U.PORTO

MESTRADO
MULTIMÉDIA - ESPECIALIZAÇÃO EM MÚSICA INTERACTIVA E SOUND DESIGN

Development of Tools for Live Networked Musical Performance System using Smartphones

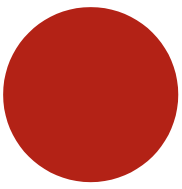
Alexandre Resende Clément

M

2015

FACULDADES PARTICIPANTES:

FACULDADE DE ENGENHARIA
FACULDADE DE BELAS ARTES
FACULDADE DE CIÊNCIAS
FACULDADE DE ECONOMIA
FACULDADE DE LETRAS



U.PORTO

Development of Tools for Live Networked Musical Performance System using Smartphones

Alexandre Resende Clément

Mestrado em Multimédia da Universidade do Porto

Orientador: Rui Luís Nogueira Penha (PhD)

Coorientador: Rui Pedro Amaral Rodrigues (PhD)

Junho de 2015

Development of Tools for Live Networked Musical Performance System using Smartphones

Alexandre Resende Clément

Mestrado em Multimédia da Universidade do Porto

Aprovado em provas públicas pelo Júri:

Presidente: Doutor Marcelo Freitas Caetano, Professor Auxiliar Convidado da Faculdade de Engenharia da Universidade do Porto

Vogal /Arguente: Doutor Paulo Maria Ferreira Rodrigues da Silva, Professor Auxiliar da Universidade de Aveiro

Vogal / Orientador: Doutor Rui Luis Nogueira Penha, Professor Auxiliar Convidado da Faculdade de Engenharia da Universidade do Porto

Resumo

Esta tese contempla o desenvolvimento de um conjunto de ferramentas de Pure Data, bem como a sua integração em aplicações móveis através de libPD, como parte de um projeto maior que abrange a concepção e implementação de um sistema que permita a interacção entre artista e público via *smartphones*. O projeto " *Bridging the gap between performers and the audience using networked smartphones*", doravante designado pelo seu nome interno – "Abel", tem como objetivo providenciar aos artistas uma forma de interagir facilmente com seu público, fazendo uso da sua contribuição e participação para criar performances participativas.

Este conjunto de ferramentas consiste de uma série de objectos para Pure Data específicos, desenvolvidos em C e acompanhados com exemplos de aplicação para ilustrar a sua utilização.

Usando a *suite* desenvolvida, os artistas multimédia poderão criar e distribuir conteúdo interativo de forma fácil para dispositivos móveis por meio deste sistema em rede, simplesmente através da criação de algoritmos de alto nível através da interface familiar do *Pure Data*, proporcionando-lhes uma forma imediata de utilização deste sistema.

Abstract

This thesis contemplates the development a suite of Pure Data tools, as well as their integration into mobile applications via libPD, as part of a larger project encompassing the design and implementation of a system allowing for interaction between performer and audience via smartphones. The “Bridging the gap between performers and the audience using networked smartphones” project, henceforth referenced in this document by its working name - “Abel”, aims to provide artists with a way to easily interact with their audience, making use of their input to effectively craft participative performances.

This toolset consists of a series of custom-built Pure Data external objects developed in C, accompanied with examples to illustrate their use.

Using the developed suite, multimedia artists can easily create and distribute interactive content unto mobile devices by means of this networked system, simply by creating high-level algorithms through the familiar interface of Pure Data, providing them with an immediate way of using this system.

Acknowledgements

First and foremost I wish to thank my project colleagues Rafael Henriques and Carlos Leocádio, who made working on this project a really enjoyable experience, whose work pushed me to try to do more and more and whose company and help ended up being the main "line-of-defense" against all the trials and tribulations that arose.

I also wish to thank Dr. Rui Penha for his support and guidance throughout this project (and for trusting my work enough to get me involved) and Dr. Rui Rodrigues for his availability, criticism and help both with my part of the work and with articulating with my colleagues' work.

Gilberto Bernardes for the help on the PD test patches and for all the criticism, ideas and suggestions.

I must also thank my former professor Pedro Santos, who first got me into audio programming, and whose review and support proved central in getting into this master's degree.

My family, who always saw more possibility in my future than even myself, and encouraged me to reach ever higher.

My girlfriend Ana for ever-present and continued support, for believing in me more than myself, giving me confidence and keeping me grounded on all the moments of doubt.

All my friends who helped and supported me in any way throughout this experience.

Dedicated to my parents.

Alexandre Resende Clément

Table of Contents

Figure list

Table list

Abbreviations and Symbols

AP	Access Point
API	Application Programming Interface
APP	Application (mobile)
APK	Android Application Package
BSSID	Basic Service Set Identification
GUI	Graphical User Interface
IDE	Integrated Development Environment
OSC	Open Sound Control
PD	Pure Data
SDK	Software Development Kit
SMELT	Small Musically Expressive Laptop Toolkit

Chapter 1

Introduction

Smartphone spread among population is growing rapidly and steadily, with a projected 50% spread in western countries by 2015 (Emarketer, 2014). Couple that with the devices' capabilities in the fields of multimedia and their growing array of diverse sensors, and they emerge as the perfect choice to use as a simple and readily available way of establishing communication and interaction between performers and audiences. By developing a system built on top of a dedicated communication network which allows the distribution of performance-specific content and data and the establishment of interaction between the performer's system and the audience's devices, a multitude of possibilities are created.

Even though the use of laptops and mobile phones in networked performance systems is not new, their role has mostly remained one of an instrument, dedicated to the performers on stage. Stanford University and Princeton University have both implemented performance systems based on laptop computers (so-called Laptop Orchestras) – the *Slork* (Wang, Bryan, Oh, & Hamilton, 2009) and *Plork* (Trueman & Cook, 2006). Since then, other implementations of the idea have been made, like the Carnegie Mellon Laptop Orchestra (Dannenberg, Cavaco, & Ang, 2007) or the Linux Laptop Orchestra (Bukvic & Martin, 2010). All these have in common the fact that they consist of a network of interconnected laptops, used simultaneously to reproduce a musical piece/performance. This kind of approach has also been taken to mobile phones, ever since 2001 with Golan Levin's "Dialtones (A Telesymphony)", which made use of the audience's ringtones as an instrument (Levin, 2001), by calling each one at a given time. This approach is, in a particular way, related to this project's, as it makes the audience a part of the performance. The major distinction lying in the audience's role: "Dialtones" makes the audience a passive participant in the performance, while this project aims to make it an active one.

Ever since 2001, however, much has changed in the field of mobile phones. Current smartphones, with increased processing, stability, number of sensors, and overall power and

precision have become a great tool not only for use as instruments in a given performance, but to also work as "feedback" mechanisms, establishing genuine interaction and interactivity. The more recent Mobile Phone Orchestra or "MoPhO" at Stanford University (Oh, Herrera, & Bryan, 2010; Wang, Essl, & Penttinen, 2008) takes advantage of the more recent device's capabilities even though devices remain as instruments used by performers on stage.

Since the main objective of this project is to allow composers to make use of audience mobile devices as part of their performance, two main issues are to be addressed:

- Firstly, a framework for multimedia content should be developed, able to run both as in musical piece/composition context and as embeddable content. This framework should provide a simple and unobtrusive way for the composers to focus on their creative work and not on the technical specificities.

- Secondly, a targeting system should be developed, allowing the composer to directly target a given specific group or section of the audience whenever needed. This would effectively allow him to, much like a maestro, address it as an independent entity and assign a specific function or behavior to it in the context of his piece.

By building this described system, we hope to allow for a performance to operate in a "Performer – System – Audience" model such as described by Bert Bongers (Bongers, 2000).

1.1 Motivation

My involvement with music dates to a long time back, from the piano lessons at 6 years of age. From that moment on it has been an integral part of my growth and development. From formal music training to garage bands, from classical to heavy metal or electronics, music has always been one of the most defining aspects of my life. At the same time, my first computer science and software development experiments also date back, although not as much. Around 16 years of age I first programmed on Visual Basic and on my Texas Instruments calculator. From then on, programming and development has also accompanied me in life. When, back in 2009, I was admitted to the Music Production and Technologies specialization of the Bachelor in Music at Porto's Escola Superior de Música, Artes e Espectáculo, I had my first contact with things that united these two passions of mine. From synthesis to audio programming, a new world opened up before me, and has been my day-to-day ever since.

This thesis contemplates a work that is part of a larger project, encompassing the design and implementation of a system allowing for interaction between performer and audience via their smartphones, taking these devices' role beyond that of personal devices, and bringing choice literally into the audience's hands.

From a personal point of view, both as a music enthusiast and as a musician, the possibility of helping develop a system that not only allows but encourages the creation of performances that create a connection between the performer and the audience is highly motivating.

Furthermore, as a multimedia developer, the possibility of experimenting and forwarding my own knowledge in the fields of mobile audio development and, in particular, of Pure Data development greatly contributed to the appeal this project had to me.

1.2 Objectives

Client/audience-side assets are to be created by composers with Pure Data, and integrated into Android/iOS applications through the libPD wrapper library, while main/performer-side assets consist of a standalone Pure Data application[1]. Both client and server patches will take care of all audio and visual manipulation and feedback.

On the server/performer side, operation will be 100% controlled and designed by the composer, both from previously created processes and on-the-fly manipulation. The client-side application, on the other hand, will operate with minimal input from the user with the previously created embedded patches taking care of all audiovisual content associated with the performance, via information gathered both from composer/performance provided data (through network communication) and from the device's own sensors.

A toolset of external objects was thus needed to give composers easy access to the system's data communication, audience's mobile devices' functionalities (sensor data or user interface feedback), and all other system's functionalities. Taking care of all data parsing, structuring and communication by themselves, removing any kind of technological know-how other than Pure Data operation out of the equation, this toolset needed to be as straight-forward and simplified as possible, allowing it to become an unnoticed part of the workflow of the composers.

1.3 Tests

Considering the nature of the global project in which this particular work is inserted, a number of tests are necessary to assess the system's performance in view of future work and further development and refinement of said system. Usability issues which can only arise with a functioning prototype should also be assessed at this stage.

After the implementation of the proposed prototype, tests were run to assess:

- Performance of developed assets

[1] Please refer to section 2.2 for an in-depth overview of Pure Data

- Adequacy of value scale and information structure
- Impact of technological specifications of mobile devices on final result
    - Audio and visual perceived synchronization
    - Device response time (from network message reception to device reaction)
    - Processing capabilities (especially on low-end Android devices)
- Results of some desired events to be used in performance (audio/visual crossfades, particular targeting)

1.4 Structure

In addition to this introduction, this dissertation is comprised of 4 other chapters.

In chapter 2, a global bibliography and project review is done, going over selected and important publications concerning this system's context, as well as some similar or in some way related project.

Chapter 3 explains the approach to this particular project, explaining the proposed tools and the structure, usage and objective of each tool. It also documents the development of the objects, going over any particularity for each (commented source code is included in Appendix A – "External object commented source code").

Chapter 4 goes over testing results.

The fifth and final chapter presents conclusions and future work.

Chapter 2

State of the Art

2.1 Introduction

Mobile phones are nowadays a common part of everybody's life. They have become part of most countries inhabitants' lifestyle. Smartphones in particular have garnered great popularity, bringing complex and powerful interfaces, alongside a multitude of additional functionalities into the day-to-day of their users. Music, already a common staple in most people's life, has found in smartphones a new approach. Multimedia capabilities of smartphones and mobile devices bring to the consumer a new myriad of resources, previously only available on personal computers. In that field, however, mobile phones have been used primarily as consumer devices, doing little more than a common multimedia player would do, while bringing some social aspect into it. In the context of concerts they have always been considered as little more than nuisances, frequently addressed at pre-performance time with notices asking for them to be turned off or muted. Nowadays smartphones have mainly become a way to record and/or share a performance on social media, which is also frequently considered as disruptive of the concert experience, since it shifts the focus from the performance to the device. Nonetheless, with the advent of technology and the fast development and evolution of smartphones, mobile phones have gained the possibility to function not only as reproduction and consumption device, but rather as control devices, directly integrated into music making.

Interaction and interactive devices have also changed, as has their adoption in media and, in particular, music. The British band *Coldplay*, for example, has been using, since 2012, interactive devices consisting of simple bracelets with embedded LEDs and an RFID chip (Pixmob, n.d.; Xylobands, n.d.). This allows the band to make the audience part of the performance, by activating the devices and producing light effects. Still, this remains as passive one-way communication, with the performer bringing the audience into the performance but still removing from them the

possibility of bringing something in by choice. This is an example of the growing desire among the artistic community to make the audience a part of the performance, instead of just passive consumers, allowing them to contribute in some way to the end result.

Networked systems have also been used as part of the performances, although mostly as static parts of the performance, like in the case of the laptop orchestras. Elements interact with each other to create a performance where the individual composer/performer is responsible for the establishment of the rules, much like a composer would go about creating a musical piece for a "conventional" orchestra.

Most digital music making tools target a single user, and thus have mostly been used by solo artists or by individuals within a larger, un-networked group. The performer uses said tools as way to operate, mutate and create his work, all from his own input and as a primarily singular means of interaction.

Building upon John Cage's pursuit of performer-audience interaction as the core of a performance's creation in itself, as a mutable, ever-changing entity, it should be possible to integrate both aspects into a single system, while at the same time introducing a second direction of communication. Instead of regarding the audience merely as end-receivers, consider them as contributors to the performance in itself.

2.2 Development Environments, Toolkits & Utilities

There are a number of possibilities from which to choose when aiming at audio development. Before listing some of the most notable options, a brief distinction between concepts is important:

- **IDE**: these are self-contained editors with specific support for developing. Typically an IDE operates on its own, with all necessary tools available integrated or accessible from its interface.

- **Toolkit** (also referred to as SDK): is a library which provides specific functionalities and tools aimed at developing software for a specific system. It doesn't feature development tools, but rather facilitates code and specific functionalities adapted to the specificities of the system it is built around. It needs an external tool for development.

2.2.1 IDEs and Programming Languages

These development environments and/or programming languages provide abstract ways to create audio specific software without having to deal with particular details such as audio engine implementation, generator coding, etc. By abstracting a given set of operations/algorithms in this

manner, the programmer/creator can focus immediately on the musical and sonic part of the implementation, instead of having to manually implement those low-level functionalities.

There are some other environments and languages that allow for audio manipulation (e.g. Csound and Faust programming languages, C++ and Python audio engine libraries, Matlab and other math-based scripting solutions, Native Instruments' *Reaktor*), but these are the ones which provide a higher-level of abstraction while retaining enough complexity for advanced customization, and at the same time are the most common in the context of interactive/real-time music/sound performance and generation.

2.2.1.1 Max/Msp

Max's development started with Miller Puckette during the 80s, while at IRCAM, as *Patcher,* an editor taking care of MIDI and control processing, communicating with outboard systems. In 1990 a commercial version was released by Opcode Systems, developed and modified by David Zicarelli. The software ended up being dropped by Opcode Systems and being picked up by Zicarelli's own company, Cycling'74 in 1999. It has since been developed and commercialized by the same company. In 1997 David Zicarelli modified the audio engine, including some innovations and enhancements, and released it as the *MSP* package for Opcode's Max. In 2003, a video processing package was introduced (Jitter) and in 2011 a code-compiling package was added (Gen). (Cycling 74 Website, n.d.; IRCAM Website, n.d.)

Max (general name) is a graphical programming environment, commonly used in music and multimedia. It is developed for the Windows and OSX operating systems. It is commonly referred to as a *building-blocks* environment, consisting of a canvas and a set of graphical modules, each encapsulating a code block responsible for a given function. Each module has built-in inputs and outputs, depending on its functionality. Module interconnection is achieved graphically by connecting each with the aid of lines (called patch-cords), symbolizing information flow. This module set can be expanded with user-created functions/modules, called externals. The whole environment consists of 4 main packages, each specializing in a different type of information/value handling and processing:

- MAX: this is the base package, and comprises the graphical user interface, timing, communications and MIDI support. The patch-cords for MAX information are solid black.
- MSP: this package handles real-time information manipulation, and is aimed mainly at audio synthesis and digital signal processing. Patch-cords for MSP information are dashed and alternate yellow and black dashes.
- Jitter: this package is aimed at video and matrix data processing. Its patch-cords are dashed, with green and black dashes.

