UNIVERSITAT POLITÈCNICA DE VALÈNCIA

MASTER'S DEGREE IN PARALLEL AND DISTRIBUTED COMPUTING

Academic Course 2015/2016

Master's Degree Thesis

Development of a Multichannel Audio Application on a distributed system using ARM Multi-Core Processors

Student:

Mihaita Alexandru Lupoiu

Supervisor:

José A. Belloch

Antonio M. Vidal

July 11, 2016

To

my family

Acknowledgements

I would first like to thank my thesis advisors Antonio M. Vidal Marciá and José A. Belloch Rodríguez. Their door office was always open whenever I ran into a trouble spot or had a question about my research or writing. They consistently allowed this paper to be my own work, but steered me in the right direction whenever they thought I needed it.

Furthermore I would like to thank Prof. Enrique S. Quintana Ortí from the University of Jaume I and Prof. Alberto González from the Universitat Politècnica de València for introducing me to the topic as well for their support on the way. I want to thank also Dr. Diego F. Larios Marín from the University of Sevilla for his collaboration in this work as well as for his implementation tips.

I would like to show my gratitude to all the people at the Universitat Politècnica de València that shared my daily work at the Institute of Telecommunications and Multimedia Applications (iTEAM). I also want to thank my classmates from the master's degree for their support and for all the good moments spent together.

Last, but not least, I would like to express my sincere gratitude to my parents and to my girlfriend for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Author

Mihaita Alexandru Lupoiu

Resumen

El sistema WFS es un sistema de reproducción de audio que recrea con bastante verosimilitud el campo producido por una fuente sonora dentro de una área de escucha. Cuando el número de fuentes a sintetizar y de altavoces en el sistema es amplio, los requisitos computacionales aumentan significativamente. La elaboración de sistemas distribuidos nos permite poder implementar un sistema modular y escalable, al tiempo que la carga computacional requerida por la implementación es repartida entre sus diferentes nodos. Un problema importante a resolver por estos sistemas distribuidos es la sincronización entre nodos ya que es un factor limitante para muchas aplicaciones de audio. El presente trabajo analiza el sincronismo que existe entre nodos para dos tipos de sistemas distribuidos que implementan un sistema WFS.

Summary

Wave Field Synthesis (WFS) is a spatial audio reproduction system that provides an accurate spatial sound field in a wide area. When the system involves multiple loudspeakers and multiple sound sources, the computational requirements increase meaningfully. The configuration of a distributed system composed of acoustic nodes allows us not only to implement a modular and scalable system, but also to share the computational load among all nodes. An important issue to tackle in distributed systems is the synchronization among all nodes, even more when spatial effect perception depends on the accurate delays in the sound signals. The present work analyzes the fundamental aspects to take into account when a WFS is implemented and proposes a communication protocol among the acoustic nodes.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

In the last decade, explicitly parallel systems are being accepted in all segments of the industry in the form of multicore processors and many-core hardware accelerators, being the field of the digital signal processing one of the benefited with this acceleration, and more specifically the synthesis of the spatial audio.

One of the spatial audio technologies available today is Wave Field Synthesis (WFS) in which a sound field is synthesized in a wide area by means of loudspeaker arrays, which are referred to as secondary sources. WFS is usually tackled via digital signal processing techniques to reproduce complex auditory scenes consisting of multiple acoustic objects, which are generally denoted as primary or virtual sources. The WFS concept was introduced at the Delft University of Technology in the 1980s. Berkhout developed the first investigations in this field [27, 28], which were followed by several dissertation works [39, 35, 38, 34, 30].

A large-scale WFS system requires costly computational operations in real time since it involves multiples input and output channels. In recent years, there have been several studies aimed at implementing a WFS system. An approach that benefited from time-invariant preprocessing in order to reduce the computational load is presented in [33]. In [36, 25, 26], a WFS implementation on different multi-core platforms, including a GPU-based implementation was presented.

However, the proposed applications present an expensive cost and low flexibility, since all of them are centralized. The main contribution of this Master Thesis is to design a distributed system composed of acoustic nodes that are able to execute a Wave Field Synthesis application. These nodes are composed of multi-core ARM processors that consume low power and allows to be moved/portable easily. The development of this distributed system allows to obtain a flexible application that can adapt easily to the acoustic requirements, i.e. the number of nodes can increase or decrease at any time.

This system presents two levels of parallelism: Computational distribution among acoustic nodes, and parallelism inside the processor (taking maximum profit of the parallel resources such as multi-core processors or vectorial registers). Next subsection summarizes the objectives of this Master Thesis.

1.2 Objectives

Considering all the motivation aspects, the main objective of this work is to develop an application capable of reproducing spatial sound. This means that a user is able to locate positions of sound sources while these are being reproduced The main contribution is that the reproduction is carried out in a distributed manner by acoustic nodes that are composed of ARM processors. The main advantage of this system is the possibility of adding or removing acoustic nodes resulting a modular system that can be adapted to the environment and to the availability of resources. More specific objectives are:

- Evaluate synchronization protocols that allow to execute a Wave Field Synthesis system.
- Analyze operations that are required to implement a Wave Field Synthesis application: convolution.
- Analyze the different possibilities of implementing the convolution on the ARM processors.
- Implementation of the required operations on the ARM processors.
- Optimization of the operations attending to the parallel resources of the processors: Multi-core and Vectorial registers.
- Design a configuration file system so that each node can dispose of a personalized configuration.

- Obtain computational performances.

1.3 Organization of the Thesis

The organization of this thesis is distributed in the following chapters. Chapter 2 describes the state of the art dedicating a section to Walve Fild Synthesis and other section to the synchronization issues. Moreover, chapter 2 presents the required tools for carrying out this application. Implementations details are devoted to chapter 3. In Chapter 4, we evaluate the computational performance of our WFS system, and we summarize our results in Chapter 5.

Chapter 2

State-of-the-Art

2.1 Wave Field Synthesis

One of the spatial audio rendering techniques available today is the Wave Field Synthesis (WFS), where sound field is synthesized by a large number of individually driven loudspeakers. These wave fronts seem to originate from a virtual starting point, the virtual source or notional source. Contrary to traditional spatialization techniques such as stereo or surround sound, the localization of virtual sources in WFS does not depend on or change with the listener's position. WFS is based on the Huygens–Fresnel principle, which states that any wave front can be regarded as a superposition of elementary spherical waves. Therefore, any wave front can be synthesized from such elementary waves.

The basic procedure of WFS was developed in 1988 by Professor Berkhout at the Technical University of Delft. Unlike conventional audio procedures (e.g stereo /surround) the perception of these wave fields is not dependent on psychoacoustic phantom sound source perception. The WFS sound field is actually reconstructed physically.

The positioning of sound within the stereo system (that is based on the same principal as surround) originates from intensity differences or short time differences between both speakers. In the "real world" when sound is perceived from a natural sound source in the environment it arrives at one ear slightly earlier that the other (that is, if the source is not situated straight in front of the listener). Because of this inter-aural time difference we (for frequencies below 1.500 Herz) use this information to determine which physical direction the sound is coming from.

With stereo reproduction technique the sound of the left loudspeaker arrives first at the left ear and later at the right ear, and sound from the right loudspeaker arrives first at the right ear and then later to the left ear. The human brain ‘makes sense’ of this intensity difference to determine the “apparent location” of the sound source.

For the perception of natural sound sources above 1.500 Herz use is made of inter-aural time differences as well as a coloration of the sound, caused amongst other factors by the physical interference of the human head and ear lobes. Again the normal stereo system is deficient in rendering this spatial information and fails to convince, because the sound comes from two loud speakers. Phantom locations emerge and the sound placement is very difficult to localize with any precision. In this situation the brain determines the apparent location of the sound based on the apparent intensity difference. The listener needs to be situated exactly in the middle of the speakers in order to correctly perceive the positioning of stereo sound, because this is the only spot where the ratios of the intensity differences are accurate. With the WFS technique by means of a large number of loud speakers the wave itself is reconstructed. This ensures that the distance between the sound source and the listener is smaller than the distance between listener and the loudspeaker, so any desired wave can be reconstructed. This possibility cannot be achieved with stereo technique, because the loudspeakers are the sources themselves and do not reconstruct a source.

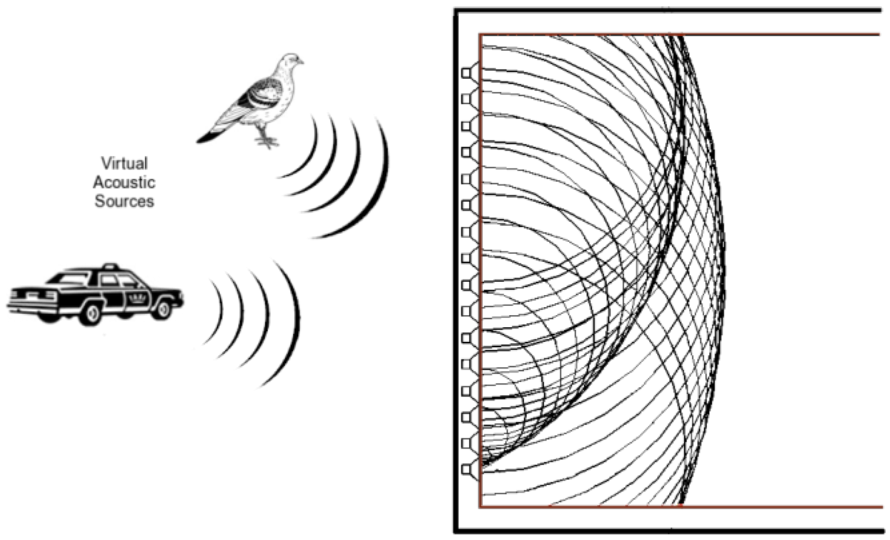


Figure 2.1: Reconstruction of the wavefront by the laudspeakers array.

In practice, a computer controls a large array of individual loudspeakers arranged as arrays around the listener and the computer synthesis activates each solitary loudspeaker membrane, at the time when the virtual wave front would pass through it.

Sounds are no longer simulated (like stereo and surround systems which use psychoacoustic principles to “fool” the perceptual system) and sound reproduction is no longer based on psychoacoustic principles, but instead on purely physical principles.

2.1.1 Convolution

Convolution is a mathematical way of combining two signals to form a third signal. It is the single most important technique in Digital Signal Processing. Using the strategy of impulse decomposition, systems are described by a signal called the impulse response. Convolution is important because it relates the three signals of interest: the input signal (f), the output signal (f*g), and the impulse response(g).

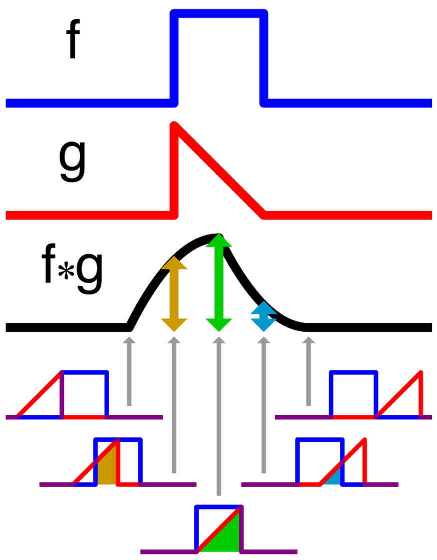


Figure 2.2: Convolution process.

The problem is to implement on-going, non-cyclic convolution with the finite-length, cyclic convolution that the FFT gives. An answer was quickly found in a clever organization of piecing together blocks of data using what is now called the overlap-add method and the overlap-save method.

Overlap-Save

Considering that h and x have a size of l_h and l_x samples, respectively with $l_x \gg l_h$, we can convolve both sequences by breaking the long signal into blocks of lo samples with the peculiarity that each block has an overlap of $l_h - 1$ samples with the previous block. The size lo is the first power-of-two integer that is $l_o \geq l_x + l_h - 1$. First block of x will be zero-padded at the beginning with with $l_h - 1$ samples. Although we take $l_o = l_x + l_h - 1$ and an overlap of $lh - 1$ samples, most of the authors in literature take a value of $lo = 2lh$ with an overlap of lh samples, which gives a better efficiency [37]. Figure 2.3 illustrates this block division where x^i represents the block i-th of signal x.

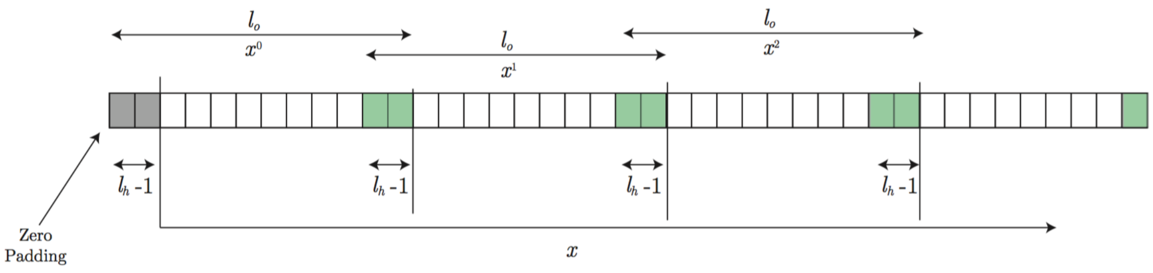


Figure 2.3: Overlap-save: Split the signal x in blocks of size l_o.

Sequence h is also zero-padded up its size is lo. Thus, each block x_i and h have the same size. The FFT is applied to each block x^i and the sequence h, then each block is element-wise multiplied by the H (Fourier transformed of h) and gives as a result another block Y^i composed of l_o samples. Figure 2.4 illustrates these operations where X^i and Y^i denote the Fourier transforms of the blocks x^i and y^i, respectively.

Finally, the first $l_h - 1$ samples of every block y^i are discarded. To configure the output signal y, all the blocks are afterwards concatenated, as is shown in Figure 2.5.

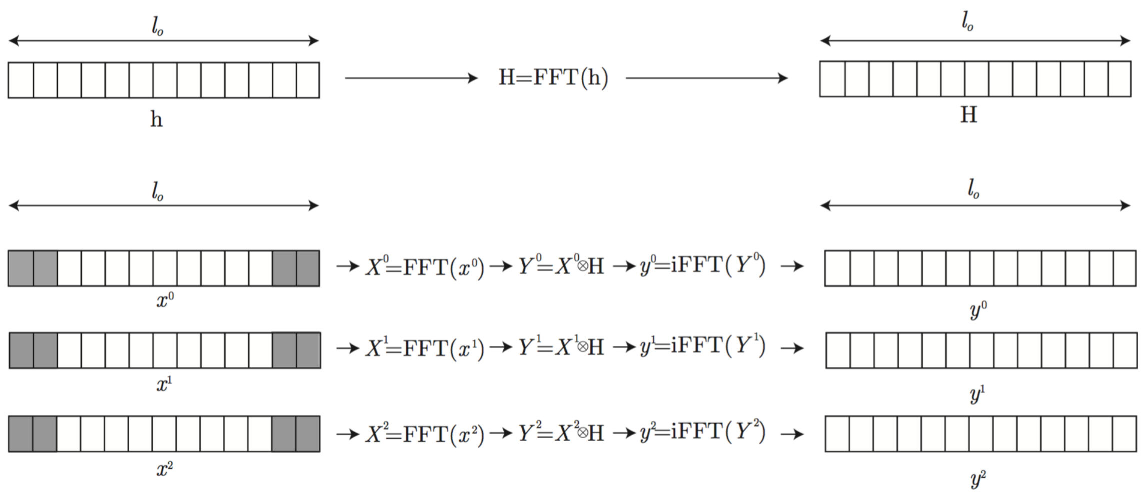

l_o
h
H=FFT(h)
H
x^0
X^0=FFT(x^0) → $Y^0=X^0 \otimes$H → y^0=iFFT(Y^0)
y^0
x^1
X^1=FFT(x^1) → $Y^1=X^1 \otimes$H → y^1=iFFT(Y^1)
y^1
x^2
X^2=FFT(x^2) → $Y^2=X^2 \otimes$H → y^2=iFFT(Y^2)
y^2

Figure 2.4: Overlap-save: Each block x^i together with h are Fourier transformed and element-wise multiplied.

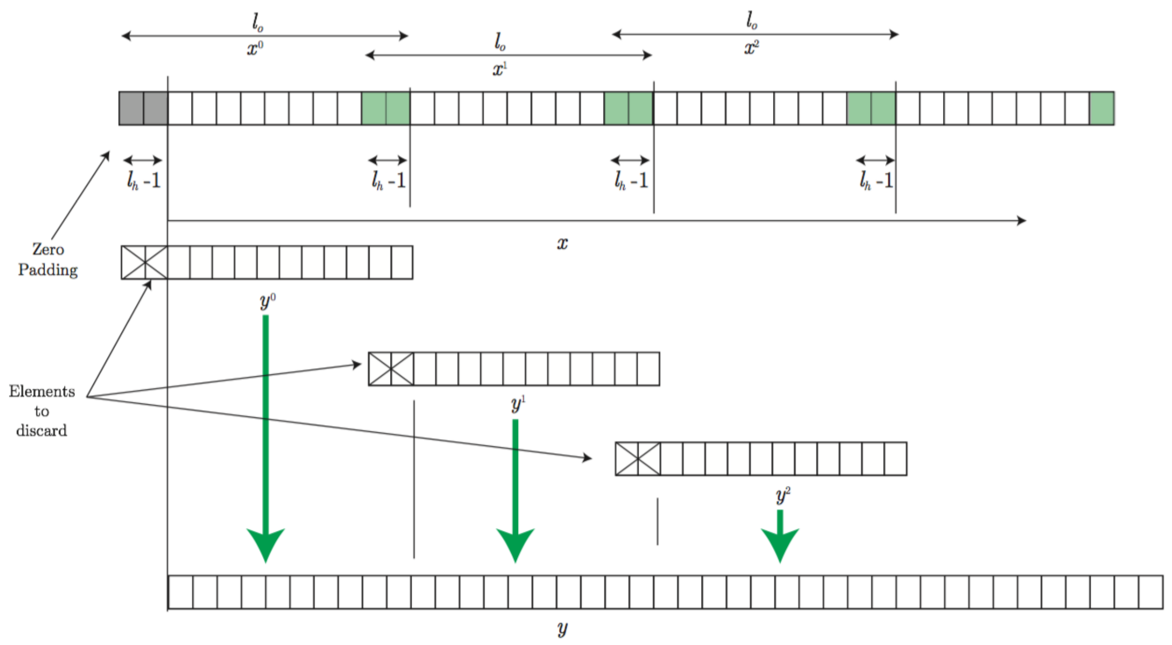


Figure 2.5: Overlap-save: To configure output signal y, the first $l_h - 1$ of every block y^i are discarded.

Overlap-Add

Overlap-add works also in blocks of size l_o, but with the peculiarity that the last $l_x - 1$ samples of the block are zero padded. Thus, each block x^i is configured by $l_o - l_h + 1$ samples of the signal x and $l_h - 1$ zeros. The sequence h is again zero-padded up its size is l_o. Fourier transforms and element-wise multiplications between the blocks x^i and the sequence h are carried out in the same way as in overlap-save, as shown in Figure 2.4.

However, to concatenate all the resulting y_i blocks, the last $l_h - 1$ samples of block y_i must be added to the first $l_h - 1$ samples of the block y^{i+1}. Figure 2.6 shows this processing.

Overlap-save and overlap-add only compute $l_o - l_h + 1$ elements per processed block of length l_o, but overlap-add has two additional steps: One of zero-padding the input, and one of summing up with the overlap from the previous iteration. In practice, this will always result in worse performance for the overlap-add algorithm. Therefore, the overlap-save method should be preferred in real-time processing if using the FFT.

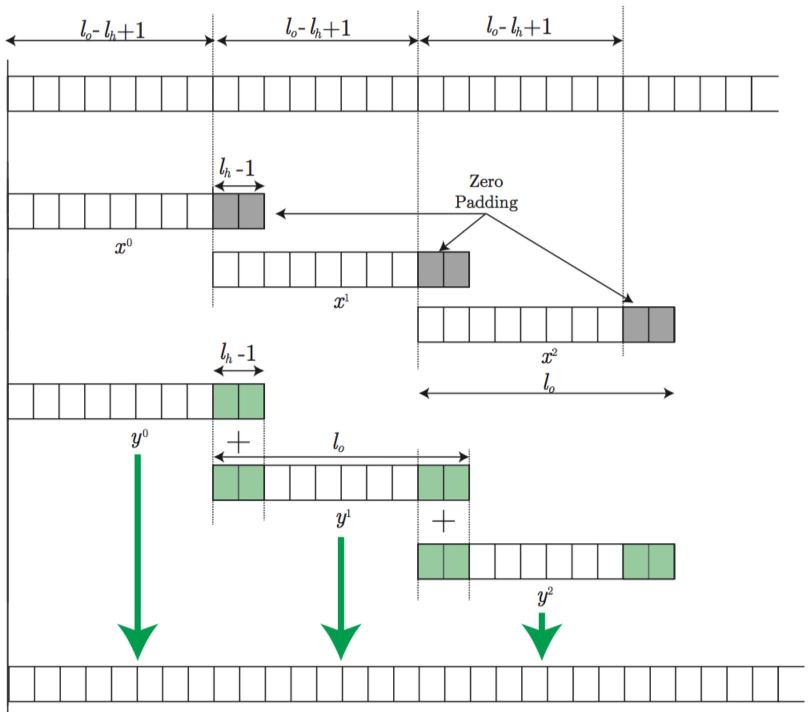


Figure 2.6: Overlap-add: To configure output signal y, the last $l_h - 1$ samples of block y^i must be added to the first $l_h - 1$ samples of the block y^{i+1}.

Fast Fourier Transform

Fast Fourier Transform (FFT) is an effective algorithm for computing the Discrete Fourier Transform. It was developed by Cooley and Tukey in 1965 [26]. This algorithm reduces the computation time of DFT for l_x points from l_x^2 to $l_x log_2(l_x)$. It is also called the Butterfly algorithm and is based on divide-and-conquer algorithms. It consists of dividing the transform into two pieces of size l_x recursively, and is therefore limited to power-of-two 2 sizes. In case l_x is not a power of two, a zero padding at the end of the data can be carried out in order to employ more efficiently the algorithm. The inverse Fast Fourier Transform (iFFT) corresponds to the effective algorithm of the inverse Discrete Fourier Transform.

