Bachelor's Thesis
Embedded Systems
2018

Félix Laguna Teno

REAL TIME DSP FOR IMMERSIVE SPEECH COMMUNICATION

TURKU AMK
TURKU UNIVERSITY OF
APPLIED SCIENCES

BACHELOR'S THESIS — ABSTRACT
TURKU UNIVERSITY OF APPLIED SCIENCES
Embedded Systems
2018 | 23 pages , 5 Appendix pages

Félix Laguna Teno

REAL TIME DSP FOR IMMERSIVE SPEECH COMMUNICATION

The field of real time digital signal processing using software has attracted attention but there is not actually a solution to apply acoustic effects to a stream of audio in real time.

The purpose of the thesis was to build a prototype of immersive communication system and set foundations for the readers to build their own ones.

The different parts of the prototype were researched, implemented and united into the final system. Several audio back end technologies, encoders and network implementations were analyzed and the best ones (PortAudio, Opus and raw sockets) were chosen for building the prototype.

The prototype developed in this thesis is working on a local environment, while working unreliably in networked conditions. The design decisions, diagrams, and research are included in this thesis, setting the required foundations. The prototype could be further improved with reliable networking and a graphical user interface.

The prototype serves as a milestone in the development of similar products and this thesis can be used by computer scientists to develop immersive systems.

KEYWORDS

Digital signal processing, real time, immersion, communication.

CONTENTS

APPENDICES

FIGURES

List of Tables

LIST OF ABBREVIATIONS (OR) SYMBOLS

API	Application Programming Interface
DSP	Digital Signal Processing
FIR	Finite Impulse Response
FPGA	Field-programmable gate array
HRTF	Head-related transfer function
IIR	Infinite Impulse Response
LTI	Linear Time Invariant
POSIX	Portable Operating System Interface (Unix)
RAM	Random-Access Memory
SIMD	Single Instruction Multiple Data
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

GLOSSARY

Application Programming Interface	Set of operations and procedures that a library offers to other software for the library to be used properly.
Back end	layer of a program which handles the logic of the system. For example, audio back end would be the part of the system which directly interacts with the audio hardware and provides data to other modules.
Binaural	Category of sound which feels three-dimensional for the listener, even if the sound comes from a non three-dimensional space, such as a pair of headphones.
Codec	Hardware or software piece in charge to execute the encoding and decoding process of a stream of data. This process converts the data from one format or code to other, usually to save space or to improve performance.
Digital Signal Processing	Mathematical manipulation of a signal to improve it or change it in any way.
Field-programmable gate array	Device whose logic and connections can be customized after being manufactured.
Head-related transfer function	Mathematical function which models the way the sound is received through the ears, including the bone resonance and the delay between ears, for example.
Immersion	The property of a system which aims to trick the user into perceiving a virtual environment as real using stimuli. In this project's context, the concept will be used to refer to ”sound immersion”.
Portable Operating System Interface	Standards and practices created by R. Stallman and later adopted by IEEE for interoperability between operating systems. The X comes from the Unix operative system, which was used as the foundation of the standard.
Real Time system	System which performs its operations with time constraints, so every operation must finish in a certain time in order for the system to be successful.
Single Instruction Multiple data	Computer architecture in which the same instruction is parallelized to multiple streams of data, allowing for example to apply the same level of saturation to all pixels in an image.

1 INTRODUCTION

There is no software solution for a communication system that can achieve true effects of immersion. The purpose of immersion is to trick the brain of the user into perceiving a virtual environment as real using stimuli. This thesis will give enough background to a computer scientist or software developer to have a starting point in developing such systems, as well as providing a prototype of a working one. Diagrams and design choices will be discussed as well, providing informed opinions about the choices done, as well as the research data.

Immersion tricks the brain of the user into believing they are in different environment instead of the one they are. The system aimed to be created in this thesis should be able to apply room acoustics and other effects (such as binaural using the HRTF function) to the sound produced in real time by the user, in other to trick the other interlocutor into perceiving them as they are in the same space.

There are software programs which can apply arbitrary effects to any sound file, but few achieve this task for real time audio. The theoretical background about the different parts of the prototype (real time programming, general purpose DSP and network communication) is abundant, but there is no documentation about trying to merge those parts together, hence the purpose of this thesis.

The goal of the thesis is to build a system which allows real time communication between interlocutors in real time, making them believe that they are in the same space as well as applying other effects such as room acoustics or binaural sound to create the feeling of immersion. This thesis aims to create functional requisites, research and choose the best back ends for every part of the system, create the intermediate structures to unite the different components of the system and create a working prototype of the described program. For that purpose, diagrams describing the system will be designed additionally.

The scope of the thesis has to be narrowed in order to create the above described system. The requisites will impose tighter constraints than the ones described here, but the system should enable two users to communicate in real time giving the impression of immersion. The thesis is intended to guide any software designer in the correct direction in case they want to build their own system.

The structure of the thesis is as follows: firstly, in the theoretical background, the necessary concepts in DSP, mathematical, real time and networking are discussed, subsequently the methodology used and the results achieved will be detailed, and the results are critically discussed.

2 THEORETICAL BACKGROUND

This section explains the theoretical background needed to fully understand later the Methodology is explained. The topics include Digital Signal Processing (DSP), mathematical theory of signals, immersive technology, real time signals, and network communications. The previous research is discussed, and the questions this thesis tries to answer will rise from it.

2.1 DSP

Digital Signal Processing (DSP) is the manipulation of a signal carrying information, in order to modify its properties. The nature of these modifications can range from amplification to the removal of certain frequencies. The field itself is vast enough that during the history of computer science, the area of DSP was created along with the first computers, and the technology, because of the expensive calculations that were needed to be performed, was only used in very critical areas, such as radar, space exploration (Smith, 1997) and others (*Figure 1*). Only later, as computers exponentially became faster and more complex, DSP was used in commercial applications, with two different approaches, using the general-purpose chips or using specialized ones.

DSP processors were for some time the most common method for processing audio. They are implemented in the SIMD (Single Instruction Multiple Data) architecture which is the same approach that gives the Graphics Processing Unit (GPU) the power needed to process graphics. This architecture allows the chip to optimize the common operations of the mathematical functions needed in the algorithms. The other implementation is using FPGA, which allows customizable high-performance chips to be created. These are faster to prototype and to produce, but their range of operations is vastly inferior than the DSP processor. The FPGA is usually faster in certain algorithms commonly used in DSP, such as FIR filters. Examples of these devices can be seen anywhere, from the pure-electronics guitar pedal, to the complex processing machine of a synthesizer.

On the other hand, thanks to the exponential increment in processing speeds of general-purpose CPUs, the algorithms used could be implemented by software, and thus they could run in these CPUs. This approach was relatively new compared to the previous one, so more optimizations can be done in this field. Programs like Audacity (Audacity, 2018) or Adobe Audition (Audition, 2018) evolved from simple DSP programs to full pledged audio edit programs. Another low-level approach to DSP software would be MatLab (MatLab, 2018) and the Simulink module. Both products perform efficient DSP, but they need to be programmed by experts with enough theoretical background. These products still need a considerable amount of processing power to work, so they usually do noy deal properly with real time streams of audio.

The two approaches (hardware and software based) are very different, neither one is better in every aspect, but each has its own advantages and disadvantages. The former has the advantage of being orders of magnitude faster than its software counterparts, both the DSP processor and the FPGA implementations. However, creating specialized hardware is much more expensive than using general purpose chips, and the software that will use the hardware has to be coded in order to use the full capabilities of it. In addition to that, they are very optimized for some specific algorithms, so if the program uses different ones, the performance hit will be considerable. Also, although hardware can be open source, most of developers lack the facilities to build their own hardware, having to rely in proprietary, already-made chips.