- Gen: this package features an integrated patching canvas with a custom set of modules, derived from the core Max/Msp modules, and a code editor. It is possible to design audio algorithms graphically in the same way as in a regular Max patch, or integrate parts via code. The result of these sub-patches / gen modules can be used as modules in other Max patches.

Max is currently in its seventh version, and aims at simplifying as much as possible the development of multimedia content, as it doesn't require any programming experience and allows for strictly visual-based editing and creation.

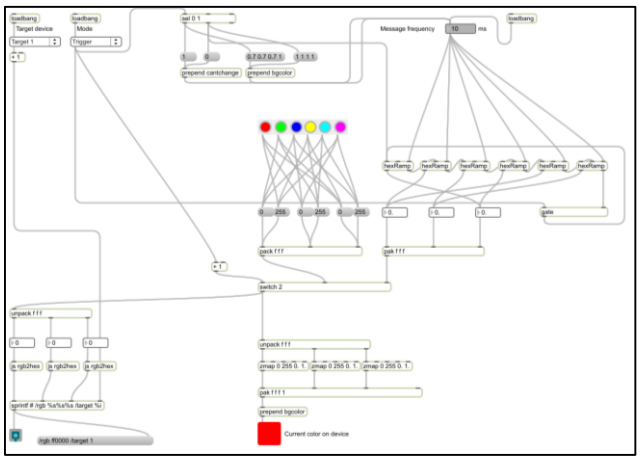


Figure 1 – Max 6 example patch

2.2.1.2 Pure Data

In 1996 Miller Puckette, the original creator of Max, started developing Pure Data aiming to correct some weaknesses in Max.

It consists of graphic-based or visual programming audio-specific development IDE, and is one of the most widely used tools in the field of computer-assisted and computer-based musical composition. It operates with graphical blocks corresponding to specific algorithms (referred to as "objects"), connected between themselves in a graphical way so as to be arranged in larger, more complex sound processing and generating programs (referred to as "patches"). As can be seen by comparing Figure 1 – Max 6 example patch and Figure 2 – Pure Data example, Pure Data and Max/Msp's *modus operandi* is really similar. PD has, nonetheless, a more simplistic GUI. It also allows external expansion via *externals*, but has a major difference from Cycling '74's Max: its open source nature. By adopting the open source approach, several new possibilities arise, most notably the access to the core code aspects of the engine, its customization and recompilation and even redeployment. (Puckette, 1997a, 1997b)

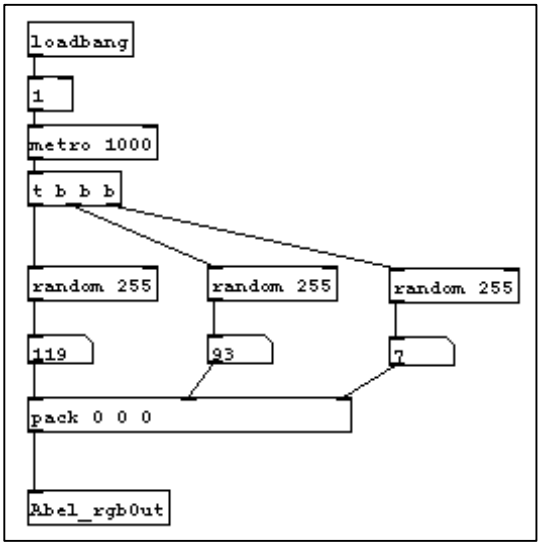


Figure 2 – Pure Data example patch

One relevant example of this repurposing aspect brought by PD's open source nature is *LibPD*, a library for external inclusion of Pure Data's audio engine into other software. Like stated, *LibPD* is a repackaging of Pure Data to allow its inclusion into other software solutions, stripping the graphical part of Pure Data and trimming some parts of its core engine, while keeping its audio engine and base generators/functions available. (Brinkmann, Mccormick, Kirn, Roth, & Lawler, 2011) This opens the possibility of using Pure Data patches in other software, thus greatly simplifying the process of audio algorithm/processing design. For example, *LibPD* has been used in videogames to allow the creation of real-time procedural sound effects and integrated into mobile applications as a sound-generation tool, while other software makes use of its already implemented audio engine to skip that implementation phase.

Another worthy example of the customization of Pure Data is tied to the L2ORk project (as described in section 2.3.1.1 – "The *Ork paradigm") and consists of PD-L2Ork. This distribution of PD was created in the context of the L2Ork project, based on PD-extended and focusing on core engine enhancements, GUI modifications and improvements and visual editor customizations, suited for use within the ensemble.

2.2.1.3 SuperCollider

> "SuperCollider is a dynamically typed, single-inheritance, single-argument dispatch, garbage collected, object-oriented language similar to Smalltalk."(McCartney, 2002)

SuperCollider was created in 1996 by James McCartney for real-time audio synthesis and algorithmic composition. It is an environment designed to sit halfway between a high-level programming language and a synthesis engine. It features a built-in programming language with, among others, an object-oriented class system, a GUI builder, a graphical wavetable and envelope designer, built-in signal and musical data processing and synthesis functions library. The same language is used to create the audio processing algorithms and the musical performance aspects. Its program flow is tied to the audio sample rate in which it is set to work: each program "cycle" will correspond to one sample. Audio synthesis, signal processing and any other function will output its result once per sample instead of just providing results "as fast and many as possible". These results are then stored in a buffer which corresponds to the end audio buffer to be reproduced or stored.

2.2.1.4 ChucK

ChucK is an on-the-fly-programming language developed at the Princeton Computer Science Department. This means that it's a programming language that the programmer modifies in real-time without the need to stop or restart the program. It is implemented as a virtual machine with a virtual instruction set (Wang & Cook, 2003) and (Wang & Cook, 2004).

ChucK derives its name from the common name given to the "=>" operator, which symbolizes flow of information (from left operand to right operand). This is the basis of ChucK programming, with said operator working as a connector between a given number of generators, in the form of:

$$gen1 => gen2 => gen3 => gen4$$

(Generator identifiers non-existent in ChucK, only for illustration purposes)

This information attribution/feeding from one operand to the next is commonly referred as *chucking*, and can be chained (as in the previous example) and nested (with parentheses) just like a regular math operation. The other basis of ChucK functioning is time. Most sound generation/manipulation ChucK operations are time-dependent, and it is up to the programmer/composer to control said timing. ChucK programs are organized in *shreds*, which are simply code blocks that can run in parallel (much like a parallel thread in traditional OOP). These are also used to manage multiple input/outputs (multiple MIDI channels inputting information simultaneously to different program parameters, for example)

2.2.2 Toolkits

2.2.2.1 Small Musically Expressive Laptop Toolkit

> "SMELT is an open-source toolkit to facilitate rapid development of and experimentation with expressive musical interfaces built on the laptop's native physical input capabilities (e.g., keyboard, mouse, motion sensing, and microphone). It's implemented in C and ChucK, and based much on our work with PLOrk." (Fiebrink, Wang, & Trueman, n.d.)

There is an extremely wide range of external inputs to be used for real-time manipulation, but each of them has particular methods and usage types, which proves to be a hurdle or hindrance to their easy inclusion into a laptop orchestra or otherwise meta-instrument ensemble. SMELT targets the laptop's various built-in input methods, and tries to standardize a number of possible interactions with them, making its usage faster. Keyboard, trackpad, webcam, microphone and speaker are components pretty much any laptop has carried for the last 10 years or so. By standardizing a set of actions and interactions with each, this toolkit provides an easy in-box access to a decent number of input methods to be used as control devices for musical/performance, without the need to use any external devices (thus avoiding the customization/testing time needed to integrate those).

2.2.2.2 NRCI Pure Data tool suite

This suite of Pure Data tools was developed at the Wisconsin-Milwaukee University to facilitate laptop ensemble performance (Burns & Surges, 2008). It was initially created targeting the Milwaukee Laptop Orchestra (MiLO) while aiming to be as open as possible, allowing for easy usage from any other ensemble.

> "NRCI provides both a friendly welcome to custom software design for novice users, and near-instant gratification which motivates more advanced learning and design" (Burns & Surges, 2008)

It is developed in the form of PD abstractions (as it is the most widely used software in MiLO) with both novice and advanced users in mind, providing modular, reusable and modifiable tools to facilitate rapid prototyping, serving at the same time as a test bed for its developers. It is not supposed to be the sole software in use for the MiLO, but its aim is to become so easy to use that it becomes the go-to choice. It draws some of its inspiration from the SMELT toolkit,

developed at Princeton University, providing abstractions to take care of most stages/components of building a laptop orchestra instrument, from networking to input parameterization, from timing handling to audio generation. The idea is to provide an empty canvas and give the user the tools to quickly and easily implement instruments, and allow for their real-time manipulation and modification.

2.3 Cooperative, Collaborative and Networked Music Systems

2.3.1 Laptop based Ensembles

2.3.1.1 The *Ork paradigm

The Princeton Laptop Orchestra (PLOrk) is an ensemble of 15 computer-based meta-instruments created in 2005 at the Princeton University. Each meta-instrument consists of a laptop, multichannel hemispherical speaker, a number of control devices and control/performance software.

The PLOrk consists of 3 different layers/ components: a set of Max/Msp abstractions responsible for mapping input from devices to the sound processing/synthesis components, a frontend layer responsible for the saving and loading of composition related presets and a set of network utilities responsible for the communication between the conductor and the meta-instrument machines, allowing broadcast type messages or single-machine targeting. These network utilities operate over OSC and allow easy communication with any other OSC-enabled hosts. Communication operates on a wireless network and has a measured latency of 30-40ms taking into account synchronization between all of the 15 machines.

One of the problems that arose early in the project was how to synchronize the performance. Traditional orchestras follow a conductor, but this ensemble could benefit from network protocols and synchronization. A system was developed to allow for inter-performer and performer-conductor communication through networking protocols. It was possible to integrate this communication system into the performance software (ChucK and Max/Msp) and also use it for tempo synchronization. Most pieces performed were created in either or both the software environments, although other pieces created in or making use of SuperCollider and Java have been performed by the ensemble too.(Trueman & Cook, 2006)

Interacting with these meta-instruments of the ensemble proved to be a challenge, as did composing for it. Even though it builds up on the traditional concept of an orchestra/ensemble,

the particularities of its instruments, both in terms of sonic capabilities and spatialization approach became an open and unknown field.

Figure 3 – A view from within PLOrk (Trueman & Cook, 2006)

The PLOrk set an implementation paradigm for laptop orchestras that was followed by many other ensembles subsequently. Ge Wang transitioned to Stanford University founding an implementation of the so-called *ORK based ensemble there – the SLOrk. It is in all aspects similar to the PLOrk, drawing from the same inspiration and making use of different tools developed in the context of the later. The configuration of the meta-instruments follows the same approach, consisting of a six-channel hemispherical speaker, an audio interface, input devices and laptop. Its software is based on the ChucK programming language, developed at Princeton University, and it makes use of the "Small Musically Expressive Laptop" toolkit as simplified way to rapidly prototype laptop instruments, also developed at Princeton University. (Wang et al., 2009)

Partly inspired by the successes of PLOrk and SLOrk and in part encouraged by the rapidly developing Linux hardware and software support, in the fall of 2008 DISIS (Digital Interactive Sound & Intermedia Studio) partnered with Virginia Tech's College of Engineering to explore the ensuing synergy and form L2Ork, the first *Ork based on Linux. (Bukvic & Martin, 2010)

Its meta-instrument components are in all aspects similar to the PLOrk and SLOrk, with the particular aspects that it was created targeting a maximum of $800 cost per workstation, and that it standardized the input device used. In terms of software, in the absence of the Windows/Mac exclusive Max/Msp, most of the development environments remain the same as in the 2 other laptop ensembles, notably ChucK, SuperCollider and Pure Data, all of which are multi-platform.

2.3.1.2 The Carnegie Mellon Laptop Orchestra (CMLO)

The CMLO is a collection of computers that communicate through a wireless network and collaborate to generate music.(Dannenberg et al., 2007)

It differs from the PLOrk mainly in its approach to the orchestra, choosing to maintain a more traditional concept, both in organization and instrumentation, and in the fact that there is no set number of instruments (new ones can join in and existent ones can exit the network at any time). Since it was created as part of a Computer Science degree, its approach to music generation takes a somewhat secondary role in face of the networking and communication design and implementation aspect. Communication is not made via OSC messages, but rather through a specifically created protocol, in this case over TCP/IP.

The system is built around a central hub machine, which works both as a conductor and as a provider of the performance's information (e.g. key, tempo, time signature, musical style). The client machines take conventional roles, such as drummer, bass player, chord player, melody player and arpeggiator. Taking these roles as base, the generative algorithms are free to compose, as long as they respect the information given by the hub.

The system's main program is responsible for setting up the main performance's details, as described above, and communicates said information to a program called Harmony, responsible for algorithmically generating a chord progression which is then sent to the individual "musician" clients in the network. These messages are time stamped and sent before real-time, so as to give time to the clients to make any necessary calculations and changes and still remain in sync. Client programs poll the main system at regular intervals, getting all incoming messages, and feature a scheduler which applies all necessary actions on the beat defined in the received messages.

2.3.1.3 The World Laptop Orchestra (WLO)

This is an ensemble of 50 performers each using live laptop computing, which gave its first performance in 2007.Its goal was to perform powerful, auditorium filling musical works over a multichannel PA system, with spatialization techniques that could be accessed independently of the physical layout of the orchestra. (Harker, Atmadjaja, & Bagust, 2008)

Contrary to the PLOrk ensemble, the WLO did not have a strict positioning of the musicians, whose location and presence might vary. Hence, mobility was one of the main issues dictating the use of WIFI technology as a communications means, and UDP as a communication protocol. Software was developed using C++, Max/Msp, Pure Data, Python and Google Earth.

2.3.2 Mobile phone Orchestras

While mobile phones have been used for artistic expression before, it isn't before 2007 they are used as part of an ensemble, in a similar way to the several laptop orchestras already in place.

In the aforementioned "Dialtones (A Telesymphony)" piece (Levin, 2001) participants registered their mobile phone numbers prior to the concert and, subsequently, had access to custom ringtones downloaded onto their phones, associated with a specific seat that was assigned to them. The performers could then, during performance, call specific phones that were at pre-determined specific locations and with specific ringtones, creating musical and spatial patterns at will. Nonetheless, this approach still viewed and used the mobile phone as a passive instrument, without interaction or participation.

Mobile phone ensembles have, since, been implemented, drawing inspiration from the aforementioned laptop ensembles, which make use of the phones as active parts of the performance creation. MoPho, the Mobile Phone Orchestra of CCRMA(Wang et al., 2008), is the foremost example of this approach. It originally consisted of 16 mobile phones (Nokia N95) and players, each corresponding to a gesture-driven instrument. Equipped with a 330MHz CPU and running Symbian OS, the system designed for use in the MoPho was developed partly in C++ (for the audio synthesis and hardware access) and Python (for the graphical interfaces). This is where one of the main differences between this orchestra and laptop orchestras resides: whereas the laptop orchestras where open to any number of input/control devices, either inbox or out-box, these instruments rely solely on the in-box input devices. Although the available hardware is diverse enough (5MP camera, front camera, microphone, speakers, 20-button keypad, 3-axis accelerometer), these constraints impose a performance limitation when creating a piece for this ensemble, whereas in the case of a laptop orchestras external devices might be added as needed and desired. The lower computational power of the devices also limits the complexity of the meta-instruments' sonic capabilities in terms of audio synthesis and processing. Sound reproduction is also limited to the phones' built-in speaker (and occasional musician vocalization).

From its original creation, the MoPho has since been updated. From the original N95, it switched to using iPhones as the mobile device of choice, and starting making use of a framework developed specifically for it – the Mobile Music Toolkit (MoMu). A number of hardware changes and additions also were implemented, namely in including mobile phone speakers, so as to provide extra amplification and allow for bass frequency boosting. (Oh et al., 2010)

A number of orchestras and ensembles based on this new incarnation of MoPho have appeared, all based on the same toolkit, hardware and general premise. From the KAIST Mobile Phone Orchestra, to the Helsinki Mobile Phone Orchestra or the Michigan Mobile Phone Ensemble, all are derived from the original MoPho at Stanford.