2.2 Synchronization

In modern computer networks time synchronization is critical because every aspect of management, securing, planning, and debugging a network involves determining when events happen. Time also provides the only frame of reference among all devices on the network. Without synchronized time, accurately correlating log files between these devices is difficult, even impossible.

For the purpose of the application we study in this work, time synchronization is critical in order to be able to create a Wave Field Synthesis and to be able to localize the source of the sound. That is because all the nodes have to start to reproduce in an exact moment of time. Otherwise the user may not perceive the sound from one source, and instead it would perceive several sources with the same sound.

There are several ways to synchronize several network nodes, but in this case we focus on the “Network Time Protocol” (NTP) given it’s wide use around the world and the “Precision Time Protocol” [3].

2.2.1 Network Time Protocol

Network Time Protocol or NTP is an Internet protocol used to synchronize the clocks of computers to some time reference. NTP is an Internet standard protocol originally developed by Professor David L. Mills at the University of Delaware [9].

NTP is organized in a hierarchical client-server model. In the top of this hierarchy there are a small number of machines known as reference clocks. A reference clock is known as stratum[1] 0 and is typically a cesium clock or a Global Positioning System (GPS) that receives time from satellites. Attached to these machines there are the so-called stratum 1 servers (that is, stratum 0 clients), which are the top level time servers available to the Internet [9].

Following this hierarchy, the next level in the structure are the stratum 2 servers which in turn are the clients for stratum 1 servers. The lowest level of the hierarchy is made up by stratum 16 servers. Generally speaking, every server synchronized with a stratum n server is termed as being at stratum n+1 level. So, there are a few stratum 1 servers which are referenced by stratum

[1]measure for synchronization distance and it refers to the number of steps that a system lies from a primary time source

2 servers, which in turn are referenced by stratum 3 servers, which are referenced by stratum 4 and so on [9, 16].

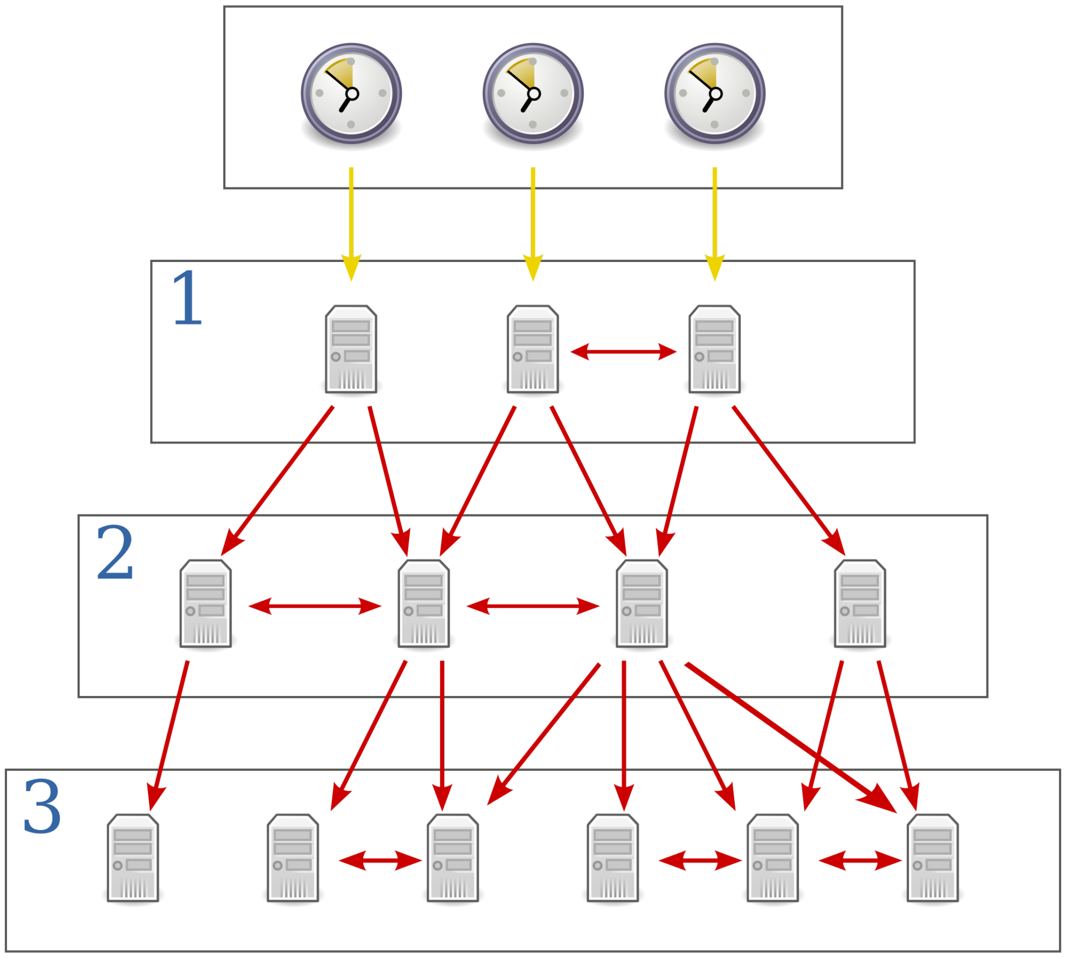


Figure 2.7: NTP communication, where yellow arrows indicate a direct connection and red arrows indicate a network connection.

NTP servers operating in the same stratum may be associated with others in a peer to peer basis, so they may decide who has the higher quality of time and then can synchronize to the most accurate.

In addition to the client-server model and the peer to peer model, a server may broadcast time to a broadcast or multicast IP addresses and clients may be configured to synchronize to these broadcast time signals.

2.2.2 Precision Time Protocol

Precision Time Protocol or PTP is a protocol used to precisely synchronize the system clock of distributed nodes in a system using standardized packet based networks like Ethernet. When used in conjunction with hardware support, PTP is capable of sub-microsecond accuracy, which is far better than is normally obtainable with NTP.

PTP was originally defined in the IEEE 1588-2002 standard, officially entitled “Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems” and published in 2002. In 2008, IEEE 1588-2008 was released as a revised standard; also known as PTP Version 2, it improves accuracy, precision and robustness but is not backward compatible with the original 2002 version [10].

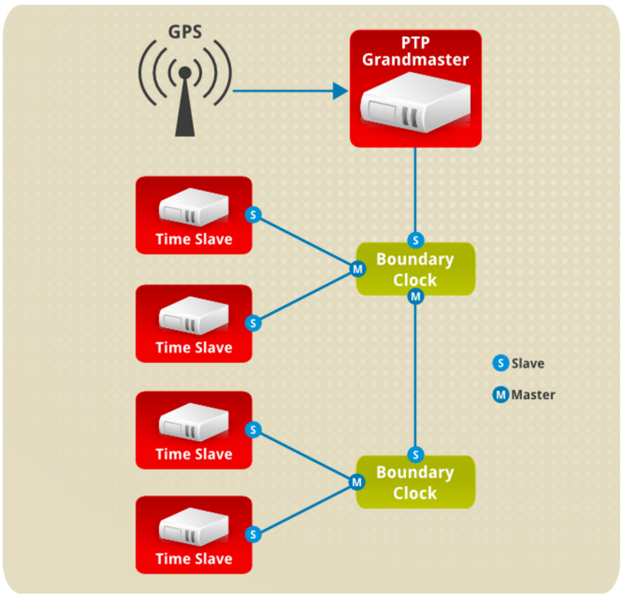


Figure 2.8: PTP grandmaster, boundary, and slave Clocks [10].

The clocks synchronized by PTP are organized in a master-slave hierarchy as in NTP. The slaves are synchronized to their masters who may be slaves to their own masters. The hierarchy is created and updated automatically by the best master clock (BMC) algorithm, which runs on every clock. When a clock has only one port, it can be master or slave, such a clock is called an ordinary clock (OC). A clock with multiple ports can be master on one port and slave on another, such a clock is called a boundary clock (BC). The top-level master is called the grandmaster clock, which can be synchronized by using a Global Positioning System (GPS) time source. By using a GPS-based time source, different networks can be synchronized with a high-degree of accuracy.

Figure 2.9 shows the unicast negotiation procedure performed between a slave and a peer clock that is selected to be the master clock. The slave clock will request Announce messages from all peer clocks but only request Sync and Delay-Resp messages from the clock selected to be the master clock.

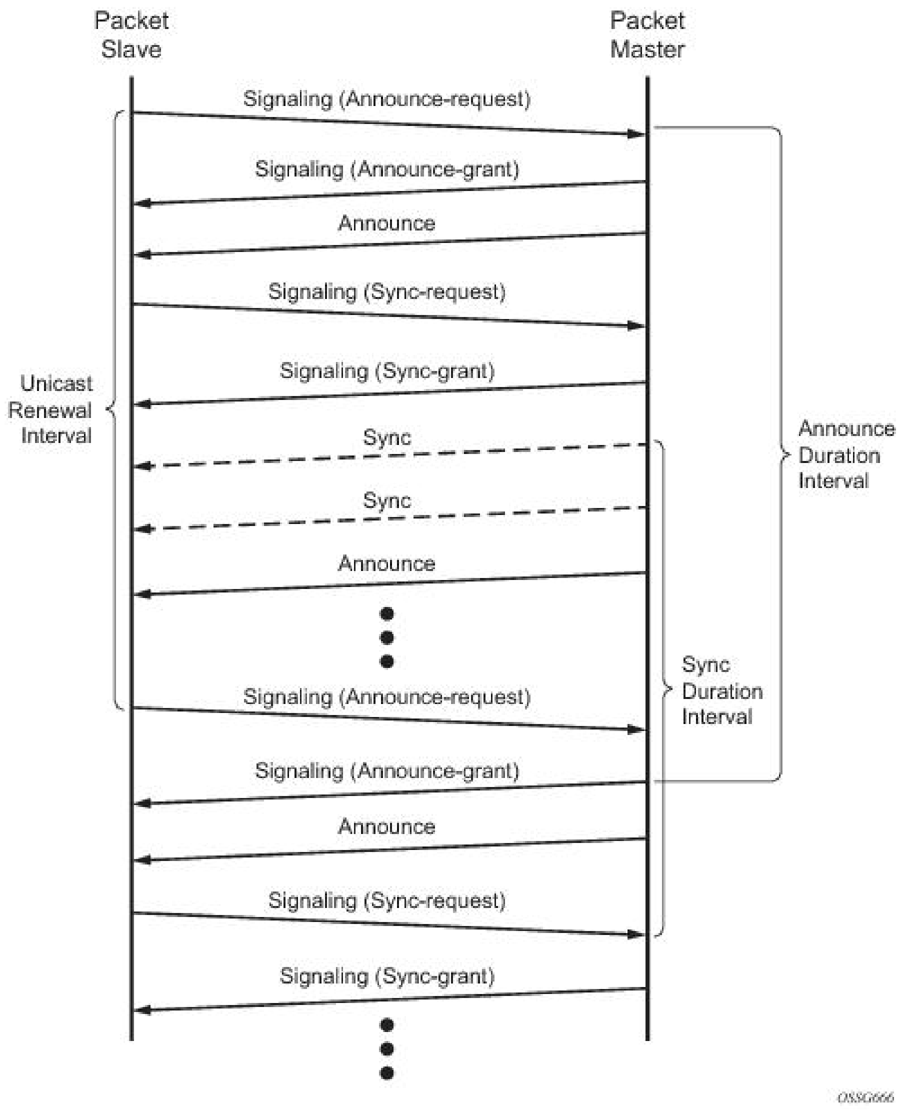


Figure 2.9: PTP Messaging Sequence Between the PTP Slave Clock and PTP Master Clock [23]

The basic synchronization timing computation between the PTP slave and PTP master is shown in Figure 2.10. This figure illustrates the offset of the slave clock referenced to the best master signal during startup.

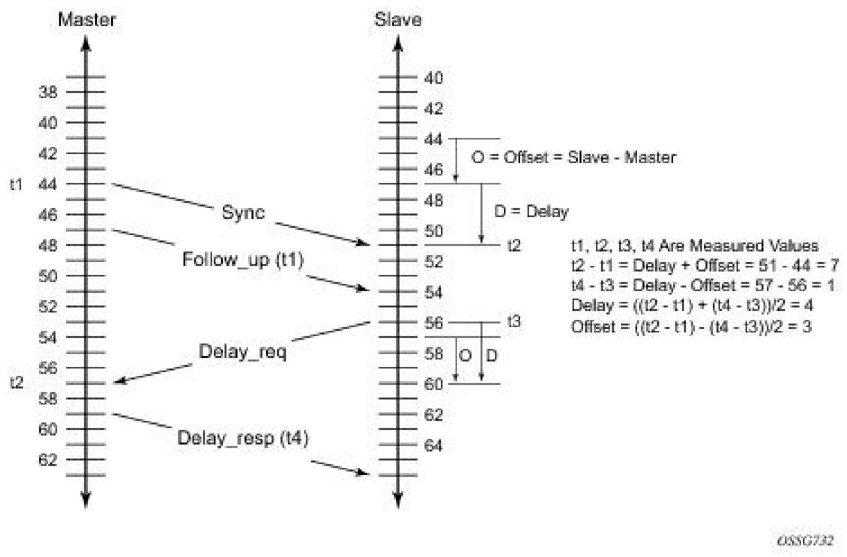

Master
Slave
t1
t2
Sync
Follow_up (t1)
Delay_req
Delay_resp (t4)
O = Offset = Slave - Master
D = Delay
t2
t3
O
D
t1, t2, t3, t4 Are Measured Values
t2 - t1 = Delay + Offset = 51 - 44 = 7
t4 - t3 = Delay - Offset = 57 - 56 = 1
Delay = ((t2 - t1) + (t4 - t3))/2 = 4
Offset = ((t2 - t1) - (t4 - t3))/2 = 3
OSSG732

Figure 2.10: PTP Messaging Sequence Between the PTP Slave Clock and PTP Master Clock [23].

The Best Master Clock Algorithm (BMCA) is the basis of PTP functionality. The BMCA specifies how each clock on the network determines the best master clock in its subdomain of all the clocks it can see, including itself. The BMCA runs on the network continuously and quickly adjusts for changes in network configuration [10].

In an ideal PTP network, the master and slave clock operate at the same frequency. However, drift[2] can occur on the network. In order to compensate drifts by using the time stamp information in the device hardware and follow-up messages (intercepted by the switch) to adjust the frequency of the local clock to match the frequency of the master clock.

NTP to PTP Time Conversion allows you to use Network Time Protocol (NTP) as a time source for PTP. Customers who use PTP for very precise synchronization within a site can use NTP across sites, where precise synchronization is not required.

[2]drift is the frequency difference between the master and slave clock.

2.3 Tools

During the development of this application, several tools have been used both hardware and software. This section summarizes some of the most important tools used.

2.3.1 Hardware

The hardware used for the project belong to two classes. The class part is the processing unit where we focused on using the ARM Architecture and the second class is hardware related with the audio like audio cards, expansion cards, etc.

Processing Unit: Jetson TK1

Jetson TK1 is NVIDIA's embedded Linux development platform featuring a Tegra K1 SOC (CPU+GPU+ISP in a single chip), which costs 192 US$. Jetson TK1 comes pre-installed with Linux4Tegra OS (basically Ubuntu 14.04 with pre-configured drivers). There is also some official support for running other distributions using the mainline kernel [31].

Hardware Features [12]:

- Dimensions: 5" x 5" (127mm x 127mm) board
- Tegra K1 SOC (CPU+GPU+ISP in a single chip, with typical power consumption between 1 to 5 Watts):
  - GPU: NVIDIA Kepler "GK20a" GPU with 192 SM3.2 CUDA cores (upto 326 GFLOPS)
  - CPU: NVIDIA "4-Plus-1" 2.32GHz ARM quad-core Cortex-A15 CPU with Cortex-A15 battery-saving shadow-core
- DRAM: 2GB DDR3L 933MHz EMC x16 using 64-bit data width
- Storage: 16GB fast eMMC 4.51 (routed to SDMMC4)
- mini-PCIe: a half-height single-lane PEX slot (such as for Wifi, SSD RAID, FireWire or Ethernet addon cards)
- SD/MMC card: a full-size slot (routed to SDMMC3)
- USB 3.0: a full-size Type-A female socket

- USB 2.0: a micro-AB female socket (for connecting to a PC, but can also be used as a spare USB 2.0 port using a micro-B male to female Type-A adapter that is sometimes included)
- HDMI: a full-size port
- RS232: a full-size DB9 serial port (routed to UART4)
- Audio: an ALC5639 Realtek HD Audio codec with Mic in and Line out jacks (routed to DAP2)
- Ethernet: a RTL8111GS Realtek 10/100/1000Base-T Gigabit LAN port using PEX SATA: a full-size port that supports 2.5" and 3.5" disks, but is not hot-pluggable. (Turn off the power before plugging in SATA disk drives)
- JTAG: a 2x10-pin 0.1" port for professional debugging
- Power: a 12V DC barrel power jack and a 4-pin PC IDE power connector, using AS3722 PMIC
- Fan: a fan-heatsink running on 12V (to allow safely running intense workloads continuously, but can usually be replaced by a heat-spreader or heatsink)

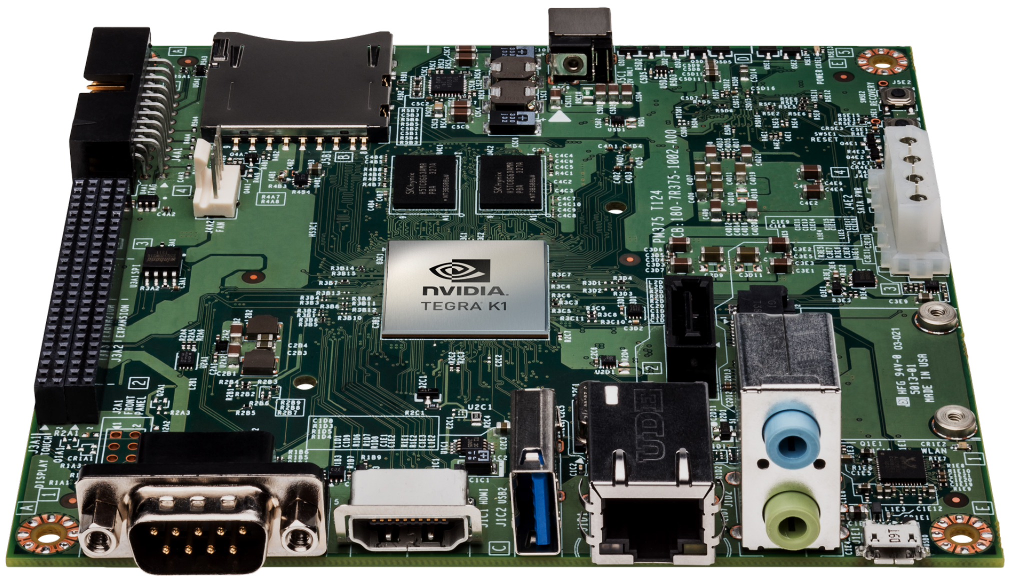


Figure 2.11: Jetson TK1 board.

Processing Unit: Raspberry Pi 2 and 3

The Raspberry Pi is a series of credit card-sized single-board computers developed in the United Kingdom by the Raspberry Pi Foundation with the intent to promote the teaching of basic computer science in schools and developing countries.

Several generations of Raspberry Pi's have been released, although in this project we used the Raspberry Pi 2 model B that was released in February 2015 and Raspberry Pi 3 model B that was released in February 2016. These boards are priced between 20 and 35 US$.

Raspberry PI 2 Hardware Features:

- Architecture: ARMv7 (32-bit) ARMv8 (64/32-bit)
- CPU: 900 MHz 32-bit quad-core ARM Cortex-A7
- GPU Broadcom VideoCore IV @ 250 MHz (BCM2837: 3D part of GPU @ 300 MHz, video part of GPU @ 400 MHz)
- Memory (SDRAM): 1 GB (shared with GPU)
- USB 2.0 ports: 4 (via the on-board 5-port USB hub)
- Video outputs: HDMI (rev 1.3), composite video (3.5 mm TRRS jack)
- Audio outputs: Analog via 3.5 mm phone jack; digital via HDMI
- On-board storage: MicroSDHC slot
- On-board network: 10/100 Mbit/s Ethernet
- Low-level peripherals: 17× GPIO plus the same specific functions, and HAT ID bus
- Power ratings: 800 mA (4.0 W)

Raspberry PI 3 Hardware Features:

Raspberry Pi 3 is an update of the Raspberry PI 2. For that reason most of its characteristics are the same but the main updates are:

- Architecture: ARMv8 (64/32-bit)
- CPU: 1.2 GHz 64-bit quad-core ARM Cortex-A53
- GPU Broadcom VideoCore IV @ 250 MHz (BCM2837: 3D part of GPU @ 300 MHz, video part of GPU @ 400 MHz)[48][49]
- On-board network: 10/100 Mbit/s Ethernet; 802.11n wireless; Bluetooth 4.1

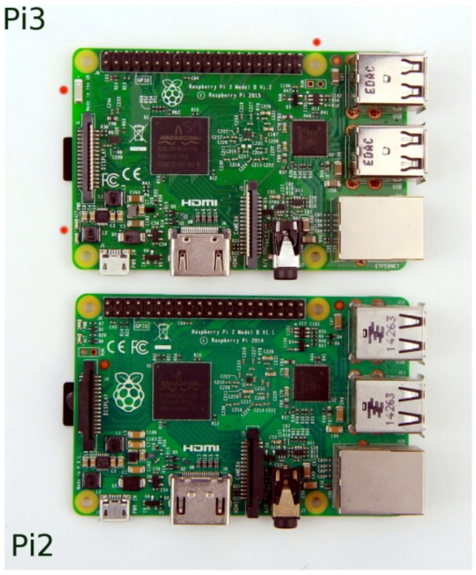


Figure 2.12: Raspberry PI 2 and 3.

Other Hardware

In order to improve the sound capability of the processing units commented previously, a search was made to discover which hardware options are available.