Conversely, software DSP implementations are usually slower than their hardware counterparts, but they have other advantages. First of all, they are highly customizable because they have a range of audio modifications vastly superior than the hardware approach. They can also be open source, like Audacity (Audacity, 2018), so any company can build their own version of it, and

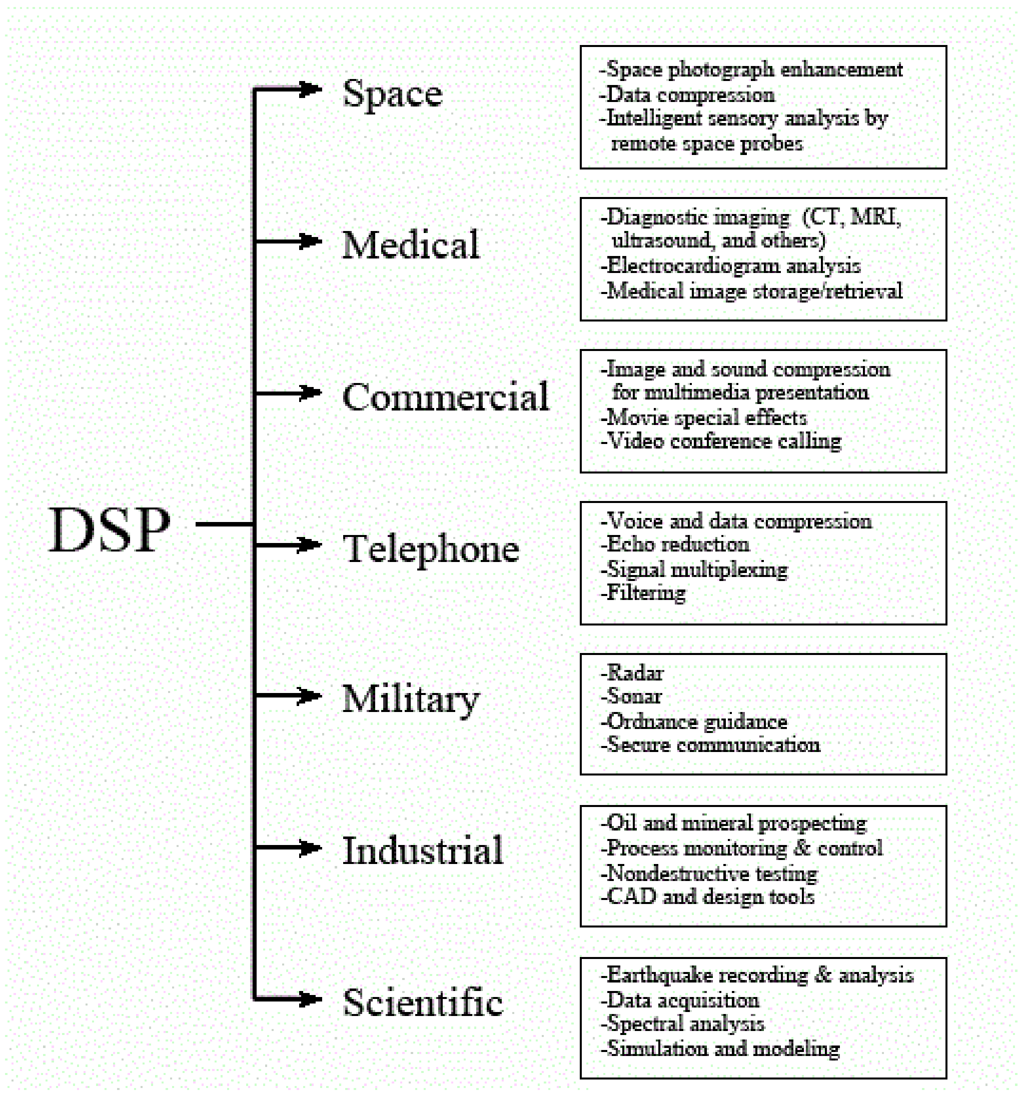
DSP
Space
-Space photograph enhancement
-Data compression
-Intelligent sensory analysis by remote space probes
Medical
-Diagnostic imaging (CT, MRI, ultrasound, and others)
-Electrocardiogram analysis
-Medical image storage/retrieval
Commercial
-Image and sound compression for multimedia presentation
-Movie special effects
-Video conference calling
Telephone
-Voice and data compression
-Echo reduction
-Signal multiplexing
-Filtering
Military
-Radar
-Sonar
-Ordnance guidance
-Secure communication
Industrial
-Oil and mineral prospecting
-Process monitoring & control
-Nondestructive testing
-CAD and design tools
Scientific
-Earthquake recording & analysis
-Data acquisition
-Spectral analysis
-Simulation and modeling

Figure 1. DSP applications (Smith, 1997).

even modify the code to fit their purposes. Lastly, the cycle of development of DSP software is orders of magnitude faster than its hardware correspondent. The former can usually follow fast development cycles like Scrum, whereas the later will usually follow system similar to the V-Model (Christie, 2008), which follows a slower and more robust process in order to minimize the chances to create faulty hardware.

The author did not have yet the technical background required to create a DSP software, so there was the need for theoretical foundation to proceed with the research. The book by (Kuo et al., 2013) introduces the basic knowledge needed for creating DSP software, specially the mathematical theory needed for it. After the foundation has been set, further research could be started. On contrast, Kuo assumes a very high mathematical background which most computer scientists do not have, so further research was needed usually. All in all, Kuo's book manages to teach the basics of DSP to any computer scientist, providing they have the appropriate mathematical background.

2.2 Mathematical background

In this section, the mathematical background needed for understanding and using the thesis contents will be discussed. A significant portion of these contents were new for the thesis author, so that is the reason of this section, as the thesis aims to be as self-contained as possible.

To begin with, the following definitions are needed in order to understand later concepts which will be built on top of them.

> " A digital signal is a sequence of numbers $x(n), -\infty < n < \infty$ where the integer *n* is the time index."
>
> ((Kuo et al., 2013) Chapter 2)

A digital signal is created by a computer using a process called quantization, which consists in sampling an analog or continuous signal at a fixed rate, as can be seen in *Figure 2*.

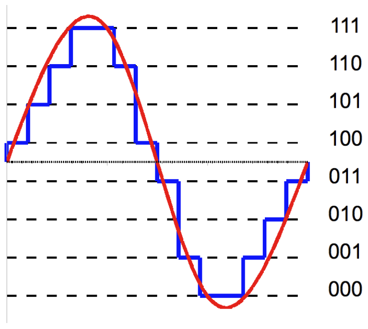

Figure 2. Quantization visualized. (Franz, 2008, p.38-9).

After defining digital signal, the unit-impulse response signal can be defined as the following formula:

$$\delta(n) = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases} \tag{1}$$

The unit-impulse signal is particularly important because all the systems that will be discussed later are defined using it. A good analogy to help the reader understand it is that the unit-impulse signal is the mathematical equivalent of shooting a gun in an free field, a very sharp sound and later silence.

DSP systems can be abstracted into the composition of 3 main "blocks", addition of signals (*Figure 3*), multiplications by an scalar (*Figure 4*), and delay or time shift (*Figure 5*). With these 3 operations every system can be defined.

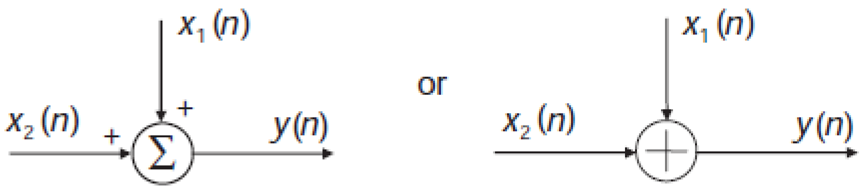

Figure 3. Addition block (Kuo et al., 2013, Chapter 2).

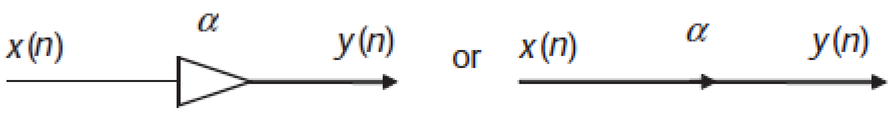

Figure 4. Multiplication block (Kuo et al., 2013, Chapter 2).

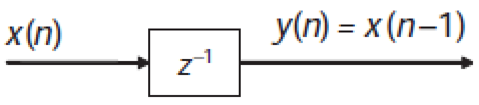

Figure 5. Delay block (Kuo et al., 2013, Chapter 2).

The focus will be on systems that follow the Linear Time Invariant (LTI). A system is linear if the input and the output are mapped linearly, or in other words, if the input is scaled and/or summed, the output is also scaled and/or summed.

$$a_1x_1(n) + a_2x_2(n) \rightarrow a_1y_1(n) + a_2y_2(n)$$

A time invariant system is one whose characteristics does not change with time, so a time shift in the input just changes a time shift in the output.

$$if\ x(n) \rightarrow y(n)\ then\ x(n-k) \rightarrow y(n-k)$$

An important operation is the convolution of two functions, which is defined by the multiplication of the values of the first function in the usual order with the values of the second one in inverted order.

i.e $x_0z_n + x_1z_{\text{n-1}} + ...$. The general formula which will be used is the following, assuming the system is causal, which means that $z(n) = 0, n < 0$. Every real-time system is casual.

$$x(n) * z(n) = \sum_{k=0}^{n} x(k)z(n-k)$$

Any system is also defined by its impulse response signal $h(n)$, which is the series of values generated after applying the unit-impulse signal to a system. In general $h(n) = y(n)\ if\ x(x) =$

$\delta(n)$. The impulse response allows to characterize a system whose behavior could not be easily, or not at all translated into mathematical formulas. In the real world, it is the series of reflections back and forth from the walls after generating a transient (sharp) sound. Following the analogy of the gun previously presented, the impluse response would be the series of echoes heard after shooting the gun in an empty room.

