channels and three time-samples per channel. The colors discriminate the channels while the numbers indicate the time sequence.

the samples are grouped per time instance, it is called an **interleaved mode**. When the samples are grouped per channel, it is called a **non-interleaved mode**. Finally, a **frame** is a group of samples of the same time instance. A visual example of the two storing modes is provided in Figure 2.3.

Some APIs allow the user to set the size of the buffer and the period, *e.g. ALSA*, while others let him choose only the period and the buffer size is set by the API accordingly, *e.g. PortAudio*. The period is often identified as the application buffer, but it is not the same thing. A common choice for the period is half the buffer size, however both depend on the application.

2.5 Audio latency

The term **audio latency** refers to the time delay it takes for an audio signal to travel through a physical or artificial channel. It may be the delay of the sound from the computer loudspeaker to the users ears, the delay from the moment the piano player presses the key up to the time the sound reaches his ears or the delay it takes for a digital audio effect to capture, process, and output the audio signal.

The time delay from the entry of an audio signal to a DSP device or application up to its exit is called **roundtrip** latency. In systems created for live audio performance, the roundtrip latency must be low enough so as to give the impression that the signal is processed and derived instantly. Although a zero roundtrip latency is practically impossible, it is perceptually unperceived below 10 ms [16].

Audio latency also plays a significant role in Network Music Performance (NMP) systems, where several musicians perform simultaneously through the network. The latency of concern in these applications is the time delay it takes for a musician's instrument to be captured, encoded, sent through the web, decoded, reproduce and finally reach another musician's ears in the other side of the network. Through real experiments, it has been found that this latency has a sweet spot at 11.5 ms [17]. Greater latency results in gradually decreasing tempo while lower results in progressive increment.

2.6 Effects of the period size in performance

In general, any hardware or software system that intervenes between the audio input and output can introduce its own latency to the system. The most significant part of the latency, however, comes from the size of the period. The period size determines the **minimum latency** of the system, which is the time it takes to collect and deliver all the samples of one period.

From another point of view, the size of the period also determines the **maximum allowed latency** of the processing algorithm, which is the time deadline for processing one audio period. Beyond that time limit, buffer overflows or underflows may occur. Such buffer inconsistencies will produce audio glitches, which can make the system unusable.

Reducing the period size would reduce the audio latency but it would increase the interrupt rate at the CPU. More frequent interrupts will increase the CPU load in a non-linear way, and can lead to unstable and unpredictable system behavior. This is because the processing overhead of the interrupt will be induced at every period. Furthermore, a smaller period size directly imposes a shorter available time for real-time processing, which may cause buffer underflows. A general conclusion is that **the period size sets a trade-off between system latency and robustness**.

2.7 Challenges of real-time processing

Sound is a continuous physical phenomenon, thus the capturing and reproduction of sound in real-time requires a constant flow of data to and from the audio interface. With no exceptions, audio data must be available whenever they are needed, or else a buffer overflow or underflow will occur. The result of a buffer overflow/underflow is that invalid data will be transferred from or to the interface, resulting in audible noise, clicks, cracks or gaps at the sound, also known as **glitches**. Glitches are highly undesirable because of their unpredictable and uncontrollable nature, that can cause serious damage in other parts of the audio system. Moreover, glitches can significantly degrade the overall acoustical sensation of the listener. To reduce the probabilities of glitches to happen, the real-time algorithm should perform optimally.

Real-time performance does not only concern the ability of a system to meet the real-time constraints, but also its ability to meet them **reliably**. In other words, the performance is not only judged by the worst-time case of execution, but also from the processing time fluctuations. For example, an algorithm that requires a **constant** 70% of the real-time is more preferable than another that requires the 30% but occasionally **spikes** up to 90%. To realize why, think of what will happen if the real-time processing is burdened with another 20% additional load caused by another application. The total system load for the first algorithm will be a high, yet constant 90%, while the second algorithm will cause a 50% load with occasional

overload to 110% that will cause audio glitches. This is a rather realistic scenario, since on a general purpose system no assumption can be made about the overall load of the CPU at the time of execution.

In general, it is very difficult to predict and prevent all the possible cases of real-time failure, i.e. inability of the system to process data in time. The used algorithms, the audio API, the operating system, the audio drivers, the audio interface, the computer hardware and the overall CPU load at the time of execution, can affect the overall processing latency and cause the system to fail. The root of the problem, however, usually lies in poor real-time performance at user-application level. Nevertheless, even if the algorithms are optimal for real-time operation, there will always be unpredictable factors that will push the system to its limits. **The challenge is to make the system as less as vulnerable to every occasion.**

2.8 Good and bad practices for real-time coding

There are several programming techniques that are commonly used in generic programming and provide simple and elegant solutions to many difficult situations. However, not all of them are safe for real-time operation. Ross Bencina aptly describes some of them in his article [18], that we find useful to briefly report here for the purposes of this project. Some equivalent but real-time compatible techniques are also provided.

Memory locks

Memory locks are widely used to allow two or more threads to safely access a common part of the memory. Real-time applications usually involve more than one threads so memory locks could prove to be very useful. However, locks are not at all safe for real-time execution. In its simple form, a lock can be acquired from only one thread, allowing the thread to gain access to the shared part of the memory, while the rest wait for the lock to be released. If the real-time thread blocks by waiting for another thread to do something, it is practically bounded to the execution time of the other thread. The latter may not be optimally written for real-time operation and may also have a lower priority of execution that the first, so it could take an unpredictable amount of time to finish. The case where a high-priority thread blocks by waiting a low-priority thread is called **priority inversion**. Additionally, in the case where multiple threads compete for the same lock, it is unknown if and when the real-time thread will get it. Moreover, blocking for a lock can cause the OS scheduler to interfere to the execution so as to reschedule the blocking thread. In generic operating systems, schedulers implement complex algorithms that are oriented more towards throughput than real-time accuracy, and can behave quite unexpectedly when the system gets stressed.

Alternative to locks are the try-lock operations, where acquiring a taken lock does not block the interested thread. If the lock is taken, the handling is left up

to the application. However, this adds extra complexity to the real-time code and there is still no guarantee that the lock will ever be available.

A simple, yet good practice, is to use ring buffers as generic purpose FIFO queues for thread communication. For one-to-one bidirectional thread communication, two FIFO queues are necessary. The first is written by thread A and read by thread B while the second is written by thread B and written by thread A. By using ring buffers, non-blocking asynchronous communication is feasible between the two threads. A disadvantage of this method is that buffering may induce delays in the communication.

Memory management

Dynamic memory allocation/de-allocation is a straight forward procedure for generic purpose applications, however it requires special care when operating at real-time. Memory allocation algorithms may require large amounts of time to allocate a block, or they may themselves block by waiting IO from the operating system. Moreover, the algorithms can vary between operating systems and different versions. If a memory allocation algorithm does not guarantee real-time safety, it should not be used. A simple alternative solution is to pre-allocate all memory. Either pre-allocate space for each variable separately or allocate a big block of memory and use a custom memory allocation algorithm that is safe for real-time operation. If pre-allocation is not possible, then dynamic allocation should happen in a separate thread that is not real-time critical.

Another issue worthy of discussion is **garbage collection**. Garbage collection is a mechanism that simplifies memory management by automatically de-allocating memory when it is no longer needed. It can block the execution flow of the program so it must be used with caution in real-time systems, ensuring that it will not lag the operation.

Lastly, **page faults** can cause delays in the processing that can result to glitches. When the operating system decides that it is necessary, e.g. if the memory is full, it can move rarely used memory blocks to the disk. Accessing those blocks can cause large delays, by waiting data to be read from the disk. This can be prevented by ensuring that memory is accessed often or by using special mechanisms provided by the operating systems, such as mlock()/munlock() for Mac OS X and Linux and VirtualLock()/VirtualUnlock() for Windows.

Algorithmic complexity

Algorithms predestined to run on real-time systems should perform optimally for the worst-time case, rather than the average-time case preferred for generic systems. Moreover, it is highly recommended that there are not any high variations in the execution delays, which can cause occasional latency spikes. As previously explained in section 2.7, these spikes may seem harmless when the system load is low but they can be the source of glitches if the system gets stressed. For this

reason, it is more preferable to use a slower algorithm which, nevertheless, requires a constant time delay, than a fast algorithm which may perform very slow occasionally. If the algorithm complexity is too high to operate in real-time, it is better to spread the computations across many periods to reduce the CPU load.

Summary

Overall, the basic precaution to avoid glitches is to avoid anything that could lead to unbounded and unpredictable execution time. This includes operations such as locking, memory allocation, disk input/output, and every system call in general that could block the application. Algorithms with poor worst-case time of execution can be the source of audio glitches too. Besides that, any external function call should be avoided if it does not, directly or indirectly, observe these rules.

Instead, wait-free FIFO queues are a simple and safe solution for whenever data transfer is needed to/from the audio thread, while memory should be pre-allocated when possible. Algorithms with good worst-case time complexity should be preferred and computational distribution across many periods is encouraged to smoothen the CPU load. Special care must be taken to prevent page-faults to happen for real-time used memory. Specific guidelines and techniques should be followed when advanced audio APIs are utilized for the development, such as like CoreAudio, ALSA, Jack etc. In this project, we apply all these techniques to libimmacs, to ensure that the provided functions will be strictly safe for real-time use.

Chapter 3

Libimmacs - ImmACS real-time processing library

Libimmacs is a shared library, that provides all the ImmACS functions for real-time immersive audio capturing, encoding, decoding, reproduction and more. There are plenty of reasons that led us to enclose all ImmACS audio operations in a library. A library implementation allows for code-autonomy, which implies easier maintenance and re-usability. In this thesis, Libimmacs is used in conjunction with Baresip, nevertheless it is a completely independent component that can be used in any future ImmACS-based project. For complete instructions on how to compile and install Libimmacs refer to section 7.2.

3.1 Features

Libimmacs provides complete functionality for

- Efficient DOA Estimation.
- Spatial encoding and decoding of multichannel audio.
- Compression/decompression and easy manipulation of the spatial audio metadata.
- Spatial audio reproduction for headphones and loudspeakers, with flexible volume control.
- Safe streaming of multichannel audio data to and from the disk.
- Safe asynchronous logging.

Libimmacs also provides a set of tools for

- Easy memory management.
- Audio data conversions.

- Useful mathematical functions.

3.2 Design challenges and practices

All Libimmacs functions have been developed in the basis of a demanding but necessary design framework. The main goals of the design follow.

Optimal real-time performance

Libimmacs is not designed especially for use with Baresip. The assumption of re-usability directed us towards a more challenging design, that requires Libimmacs to be versatile and robust for every kind of application. The primary design goal is optimal real-time performance, that will make the library suitable even for demanding real-time applications. For this reason, every operation that is expected to run in real-time is designed to comply with the practices described in Section 2.8. Furthermore, we chose to use the C programming language, so as to achieve better performance. In this point, it is worthy to mention that Baresip, in which we will incorporate Libimmacs in this project, is not designed for optimal real-time performance.

Flexibility

The second significant challenge of the design is flexibility. Even though Libimmacs is a realization of a specific system, which is described by Alexandridis et al. at [1], it has been designed as several autonomous pieces, i.e. objects, that can be used individually and independently if needed. This complete modular structure also allows for quite easy extensions to the library. The flexibilities of Libimmacs will become more apparent in the upcoming sections, as we will analyze the code structure.

Portability

Another desirable feature is portability. All internal code written and all external libraries used are portable to the three major operating systems, Linux, Mac OS X and Windows. The **cmake** build manager is also used for portability on the mentioned systems [19]. Despite the fact that the library has been developed and tested in Linux and Mac OS X, compatibility with Windows is expected to be supported. However, no actual testing on Windows has been made, but it remains open for future work.

System testing

A very important issue is system testing. Due to the fact that a fully-controlled

experiment is difficult to be set in a real environment, a simple testing framework is provided for experimental purposes. Libimmacs provides the ability to read or write multiple or multichannel audio files in real-time from the hard disk. This feature allows to import/export the microphone inputs/loudspeaker outputs from/to files on disk, eliminating the need for external hardware equipment and permitting a fully-controlled testing environment. Although audio streaming from/to the disk is provided in the basis of a testing framework, it is a completely independent part that can be used whenever streaming of audio from the disk is needed.

Easy maintenance

Libimmacs is still in an experimental state that leaves room for many corrections, modifications or extensions. To facilitate for these cases, a conceptually simple structure and easily readable code is provided, to help future developers. Code is written to be self-explanatory and comments are given everywhere needed. Nevertheless, a complete documentation of the library is left for future work.

3.3 Code overview

Libimmacs is written in C but its design is object oriented. C++ should be more compatible with the design but we chose C to have more control of the execution and memory management. Objects are defined as structures, whose composition is conceptually hidden from the user. Objects can be created, destroyed, modified and accessed only from functions that are public through the interface of the library. Figure 3.1 depicts the architecture of the main objects of Libimmacs. The components will be explained in Section 3.4.

Object and function declarations are separated from their definitions. All declarations that are public to the user are located in the "*immacs.h*" header, while declarations that are used only internally in the library are placed in the "*immacs_core.h*" header. On the other hand, the function definitions are grouped as one source file per object. As a general rule, the name of the object is used as the name of the source file.