Flexible MiniPCI Express to PCI Bus Adapter

The Flexible MiniPCI Express to PCI Bus Adapter (Adapter) was designed to convert the MiniPCI Express bus into two 32-bit PCI buses. It allows the connection of up to 2 32-bit PCI expansion boards to the motherboard MiniPCI Express connector. This adapter cost 200.

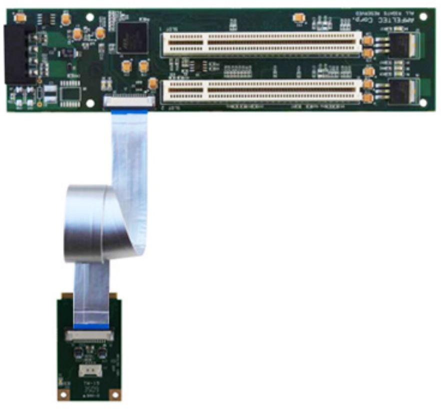


Figure 2.13: Flexible MiniPCI Express to PCI Bus Adapter.

PCI Card Cmedia CMI8738

This card works with the default drivers in Linux and costs less than 30 euros. It gives the possibility to use up to 6 outputs, but it is necessary to use it with the MiniPCI adapter 2.13.

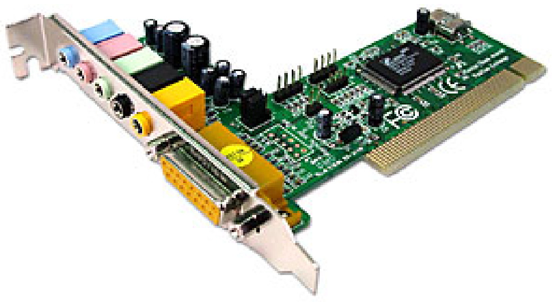


Figure 2.14: PCI Card Cmedia CMI8738.

PCI Card Asus Xonar DS

This card also works with the default drivers in Linux, but it costs 50 euros. The main difference reside in the possibility of using up to 8 outputs, and it is also necessary to use the MiniPCI adapter.

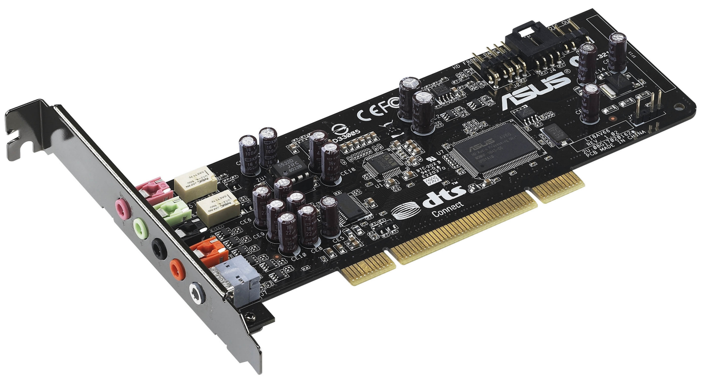


Figure 2.15: PCI Card Asus Xonar DS.

M-Audio Fast Track Ultra

It is an old model that has been discontinued by the manufacturer. The advantage is that also works in Linux with the drivers installed by default and can generate up to 6 outputs by using the USB port.

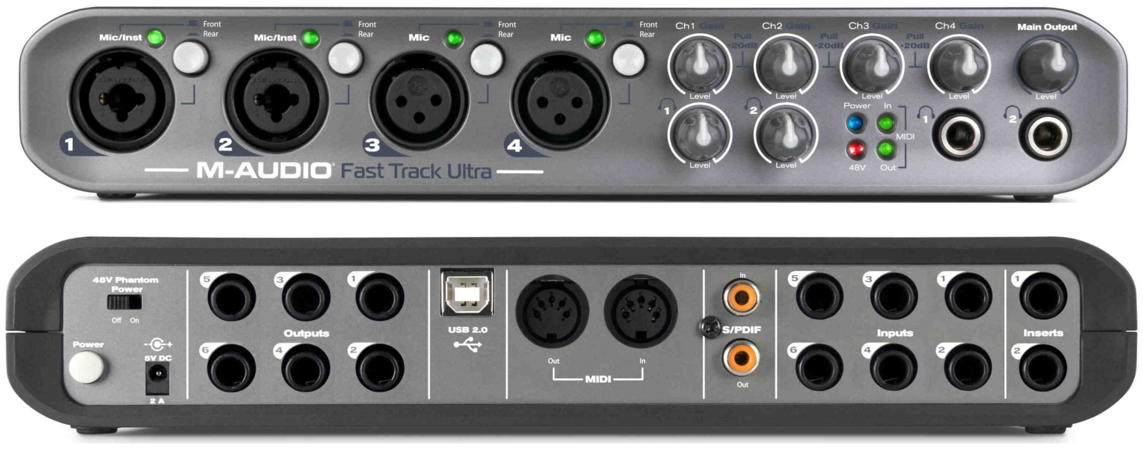


Figure 2.16: M-Audio Fast Track Ultra.

Audio Card USB CSL 7.1

This audio device uses the C-Media audio chipset which works in Linux with the drivers installed by default. It can generate up to 8 outputs and it cost 20 euros on Amazon and it is also connected by USB port.

Figure 2.17: Audio Card USB CSL 7.1.

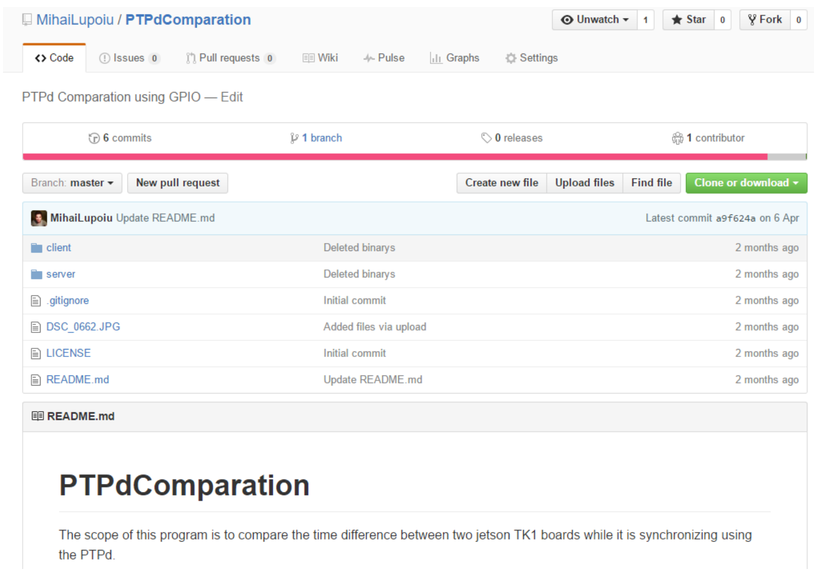
2.3.2 Software

This section briefly summarizes some of the software and libraries used for the development of the project.

Git and GitHub

Git is a version control system that is widely used for software development and other version control tasks. It is a distributed revision control system with an emphasis on speed, data integrity, and support for distributed, non-linear workflows. Git was created by Linus Torvalds in 2005 for development of the Linux kernel.

As with most other distributed version control systems, and unlike most client–server systems, every Git working directory is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server. Like the Linux kernel, Git is free software distributed under the terms of the GNU General Public License version 2.



PTPd Comparation using GPIO — Edit

MihaiLupoiu Update README.md — Latest commit a9f624a on 6 Apr

client	Deleted binarys	2 months ago
server	Deleted binarys	2 months ago
.gitignore	Initial commit	2 months ago
DSC_0662.JPG	Added files via upload	2 months ago
LICENSE	Initial commit	2 months ago
README.md	Update README.md	2 months ago

README.md

PTPdComparation

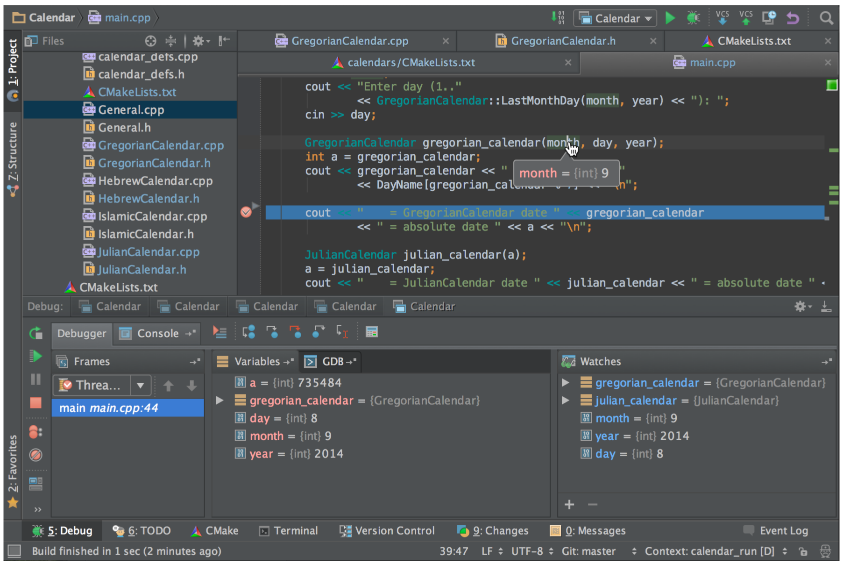
The scope of this program is to compare the time difference between two jetson TK1 boards while it is synchronizing using the PTPd.

Figure 2.18: Github PTPdComparation Repository

GitHub is a Git repository hosting service. While Git is a command line tool, GitHub provides a Web-based graphical interface. It also provides access control and several collaboration features, such as a wikis and basic task management tools for every project.

Clion IDE

Clion is an Integrated Development Environment (IDE) that provides comprehensive facilities for developing for C/C++. Some of those facilities are a source code editor, build automation, debugger, intelligent code completion, version control system and more [5].



```
cout << "Enter day (1.."
        << GregorianCalendar::LastMonthDay(month, year) << "): ";
cin >> day;

GregorianCalendar gregorian_calendar(month, day, year);
int a = gregorian_calendar;
cout << gregorian_calendar << "
        << DayName[gregorian_cal                     n";

cout << "    = GregorianCalendar date " << gregorian_calendar
        << " = absolute date " << a << "\n";

JulianCalendar julian_calendar(a);
a = julian_calendar;
cout << "    = JulianCalendar date " << julian_calendar << " = absolute date "
```

```
a = {int} 735484
gregorian_calendar = {GregorianCalendar}
day = {int} 8
month = {int} 9
year = {int} 2014
```

```
gregorian_calendar = {GregorianCalendar}
julian_calendar = {JulianCalendar}
month = {int} 9
year = {int} 2014
day = {int} 8
```

```
Build finished in 1 sec (2 minutes ago)
```

Figure 2.19: Clion IDE.

Sublime Text 3

Sublime Text is a cross-platform source code editor. It natively supports many programming languages and markup languages, and its functionality can be extended by users with plugins, typically community-built and maintained under free-software licenses [22].

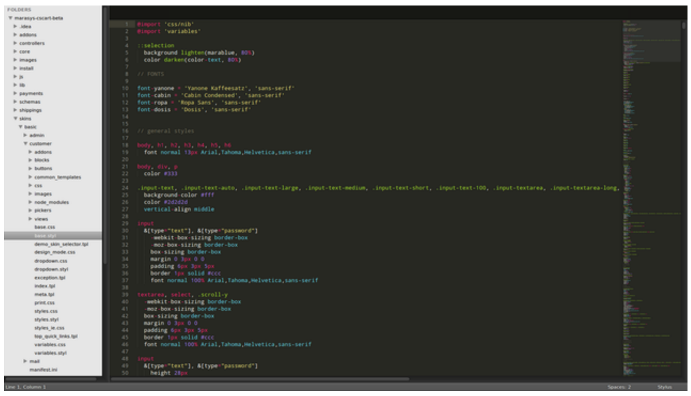
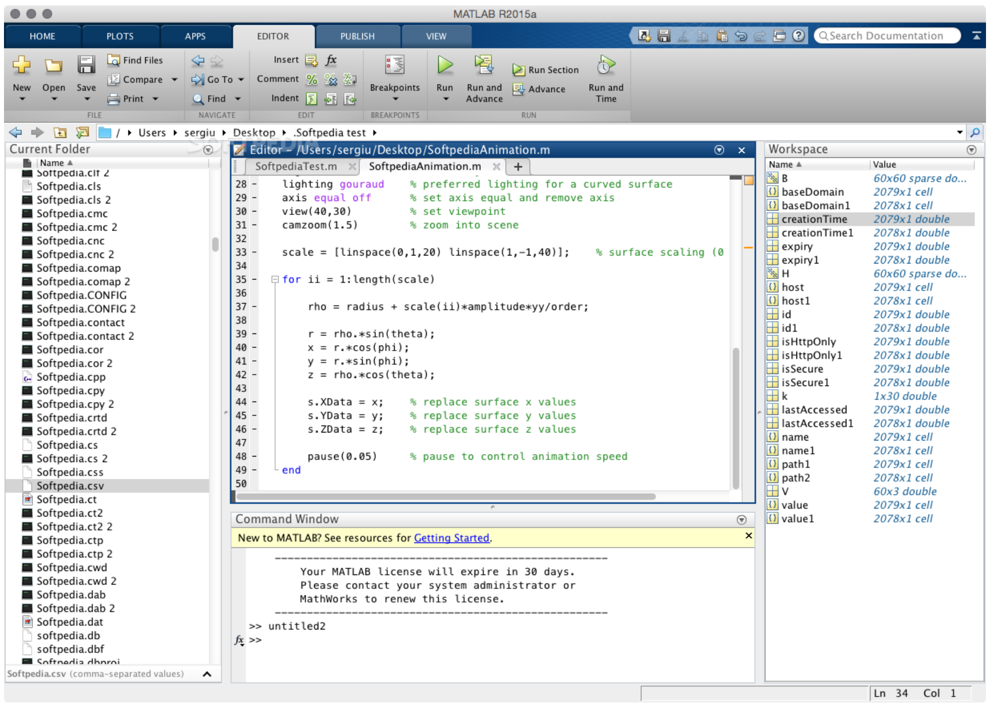


Figure 2.20: Sublime Text.

Matlab

MATLAB is a high-level language and interactive environment for numerical computation, visualization, and programming. The use of MATLAB was mainly for developing algorithms and a prototype application [32].



Softpedia.clf 2
Softpedia.cls
Softpedia.cls 2
Softpedia.cmc
Softpedia.cmc 2
Softpedia.cnc
Softpedia.cnc 2
Softpedia.comap
Softpedia.comap 2
Softpedia.CONFIG
Softpedia.CONFIG 2
Softpedia.contact
Softpedia.contact 2
Softpedia.cor
Softpedia.cor 2
Softpedia.cpp
Softpedia.cpy
Softpedia.cpy 2
Softpedia.crtd
Softpedia.crtd 2
Softpedia.cs
Softpedia.cs 2
Softpedia.css
Softpedia.csv
Softpedia.ct
Softpedia.ct2
Softpedia.ct2 2
Softpedia.ctp
Softpedia.ctp 2
Softpedia.cwd
Softpedia.cwd 2
Softpedia.dab
Softpedia.dab 2
Softpedia.dat
softpedia.db
softpedia.dbf
Softpedia.dbproj

Softpedia.csv (comma-separated values)

```matlab
    lighting gouraud    % preferred lighting for a curved surface
    axis equal off      % set axis equal and remove axis
    view(40,30)         % set viewpoint
    camzoom(1.5)        % zoom into scene

    scale = [linspace(0,1,20) linspace(1,-1,40)];    % surface scaling (0

    for ii = 1:length(scale)

        rho = radius + scale(ii)*amplitude*yy/order;

        r = rho.*sin(theta);
        x = r.*cos(phi);
        y = r.*sin(phi);
        z = rho.*cos(theta);

        s.XData = x;    % replace surface x values
        s.YData = y;    % replace surface y values
        s.ZData = z;    % replace surface z values

        pause(0.05)     % pause to control animation speed
    end
```

Name	Value
B	60x60 sparse do...
baseDomain	2079x1 cell
baseDomain1	2078x1 cell
creationTime	2079x1 double
creationTime1	2078x1 double
expiry	2079x1 double
expiry1	2078x1 double
H	60x60 sparse do...
host	2079x1 cell
host1	2078x1 cell
id	2079x1 double
id1	2078x1 double
isHttpOnly	2079x1 double
isHttpOnly1	2078x1 double
isSecure	2079x1 double
isSecure1	2078x1 double
k	1x30 double
lastAccessed	2079x1 double
lastAccessed1	2078x1 double
name	2079x1 cell
name1	2078x1 cell
path1	2079x1 cell
path2	2078x1 cell
V	60x3 double
value	2079x1 cell
value1	2078x1 cell

New to MATLAB? See resources for Getting Started.

```
    ---------------------------------------------------
        Your MATLAB license will expire in 30 days.
        Please contact your system administrator or
        MathWorks to renew this license.
    ---------------------------------------------------
>> untitled2
>>
```

Figure 2.21: Matlab.

C libraries

The development of the application includes the use of several libraries that are listed below:

- Advanced Linux Sound Architecture (ALSA) provides audio functionality to the Linux operating system, and the **libasound2** is the interface to the ALSA drivers [1].

- **Libconfig** is a simple library for processing structured configuration files. This file format is more compact and more readable than XML. And unlike XML, it is type-aware, so it is not necessary to do string parsing in application code [13].

- **Sockets** is used to communicate with the server[21].

- **OpenMP** is used to create a Multithread application [18].

- **FFTW** is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data [8].

Python modules

- **ServerSocket** is a module for python that simplifies the task of writing network servers. It is mainly used for the server side [20].
- **ConfigParser** is a module to read and parse the configuration for a python program just like Libconfig for C/C++ [7].

PTP Implementation

The Precision Time Protocol implementation that has been used for the synchronization were two. The first one was the ptpdv2 from 2012 that is available in the Advanced Package Tool on ubuntu. The second one was the ptpd implementation that is available on github to download and compile.

NEON

An ARM processor is one of a family of CPUs based on the RISC (reduced instruction set computer) architecture developed by Advanced RISC Machines (ARM).

ARM makes 32-bit and 64-bit RISC multi-core processors. RISC processors are designed to perform a smaller number of types of computer instructions so that they can operate at a higher speed, performing more millions of instructions per second (MIPS). By stripping out unneeded instructions and optimizing pathways, RISC processors provide outstanding performance at a fraction of the power demand of CISC (complex instruction set computing) devices.

ARM processors are extensively used in consumer electronic devices such as smartphones, tablets, multimedia players and other mobile devices, such as wearables. Because of their reduced instruction set, they require fewer transistors, which enable a smaller die size for the integrated circuitry (IC). The ARM processors smaller size reduced complexity and lower power consumption makes them suitable for increasingly miniaturized devices.

ARMv6 architecture introduced a small set of SIMD instructions, operating on multiple 16-bit or 8-bit values packed into standard 32-bit general purpose registers. This permits certain operations to execute twice or four times as quickly, without implementing additional computation units. The mnemonics for these instructions are recognized by having 8 or 16 appended to the base form, indicating the size of data values operated on [2].

ARMv7 architecture introduced the Advanced SIMD extension which extends the SIMD concept by defining groups of instructions operating on vectors stored in 64-bit D, doubleword, registers and 128-bit Q, quadword, vector registers.

Figure 2.22 shows how the VADD.I16 Q0, Q1, Q2 instruction performs a parallel addition of eight lanes of 16-bit elements from vectors in Q1 and Q2, storing the result in Q0.

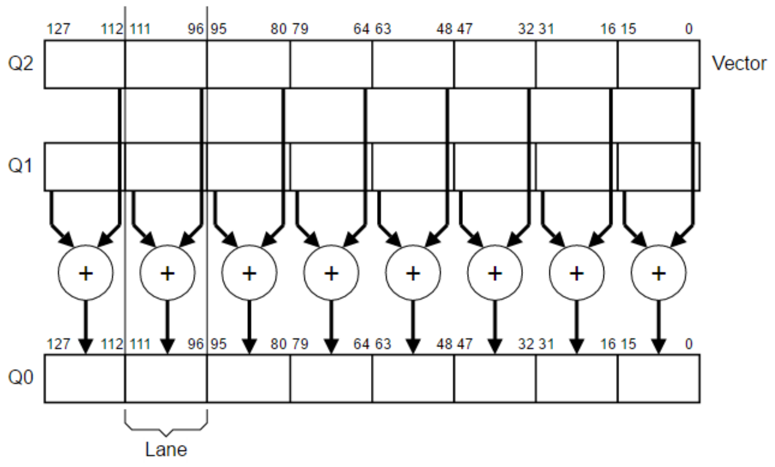


Figure 2.22: 8-way 16-bit integer add operation[2].

The implementation of the Advanced SIMD extension used in ARM processors is called NEON, and this is the common terminology used outside architecture specifications. NEON technology is implemented on all current ARM Cortex-A series processors. NEON instructions are executed as part of the ARM or Thumb instruction stream. This simplifies software development, debugging, and integration compared to using an external accelerator. Traditional ARM or Thumb instructions manage all program flow and synchronization [24, 15]. The NEON instructions perform:

- Memory accesses.
- Data copying between NEON and general purpose registers.
- Data type conversion.
- Data processing.

Chapter 3

Design and Implementation

3.1 Context of the Application

As mentioned before, our main objective is to develop a distributed application capable of reproducing sound that gives the user a special location of the source's position. All the actual implementations of this type of applications are based on a centralized system which is expensive and not flexible to add or remove speakers.