Figure 4 – The Stanford Mobile Phone Orchestra (Oh et al., 2010)

Another example of this approach lies in the Casa da Música iPhone Orchestra (Carvalho, 2009), created in 2009 as an initiative of the Digitópia project, supported by the Educational services of Porto's Casa da Música. This ensemble, organized by Rui Penha and Filipe Lopes, was presented in a public workshop, where people were invited to become an active part of it. The ensemble as a whole acted as controller for the robotic gamelan instrument at Casa da Música, with each of the iPhones being assigned to one of three distinct sections of the instrument. By choosing and changing parameters on the custom application developed for this purpose, the participant would control some of the robotic instrument's mechanic actions and, alongside other devices, create a unique performance.

2.4 Mobile apps

2.4.1 Cooperative/collaborative apps

- **MoodifierLive** is a mobile application for interactive control of rule-based music performance. (Fabiani, Dubus, & Bresin, 2011). It is written in Python and designed to run on Nokia S60 phones with the PyS60 interpreter, and aims at combining automatic performance with gesture analysis. It operates by giving the user several different modes of operation, manipulating the musical performance by means of his gestures.

- **Mobile Phone Orchestra** is a free iOS app which plays unique five minutes compositions by taking small snippets from the user's media library. It requires at least four iPhones which play together a unique performance. The phones are placed close together (20-100cm) and can be arranged in any spatial pattern. The app performs the composition based on the "now playing" song on the music player. If no song is playing, the app selects random songs from the library (Bluff, n.d.)

2.4.2 LibPD based apps

These applications allow the user to upload externally created PD assets into a mobile application.

- **RjDj**, by Reality Jockey Ltd. is considered the original PD-based musical app. RjDj allowed to load Pure Data patches into an iOS app, with minimal GUI support. It is now defunct and unavailable, but allowed the user to get data from a number of sensors on the device and using simple image and text visual output.(Kincaid, n.d.)

- **ScenePlayer** is Android's version of RjDj, and is mostly compatible with the former. It is included in PdForAndroid, the libPd Android distribution ("PDforAndroid GitHub," n.d.).

- **MobMuPlat** is a free (although closed-source) mobile application. Its use comprises two distinct parts: the mobile application in itself, and a GUI creation application (MobMuPlat Editor). The former is the one responsible for loading both the Pure Data assets and the created GUI and allowing both to be run and operated, while the latter is responsible for creating the frontend for the developed patches, from control inputs to visual arrangement (Iglesia, 2013).

- **PdDroidParty** runs all the core audio functionalities just like *libPD* and, in addition, renders some of Pure Data's GUI elements as interactable Android controls (number boxes, sliders, toggles, bangs, comments and canvases). It also implements a number of particular objects, allowing the user to use input types not particular to Pure Data via GUI controls, as well as communicate internally between patches loaded by PdDroidParty (Mccormick, 2011). PdDroidParty is open source and available as a GIT repository (Mccormick, n.d.).

- **PdParty** is to iOS as PdDroidParty is to Android (Wilcox, n.d.)

- **mPD** has little information other than its Google Store page and a thread by the author on the Pure Data forum presenting the release of the application. The application appears

to attempt to translate Pure Data's GUI onto a mobile application, providing a usage equal to the regular usage of the Pure Data IDE (Viejo, n.d.).

2.4.3 OSC control apps

OSC control apps are slightly different considering they aren't, by themselves, tools for music creation or collaboration tools. However, by allowing to remotely control other tools or software and to establish communication over a network, they also fall into the context of this work. These kind of applications are nowadays very common, making it impossible to list them all. Some do stand out and are worthy of mention, either because they bring some particular aspect or are especially different in their approach, but a search for "OSC" in either the Google Play Store or Apple Store wields a very long list of both free and commercial applications, ranging from customizable control interface to simple sensor data output.

- **Control** is available for both Android and iOS, and has a number of particularities that make it a distinctive application. In addition to the normal sensor access, it features MIDI output (in addition to OSC), dynamic interface creation via JavaScript and dynamical interface "pushing" via OSC (Roberts, n.d.)

- **TouchOSC** also is available for Android and iOS, It also allows for MIDI communication and allows advanced customization of the interface with a wide variety of control widgets, to better adapt it to what is being remotely controlled ("h e x l e r . n e t | TouchOSC," n.d.)

- **Lemur** is an advanced OSC controller application which implements customized controls and custom code editor, allowing for development of complex control widgets. It also features an in-app sequencer widget ("Lemur – Liine," n.d.)

2.5 Summary

It becomes clear that neither the notion of computer/mobile based collaborative systems nor that of embeddable asset mobile applications have anything really novel or new. Both concepts have been approached and implemented in a growing number of different ways. However, considering the aforementioned projects and systems, joining both ideas is something somewhat different from what has been done so far. Networked collaborative systems are mainly in-house dedicated systems with specifically designed APIs and frameworks, forcing anyone wishing to use the system either as composer or participant to learn it, making previous experience and knowledge something to be ported instead of immediately applied.

The listed libPD based applications aim at addressing that issue: allowing users with previous knowledge and experience with a free, user-friendly, powerful and easily accessible development environment to make use of when creating mobile based interactive musical assets.

In the end of this reviewing and analysis process the choice befell onto Pure Data. Using Pure Data allows the system to take advantage of its implantation into the field of multimedia arts, and in particular into sound processing and electronic and electro acoustic music composing, and the corresponding knowledge base that comes associated to it. With the objectives of universality and exportability in mind surrounding the idealization and conception of this present system, the best choice would be, logically, a well-known and documented, free and multi-platform base software. The existence of the libPD mobile port, and the open-source nature of both PD and libPD make this an almost immediate choice, by allowing a greater degree of abstraction while developing the multi-platform mobile applications as well as the customized functionalities to be handled via custom external objects. By making use of the already multi-platform implemented PD engine, a tested and documented framework is available immediately, and leaves the door open for more extensive and low-level customizations or adaptions to the system, if need be, without having to code a framework from scratch.

Chapter 3

The Abel project

3.1 Overview

This system is intended to serve at least two distinctive groups of people: artist/composer and user/audience. For composers it needs to facilitate an easy integration, and inflict as few disruptions as possible unto their existing workflow, providing straightforward tools which will allow them to develop their content and distribute it to the audience's devices with simplicity. For the user/audience, it should serve as a transparent and immediate way to participate in a given live performance, with minimum required actions apart from his interaction with the device.

There are four major development premises for such a system:

- Develop a way of connecting hundreds or thousands of mobile devices to a network in a concert environment with guaranteed stability, trustworthy data communication and ease of implementation and deployment
- Develop a solution for localization and synchronization of mobile devices, providing a way of specific targeting, and ensuring audio and visual feedback on the mobile devices maintains the desired timing and sequencing
- Develop a mobile application which serve as host to distribute performance specific content and provide a way for the user to participate
- Develop tools, templates and content examples which will allow to easily integrate the creation of assets for this system into a pre-existing workflow

In the aim of developing a prototype which could cover these 4 premises, work was divided between 3 people, each assigned to a specific area of development. Network infrastructure and

implementation to be handled by Carlos Leocádio, mobile application development by Rafael Henriques and content creation framework development to be handled by myself.

3.1.1 Overall system description

The system can be divided in three major blocks, establishing a chain of communication and dataflow which results in the desired system: main application/client, network infrastructure, and mobile application/client.

The main application will consist of the developed performance material (musical piece) and the networking management application, which will use the implemented network infrastructure to send data to the mobile application. This mobile application comprises both the native application and the distributed performance specific assets.

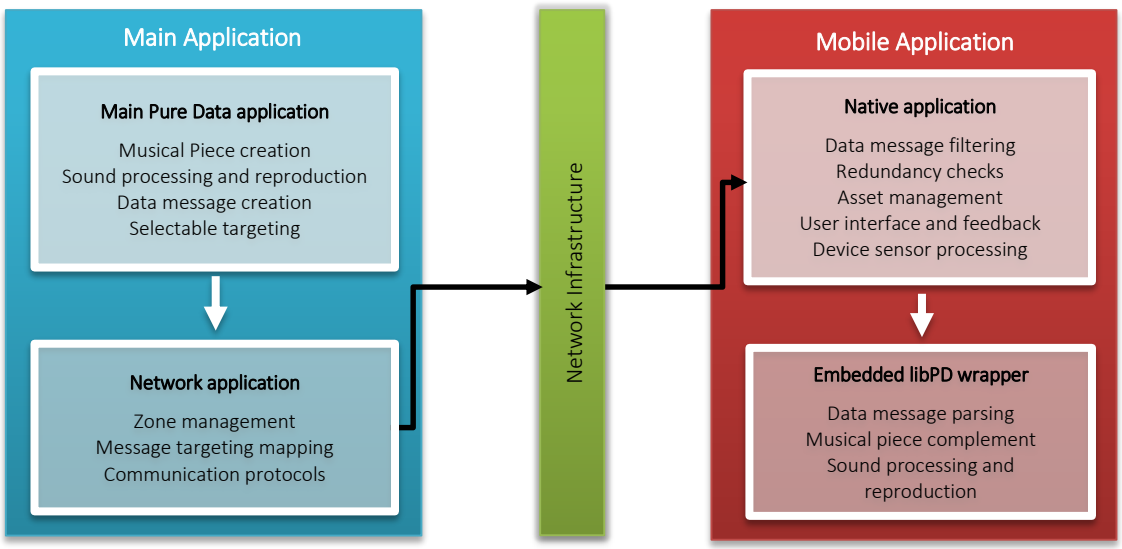


Figure 5 – Global system structure and data-flow

There is the need to create adapted content to be used as main musical performance (musical piece), and as audience distributable assets.

When creating these assets, composers should not have to worry about network communication, device sensor access, or otherwise deal with any of the inner workings of the system. These assets, corresponding to the "Main Pure Data application" and "Embedded libPD wrapper" sections of figure 2, must allow the composer to make use of the listed functionalities. Musical piece creation and sound processing and reproduction tasks are natively handled by Pure Data, but message creation and parsing, targeting, sensor access and all other system-specific functionalities are not. The work presented in this dissertation is linked to the creation of a toolset

to allow easy and straightforward creation of said assets, and its integration onto both the main and mobile applications from within the scope of regular Pure Data development.

3.1.2 Localization

The system is designed to allow specific targeting for data messages through the implementation of a virtual sub-division of the physical audience area into targeting zones. These zones are predefined when creating the musical performance in itself, and attributed according to physical location of the device in the audience area. This zone assignment is connected to the venue's ticket system, which will allow for easy placement of the device in the venue's audience, and automatic inclusion in one of the mapped targeting zones.

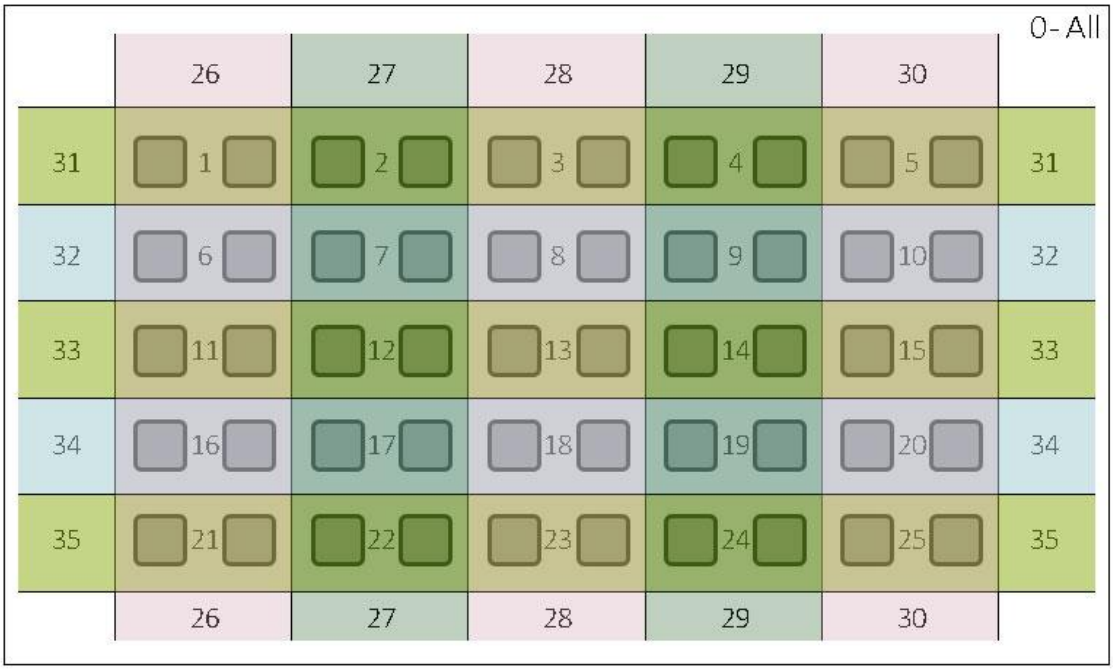


Figure 6 – Zone targeting sub-division example

As the zoning sub-division is known beforehand, each seat number is associated to a given base zone (1 to 25 in Figure 6). Each zone is attributed a given BSSID, corresponding to a specific access point of the system's network, and connects only to that AP. The network application, which takes care of message sending, maps each zone to 2 additional super groups, corresponding to lines and columns of the audience area where that zone falls into. In conclusion, each zone can be targeted via 4 different ways: direct zone identifier, line super group, column super group or global "all zone" message.

Furthering the technical details of this mapping and hardware infrastructure specificities is beyond the scope of the particular work presented herein.

3.1.3 Synchronization and synchronicity

Due to the nature of musical performances, one of the main concerns with such a system lies with synchronicity, timing and sequence of events. Both immediate and pre-timed instructions are desirable in such a system, although in the context of the prototype developed in this work, only immediate messages are implemented. Considering this approach, system latency (from message triggering to device response) becomes paramount in ensuring musical usability of said system.

With any kind of network communication there is the risk of data corruption, transmission delay and even information loss. To minimize the probability of any of these happening, the system should adopt both redundant message transmission and message numbering. Each message would be transmitted three times in immediate sequence following a data structure which allows the mobile system to check for data integrity as well as minimize the chances of data loss. Message numbering aims at allowing to filter information, by checking if any subsequent instructions has been received and processed beforehand and discarding any delayed data. In the context of this prototype, only message numbering and filtering has been implemented.

3.1.4 Tool development for content creation

Pure Data has two main versions:

- **Pd-vanilla:** this is the base and original version of Pure Data, developed and maintained by Miller Puckette. It focuses on audio and MIDI processing and features a limited array of objects.
- **Pd-extended:** this is (like the name implies) an extensions of the base pd-vanilla distribution, bundled with a great number of external objects, opening PD to graphics processing, OSC communication, binary file processing, microcontroller connection, and many more functionalities.

In the context of this particular project, the main limitation befell on the embeddable mobile assets. *LibPD* is based off of Pd-Vanilla, which meant it only featured a handful of core-bundled objects. Furthermore, certain specific functionalities were needed, not covered by any of Pure Data's features. Things like network data communication or device sensor access might be available via third-party objects, but particularities of the system made it important to be able to customize all parameters and interaction with said objects. At the same time, message targeting, message parsing and target sequencing are specificities of this particular system. Considering both situations, it became clear that it would be necessary to create a suite of objects for use by artists in order to make use of them. The development of such tools, and general evaluation of the system's performance constituted the core of my contribution to this project.

3.2 Tools

Developed code was found to be completely cross-platform compatible, with adjustments having to be made in terms of IDE and system setup so as to correctly compiling the objects on the particular system.

3.2.1 Windows

Windows development was initially setup with Visual Studio Express 2013, which worked without problems. For the sake of ease of portability, the projects were migrated to Eclipse Luna and compiled with the Mingw32 toolkit.