External open-source code

Some external open-source code has been used in Libimmacs. This code is placed at a separate folder to be logically separated from the rest of the project. The code we use is an efficient implementation of a ring buffer, found in **PortAudio** [11], that allows non-blocking, asynchronous read and write of data. We also used **TinyCThread**, which is a minimalistic implementation of a thread library API in C11 [20]. Essentially, it is a wrapper API to provide thread portability over various systems, such as Windows, Mac OS X and Linux. Lastly, we also used **pstdint**, which provides portability of the integer types for the systems we mentioned earlier

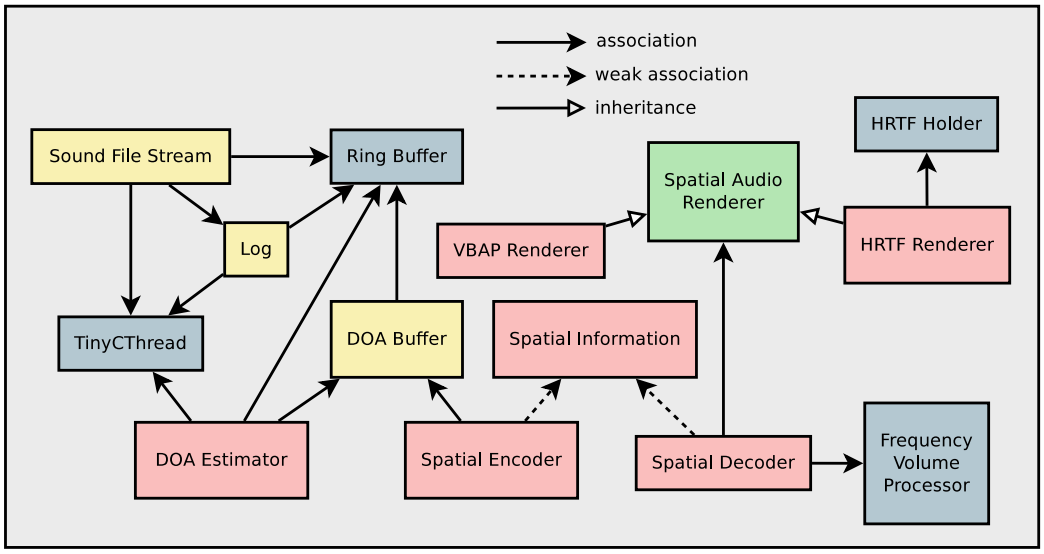

Figure 3.1: Libimmacs architecture. Warmer colors declare greater significancy of the objects for the library user. Cardinality has been omitted for clarity.

[21].

External dependencies

Libimmacs also depends on four external libraries. For fast, efficient and portable Discrete Fourier Transform computation, we utilize **FFTW** [22]. To allow reading/writing audio files from/to the hard disk, we use **libsndfile**, which is written in C and is portable to many systems [23]. We also use **ITPP**, a C++ library of mathematical, signal processing and communication classes and functions [24]. ITPP was an inevitable choice for Libimmacs. It uses advanced algorithms that perform extremely fast on some required operations, that would take an unacceptable amount of time written in a simpler form. ITTP is the only C++ dependency of Libimmacs, that makes it not a strictly C library. However, it is used only in a very small part and can be avoided if the same algorithms are provided manually. This remains open for a future work. For instructions on how to install the dependencies of Libimmacs refer to section 7.2.1.

3.4 Code analysis

Custom Types

Libimmacs defines three custom types that are public to the user and are used throughout the library. The "*immacs_sample_t*" refers to the format of the audio sample which is practically declared as a float number. The "*immacs_frames_n*"

type is used for counting audio frames and is practically a non-negative integer. Lastly, the "*immacs_complex_t*" defines a complex number type and it is an array of two float elements. The first element represents the real part and the second the imaginary part. Changing of these types is possible but not is recommended.

DOA Estimator

Libimmacs provides efficient real-time DOA estimation, using the technique of Pavlidi et al. for circular microphone arrays [25]. This functionality is implemented in the "*immacs_doa.c*" file, and it is provided by the public interface of Libimmacs. It allows the arbitrary selection of the microphone number and the radius of the array, the maximum number of sources and the source thresholds from the library user. While the sample rate is also defined by the user, the only allowed values are 44100 and 48000 Hz.

The processing of audio data uses a constant block size of 2048 samples, with 50% overlap [25]. However, DOA estimation is expected to be used with audio periods of an arbitrary size. For this reason, audio data are first collected in a memory buffer before processing. When the buffer has enough data, processing occurs.

Collecting the right number of data across many audio callbacks can lead to periodic processing bursts of the CPU, decreasing the real-time reliability of the system. For this reason, DOA estimation is performed in a separate thread to equally distribute the processing load across many callbacks. A ring buffer is used to allow non-blocking, asynchronous read and write of the audio data between the audio thread and the DOA estimation thread.

The audio buffer stores up to 16 audio periods. If there is no space in the buffer to store new data (buffer overflow), the data are dropped, without blocking the provider, i.e. the audio thread. This is a rather rare case except if the system's processing power is inadequate for DOA estimation. From the side of the data consumer, i.e. the estimation thread, buffer underflows are very often expected when the period size of the audio callback is less than the period size of the DOA estimator, which is 1024 samples. In that case, the estimation thread waits for the amount of time needed, so as to complete one period. It then re-checks and repeats the same procedure until the required number of audio frames is available. Buffer overflow and underflow statistics are also provided for system monitoring.

DOA Buffer

Estimated DOAs must be provided back to other interested objects in real-time. This is feasible by using DOA Buffers. A DOA Buffer allows the safe asynchronous retrieval of the DOA estimates from the DOA Estimator. It is necessary because the estimates are calculated by a separate thread in the DOA Estimator. The code of the DOA Buffer lies in the "*immacs_doa_buffer.c*" file and is part of the public interface of Libimmacs.

Practically, a DOA Buffer is a FIFO queue that is registered to a DOA Estimator for update. Multiple DOA Buffers can be registered in the DOA Estimator. Each time a new estimation is available, it is written in all the registered buffers. Each DOA Buffer is used by exactly one consumer. The consumers can read the estimates from the buffers whenever they are needed.

The DOA Buffer internally utilizes a ring buffer to achieve non-blocking, asynchronous read and write operations. It stores estimations up to 64 audio periods, which is about 1.3 seconds in 48000 sample rate and it is found adequate for the purposes of this project. If the DOA Estimator is unable to write new data to the DOA buffer because it is full (overflow case), the data are dropped. The DOA Buffer also provides statistics about the number of buffer underflows and overflows.

Spatial Encoder

The Spatial Encoder encodes multichannel audio data captured with a circular microphone array, to monophonic audio data with corresponding spatial metadata, i.e. side information. This is achieved by using the beamforming technique for source separation and downmix described by Alexandridis et al. at [1]. The interface of the Spatial Encoder is part of the public interface of Libimmacs, while the code is found in the file "*immacs_spatial_encoder.c*".

Beamforming requires the DOA estimates for the processing of each period of audio data. The Spatial Encoder instantiates a DOA Buffer that can be registered to the DOA Estimator to retrieve the DOAs as described in the previous section. Notice that the design does not restrict the Spatial Encoder to work only with the provided DOA Estimator; it can be easily integrated to work with other estimators as long as they provide the data through the DOA Buffer.

When the DOA buffer is empty (underflow case), the previous DOA estimates are used, if they exist, or else silence is encoded. Buffer underflows are very likely to happen when the period of the audio callback is less than the period of DOA Estimation. This is also the main case for the integration of Libimmacs to Baresip, in which the audio period is smaller than 1024 frames in 48000 sample rate operation.

The Spatial Encoder works by processing one audio period at a time. It can work with an arbitrary audio period size, which it is provided by the user at the object's initialization and is considered to be constant among all callbacks. Internally it processes a block size twice the size of the audio period using a 50% overlap [1]. It also allows the library user to set the desirable number of frequency coefficients used in the encoding. It is obvious that this number should be at least twice the size of the audio period.

The variable size of the audio period requires that the weights of the beamformer are dynamically calculated at the object's initialization. Due to the fact that weight calculation implies complex and heavy operations such as matrix inversions, we use the **ITPP** library to achieve optimal speed [24].

The process interface of the Spatial Encoder takes multiple audio periods of

the same time instance as input, which correspond to the multiple microphones of the array. It produces a single-channel audio output, which is the down mixed signal derived from the beamforming source separation, and the spatial meta-data according to the current sources.

The spatial information structure is not part of the Spatial Encoder, but it is an independent object provided by the user. This allows for more flexibility on the manipulation of the spatial information. No compression of the spatial information is made, just the initialization of the structure.

Finally, it is worthy to mention that in contrast to the DOA Estimator, all process is executed in sync, meaning that the process function will block the calling thread, i.e. the audio thread.

Spatial Audio Renderer

The Spatial Audio Renderer is the base class of all audio renderer objects. A spatial renderer defines how the frequency components of one monophonic audio period will be rendered, to a specific output setup, to produce immersive audio according to their corresponding spatial meta-data. It works only in conjunction with the Spatial Decoder, nevertheless it is a completely independent component. We will refer in detail to the Spatial Decoder in the next section. The separation of the rendering part from the decoding process allows the easy expansion of the system to support multiple renderers. The two renderers provided by Libimmacs are the HRTF Renderer and the VBAP Renderer, for headphone and loudspeaker reproduction respectively.

The Spatial Audio Renderer is a virtual object, meaning that it cannot be used directly on its own, but only through a child class. It provides the common function and data interface for all renderers. Of course, C does not support classes neither inheritance, but the functionality for this case can be easily implemented. The Spatial Audio Renderer is a structure, holding the renderer common variables and a pointer to the rendering function. A child object must hold and initialize an instance of the parent class in its structure, according to its needs. The Spatial Decoder calls the rendering function indirectly through the function pointer and uses the variables of the parent structure, without knowing anything about the actual renderer. The rendering function has a standard interface which is declared at the public immacs interface.

Spatial Decoder

The Spatial Decoder converts a spatial-encoded monophonic audio input to an immersive audio output, using its corresponding spatial meta-data, as described by Alexandridis et al. at [1]. It works in conjunction with a Spatial Audio Renderer object, as described in the previous section. The manipulation of the volume of each source, according to its DOA, is also supported. The code of the Spatial Decoder lies in the file "*immacs_spatial_decoder.c*", while its declared as a part of

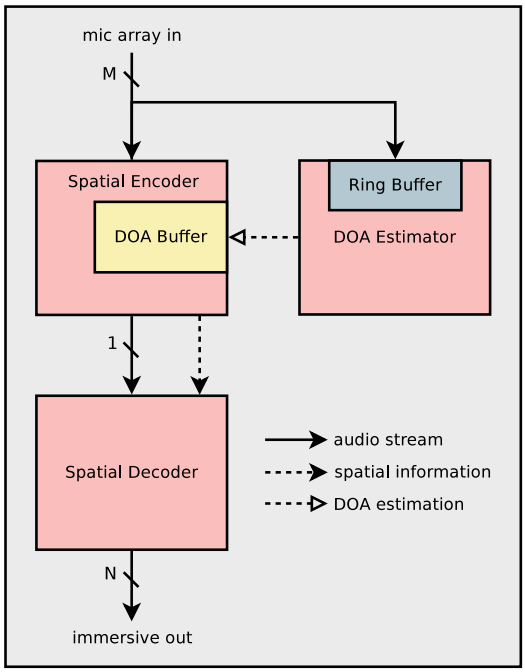

Figure 3.2: Example of combining parts of Libimmacs for the capturing and reproduction of immersive audio. The diagram emphasizes on the data flow between the components. Notice that DOA estimation is performed in parallel with the encoding/decoding process, justifying the use of buffers for data communication.

the public Libimmacs interface. Figure 3.2 depicts a simple example of utilizing the three major components of Libimmacs (DOA Estimator, Spatial Encoder, Spatial Decoder) to capture and reproduce immersive audio.

The Spatial Decoder works in a very similar way to the Spatial Encoder. It processes one audio period at a time, whose size is constant and is set by the user at the object's creation. The synthesis uses a block size twice the audio period with 50% overlap [1]. The audio period is transformed in the frequency domain and then the volume processing takes place. After that, the data are given to the spatial renderer for processing. The spatial renderer returns the immersive audio also in the frequency domain, which is further processed to provide the signal in time. Respectively to the Spatial Encoder, the spatial information is provided in its de-comporessed form. Moreover, the process function blocks the execution of the calling thread until finished.

VBAP Renderer

The VBAP renderer is a child object of the Spatial Audio Renderer that performs rendering of audio for loudspeakers, using the Vector Based Amplitude Panning (VBAP) method [5]. It is part of the public Libimmacs interface and its code lies

in the file "*immacs_vbap_renderer.c*". It works for an arbitrary number of loudspeakers, provided by the user. Even though VBAP also supports an arbitrary geometry for the loudspeakers, it is assumed that they are placed uniformly at [0°, 360°), at equal distances from the listener. This geometry model, although simplified, meets the requirements of this project and simplifies the code. The VBAP coefficients are calculated dynamically at initialization, and stored in memory for instant access.

HRTF Renderer

The HRTF Renderer is a child object of Spatial Audio Renderer that performs rendering of audio for headphones using Head Related Transfer Functions (HRTF). It is part of the public Libimmacs interface and its code lies in the file "*immacs_hrtf_renderer.c*". It uses an instance of the HRTF Holder object to retrieve the filters in real-time. The HRTF Holder will be explained later in this chapter. This renderer uses an ad-hoc lower limit of 128 frequency coefficients, which implies straight from the fact that the HRTF filters used consist of 128 time samples.

Spatial Information

The Spatial Information object holds the spatial meta-data of a single audio period, produced by the Spatial Encoder and consumed by the Spatial Decoder. Its independent form allows the user to easily modify its data or completely create one from scratch. Furthermore, it provides functions for the compression and decompression of the spatial information according to the technique described by Alexandridis et al. at [1], which can be useful for network transmission or file storing. The code of this object is located at the "*immacs_spatial_info.c*" file and its interface is public to the library user.

The Spatial Information stores data that concern the frequency domain. The number of the frequency coefficients is given in its creation and is considered constant in every operation. The DOAs are stored only once in an array and an index for each frequency component is used that indicates the corresponding DOA. In this way, the modification of the DOA of a source requires the change of only one number. This also simplifies the compression and decompression algorithms, reducing the computational cost.