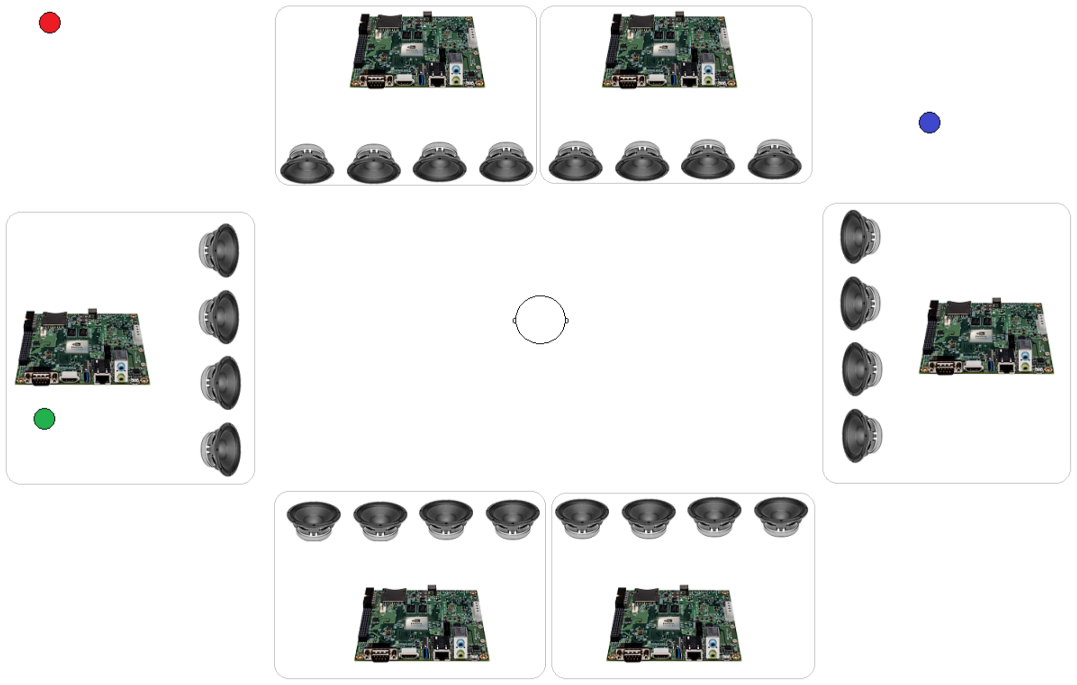


Figure 3.1: Example of the system's distribution.

This application distributes the process among several nodes in order to reduce computational cost and to be totally flexible (a modular system). We connect as many nodes as we desire and thus, we can modify the initial design without the need to change the whole system, only the required nodes. In the Figure 3.1 we can observe an example of the system, where each Nvidia Jetson TK1 board has four speakers connected and each colored dot represents the source position from where the user should perceive the sound.

In this chapter, we explain the synchronization method used in order to synchronize the acoustic nodes, how the communication between nodes was achieved, how is the configuration carried out for each node, and what optimizations were performed in the operations that involve the Wave Field Synthesis system. This is carried out in order to connect and process as many speakers and virtual sounds sources as possible. We also explain some details about the internal structure and implementations.

3.2 Synchronization

A distributed system consists of several acoustic nodes that exchange information collaboratively to achieve a common goal. In this case, the common goal is to develop a Wave Field Synthesis application, which requires maximum synchronization between nodes. If the nodes are not synchronized, then the sound would be translated from one node to other we will perceive a pause in the reproduction or an echo.

In order to solve that problem we use the Precision Time Protocol because it can operate with normal Ethernet network traffic on a Local Area Network (LAN) with switches, while maintaining synchronization accuracy to the sub–microseconds if it is used with hardware support [10]. This accuracy is better than the one that is achieved with the Network Time Protocol (NTP) [17, 4, 6].

Protoco	**Media**	**Sync Accuracy**
NTP	Ethernet	50-100 milliseconds
PTP	Ethernet	20-100 nanoseconds

Table 3.1: Syncronization accurancy using NTP and PTP [10]

An important step before using the protocol is to check the compatibility of the Ethernet interface to the protocol by using the command **ethtool** [10].

If we want to use the PTP protocol via hardware, it is important to check that the ethernet card has these characteristics:

```
SOF_TIMESTAMPING_RAW_HARDWARE
SOF_TIMESTAMPING_TX_HARDWARE
SOF_TIMESTAMPING_RX_HARDWARE
```

While if we want to use the protocol PTP via software, the ethernet should have this characteristics:

```
SOF_TIMESTAMPING_SOFTWARE
SOF_TIMESTAMPING_TX_SOFTWARE
SOF_TIMESTAMPING_RX_SOFTWARE
```

If we run the ethtool command in the Nvidia Jetson TK1 board the result can be observed below:

```
#]~$ ethtool -T eth0
Time stamping parameters for eth0:
Capabilities:
software-transmit     (SOF_TIMESTAMPING_TX_SOFTWARE)
software-receive      (SOF_TIMESTAMPING_RX_SOFTWARE)
software-system-clock (SOF_TIMESTAMPING_SOFTWARE)
PTP Hardware Clock: none
Hardware Transmit Timestamp Modes: none
Hardware Receive Filter Modes: none
```

What it indicates is that the Nvidia Jetson TK1 board only is capable of doing a PTP synchronization via software on integrated ethernet card. Therefore, this synchronization could be worse since all the process is done in the CPU.

An implementation of the protocol is available in the Advanced Packaging Tool for any system that is based on Debian. But it can also be downloaded and compiled from the official repository [19].

Once configured the Nvidia Jetson TK1 board in order to use the PTP protocol via software it is necessary to assess the feasibility of implementing WFS. To do this, it will perform multiple measurements of time in both boards in order to evaluate its delay. These measurements are shown in the next two sections.

3.3 Initial Design

The first design was a system composed of two Nvidia Jetson TK1 boards, where we can observe the accuracy that is obtained when one of the boards is configured with a role (Figure 3.2). One of the boards has the Master-Server role and its clock will be used as a reference in the system, while the other board takes the role of Slave-Client.

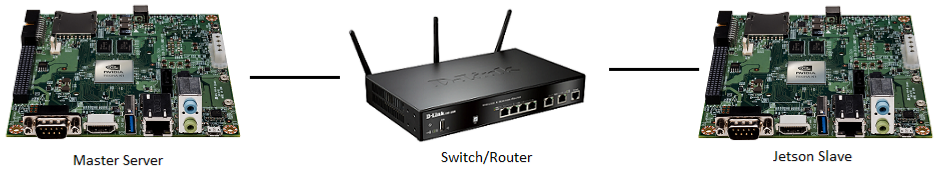


Figure 3.2: System composed of two Nvidia Jetsons TK1.

In Figure 3.3,it can be seen as a first result that, on average, between the two Nvidia Jetsons TK1 boards there is a lag of 30 milliseconds. This value makes it unfeasible to an acoustic wave synthesis system where the maximum deviation that can be tolerate is 5 milliseconds [29]. In the case the gap is greater than 5 milliseconds, a listener perceive a sound with more echo instead of a single sound. Therefore, we must find a design that reduces the time synchronization.

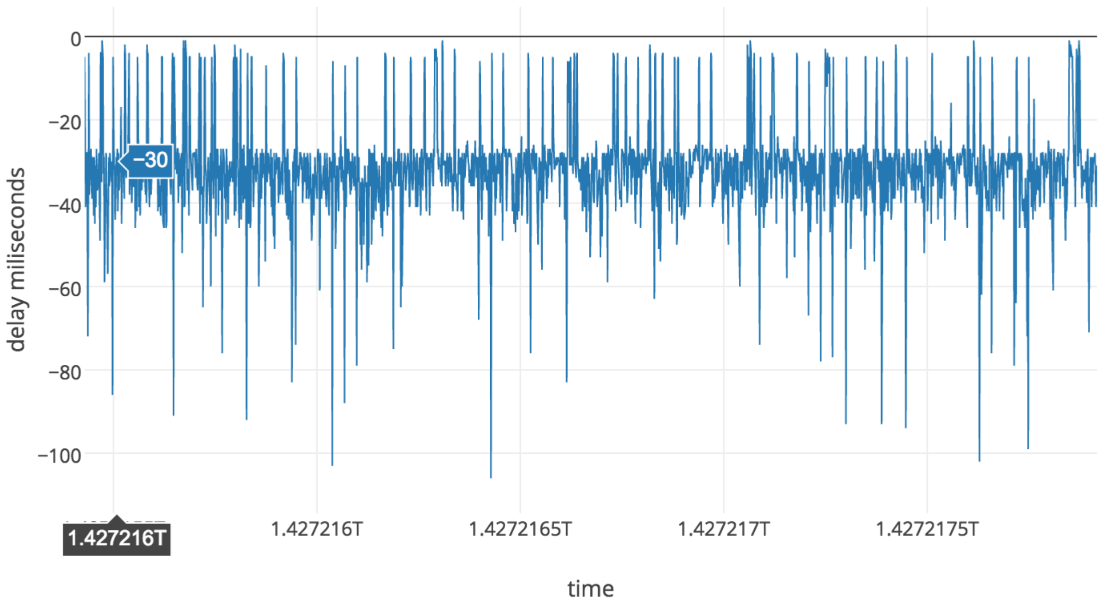


Figure 3.3: Time synchronization between the Nvidia Jetson TK1 Master-Server and Slave-Client.

3.4 Second and Final Design

The second design is a system composed of one separate machine and two Nvidia Jetson TK1 boards that are connected to the same network. The Figure 3.4 shows the system design where the clock represents the machine that provides the time reference and will play the role of Master-Server. In the network there can also be a general purpose computer that will be used by a user to run the application of WFS as a orchestra conductor. Its only role is to broadcast to each Nvidia Jetson tk1 board the position of the sound sources in the WFS system.

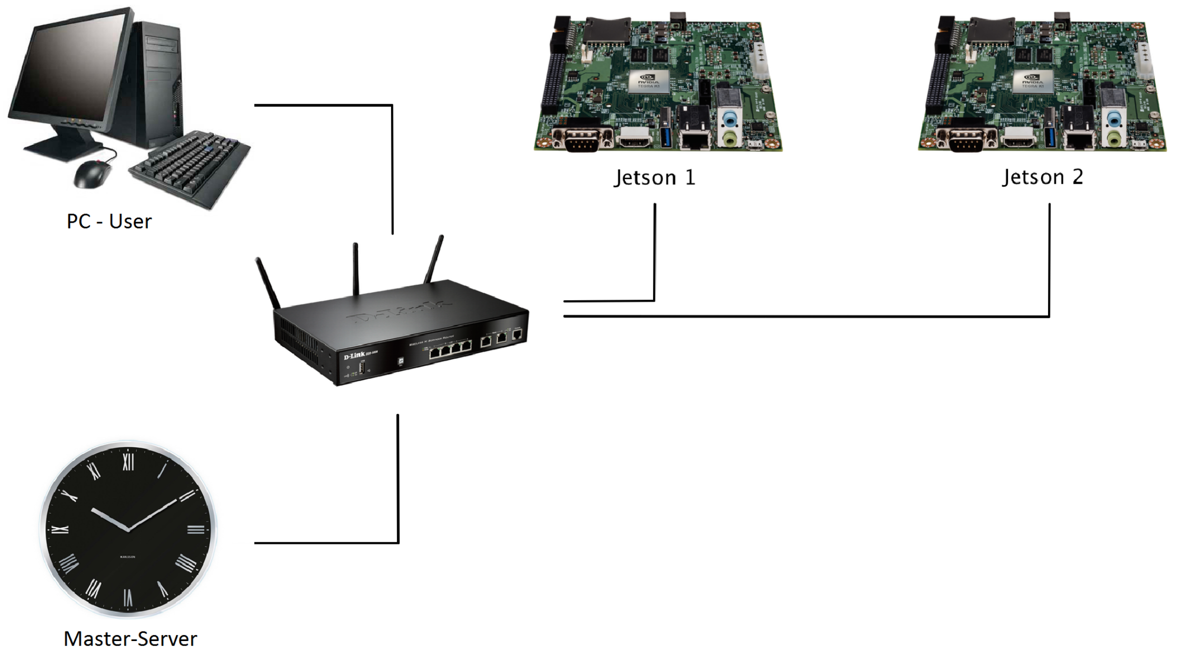


Figure 3.4: System design composed of two Nvidia Jetsons TK1, a machine used as a time reference (master-server) and a general purpose computer.

After mounting the new design, we execute the PTP protocol in the Master-Server to share its time. Then, we execute this same protocol in the Nvidia Jetsons TK1 boards, which will act as Slave-Client. After a few seconds after running the protocol in the boards, they will start to synchronize their time with the Master-Server.

The fact that in this distribution both Nvidia Jetson TK1 boards take the same role as Slave-Client, makes both to be out of phase with respect to the Master-Server, but they are synchronized between them. Figure 3.5 shows the offset in milliseconds between each board and Server.

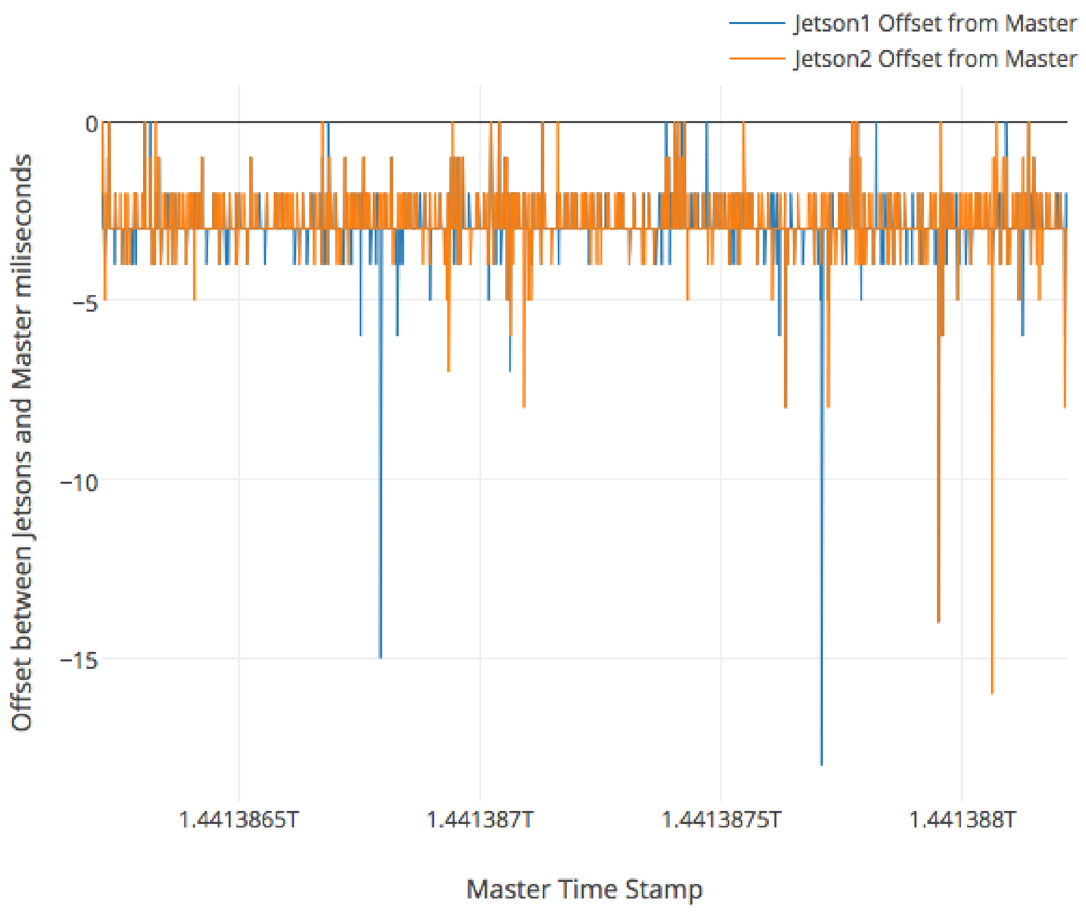


Figure 3.5: Time difference between the Nvidia Jetsons Tk1 boards in client mode Slave respect to Master-Server.

Moreover, Figure 3.6 shows the time difference between the two Nvidia Jetsons TK1 boards is most of the time under 5 milliseconds.

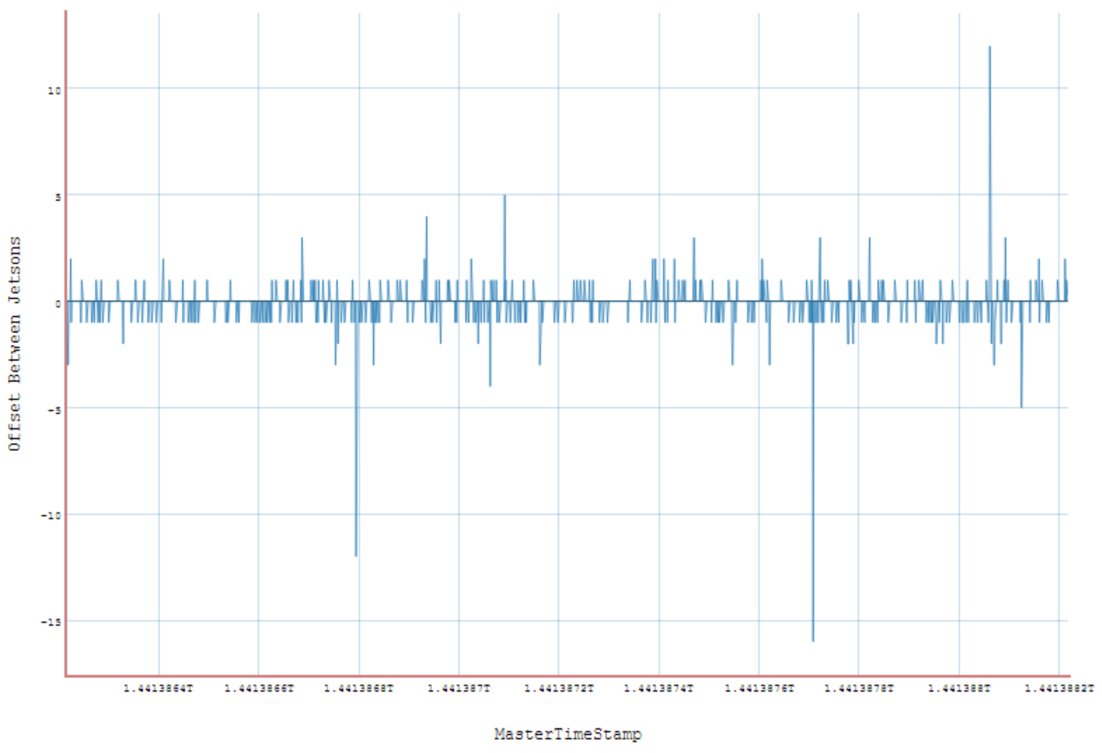


Figure 3.6: Synchronization time difference between the two Nvidia Jetsons TK1 boards.

In order to check the synchronism between the Jetsons boards, we have taken 2,000 samples clock each and calculated the difference. As shown in Table 13.2, more than 80% of the time are fully synchronized and more than 15% of the time are synchronized with one millisecond of difference. This data shows the high synchronization between the two Jetsons. Despite these good results, a small percentage (0.15%) of time remains when the Jetsons are not synchronized.

Milliseconds difference	16	12	5	4	3	2	1	0
Repetitions	1	2	2	2	14	34	303	1642
Percentage %	0,05	0,1	0,1	0,1	0,7	1,7	15,15	82,1

Table 3.2: Synchronization Time between Nvidia Jetson TK1 boards

The results were achieved using the network layer in order to check the time of each Nvidia Jetson TK1, which adds some uncontrollable delay because of the communication. In order to have absolute control of the environment, the application was afterwards implemented using the GPIO connection.

Firstly, it was necessary to know the time that it takes the GPIO to send and receive an impulse. We measured the time it has taken to send and receive an impulse from one pin to another, which was approximately 1 millisecond.

After knowing the time it takes to send and receive an impulse using the GPIO, we implemented an application that consists of two parts.

The first one is the Server. This can be executed in any machine, but not on the Jetson TK1 boards. Its role is to send a time to start comparing the time they have with the board. The actual time to start is hardwired to 5 seconds. Also the number of boards that can be connected is hardwired to 2, but it can be incremented to as many as it is needed.

The second part is the Client. The **IP** and **PORT** of the server the clients has to connect is hardwired and at the moment is 192.168.1.12:150. In order for the application to work there has to be at least 2 clients. The reason is because the system will use the GPIO to send a signal on the GPIO pin 160 when it reaches X time to wake up. In this case X = 500 milliseconds. At the same time, the system will receive a signal at the GPIO pin 161, from the other board when has reached the same time, and will compare the difference between them [11].

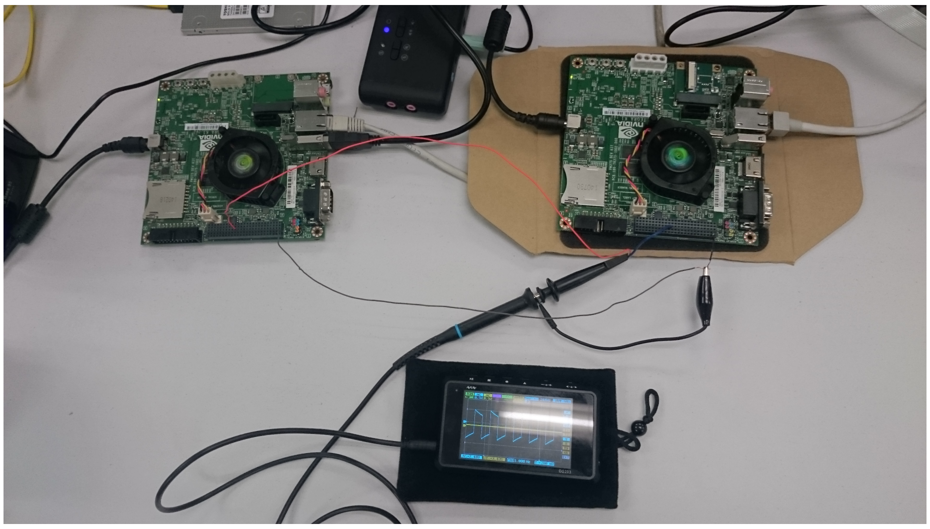


Figure 3.7: Setup of the application to compare the time difference between two jetson TK1 boards while it is synchronizing using the PTPd.

In the Figure 3.8 we can observe the result obtained by using the same number of samples, 2000. The offset between the Nvidia Jetson TK1 boards is approximately 2 milliseconds of which we should note as mentioned before that 1 milliseconds corresponds to write and read to the GPIO. Thus, the average delay between the boards is approximately 1 millisecond.