3.2.2 OSX

Similarly to Windows development, OSX was initially setup with XCode 5.1, but was then migrated to Eclipse Luna and compiled with the GCC toolkit.

3.2.3 Linux

Linux development was also conducted under Eclipse Luna with the GCC toolkit.

3.2.4 Android

Android porting of the objects consisted of recompiling the source code using the Android NDK, which allows to compile C/C++ code to be run in regular Android applications. Externals were compiled as shared libraries for the armeabi, armeabi-v7a and x86 CPU architectures.

Source code for the mobile versions of the objects is mostly the same as for the corresponding desktop versions, although all the “post” and “error” output messages have been removed, as they serve only as feedback mechanisms for object operation at the moment of content creation.

Source code for the Android test application is included in Appendix B – “Test applications source code”

3.2.5 IOS

IOS porting is an immediate thing, consisting of just including the mac OSX developed source code into the native application code and letting it be compiled at the same time.

3.3 Developed objects

Like previously stated, Pure Data external development isn't a widely documented process, with the most immediate source dating back to 2001 (Zmölnig, 2001). At least one book has been published in recent years covering the process (Lyon, 2012). Therefore, the main sources of material in the learning and debugging process remained the aforementioned 2001 publication, as well as the Pure Data developer forums and the Pd-dev mailing list.

In the case of the main patch (server), with the possibility of using Pure Data Extended, it is possible to use any available object without limitations. In the case of the embedded patches for the application, however, the creation of this suite is paramount, so as to make it available both in desktop versions (Windows, Mac OSX and Linux) for use at the time of creating the patches, and in mobile versions (Android and iOS), for internal operation in the application.

Developed objects can be divided into two sub-sets, depending on their function:

- **System objects:** This sub-set of objects implements functionalities inherent and specific to the system's operation (e.g. data communication, device sensor access).

- **Auxiliary objects:** This sub-set of auxiliary objects implement functionalities which could be achieved via other Pure Data processes (abstractions, objects or other) but are provided so as to standardize and simplify their usage between the desktop and mobile systems (e.g. scale remapping, target cycling)

The developed objects are to be used under two distinct contexts: main patch and embedded patch.

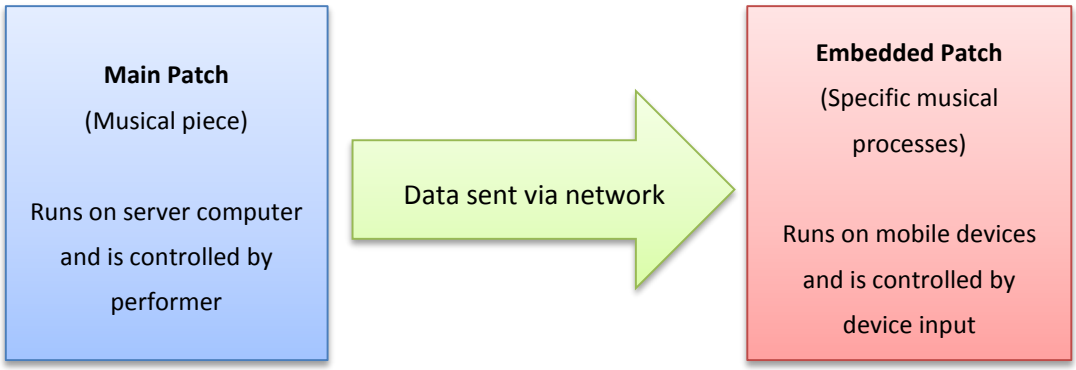


Figure 7 – Pure Data patch contexts

Data is sent and received through a series of "data slots". These "data slots" correspond to a series of user-definable variables, which are responsible for storing the data values to be sent via network to the mobile devices. A maximum of 40 data slots has been decided to be made available, with the used number to be freely decided by the composer according to his needs.

They accept a MIDI-like scale of 0-127 integer values, which are grouped and converted to byte values and sent over network in the form of a string block, and read back by the application upon reception. This string is composed of 2 different characters for each data slot value, the first corresponding to the data slot index (for correct parsing and output in the embedded patch) and the second corresponding to that slot's sent value.

Communication was decided to only be unidirectional, implementing no device-to-server communication but, instead, relying solely on device sensor input as form of interaction and manipulation of the pre-designed processes included in the embedded patches.

In this section certain system object particularities and specificities are explained. Although in most cases the source code is heavily commented and that alone is enough to explain the object's functionality, some options and approaches need further explaining or detailing. Auxiliary objects have no particular specificities and their functionality is explained in code comments [2] and in the include helper patches screenshots.

It is important to note that only 3 sensor-access objects were developed for the implementation test, making use of the accelerometer and proximity sensor, as well as the touchscreen. These were chosen since they are the most commonly available sensors on most devices.

3.3.1 Abel_dataIn

3.3.1.1 Definition

This object is responsible for receiving, parsing and outputting data sent from the main patch. It has no inlets, receiving the data directly from the application, and has outlets corresponding to the number of data slots.

Arguments:

1. Number of data slots (optional: default = 1 slot): This argument determines the number of outlets for the object, and should correspond to the same number of slots defined on the main patch for the Abel_dataOut object.

Multiple copies: No

Context: Main patch

3.3.1.2 Implementation

This object's main functionality in the implemented prototype simply consists of receiving a sequence of 2 number lists from the application (generated through the parsing of the received

[2] Please refer to Appendix A – "External object commented source code" on page 58

data message), checking if the list obeys the expected format, and outputting corresponding data. The two numbers correspond, in order, to a slot identifier index and its corresponding value.

The Abel_dataIn has a hidden functionality, which is used at asset creation time. In addition to the regular list-receiving behavior, this object also accepts the complete data block/string, with all encoded slot and value, sent directly by the Abel_dataOut object. Upon reception of the message, it attempts to verify format and parse all data pairs, outputting the corresponding values upon completion.

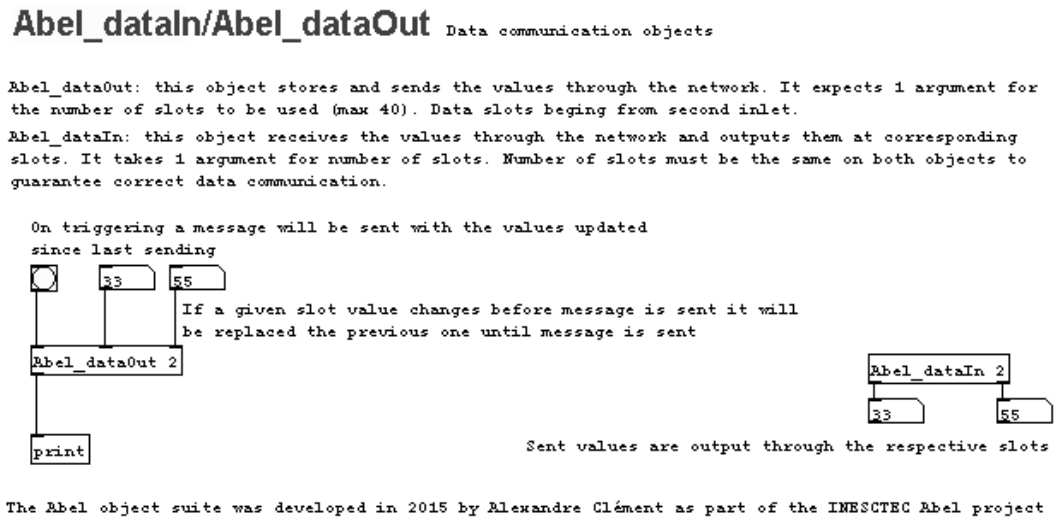


Figure 8 – Abel_dataIn helper patch

3.3.2 Abel_dataOut

3.3.2.1 Definition

This object is responsible for grouping data values and targets, and building the message to be sent to the network application for processing. It has the same number of inlets as the desired data slots, and has one outlet from which the final message is output.

Arguments:

1. Number of data slots (optional: default = 1 slot): This argument determines the number of created inlets and should match the total number of data slots to send from the main patch.

Multiple copies: No

Context: Main patch

3.3.2.2 Implementation

Object functionality is pretty well detailed in the source code's comments, although some details should be further explained.

The object is responsible for receiving the values to be sent via network. It receives them passively, meaning if a new value is received on the same slot before the message is sent the previous value is replaced. On message triggering each value is encoded into a string along with an identifier corresponding to its slot number, both converted to char variables. Values follow the same scale as MIDI (0-127), while slots are limited to a maximum of 40. Considering this, a char variable is the ideal type, since it's the simplest one available in C and allows for a range of 0-255 (unsigned char). Due to specific behavior of the *netsend* object when encoding certain ascii characters (namely the "/" character, which gets escaped and sent as "//"), and given the reduced range of values needed, the decision was made to shift the encoded values by 127, in order to avoid those escaped characters, thus using a scale of 127-254 to encode received values. This scale modifier is then removed in the native application while parsing the received data block string.

If it detects an Abel_dataIn is currently active (meaning an embedded patch is open) it will send the data block directly to it, allowing simulated network connectivity.

This object's helper patch is the same as Abel_dataIn

3.3.3 Abel_accIn

3.3.3.1 Definition

This object is responsible for receiving the client device's accelerometer values. It has no inlets, receiving the raw acceleration values in x, y and z axes directly from the application. It is an object with 7 outlets, calculating tilt and roll values from the received raw accelerator coordinates and outputting the values through its first 2 outlets. It also calculates the overall acceleration magnitude, outputting it through outlet 3. The next 3 outlets output the raw accelerator values in each of the axes (x, y, z), remapped to the target scale. The last outlet will output a bang whenever the device is shaken.

Arguments:

1. Lower limit of the re-mapping scale (optional – defaults to 0)
2. Upper limit of the re-mapping scale (optional – defaults to 127)

In the case of only one parameter being defined, it will be ignored.

Multiple copies: Yes
Context: Embedded patch

3.3.3.2 Implementation

This object implements 3 distinct calculations, in order to provide higher-level interpretations of the device's accelerometer. Instead of only outputting the raw acceleration values for each axis, the object uses them to calculate acceleration magnitude (global force applied to device), pitch (rotation on the x axis) and roll (rotation on the y axis). Yaw (rotation on the z axis) depends on geomagnetic readings, which aren't provided by the accelerator sensor. Pitch and roll are expressed as an angle, going from 0º to 90º.

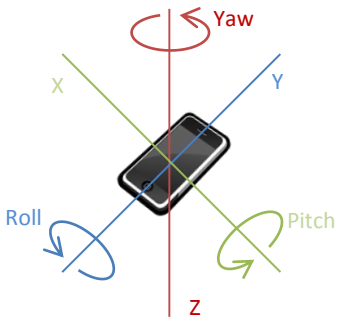


Figure 9 – Rotation types in 3D space

These values are calculated making use of the measured acceleration values on each of the 3 axes, through the following equations:

$$A_{magnitude} = \sqrt{A_x^2 + A_y^2 + A_z^2} \quad (1)$$

$$Pitch\ \alpha = \arctan\left(\frac{A_x}{\sqrt{(A_y)^2+(A_z)^2}}\right) \times \frac{360}{\pi} \quad (2)$$

$$Roll\ \beta = \arctan\left(\frac{A_y}{\sqrt{(A_x)^2+(A_z)^2}}\right) \times \frac{360}{\pi} \quad (3)$$

This object receives a "setdefault" message from the application upon application initialization, alongside with 1 float value, corresponding to the maximum value the device's accelerometer is able to measure. From this value the object is able to determine the original scale for the acceleration values (0-max) and remap the readings to the target scale.

One thing to consider for this object's operation and concerning the particular values it outputs: Pitch and Roll values assume that little force is being applied to the device, similar to holding it in the hand and simply rolling it around, with gravity being the main source of

acceleration. Shake or acceleration force related processes should make use of the acceleration magnitude and/or the raw acceleration values.

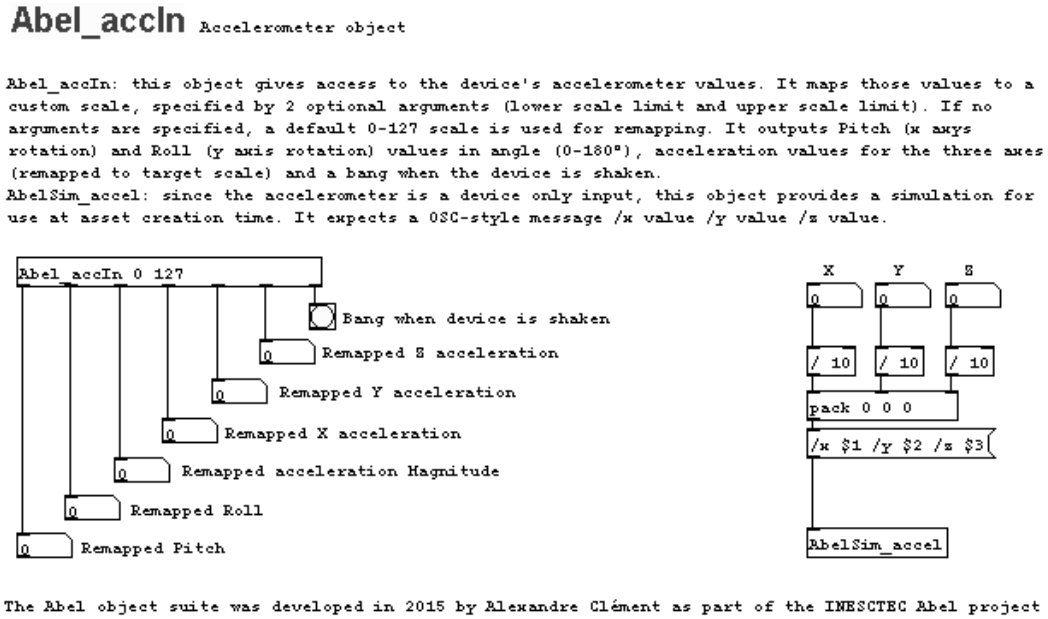


Figure 10 – Abel_accIn helper patch

3.3.4 Abel_proximityIn

3.3.4.1 Definition

This object is responsible for receiving the values of the proximity sensor from the client device. It has no inlets, receiving the raw sensor value directly from the application. It has one single outlet, outputting 1 if the device is "far" and 0 if the device is "near".

Arguments: none

Multiple copies: Yes

Context: Embedded patch

3.3.4.2 Implementation

Functionality is pretty straightforward, operating on a simple Boolean logic of "near" VS "far", and outputting a value for each corresponding state. Despite the fact that some proximity sensors measure and provide a distance value most only provide a Boolean value corresponding to those two aforementioned states. Taking this into account, the choice was made to adopt the most common situation. The object state verification (near/far) is therefore a simple check (0 or different than 0) to determine the device's proximity state.

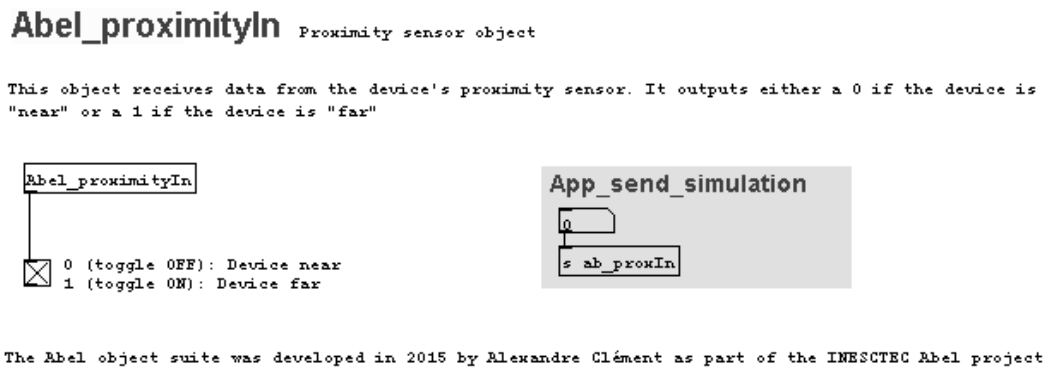


Figure 11 – Abel_proximityIn helper patch

3.3.5 Abel_touchIn

3.3.5.1 Definition

This object receives data from the device's touchscreen sensor. Its 2 last outlets are fixed and output touch/tap coordinates, in percentages of screen size (e.g. 50 / 50 would correspond to a tap at the device's screen center point). It has a variable number of outlets before those, depending on implemented gestures. It currently implements 4 different gestures, and outputs a bang through a correspondent outlet whenever either is detected.