To allow for more convenience and flexibility, we define some arbitrary but sufficient conventions for the data, that should be taken into consideration by every spatial renderer. These conventions are

- Valid DOAs can be real numbers, greater-equal to 0°and smaller than 360°.
- Negative DOAs indicate "special" directions to the spatial renderer
  - -1 indicates that the corresponding frequency component should be played from every output.

- -2 indicates that there is no spatial information for this component and handling is left up to the respective renderer.

We choose to use negative DOA values as special indications, because in this way they can be seamlessly compressed all together with the valid DOAs by the compression algorithm. As a future work, we intend to add support for a special diffuseness indication of a frequency component, that will be supported by the two provided renderers.

Sound File Stream

The Sound File Stream object provides real-time streaming of audio from/to the hard disk. It has two modes of operation, read and write. At read mode, audio is read from files at disk and converted to an audio stream, while inversely at write mode an audio stream is converted and written to a file at the disk. File operations support concurrent read/write of a single multi-channel, or multiple single-channel, audio files. The format of the input/output stream can be interleaved or non-interleaved. The stream format is independent of the file format, with all the necessary conversions made automatically and concealed from the user, allowing for simple use and maximum flexibility. The source code of this object can be found in the file "*immacs_sndfile_stream.c*" and it is part of the public Libimmacs interface.

The Sound File Stream utilizes the **libsndfile** library to read and write audio files from/to the disk [23]. However, accessing the hard disk directly from the audio thread is a very dangerous practice for real-time, since large and unpredictable latencies can occur. These latencies may be induced by waiting the mechanical parts of the hard disk to move, or the driver and the operating system to collect the data. For this, we use an asynchronous mechanism where a separate thread is utilized to perform all write and read operations to and from the disk, where all data transfer is carried out through ring buffers. The intermediate audio buffers compensate for the latency variations due to the disk IO, allowing the non-blocking transfer of data between the audio thread and the disk thread.

The Sound File Stream can be set to either read or write mode at the time of its creation. The path to the files for reading/writing is given also as input, following a specific template. More details about the file name template can be found in the section 7.3.4. At both modes of operation, the disk transaction thread transfers data in blocks of 4096 frames, with a special handing case of the End Of File (EOF) for the read mode. The ring buffer size used is 131072 frames, which is about 2.7 seconds when using 48000 Hz sample rate. Statistics of the buffer overflows and underflows are also provided. Now we will extend to some specific details for each mode of operation.

At **read mode**, the whole buffer is filled at the time of the object's initialization, or the whole file is loaded if its size is less than the buffer size. This guarantees that audio data will be immediately available from the very first request for data.

If there is no space available for the disk thread to write new data (buffer overflow), the thread waits for 4096 samples time by blocking, and then retries. If there is no data available for the calling thread to read from the buffer (buffer underflow), an analogous indication is returned without blocking and handling is left up to the user.

Write mode supports a synchronization operation, which blocks until the write thread has finished writing all the buffer data to the disk. This function can be used to guarantee that all data will be successfully written to the files before the object's destruction. If there are no data to write to the disk (buffer underflow), the thread blocks for 4096 samples time and then tries again. The case where the buffer is full, preventing the calling thread to write to the buffer (overflow), is left for handling to the user.

Logging

Almost every application requires a logging system for recording or displaying messages relative to information or debugging. When it comes to audio applications, special care must be given due to the fact that logging may occur from inside the audio thread. Libimmacs provides a general purpose logging infrastructure, that is guaranteed to be real-time safe for use with an audio application. The log's source code lies in the file "*immacs_log.c*" and it is part of the public Libimmacs interface.

The log system consists of the log manager, which is practically a separate thread that processes the log messages, and the log objects, which are practically ring buffers of the messages to be printed. A log object is automatically registered to the log manager at the time of its creation. Messages can be provided to the log object for non-blocking processing, but only from one execution thread. The log manager polls all the registered logs every 200 milliseconds and processes all the pending messages. By default, the messages are printed on the screen, but the library user can define his own process function.

Log messages can be characterized by four labels: debug, info, warning and error. Each message can consist of at most 512 characters, and each log object can hold up to 64 messages. These sizes do not constitute a practical limitation for this project.

HRTF Holder

Spatial audio reproduction with headphones is feasible at Libimmacs with the use of Head Related Transfer Functions (HRTF). The HRTF Holder object, located in the "*hrtf_holder.c*" file, provides an abstraction level for loading, storing and retrieving HRTF data from the hard disk. Its interface is not public to the user, it is only used indirectly through the HRTF Renderer object.

At the time of its creation, the HRTF Holder reads all HRTF data from a single disk-file, whose format will be reported later in this section. Due to the fact

that the HRTF will be only used in the frequency domain, the discrete Fourier transform of all HRTF is calculated at the object's initialization and only the frequency components are finally stored. This optimizes the performance, because the frequency coefficients of the HRTF will be instantly available for use whenever needed.

The user can provide an arbitrary angle to retrieve the HRTF of, and the HRTF Holder provides the HRTF whose angle more closely matches the desirable angle. No search is required; it uses a simple indexing operation that comes up with O(1) complexity.

The HRTF data files, which are used for the holder's initialization, are provided by default by the library. We use the MIT "compact" HRTF measurements which consist of 128 time-samples [26]. After a successful installation of Libimmacs the HRTF data files are expected to be inside the folder named "*hrtf*" where Libimmacs header is installed, *e.g. "/usr/local/include/immacs/hrtf"*. This path may change among different systems. The name of the file varies, according to the sample rate. The name consists of the "*hrtf_*" prefix, followed by the sample rate and the "*.dat*" extension, *e.g. "hrtf_44100.dat"*. The desired sample rate is provided at the holder's initialization. Two sample rates are currently supported, 44100 and 48000 Hz.

Extension or modification of the current HRTF database is not excluded. For this reason the data format is essential to be provided. Some assumptions are considered to simplify the data parsing

- All HRTF are for zero-th elevation.
- HRTF start from zero-th azimuth and are uniformly spread at [0°, 360°).
- All HRTF have the same size.
- HRTF are placed into the file by an increasing angle.

The data are placed in the file using the following sequence

1. the number of hrtfs (32-bit integer)
2. the number of each filter samples (32-bit integer)
3. 1st HRTF left channel all samples in series (32-bit floats)
4. 1st HRTF right channel all samples in series (32-bit floats)
5. ⋮
6. N-th HRTF left channel all samples in series (32-bit floats)
7. N-th HRTF right channel all samples in series (32-bit floats)

HRTF Data Generator

The HRTF Data Generator is a matlab script that helps to convert the HRTF from raw data in the form readable by the HRTF Holder. Furthermore, it allows resampling of the filters to the desirable sample rates. By default, the raw HRTF data are stored in the file "*mit_compact_hrtf_elev0.mat*" in an $72 \times 128 \times 2$ matrix. This file can be found in the source of Libimmacs.

Frequency Volume Processor

The Frequency Volume Processor allows the manipulation of the volume of each frequency component of an audio period. It is used only indirectly through the Spatial Decoder. Its code lies in the "*immacs_freq_volume_processor.c*" file and it is part of the core interface of Libimmacs. The volume modification is provided by the user in the magnitude scale (decibel). It also supports the calculation of the period energy before and after the volume change.

Memory Interface

Libimmacs provides an abstraction level for memory operations that concern the allocation and de-allocation of raw memory and multi-dimensional matrices. This code lies in the file "*immacs_mem.c*" and the declarations are found in the public Libimmacs interface. The main features are

- **Dynamic allocation of raw memory with optional destructor support.** Provides the same functionality as the malloc function but a user-defined destructor can be called at the time of de-allocation.
- **Dynamic allocation of 1-D, 2-D and 3-D memory arrays of objects of user-defined size.** Very useful tool for easy and condensed allocation of the multiple matrices real-time audio processing requires.
- **All above allocations initialized with zeros.**
- **Unified memory de-allocation interface.** All memory allocations mentioned above (raw memory, N-D, zero-initialized) can be de-allocated using the same function, hiding information about the underlying structure of memory.

Complex Numbers

The processing of audio in the frequency domain requires complex number operations. To facilitate for these operations, Libimmacs provides functions that implement common complex number functions, such as addition, multiplication, division etc. These functions are all inline for speed and they are provided through the "*immacs_complex.h*" header. Despite the fact that the "*immacs_complex_t*"

type is public, this header is used only internally in Libimmacs and is not accessible by the user.

Audio Conversion Tools

Libimmacs features some useful functions to convert between common sample types and audio formats. These audio conversion tools are given in the "*immacs_convert_tools.c*" file and are part of the public Libimmacs interface. The provided functions allow conversion between single/multiple Libimmacs samples (float) and Pulse Width Modulation samples (16-bit integers) and conversion between interleaved and non-interleaved audio streams.

Various Tools

Some generic, commonly used functions by the Libimmacs objects have been grouped together in the file "*immacs_var_tools.c*". These functions allow window-initialization such as blackman, hann and hamming, sorting algorithms, "linspace" initialization and angular distance estimation. These functions are declared through the public interface of Libimmacs.

Double Linked List

Operations regarding the manipulation of double linked lists can be found in "*immacs_list.h*" file. This list implementation is part of the core functions and is used only internally from other Libimmacs objects. Functionalities include initialization of the list, append of new objects, searching, unlinking, and finally flushing, that automatically uses immacs memory manager to de-allocate nodes.

Chapter 4

Immacssip - the ImmACS client

Immacssip is the outcome of incorporating the features of ImmACS into Baresip. This is feasible by using Libimmacs, which has been thoroughly studied in chapter 3. In this chapter, we describe the design choices and the technical aspects of the implementation of Immacssip. We provide a basic framework for someone to understand and even modify Immacssip in code level if it is needed. All the modifications that converted Baresip to Immacssip can be found in the code by searching for the "immacs mod" keyword throughout the project.

4.1 Goals and challenges

Libimmacs has a very modular structure that facilitates its incorporation into Baresip. However, this must be performed in such way so that the user can take advantage of all the features of Libimmacs. In addition, the desired ability to receive and fully control the audio streams of all the peers raises the complexity of the system even more. The implementation of these properties becomes even more challenging as we restrict only to changes in Baresip, leaving libre and librem untouched for convenience regarding portability. To sum up, the main goals of the implementation of Immacssip are

- Incorporate Libimmacs functions into Baresip.
- Receive multiple audio and video streams per call.
- Manipulate audio streams through Immacs Control.

The first two tasks require modifications in almost independent parts of Baresip, so each operation can be applied without any conflict on the other. We continue with the analysis of these three tasks in the next sections.

4.2 Incorporating Libimmacs to Baresip

4.2.1 Default audio structure

The integration of Libimmacs to Baresip requires in-depth changes to the audio transmit and receive pipeline. We find it useful to give a short description of these pipelines here, to provide the basis for the following section.

Transmit

The flow of the transmit pipeline starts at the audio source, which is the module that provides the data from the audio interface. For this function, Baresip integrates with various audio APIs, such as ALSA, CoreAudio and PortAudio. The data are then stored in the audio buffer, waiting to be read when needed. Data are consumed from the audio buffer in packets of constant size, which is defined by the packet time (ptime) parameter. When an audio packet is read from the buffer, an optional resampling process takes place and audio filters are applied, if any. The encoding of audio is performed afterwards and lastly, the encoded audio is packetized and sent using the Real-time Transport Protocol (RTP). A visual view of the default transmit pipeline is given in Figure 4.1.

Receive

The receive pipeline is the inverse equivalent to the transmit pipeline. Audio data is received encoded in the RTP protocol. RTP and audio decoding takes place, and then the audio filters are applied. Optional resampling is performed next and the data are stored in the audio buffer. Finally, the data are provided to the audio interface for reproduction through the audio player module. The default receive pipeline is displayed in Figure 4.3.

Asynchronous buffering

The audio buffer used in both transmit and receive pipelines allows the asynchronous read and write of data. This is necessary so as to permit the interchange of data between the audio thread and Baresip's main thread.

Concerning the transmission part, the audio thread writes data on the buffer and the main Baresip thread polls it at packet-time intervals using the internal timer mechanism of Baresip. This is the default option, where the process from resampling through sending blocks the main thread. Baresip also provides the option for asynchronous transmission of audio, where a separate thread is used for encoding and sending the audio data, relieving the main thread from this task.

Regarding the receive part, the main thread listens by continuously polling in a loop for incoming packets. All the processing of the received packets is performed in the main thread, blocking all other operations.

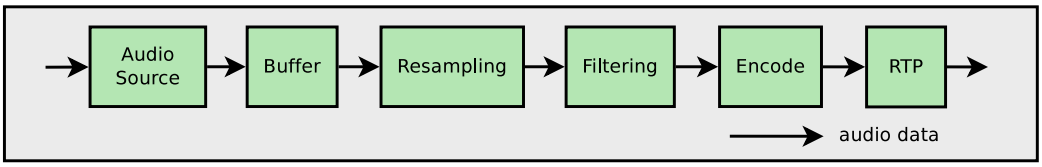

Figure 4.1: Baresip audio transmit pipeline.

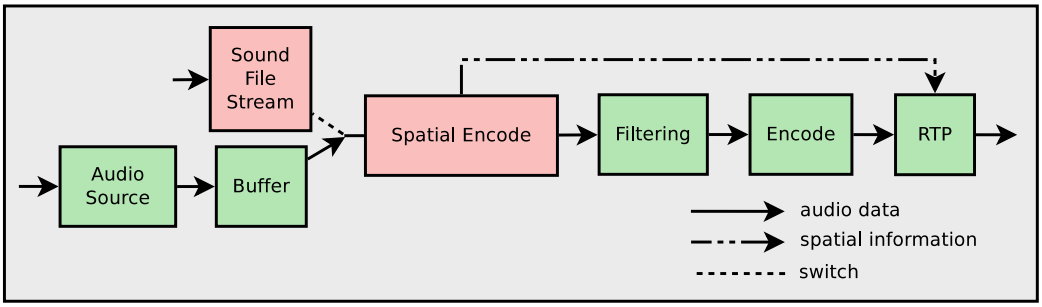

Figure 4.2: Immacssip audio transmit pipeline. The pink components indicate the modifications to the default transmit pipeline. Notice that the resampling part does not exist here.