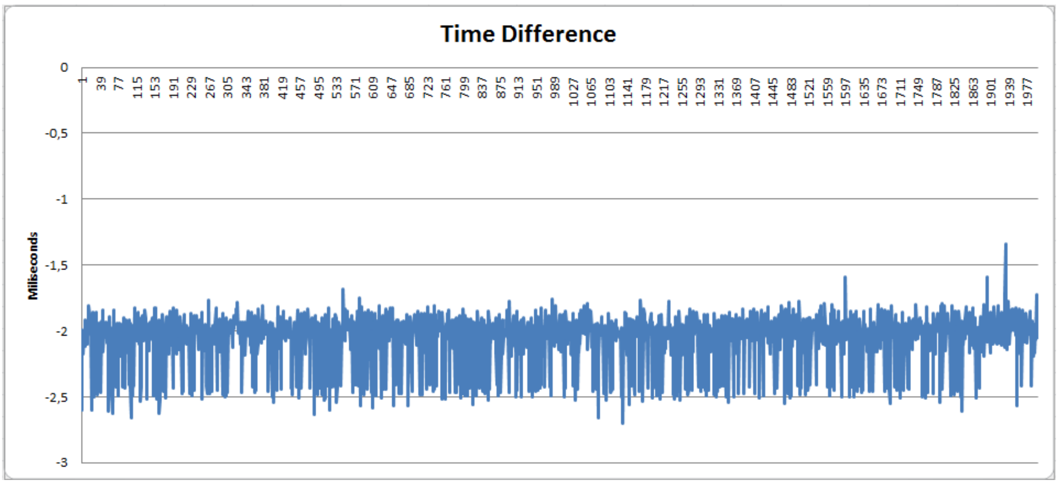


Figure 3.8: Synchronization time difference between the two Nvidia Jetsons TK1 boards with the GPIO application with 2000 samples.

If we increase the number of samples to analyze to almost 10000, as shown in Figure 3.9, we can observe that there are four points in which offset between the boards increases drastically. The 4 points offset vary from 200 milliseconds to 800 milliseconds, but it is rectified in by the PTP protocol.

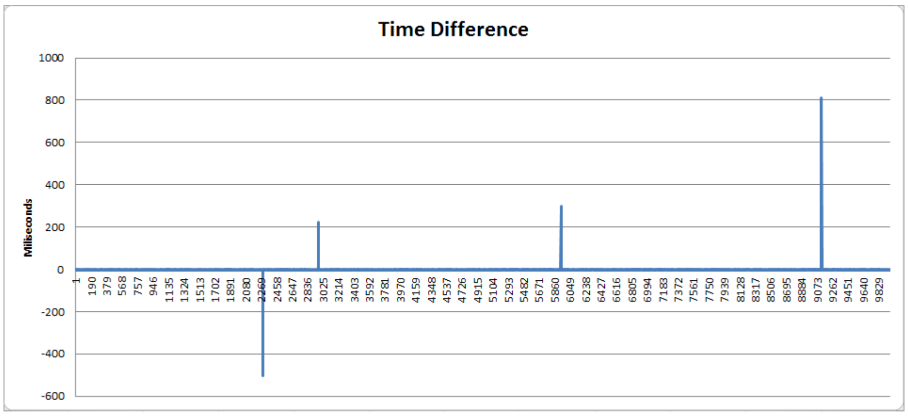


Figure 3.9: Synchronization time difference between the two Nvidia Jetsons TK1 boards with the GPIO application.

3.5 Implementation Details

The application is divided in two parts where one is the server and the other one is the client. The server as mentioned in the previous section is compared to the orchestra conductor. Its only role is to broadcast the position of the sounds. On the other hand the clients can be compared to the instruments of an orchestra. They will have to start or stop playing when the orchestra conductor tells them to.

3.5.1 Message Passing

In order to communicate the server with the clients we have used the socket TCP connection due to its reliability. That is also a problem because it's more heavyweight. If a message is received in the wrong order a resend requests will to be sent. If there is a connection lost, the server will request the lost part, so there is no corruption while transferring a message. The decision of using the TCP connection was also made with the idea of developing a streaming reproduction in the near future.

After the connection was made by the client using the IP and PORT number of the server, the server will send an "action" that is "identification" to identify each client with a number:

```
[SERVER]:"Action:identification,ID:1"
```

When the server decides to start play, it will send a message where the action is "start", the time when it will start to play, the position of a song where to start, or a value of "-1" if it will start all the songs to that same position.

```
[SERVER]:"Action:start,StartTime:123456789123,Song:-1,
SongPosX:11,SongPosY:0"
```

If the server changes the position of one song it will have to send the action "song", with the song number that will have to be moved and the position. In this case the "-1" value is also available to move all the sounds to the same position.

```
[SERVER]:"Action:song,StartTime:0,Song:1,SongPosX:5,SongPosY:0"
```

When the server wants to stop the reproduction process, it will send the action "exit".

```
[SERVER]:"Action:exit"
```

3.5.2 Configuration File

In order to facilitate a personalized configuration of each node the “Libconfig” library was used. It is a simple library for processing structured configuration files. It is compact, type-aware, so it is not necessary to do any string parsing in application code and easy to modify [14].

An example of a configuration file for the application would be:

```
client ={
    card = {
        pcm_name = "plughw:1,0";
        frame_Rate = 44100;
        pcm_buffer_size = 2048;
        pcm_period_size = 256;
    }
    sound = {
        word_length = 50;
        sound_folder = "../../bin/sound/";
        sounds_number = 4;
        sounds_list = ( { file_name = "001_piano.wav";},
            { file_name = "voz4408.wav";},
            { file_name = "001_bajo.wav";},
            { file_name = "001_bateriabuena.wav";}
        );
    }
    speakers = {
            speakers_number = 2;
            chanels_number = 2;
            speakers_position = ( { posX = 2.0; posY = 0.0; angle = 0.0;
                { posX = 4.0; posY = 0.0; angle = 180.0;}
            );
    }
}
```

As shown in the previous configuration file example there are three parts “card”, “sound” and “speakers”. Each part corresponds to a specific configuration in the application.

The first part is the audio card. In order to reproduce sound we have used the Advanced Linux Sound Architecture (ALSA), which needs some configuration parameters. The first parameter is the audio card that will be used. In order to do that we use “plughw:1,0”, where the first digit represents the card to use and the second digit the device type. To see the cards and devices available we use the “aplay -l” command with superuser privileges. The output of this command in the Nvidia Jetson TK1 is the follow:

```
~$ sudo aplay -l
**** List of PLAYBACK Hardware Devices ****
card 0: Tegra [HDA NVIDIA Tegra], device 3: HDMI 0 [HDMI 0]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 1: tegrart5639 [tegra-rt5639], device 0: rt5639 PCM rt5639-aif1-0 []
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 1: tegrart5639 [tegra-rt5639], device 1: SPDIF PCM dit-hifi-1 []
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 1: tegrart5639 [tegra-rt5639], device 2: BT SCO PCM dit-hifi-2 []
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 1: tegrart5639 [tegra-rt5639], device 3: VOICE CALL PCM rt5639-aif1-
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 1: tegrart5639 [tegra-rt5639], device 4: BT VOICE CALL PCM dit-hifi-
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

The second parameter is the frame rate in which we want to reproduce, in this case 44.100.

The third parameter is the size in bytes that we want to use. In this case 2048, this equals to an array of 512 integers. The last parameter is the period size which indicates ALSA the frequency to update the status of its buffer. In this case there would be 8 interruptions to check the status of the buffer and update it if necessary.

The second part corresponds to the sound that the application will use. It is necessary to indicate the length of the files name, the location of all the sounds, the number of sounds and the name of audio files in a list format.

The third and last part correspond to the configuration of the speakers. In this case, the number of speakers that are connected, the number of channels we want to use an the position of the speakers. For the position of the speakers we use a the coordinated X,Y position and the angle where the speaker are oriented.

In Figure 3.10 we can observe that all the angles of the speakers that are faced upside down have negative values (-10, -30, -45, -90, -125, -180). All the rest of the speakers have positive values, but 180^{o} and -180^{o} or 0^{o} and -0^{o} would be the same. This is used later in the application when the Wave Field Synthesis is running and determines which speaker is active.

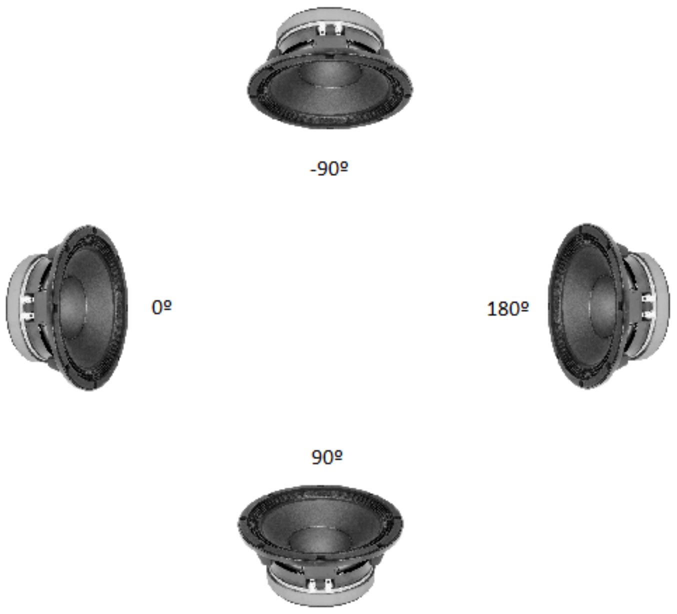


Figure 3.10: Angle orientation of the speakers.

3.5.3 Apply Wave Field Synthesis

In order to make the application and synthesize a wave field from a virtual position of a sound it is necessary to know the position of the systems speakers, the virtual position of where we want to perceive the sound and the orientation of the speakers. The system will only work if the virtual position of the sound is located behind the speakers between 90^o and -90^o as shown in the Figure 3.11.

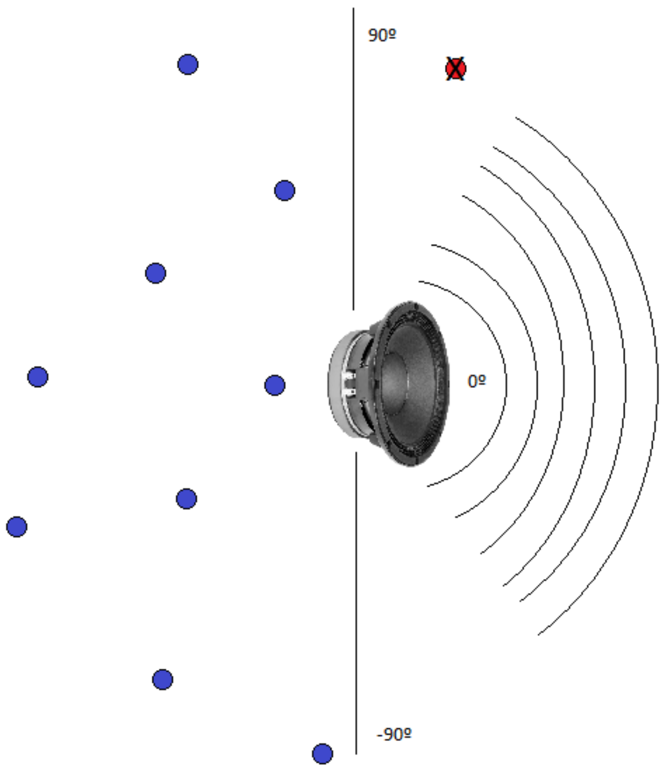


Figure 3.11: Wave Field Shyntesis Example.

Every blue dot of the Figure 3.11 is sound position that will be reproduced, but the red dot will not, because it is out of speakers range. For each blue dot, the system will calculate its distance to the speaker in order to achieve the delay it has to introduce before reproducing an amplitude at which it will reproduce.

Once it is calculated it will start to prepare a buffer for each sound. After all the sounds are calculated, it will sum the buffers in only one for each channel. Then the final buffer must be sent to ALSA drive. But before it is sent to ALSA we apply a filter by using the convolution process which has to be done for each channel.

3.5.4 Operation sequence of the application

The operation sequence of the application, Figure 3.12, starts by reading the configuration file of the system. Once the configuration file has been read successfully, the client will connect to the server, and receives an identification number.

Afterwards the client will load the songs that have been described in the configuration file and waits for the start reproducing message. Once the message has been received it will create a new process which will be in charge of reproducing. This process will also receive the virtual position of the sounds and the exact moment it will have to start to reproduce.

The second process will initialize the audio card with the configuration file. It will also create the buffers and apply the Wave Field Synthesis using the virtual position received from the server. After applying the Wave Field Synthesis, it sums the separate buffers songs in one buffer per channel and afterwards it will send it to ALSA. But before sending the resulting buffer to ALSA, it will apply a filter using the convolution process. Once the process of applying the WFS has ended, it will send the buffer to ALSA. This process will repeat until the largest song has finished or the first process has sent an exit message from the server.

Meanwhile the second process is producing; the first process will wait indefinitely for any more messages from the server. If it receives a new message it will process the message and while this message is different than an exit action, it means there is a new virtual position of the song. Thus, the information will go to the second process.

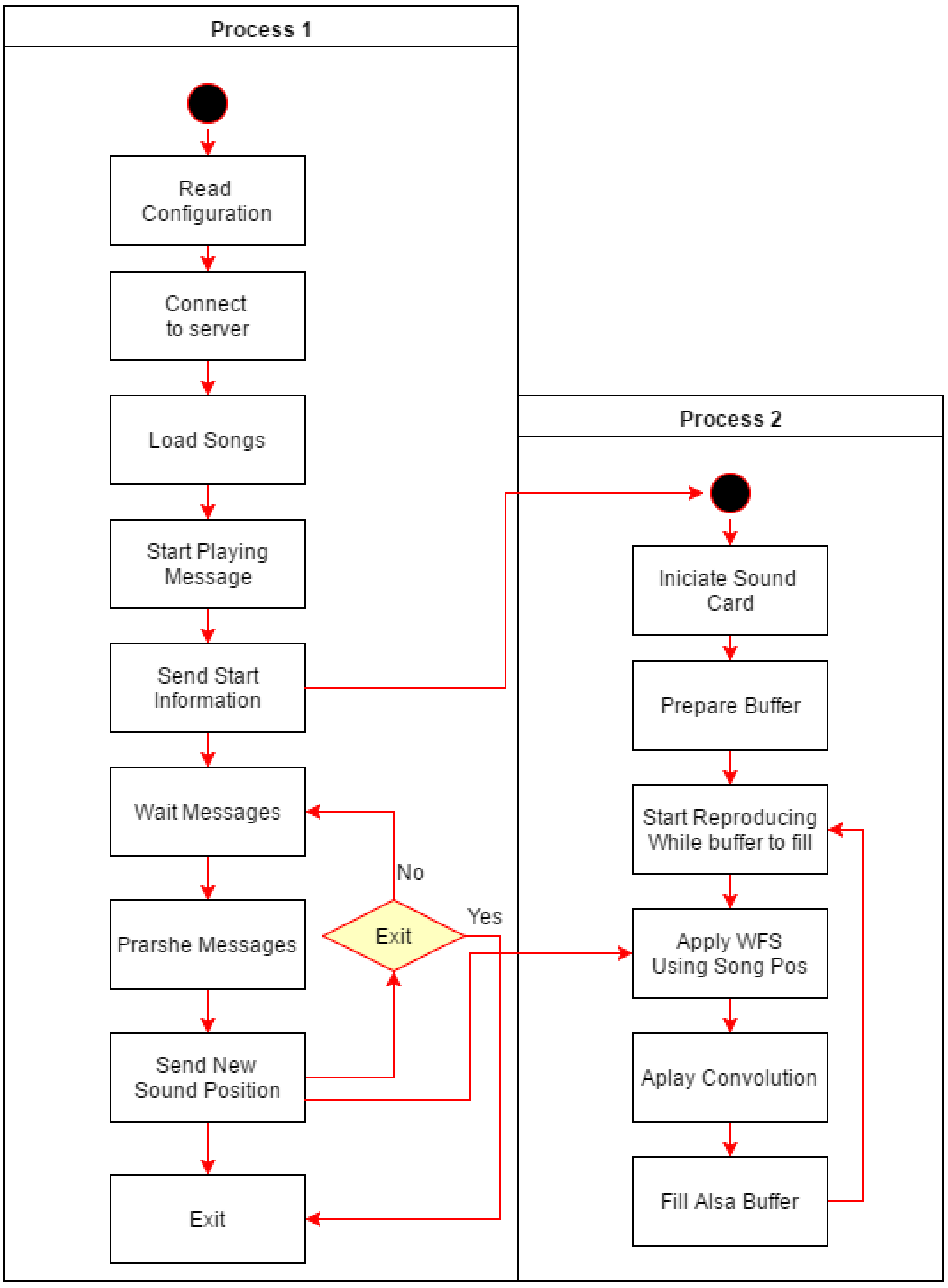


Figure 3.12: Operating Sequence Diagram.

3.5.5 Code Organization

The application code is divided in several files, so that each file only contains the functions needed for a certain purpose. The dependency of these files is shown in the Figure 3.13 and summarized below:

- Function.h: It contains general functions that are used in most of the application.
- SpatialLib.h: It contains all the functions necessary to open audio files.
- ConfigurationLib.h: It contains all the functions related with opening and reading the configuration file. It contains the structure in which it is stored.
- ProcessSoundLib.h: It contains the necessary functions to apply and use the Wave Field Synthesis. Is also stores a structure with the necessary variable in order to be able to communicate between the process.
- Convolution.h: It contains all the functions to perform the convolution.
- SuperWAVLib.h: It is the part that performs the reproduction. It initializes the ALSA library using the configuration file, applies the WFS, performs the convolution and delivers the resulting sound to ALSA to reproduce it.
- SocketUtils.h: It contains all the functions related to the server/client communication.
- Parsher.h: It transforms the received message into a certain action. One example would be when the client receives a new the sound position.
- Client.h: It connects to the server, receives the messages and uses the parser to create an action. It also initializes the reproduction process and transfers the new actions.

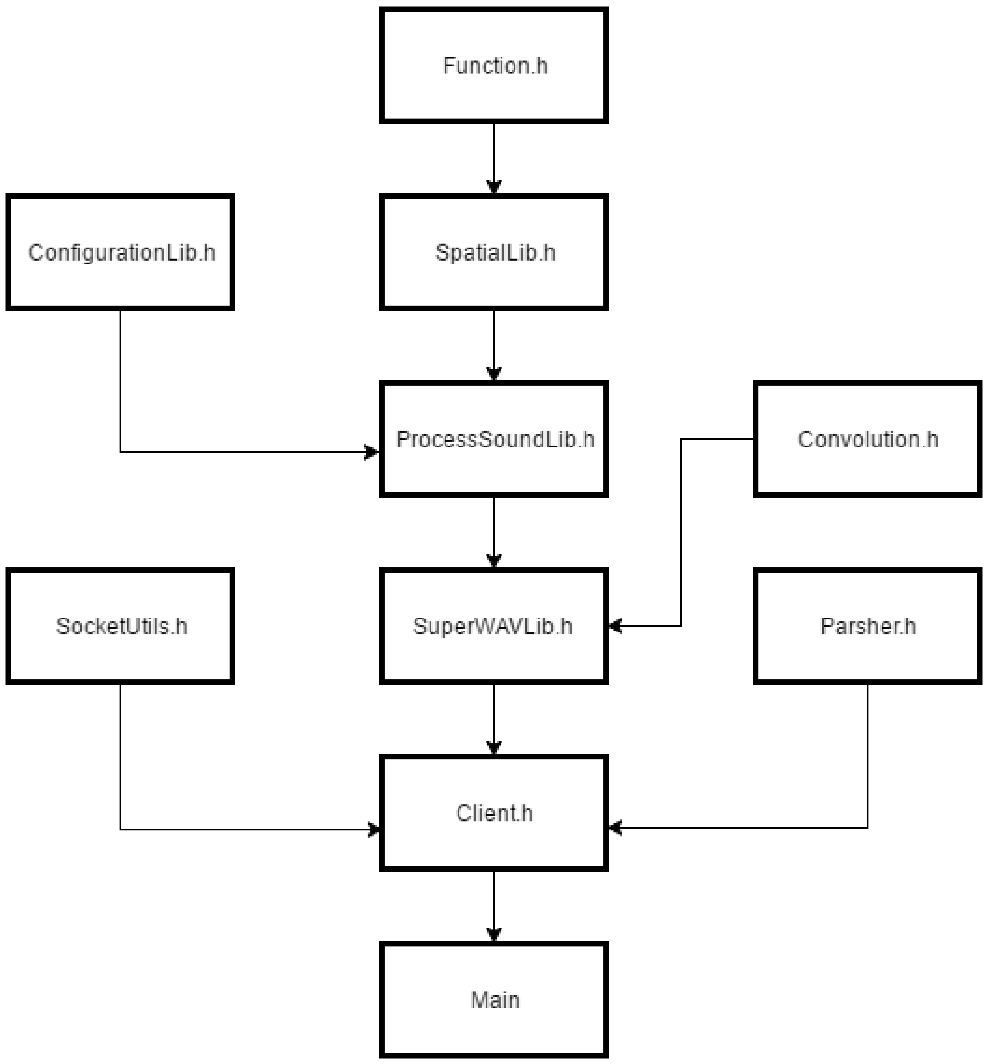


Figure 3.13: File Dependency.

3.6 Optimizations

The operation that requires most time needs to be completed is the convolution process. This operation has to be done as fast as possible in order to be able to process as many channels and sound as possible in real time.

There are two possible ways of doing it, in the time domain or in the frequency domain. In the next part we present the several forms and the optimizations done in order to achieve that purpose.

3.6.1 Time Domain

Using the formula directly that is to multiply each coefficient of the signal with all the filters coefficients requires O(n^2) arithmetic operations as shown in Figure 3.14.