Arguments: none

Multiple copies: Yes

Context: Embedded patch

3.3.5.2 Implementation

This object's behavior is simple, receiving an integer gesture code from the app and outputting a bang through a correspondent outlet. It also features a list receiver for x and y coordinates of the touch event.

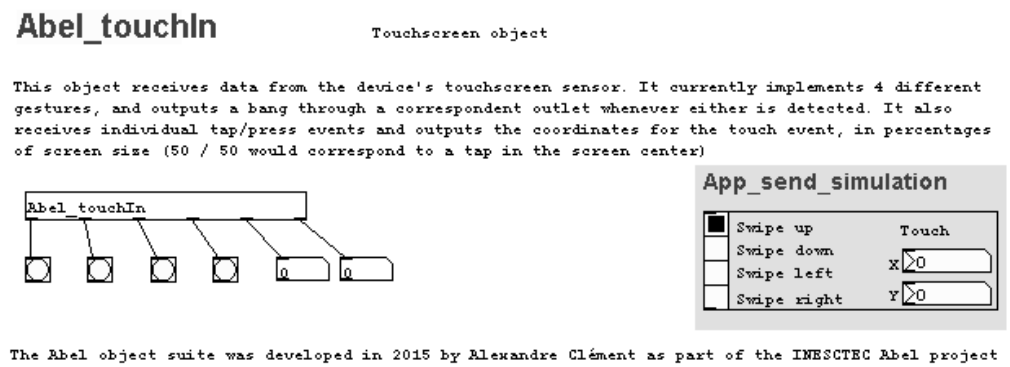


Figure 12 – Abel_touchIn helper patch

3.3.6 Abel_colorOut

3.3.6.1 Definition

This object is responsible for sending background color change instructions to the application. It has one inlet per color component (R, G, B), where each takes an integer from 0 to 255. It outputs the code directly to the app, and has no outlets.

Arguments: none

Multiple copies: Yes

Context: Embedded patch

3.3.6.2 Implementation

Upon reception of the 3 value list, the object converts them to a hexadecimal code (truncating to the 0-255 range if needed) and outputs the generated hex code to the application.

It also features a "hidden" functionality, sending a specifically calculated color code to the AbelSim_deviceOut abstraction, if it exists.

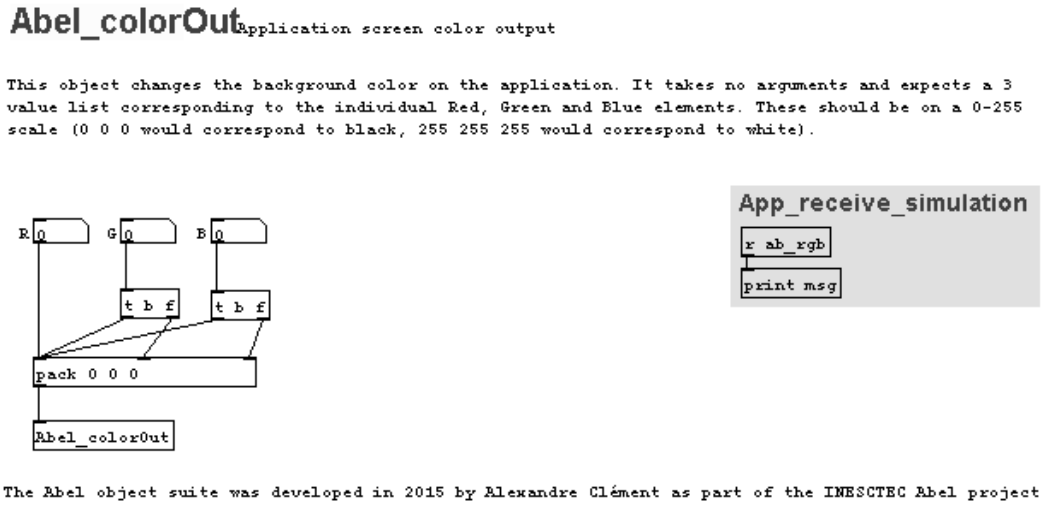


Figure 13 – Abel_colorOut helper patch

3.3.7 Abel_msgOut

3.3.7.1 Definition

This objects sends the application the text message to be displayed on-screen. It has 1 inlet, corresponding to the message to be sent, and has no outlets, sending the message directly to the application.

Arguments: none
Multiple copies: Yes
Context: Embedded patch

3.3.7.2 Implementation

This object has no particular specificities and its functionality is explained in code comments.

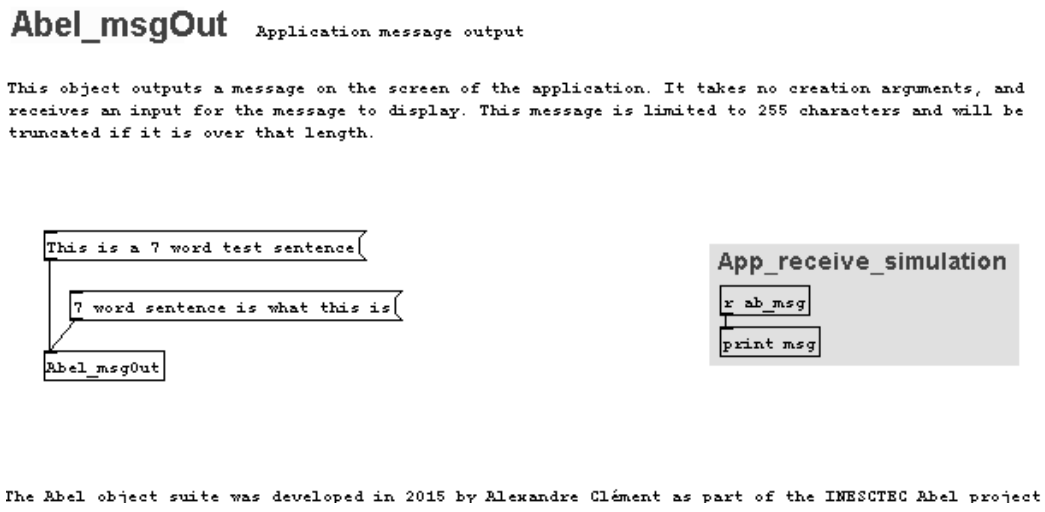


Figure 14 – Abel_msgOut helper patch

3.3.8 Abel_scale

3.3.8.1 Definition

This object is responsible for remapping values from one scale to another. It has 1 inlet and 1 outlet, corresponding respectively to the value in the original scale and value remapped to the new scale. (Similar functionality to the zmap/scale objects in max / msp).

Arguments:

1. Lower limit of the original scale (required)
2. Upper limit of the original scale (required)
3. Lower limit of the target scale (optional: default = 0)
4. Upper limit of the target scale (optional: default = 127)

In case no arguments are provided on creation, the object will do nothing and will output the original value. I accepts either 2 or 4 arguments (2 pairs).

Multiple copies: Yes

Context: Main patch, embedded patch

3.3.8.2 Implementation

This object is simple in its operation, receiving a value, calculating its correspondent value in the new scale, and outputting the result.

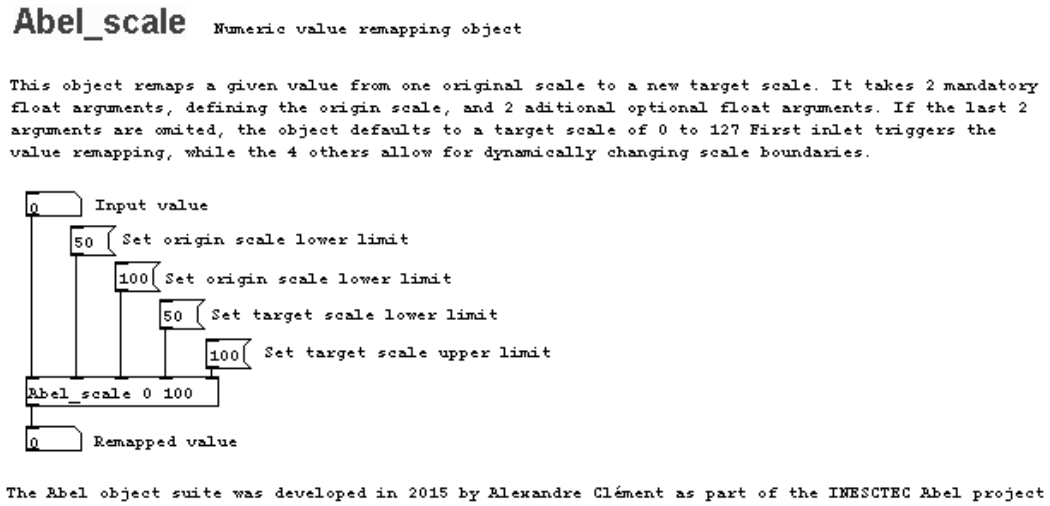


Figure 15 – Abel_scale helper patch

3.3.9 Abel_seqTarget

3.3.9.1 Definition

This object is responsible for automatically generating a progression of target zones for use by the Abel_dataOut object. It has 4 inlets and 1 outlet. In the first inlet it receives either a bang message to increase the target zone count, or a "reset" message which will reset the starting zone value to the one provided on object creation. On its other inlets it can receive float values to immediately set the number of zones, start zone and current zone, respectively. The outlet outputs the zone targeting code (should be connected to the first inlet of Abel_dataOut object).

Arguments:

1. Number of areas to cycle through (required)
2. Starting zone (optional: default = 0)

Multiple copies: Yes

Context: Main patch

3.3.9.2 Implementation

The target zone sequence progresses in a “round robin” pattern, cycling back to the starting zone when the maximum value for the target is reached. If current zone is defined (through the corresponding inlet) to a value lower than the start zone value, it gets changed to the start zone value.

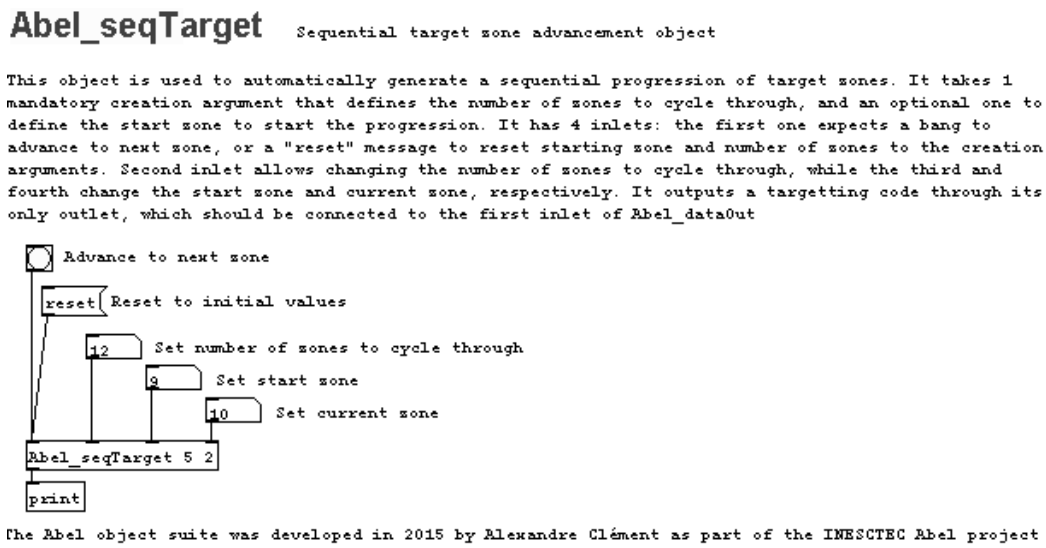


Figure 16 – Abel_seqTarget helper patch

3.3.10 Abel_movTarget

3.3.10.1 Definition

This object has similar operation to the Abel_seqTarget object, but allows the creation of a custom list of zones to go through sequentially. It has 2 inlets, corresponding to the target sequence increment bang message and a “reset” message on the first inlets, and a list of floats on the second one to redefine the target zones list. Like in Abel_seqTarget the outlet outputs the zone targeting code and should be connected to the first inlet of the Abel_dataOut object.

Arguments:

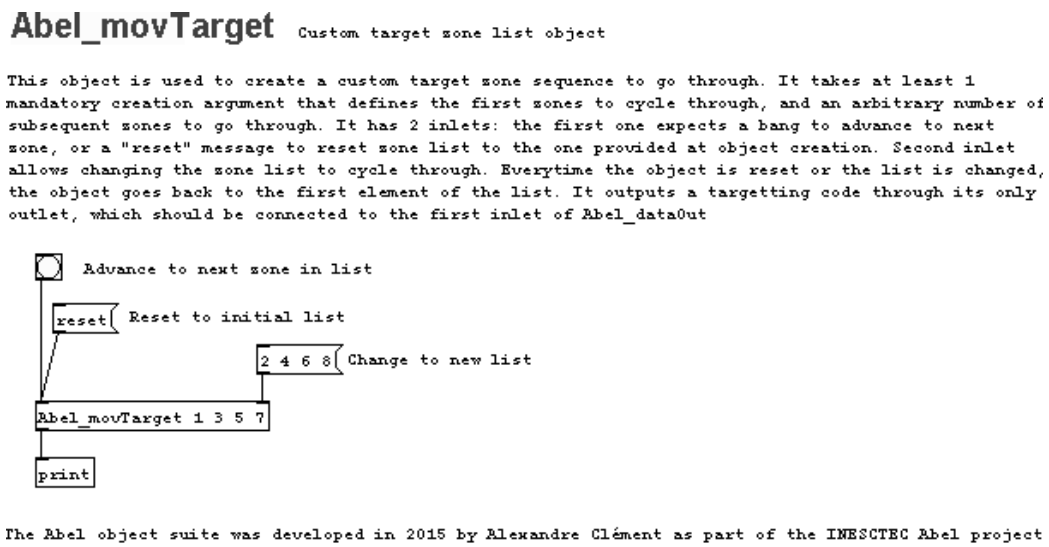
n. N floats corresponding to the list of areas to go through (requires at least 1)

Multiple copies: Yes

Context: Main patch

3.3.10.2 Implementation

Just like Abel_seqTarget, the target sequence progresses in a "round robin" pattern.



3.4 Developed abstractions

As the system is not a self-contained portable solution, it is mostly unusable at time of content creation, as network communication is unavailable without the system's network hardware infrastructure. Composers need, therefore, a way to simulate the system's behavior and the device's sensors. As such, there is the need to provide functionality simulators/emulators to account for these needs. This is achieved through a number of PD abstractions (self-contained patches usable as objects).

3.4.1 AbelSim_accel

3.4.1.1 Definition

This abstraction provides a simulation of the device's accelerometer. It receives an osc formatted message in the form of "/accel/x VAL / accel/y VAL /accel/z VAL" and parses and sends it to the *Abel_accIn* object.

It can receive this message from any source, but is aimed at receiving OSC output from a mobile app sending sensor data via osc (something like Sensors2OSC[3]).

[3] http://sensors2.org/osc/

3.4.1.2 Implementation

The abstraction receives a message and attempts to parse it following the described format. Upon parsing success (meaning the message follows the exact same format), a *spigot* object is opened, allowing for the built list to be sent to the Abel_accIn object. Upon failure, the *spigot* object is closed, preventing any data from being sent out. List output is only triggered if the z coordinate is reached and correct. If on either "route" check there is no match found for the required block, the spigot is closed.