4.2.2 Custom audio structure and implementation details

Considering the audio structure reported above, incorporating Libimmacs requires the following modifications in the structure of Baresip:

1. Embody the spatial encoding of Libimmacs before the default Baresip audio encoding.
2. Embody the spatial decoding of Libimmacs after the default Baresip audio decoding.
3. Encode and transmit the spatial meta-data within every audio packet.
4. Allow capturing and reproduction through files at disk.
5. Extend the default Baresip configuration to support the libimmacs-related options.

Encoding and decoding

For the encoding part, we combine the DOA Estimator and Spatial Encoder objects of Libimmacs. To support decoding, we combine the HRTF Renderer, VBAP Renderer and Spatial Decoder components. For the handling of the spatial meta-data we use the Spatial Information object. For the special case where audio data need to be read or written to disk files, we utilize the Sound File Stream object. The reported objects exist only for the duration of each call.

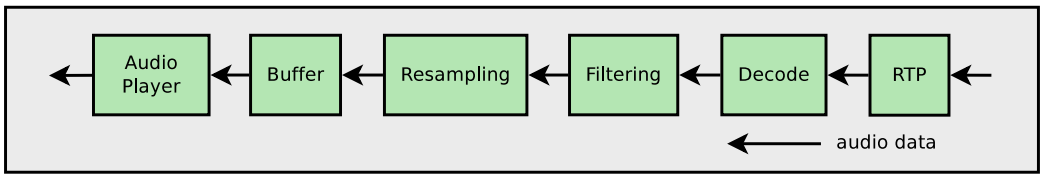

Figure 4.3: Baresip audio receive pipeline.

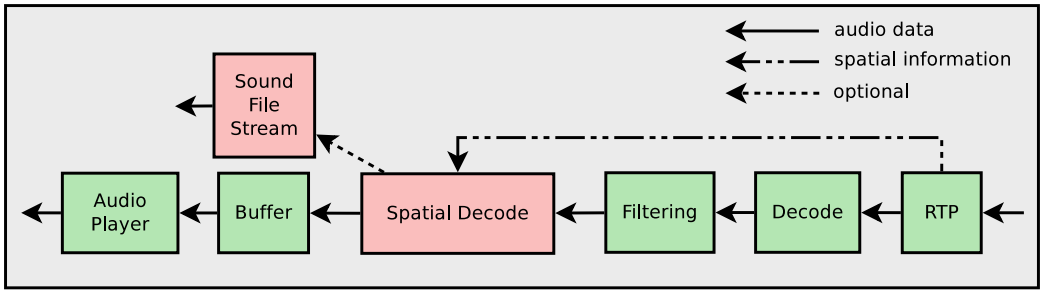

Figure 4.4: Immacssip audio receive pipeline. The pink components indicate the modifications to the default receive pipeline. Notice that the resampling part does not exist here.

Through the configuration interface of Baresip we provide the ability to set various parameters regarding the objects' initialization, such as the number of microphones of the array, the radius, the number of the loudspeakers, the name of the sound file etc. We have used the default options of Baresip as an example so as to add the required for ImmACS. A detailed description of the ImmACS configuration parameters can be found in section 7.3.4.

The encoding and decoding components of Libimmacs are placed in-between the audio buffer and the audio filters in the respective pipelines. We have disabled the optional resampling function as it requires special care and has no significant importance for this project. The only allowed sample rates for ImmACS are 44100 and 48000 Hz, as defined from the limitations of Libimmacs. Concerning the encoding process, audio data are read either from the audio buffer or from audio files at disk. Concerning the decoding process, audio data can be also exported to disk if recording to files is enabled. The audio stream of each peer is recorded in a separate disk file.

Encoding of the spatial information

The ImmACS encoded audio is always accompanied with spatial information, which stores the all the necessary instructions for its decoding [1]. It is crucial that every ImmACS-encoded audio packet contains also its corresponding spatial information.

To realize this task we took advantage of the optional extension header that the RTP protocol provides. This feature allows to add custom data to the header

that will be transferred untouched through the web. There is no limit on the size of the data and an application-specific ID is used so that different kinds of data can be discriminated. In the default implementation of Baresip this extension is not used, so there are not any conflicts with our implementation. In fact, Baresip is designed to ignore the extension header of the received packets, if it exists.

In the transmit pipeline, the spatial information is provided in its uncompressed form by the Spatial Encoder. The parts of audio filtering and encoding are skipped as they concern only audio. At RTP level, the spatial information is compressed using the corresponding function provided by Libimmacs, which implements the compression described in [1]. Data are then written to the extension header, before the audio encoded data. We arbitrarily use the number 456 as the application's ID. The transmit pipeline of Immacssip is displayed in Figure 4.2.

In the receive pipeline, the spatial information of the received packet is separated from the audio data in the RTP level. It is decompressed and given as input to the Spatial Decoder, among with the decoded audio data. The receive pipeline of Immacssip is displayed in Figure 4.4.

Real-time safe buffering

The default audio buffer of Baresip is designed as a linked list where audio data are stored as list nodes. This implementation is unsafe for real-time operation for two reasons. Firstly, the memory for the nodes is allocated instantly at the time of their creation, which implies regular system calls for memory. Secondly, the asynchronous operation utilizes memory locks, which are also a bad practice for real-time coding. For the above reasons, we replaced the audio buffer with a ring buffer, which performs optimally in real-time. For convenience, we have written a wrapper class so as to provide the same interface with the default buffer, but the implementation internally uses the ring buffer found in the source of PortAudio [11].

Discussion: audio stream flexibility

From its definition, the ImmACS audio stream allows for great flexibility regarding reproduction. Specifically, an ImmACS encoded audio stream can still be reproduced as simple, monophonic audio. This permits the communication between a client that transmits immersive audio and a client that receives only simple audio. In that case, the latter can ignore the spatial information and reproduce the signal monophonically.

Inversely, the explicit reproduction of a monophonic stream immersively is also supported. This can be achieved by creating custom spatial information and assign it to the monophonic audio. Of course, the reproduction of multiple separate sources from one side will not be possible, but even the reproduction as only one source can be very useful, by assigning all the frequency components to the desirable direction. This is for the case where only one person uses a client and transmits simple monophonic audio. The other clients will be able to reproduce his stream

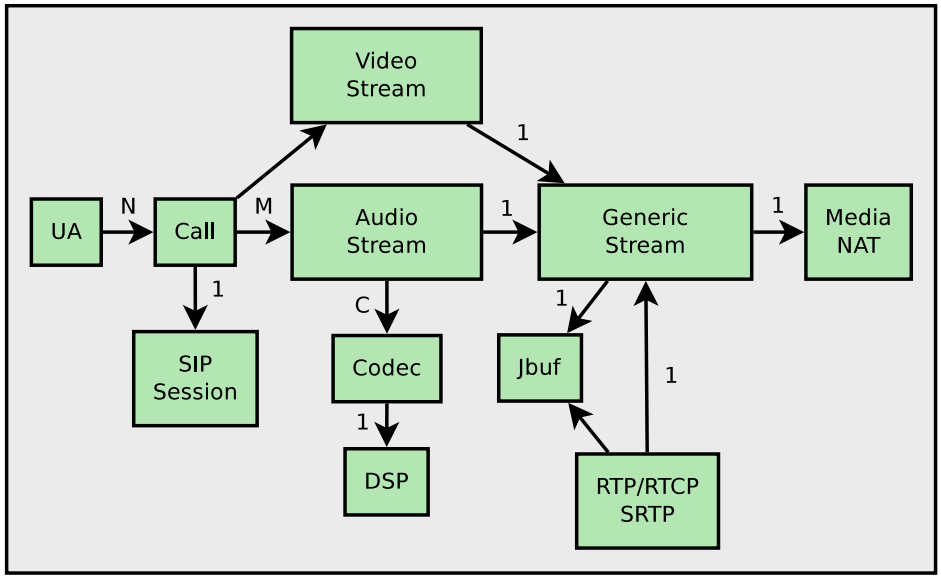

Figure 4.5: Baresip architecture.

immersively from the desired direction.

Generalizing the above cases, the spatial information is an independent component that can be easily manipulated. Even in the case where ImmACS audio is supported by both the transmitter and the receiver, the latter can change the directions of the sources according to its preference. This idea is the basis for the implementation of the spatial filters of Immacssip, controlled by the graphical interface.

4.3 Multiple streams per call

4.3.1 Default communication model

Baresip is designed to support only one transmit and one receive audio stream per call. The same applies also for video. Even when a client communicates with multiple others in teleconferencing mode, he sends one audio stream and also receives one audio stream. This is possible because the conference server performs transcoding and down mix of all the streams to one. Practically, Baresip is designed to communicate only with one peer per call, regardless if this peer is the conference server or another VoIP client. Figure 4.5 displays the default architecture of Baresip, as provided by its official documentation. Although the theoretical number of streams is given as an arbitrary M number, in practice is two streams for audio and two for video.

To allow Baresip to receive multiple streams per call, the whole communication model with the server must be changed. The server should be modified to send all the streams to each client without performing any transcoding or down mix. The

clients, on the other hand, must be able to receive and reproduce multiple streams. Next, we continue with describing our custom client-server communication model and we deepen into the implementation details.

4.3.2 Custom communication model

Considering that both Immacssip and Bareserver are based on Baresip, which is a VoIP client, their connection is established as peer-to-peer. Each SIP account in the server represents a different conference room. The client dials to the server's account, i.e. room that wants to join. The server automatically answers the incoming call. When the connection is established, the client starts to transmit its audio and video streams to the server.

On the other hand, the server starts the relaying of the audio and video packets. Before relaying, it marks each packet with a unique ID, which corresponds to the client that sent it. Alongside, it continuously transmits a list to all the clients as side information, containing the IDs and various information about the peers.

A client will receive the packets of all the peers. Initially, the client is unaware of how many and which peers exist in the room. Each client holds its own internal list of the participants and updates it every time it receives the new list from the server. When a packet is received, if its owner exists in the internal list then the packet is fed to the corresponding receive stream for decoding and reproduction. If the owner does not exist in the list, the packet is simply dropped. The identification of the packet's owner is performed using the packet ID.

Finally, if the client has not received the list of the participants from the server for 60 continuous seconds, then the server is considered inactive and the call is terminated. Respectively, if the server has not received any audio or video packet from a client for 60 continuous seconds, it considers the client inactive and terminates the call.

4.3.3 Implementation details

The architecture of Immacssip is depicted in Figure 4.6. We proceed here with further analysis of our design choices.

Peers

To allow Baresip to receive multiple audio and video streams per call, we introduce the notion of **peers per call**. From the point of view of an arbitrary client, a peer represents another room participant, whose packets are received through the relay server. We implement the notion of a peer as a new structure, i.e. class. A call instance may contain zero or more peer instances. The peers are created and destroyed dynamically at the course of a call, according to the peer list received as side information by the ImmACS server.

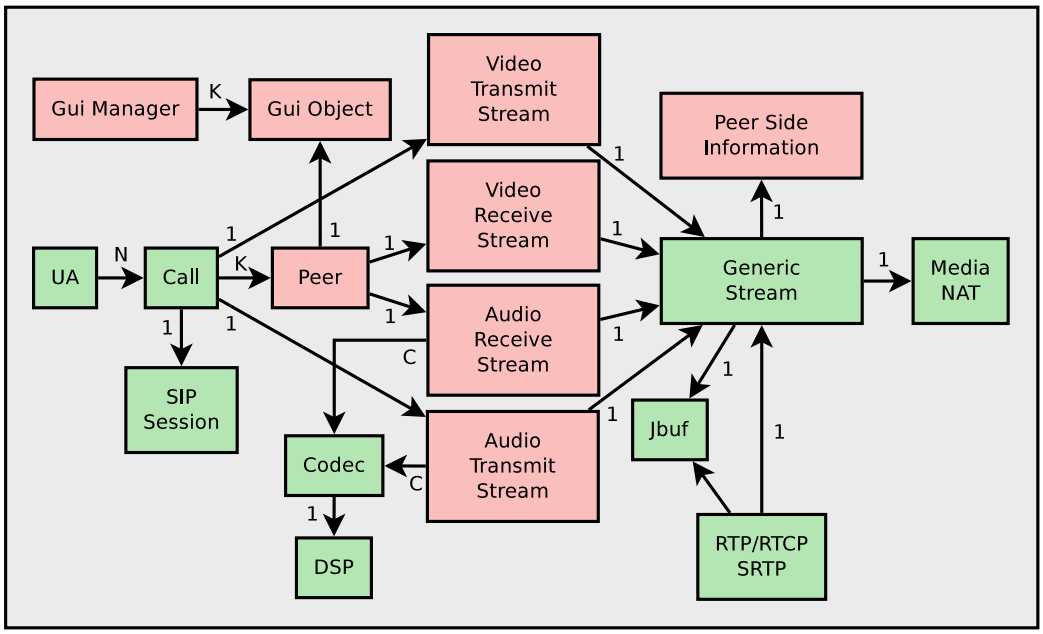

Figure 4.6: Immacssip architecture. The pink components indicate the modifications to the default architecture.

Separation of the transmit and receive streams

In the default implementation of Baresip, the audio/video transmit and receive streams are created and destroyed together, at the time that a call starts and ends respectively. This is not the case however for Immacssip, where each receive stream corresponds to a different peer that can be created and destroyed at any time in the course of a call. For this to be possible, we separate the creation and the destruction functions of the transmit and receive streams in the code.