The impulse response has a determined number of values or ”magnitude”, which is correlated with the amount of computer power required to process it. To apply the impulse response to a signal the following formula has to be applied:

$$y(n) = x(n) * h(n) \; or \; y(n) = h(n) * x(n)$$

Using the definition of impulse response the filtering process can be discussed. The process consists on applying an effect or filter to an audio signal. The system used depends on the effect desired, but it can be generalized by the previous idea that every system can be described by its impulse response $h(n)$. However, there are signals which output depends also on the past output signal, not only on the input. Therefore, the definition can be expanded to the following:

$$y(n) = \sum_{l=0}^{L-1} b_l x(n-l) - \sum_{m=1}^{M} a_m y(n-m) \rightarrow \; y(n) = h(n) * x(n) - g(n-1) * y(n-1)$$

The b_n coefficients are the $h(n)$ part of the impulse response and the a_n coefficients are the $g(n)$ part of the impulse response. The a coefficients are subtracted from the final result of the $h(n)$ convolution. The sub-indexes n and m usually are the same in most systems, but that is not compulsory.

This is the foundation of the filtering operation which includes finding the coefficients a_m and b_n and applying the convolution operation. If the a_m coefficients are 0, that means the system is a Finite Impulse Response (FIR) filter. These filters only are modified by the previous inputs, and the outputs have no effect in them. That allows the system to be implemented using usually half of the memory and one third of the operations of a full filter, which receives the name of Infinite Impulse Response (IIR) filter. These two systems have different properties which escape the scope of this thesis, but generally, the IIR filters are more versatile and powerful, at the cost of more computational power and memory. The following diagrams extracted from (Kuo et al., 2013) represent the systems with the three blocks discussed previously:

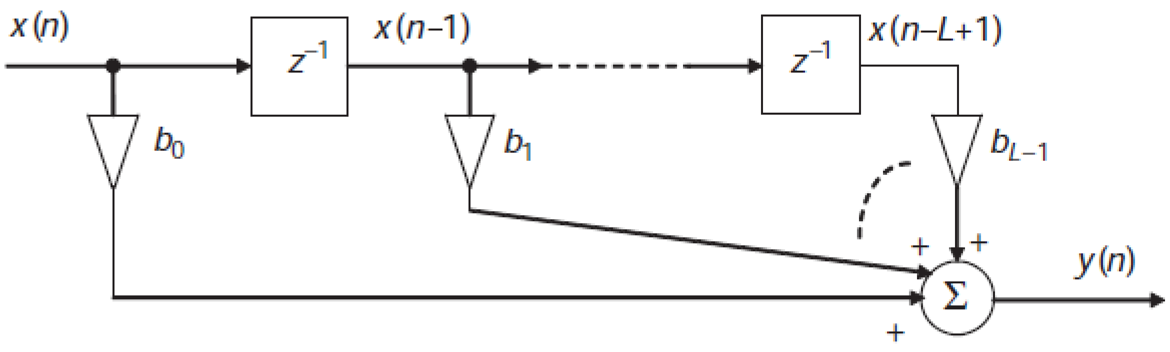

Figure 6. FIR filter (Kuo et al., 2013, Chapter 2).

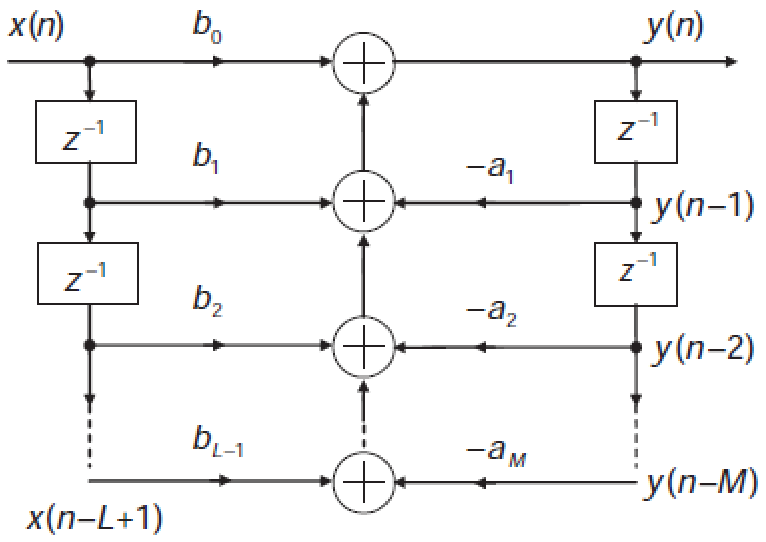

Figure 7. IIR filter (Kuo et al., 2013, Chapter 2).

After discussing how to obtain an impulse response from a real environment, the question of how to obtain it from a mathematical calculation arises. This topic is outside of the scope of the thesis, but if the reader is interested, deep knowledge about Fourier's, Laplace's and Z's transform is required. MathLab provides functions such as $fir1$ to simplify this task.

2.3 Immersive Technology

The aim of immersive technology is to allow the user to perceive a digital environment as real. In order to succeed, the senses must be tricked so they believe that the stimuli perceived correspond to ones coming from the real world. This thesis focuses on audio processing, so only the ideas regarding audio immersion will be discussed. Three main ideas need to be applied to convert audio coming from the computer to stimuli immersive enough it can deceive the brain to hear the sound as a real-world one.

- The first concept is the Head-Related Transfer Function (HRTF), which is a transfer function that describes the impulse response of both ears of a human being. HRTF is the Fourier transform of the impulse response of each ear, $h_L(t)$ and $h_R(t)$. Simple convolution operation can be applied with the input signal, as human ears behave like FIR systems. Obtaining or calculating the binaural functions are outside of the scope of this thesis. Correct use of HRTF is required to produce binaural audio. Extra information at (Bilinski et al., 2014) and (Tashev, 2014) for example.
- The second concept is the room acoustics, which is the impulse response of a particular room. The topic was already discussed in the section 2.1. It is particularly important that the recording of the impulse response is done with a unit-impulse signal of different frequencies, so a more accurate impulse response function can be constructed. This part gives the reverberation or "echo" effect.
- The third concept is sound diffraction. The main idea explains how the change of medium affects the sound, for example a small opening in a wall allows the sound to travel across it. It is a very complicated topic which includes material analysis, which is far away from this thesis topic. This last part is rarely implemented into consumer software due to the complexity.

Nowadays, there research in immersive technologies focuses in creating HRTF functions for general use, and to generate mathematically the impulse response of a virtual room. The framework of Google Resonance (Google, 2018) tries to accomplish these tasks, but it is lacking proper sound reflection in real time, among others, but with an static environment it can give a sense of immersion.

2.4 Real Time systems

The concepts of real time and concurrency are introduced next. A real time system is one which time of operation is constrained by ”deadlines”. Missing the deadlines is not allowed, in the best scenario the experience will be degraded (audio stream) and in the worst people could die (nuclear reactor control program or plane control system). The former is called soft real time systems and the latter, hard real time systems. As the thesis topic focuses in audio processing and transmission, losses are allowed and even expected to occur at some point in the execution of the program. Concurrency in a system happens when two or more activities are happening at the same time, for example a human executes concurrently the actions of talking and walking usually. In computers, that can happen in *parallel*, so each task has its own processor, or sequentially switching tasks very fast in order for the deadlines of each one to be met.

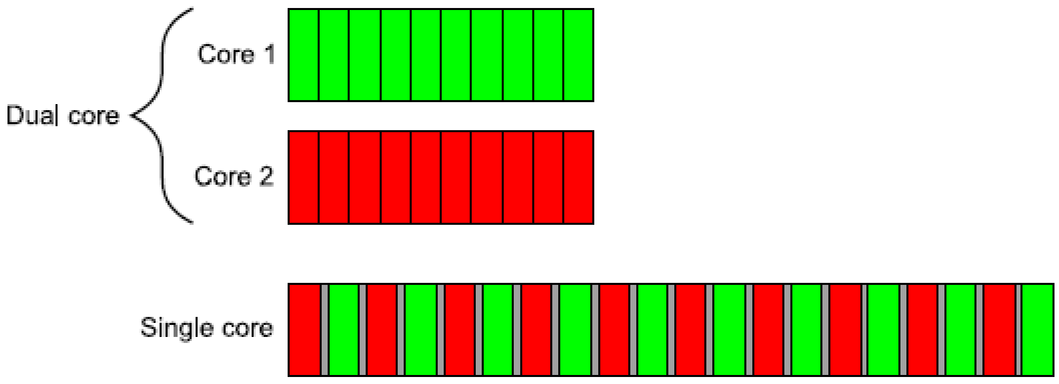

Figure 8. Approaches to concurrency. (Williams, 2012, p. 26).

Because of this fast change of tasks, all the variables and related data has to be stored and moved as each task changes. When two programs want to communicate with each other, or access common data, the information has to be stored on a common place as well, usually Random-Access Memory (RAM). Here the problems related to real time start to emerge, in the form of data races. This idea includes any action which concludes in unwanted modifications or inconsistent states of the data. Other problems included in concurrency are, for example, synchronization between programs, or protection of data from other programs.