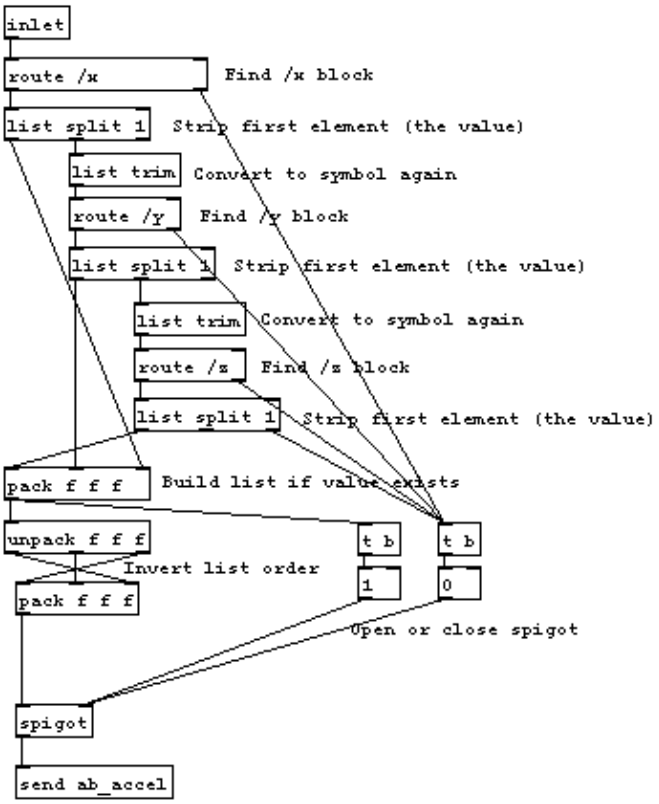


Figure 17 – Structure of the AbelSim_accel abstraction

3.4.2 AbelSim_proxim

3.4.2.1 Definition

This abstraction provides a simulation of the device's proximity sensor. It receives an OSC formatted message in the form of "/proximity VAL", parses and send it to the *Abel_proximityIn* object.

It is built with the same idea behind AbelSim_accel regarding data source.

3.4.2.2 Implementation

This abstraction's structure is almost like a simplified version of the AbelSim_accel abstraction.

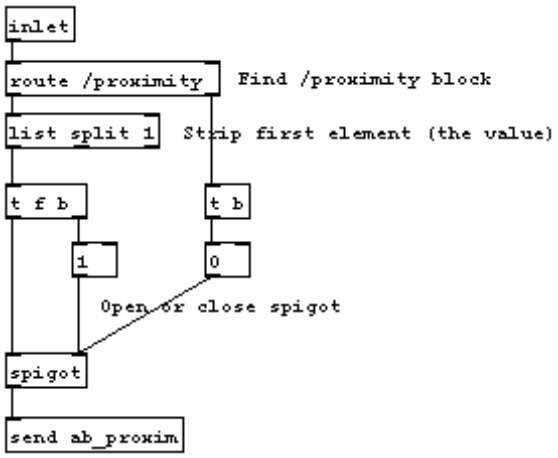


Figure 18 – Structure of the AbelSim_proxim abstraction

3.4.3 AbelSim_touchIn

3.4.3.1 Definition

This abstraction simulates a number of touchscreen interactions. Namely 4 types of gestures and individual touch event coordinates. Upon creation it features a simple graphical interface for selection[4].

3.4.3.2 Implementation

Its operation is simple, sending values to the Abel_touchIn object in the same way the mobile application would.

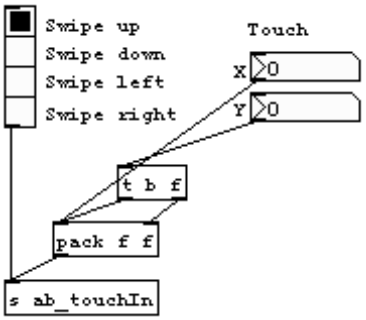


Figure 19 – Structure of the AbelSim_touchIn abstraction

[4] Please refer to Abel_touchIn helper patch (p.31) for a visual representation of this

3.4.4 AbelSim_deviceOut

3.4.4.1 Definition

This abstraction provides a preview of what the device should be showing on-screen at any given time. It receives output from the Abel_colorOut and Abel_msgOut objects and shows current status.

3.4.4.2 Implementation

This abstraction receives the regular output from the Abel_msgOut object, and the "hidden" message from the Abel_colorOut object with a color code suited for the canvas object, to set its background color.

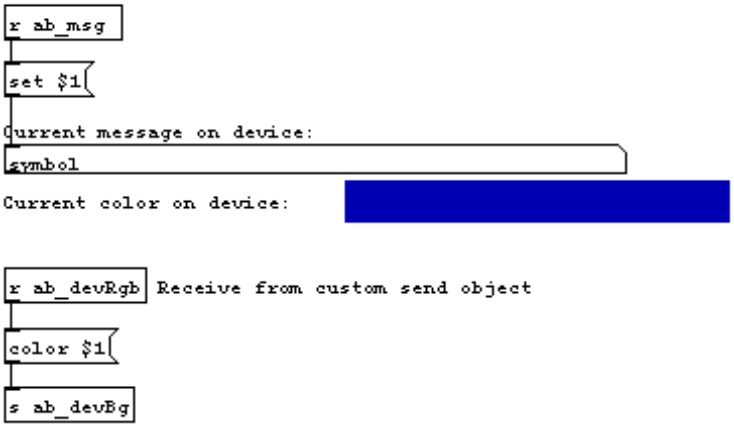


Figure 20 – Structure of the AbelSim_deviceOut abstraction

3.4.5 AbelSim_deviceSimulator

3.4.5.1 Definition

This abstraction groups all the above simulators into one single patch, adding some extra functionalities, in order to serve as a simulation for the mobile device, showing its current GUI state and providing easy access to the sensor simulators.

3.4.5.2 Implementation

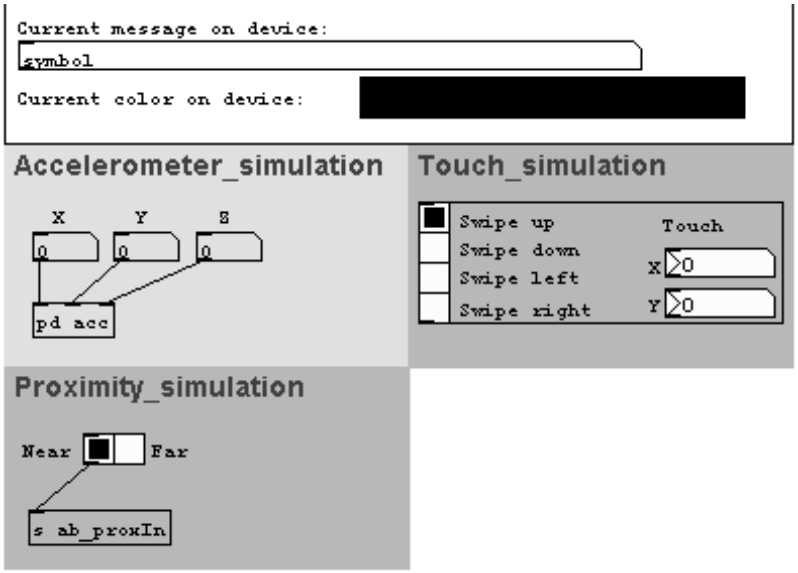


Figure 21 – Structure of the AbelSim_deviceSimulator abstraction

3.5 Test application

The ultimate goal of this implementation was to include the developed objects in the final mobile libPD embedded applications. But this final application would carry a lot of unnecessary functions and sections, such as all the network connection, message filtering, QR code reading, and more. Hence, there was the need to develop a simple application to serve as test bed for the testing of objects' functionalities. The test applications should allow to assess both the correct compilation and correct functioning of the objects.

Development of the application, embedding of libPD and overall testing followed processes described in (Brinkmann, 2012) and (Hillerson, 2014) as base for creating the final result [5].

The patch shown in Figure 22 was embedded into to the test applications in order to assess all functionalities of the objects meant to be used in embedded context. Abel_dataIn is the only object not included, whose functionality was validated through the working prototype developed in conjunction with Rafael Henriques.

[5] Time constraints surrounding the implementation of the working prototype dictated that only the Android test application was developed, leaving the iOS version in an unfinished state, with its implementation pushed back to after the submission of this dissertation

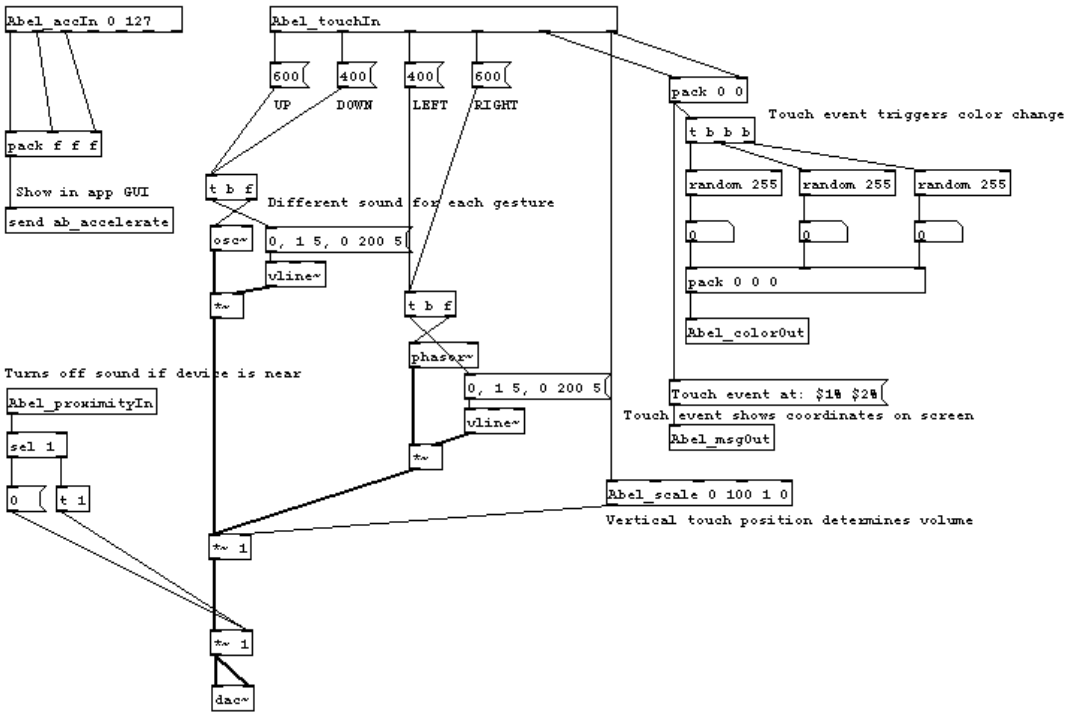


Figure 22 – Test patch embedded into application

3.6 Final considerations

While some of these functionalities could be achieved with relative ease through available Pure Data objects or through abstractions, and left to be coped with by composers/content creators, it isn't an ideal solution. Not only would that open the door to potential unforeseen problems, but it would also become counter-productive to the "make it as straightforward and easy as possible" ideal, which is one of the cornerstones of this project.

At the same time, in order to implement the detailed emulation functions, custom development was needed.

Chapter 4

Tests

4.1 Overview

Developed objects were tested through extensive debugging and simulated use case testing to ensure stability and the absence of crashes or unhandled possibilities that might result in the incorrect behavior. Use case testing would require continued use by the end-user of the toolset (in this case the composers), which will only come after the timeframe of this thesis. Nonetheless, an extensive array of crash-testing possibilities were tested to ensure a minimum of potential uses/situations could result in any object's instability or non-usability:

- Creation with wrong parameters
- Sending of unsupported messages or message formats
- Managing number of copies of the objects (prevent or allow more than one)
- Changing object parameters in real time
- Changing inlet and outlet number (where applicable)

In order to test the correct behavior of each of the objects to be used in the context of the embedded patches, and considering most objects would not be included in the working prototype being developed, a particular test applications was developed, making use of all the sensors and implemented functionalities. Its source code, as well as an image of the embedded testing patch are included in Appendix B – "Test applications source code"

The one aspect that could be assessed was performance, so as to provide a benchmark of the system's performance on a variety of devices, and allow conclusions to be reached on expected average and minimum system behavior. Audio tests were conducted in two different ways – single device independent testing and multi-device simultaneous testing – while visual tests were conducted just in multi-device simultaneous testing.

4.1.1 Audio testing

In terms of audio testing, two vital variables impact the performance of the system: device reproduction latency (the total time it takes for any given device to react to a message sent from the server), synchronicity over time (whether the measured latency is steady or fluctuates over time) and synchronicity/reliability related to message sending frequency and device processing power (the faster the messages are sent, the more it is expected for the device to be unable to cope with all the messages or to ignore/lose some). This would allow to get a better understanding of the average possibilities of the system and determine more concrete limits to be implemented into the objects and ensure ideal response from the system. For example: determining average final sounding latency over a number of devices (taking into account network latency, native app processing time, libPD/PD processing time and audio reproduction) coupled with global performance *VS* message sending frequency would allow to hardcode a limit for message sending frequency into the *Abel_dataOut* object, thus guaranteeing an optimal behavior in a greater number of target devices.

4.1.2 UI/Visual feedback testing

In terms of visual feedback, and given that only two forms of visual feedback are to be implemented in this system (background color change and text message prompting), the main issue to verify would be synchronicity between devices and overall latency from message sending to visual effect performance.

4.2 Single device testing

Single device testing aimed at getting more extensive latency data over a more reduced number of devices. It focused mainly on the low and mid-end range of smartphones and tablets, as these would most likely constitute the major part the available devices in a potential audience, while at the same time posing the most potential problems to the system's performance.

Since the objective of these tests was to get a global overview of end-latency of the system, from sending a message from the main patch to the corresponding reaction from the device's embedded patch, the easiest way to achieve this objective would be by ways of audio comparing.

The audio output from both the server computer and devices were recorded at the same time with an Edirol UA1-EX sound card on a different computer, into Cubase 5. Latency was then measured by hand using the selection tool with the "snap to zero crossings" option active. The output of the server computer was recorded into the soundcard instead of recording it directly in the same computer so the audio signal chain was exactly the same for both inputs, taking that extra latency (albeit small) out of the final latency of the system.

Important variables to get from these tests were minimum and average latency values, both in different device ranges and in global terms. Determining standard deviation was also important, so as to get a general idea of the expected latency fluctuation throughout a performance, as well as determining how the system performed at a number of message sending intervals.

Tests were run to assess system latency with different message triggering intervals (1500ms, 1000ms, 750ms, 500ms, 250ms, 200ms, 150ms, 100ms and 50ms).

4.2.1 Devices

Table 1 – Single device testing device specifications

Device	Brand/model	Android Version	CPU	Ram
1	HTM	4.2	Dual Core 1.3GHz	512 MB
2	Galaxy Tab 10.1 LTE	4.4.2	Quad-core 2.3 GHz	3 GB
3	Samsung Galaxy S3 LTE	4.4.4	Quad-core 1.4 GHz	2 GB
4	Samsung Galaxy S4 Mini LTE	4.2.2	Dual-core 1.7 GHz	1.5 GB
5	Ainol Novo 7 Venus	4.1.1	Quad-core 1.2 GHz	1 GB
6	Jiayu G3s	4.2.1	Quad-core 1.5 GHz	1 GB
7	One Plus One	5.0.1 (cyanogen 12s)	Quad-core 2.5 GHz	3 GB
8	Lazer Capacitive 10	4.0.3	Single-core 1 GHz	1 GB
9	Motorola Moto G (2nd gen)	5.0.2	Quad-core 1.2 GHz	1 GB
10	Asus Google Nexus 7 (2013)	5.1.1	Quad-core 1.5 GHz	2 GB
11	Samsung i9300 Galaxy S3	5.1.1. (Cyanogen)	Quad-core 1.4 GHz	1 GB

4.2.2 Results

All devices exhibited chaotic response at 50ms frequency, and some even at 100ms and 150ms. In such cases the device's presented response made it impossible to operate under the system's assumptions. As such, these situations were considered inadmissible for the test calculations, but admissible in the usability analysis of the system.

These were shown to fall under one or two situations, and were classified as follows:

- **Erratic:** over 50% of the messages were lost/ignored
- **Unstable:** synchronization was chaotic, and some messages were processed at the same time, making it impossible to determine precise latency
- **Chaotic:** both situations were observed

4.3 Multiple device testing

Multi-device testing aimed at assessing visual performance, overall synchronicity between devices in a "group" use case, as well as evaluating the use of certain objects (mainly the targeting objects). Since audio testing already had more in-depth testing done in the single device tests, this aspect was in a way secondary on this test, in the sense that it didn't need as much detail.