Multi-threaded decoding

The decoding and reproduction of multiple streams per call implies an unbounded computational cost, that is directly affected by the number of the conference participants. Furthermore, Baresip processes the received packets in the main thread, blocking all other operations. This can cause a major processing bottleneck when multiple packets must be processed simultaneously. To increase the throughput of the system and take advantage of the multi-core systems widely used today, we implement multi-threaded decoding of the packets.

We use a thread for each receive stream, which corresponds to two threads per peer; one thread for the audio stream and one for the video stream. For clarity and easier maintenance, we have transferred the code regarding the decoding in a separate file. A thread is starting and stopping when its corresponding receive

stream is created and destroyed respectively. After a packet's owner has been identified, the packet is given to the corresponding receive thread for further processing and reproduction, unblocking the main thread. The packets are provided to each thread using a FIFO queue, implemented using a linked list.

4.4 Immacs Control integration

The integration of ImmACS Control to Immacssip consists of two main parts. Firstly, the communication model must be defined so as to allow the interchange of data between the two applications. Secondly, the audio effects must be implemented in the audio pipeline of Immacssip. Notice that no signal processing takes place in the graphical interface; it only provides meta-data that describe the spatial filters to the audio thread. However, as the spatial audio filters conceptually belong to the graphical interface, we choose to describe only the communication model here. We provide the signal processing details in the chapter 5 of Immacs Control.

4.4.1 Immacssip-ImmACS Control communication model

While Immacssip is implemented in C, we chose to implement ImmACS Control in Java, so as to take advantage of the easy and flexible API that Java provides. This choice requires the specification of the communication model between Immacssip and ImmACS Control, as both are independent applications. To allow the interprocess communication of ImmACS Control and Immacssip, we utilize datagram network sockets. Data are transferred as messages through the network. For the purposes of this project, ImmACS Control is expected to operate in the same system with Immacssip. However, network sockets also permit operation on different systems. The communication model involves the exchange of data with the audio thread, so it must comply with the restrictions of real-time processing.

Output to ImmACS Control

Immacssip transmits to the GUI data for each peer separately. These data are transmitted as constant-size messages and have a unified format for all the peers. The transmission is performed ten times per second for each peer, so as to keep the data constantly updated. ImmACS Control holds its own internal list of peers. The ID of the corresponding peer is included in each message, to help ImmACS Control associate the messages to their owners. If the owner of a message does not exist in the internal list, it is registered as a new one. When a peer leaves the room, a flag is included to a "dummy" message to notify the GUI accordingly. The special case where the ID of the message is zero is used to indicate that the data belong to the local client. This case is used for visualizing the transmitted audio stream. The main body of the message contains static information like the peer's alias name and URI and dynamic information such as the current directions of the

sources. There is no need to transmit the static data in every packet, however we prefer it so as to simplify the communication model.

Input from ImmACS Control

Similarly to the output messages, Immacssip receives data separately for each peer from the GUI. The input messages have a constant size and a unified format for all peers, however the constant update of data is not necessary. New data are transferred only when there is an interaction with the user, *e.g. the manipulation of the volume or the direction of a peer*. Each message contains the ID of its corresponding peer, to help Immacssip assign the data to its owner. An input message contains all the settings for a peer, regardless which setting has been changed by the user *e.g. the mute/solo flags, the volume modification, the spatial effects etc.* In a more defined model only the altered settings would be transmitted in each message, however this raises the complexity and has no significant value for this implementation.

4.4.2 Implementation details

Each peer object is associated with a **GUI object**. A GUI object contains two FIFO queues; one queue to provide data from the GUI to the audio receive thread (input queue) and one to provide data from the receive thread to the GUI (output queue). These queues are based on ring buffers, because asynchronous real-time operation is required. The queues work only as the storage of the data; we also introduce another object, the **GUI manager**, which is responsible for sending and receiving these data to and from the GUI. The GUI Manager runs independently of the receive threads in the main thread of Baresip.

For data transmission, the GUI Manager uses the internal timer mechanism to transmit data ten times per second. It iterates all the peers and sends the data found in their output queues. It only sends the last node of the queue, ignoring the rest, so as to provide the most recent data to the GUI.

A similar procedure is followed when data is received from the GUI. The GUI manager appends these data at the input queue of the corresponding peer. The audio thread uses only the last item of the queue at each time, ignoring the rest.

The queues in fact compensate for the possible delays due to the asynchronous exchange of data between the audio receive threads and the GUI manager. Immacssip transmits data to Immacs Control using the port 7484 by default, but this can change if the user wishes to. For receiving the data from the GUI, it uses the very next port of what he sends, i.e. 7485. The association of the GUI object and GUI manager to the overall architecture can be seen in figure 4.6.

Chapter 5

Immacs Control - the graphical user interface of ImmACS

In this chapter we explain our design choices for the implementation of the Immacs Control graphical interface. Immacs Control is written in Java, to facilitate the creation of the graphical environment. The general aspects of the communication model of Immacs Control with Immacssip are reported in Section 4.4.1. Here we explain that model from the point of view of the graphical interface. Moreover, we provide the details for the implementation of the two spatial filters, the Spatial Equalizer and the Spatial Mapper.

5.1 Communication with Immacssip

Immacs Control utilizes two threads for the communication. The main (default) application thread, that is used for sending and receiving the data from Immacssip, and the graphics thread that is used to update the graphics. The main thread polls for incoming or outgoing messages in a continuous loop. In the communication loop, the outgoing messages are handled first and then the incoming messages.

Outgoing messages are sent only when the user interacts with the interface, i.e. *changes the volume of a peer*. The interactions take place in the graphics thread and the data for the peers change accordingly. To notify the main thread, we use a "dirty" bit that is enabled in every interaction. The main thread iterates the data of all the peers and checks their dirty bit in each loop. If the "dirty" bit is enabled, it sends the new data to Immacssip and disables it again. To avoid overloading Immacssip with packets, transmission takes place every 200 milliseconds. If a message has been received, it is immediately invoked for handling at the graphics thread. If there is no received message the application waits by blocking for 50 milliseconds. We use memory locks to ensure the safe exchange of data between the graphics and the main thread.

5.2 The spatial equalizer

Through the Spatial Equalizer, the user manipulates the **volume in relation to the direction** of the sources, using a graphical curve as a visualized feedback for the changes. We provide the details of its operation, from a user's perspective, in Section 7.4.2. We recommend the reader to read that section first, as a basis for the next analysis.

Theoretical framework

Given a separated source signal S in the direction x, from an ImmACS audio stream, we want to attenuate its volume according to the curve of the equalizer. The user can add several nodes to alter this curve. Each node N_i has three attributes, x_i, y_i, r_i that represent its position on the x-axis (degrees), its position on the y-axis (decibel), and its range (degrees), respectively. The volume attenuation, in decibels, of the direction x due to the node N_i is given as

$$V_i(x) = \frac{y_i G(A(x, x_i), 0, \frac{r_i}{2})}{G(0, 0, \frac{r_i}{2})} \tag{5.1}$$

where $G(\xi, \mu, \sigma)$ represents the normal distribution with mean μ and standard deviation σ sampled at ξ, and $A(x, x_i)$ is the angular distance of x from x_i, given as

$$A(\alpha, \beta) = 180 - ||\alpha - \beta| - 180| \tag{5.2}$$

Then, the overall volume attenuation at x is given as

$$V(x) = \sum_{i=1}^{K} V_i(x) \tag{5.3}$$

where K is the number of the total nodes of the equalizer. If no nodes exist ($K = 0$), then $C(x)$ is set to 0. We choose the values for the nodes to operate cumulatively, because after practical tests we found that it facilitates the manipulation of the graphical curve. Even though adding decibel values is not valid in general, in this case it is. This is because these values do not represent real measurements, but are given arbitrarily from the user. Finally, the attenuated signal is given as

$$S' = S\sqrt{10^{V(x)/10}} \tag{5.4}$$

In practice, many ad-hoc changes where introduced to the above framework so as to also support the "solo" function. The general idea, however, is aptly depicted above.

Implementation

The implementation of the Spatial Equalizer utilizes two classes, the EQ class and the GEQ class. The EQ class holds all the data for the equalizer and provides the functionality of 5.3. The GEQ class instantiates the EQ class internally and handles all the interaction with the user. The user adds graphical nodes to the GEQ and the GEQ converts their x and y coordinates from pixels to degrees and magnitude respectively. The magnitude is converted from pixels using a logarithmic formula, to provide a fine accuracy for small changes and rough accuracy for large changes in the volume.

5.3 The spatial mapper

The Spatial Mapper allows the user to independently change the direction of the sources of an ImmACS audio stream. The basic details of its operation are provided in Section 7.4.2.

Theoretical framework

Given a separated source signal S in the direction x, from an ImmACS audio stream, we want to change its direction according to the spatial mappings provided by the user. The user can insert nodes that represent these mappings. Each node N_i has three attributes, x_i, y_i, r_i that represent its position on the x-axis, y-axis and its range, respectively, all in degrees. The new direction x' of S, due to the node N_i is given as

$$x' = \begin{cases} y_i, & \text{if } A(x, x_i) < \frac{r_i}{2} \\ x, & \text{else} \end{cases} \tag{5.5}$$

where $A(x, x_i)$ is the angular distance of x from x_i as defined in 5.2. In the case where $x' = y_i$ applies for more than one node in the above equation, then the value of the node that was created first in time is used.

Implementation

The Spatial Mapper implementation is very similar to that of the Spatial Equalizer. It utilizes two classes, named SpatialMapper and GSpatialMapper that concern the core and the graphical interface of the effect respectively. The user adds graphical nodes that are converted from pixels to degrees, and the core of the effect is updated accordingly. Notice that in this case both axes represent degrees, so both work in a wrap-around fashion.

Chapter 6

Bareserver - the ImmACS server

In this chapter, we describe the implementation of Bareserver, the server of the ImmACS system. The Bareserver is a necessary part of the communication, but its contribution is quite minimal. As all the clients receive all the streams, there is no need for transcoding and down mix at the server. It only performs plain relaying of the packets to the clients. Based on the simplicity of the server and our experience on Baresip from the implementation of the client, we considered that modifying Baresip as a server was the best and fastest solution for this project. All the modifications that converted Baresip to Bareserver are marked with the "bareserver mod" comment in the code.

6.1 Communication with Immacssip

Session initiation

The Bareserver handles each connection with an Immacssip client is handled as a different call. The default implementation supports this for peer-to-peer calls, however it bounds the number of calls to four per user account. For Bareserver, this practically means that only four participants per conference room are allowed. We change this number arbitrarily to twenty, which is a sufficient number for the purposes of the project.

When the server establishes a new connection with a client, it assigns it with a unique ID that is preserved for the duration of the call. The ID is implemented as an integer, starting from one and increased by one for every new client. For the implementation we use a global ID counter that is used for every call, that is initiated when Bareserver starts.

Packet relay

The Bareserver receives the packets from all the clients. Their RTP header is extracted first, using the framework of libre. The remaining packet data are then

provided to the stream level, among with the RTP data, for further decoding. In that point, we add the ID of the packet's owner to the original RTP header, and re-encode it into the packet, without changing any other data. The ID is added to the first CSRC slot of the RTP header. The packet is then sent to every client connected to the same SIP account. While the encoding of the RTP header is by default performed in the RTP level, we implement it in the stream level. This is to avoid any changes to the code of libre due to the encoding of the ID. We instead copy the required code from libre where is needed and use it from Baresip.

Peer side information

The Bareserver transmits the IDs and other useful information about the clients as side information, to facilitate packet discrimination. The side information is transmitted two times per second to each client, through the application specific Real-time Transmit Control Protocol (RTCP). The transmission is performed in the main thread of Bareserver and we use the internal timer mechanism of Baresip to transmit it periodically. The side information contains the ID, the display name (alias) and the URI of each client. We create the "peersinf" object to handle the side information.

6.2 Multi-threaded relaying

By default the relaying of the packets is performed in the main thread of Bareserver, blocking it from receiving other packets or establishing new calls. To increase the throughput of the system and relieve the main thread from the task of relaying, we provide optional multi-threaded relaying. We use an arbitrary number of worker threads, that can be set by the user through the configuration of Bareserver. Each worker thread holds a FIFO queue of the packets to relay. The main thread appends the received packets to these queues after adding the appropriate ID.

6.3 Echo server

The Bareserver supports a special function where it relays the received packets only back to their senders, without performing any sharing to the others. This function is not useful for the common user; we implemented it to facilitate the testing of the system. The user can set the server in "echo" mode by enabling the corresponding parameter in the configuration file of Bareserver. In that mode, the side information is also affected. Notice that each client is not aware of its own ID, because the ID is set at the server. For this reason, each client receives side information that contains only itself, so as to be able to reproduce its own packets.

Chapter 7

ImmACS from a user's perspective

In this chapter we describe ImmACS from a user's perspective. We provide all the necessary instructions needed in order for someone to build, install, configure and finally operate ImmACS. An overview of the system is provided at the introduction, which is recommended for the user to read before proceeding with this chapter.

ImmACS is an experimental project that utilizes the original Baresip code with many ad-hoc modifications. While the most expected cases of function have already been tested and working, the system stability is not yet guarranteed for any possible configuration. For this reason, it is recommended to carefully follow to the guidelines below.

7.1 Bareserver

7.1.1 Dependencies

As a modification of Baresip, Bareserver requires **libre** and **librem** libraries to operate. It has been successfully tested and working using **libre-0.4.13** and **librem-0.4.6** versions. The source code for these libraries and installation instructions can be found at **www.creytiv.com**. No changes in those libraries are needed in order for Bareserver to operate.

7.1.2 Installation and first run

After the installation of libre and librem libraries, Bareserver can be built on both **OS X** and **Linux** operating systems by typing the following commands in a terminal

```
$ cd bareserver
$ make
$ sudo make install
```

Run Bareserver by typing **bareserver** from anywhere in a terminal. At the first run, the configuration files "*accounts*" and "*config*" will be created inside the "*.bareserver*" directory, which lies the user's home directory. Close Bareserver by pressing "**q**", so as to modify the configuration files.