Until C++11 (revision of the C++ programming language in 2011), there was not platform agnostic way of creating concurrent programs in C++. Each operating system had its own low-level Application Programming Interface (API) for creating concurrent programs. Each program could create threads (different series of instructions which share the same execution context) using those APIs, and had to create specific code for each system. In C++11, a general way of using concurrency in C++ was introduced by exposing classes representing threads, shared data structures or protected areas of memory, for example. This implementation is generic, so there is no need for vendor-specific code, however this creates some overhead and requires a bit more processing power and memory than the platform-specific APIs. The overhead is small enough to be ignored in most of the cases, even in the ones the performance is an issue. The Methodology section will examine the techniques used in order to create the real-time system of the thesis.

The author already knew the abstract concepts about concurrency and real time systems. However, there was a lack of knowledge in the implementation part using C++, so the following book was used as a foundation for C++ implementation of concurrent systems: (Williams, 2012).

2.5 Network communication

The last part of the background discussed is network transfer options for real-time systems. As the users who communicate can be on different parts of the world, the abstraction level should not go lower than the transport layer of OSI (Open System Interconnection) model. This allows that the information can be routed out the local network. Following the de facto standard of TCP/IP (because it is the most used by a wide margin), the two main options without developing a new network protocol are TCP and UDP. The Transmission Control Protocol (TCP) is an ordered (the packets arrive in order) and reliable (the packets do not get lost, they are resent if needed and the sender receives acknowledgment of reception) protocol for applications that cannot allow their packets to be lost in detriment of speed. It uses a session system to facilitate the sender to send multiple packages to the same recipient. On contrast, User Datagram Protocol (UDP) is a non-reliable, unordered and connectionless protocol. Without all the overhead that these features bring, UDP is able to achieve greater speeds than TCP, with the disadvantage of receiving the packages in different order they were sent, and the possibility of losing packages without the sender knowledge.

In C++, there are multiple options for networking, but this thesis will focus on the low-level API of sockets (both POSIX and WinSockets have almost the same API). The socket is a point on the networking stack which can receive and send packages. By accessing directly these sockets, the program can send and receive serialized (converted to a format suitable for transmission or storage) data. The book (Donahoo and Calvert, 2009) provides a reference to this API and addresses with more insight the concepts discussed in this subsection.

3 METHODOLOGY

The methodology of the thesis is discussed in this section. It includes a description of what was planned to be done and how, and a discussion of why certain plans were made and not others. An abstract system (system for the thesis scope) is a concept which hides the implementation details of a piece of software, leaving the logic as the main focus. The ”system” is the prototype being designed and any external piece of software will be named ”external system”.

3.1 Functional requirements

As in every system, the work started with the design and definition of the functional requirements, which outlines the goals of the system. The following requirements were initially set and the intention was that they should not be changed unless strictly necessary:

1. The design of an immersive communication system is desired.
2. The system could implement Peer to Peer (P2P) or client-server architectures.
3. The system will implement a binaural sound system with spatial location.
4. The HRTF and impulse response functions will be received from an external system, the system should only apply them.
5. The system should have a latency small enough so the communication is bearable between people using different devices in different spaces.
6. The system can compromise the audio frequency range in exchange of better performance.
7. The system should optimize the bandwidth use.
8. The system should support immersive communication between two people. Possibility to expand to more would be analyzed.

During the research and implementation parts, some concessions had to be made, in order to mitigate some implementation problems, and in order to meet the deadlines and other obligations of the author . These topics will be discussed in the Results and Discussion part. The design choices must be motivated by the requirements.

3.2 System concept

A general dataflow diagram which could be used as a reference for the system was created. This diagram was selected because it is based on the idea that in a very abstract level, the only concern is to move and transform the audio signal, so the best representation for that is a diagram which can represent the flow of data. For the 3rd,4th and 6th requirements one subsystem will be used in order to record the audio and process it applying the necessary FIR or IIR filters. For the 5th and 7th a subsystem for encoding and decoding the process audio should be used. Finally, for the 2nd, 5th and 8th a subsystem for network communication will be used. Those three subsystems will communicate with each other by the use of buffers, because some systems can work faster or in different paces than others. For format reasons, the diagram was split in two, the former (*Figure 9*) represents the flow of the data while recording and sending, whereas the latter (*Figure 10*) represents the opposite process of receiving the data.

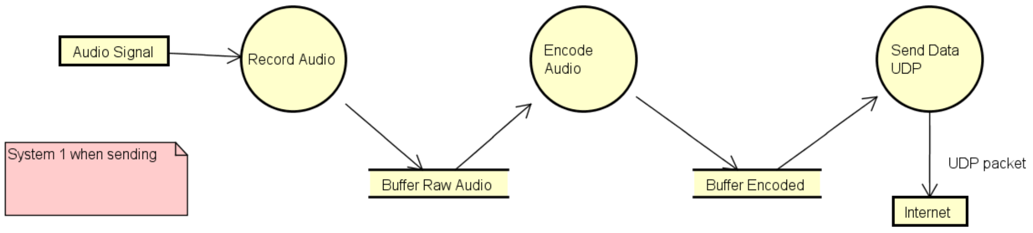

Figure 9. Dataflow diagram of the system while sending audio.

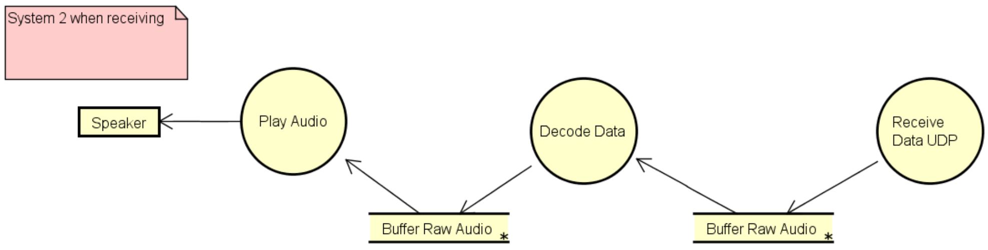

Figure 10. Dataflow diagram of the system while receiving audio.

These diagrams (*Figure 9 and Figure 10*) were very useful to represent the concept of the system, and they were used to model everything after it. The following diagram (*Figure 11*) was also used to model the system, in the form of sequence of actions and constraints, representing the asynchronous messages between the parts of the system.

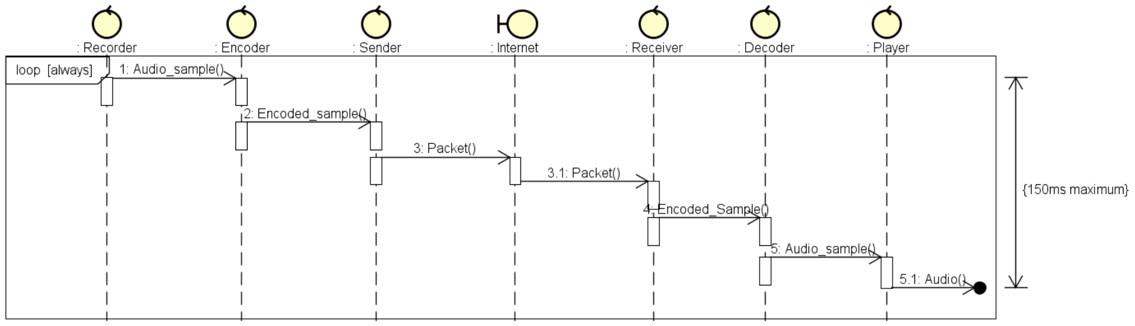

Figure 11. Sequence diagram of the communication process.

All these diagrams (*Figure 9, Figure 10 and Figure 11*) only represent part of the process. The system should be able to work in full duplex, receiving and sending at the same time. In the thesis, both systems will be seen as the same in reverted order unless specified. The first choice for developing this system is based on the programming language used. C++11 implements an easy way to manage real time systems, as well as low-level properties inherited from C, which makes it a suitable language for the implementation of the system. The author was familiar with the language, which was a major factor in the choice in order to speed up the development.

After the general design choices the implementation details are discussed. It was needed to consider the way of implementing the three subsystems (recorder/player, encoder/decoder and sender/receiver) into the system. The number of threads should be considered carefully, as it can introduce overhead. The initial diagrams imply that there should be six threads, one for each subsystem, but after performance testing will would be discussed later the number was reduced to five, one for the audio recording, one for the audio player, one for both decoding and encoding, one for sending and one for receiving.

3.3 Audio back end

The second important choice which had to be made was to choose an audio back end. Back end is the layer of a program which handles the logic of the system. The audio back end ise the part of the system which directly interacts with the audio hardware and provides data to other modules. In modern operative systems, the access to hardware is restricted by them, making the need of specialized driver and libraries to handle the use of certain devices. Microphones in particular are low-latency devices in which the real time constraints are important. A delay of half a second in the audio recording would be inadmissible. The author gathered information about very different libraries to try to find the most suitable for the purpose of the thesis. The following libraries were analyzed and researched in *Table 1*.