```
void convolution(float* x, size_t size, float *h, size_t lh, float *y){
    int i,j;
    for(i=0;i<size; i++){
        for(j=0;j<lh;j++)
            y[i+j]=y[i+j]+x[i]*h[j];
    }
}
```

Figure 3.14: Simple Time Domain Convolution.

In order to improve its performance, a technique called loop unrolling was used. Loop unrolling will reduce (or eliminate) instructions that control the loop, such as pointer arithmetic and “end of loop” tests on each iteration as well as “hiding latencies, in particular, the delay in reading data from memory”. To eliminate this overhead, loops are re-written as a repeated sequence of similar independent statements Figure 3.15.

Apart of using the “Loop Unroll” technique, since the operations that are performed are commutative, we change the order of the multiplication in an attempt to take advantage of the cache memory. Unfortunately there were no improvements.

```c
void convolution_Loop_Unroling_8(float* x, size_t size, float *h, size_t lh
    , float* y){
    int i,j;

    float temp[8];
    for(i=0;i<size; i++){
        for(j=0;j<lh;j+=8){

            temp[0] = x[i]*h[j+0];
            temp[1] = x[i]*h[j+1];
            temp[2] = x[i]*h[j+2];
            temp[3] = x[i]*h[j+3];

            temp[4] = x[i]*h[j+4];
            temp[5] = x[i]*h[j+5];
            temp[6] = x[i]*h[j+6];
            temp[7] = x[i]*h[j+7];

            y[i+j+0] += temp[0];
            y[i+j+1] += temp[1];
            y[i+j+2] += temp[2];
            y[i+j+3] += temp[3];

            y[i+j+4] += temp[4];
            y[i+j+5] += temp[5];
            y[i+j+6] += temp[6];
            y[i+j+7] += temp[7];
        }
    }
}
```

Figure 3.15: Simple Time Domain Convolution Using Loop Unroll.

The next step in optimizing the convolution was to use the single instruction, multiple data (SIMD) that the ARM processor disposes by using the NEON technology [15]. In order to do that, it is necessary to use some special instructions and some special variables type. The first step is to load the data into a special vector type, then apply the desired operations using the NEON instructions, and at the end retrieve the data. The result is shown in the Figure 3.16.

```
void convolution_NEON(float* x, size_t size, float *h, size_t lh, float *y)
    {
    int i,j;
    float32_t xi;
    float32x4_t t;
    float32x4_t hi;
    float32x4_t acum;

    for(i=0;i<size; i++){
        xi = x[i];
        for(j=0;j<lh;j+=4){

            hi = vld1q_f32(h+j);
            t = vmulq_n_f32(hi,xi);

            acum = vld1q_f32(&y[i+j]);
            acum = vaddq_f32(acum,t);
            vst1q_f32(&y[i+j],acum);
    }
  }
}
```

Figure 3.16: Simple Time Domain Convolution NEON.

This last implementation resulted to be the optimal of all for the moment. More details of the results are explained in the Chapter 4.

In our application we cannot use this implementation of the convolution, because it needs the entire signal. For example, if the user moves the position of the source, the signal's values will change. Since we can not anticipate when the user will move the source, the entire signal cannot be known in advance.

The overlap add method was implements to solve this problem, so we could take advantage of the convolution implemented. The method gets a block size of the signal and process the convolution. The result is the convolution of that block size, and the remainder of the convolution is stored in a temporal buffer so it can be summed to the next iteration. The resulting code is shown in the Figure 3.17, which also has the loop unroll optimization technique.

```
void convolutionOverlapAdd(float* partX, size_t lx, float *localBuffer,
    size_t lb, float* h, size_t lh, float *y) {
    int i,j;
    convolution_NEON(partX, lx, h, lh, localBuffer);
    for (j = 0; j < lx; j+=4) {
        y[j+0]= localBuffer[j+0];
        y[j+1]= localBuffer[j+1];
        y[j+2]= localBuffer[j+2];
        y[j+3]= localBuffer[j+3];
        localBuffer[j+0], localBuffer[j+1] = 0;
        localBuffer[j+2], localBuffer[j+3] = 0;

    }
    for (i = 0; i < lx; i+=4) {
        localBuffer[i+0] = localBuffer[j+0];
        localBuffer[i+1] = localBuffer[j+1];
        localBuffer[i+2] = localBuffer[j+2];
        localBuffer[i+3] = localBuffer[j+3];
        localBuffer[j+0], localBuffer[j+1] = 0;
        localBuffer[j+2], localBuffer[j+3] = 0;
        j+=4;
    }
}
```

Figure 3.17: Simple Time Domain Convolution Overla ADD.

This implementation can use any implementation of the convolutions explained above, but its performance is slower than the normal method. In order to achieve the best performance, we used the ARM version.

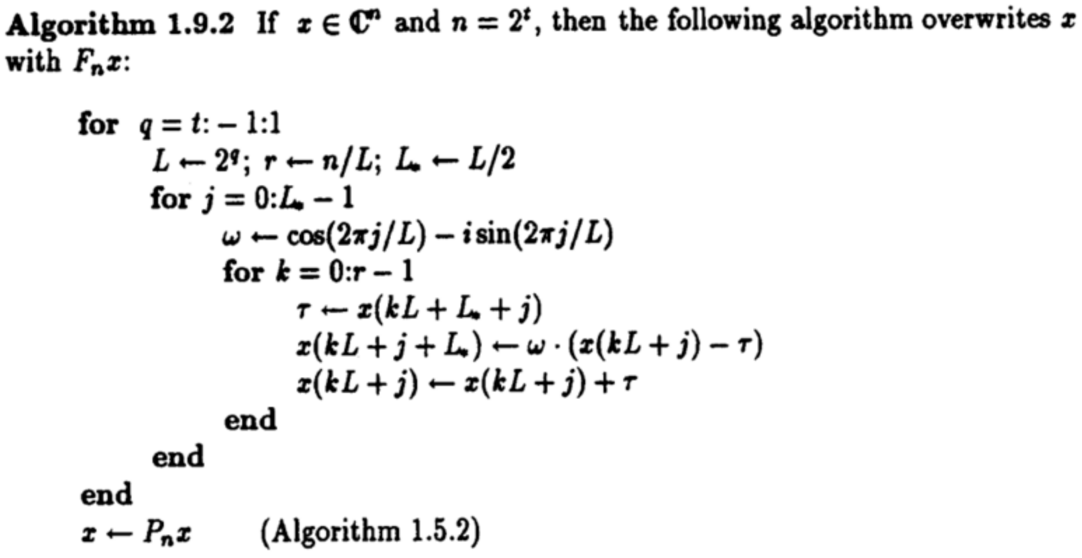
3.6.2 Frequency Domain

The second form of implementing the convolution is to do it in the frequency domain by using the Fast Fourier Transform. This method costs O(n log n) operations.

In order to create the convolution using the FFT the have been two approaches. The first one was to implement the FFT of Gentleman-Sande. The second approach was to use the FFTW library.

FFT Gentleman-Sande

The implementation of the FFT algorithm is based on 1.9.2, page 67 of the book “Computational Frameworks for the Fast Fourier Transform”. The peculiarity of this implementation is that it can only handle vector sizes that are power of 2.



Algorithm 1.9.2 If $x \in \mathbb{C}^n$ and $n = 2^t$, then the following algorithm overwrites x with $F_n x$:

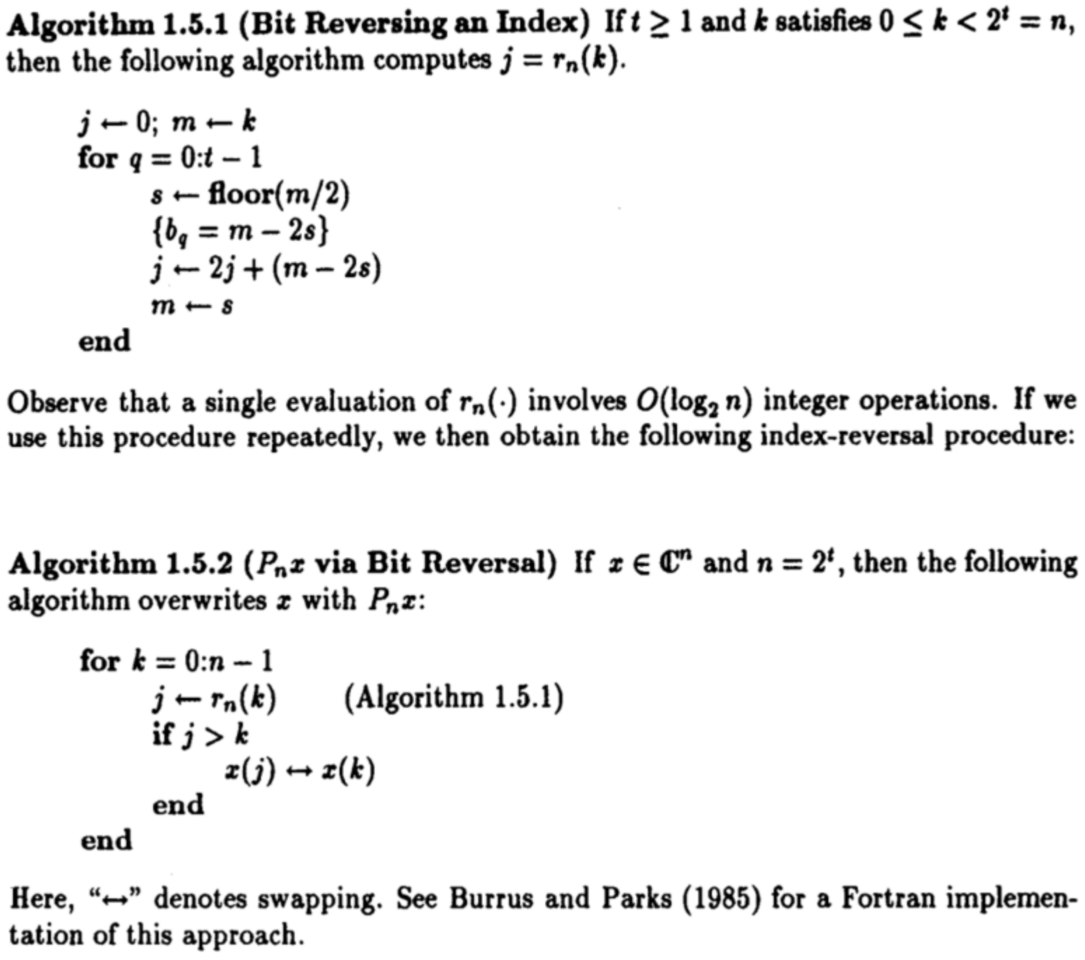
```
for $q = t : -1 : 1$
    $L \leftarrow 2^q$; $r \leftarrow n/L$; $L_* \leftarrow L/2$
    for $j = 0 : L_* - 1$
        $\omega \leftarrow \cos(2\pi j/L) - i\sin(2\pi j/L)$
        for $k = 0 : r - 1$
            $\tau \leftarrow x(kL + L_* + j)$
            $x(kL + j + L_*) \leftarrow \omega \cdot (x(kL + j) - \tau)$
            $x(kL + j) \leftarrow x(kL + j) + \tau$
        end
    end
end
$x \leftarrow P_n x$        (Algorithm 1.5.2)
```

Figure 3.18: Pseudo code FFT Gentleman-Sande.

Besides implementing the FFT algorithm Figure 3.20, a method was implemented in order to reorganize the data after applying the algorithm. For that purpose was used Algorithm 1.5.2 “ Bit reversal ” and 1.5.1 Algorithm “ Bit Reversing and Index ”.

The implementation of the Convolution is shown in the Figure 3.23, which also used the “loop unroll” technique in order to improve its performance. Also wrappers ware used in order to simplify the use of the FFT function Figure 3.21 and Figure 3.22.

The main problem of this method is the same as in the convolution carried out in the time domain. The problem is the same as before: it can only be used if it disposes of the entire signal. For that purpose, the Overlap Save Method is implemented (Figure 3.24) in this case in order to be able to do the convolution in block sizes.



Algorithm 1.5.1 (Bit Reversing an Index) If $t \geq 1$ and k satisfies $0 \leq k < 2^t = n$, then the following algorithm computes $j = r_n(k)$.

```
$j \leftarrow 0$; $m \leftarrow k$
for $q = 0{:}t-1$
    $s \leftarrow \text{floor}(m/2)$
    $\{b_q = m - 2s\}$
    $j \leftarrow 2j + (m - 2s)$
    $m \leftarrow s$
end
```

Observe that a single evaluation of $r_n(\cdot)$ involves $O(\log_2 n)$ integer operations. If we use this procedure repeatedly, we then obtain the following index-reversal procedure:

Algorithm 1.5.2 ($P_n x$ via Bit Reversal) If $x \in \mathbb{C}^n$ and $n = 2^t$, then the following algorithm overwrites x with $P_n x$:

```
for $k = 0{:}n-1$
    $j \leftarrow r_n(k)$        (Algorithm 1.5.1)
    if $j > k$
        $x(j) \leftrightarrow x(k)$
    end
end
```

Here, "$\leftrightarrow$" denotes swapping. See Burrus and Parks (1985) for a Fortran implementation of this approach.

Figure 3.19: Pseudo code Bit Reversal.

In the implementation of the FFT Gentleman-Sande (Figure 3.20) there has not been applied any type of optimizations. For that reason, we later decided to use an optimized library in order to take advantage of the ARM architecture and improve its performance.

Our goal is to reduce the processing time of the convolution for small block sizes. Unfortunately this implementation performs better for large signal sizes.

```c
void fftGS_General(double complex * x, int ncol, enum SIGN sig){
  int t = ceil(log(ncol)/log(2));
  for (int q = t; 1 <= q; --q){
    int L = pow(2, q);
    int r = ncol/L;
    int L_ = L/2;

    for (int j = 0; j < L_; ++j){
      double complex w = cos(2*M_PI*j/L)+( sig * sin(2*M_PI*j/L)*I);
      for (int k = 0; k < r; ++k){
        int index = (k*L)+L_+j;
        double complex tau = x[k*L+L_+j];
        x[k*L+j+L_]=w*(x[k*L+j]-tau);
        x[k*L+j]=x[k*L+j]+tau;
      }
    }
  }
  // Bit Reversal: Algoritm 1.5.1
  for (int k = 0; k < ncol; ++k){
    // Algoritm 1.5.2
    int j = 0; int m = k;
    for (int q = 0; q < t; ++q){
      int s = floor(m/2);
      j=2*j+(m-2*s);
      m=s;
    }// End Algoritm 1.5.2
    double complex temportal_swap;
    if (j>k){
      temportal_swap = x[k];
      x[k]=x[j];
      x[j]=temportal_swap;
}}}
```

Figure 3.20: Implementation General FFT using Gentleman-Sande.

```c
enum SIGN{
  FORWARD = 1,
  REVERSE = -1
};

double complex * fftGS(double * vector, int ncol){
  double complex * x =(double complex *)malloc(ncol*sizeof(double complex))
    ;
  for (int i = 0; i < ncol; ++i){
    x[i] = vector[i];
  }
  fftGS_General(x,ncol,FORWARD);
  return x;
}
```

Figure 3.21: Implementation rwapper FFT using Gentleman-Sande.

```c
double * ifftGS(double complex * x, int ncol){
  double * X = (double *)malloc(ncol*sizeof(double));
  fftGS_General(x,ncol,REVERSE);
  for (int i = 0; i < ncol; ++i){
    X[i]=creal(x[i])/ncol;
  }
  return X;
}
```

Figure 3.22: Implementation rwapper iFFT using Gentleman-Sande.

```
void convolutionFFT(float* x, size_t lx, float *h, size_t lh, float* y){
     size_t i,j;
     float _Complex * H_FFT =(float _Complex *)malloc(lh*sizeof(float
    _Complex));
     fftGS(h, lh,H_FFT);
     float _Complex * A_FFT =(float _Complex *)malloc(lx*sizeof(float
    _Complex));
  fftGS(x, lh, A_FFT);
     for (i = 0; i < lx; i=i+8) {
          A_FFT[i+0] = A_FFT[i+0]*H_FFT[i+0];
          A_FFT[i+1] = A_FFT[i+1]*H_FFT[i+1];
          A_FFT[i+2] = A_FFT[i+2]*H_FFT[i+2];
          A_FFT[i+3] = A_FFT[i+3]*H_FFT[i+3];
          A_FFT[i+4] = A_FFT[i+4]*H_FFT[i+4];
          A_FFT[i+5] = A_FFT[i+5]*H_FFT[i+5];
          A_FFT[i+6] = A_FFT[i+6]*H_FFT[i+6];
          A_FFT[i+7] = A_FFT[i+7]*H_FFT[i+7];
     }
     ifftGS(A_FFT,lx,y);
     free(H_FFT);
     free(A_FFT);
}
```

Figure 3.23: Implementation Convolution that uses FFT Gentleman-Sande.

```
void convolutionOverlapSave(float* x, size_t lx, float* localBuffer, size_t
    lb, float* h, float* y) {
    size_t i,j = 0;
  for (j = lx; j < lb; j=j+8) {
        localBuffer[j+0] = x[j+0-lx]; localBuffer[j+1] = x[j+1-lx];
        localBuffer[j+2] = x[j+2-lx]; localBuffer[j+3] = x[j+3-lx];
        localBuffer[j+4] = x[j+4-lx]; localBuffer[j+5] = x[j+5-lx];
        localBuffer[j+6] = x[j+6-lx]; localBuffer[j+7] = x[j+7-lx];
    }
    convolutionFFT(localBuffer,lb, h, lb, localBuffer);
    for (j = 0; j < lx; j=j+8) {
      localBuffer[j+0]=x[j+0]; localBuffer[j+1]=x[j+1];
      localBuffer[j+2]=x[j+2]; localBuffer[j+3]=x[j+3];
      localBuffer[j+4]=x[j+4]; localBuffer[j+5]=x[j+5];
      localBuffer[j+6]=x[j+6]; localBuffer[j+7]=x[j+7];

    y[j+0]= localBuffer[j+lx+0]; y[j+1]= localBuffer[j+lx+1];
        y[j+2]= localBuffer[j+lx+2]; y[j+3]= localBuffer[j+lx+3];
        y[j+4]= localBuffer[j+lx+4]; y[j+5]= localBuffer[j+lx+5];
        y[j+6]= localBuffer[j+lx+6]; y[j+7]= localBuffer[j+lx+7];

        localBuffer[j+0+lx] = 0; localBuffer[j+1+lx] = 0;
        localBuffer[j+2+lx] = 0; localBuffer[j+3+lx] = 0;
        localBuffer[j+4+lx] = 0; localBuffer[j+5+lx] = 0;
        localBuffer[j+6+lx] = 0; localBuffer[j+7+lx] = 0;
    }
}
```

Figure 3.24: Implementation Convolution Overlap Save that uses FFT Gentleman-Sande.

FFTW

FFTW is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data. The version used is the 3.3.4 which supports the ARM NEON extensions.

The creators of the software claims that the benchmarks they performed on a variety of platforms, show that FFTW's performance is typically superior to that of other publicly available FFT software.

For the installation at least in the ARM boards it is important to do a series of steps in order to install the software. The first one is to change the file "simd-support/neon.c " and replace function "really_have_neon()" with:

```
static int really_have_neon(void)
{
        return 1;
}
```

That way we force the application to use the NEON instructions. Afterwards, it is necesary to configurate the installation with the flags: "–enable-float","–enable-neon" and "–enable-openmp" to enable multithread using OpenMP.

```
./configure --enable-float --enable-neon --enable-openmp
```

The application used the same method of Overlap Save shown in the Figure 3.24 but instead of executing the implementation of the FFT Gentleman-Sande, it executes the implementation shown in the Figure 3.25.

This implementation of the convolution using the FFTW library is two or three times faster than the implementation of Gentleman-Sande. But it was not as fast as the Overlap Add method that uses NEON instructions for small block sizes.