7.1.3 SIP account setup

The "*accounts*" file holds the list with the Bareserver SIP accounts and their configuration. Different SIP accounts represent different conference rooms. Some useful examples and configuration instructions are provided in the default "*accounts*" file. The user can follow these examples to create conference rooms according to his needs. Two SIP accounts are provided by default, "*room_A*" and "*room_B*".

Each SIP account consists of four parameters. The room name, *e.g. room_A*, is arbitrarily set by the user and is what an ImmACS client will **dial** in order to connect to that conference room. The other three parameters describe the basic communication protocol, which consists of the audio and video codecs and the packet time (ptime). **It is obligatory for a client to support the exact same communication parameters in order to connect to a conference room. Otherwise, the call is rejected.** These three parameters are optional; if no communication parameters are provided by the user, the defaults are **opus/48000/1**, **ptime=20** and **MP4V-ES** for audio codecs, packet time and video codecs respectively.

Packet time controls the packet size, which indirectly sets a trade-off between audio latency and quality. Lower values reduce the audio latency but increase the risk of audio buffer underflows occuring, which can cause degradation at the audio quality. Larger values provide robustness as it concerns underflows but cause larger end-to-end audio delays.

7.1.4 Configuration

The "*config*" file allows the user to configure basic parameters of the Bareserver operation. The basic structure of the default Baresip "*config*" file has been preserved but a lot of unnecessary options have been removed. Most of the remaining options are considered to be used as in the default Baresip configuration. Here we will focus only at those that have a significant meaning for Bareserver.

The **sip_listen** option must be enabled (not commented) for the Bareserver to be able to accept calls. The default listen port for Bareserver is set to **4040** in contrast to the SIP default which is **5060**. This is to avoid any possible conflict with other SIP clients in the same machine.

The **echo_server** option has been created especially for Bareserver and does not exist in the default Baresip configuration. When enabled, the Bareserver sends the packets back to their senders. The same communication protocol for the conference rooms applies here, but the clients connected in any room, receive back only their own packets and nothing else. This option is useful for experimental

purposes such as client transmit and receive testing and latency measurements. It is disabled by default.

Another new parameter added at Bareserver is the "**relay_threads_cnt**". This parameter controls the number of worker threads that will do the relay of the received packets to all the clients. This is to increase the system throughput when Bareserver is used in multi-core systems. When set to zero, no worker threads are used; the system transmits a packet immediately when it is received, blocking the system. For a positive thread count, the packets are fed to the worker threads for transmission, keeping the system unblocked so as to be able to receive other packets. This parameter is set to zero by default.

7.1.5 Operation

After the "*accounts*" and "*config*" files have been set, the system is ready to run. Type "**bareserver**" at a terminal to run Bareserver. At the initialization, the account information and some info of the configuration file will be printed. Pressing the "**h**" button will print the help menu.

All the information of the connected clients is printed and constantly updated for the current toggled room. The client information is printed in the following format:
Call: (**client id**) <**client alias**> [**elapsed time**] audio=**autx**/**aurx** video=**vitx**/**virx** (bit/s)
where **autx**/**aurx vitx**/**virx** is the audio transmit/receive, video transmit/receive bitrate to/from that client respectively. For example
Call: (1) <Steve_Smith> [0:10:48] audio=100345/50885 video=1220994/617994 (bit/s)
When no packet has been received from a client for 60 continuous seconds, Bareserver considers the client inactive and closes the connection, releasing any resources and informing the rest of the connected clients appropriately.

7.1.6 Uninstall

Bareserver can be uninstalled by typing

```
$ cd bareserver
$ sudo make uninstall
```

Note that the "*.bareserver*" directory and its contents are not removed using the above commands and must be manually removed if the user wishes to. This allows the bareserver configuration files to remain untouched in case of a future re-install.

7.2 Libimmacs

Libimmacs is a necessary dependence of Immacssip. In this section we describe how to setup Libimmacs before installing Immacssip.

7.2.1 Dependencies

Libimmacs depends on **cmake** and three external libraries, **fftw**, **itpp** and **libsndfile**. At **Ubuntu Linux** these packages can be downloaded and installed using the default package manager. In a terminal, type

```
$ sudo apt-get install cmake libfftw3-dev libitpp-dev
  libsndfile1-dev
```

At **OS X** these libraries can be easily installed using the **homebrew** package manager. Homebrew does not exist by default in OS X. To install it follow the instructions at **brew.sh**. After homebrew is installed, Libimmacs dependencies can be downloaded and installed on OS X by typing

```
$ brew install cmake fftw itpp libsndfile
```

7.2.2 Installation

After all the required dependencies have been installed, Libimmacs can be installed on both **OS X** and **Linux** systems by typing

```
$ cd Libimmacs/build
$ cmake ..
$ make
$ sudo make install
```

7.2.3 Usage

After Libimmacs is installed, it can be used as a shared library to any C/C++ coding project, so to provide the ImmACS functionalities. The header files can be included in a C/C++ file as

```
#include <immacs/immacs.h>
```

while the library can be linked at compile time as **-limmacs**.

Although Libimmacs interface is very clear and intuitive to use, a full on-code documentation is required to assist future developers. For the scope of this project however, this is omitted but it is planned as a future work. In any case, it is strongly recommended for someone to read chapter 3 before using Libimmacs, in which we describe the design of the library. This chapter can prove to be even more useful than the on-code documentation, because there Libimmacs is explained thoroughly.

7.2.4 Uninstall

Libimmacs can be completely removed from a system by typing

```
$ cd Libimmacs/build
$ cmake ..
$ sudo make uninstall
```

7.3 Immacssip

7.3.1 Dependencies

Similarly to the Bareserver, Immacssip is also a modification of Baresip, thus it requires **libre** and **librem** libraries to operate. Installation instructions for these libraries can be found in section 7.1.1.

In addition, Immacssip depends on **Libimmacs** library to operate. Libimmacs can be installed as described in section 7.2.

Immacssip can operate using any audio codec that supports one-channel audio streaming of 44100 or 48000 Hz sample rate. However, the default recommended for ImmACS is the **Opus** audio codec, which provides very good audio quality and supports very low latencies and bitrates. For video operation the **MP4V-ES** video codec is recommended, that has been tested and works well for the purposes of ImmACS. These codecs must be manually installed before building Immacssip. On **OS X** systems, they can be easily installed using the **Homebrew** package manager. Homebrew can be installed using the instructions at **brew.sh**. After Homebrew is installed, type

```
$ brew install opus ffmpeg
```

On **Ubuntu Linux** these codecs can be installed by using the default package manager. Some extra packages are also required for the default operation of video source and display. To install all the audio and video codecs type

```
$ sudo apt-get install libopus-dev libavcodec-dev
```

To install the packages for video source/display type

```
$ sudo apt-get install libv4l-dev libxext-dev
```

7.3.2 Installation and first run

After all the dependencies for Immacssip have been set up, it can be installed by typing

```
$ cd immacssip
$ make
$ sudo make install
```

As in Bareserver, Immacssip uses configuration files that are generated at the first time the program runs. After installation, type **immacssip** in a terminal to run Immacssip. A notification that the configuration files have been created should be displayed when the program starts. **A warning message that no SIP account exists will be also printed due to the fact that there are no active SIP accounts yet.** Press "**q**" to quit. The files "*accounts*" and "*config*" will be created inside the folder "*.immacssip*" in the user's home directory.

7.3.3 SIP Account Setup

The default "*accounts*" file contains a not active (commented) SIP account example, to work as a basis for the user to configure his own SIP account. The user must replace the default IP and port with those of the Bareserver. Simple instructions are provided inside the "*accounts*" file.

Similarly to the Bareserver, each SIP account in Immacssip consists of the four basic communication parameters, plus the display name of the account, which is how the account will be displayed to the other clients; its alias. The same rules for the communication parameters apply here; **the client must support the communication protocol of the room it wants to connect to, otherwise the call is rejected.** All four communication parameters are optional. The defaults are **opus/48000/1**, **ptime=20** and **MP4V-ES** for audio codecs, packet time and video codecs respectively.

At initialization, Immacssip will try to locate the corresponding modules of all the codec parameters at every account. If a module does not exist, an error message will be printed. The Bareserver on the other hand, does not search for these modules. The codec parameters at the Bareserver's accounts work only as a description of the communication protocol that the clients must support.

7.3.4 Configuration

The default "*config*" file of Immacssip is based on the default "*config*" file of Baresip, but some unnecessary options have been removed and a lot of ImmACS-related options have been added. The added options allow the user to configure various parameters regarding ImmACS input and output.

ImmACS Input

The **input** parameters concern immersive audio capturing. The option **immacs_input_enabled** allows the user to enable or disable the ImmACS input operation. When disabled, the default method of Baresip for audio input is used. The transmited audio stream will not contain any spatial information. ImmACS input is disabled by default.

When the ImmACS input is enabled, a real or simulated (from files on disk) circular microphone array is used for audio input. The user can set the number of microphones of the array, the array radius (in meters) and the maximum number of sources, using the parameters **immacs_input_mic_N**, **immacs_input_radius** and **immacs_input_source_N** respectively.

The actual number of sources varies at execution time and is affected by the source thresholds. The source thresholds control the sensitivity of the system on detecting possible sources. Lower thresholds will increase the sensitivity of the system but it will also make it prone to spurious sources or environmental noise. On the other hand, a system with higher thresholds will tend to detect

more prominent sources. The threshold of each source can be set using the **immacs_source_$SN_threshold** template. The **$SN** field is replaced by the source number which is a positive integer starting from zero. For example, *immacs_source_0_threshold 0.22* sets the threshold of the first source at 0.22. A threshold can take any positive real value from zero to one. A threshold entry for each and every source must be declared, or else an error will occur.

The ImmACS input system allows the user to simulate a microphone array setup by using audio files from the hard disk. This feature is very useful for experimental and testing purposes. The **immacs_input_from_file** parameter is used to enable or disable this function. When disabled, all audio data are read normally using the audio interface. When enabled, which is the default option, Immacssip reads all audio data from files in the hard disk.

The **immacs_input_file_path** parameter indicates the path from where all audio files will be read. Currently only **wav** files are supported. The path should contain the absolute or relative (to where Immacssip is executed) path to the audio file including the file name, but without the "*.wav*" extension, *e.g. "/home/user/steve/input_example_"*. The **immacs_input_single_file** parameter indicates whether all the audio data are contained in one, multi-channel audio file (enabled), or in multiple single-channel audio files (disabled). When enabled (the default option), the "*.wav*" extension is automatically appended to the provided file name and then the file is read. When disabled, the audio data is expected to be on multiple files that share the same base name but have different endings. The ending is a positive integer starting from zero that indicates the audio channel, *e.g. "input_example_0.wav" for the first channel, "input_example_1.wav" for the second etc.* The name suffix and the "*.wav*" extension will be automatically appended to the base name before reading. The system will try to read from as many files as the microphone number. In multiple file mode, all files must have the same sample rate and the same number of samples, or else an error will occur. Finally, when the **immacs_input_file_repeat** option is enabled, the audio file is repeatedly re-read from the beginning as soon as reading reaches its end, entering a non-stop reading loop.

Another feature of Immacssip is input mapping. Input mapping is a simple mechanism that allows the user to specify which microphone of the array corresponds to which audio interface/file channel. The **mic_$MN_to_channel** parameter template is used for that option, where the **$MN** variable indicates the number of the microphone. For example, *mic_4_to_channel 2* indicates that the fourth microphone of the array is provided by the second channel of the interface/file. The input mapping is optional; if no input mapping for a microphone is provided, the microphone is supposed to be provided by the same channel number of the interface/file, *e.g. the third microphone will be provided by the third channel.* No input mappings are enabled by default.

Immacs Output

The **output** parameters concern immersive audio reproduction. The option **immacs_output_enabled** allows the user to enable or disable this function. When disabled, the default Baresip audio output method is used, using any available system output. ImmACS output is disabled by default.

The parameter **immacs_output_use_loudspeakers** controls whether the sound will be reproduced from loudspeakers (enabled) or headphones (disabled). Reproduction from headphones is the default option. When loudspeaker reproduction is enabled, the **immacs_output_loudspeaker_N** parameter sets the number of loudspeakers for reproduction. The loudspeakers are considered to be uniformly placed cyclically in equal distances from the listener.

ImmACS output supports simultaneous real-time reproduction from the headphones/loudspeakers and recording of the audio streams to the disk. To enable audio recording, use the **immacs_output_to_file** option, which is disabled by default. Each received audio stream, i.e. peer, is recorded in a separate file at disk. The parameter **immacs_output_file_path** defines the file base name for the output, *e.g. "/home/user/steve/immacs_out"*. When recording, the alias of each client is appended to the base name, *e.g. "/home/user/steve/immacs_out_john"*. The **immacs_output_single_file** parameter controls whether the audio stream will be recorded in one multi-channel file (enabled) or several one-channel files (disabled). When enabled, the "*.wav*" extension is appended to the file name before recording, *e.g. "/home/user/steve/immacs_out_john.wav"*. When disabled, the channel number is also appended to the each file, *e.g. "/home/user/steve/immacs_out_john0.wav", "home/user/steve/immacs_out_john1.wav etc."*

Similarly to the input mapping, ImmACS also supports output mapping. The output mapping is a simple mechanism to assign an ImmACS output to an audio interface/file channel. The template **out_$ONU_to_channel** is used, where **$ONU** is replaced by the output number. For example, *out_1_to_channel 5* sends the second ImmACS output to the fifth channel of the audio interface/file. Output mapping is optional; if a mapping for an output is not set, the output is by default mapped to the same output, *e.g. first output to first channel, second output to second channel etc.*

Immacs Advanced Options

ImmACS allows some more advanced options to be configured. However, any modification of these options is only suggested for users very familiar to the ImmACS theoretical background. The **srate** parameter defines the sample rate of audio input and output. In this Immacssip version only **44100** and **48000** Hz is supported. The 48000 Hz sample rate is the default value. The **fft_size** parameter sets the number of the frequency coefficients used for ImmACS encoding and decoding. This parameter does not affect DOA estimation, as it uses a static number coefficients. Setting this parameter to zero (default) forces the system to automatically re-set it to twice the size of the audio period. When setting a custom value, it must not be less than twice the size of the audio period. The last option

gui_comm_port sets the communication port for the ImmACS GUI, which by default is set to 7484. Special care must be taken for the fact that although only one port is set by the user, the port after that is also used, which in the default case is 7485.