Table 1. Analysis of audio back ends (part 1).

Library	**Performance**	**DSP capabilities**	**Documentation**
$RTC - Mix$	High	All	Medium
$Common - Lisp - Music$	Low	Moderate	Good
$snd - rt$	High	All	Good
$cSound$	High	All	Medium
$FAUST$	Medium	All	Medium
$jSyn$	Low	All	Good
$Nyquist$	Low	All	Bad
$Puredata$	High	All	Moderate
$SuperColider$	High	All	Good
$VVVV$	High	All	Medium
$PortAudio$	High	All	Good

Table 1. Analysis of audio back ends (part 2).

Library	Ease to Code	License	Platform
$RTC - Mix$	Hard	Apache 2	*Nix
$Common - Lisp - Music$	Moderate	Unknown	Desktop OS
$snd - rt$	Moderate	Unknown	Desktop OS
$cSound$	Moderate	LGPL	*Nix
$FAUST$	Easy	GPL	All
$jSyn$	Moderate	Apache 2	All (Java support)
$Nyquist$	Hard	Unknown	All (Can run Lisp)
$Puredata$	Hard	BSD	All
$SuperColider$	Moderate	GPL	Desktop OS
$VVVV$	Hard	Free non comercial	Windows
$PortAudio$	Moderate	MIT	All

In the Appendix a longer description on the results on those tables is provided. The libraries were tested by trying to implement a simple FIR filter in each of them to process an audio file. The results are subjective to the author opinion in certain fields, for example *Ease to Code* or *Documentation*. The reason is that there is no good metric to measure the difficulty of coding using a specific library or to measure the quality of the documentation. The performance column is based on the real-time capabilities, being *High* able to process real time audio without hiccups and *Low* not being able to.

The audio back end selected was PortAudio according to two relevant criteria, the data presented in *Table 1* and the author's experience in programming. The license, performance and documentation were the most determinant factors in the choice. An audio back end with GPL (GNU General Public License) license should not be chosen because GPL forces to disclose the source code in case the application has commercial uses, which makes more difficult to write proprietary code. The license of PortAudio is very flexible in terms of code disclosure and commercial uses. Documentation of PortAudio is complete and answers most questions any coder would normally have. Performance-wise PortAudio allows access to low-level methods giving the developer the tools for creating fast and secure code.

3.4 Codec and Networking choices

The next choice needed is the encoding and decoding algorithm (codec) and its implementation into the system. There are different options available like MP3 codec with LAME implementation (Lame, 2018) or Opus codec with the C++ implementation (Opus, 2018). There are other codecs and implementations, but MP3/LAME and Opus were selected because of the refinement they have had during recent years. Regarding licenses, Opus uses a 3-clause BSD, which can be compared to MIT for the purposes of the thesis, and LAME uses GPL. Opus was chosen over MP3/LAME according to the same criteria used in the audio back end selection. Opus codec takes different parameters in order to improve its efficiency and quality.

- Complexity: This parameter sets the amount of CPU that the algorithm should use. Higher values increase the CPU usage, but that also *might* reduce the output size of the encoded audio.
- Signal type: There are different values which improve the functioning of the codec if the audio provided is the one the codec expects.
- LSB depth: This parameter sets the depth of the signal being encoded, useful for silence identification.
- Band used: This parameter creates a passband of different sizes, being the rest of the band ignored for the encoding.

One last important choice which had to be done was the networking system used. C implements a socket API which is thoughtfully discussed by (Donahoo and Calvert, 2009). This API accesses the low-level sockets, which make networking harder to program, but creates more efficient code when the programmer has enough time to spend in optimizing the code. Other option which was considered was using the network API of the Boost suite, which is called ASIO (Boost, 2018). This API provides an easier approach, with the cost of some overhead and some performance hits. The socket API of C was used for the prototype because of the need of performance.

3.5 Intermediate structures

In real time and concurrent systems, a great concern is how to allow processes to communicate with each other. The problem is even bigger if they want to transfer big amounts of data or if there is the need to have a steady stream of data. The data structure or buffer needs to allow read and write simultaneously. The recording thread records raw audio samples into a buffer. The encoding thread reads the samples from the structure, encodes them and writes the result to the next buffer. This last buffer is read by the sender thread. The process is analogous in the receiving part. The data structure needs to solve the concurrency problems that can appear, as well as discarding non-relevant data. For this purpose, a structure similar to a ringbuffer was needed.

A ringbuffer is a circular list (array or linked list) which cannot overflow because after the last position of the list comes the first again (Chandrasekaran, 2018). There can be two concurrent processes, one reading and one writing at the same time without data races, using one index for the reader and one for the writer. When the reader index is at the same point than the one of the writer, the reader can block until more data is available or can read a default value set by the programmer (i.e. if the data stored are floating values, a default value could be ”0.0f”). In this thesis, the latter approach was used because it is better to read silence than to block. Another concern was the data relevancy (the old data might be not relevant anymore). For that matter, in the implementation, when the data in the buffer exceeds a threshold value, the buffer will skip to the most recent audio, which will lead to delay correction. The buffer does two main operations, read and write to keep the implementation close to the general idea. Previous implementations like the one by (Thrasher, 2018) were available, but the author ended up creating his own implementation in order to meet specific features as the skipping to new data.

3.6 DSP effects

The FIR and IIR filters implementations were implemented in C++ aiming to optimal performance. For IIR filter, the direct form I and II (Kuo et al., 2013, pg. 158-159) were chosen. To test the filtering structures, actual parameters were needed. In order to identify if the filters were working correctly, simpler parameters were used, for example, to filter frequencies below or above some threshold. For this purpose, the program MatLab was used. MatLab allows an easy generation of the parameters for FIR and IIR filters. In the Appendix sample code to generate example parameters is provided.

3.7 Networking constraints and possible problems

The network design was considered next. TCP protocol is reliable and ordered, which is very convenient when dealing with streams of data. However, reliability and ordering provide performance issues in real time systems. UDP is neither reliable nor ordered, but it does not incur in the efficiency issues as TCP. Because of that, the concept of the network would be to use TCP for exchanging data apart from audio, like position or protocols, and use UDP for all of the real time communications. *Figure 12* illustrates the general concept of the system. The labels ”client” and ”server” are only useful to explain the order of connection. The client connects to the server using TCP. The client sends the desired UDP port to the server, which answers with its own UDP port. Now the two users can create an UDP connection as peers, which will be used to send the encoded audio in real time back and forth.

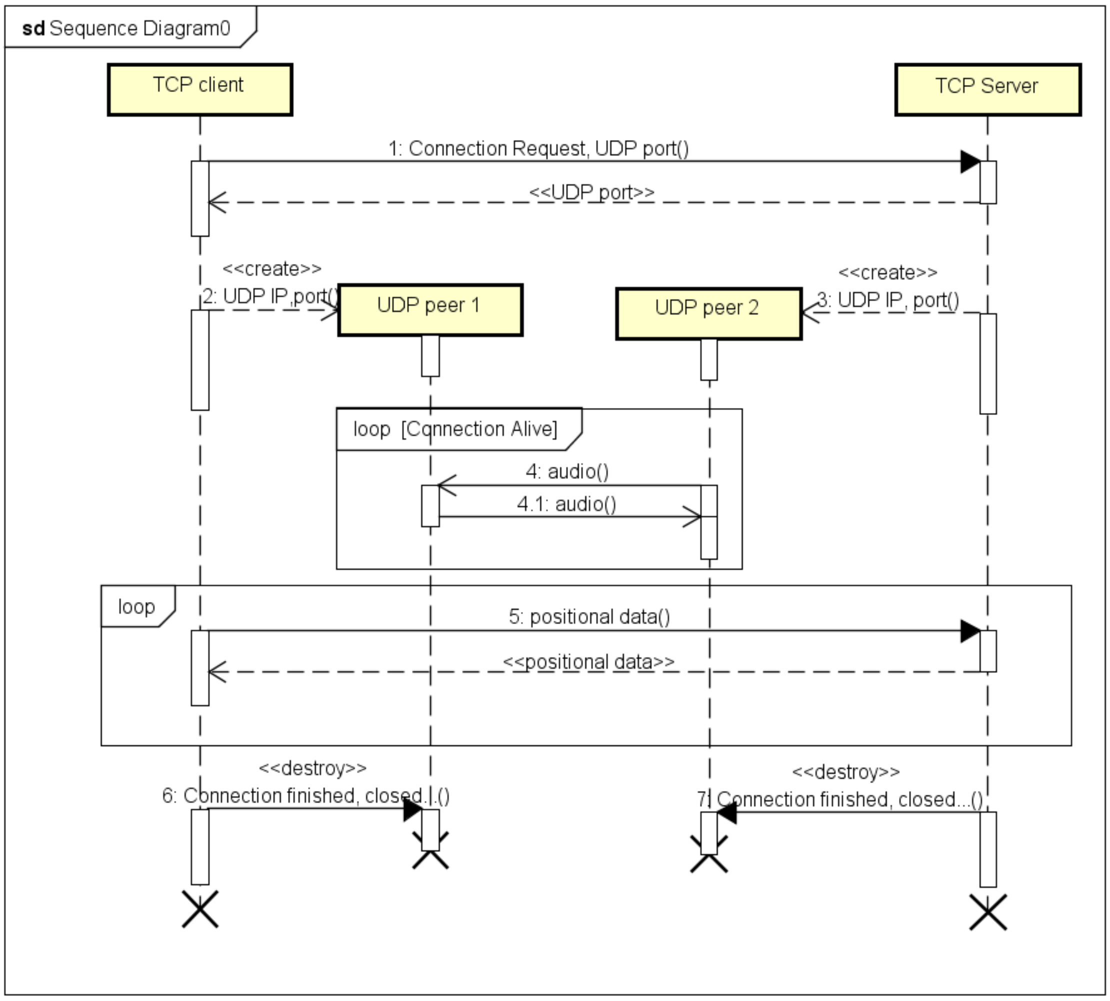

Figure 12. Network concept diagram.