```
void convolutionFFTW(float *x, size_t lx, float *h, size_t lh, float *y){
    int i = 0; int size_lx_lh = nextpw2(lx+1);
    fftw_complex *X,*H;
    /* define fftw_plan which contains all the data needed to compute*/
    fftw_plan fftX, fftH, ifftR;
    /* allocate memory */
   X = (fftw_complex*)fftw_malloc(sizeof(fftw_complex)*size_lx_lh);
   H = (fftw_complex*)fftw_malloc(sizeof(fftw_complex)*size_lx_lh);
    fftX=fftw_plan_dft_1d(size_lx_lh,X,X,FFTW_FORWARD,FFTW_ESTIMATE);
    fftH=fftw_plan_dft_1d(size_lx_lh,H,H,FFTW_FORWARD,FFTW_ESTIMATE);
    ifftR=fftw_plan_dft_1d(size_lx_lh,X,X,FFTW_BACKWARD,FFTW_ESTIMATE);

    for(i=0;i<lx;i=i+8) { /*Copy data X*/
        X[i+0] = x[i+0]+0.0f*_Complex_I; X[i+1] = x[i+1]+0.0f*_Complex_I;
        X[i+2] = x[i+2]+0.0f*_Complex_I; X[i+3] = x[i+3]+0.0f*_Complex_I;
        X[i+4] = x[i+4]+0.0f*_Complex_I; X[i+5] = x[i+5]+0.0f*_Complex_I;
        X[i+6] = x[i+6]+0.0f*_Complex_I; X[i+7] = x[i+7]+0.0f*_Complex_I;
    }for(i=0;i<lh;i=i+8) { /*Copy data H*/
        H[i+0] = h[i+0]+0.0f*_Complex_I; H[i+1] = h[i+1]+0.0f*_Complex_I;
        H[i+2] = h[i+2]+0.0f*_Complex_I; H[i+3] = h[i+3]+0.0f*_Complex_I;
        H[i+4] = h[i+4]+0.0f*_Complex_I; H[i+5] = h[i+5]+0.0f*_Complex_I;
        H[i+6] = h[i+6]+0.0f*_Complex_I; H[i+7] = h[i+7]+0.0f*_Complex_I;
    }
    fftw_execute(fftX); fftw_execute(fftH); /*Execute FFT for X and H*/
    for (i = 0; i < size_lx_lh; i=i+8) {
        X[i+0] = X[i+0]*H[i+0]; X[i+1] = X[i+1]*H[i+1];
        X[i+2] = X[i+2]*H[i+2]; X[i+3] = X[i+3]*H[i+3];
        X[i+4] = X[i+4]*H[i+4]; X[i+5] = X[i+5]*H[i+5];
        X[i+6] = X[i+6]*H[i+6]; X[i+7] = X[i+7]*H[i+7];
    }
    fftw_execute(ifftR);  /*Execute iFFT for result*/
    for (i = 0; i < lx+lh-1; ++i)
        y[i] = creal(X[i])/(double)size_lx_lh;
    fftw_destroy_plan(fftX); fftw_destroy_plan(fftH);
    fftw_destroy_plan(ifftR);
    fftw_free(X); fftw_free(H);
}
```

Figure 3.25: Implementation Convolution using FFTW.

Chapter 4

Test and Results

4.1 Optimization Results

As shown in the previous chapter, Section 3.6, we focused on optimizing the convolution process in order to be able to process as many channels and possible sounds in real time. In order to process sound in real time, it is necessary to know the buffer that is going to be used for playback. The bigger the buffer is, the more time it will reproduce, and the more time we also have to prepare the next buffer for playback.

The problem is that if the user changes the position of a sound source, it will not take effect until the next buffer is loaded. For that reason, if the buffer is large it will take more time to make the change visible for the user.

This chapter is divided in three sections. In the first section, we explain the results we have obtain performing the convolution in time domain where the entire signal is available. Results of the convolution are carried out using signal blocks. We take advantage of all the cores by using openMP. The second section shows the final results of the application, and the third section is devoted to the analysis of the complexity.

4.1.1 Convolution

As explained in the previous chapter, there are several ways of implementing the convolution. In this section, we are going to analyze the optimization results of the implementation of the convolution in the time domain where we dispose of the signal.

The time results are shown in the Table 4.1, but before we continue, there are few notes about Table 4.1 that needs to be clarified. The first one is that “lx” corresponds to the size of the signal and “lh” corresponds to the size of the filter.

The names of the columns are explained below:

- “Normal” is the implementation of the convolution without any optimization.
- “Normal v1 4” and “Normal v1 8” is the implementation of the convolution and uses a loop unroll of 4 and 8.
- “Normal v2 4” and “Normal v2 8” is the implementation of the convolution where the operations order is changed. It also uses a loop unroll of 4 and 8.
- “Neon v1 4” and “Neon v1 8” is the implementation of the convolution where it uses the NEON instructions similar to loop unroll of 4 and 8.
- “Neon v2 4” and “Neon v2 8” is the implementation of the convolution where it uses the NEON instructions similar to loop unroll of 4 and 8 with the operations order changed.

The conclusion we can obtain by analyzing Table 4.1 is that the best time was achieved without using the NEON instructions is the “Normal v1 8”, which uses the loop unroll technique of 8 operations. The “Normal v2 8” is very similar, but it is not the most efficient. Although in the Table 4.1 does not appear, the use of the loop unroll of more than 8 operations decreases the efficiency of the application.

lx-lh	**Normal**	**Normal v1 4**	**Normal v1 8**	**Normal v2 4**	**Normal v2 8**	**Neon v1 4**	**Neon v1 8**	**Neon v1 16**	**Neon v2 4**	**Neon v2 8**
64-64	0,008	0,007	**0,005**	0,007	0,006	0,005	**0,003**	**0,003**	0,004	0,004
128-64	0,016	0,012	**0,009**	0,011	0,01	0,008	**0,005**	*0,006*	0,007	0,007
128-128	0,031	0,022	**0,019**	0,022	0,02	0,013	**0,011**	**0,011**	0,013	0,013
256-64	0,031	0,024	**0,019**	0,022	0,02	0,016	**0,01**	0,011	0,012	0,012
256-128	0,058	0,044	*0,041*	0,043	**0,038**	0,026	*0,022*	**0,02**	0,024	0,023
256-256	0,112	0,084	**0,069**	0,1	0,076	0,049	*0,041*	**0,04**	0,048	0,047
512-64	0,063	0,046	**0,036**	0,043	0,038	0,031	**0,02**	*0,021*	0,023	0,044
512-128	0,114	0,086	**0,073**	0,083	0,075	0,07	*0,043*	**0,039**	0,045	0,044
512-256	0,221	0,17	**0,138**	0,168	0,152	0,098	**0,079**	**0,079**	0,102	0,088
512-512	0,426	0,331	**0,289**	0,331	0,303	0,194	*0,156*	**0,154**	0,182	0,174

Table 4.1: Time Processing Convolution in milliseconds.

The convolution results obtained without using the NEON instructions are satisfying, because it takes for a signal length of 512 and filter length of 64 only 0.036 milliseconds. But by using the NEON instructions for the same sizes it takes 0.02 milliseconds if we use the

"Neon v1 8" or 0.021 milliseconds if we use the "Neon v1 16". The difference between them is negligible.

4.1.2 Overlap Add and Overlap Save

Given the purpose of the application, we need to do the convolution in blocks. For that reason, we will analyze the results of the Table 4.2. This table contains the result of the convolution in the frequency domain and the convolution in block by using the Overlap add and Overlap Save methods.

In this case the "lx" and "lh" are the sizes of the signal and filter as in the previous table, but the names of the columns are explained below:

- "Conv" is the implementation of the convolution without any optimization.
- "Conv LU8" is the implementation of the convolution that uses the loop unroll of 8 operations.
- "Conv ARM 8" implements the convolution where it uses the NEON instructions similar to loop unroll of 8 operations.
- "Conv FFT" is the implementation of the convolution of Gentleman-Sande.
- "Conv FFTW" is the implementation of the convolution using the FFT library.
- "Conv OA" is the implementation of the Overlap ADD method using the "Conv".
- "Conv OA 8" is the implementation of the Overlap ADD method using the "Conv LU8".
- "Conv ARM OA" is the implementation of the Overlap ADD method using the "Conv ARM 8".
- "Conv OS" is the implementation of the Overlap Save method using the "Conv FFT".
- "Conv OS FFTW" is the implementation of the Overlap Save method using the "Conv FFTW".

As shown in the Table 4.2 the best method to use is the "Conv ARM 8" most of the time for filter sizes that are less than 512. This convolution is implemented in the time domain and uses the NEON instructions. But if the filter size is superior to 512, it is better to use the "Conv OS FFTW", which is a convolution implemented in the frequency domain, and uses the FFTW library.

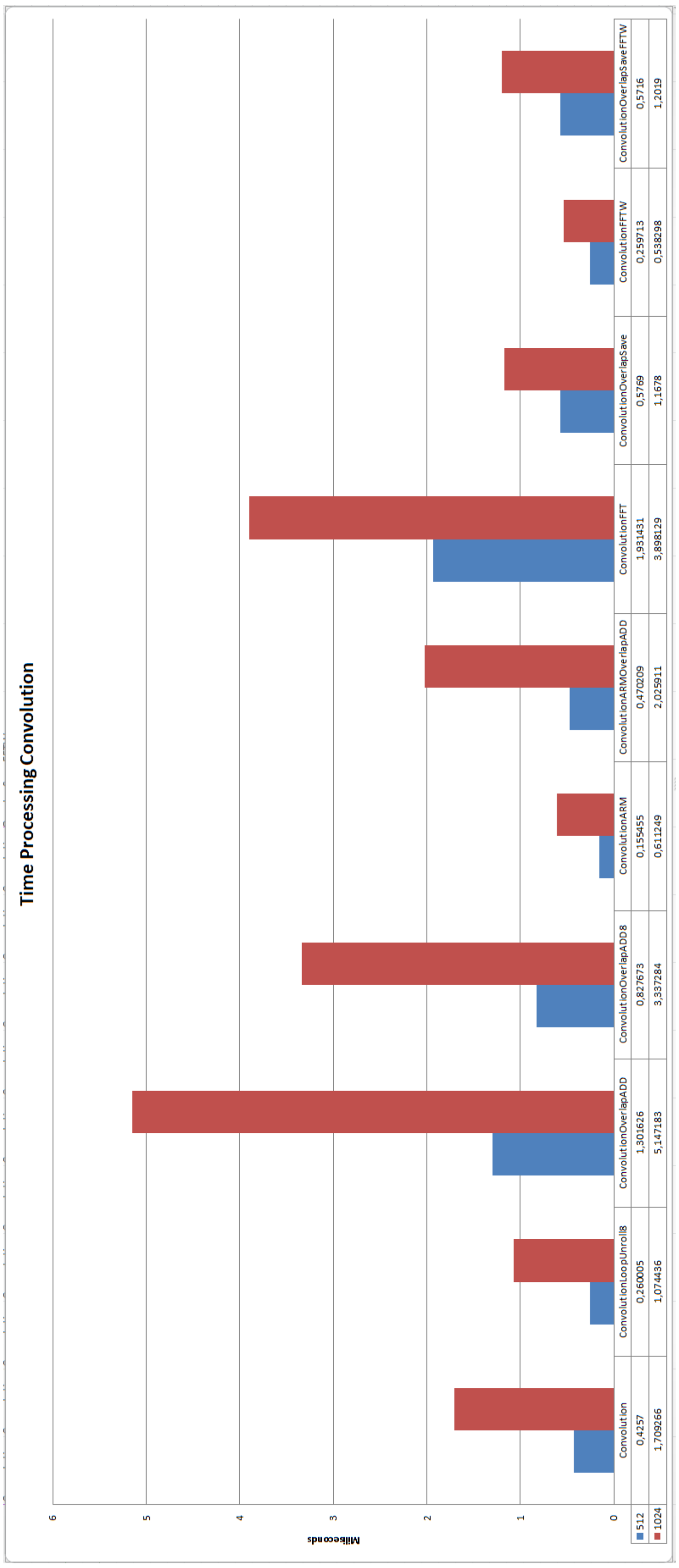
bs-lh	**Conv**	**Conv LU8**	**Conv ARM 8**	**Conv FFT**	**Conv FFTW**	**Conv OA**	**Conv OA 8**	**Conv ARM OA**	**Conv OS**	**Conv OS FFTW**
64 64	0,007887	0,004108	**0,002419**	0,234495	0,185689	0,024126	0,013563	**0,007794**	0,0742	0,0741
128 64	0,015504	0,008205	**0,004829**	0,476359	0,121291	0,047785	0,02705	**0,015563**	0,1561	0,15
256 64	0,03096	0,016397	**0,00964**	0,947345	0,179036	0,097499	0,053763	**0,030912**	0,2884	0,2875
512 64	0,061879	0,032783	**0,019276**	1,921678	0,257586	0,194447	0,107408	**0,061622**	0,5746	0,5763
1024 64	0,123843	0,065559	**0,038546**	3,987108	0,520396	0,388998	0,23942	**0,123127**	1,909	1,169
2048 64	0,247889	0,131195	**0,101713**	7,979112	1,98337	0,805069	0,432292	**0,25012**	2,4748	2,4581
512 512	0,426543	0,259894	**0,156147**	1,920184	0,265013	1,310526	0,830211	**0,470546**	0,5733	0,5731
1024 512	0,873896	0,521542	**0,332754**	3,907836	0,51035	2,601406	1,677226	**0,966494**	1,1677	1,1849
2048 512	1,724677	1,06074	**0,63068**	7,972815	1,967552	5,225386	3,373001	**1,912579**	2,5014	2,3956
2046 2048	6,808647	4,18615	2,483896	7,999225	**1,973582**	20,453229	13,266856	12,426696	2,4722	**2,2795**

Table 4.2: Time Processing Convolution in milliseconds using the FFT and Overlap methods.

The same thing happens if instead of performing the convolution of the entire signal, it performed the convolution in blocks using the "Overlap" method. While the filter size is less than 512 it is better to use the "Conv ARM OA", but if the filter size is greater than 512 it is better to use the "Conv OS FFTW".

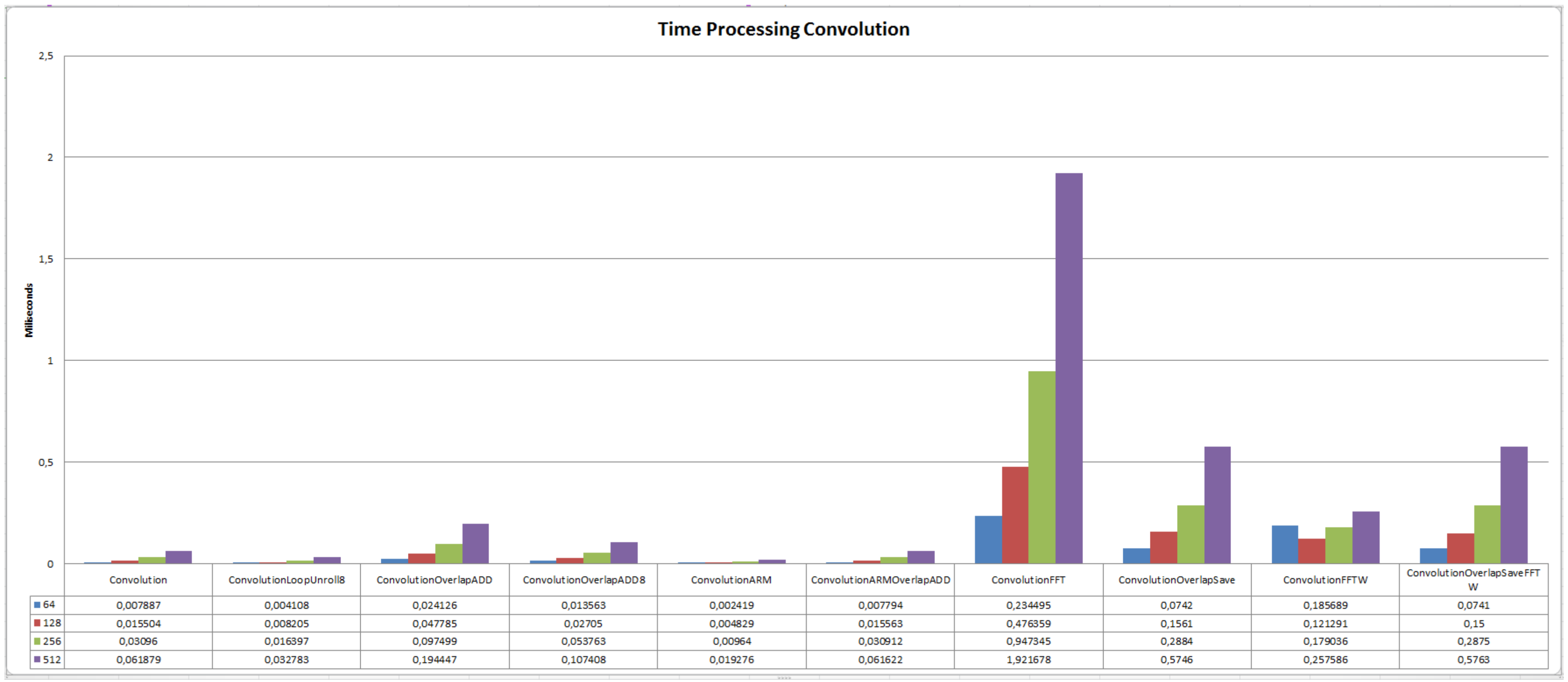
In order to determine which method is better in each case a test was carried out in where the size of signal and filter is the same. The results are shown in the Figure 4.1, and we can conclude that while we should use the NEON implementation to sized up to 512, but for superior sizes we should use the FFTW implementation.

In this case, the application uses a filter of 64 coefficients and we are interested in small sizes. For that reason, the best method we can use is the convolution implemented using the Overlap Add method with the NEON instructions Figure 4.2.



	Convolution	ConvolutionLoopUnroll8	ConvolutionOverlapADD	ConvolutionOverlapADD8	ConvolutionARM	ConvolutionARMOverlapADD	ConvolutionFFT	ConvolutionOverlapSave	ConvolutionFFTW	ConvolutionOverlapSaveFFTW
🟦 512	0,4257	0,260005	1,301626	0,827673	0,155455	0,470209	1,931431	0,5769	0,259713	0,5716
🟥 1024	1,709266	1,074436	5,147183	3,337284	0,611249	2,025911	3,898129	1,1678	0,538298	1,2019

Figure 4.1: Difference between convolution in time domain and frequency domain. .



	Convolution	ConvolutionLoopUnroll8	ConvolutionOverlapADD	ConvolutionOverlapADD8	ConvolutionARM	ConvolutionARMOverlapADD	ConvolutionFFT	ConvolutionOverlapSave	ConvolutionFFTW	ConvolutionOverlapSaveFFTW
🟦 64	0,007887	0,004108	0,024126	0,013563	0,002419	0,007794	0,234495	0,0742	0,185689	0,0741
🟥 128	0,015504	0,008205	0,047785	0,02705	0,004829	0,015563	0,476359	0,1561	0,121291	0,15
🟩 256	0,03096	0,016397	0,097499	0,053763	0,00964	0,030912	0,947345	0,2884	0,179036	0,2875
🟪 512	0,061879	0,032783	0,194447	0,107408	0,019276	0,061622	1,921678	0,5746	0,257586	0,5763

Figure 4.2: Performance of the each type of convolution for the purpose of the application.

4.1.3 OpenMP

Nowadays most of the ARM boards have more than one core, like for example both the Raspberry Pi and the Nvidia Jetson TK1 boards have 4 cores. Thus, in order to take advantage of all the computational capability available, we use OpenMP to process each output channel independently.

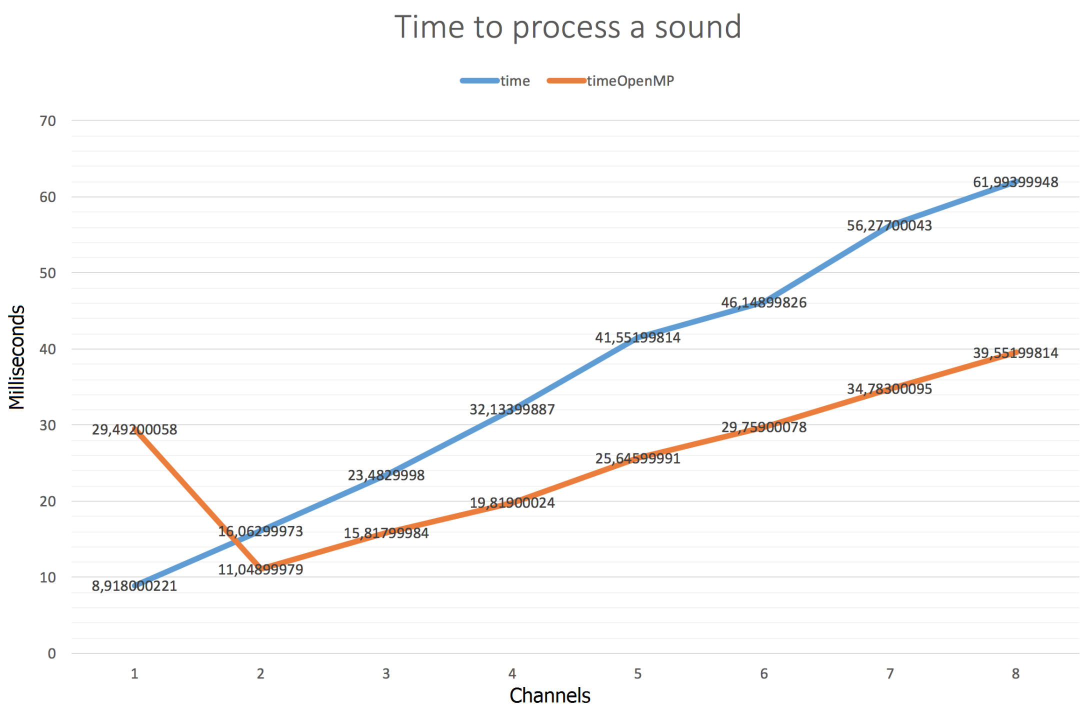


Figure 4.3: Time employed in processing a sound with and without OpenMP.

The results are shown in the Figure 4.3, and represent the time involved in processing a sound for different number of channels. If there are more than one sound involved than the time it is necessary to process increases. Also, if the number of channels increases, the time required for processing also increases.

OpenMP is capable of improving the processing time while there is more than 2 output channels. Otherwise, the computation time of the whole system increases because of the time needed to create the threads.

The graphic only has up to 8 output channels because most of the audio cards are 5.1 and 7.1. That means that it has only 6 and 8 audio outputs, although there are also audio cards that can achieve up to 52 audio outputs like for example the “RME HDSP 9652”. But they are very expensive and it is not the purpose of this system.

4.2 Validation

In order to demonstrate that the application works and the nodes are perfectly synchronized, we compared the speaker outputs with the original sound using the correlation. The correlation will compare the similarity between two observers and if there is any information that is similar will be represented by a delta. The delta represents a high value in the graphic as for example in the Figure 4.4, where the correlation has been made with the same signal.

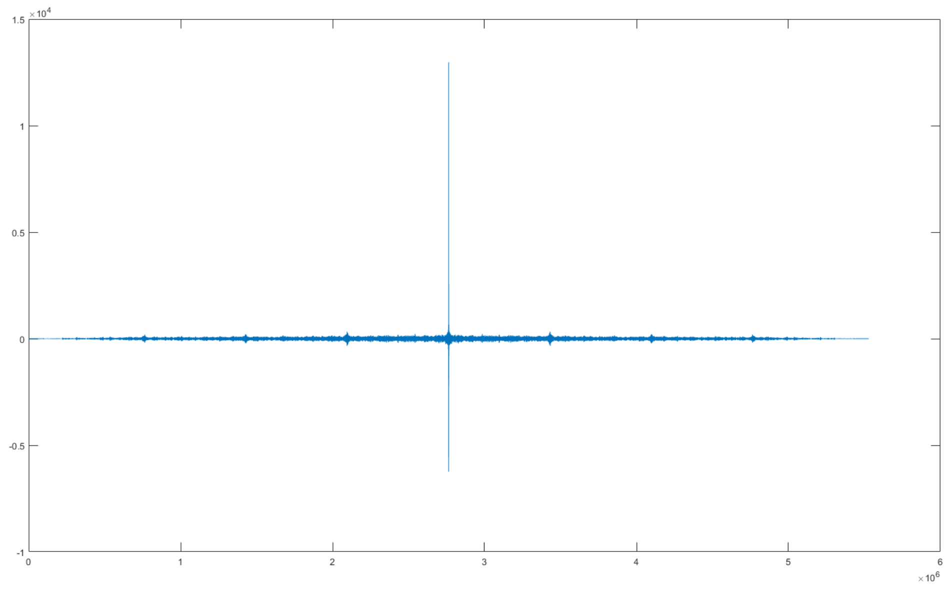


Figure 4.4: Example of the correlation made with the same sound.

The results that are shown in the Figure 4.5 and Figure 4.6 are not 100% reliable due to the fact that between the output of the speakers and the input of the microphone there has been applied a filter due to the environment. These graphs are merely indicative for the reader.

The Figure 4.5 shows the result of comparing the output of the application when the nodes are not synchronized with the original sound. By analyzing it, we can deduce that the output of the application does not have much information in common with the original sound due to the lack of a delta.

Instead of we analyze the Figure 4.6 we the nodes are synchronized; we can observe a delta value that is higher than the rest of the graph values. This indicates that there is common information between the output of the application and the original sound.

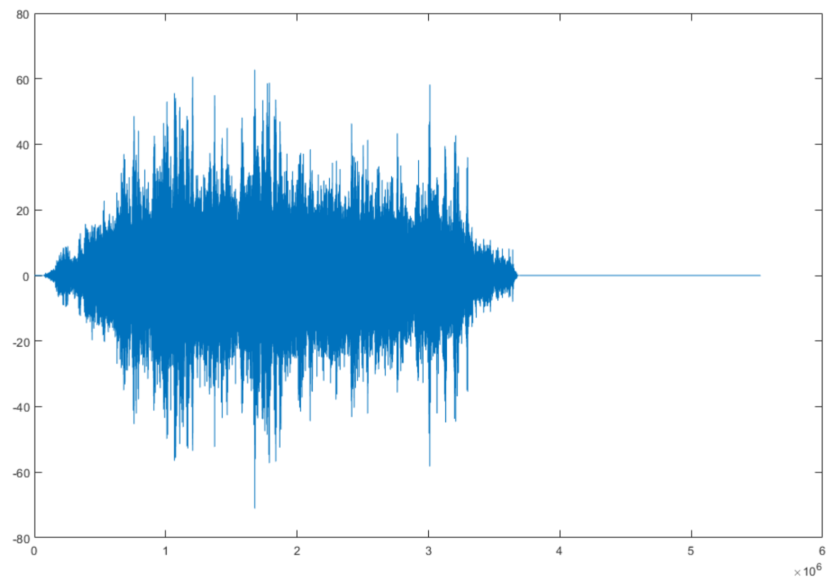


Figure 4.5: Unsynchronized output of the application.

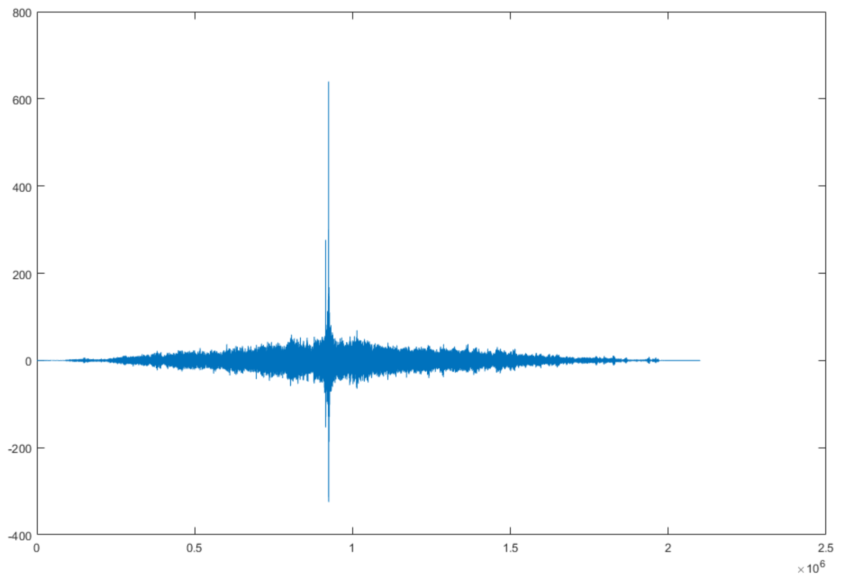


Figure 4.6: Synchronized output of the application.

4.3 Analysis of Complexity

The main focus of the application is to process the sound in real time, so when a user changes the position of a sound it will be processed and applied immediately.

In order to achieve that purpose it is necessary to use small buffers sizes. For example, if we use a buffer size of 512 integers and the frame rate is 44100, it will reproduce 0,01160997732 seconds, or approximately 11,6 milliseconds. After the first buffer has started reproducing, we have 11,6 milliseconds to prepare the next buffer, artifacts will be perceived.

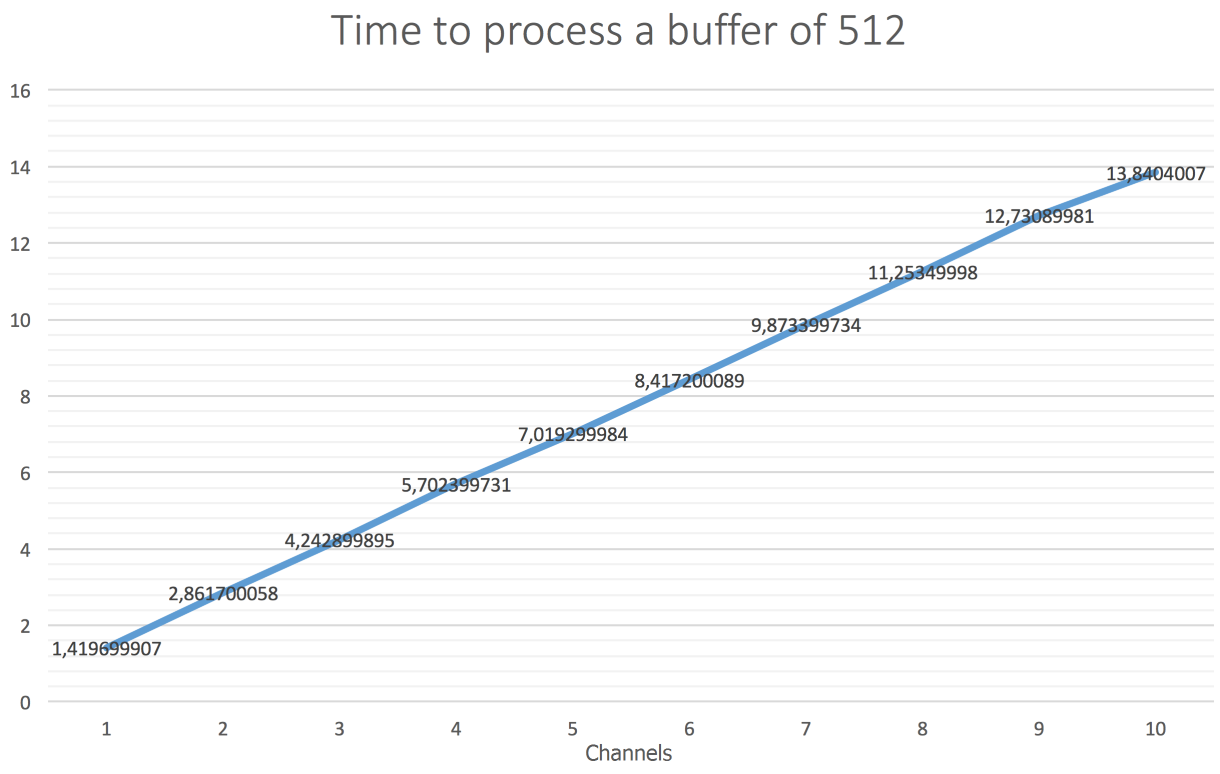


Figure 4.7: Time spent in processing a buffer size of 512.

The Figure 4.7 shows the time necessary to process one sound depending on the number of output channels. From that figure, we can deduce that it is not possible to process more than 8 channels with only one core. The time to process 8 output channels is 11,25 milliseconds which is less than 11,6 milliseconds available. Thus, it is better not to overcome the 7 output channels.

Chapter 5

Conclusion

The overall aim of this research was to study the possibility of implementing a distributed Wave Field Synthesis system application using low power ARM processors and evaluate its potential. The motivation of this research comes from the necessity of reducing the cost of usual centralized system, and adds flexibility in modifying the design.

This chapter summarizes the findings of this research work. In the Section 5.1, we have reviewed the research objectives given in the introductory chapter. In the Section 5.2, we have explained how the knowledge achieved in the Master's Degree is applied. Additionally, Recommendations for future research and personal evaluations are discussed in Section 5.3.

5.1 Achievement of Objectives

The first and one of the most important parts of this research was to evaluate the efficiency of the Precision Time Protocol in order to achieve a synchronization between the nodes below 5 milliseconds. By synchronizing the clients with a separated server, the average time of synchronization was approximately 2 milliseconds in the worst case.

The second step was the analysis of the operations required to implement the Wave Field Synthesis, from which the convolution operation would be the most important. For the purpose of the application, which is to perceive the movement of the virtual sound as soon as the user changed it, its necessary to implement the convolution operation.

The third step was to analyze the different possibilities of implementing the convolution with a block size using the ARM processors. The time domain using the Overlap Add method was one solution and the frequency domain using the Overlap Save method was the second solution.

In order to be able to run on a real time level, the time available to prepare each block size to reproduce depends on the buffer size. For example by using a 512 integer buffer the time available to process is of approximately 11,6 milliseconds. In that time, the number of sound that can be processed is limited, and it depends on the number of speakers that are connected. For that reason in the fourth step the use the loop unroll technique, OpenMP, NEON instructions and the FFTW library were necessary in order to reduce the process time. As a result, it is possible to reproduce approximately 10 sounds for 7 speakers with a cost of below 10 milliseconds.

In fifth and last step, a configuration file was created using the LibConfig library, where it is necessary to explain all the configuration of the node regarding the number of sounds and speakers available and which device to use.

The important conclusion to point out is that it is possible to implement a distributed Wave Field Synthesis application by using the ARM processors in order to reduce cost and complexity of a classical centralized system.

5.2 Application of the Knowledge Achieve in the Master's Degree

The knowledge achieved in several subjects during the Master's degree was useful for the implementation the application like for example:

- **Parallel Computing: Concepts And Methods:** the multicore architectures, parallel computational models, parallel programming environments.
- **High Performance Computing Tools:** the basics of high performance computing, techniques for storing arrays and code optimization like the use of vector instructions.
- **Parallel Programming Technologies:** this showed me in more detail how to implement applications for the share memory multi-thread paradigm and message passing paradigm. It also given me an introduction parallel development tools software, especially for debugging and performance analysis.
- **Parallel Algorithms for Signal Processing:** the basic computational concepts in signal

processing. It also given me the basic information of sequential and parallel algorithms for signal processing.

- **Throughput And Scalability:** concepts of scalability, decentralization and data distribution for the client nodes.
- **Life Cycle of Services:** how to organize the code in order to facilitate actualizations.

5.3 Future Work

There are still a large number of improvements that can be added to the application. Some examples of improvements are listed below:

- Improve the translation of the sound from one position to another in order to eliminate sensation of sudden change.
- Implement a searching protocol, so it is not necessary to specify the IP of the server.
- Implement an interface for an web application, so it can be accessible from all the platforms. The actual application only uses the command line as interface.
- Implement the steaming protocol or a before starting to reproduce, send all the sound needed. The actual application needs to have the sound it will reproduce stored locally. If we want to add a sound, we have to add the file to each client node and to the configuration file. By implementing the upload file from the server side it would add simplification to the final application.
- Implement a way to modify the configuration of each node from the server side remotely. In order to modify the file configuration, we need to connect to the node and modify the config.conf file.
- Extract all the functions and create separate libraries. For example, one library would be the basic functions to play a song, or the communication library. That way if a change is required, we could modify all the files of the library and only have to link the new library.
- Study the use of the GPU that the Nvidia Jetson Tk1 has at its architecture, in order to obtain more performance from the board.
- Study the implementation of a system capable of detecting the position of sound using an microphone array instead of a speakers array.

5.4 List of Publications

- Mihaita Lupoiu, Jose A. Belloch, Enrique S. Quintana-Ortí, Alberto Gonzalez, Antonio M. Vidal "Implementación de un sistema Wave Field Synthesis (WFS) sobre nodos acústicos distribuidos", Procedings of TECNIACUSTICA 2015 pp.1.013-1.020, Valencia, Spain, October 2015.

5.5 Institutional Acknowledgments

This work has received financial support of the following projects:

- Project TIN2011-23283:"PA-HPC. Power-Aware High Performance Computing"(Spanish Ministry of Science and Innovation).
- Project TIN2014-53495-R: "COHEBACO. Computación Heterogénea de Bajo Consumo" (Spanish Ministry of Science and Innovation).
- TEC2012-38142-C04-01 PHASE II PROMETHEUS 2014/003 High Performance Computing Tools for solving Signal Processing Problems on Parallel Architectures. (PROMETEO/2009/013. Generalitat Valenciana. Spain).
- Project P1-1B2013-20: Fundació Caixa Castelló-Bancaixa.
- EU FEDER.

5.6 Personal Evaluation

With the development of the application I improved my skills of using C and got use to think at a lower level programming. I also learned how to design and implement a real message passing application, learned to take advantage of the architecture characteristics and improve my knowledge about Linux.

I would have implemented some of the improvements commented before and have an application easier to use for the user. Unfortunately given the lack of time I've focus on the functionality of the application more than in other aspects.

I hope in the near future the development of this application will continue, because it is a project where there are very few people researching and many aspects to contribute.

Appendix A

User Manual

A.1 PTPd

First it is necessary to install the ptpd daemon. It can be installed from the Advanced Package Tool by having available the universe package.