Jitter Buffer

The jitter buffer option **jitter_buffer_delay** is provided by the default Baresip configuration but it is worthy to mention its influence on Immacssip. The jitter buffer compensates for the cases that packets are received in a different order than that they where sent. This option controls the jitter buffer minimum and maximum values, in packets. The minimum value of this buffer is a direct overhead to the overall system latency. It defines how much "packet-time" later a packet will be reproduced after it is received. Reducing the minimum size decreases the latency but increases the risk of audio glitches as the system becomes more vulnerable to sequence inconsistencies. The jitter buffer can be completely disabled by setting the minimum buffer size to zero. Its default values are five packets for the minimum and ten packets for the maximum.

7.3.5 Operation

To run Immacssip, type

```
$ immacssip
```

Immacssip initialization is very similar to Baresip. The same information messages are expected to be displayed. Pressing "**h**" will show the **help** menu. Dial to a conference room by pressing "**d**" and the room name, *e.g. room_A*. Information about the call bitrate will be printed for the current toggled account. The format will be similar to that of the Bareserver.

[**elapsed time**] audio=**autx**/**aurx** video=**vitx**/**virx** (bit/s) efps(tx)=**txfps**

where **autx**/**aurx vitx**/**virx** is the audio transmit/receive, video transmit/receive bitrate to/from the Bareserver respectively, and **txfps** is the transmit video frames per second when video is enabled. For example,

[0:15:13] audio=50767/104350 video=273082/641034 (bit/s) efps(tx)=12

7.3.6 Operation with video support

Immacssip also supports video streams. To start Immacssip with video support, type

```
$ immacssip -v
```

In video mode, the client is able to send and receive video streams. The video streams of the peers are displayed all simultaneously, using one window per peer. Self-view is disabled by default, but it can be enabled by uncommenting the **self-view** module in the "*config*" file. Clients that operate in video mode can co-exist in the same room with clients that have video mode disabled, however the latter will not be able to receive or send any video stream.

In the case where a client operates in video mode but the system does not have a local camera, the client will be able to see the others but will not send any video information. Video support is still in experimental state and can cause the system to lag when running on low bandwidth connections. Video support should stay disabled for greater system stability.

7.3.7 Uninstall

Immacssip can be uninstalled by typing

```
$ cd immacssip
$ sudo make uninstall
```

To completely remove all the files of Immacssip, the "*.immacssip*" directory and its contents must be removed manually by the user. However, removing those is not recommended if a future re-install is expected. Note that when uninstalling Immacssip, all its dependencies still remain on the system.

7.4 Immacs Control

Immacs Control is a graphical user interface for Immacssip that allows the user to manipulate the transmitted and received audio streams.

7.4.1 Dependencies And Installation

Immacs Control is written in Java, so it requires the Java run-time environment. It has been tested and working using the java version **1.8**, which can be downloaded from **https://java.com/en/download/**.

Java Control executable consists of only one file and installation is as simple as manually copying the file at the desired location in the computer. This can be done by typing

```
$ cd immacs_control
$ cp bin/immacsctl.jar dest_path
```

where "*dest_path*" is the desired destination path for installation.

7.4.2 Operation

Immacs Control can be executed by typing

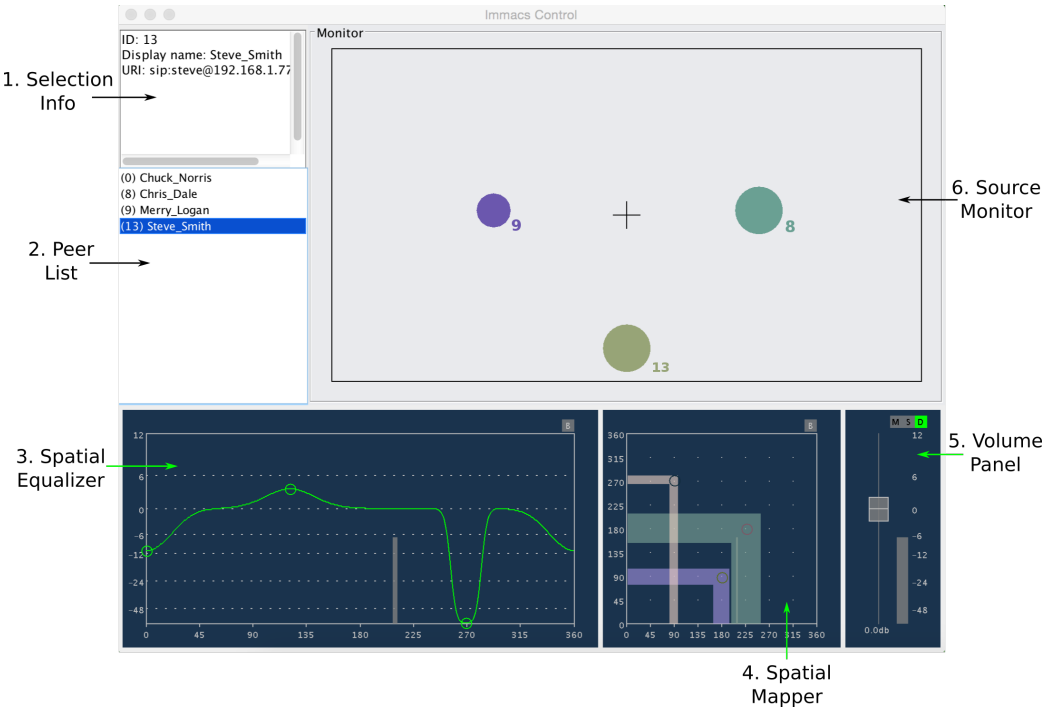

Figure 7.1: Immacs Control screenshot.

```
$ java -jar install_path/immacsctl.jar
```

where "*install_path*" is the path where Immacs Control has been installed. By default Immacs Control uses the port 7484 to communicate with Immacssip. However, a different port can be set as a command line argument. For example,

```
$ java -jar install_path/immacsctl.jar 14500
```

will start Immacs Control listening at port 14500.

As already mentioned, GUI communication utilizes two concecutive ports, with the first choosen by the user. An Immacs Control instance is expected to operate in conjunction with only one Immacssip instance. If multiple Immacssip or Immacs Control instances operate simultaneously on the same system, the user must ensure that they use different GUI ports. Otherwise, the system will have unexpected behaviour and may crash.

When an existing Immacs Control instance quits, its settings remain on Immacssip. When a new Immacs Control instance starts, the effects are automatically reset to the default values.

Immacs Control consists of six parts which are shown at the screenshot in Figure 7.1.

1. Selection Info Panel
2. Peer List
3. Spatial Equalizer
4. Spatial Mapper
5. Volume Panels
6. Source Monitor

The user selects a connected peer from the **Peer List**. He can manipulate the selected peer's audio stream by using the **Spatial Equalizer**, the **Spatial Mapper** and the **Volume Panel**. Each peer has its own settings. The DOAs of all peers can be displayed simultaneously at the **Source Monitor**. Each of the six parts is more thoroughly explained below.

Selection Info Panel

This panel displays detailed information about the selected peer in the Peer List. It displays the ID, the display name (alias) and the URI of the peer.

Peer List

The list of the remote peers. For each peer, its ID is printed in parentheses followed by its alias or URI, depending on which is available. The end-user can select only one entry at a time.

A special case has been added to allow the user to manipulate the volume of its own audio stream. The first entry of the Peer List is always the user's local account. The ID of the local account is always set to zero, while a remote peer cannot have a zero ID. By selecting the first entry the user can manipulate its own volume or choose to display its own spatial information.

Spatial Equalizer

The Spatial Equalizer is a tool that helps the user control the **volume per angle** of an ImmACS audio stream. It affects only the volume of the sources in space and not their position. It is very similar to a frequency equalizer with the difference that the x-axis indicates the angle and not the frequency. An example of use can be seen in Figure 7.2.

The Spatial Equalizer has no effect on the received stream by default. The user can add **nodes**, that he may use to alter the response curve of the equalizer.

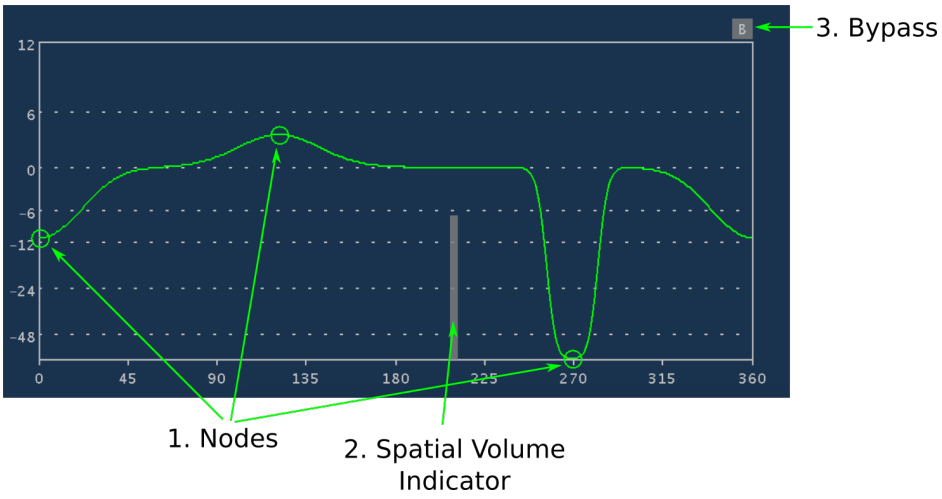

Figure 7.2: Screenshot of the Spatial Equalizer.

Each node has a position in the x-y space. The "**x**" coordinate indicates the angle (degrees), while the "**y**" indicates the magnitude (decibel). A spatial equalizer node also has a range (degrees) on the response curve. The system allows the user to insert up to 16 nodes to achieve the desired angular response curve.

An existing node can be set in **solo** mode by double left-clicking. When at least one node is set solo, the floor of the response curve drops to about -115 db, to practically mute all the other nodes. Multiple solo nodes may exist. An example of the Spatial Equalizer in solo mode is given in Figure 7.3.

The **bypass** button disables the effect without changing the effects settings. It can be intuitively perceived as a simple "off" button.

To help the user adjust the equalizer better, the angular spectrogram of the audio stream is displayed in the background. The angular spectrogram displays the signal's **energy per angle**. It is visible in the background with a light gray color in Figure 7.2.

The y-axis (magnitude) provides a range from about -115 to 12 db. Notice that the scale of the y-axis is not linear. More space is given from -12 to 12 db to provide a higher resolution for small changes in the amplitude. Below -12 db the scale converges very fast to -115 db, so as to provide a mute-like functionality. The x-axis uses a linear scale and works in a wrap-around fashion, due to the fact the x-axis expresses angle.

The Spatial Equalizer is meaningful to be used only with audio streams that carry several sources in space, in other words *with peers that utilize a microphone array and support the ImmACS Input function.* It will be disabled for peers that do not transmit immersive audio. It is also disabled for the local host.

The Spatial Equalizer can be easily manipulated by using the mouse. More specifically

Figure 7.3: Spatial Equalizer screenshot in solo mode.

- Left click adds a new node to the point where the cursor is.
- Moving the mouse over an existing node will display the node information at the top left corner of the panel.
- Right click removes the node where the cursor is.
- Double left click enables/disables solo mode at the node where the cursor is.
- Existing nodes can be dragged anywhere using the mouse.
- Moving the mouse wheel up/down will change the range of the node where the cursor is.

Spatial Mapper

The Spatial Mapper is a tool that allows the user to alter **the position** of the sources of an immersive audio stream. The Spatial Mapper does not affect the volume. A more detailed example can be seen in Figure 7.4.

The Spatial Mapper initially has no influence on the audio stream. The user can add **nodes** that are used to change the position of the sources. A node is characterized by its position in space (the **x-y** coordinates) and its **range**, all measured in degrees. When a source lies into the range of a node on the x-axis, it is automatically changed (mapped) to corresponding angle to the position of the node on the y-axis. Both x and y axis work in a wrap-around way. Up to 16 nodes can be added.

When two or more nodes share the same part on the x-axis, it is unclear for the system which node to use for mapping. Overlapping is allowed in the x-axis, but in that case the node that was created first in time is used for mapping. The ranges are displayed with random, moderately transparent colors, to provide a better optical descrimination.

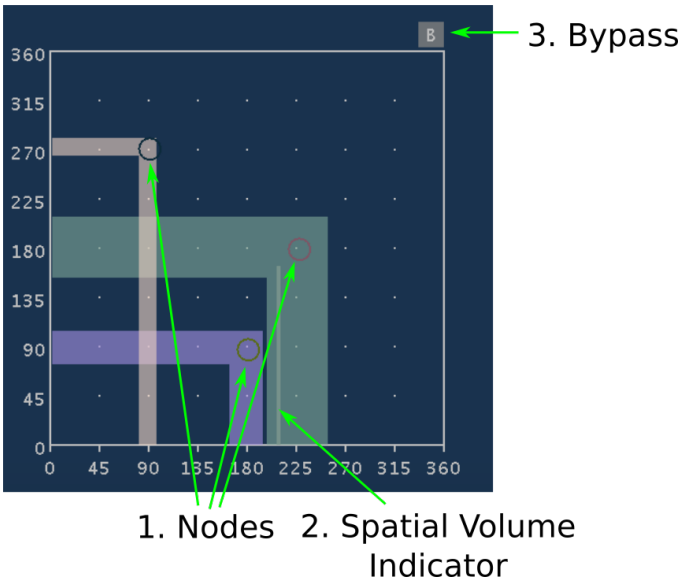

Figure 7.4: Screenshot of the Spatial Mapper.