The problem with UDP, as introduced before, is that packets can get lost or the received and sent order can be mismatched. The first problem is easily fixed by the encoder itself. Opus provides tolerance to missing packets, so silence is inserted and the decoder tries to recover the best audio as possible. The order of packets, on the other hand, is a more complicated question. This problem can be solved creating a data protocol to encode the packets before sending them, similar to the ordering by TCP, so they are reordered by the receiver.

3.8 Testing

As a last note, the testing process will be discussed. There are multiple parts of the system which could malfunction, so after every step the audio will be recorded into a file in binary format to be analyzed with Audacity. The raw audio will be stored, then the DSP audio, and after that the decoded audio. Also, the system will be subjectively tested by the author in real time, to calculate delays and quality problems. To automate the process in some way, sample sounds will be used as well as real time streams of audio to, for example, test only the DSP part.

On the other hand, the networking part of the system is not easily automated. The Opus codec is greatly capable of dealing with missing packets and error tolerance makes the sound degrade or even shutter, which is metric hardly quantifiable. The way this part was tested was by the author listening to the received audio and subjectively analyzing the quality of the sound. Other methods like network analysis with programs like (Wireshark-Foundation, 2018) were considered, but discarded as analyzing unordered packets is requires expertise or it can be very time consuming.

This is an example of a testing session. First, the author generates an arbitrary filter with MatLab to ensure the DSP work, for example a low-pass filter of 1000 Hz. This allows the voice to be greatly distorted with a characteristic effect. Then the filter is imported in the program, an the program is run in local mode, so the sound recorded in real time is played back as soon as it has been processed. The author then studies if the delay and the applied effects are acceptable and if there is any problem in this part.

Another example of session is the following. The program is run receiving an audio file instead of a recording of the microphone. The author then runs the program in another device and studies the delays and problems in the transmission of audio. The program in the first computer records as well any received audio by the network, which is analyzed later by the author in order to find silences or problems.

4 RESULTS

The results achieved with the methods listed in the Section 3 are detailed. The prototype of the system was built, but some problems rose in the process. Results are grouped together in the same subsections than in the Methodology, but with a chronological order instead.

4.1 Audio back end

The selected audio back end is PortAudio, for the reasons stated in the Methodology. PortAudio's performance is good enough to perform real time DSP with FIR and IIR filters. These filters were implemented with only one function, *dsp(float value)*, which receives the new input, and returns the processed output using the convolution algorithm. The canonical design was slightly modified by the author to improve performance in some cases. In order to fulfill the confidentiality agreement with myTrueSound the code will not be disclosed.

In *Figure 13* a Butterworth filter (simple example of IIR filter) was applied to a voice recording of the audio, with a passband of 500Hz (3dB ripple) and a stop-band of 3000Hz (60dB ripple), so the frequencies started to get attenuated after 500Hz and at 3000Hz the attenuation is extreme. As human hearing ranges from 20Hz to 20000Hz (Rosen and Howell, 2010), a reduction of such degree results on a significant reduction of the understandability of the sound. The sample used was the author's voice pronouncing the five vowels.

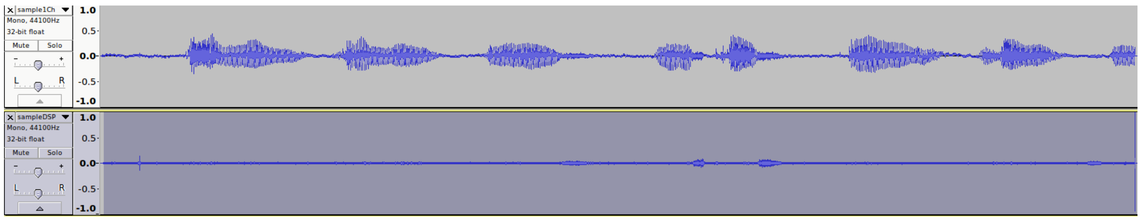

Figure 13. Audacity screen-shot of raw audio before and after DSP with filter of 500-3000Hz.

The processed audio shows the desired effect with the example effect. The filter was designed with MatLab, and examples are provided in the Appendix. This result shows proper DSP was implemented.

4.2 Codec tweaks

The codec chosen was Opus as it was written in the Methodology Section. This codec was tested with 6 seconds of voice of the author, as well as with real-time audio. Both the real-time audio and the static audio sizes were reduced drastically, from 392 KB/s to 12 KB/s, on average. Multiple audio samples were taken, like the one in *Figure 14*.

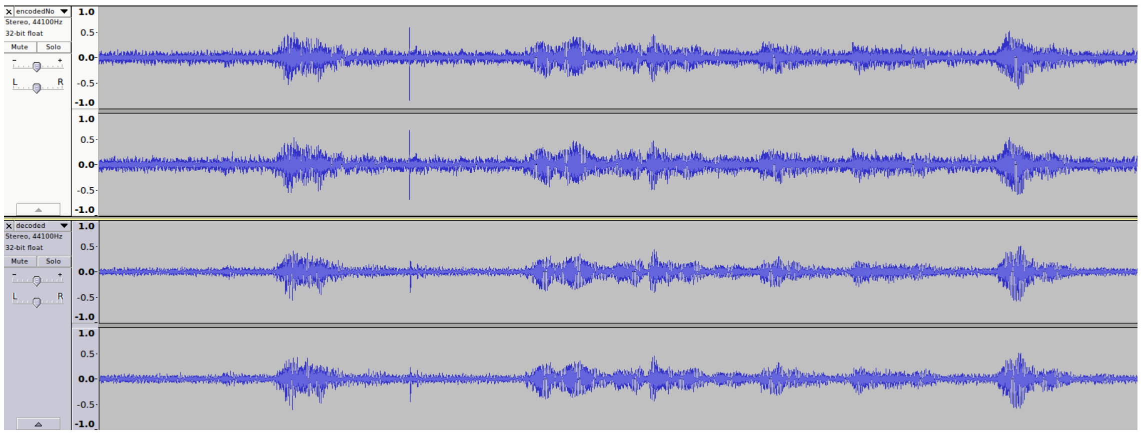

Figure 14. Audacity screen-shot of raw audio (top) and the same audio after being encoded and decoded (bottom).

It can be seen that the audio maintains a very similar structure, however the size of the encoded one is very reduced. Opus receives a series of parameters which could be further adjusted for different objectives:

- Complexity: This parameter sets the amount of CPU that the algorithm should use. Higher values increase the CPU usage, but that also *might* reduce the output size of the encoded audio. The author found that a value of 5 delivered a proper compression without an excessive CPU usage.
- Signal type: There are different values which improve the functioning of the codec if the audio provided is the one the codec expects. The proper value is *OPUS_SIGNAL_VOICE*.
- LSB depth: This parameter sets the depth of the signal being encoded, useful for silence identification. The author found a value of 16 performed optimally on his hardware conditions, but it can be tweaked if there are silences where not supposed.
- Band used: This parameter creates a passband of different sizes, being the rest of the band ignored for the encoding. The author found that a medium band (6 KHz max) performed good on his voice.

4.3 Intermediate structures

The ringbuffer data structure was implemented successfully, with the needed performance for transmitting audio samples between the parts of the system. The implementation details are not shown in order to fulfill the confidentiality agreement with myTrueSound, but a design similar to the one provided by (Thrasher, 2018) could be modified to behave similarly. The ringbuffer will skip to the most recent audio samples if the 15% percent of the buffer has been written and not read. This value was found reliably able to reduce any delay in communication. If the buffer is empty, the reader will read a default value of silence *0.0f*.

4.4 Network problems

Conclusive results in networking were not achieved. The network capabilities were partially implemented, with working serialization methods for the Opus packets to be sent. However, the lack of order and reliability of UDP could not be solved by the author in the time of the thesis. The prototype is able to send and receive data via TCP, but UDP transmission of data is unreliable.