This test took place at INESC-TEC during a group meeting. Its aim was to assess global prototype behavior, validate some aspects of the mobile prototype application (area of development outside of this particular thesis' work) and get a better understanding of the visual behavior of the devices. The participants of the meeting were kind enough to facilitate their personal smartphones in order to go through the whole install, setup and operation of the prototype. In total, 8 devices (mid to high range) were made available for testing: 1 One plus One smartphone, 1 Huawei smartphone, 1 Nexus tablet and 3 Samsung smartphones and 2 unbranded smartphones. They were divided in 3 distinct targeting zones and the same testing process/patches from the single-device tests was run.

What could be verified was that, contrary to expected, message reception was considerably worse than that observed on the single-device tests. The One plus One smartphone and Nexus 7 showed (as in the single device tests) the most consistent behavior with occasional message loss but overall stability in terms of message reception and stability over time. A number of other devices exhibited erratic behavior (over 50% messages lost and severe desynchronicity), with some going as far as reacting less than once per test cycle. On devices which presented acceptable message reception some visual artefacts were observed, derived from the implementation of the message parsing and color processing. The current message parsing cycle is run on the native Android application code, sending each value block to libPD one at a time. Individual color components (Red, Green and Blue) each correspond to one of these value blocks. On devices with faster CPUs, the final color was presented correctly, with the processing happening fast enough

to consider and use the three components at once. On slower, less capable devices, nonetheless, colors were processed taking into account only one or two of these, resulting in differences in visual result. This is most likely a side-effect from the data parsing cycle, which isn't sending all values at a fast enough speed for them to be processed by the Abel_colorOut object at the same time.

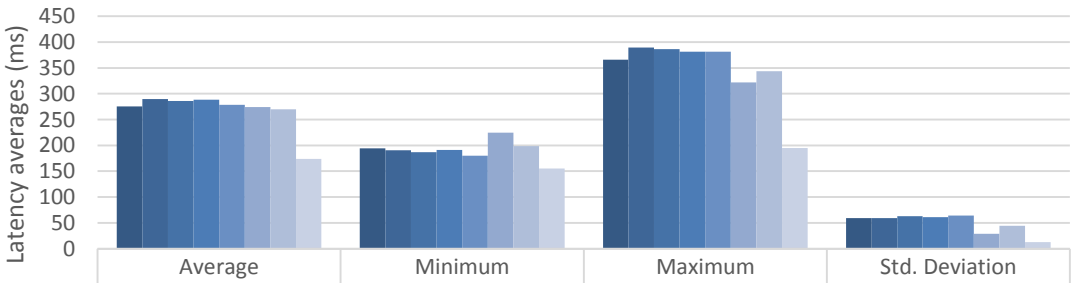
In the end, these tests showed that the prototype's functionalities worked, as all expected behavior from either of the developed PD externals happened as expected, but some of their usage approaches have to be re-designed and re-implemented.

4.4 Conclusions

As a benchmark for the system, these tests allowed to determine expected behavior boundaries for target devices. Individual tables for each device's test results are shown in Appendix C - "Testing result charts", on page 89.

4.4.1 Device response times

Table 2 – Averages per sending interval



	Average	Minimum	Maximum	Std. Deviation
1500	276	194	366	59
1000	290	190	389	59
750	286	187	386	63
500	289	191	381	61
250	279	180	381	64
200	274	225	322	29
150	270	198	343	44
100	174	155	195	13

In Table 2 we can see that average response time is pretty much similar across all message sending intervals, except for 150ms and 100ms, which apparently show lower response times.

If we take into account

Table 3, listing device behavior at each sending interval, we see that at 150ms half of the devices presented unusable response (and were thus excluded from calculations), while at 100ms

all but one device presented unusable response. Considering this fact, both sending intervals can be deemed irregular and considered unusable in terms of overall system response calculations and evaluation. In consequence, the best response time would be the ones measured with the 200ms message sending interval.

Table 3 – Device behavior over sending intervals

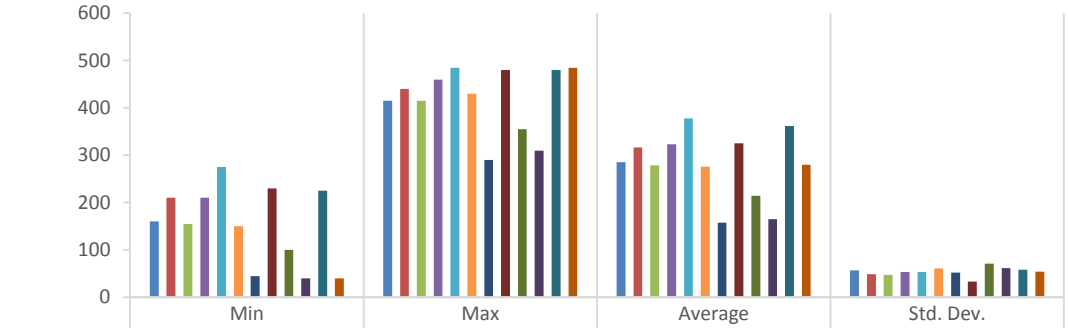
Device	HTM	Galaxy Tab	S3 LTE	S4	Novo 7	Jiayu G3s	One Plus	Lazer	Moto G	Nexus 7	S3 i9300
1500	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
1000	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
750	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
500	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
250	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
200	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
150	✓	✓	✓	✓	Err	✓	Err	Err	Err	✓	Err
100	Cha	Cha	✓	Cha	Cha	Cha	Err	Err	Err	Err	Err
50	Chaotic										

✓ - Usable/admissible device behavior

Table 4 shows overall device measurements over all sending intervals, as well as a calculated global values. We can see the global values are far from the best ones extrapolated in the paragraph above (200ms message sending interval). With the global average standard deviation of 54ms, one could say that expected response time from devices would fall between around 226ms and 334ms, which is considerably higher than ideal real-time audio latencies.

Table 4 – Single device averaged global latency test results

Tests



	Min	Max	Average	Std. Dev.
HTM	160	415	285	57
Galaxy Tab	210	440	316	49
S3 LTE	155	415	279	48
S4	210	460	324	53
Novo 7	275	485	378	53
Jiayu G3s	150	430	276	61
One Plus	45	290	158	52
Lazer	230	480	325	34
Moto G	100	355	214	71
Nexus 7	40	310	165	61
S3 i9300	225	480	361	58
Global	40	485	280	54

Table 5 – Device message reception/loss overview

Device	HTM	Galaxy Tab	S3 LTE	S4	Novo 7	Jiayu G3s	One Plus	Lazer	Moto G	Nexus 7	S3 i9300
Messages received	98	98	112	98	84	98	84	84	84	98	84
Messages lost	2	16	5	8	0	1	5	1	2	5	9
Loss (%)	2	16	4	8	0	1	6	1	2	5	11

Overall, only two devices were able to consistently present low enough latencies to have an average response time of under 200ms, while 5 devices (almost half) go above the 300ms average latency mark. Surprisingly enough, only one of them is a low- end device, with most being common medium/high range devices.

Taking these results into consideration, it is expected that the bulk of available devices for use by the public will fall on the most common 270-320ms response times. These results are far from ideal and hint at a need to review the implementation and the various stages of the system (network communication, data parsing, app/libPD communication), in order to optimize and lower the response times. Android latencies were expected to be higher than ideal, due to its audio layer implementation, and to the extra latency that could come from the libPD processing and CPU usage. Nonetheless, measured results were considerably higher than expected, and severe

optimization is mandatory. Failing to do so, or in the event the best expected results after optimization are in line with the ones measured in these tests, the real-time aspect of the system (in terms of immediate audio responses from the devices in music performances) would have to be reconsidered or rethought.

4.4.2 Stability over different message sending intervals

Considering the results in Table 2, the most stable message sending interval (lowest standard deviation) is 200ms which can be, thus, considered the most stable frequency for sending messages. It would be interesting to hard-code a sending interval limit into the Abel_dataOut object, to ensure messages will never be sent at any of the instability-inducing intervals. In practical terms, a maximum of 200ms (corresponding to 5 messages per second) would be benefic in terms of trying to maximize stability in device response.

4.4.3 Video VS audio

The multi device video tests showed a number of potential problems, namely tied to the message structure and communication. The disparity between background colors, for example, is tied to the way the data message is handled, and the impact of each individual device's processing power (and corresponding speed of operation) on that process. Nonetheless, visual behavior and audio behavior appeared consistent enough to expect optimizations made to affect audio latencies will also reflect on visual latencies. Since both processes originate from within libPD, the message parsing/processing will have the same impact on both. The extra libPD to Android communication (for screen color change) shows little impact in terms of extra latency.

Chapter 5

Conclusions

The development process of Pure Data externals has a logarithmic learning curve. That would be the first appreciation to be taken from this work. First contact and experiences before starting to grasp the organization, structure and general setup for PD development is quite daunting and, at times, confusing. But once that first phase is overcome, the process becomes a lot easier, and opens up a wide array of new possibilities for use of Pure Data. One primary concern before starting this work surrounded the adequacy of assigning certain processing tasks (value scaling, parsing, and calculations) to Pure Data objects instead of them being handled by the native application layer. What I have come to believe, from experience and observation, is that for the sake of data processing synchronicity the less intra-layer data communication, the better. In practical terms, in the context of this particular work, it would mean: if data needs to be processed, it's better to send a data block at once from the native Android app to *libPD,* and handle its processing from within the Pure Data wrapper. Having the data processing happening from within PD itself, from reception, to parsing, to addressing and finally to responding to it also guarantees a much more controllable and predictable data flow and data integrity/usage than dividing the parsing tasks and addressing between one layer and the other.

In terms of the overall system design and implementation, the main center of problems was, from the beginning, the structuring, building and communication of the data message. The initial approach, which seemed from the start as the most straightforward and easy way to implement the transmission of values, ended up bringing a lot of rebuilding and rethinking both in building and parsing the values. The adopted solution of parsing the data block in the application and sending value pairs to libPD one by one is all but ideal. Although the simple *for* cycle should take few CPU cycles to run, the massive differences in technical specs of the devices end up resulting in a desynchronization of the expected results. I.e.: color components (R, G, B), which were

supposed to be received (and processed) at the same time by the *Abel_colorOut* object were received with slight time differences between them. This resulted in the processing and generation of hexadecimal color codes different than expected, due to receiving the individual components one at a time.

In what concerns audio testing and device performance, a number of corrections and changes in direction/approach are in order if the system is to be considered as usable in audio context. Expected latencies in the vicinity of ≈280ms are unbearable in the context of traditional real-time audio/musical performance. If we take into account that the human hear is able to differentiate sounds roughly 30ms apart, we could (arguably) allow for 50-60ms in big, dense groups of (well over 10) devices as passable; but even if we consider the possibility of well performing devices masking the performance of poorly performing devices, the results measured make the possibility of having real-time response something unusable if the aim is simultaneity, as even higher-end Android devices had unexpectedly low performance in terms of audio latency. For it to be usable, as it stands now, the system requires a creative approach from the composers, making use of these latencies and jitter as musical features.

IOS performance is expected to be considerable better, taking into account the superior implementation of its audio layer when compared to the Android OS, but only proper implementation and testing of the objects and system could allow for concrete conclusions on this matter.

5.1 Future work

Given that the final product of this work was integrated into a small scale working prototype, it could be argued that its implementation is more of a stepping stone than an actual finished or usable asset. As such its objective was to, to some extent, break ground and provide results and error analysis surrounding the initial ideas in the development of the global networked system.

In terms of direct derivation from this particular work, some additional task should prove of great value, considering the conclusions that came from the whole process:

- Creating some sort of *how-to* guide for Pure Data external development, free and easily accessible, covering not only the architecture and library functions pertaining to the development, but also the initial setup of all resources on the 3 main operative systems (Windows, Linux and Mac OSX). As this would have proven to be a great resource in the initial stages of this herein work, I believe this would allow the simplification of the same process for other people, and eventually speeding up this initial stage for other academic projects or otherwise, becoming an extra contribution of great value derived from this work

- Implementing the external objects to be used in the embedded PD patches in simple iOS apps with embedded *libPD* as a way to, at least, asses the porting process and eventual recoding needs. This would serve as a step further in the global project, allowing for further comprehension of specific code porting needs, and would also become a valuable addition to the aforementioned *how-to* guide

- Implementing additional objects to handle a wider variety of sensors, given the great number available in most devices nowadays

- Developing an application which would further the device simulation without having to rely on PD patches. Possibly giving visual feedback and interactivity. One possibility would be developing an application in Processing with an interactable 3D model of a mobile device, allowing to manipulate its rotation, simulate its sensors, etc. At the same time, the application would allow communication via OSC from one of the several sensor mobile applications to use a real device as sensor data interface

In terms of the overall system, not just in the scope of this particular project concerning PD external development, one very important aspect that might allow to in some way circumvent the real-time approach shortcomings that arise from the deficient response times, would be to implement pre-timed messaging. Along with this, a zero-time synchronization system is needed, to ensure stability over time and device synchronization. One possible approach to this would be to have a pre-performance message which would carry a timecode from the server, and have the devices calculate the time difference between its own internal clock and the server time. Subsequent messages would have a "target time" field, with a timecode identifier of the kind. Since the device would now have an individual and specific time-shift value, it would in theory be able to reproduce the given instruction at the desired time.

The immediate goal, however, should be to reconsider all the shortcomings and limitations that arose from this development and testing process, and reassess the implemented solutions. The most critical part would be the redesign of the message structure and communication, which would significantly change the architecture of the objects, and even the possibilities in terms of data communication, and to find room for optimization in the code implementation.

As previously stated, one of the core problems with the prototype lied in the data block communication, with the current char buffer solution proving somewhat shortcoming. This approach was initially thought to be the most straightforward, making use of the *netsend* PD object, which works with string-like messages (PD symbols). Character encoding problems, as well as variable size buffers resulted in multiple iterations through the whole message in order to analyze and parse all values and send them to *libPD* for use in the deployed patches showed this string/char buffer approach to bring more problems than ease of use. One of the possible solutions would be to implement a custom data structure (a C struct data type) comprised of any and all

variables which should be sent via network to the mobile applications (e.g. timecode, target time for app response, data slot number, data slot identifier and values). This data structure would allow to access variable content directly without need for iteration, which would speed up data processing. It would, nonetheless, have to bypass the *netsend* object and, as such, all network communication protocols and functions would have to be created from scratch and integrated into the dedicated *Abel_dataOut* data parsing object. Implementing this new data structure would be an urgent step before further developing the system, following by iOS porting and testing of the prototype, so as to assess the same parameters on said system as done for the Android OS.

References

Bluff, A. (n.d.). Mobile Phone Orchestra. Retrieved June 26, 2015, from http://www.rollerchimp.com/project/mobile-phone-orchestra/

Bongers, B. (2000). Physical Interfaces in the Electronic Arts Interaction Theory and Interfacing Techniques for Real-time Performance. *Trends in Gestural Control of Music*, *2000*(January), 41–70.

Brinkmann, P. (2012). *Making Musical Apps*. (S. Wallace, Ed.) (First.). "O'Reilly Media, Inc."