Similarly to the Spatial Equalizer, the angular spectrogram of the audio stream is displayed in the background with an light gray color. A **bypass** button also exists, that works as a simple "on/off" switch for the effect.

The Spatial Mapper can be used with both immersive and normal audio streams. For streams that do not contain spatial information, the source is automatically placed at 180°, allowing the user to change the position using the Spatial Mapper if he wishes to. However, the Spatial Mapper can not be used with the local stream.

The Spatial Mapper can be easily handled by the mouse. Specifically

- Left click adds a new node to the point where the cursor is.
- Moving the mouse over an existing node will display the node information at the top left corner of the panel.
- Right click removes the node where the cursor is.
- Existing nodes can be dragged anywhere using the mouse.
- Moving the mouse wheel up/down will change the range of the node where the cursor is.

Volume Panel

The Volume Panel allows the user to monitor and control the overall volume of an audio stream. A simple example of use is given in Figure 7.5. The Volume Panel can be used with any kind of audio stream, either received or local.

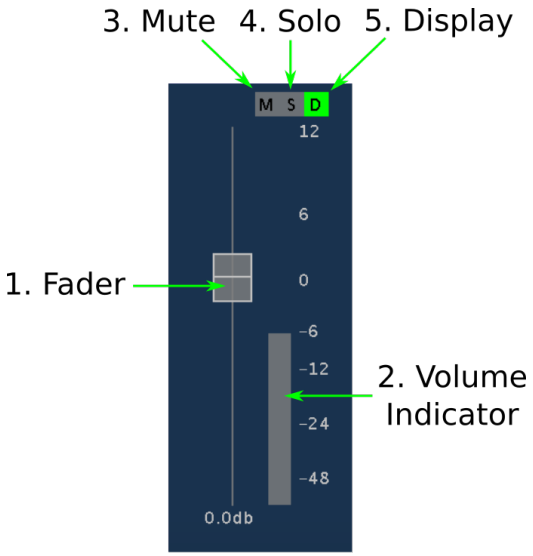

Figure 7.5: Screenshot of the Volume Panel.

The fader controls the overall stream volume. It can be easily manipulated by using the mouse, either by dragging or by moving the mouse wheel when the mouse lies over the fader. The overall volume is displayed by the volume indicator. It is worthy to mention that the scale of the volume, in decibel, is not linear. It is the same with the scale of the Spatial Equalizer; high analysis is provided from -12 to 12 db and more raw analysis is provided from -115 to -12 db.

The "**M**" (Mute) button silences the audio stream, while the "**S**" (Solo) button will force the system to reproduce only that stream. Multiple muted or solo streams are allowed. Mute has a greater priority than solo, meaning that when both buttons are enabled for a stream the mute button will prevail. When at least one channel is at solo mode, an indication is displayed in the top left corner of the Source Monitor. Both buttons are disabled by default.

The "**D**" (Display) button enables/disables the monitoring function for an audio stream. When enabled, the stream will be displayed at the Source Monitor. By default, it is enabled for the received audio streams and disabled for the local audio stream.

Source Monitor

The Source Monitor provides visual feedback to the user, regarding the position of the received and local audio sources. The reference point is set at the center, with 0°lying in the front, the 90°directly to the right etc. A source can be selected to be displayed using the "**D**" (Display) button of the Volume Panel. Each source is displayed as a circle whose radius is proportional to the source power. The ID of the peer is also displayed at the bottom right corner of each source. Each peer is

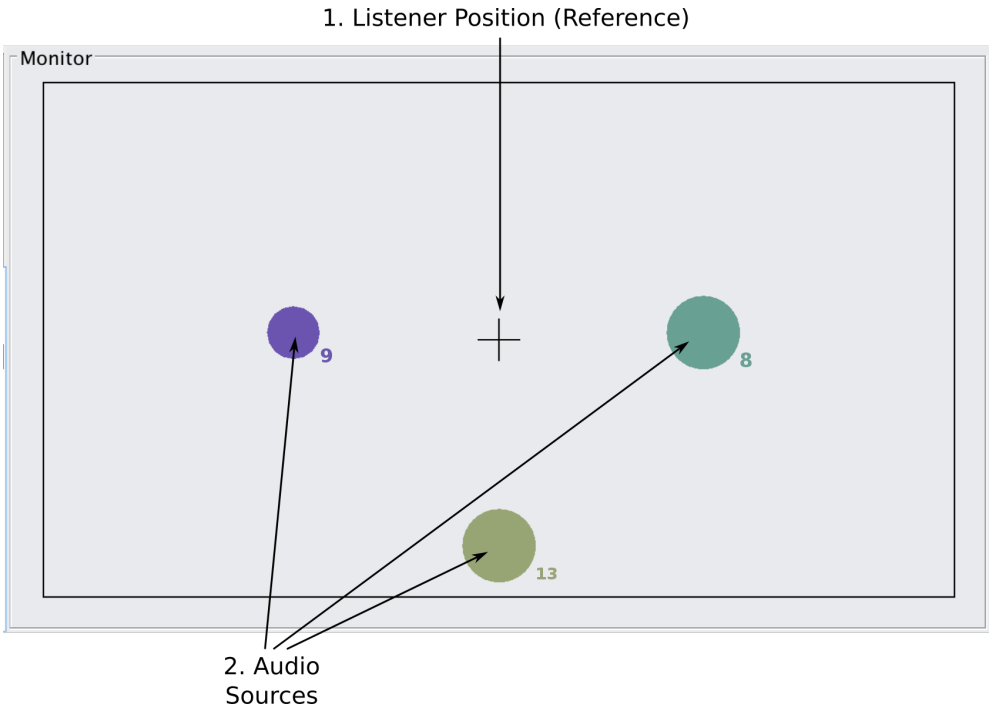

Figure 7.6: Screenshot of the Source Monitor.

displayed using a random, moderatelly transparent color, to help the user discriminate the sources more easily.

Discussion: effects pipeline

The three above effects (Spatial Mapper, Volume Panel, Source Monitor) allow the user to alter the spatial and volume characteristics of the audio streams. Their order in audio processing is the same with their visual precedence from left to the right; Spatial Equalizer -> Spatial Mapper -> Volume Panel. The Spatial Equalizer and the Volume Panel affect only the volume and the Spatial Mapper affects only the direction of the carried sources.

For the ease of use, the angular spectrograms in Spatial Equalizer and Spatial Mapper are displayed before any of the three effects apply. This means that the displaying of the volume will not be affected in the Spatial Equalizer or the Volume Panel and the displaying of the angle will not be affected in the Spatial Mapper. On the other hand, the volume bar on the Volume Panel is displayed after all three effects have been applied, but it is not affected by the mute button.

The sources displayed at the Source Monitor are only affected by the Spatial Mapper. This means that any change of the source position will be also displayed on the Source Monitor. Changes of the volume in Spatial Equalizer and Volume Panel will not affect the Source Monitor.

Chapter 8

Conclusions and Future Work

In this thesis, we designed and implemented a communication system that supports immersive audio capturing and reproduction. Our system supports the communication of multiple users through a conference server. We used a flexible and robust technique for spatial audio reproduction, capturing and encoding that is suitable for real-time applications [1]. We realized the functions of that technique in **Libimmacs**, a library we designed for optimal real-time performance and flexibility. We then incorporated Libimmacs to Baresip, an existing open-source VoIP client, so as to create **Immacssip**, the client of our system. We also created a custom conference server, the **Bareserver**, which is based on Baresip and performs a transparent relay of the packets that involves no transcoding or down mix. Each client receives the streams of all the clients, so that it is able to manipulate each stream separately. To facilitate the control of the streams, we created **Immacs Control**, a graphical user interface in Java, that allows for maximum flexibility on the manipulation of the spatial audio. Our system is most suitable for teleconference applications but it could also be used for music performances through the network. To our knowledge, this is the first implemented system that allows for real-time bidirectional immersive audio communication.

Considering the use of Libimmacs in future projects, the major challenge of the design was to compensate between abstraction and flexibility, within the strict framework of real-time audio programming. We had to consider that many useful programming techniques, that are commonly used in general purpose applications, are not safe for real-time. For this reason we used alternative methods that are reliable for real-time, but these increased the complexity of the design and made the implementation of the system harder. Eventually, we managed to meet our initial goals by creating a simple-to-use yet powerful library for real-time spatial audio capturing, encoding and reproduction.

For the implementation of Immacssip we came across two big challenges. The first was to incorporate Libimmacs into Baresip and transmit/receive the spatial meta-data along with the audio data, and the second was to modify Baresip so that it receives multiple streams per call. The main challenge, however, was to

study, analyze and understand the code of Baresip sufficiently from scratch, so as to design and implement the required changes to achieve our goals. The same difficulty applies also for Bareserver, where we converted Baresip from a VoIP client to a VoIP server suitable for our system. Finally, the challenge with Immacs Control was to create a flexible, yet intuitive to use graphical interface, so as to exploit all the possibilities of Libimmacs.

As a future work, we intent to add echo cancellation functionality to the system, so as to prevent the leakage of the loudspeaker output to the microphones. Overall, this thesis provided us with significant experience on understanding, designing and implementing real-time audio applications and professional applications in general.

Bibliography

[1] A. Alexandridis, A. Griffin, and A. Mouchtaris, "Capturing and reproducing spatial audio based on a circular microphone array," *Journal of Electrical and Computer Engineering*, vol. 2013, p. 16, 2013.

[2] V. Pulkki, "Spatial Sound Reproduction with Directional Audio Coding," *J. Audio Eng. Soc*, vol. 55, no. 6, pp. 503–516, 2007. [Online]. Available: http://www.aes.org/e-lib/browse.cfm?elib=14170

[3] M. Cobos, J. Lopez, and S. Spors, "A sparsity-based approach to 3d binaural sound synthesis using time-frequency array processing," *EURASIP Journal on Advances in Signal Processing*, vol. 2010, no. 1, p. 415840, 2010. [Online]. Available: http://asp.eurasipjournals.com/content/2010/1/415840

[4] K. Kowalczyk, O. Thiergart, M. Taseska, G. Del Galdo, V. Pulkki, and E. Habets, "Parametric spatial sound processing: A flexible and efficient solution to sound scene acquisition, modification, and reproduction," *Signal Processing Magazine, IEEE*, vol. 32, no. 2, pp. 31–42, March 2015.

[5] V. Pulkki, "Virtual Sound Source Positioning Using Vector Base Amplitude Panning," p. 466, 1997.

[6] "BareSIP," http://www.creytiv.com/baresip.html, accessed: 2015-09-08.

[7] J. Tranter, "Introduction to sound programming with alsa," *Linux J.*, vol. 2004, no. 126, pp. 4–, Oct. 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1017973.1017977

[8] D. Phillips, "A user's guide to alsa," *Linux J.*, vol. 2005, no. 136, pp. 3–, Aug. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1080072.1080075

[9] C. Adamson and K. Avila, *Learning Core Audio: A Hands-On Guide to Audio Programming for Mac and iOS*, 1st ed. Addison-Wesley Professional, 2012.

[10] "Windows Audio Session API," https://msdn.microsoft.com/en-us/library/windows/desktop/dd371455(v=vs.85).aspx, accessed: 2015-06-18.

[11] R. Bencina and P. Burk, "Portaudio–an open source cross platform audio api," *Proceedings of the International Computer Music Conference, Havana*, pp. 263–266, 2001.

[12] G. P. Scavone, "Rtaudio: A cross-platform c++ class for realtime audio input/output," in *in Proceedings of the 2002 International Computer Music (ICMC'02*, 2002, pp. 196–199.

[13] "JACK Audio Connection Kit," http://jackaudio.org/, accessed: 2015-06-14.

[14] "PulseAudio," http://www.freedesktop.org/wiki/Software/PulseAudio/, accessed: 2015-06-14.

[15] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. San Diego, CA, USA: California Technical Publishing, 1997.

[16] R. Boulanger, Ed., *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming*. Cambridge, MA, USA: MIT Press, 2000.

[17] C. Chafe, M. Gurevich, G. Leslie, and S. Tyan, "Effect of time delay on ensemble accuracy," in *In Proceedings of the International Symposium on Musical Acoustics*, 2004.

[18] "Real-Time Audio Programming 101: Time waits for nothing," http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothing, accessed: 2015-06-24.

[19] "CMake," http://www.cmake.org/, accessed: 2015-09-10.

[20] "TinyCThread," https://tinycthread.github.io/, accessed: 2015-09-10.

[21] "pstdint.h," http://www.azillionmonkeys.com/qed/pstdint.h, accessed: 2015-09-10.

[22] "FFTW," http://www.fftw.org/, accessed: 2015-09-10.

[23] "Libsndfile," http://www.mega-nerd.com/libsndfile/, accessed: 2015-09-10.

[24] "ITPP," http://itpp.sourceforge.net/4.3.1/, accessed: 2015-09-10.

[25] D. Pavlidi, A. Griffin, M. Puigt, and A. Mouchtaris, "Real-time multiple sound source localization and counting using a circular microphone array," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 21, no. 10, pp. 2193–2206, Oct 2013.

[26] B. Gardner and K. Martin, "Hrtf measurements of a kemar dummy-head microphone," MIT Media Lab Perceptual Computing, Tech. Rep., 1994.