This results in audio not being decoded correctly and silences appearing where they should not. In the Section 3.8 the testing methods were listed, however the audio received was in a quality not good enough to be able to be analyzed with Audacity, which crashes at opening the samples.

4.5 Requirements

At the Methodology part eight requirements were presented, which had to be fulfilled by the thesis program. Not all the requirements were met. The following symbols will be used, ✓ for completed requisites, ✗ for failed requisites and **!** for partially completed requisites

1. *The design of an immersive communication system is desired.*

! The system was designed and it can provide arbitrary impulse response DSP to any stream of audio. However it does not communicate properly with other systems over the network.

2. *The system could implement Peer to Peer (P2P) or client-server architectures.*

! The system implemented P2P communication, but it was not robust enough to handle the transfer of real time audio by UDP.

3. *The system will implement a binaural sound system with spatial location.*

✓ The system applied arbitrary impulse response. It was extended to apply different impulse to different audio channels, only needing the HRTF function and location data from a external system.

4. *The HRTF and impulse response functions will be received from an external system, the system should only apply them.*

✓ The system could apply HRTF and impulse response functions.

5. *The system should have a latency small enough the communication is bearable between people.*

✗ The system could not maintain the latency small enough.

6. *The system can compromise the audio frequency range in exchange of better performance.*

✓ The codec used provided this requirement to the system, as Opus can adjust the frequency range depending on the data.

7. *The system should optimize the bandwidth use.*

✓ The system optimized the data used to an average of a 3% of the raw audio data.

8. *The system should support two people in the beginning, with possibility to expand to more.*

✗ The network capabilities failed to be properly implemented.

5 DISCUSSION

In this section the results are discussed. To begin with, not all the requirements were fulfilled in the thesis prototype. The DSP capabilities were implemented whereas the network communication part failed to be properly developed. The major cause of that was underestimation of the complexity of the requirements. The author was neither an expert in networking, nor in DSP, so he had to research most of those fields. Because of that, the project grew bigger than initially thought, meaning some requirements could not been implemented. The requirements which could not be implemented were the ones related to the networking parts. The ones related to the rest of capabilities were properly implemented.

The audio back end was the first part of the system which was chosen. The author's choice was PortAudio, and for this thesis purposes worked flawlessly. This library accesses the hardware via different means which are not discussed, and it provides specific buffer from which bytes can be directly written or read, allowing a low-level approach to DSP implementations. The processed audio shows the desired effect with the example effect. The filter was designed with MatLab. Different filters were tried during the testing process, but the results would be redundant. There were either calculated with MatLab or extracted from real impulse responses from rooms. As shown in *Figure 13* the DSP capabilities work with any given set of coefficients. Coding the DSP implementation for PortAudio was very straightforward when a general filter was implemented, as the thread just needed to pass the new float values to said filter and then retrieve the values from the filter.

The codec chosen was Opus and the performance was excellent as expected and shown in *Figure 14*. The audio did not lose enough quality to be noticeable. If the device were not powerful enough to run the codec on the parameters specified, the complexity could be decreased, making the bandwidth increase as an effect. On the other hand, if the bandwidth was a problem then both the band could be reduced, impacting the audio quality, and the complexity could be increased, impacting the CPU usage. Careful tweaking of all the parameters would be necessary for optimal performance in any system.

Two implementations of ring buffers were implemented. One with only performance in mind, for transmission between the recorder and the encoder, and from the decoder and the player. The other one with a compromise between performance and other capabilities, in particular the ability to have sequential reads with random writes. This second buffer was created with the unordered UDP packets in mind. The packets are inserted where they belong and there they wait until the previous packets arrive or when the previous audio was processed.

Mistakes in the network implementation and complexity analysis were made. The author underestimated the time and complexity of trying to synchronize an unordered stream of packets. He also tried to implement all the parts instead on relying on already built libraries like Boost. The packets of Opus were encapsulated, given an identifier, serialized and then sent. The idea was that the packets were ordered on the receiver, sent to the second kind of ring buffer and then it will continue its path towards the headphones of the user. The author spent the most of the time of the thesis developing and testing the DSP parts of the system, not reserving enough time for networking.

The first mistake was that the code was not able to properly reorder the packets, leading to problematic mistakes in the decoder, as well as being a problem very time-consuming to debug. The code overwrote packets at certain positions, which lead to corruption of audio and cryptic errors of the decoder. If there was a stream of packets that managed to arrive and be reordered properly against the odds, then the decoder managed to decode it and properly do the playback of the sound.

Another problem, was to manage the low-level sockets to perform full duplex transmission of data. This problem was solved eventually unlike the previous one. The last part was to properly serialize and deserialize the packets from the encoder. A simple protocol was created by the author, with one unsigned integer for the index (so when the maximum number was reached, it will automatically go back to zero according to the C language implementation), one integer for storing the size of the packet and then the packet itself, an array of signed chars, which is the output of the Opus encoder. This was converted to signed chars and stored in a new memory address so the sender thread could access it without any problem. The deserialization was just the same steps in the opposite direction. This part was properly implemented as well, leaving only the problem of ordering the packets in real time unresolved.

6 CONCLUSION

This thesis aimed to create a new system capable of creating an immersive channel of communication between two people. For that purpose, the system should be able to apply any DSP effect (i.e room acoustics or HRTF) to the real time stream of audio that comes through the microphone. It shoudl also reduce the size to an acceptable level and then send it through the Internet to the other interlocutor, where it will have to be decoded and replayed. This process should work both ways simultaneously (full duplex).

The system was able to apply the required effects to the real time stream from the microphone. Different audio back ends were then studied and PortAudio was chosen because of its DSP capabilities, performance and license. Different DSP structures were implemented, mainly different algorithms for FIR and IIR filters, which form the foundation of digital signal processing. The system was able to import arbitrary FIR and IIR filters on will of the author, which means HRTF and acoustics could be effectively applied.

The system was able to reduce the data throughput created by the system from the raw data of the microphone. The rate of reduction was, on average, a 3% of the input size. The codec responsible for most of this reduction was Opus, which was configured to create the best audio quality for conversation, reducing the size to the maximum without affecting the intelligibility. For communicating both systems, Opus and PortAudio, a circular buffer or ring buffer was implementing, capable of transferring the data as fast as necessary.

The system was able to convert the resulting data created by Opus to a network-safe format, being capable of effectively serializing and deserializing data into and from a simple protocol created for this purpose. However, the difficulties of correct networking were underestimated and the system was not able to reliably send and receive audio in real time. This was not fully solved by the author as UDP packets arrived unordered, corrupting the audio. Thus, the system was only able to work reliably in a local environment, and not in a network.

Further development is required, and it is grouped in two main areas. The first and most obvious one would be to properly implement the networking system so the system works reliably even with many unordered packets and some tolerance to lost packets. For that the authors recommendation is to use one of the already established libraries for network communication such as Boost instead of implementing their own one from scratch, hence repeating the mistake of the author. The only proper reason to implement networking from scratch is the will to learn and the lack of performance of an already built solution using libraries.

The other area for further development is the improvement of usability. Currently the program lacks proper interface and the way of introducing new effects is not totally polished, so a graphical user interface (GUI) could be built around the system, allowing easier management. An API for the system could be created so external programs could introduce their own FIR or IIR coefficients in real time.

Summing up, the thesis achieved the most difficult parts of the objectives and the created system serves as a prototype which can be used to build an efficient system. Computer scientists with no background in digital signal processing can use this document to develop immersive systems that work in real time, which can increase the number of software with immersive software available.

References

Audacity (2018). *Audacity Online Manual*. https://manual.audacityteam.org/man/faq.html Retrieved 28-05-2018.

Audition, A. (2018). Abobe audition web page. https://www.adobe.com/la/products/audition.html Retrieved 28-05-2018.

Bilinski, P., Ahrens, J., Thomas, M. R. P., Tashev, I. J., and Platt, J. C. (2014). HRTF magnitude synthesis via sparse representation of anthropometric features. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE.

Boost (2018). Boost web page. https://www.boost.org Retrieved 28-05-2018.

Chandrasekaran, S. (2018). Implementing circular/ring buffer in embedded c. *Embed Journal*. https://embedjournal.com/implementing-circular-buffer-embedded-c/ Retrieved 28-05-2018.

Christie, J. (2008). The seductive and dangerous v model. *Testing Experience*. http://www.clarotesting.com/page11.htm.

Donahoo, M. J. and Calvert, K. L. (2009). *TCP/IP Sockets in C*. Elsevier Science & Technology.

Franz, D. (2008). *Recording and Producing in the Home Studio*. Berklee Press Publications.

Google (2018). Google resonance. https://developers.google.com/resonance-audio/ Retrieved 28-05-2018.

Kuo, S. M., Lee, B. H., and Tian, W. (2013). *Real-Time Digital Signal Processing*. John Wiley & Sons Inc.