```
sudo add-apt-repository universe
```

Then it is necessary to update the package list:

```
sudo apt-get update
```

The command for installing the ptpd daemon :

```
sudo apt-get install ptpd
```

Once installed in order to be able to run the daemon is necessary to stop the NTP daemon if it is started.

```
/etc/init.d/ntp stop
```

To run the daemon as master:

```
sudo ptpd -W -b eth0 -C
```

To run the demon as slave:

```
sudo ptpd -g -b eth0 -h -C
```

The alternative is to install gcc and the essential library using the command:

```
sudo apt-get build-essential automake, autoconf libtool
```

Install git to download the last version of the ptpd daemon from the official repository:

```
sudo apt-get git git
```

And download using git and compile its contents:

```
git clone https://github.com/ptpd/ptpd.git
cd ptpd
autoreconf -vi
./configure
```

If we only want to install the slave version only:

```
./configure --enable-slave-only
make
```

Update test/client-e2e-socket.conf so that its "ptpengine:interface = " setting points to a network interface on the test machine that can see PTP packets from a grandmaster.

To test it in place:

```
./src/ptpd2 -c test/client-e2e-socket.conf
```

If everything is correct we can install the daemon:

```
make install
```

The the log output of the daemon is located in $/var/run/ptpd2.event.log$, the statistics output of the daemon in $/var/run/ptpd2.stats.log$ and the status file in $/var/run/ptpd2.status.log$

A.2 Server

In order to install the server it is only necessary to download the source code from my personal repository. Change the configuration file and run the main program indicating the port to use.

```
git clone https://github.com/MihaiLupoiu/SuperwavServer
cd SuperwavServer
vim ./config/default.cfg
cd ./src/
python main.py 4444
```

If we press the "f" key while the application is running it will ask which audio we would like to move and to specify the X and Y position. If we press the "e" key it will close the application.

A.3 Client

In order to install the client application it is also available to download from my personal repository:

```
git clone https://github.com/MihaiLupoiu/Comunicacion.git
cd Comunicacion
```

In order to be able to reproduce using the ALSA driver it is necessary to install:

```
sudo apt-get install libasound2-dev
```

The installation of libconfig is done in the “install.sh” file, but it can also be manually installed:

```
cd ./bin/configlib/libconfig-1.5.tar.gz -C /tmp
tar -xvf ./../../bin/configlib/libconfig-1.5.tar.gz -C /tmp
cd /tmp/libconfig-1.5/
sudo ./configure
sudo make
sudo make check
sudo make install
sudo ldconfig -v
```

To install the application for any platform use:

```
cd ./Comunicacion/client/src/
make
```

To install the application for the ARM platform use:

```
cd ./Comunicacion/client/src/
make ARM
```

To run the application first it is necessary to have the server running, and then use:

```
./SuperWAVAppClient.o 192.168.1.12 4444 -config ../config/default.cfg
```

The flag “-config” is used to indicate which configuration file does the application has to read. If there flag is not passed it will use de default.cfg.

Bibliography

[1] ALSA Project.
`http://www.alsa-project.org`. (accessed 2016 July 05).

[2] ARM SIMD instructions.
`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0002a/ch01s01s01.html`. (accessed 2016 July 05).

[3] Choosing between PTP and NTP.
`http://www.fsmlabs.com/news/2012/06/28/choosing-between-ptp-and-ntp.html`. (accessed 2016 July 05).

[4] Choosing the correct Time Synchronization Protocol.
`http://literature.rockwellautomation.com/idc/groups/literature/documents/wp/enet-wp030_-en-e.pdf`. (accessed 2016 July 05).

[5] Clion IDE.
`https://www.jetbrains.com/clion/`. (accessed 2016 July 05).

[6] Comparing NTP and PTP (2015).
`http://www.fsmlabs.com/news/2015/03/12/ptpvsntp.html`. (accessed 2016 July 05).

[7] Config Parsher.
`https://docs.python.org/2/library/configparser.html`. (accessed 2016 July 05).

[8] FFTW.
`http://www.fftw.org/`. (accessed 2016 July 05).

[9] Internet Time Synchronization: The Network Time Protocol. *online at: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=103043tag=1*.

[10] Introduction to PTP.
`https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Deployment_Guide/ch-Configuring_PTP_Using_ptp4l.html`. (accessed 2016 July 05).

[11] Jetson GPIO.
`http://elinux.org/Jetson/GPIO`. (accessed 2016 July 05).

[12] Jetson TK1, howpublished =
`http://elinux.org/jetson_tk1`, note = (accessed 2016 July 05).

[13] Libconfig.
`http://www.hyperrealm.com/libconfig`. (accessed 2016 July 05).

[14] Libconfig Manual.
`http://www.hyperrealm.com/libconfig/libconfig.pdf`. (accessed 2016 July 05).

[15] NEON.
`http://www.arm.com/products/processors/technologies/neon.php`. (accessed 2016 July 05).

[16] Network Time Protocol (Version 3) Specification, Implementation and Analysis.
`https://tools.ietf.org/html/rfc1305`. (accessed 2016 July 05).

[17] NTP and PTP (IEEE 1588) A Brief Comparison.
`http://www.en4tel.com/pdfs/NTPandPTP-A-Brief-Comparison.pdf`. (accessed 2016 July 05).

[18] openMP API Specifications.
`http://www.openmp.org`. (accessed 2016 June 05).

[19] PTPd Repository.
`https://github.com/ptpd/ptpd`. (accessed 2016 July 05).

[20] Server Socket.
`https://docs.python.org/2/library/socketserver.html`. (accessed 2016 July 05).

[21] Sockets.
`http://www.linuxhowtos.org/C_C++/socket.htm`. (accessed 2016 July 05).

[22] Sublime Text 3.
`https://www.sublimetext.com/3`. (accessed 2016 July 05).

[23] System Management: Network Synchronization.
`https://infoproducts.alcatel-lucent.com/html/0_add-h-f/93-0070-10-01/7750_SR_OS_System_Basics_Guide/System-Intro.html#3110746`. (accessed 2016 July 05).

[24] What is NEON?
`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0002a/BABIIFHA.html`. (accessed 2016 July 05).

[25] J. Belloch, M. Ferrer, A. Gonzalez, J. Lorente, and A. Vidal. GPU-based WFS systems with mobile virtual sound sources and room compensation. In *Proc. 52nd AES Conference*, Guildford, U.K., September 2013.

[26] J. A. Belloch. *Performance Improvement of Multichannel Audio by Graphics Processing Units*. PhD thesis, Universitat Politecnica de Valencia, 2014.

[27] A. Berkhout. A holographic approach to acoustic control. *J. Audio Engineering Society*, 36:2764–2778, May 1988.

[28] A. Berkhout, D. de Vries, and P. Vogel. Acoustic control by wave field synthesis. *J. Acoustic. Soc. Amer*, 93:2764–2778, May 1993.

[29] J. Blauert. Spatial Hearing-Revised Edition:The Psychophysics of Human Sound Localization. The MIT Press, 1996.

[30] E. Hulsebos. *Auralization using Wave Field Synthesis*. PhD thesis, Delft University of Technology, 2004.

[31] Jetson. Mobile GPU: Jetson. `http://developer.download.nvidia.com/embedded/jetson/TK1/docs/Jetson\_platform\_brief\_May2015.pdf`, 2015. (accessed 2015 November 22).

[32] Matlab. *online at: http://www.mathworks.com/products/matlab/.*

[33] L. Romoli, P. Peretti, S. Cecchi, L. Palestini, and F. Piazza. Real-time implementation of wave field synthesis for sound reproduction systems. In *Proc. IEEE Asia Pacific Conference on Circuits and Systems, 2008. APCCAS 2008.*, pages 430 –433, 30 2008-dec. 3 2008.

[34] J.-J. Sonke. *Variable acoustics by Wave Field Synthesis*. PhD thesis, Delft University of Technology, 2000.

[35] E. Start. *Direct Sound Enhancement by Wave Field Synthesis*. PhD thesis, Delft University of Technology, 1997.

[36] D. Theodoropoulos, G. Kuzmanov, and G. Gaydadjiev. Multi-core platforms for beamforming and Wave Field Synthesis. *IEEE Transactions on Multimedia*, 3(2):235–245, April 2011.

[37] A. Torger and A. Farina. Real-time partitioned convolution for ambiophonics surround sound. In *IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics*, pages 195–198, New York,U.S.A., October 2001.

[38] E. Verheijen. *Sound Reproduction by Wave Field Synthesis*. PhD thesis, Delft University of Technology, 1997.

[39] P. Vogel. *Application of Wave Field Synthesis in Room Acoustics*. PhD thesis, Delft University of Technology, 1993.