Lame (2018). Lame web page. http://lame.sourceforge.net/ Retrieved 28-05-2018.

MatLab (2018). Matlab web page. https://mathworks.com/ Retrieved 28-05-2018.

Opus (2018). Opus web page. http://opus-codec.org/ Retrieved 28-05-2018.

Rosen, S. and Howell, P. (2010). *Signals and Systems for Speech and Hearing 2nd edition*. Emerald Group Publishing Limited.

Smith, S. W. (1997). *The Scientist & Engineer's Guide to Digital Signal Processing*. California Technical Pub.

Tashev, I. (2014). HRTF phase synthesis via sparse representation of anthropometric features. In *2014 Information Theory and Applications Workshop (ITA)*. IEEE.

Thrasher, P. (2018). Philip thrasher's crazy awesome ring buffer macros! https://github.com/pthrasher/c-generic-ring-buffer Retrieved 28-05-2018.

Williams, A. (2012). *C++ Concurrency*. Manning.

Wireshark-Foundation (2018). Wireshark. https://www.wireshark.org/ Retrieved 28-05-2018.

Appendix 1. Audio Table

Table 1 and Table 2 gather the results of a highly subjective research conducted by the author. The objective was to provide an easy summary of most audio back end available for the thesis scope. The columns will now be listed and the methodology will be explained:

- **Product:** the different audio back ends researched.
- **Performance** if the library was able to perform real time operations with audio, more specifically, if it was able to apply any DSP effect to the input from the microphone. High means that the system is totally able to apply effects in real time, Medium means that the system can perform in real time under specific conditions (FAUST creates an object which could be modified to work in real time) and Low means that the system cannot perform real time DSP without important modifications.
- **DSP capabilities:** in this column the author considered that implementing a FIR structure and/or a IIR structure proves that any DSP operations can be performed, as both systems are composed by the 3 basic parts of any DSP system. All means that they could be implemented, Moderate means that some DSP can be applied, but it is limited in some way (Common-Lisp-Music is designed for composer, so it allows high customization, but total).
- **Documentation:** this column is totally subjective and its values come from how difficult the author found the necessary documentation for building the FIR and IIR filters. The different values do not relate to different amounts of time, just the impression of the author.
- **Ease to code:** this column is similar to the Documentation one, as it measures how hard the author found the coding of the former mentioned structures.
- **License:** this column displays the license of every back end library. The ones marked with *Unknown* were license which the author was uncapable to find in their own web pages and documentation.
- **Platform:** this column gathers the platform in which the library works, according to the library web page and documentation. The author did not test every library with every operating system, as it would go outside of the thesis topic.

Appendix 2. MatLab filter

In this subsection some example code for generating FIR and IIR filters. In the following code, an example IIR filter is created using the butterworth method.

```
        function [b,a]=createButter(nyquist,passband,stopband,ripple,attenuation)
    Wp=passband/nyquist;
    Ws=stopband/nyquist;
    Rp=ripple;
    Rs=attenuation;
    [N,Wn]=buttord(Wp,Ws,Rp,Rs);
    [b,a]=butter(N,Wn);
```

And in this one, a highpass FIR filter is created. The kind parameter can only take ’high’ and ’low’, according to the documentation for *fir1*

```
function b=createFIR(taps,frequency,nyquist,ripple,kind)
        Wn=frequency/nyquist;
        b=fir1(taps,Wn,kind,chebwin(taps+1,ripple));
```

With both example functions, the coefficients used in the filters can be generated for testing purposes.

Appendix 3. Filter implementation

In this subsection, the way FIR filters and IIR filters will be shown. Some details of the implementation, like how to efficiently store the data will not be shown in order to fulfill the confidentiality agreement with myTrueSound.

FIR filter

```
#ifndef SPFIRFILTER_H
#define SPFIRFILTER_H
#include "SPGenericFilter.h"
template<class T>
class SPFirFilter : public SPGenericFilter<T>{
    public:
        SPFirFilter(int samples,T* coef){
            this->size=samples;
            this->coef=coef;
            this->buffer=new firbuffer<T>(this->size);
        }
        SPFirFilter(){
            this->buffer=NULL;
        }
        ~SPFirFilter(){
            std::cout<<"Destructor of FirFilter S\n"<<std::flush;
            printf("%p\n",buffer);
            if (buffer){
                delete buffer;
            }
            std::cout<<"Destructor of FirFilter F\n"<<std::flush;
        }
        T getTap(T input){
            this->buffer->add(input);
            this->buffer->reset();
            T acc=0;
            for (int i=0;i<this->size;++i){
                acc+=buffer->read()*coef[i];
            }
            return acc;
        }
        protected:
            T* coef;
        private:
            firbuffer<T> *buffer;
};
#endif /* SPFIRFILTER_H */
```

IIR filter

```
#ifndef SPIIRFILTER_H
#define SPIIRFILTER_H
#include "SPGenericFilter.h"
//FORM I
template<class T>
class SPIirFilter: public SPGenericFilter<T>{
    public:
        SPIirFilter(){};
        ~SPIirFilter(){
            delete bufferB;
            delete bufferA;
        }
        SPIirFilter(int size,T* coefA,T* coefB){
            this->coefA=coefA;
            this->coefB=coefB;
            this->size=size;
            this->bufferB=new firbuffer<T>(size);
            this->bufferA=new firbuffer<T>(size);
        }
        T getTap(T input){
            bufferB->add(input);
            bufferB->reset();
            double acc=0.0;
            for (int i=0;i<this->size;++i){
                double temp=bufferB->read();
                temp*=(double)coefB[i];
                acc+=temp;
            }

            for (int i=1;i<this->size;++i){
                acc-=bufferA->read()*coefA[i];
            }
            bufferA->add(acc);
            bufferA->reset();
            return acc;
        }
        protected:
            T* coefA;
            T* coefB;
    private:
        firbuffer<T> *bufferB=NULL;
        firbuffer<T> *bufferA=NULL;
};
#endif /* SPIIRFILTER_H */
```

Appendix 4. PortAudio

In this subsection, reference code will be provided for PortAudio. The following snippet records and apply an effect provided by an external structure. There are two main parts, the callback, which is the low-level function PortAudio executes, and the initial call, which sets the parameters for the callback.

```
//Callback function
static int recordCallback( const void *inputBuffer, void *outputBuffer,
        unsigned long framesPerBuffer,
        const PaStreamCallbackTimeInfo* timeInfo,
        PaStreamCallbackFlags statusFlags,
        void *userData )
{
    const SAMPLE *rptr = (const SAMPLE*)inputBuffer;
    (void) outputBuffer; /* Prevent unused variable warnings. */
    (void) timeInfo;
    (void) statusFlags;
    unsigned int i;
    int finished;
    //Input data to the thread
    SPSoundThread* thread=(SPSoundThread*)userData;
    SPRingBuffer<RAW_TYPE> *bufferRaw=thread->getBuffer();
    SAMPLE sample;
    if( inputBuffer != NULL ){
        for( i=0; i<framesPerBuffer; i++ ){
            sample=thread->dsp(*rptr++);//First channel
            bufferRaw->write(sample);//Writes to a ringbuffer
            sample=thread->dsp(*rptr++);//Second channel
            bufferRaw->write(sample);
        }
    }
    //This part controlls if the thread continues or stops, from an external
    ↪  variable.
    if (thread->getState()==THREAD_START) finished=paContinue;
    else finished=paComplete;
    return finished;
}
//Init function
void SPSoundThread::init(){
    PaStreamParameters  inputParameters;
    inputParameters.device = Pa_GetDefaultInputDevice(); /* default input device
    ↪  */
    if (inputParameters.device == paNoDevice) {
        throw SPGenericException("Error: No default input device");
    }
    inputParameters.channelCount = 2;                    /* stereo input */
    inputParameters.sampleFormat = PA_SAMPLE_TYPE;
    inputParameters.suggestedLatency = Pa_GetDeviceInfo( inputParameters.device
    ↪  )->defaultLowInputLatency;
    inputParameters.hostApiSpecificStreamInfo = NULL;

    err = Pa_OpenStream(
            &stream,
            &inputParameters,
                NULL,                  /* &outputParameters, */
                SAMPLE_RATE, //Defined elsewhere
                FRAME_SIZE,  //Defined elsewhere
                paClipOff,
                recordCallback,
                this ); //Input parameter
    if( err != paNoError ){
        throw SPGenericException(Pa_GetErrorText(err));
    }
    err = Pa_StartStream( stream );
    if( err != paNoError ){
        throw SPGenericException(Pa_GetErrorText(err));
    }
    while( ( err = Pa_IsStreamActive( stream ) ) == 1 &&
    ↪   this->threadState==THREAD_START)
    {
        Pa_Sleep(100);
    }
    if( err < 0 ){
        throw SPGenericException(Pa_GetErrorText(err));
    }

    err = Pa_CloseStream( stream );
    if( err != paNoError ){
        throw SPGenericException(Pa_GetErrorText(err));
    }
}
```