Brinkmann, P., Mccormick, C., Kirn, P., Roth, M., & Lawler, R. (2011). Embedding Pure Data with libpd. In *Proceeding of the Fourth International Pure Data Convention* (pp. 291–301). Retrieved from http://www.uni-weimar.de/medien/wiki/images/Embedding_Pure_Data_with_libpd.pdf

Bukvic, I., & Martin, T. (2010). Introducing l2ork: Linux laptop orchestra. *Proceedings of the 2010 Conference on New Interfaces for Musical Expression (NIME 2010), Sydney, Australia*, (Nime), 170–173. Retrieved from http://www.educ.dab.uts.edu.au/nime/PROCEEDINGS/papers/Paper H1-H4/P170_Bukvic.pdf

Burns, C., & Surges, G. (2008). NRCI: Software tools for laptop ensemble. *Proceedings of the International Computer …*. Retrieved from http://classes.berklee.edu/mbierylo/ICMC08/defevent/papers/cr1357.pdf

Carvalho, M. de A. (2009). Casa da Música: fazer música com o iPhone. Retrieved June 24, 2015, from http://jpn.up.pt/2009/05/10/casa-da-musica-fazer-musica-com-o-iphone/

Cycling 74 Website. (n.d.). Max/MSP History (archived copy). Retrieved January 12, 2015, from http://web.archive.org/web/20090609205550/http://www.cycling74.com/twiki/bin/view/FAQs/MaxMSPHistory

Dannenberg, R., Cavaco, S., & Ang, E. (2007). The Carnegie Mellon Laptop Orchestra. Retrieved from http://repository.cmu.edu/compsci/513/?utm_source=repository.cmu.edu%2Fcompsci%2F513&utm_medium=PDF&utm_campaign=PDFCoverPages

Emarketer. (2014). Worldwide Smartphone Usage to Grow 25% in 2014. Retrieved October 22, 2014, from http://www.emarketer.com/Article/Worldwide-Smartphone-Usage-Grow-25-2014/1010920

Fabiani, M., Dubus, G., & Bresin, R. (2011). MoodifierLive: Interactive and collaborative expressive music performance on mobile devices. *Proceedings of the NIME*, (June), 116–119. Retrieved from http://www.nime2011.org/proceedings/papers/B23-Fabiani.pdf

References

Fiebrink, R., Wang, G., & Trueman, D. (n.d.). S.M.E.L.T. Retrieved January 21, 2015, from http://smelt.cs.princeton.edu/

h e x l e r . n e t | TouchOSC. (n.d.). Retrieved June 24, 2015, from http://hexler.net/software/touchosc

Harker, a, Atmadjaja, A., & Bagust, J. (2008). The Worldscape Laptop Orchestra: Creating live, interactive digital music for an ensemble of fifty performers. In *Proceedings of the 2008 ICMC*. Retrieved from http://classes.berklee.edu/mbierylo/ICMC08/defevent/papers/cr1354.pdf\npapers2://publication/uuid/4F9DE64A-F1F4-48E1-9AA0-3899B4F31525

Hillerson, T. (2014). Programming sound with pure data. Retrieved from http://cds.cern.ch/record/1970205

Iglesia, D. (2013). MobMuPlat. Retrieved May 21, 2015, from http://www.mobmuplat.com/

IRCAM Website. (n.d.). A brief history of MAX (archived copy). Retrieved January 12, 2015, from http://web.archive.org/web/20090603230029/http://freesoftware.ircam.fr/article.php3?id_article=5

Kincaid, J. (n.d.). RjDj Generates An Awesome, Trippy Soundtrack For Your Life. Retrieved May 24, 2015, from http://techcrunch.com/2008/10/13/rjdj-generates-an-awesome-trippy-soundtrack-for-your-life/

Lemur – Liine. (n.d.). Retrieved June 27, 2015, from https://liine.net/en/products/lemur/

Levin, G. (2001). DIALTONES (A TELESYMPHONY). Retrieved February 11, 2014, from http://www.flong.com/storage/experience/telesymphony/index.html

Lyon, E. (2012). *Designing Audio Objects for Max/MSP and Pd*. Ar. Editions, Inc. Retrieved from http://books.google.it/books?id=9yHCvrfxPwUC

McCartney, J. (2002). Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, *26*, 61–68. doi:10.1162/014892602320991383

Mccormick, C. (n.d.). PdDroidParty GitHub. Retrieved May 24, 2015, from https://github.com/chr15m/PdDroidParty

Mccormick, C. (2011). PdDroidParty - Pure Data patches on Android devices. Retrieved June 21, 2015, from http://droidparty.net/

Oh, J., Herrera, J., & Bryan, N. (2010). Evolving the mobile phone orchestra. *Proceedings of the NIME*. Retrieved from http://mopho.stanford.edu/publish/mopho-evo_nime2010.pdf

PDforAndroid GitHub. (n.d.). Retrieved May 24, 2015, from https://github.com/libpd/pd-for-android

Pixmob. (n.d.). Pixmob Wristbands. Retrieved June 24, 2015, from http://pixmob.com/project/

References

Puckette, M. (1997a). Pure Data : another integrated computer music environment. *Proceedings, Second Intercollege Computer Music Concerts*, (FEBRUARY 1970), 37–41.

Puckette, M. (1997b). Pure data: recent progress. In *Proceedings, Third Intercollege Computer Music Festival* (pp. 1–4).

Roberts, C. (n.d.). Control. Retrieved June 24, 2015, from http://charlie-roberts.com/Control/?page_id=19

Trueman, D., & Cook, P. (2006). PLOrk: the Princeton laptop orchestra, year 1. *Proceedings of the NIME*. Retrieved from http://www.scott-smallwood.com/pdf/plork_icmc2006.pdf

Viejo, C. (n.d.). mPD – Android Apps on Google Play. Retrieved June 21, 2015, from https://play.google.com/store/apps/details?id=org.mpd

Wang, G., Bryan, N., Oh, J., & Hamilton, R. (2009). Stanford laptop orchestra (slork). *International Computer Music Conference (ICMC 2009)*, (Icmc). Retrieved from https://ccrma.stanford.edu/~ge/publish/slork-icmc2009.pdf

Wang, G., & Cook, P. R. (2003). ChucK : A Concurrent , On-the-fly , Audio Programming Language 2 . The ChucK Operator 1 . Ideas in ChucK. *International Computer Music Conference*, 1–8.

Wang, G., & Cook, P. R. (2004). On-the-fly programming: using code as an expressive musical instrument. In *Proceedings of the 2004 International Conference on New interfaces for musical expression* (pp. 138–143). National University of Singapore. Retrieved from http://portal.acm.org/citation.cfm?id=1085915

Wang, G., Essl, G., & Penttinen, H. (2008). Do mobile phones dream of electric orchestras. *Proceedings of the ICMC*, *16*(10), 1252–61. Retrieved from https://ccrma.stanford.edu/groups/mopho/publish/mopho_icmc2008.pdf

Wilcox, D. (n.d.). PdParty GitHub. Retrieved May 24, 2015, from https://github.com/danomatika/PdParty

Xylobands. (n.d.). LED wristbands and wearable technology products | Xylobands. Retrieved June 24, 2015, from http://www.xylobands.com/

Zmölnig, J. (2001). How to write an external for pure-data. *Institute for Electronic Music and Acoustics*. Retrieved from http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:How+to+write+an+External+for+Pure+Data#0\nhttp://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:How+to+write+an+external+for+pure-data#0

Appendix A

External object commented source code

7.1 Abel_dataIn

```
#include "m_pd.h"
#include <string.h>

// global variable to check if another Abel_dataOut exists
int abel_datain;

static t_class *Abel_dataIn_class;

typedef struct _Abel_dataIn {
    t_object x_obj;

    t_float numSlots;
    t_outlet ** slots;

}t_Abel_dataIn;

// method for 2 number list processing (data block parsing in native application)
void Abel_dataIn_list(t_Abel_dataIn *x, t_symbol *s, int argc, t_atom *argv)
{
        int targetSlot, targetValue;

        // verify if the received list conforms to the expected "slot value" format:
        // 2 float arguments and return from function (do nothing) if it doesn't
        if(argc != 2 ||
                        argv[0].a_type != A_FLOAT ||
                        argv[1].a_type != A_FLOAT)
                return;

        // convert float arguments to int
        targetSlot = (int) atom_getfloat(&argv[0]);
        targetValue = (int) atom_getfloat(&argv[1]);

        // if slot exists (0 < targetSlot < number of slots) output value
        if(targetSlot >= 0 && targetSlot < (int)x->numSlots)
                outlet_float(x->slots[targetSlot], targetValue);
}

// method for full message processing (data block parsing inside)
void Abel_dataIn_message(t_Abel_dataIn *x, t_symbol *s)
```

```
{
    char *msg = s->s_name;
    int i, targetSlot, val;
    unsigned char tempChar;

    // iterate through the message by pairs of characters
    for (i = 0; i < strlen(msg); i+=2) {
        // this is needed because by default char assumes the leftmost byte
                // represents sign. since we're using values that go from 127 to 254 (to avoid
escaped chars)
                // the number will need the leftmost byte (anything over 127 = 01111111
                // will be considered as a negative value. 10000001 is considered to be -1
instead of 129.
                // Assigning the value char to an unsigned char variable forces the leftmost
                // byte to be considered for the calculated value instead of defining its sign
        tempChar = msg[i];
        targetSlot = (int)tempChar - 127;

        // do the same for the value
        tempChar = msg[i+1];
        val = (int)tempChar - 127;

        // if end of string is not reached and the target slot exists, output the value
        if ((msg[i] != '\0') && (targetSlot < (int)x->numSlots)) {
            outlet_float(x->slots[targetSlot], val);
        }
    }
}

void *Abel_dataIn_new(t_floatarg count_arg) {
    t_Abel_dataIn *x = (t_Abel_dataIn *)pd_new(Abel_dataIn_class);

    int i;

    // since C initializes global variables to 0, if it is equal to 1
    // it means another Abel_dataIn has already set it to 1
    if (abel_datain == 1) {
        return 0;
    }

    abel_datain = 1;

    // if no argument or value is 0 or 1, default to 1
    // else assign argument value
    x->numSlots = count_arg <= 1 ? 1 : count_arg;
    // if argument is over 40, truncate to 40
    x->numSlots = x->numSlots > 40 ? 40 : x->numSlots;

    // allocate space for needed outlets
    x->slots = getbytes(x->numSlots * sizeof(t_outlet*));

    // create outlets
    for (i=0; i< (int) x->numSlots; i++) {
        x->slots[i] = outlet_new(&x->x_obj, gensym("float"));
    }

    // bind object to "ab_dataMsg" send identifier (received from app)
    pd_bind(&x->x_obj.ob_pd, gensym("ab_dataMsg"));

    return (void *) x;
}

// object destructor
void Abel_dataIn_die(t_Abel_dataIn *x){
    int i;
    // free memory allocated to outlets
    for (i=0; i< (int) x->numSlots; i++) {
        outlet_free(x->slots[i]);
    }
    // reset global variable to allow the creation of a new object
    abel_datain = 0;
```

```
}

void Abel_dataIn_setup(void) {
    Abel_dataIn_class = class_new(gensym("Abel_dataIn"),
            (t_newmethod)Abel_dataIn_new,
            (t_method)Abel_dataIn_die,
            sizeof(t_Abel_dataIn),
            CLASS_NOINLET,
            A_DEFFLOAT,
            0);

    class_sethelpsymbol(Abel_dataIn_class, gensym("Abel_data-help"));

    // method for message parsing
    class_addsymbol(Abel_dataIn_class, Abel_dataIn_message);
    // method for list input (message parsing in application)
    class_addlist(Abel_dataIn_class, Abel_dataIn_list);
}
```

7.2 Abel_dataOut

```
#include "m_pd.h"
#include <string.h>
#include <stdlib.h>

// placeholder for boolean type (non-C9 C has no bool type)
typedef enum { false, true } bool;

// global variable to check if another Abel_dataOut exists
int abel_dataout;

static t_class *Abel_dataOut_class;

typedef struct _Abel_dataOut {
    t_object x_obj;

    // total zones (for message variables size allocation)
    t_float totalZones;
    // targetting section of message
    char *targets;

    // data variables
    t_float numSlots;
    t_float * slotValues;

    // message outlet
    t_outlet * message_out;
}t_Abel_dataOut;

// method for assignment of message targets
void Abel_dataOut_targetting(t_Abel_dataOut *x, t_symbol *s, int argc, t_atom *argv) {
    int i;

    // max size is 2 times the number of arguments (targer number + trailing space)
    // + 1 for last index '\0' string formating
    char *mess = getbytes(( 2 * argc + 1)* sizeof(char));

    // current index
    int num_t = 0;

    for (i=0; i < argc; i++) {
        if(argv[i].a_type == A_FLOAT){
            // if there is a 0 (all targets) in the list, it should take
            // precedence over all other targets
            if(atom_getfloat(&argv[i]) == 0) {
                // define message: "/target" + " " + "0" (1)
                x->targets = "/target 0";
                // free memory allocated to mess char array
                freebytes(mess,sizeof(mess));
                // break out of function
                return;
            }

            // add space
            mess[num_t++] = ' ';
            // convert integer to corresponding character
            mess[num_t++] = (int)atom_getfloat(&argv[i]) + '0';
        }
    }

    // turn mess into a string
    mess[num_t] = '\0';

    // start targets with /target
    strcpy(x->targets, "/target");
    strcat(x->targets, mess);

    // unallocate space for mess variable
    freebytes(mess,sizeof(mess));
```

```
}

// method for building final message and outputing
void Abel_dataOut_trigger(t_Abel_dataOut *x) {
    int i, num_v = 0;
    bool changed = false;

    t_atom *temp;

    char *message = getbytes(((int)x->numSlots * 2 + 1) * sizeof(char));
    char *msg;

    for (i=0; i < (int) x->numSlots; i++) {
        int newVal = (int)x->slotValues[i];

        newVal = newVal > 127 ? 127 : newVal;

        // By default values in array are -1, to signal an unchanged
        // slot and avoid re-sending it. This value gets overwritten
        // if the corresponding inlet receives any input.
        if (newVal > -1) {
        // add data pair (slot number + slot value) to end message
            message[num_v++] = 127 + i;
            message[num_v++] = (char) 127 + newVal;

            // reset value to -1 (unchanged)
            x->slotValues[i] = -1;

            // set flag saying at least one slot has a new value
            // to trigger message output
            changed = true;
        }
    }

    // if any inlet has received a new value
    if(changed == true) {
        // turn mess into a string
        message[num_v] = '\0';

        // message needs to have size = targeting section length + 6 chars (" /msg ")
        // + data block length
        msg = getbytes( ( strlen(x->targets) + strlen(message) + 6 ) * sizeof(char) );

        // build message with the 3 parts
        strcpy(msg, x->targets);
        strcat(msg, " /msg ");
        strcat(msg, message);

        // we want to output the message with a "send" command in first place,
        // hence no PD type should appear before it (list, float, symbol, etc.).
        // The message needs to be output with type "anything" and the command
        // needs to be defined as the "send" symbol. However, since the outlet_anything
        // method expects the content to output to be a t_atom, we can't use gensym
        // to turn the message string into a symbol, and have to create a new atom
        // and assign a symbol type to it by hand.
        temp = getbytes(sizeof(t_atom));
        SETSYMBOL(temp, gensym(msg));

        outlet_anything(x->message_out, gensym("send"), 1, temp);

        // if this receiver is set, it means an Abel_dataIn is currently open/created.
        // sending the message directly will trigger its data parsing method
        // and allow simulation of network communication between server
        // and embedded patches
        if (gensym("ab_dataMsg")->s_thing)
        pd_symbol(gensym("ab_dataMsg")->s_thing, gensym(message));

        // unallocate final message variable
        freebytes(msg,sizeof(msg));
    }
```

```
    // unallocate data block variable
    freebytes(message,sizeof(message));
}

void *Abel_dataOut_new(t_floatarg count_arg) {

    t_Abel_dataOut *x = (t_Abel_dataOut *)pd_new(Abel_dataOut_class);

    int i;

    // since C initializes global variables to 0, if it is equal to 1
    // it means another Abel_dataOut has already set it to 1
    if (abel_dataout == 1) {
        return 0;
    }

    abel_dataout = 1;

    // if slot number is 0 or 1, default to 1
    x->numSlots = (int)count_arg <= 1 ? 1 : (int)count_arg;
    // if slot number is over 40, truncate to 40
    x->numSlots = (int)count_arg > 40 ? 40 : (int)count_arg;

    // allocate space for needed slot values
    x->slotValues = getbytes(x->numSlots * sizeof(t_float));

    // create slot inlets and bind them to values array
    for (i=0; i< (int) x->numSlots; i++) {
        floatinlet_new(&x->x_obj, &x->slotValues[i]);
        x->slotValues[i] = -1;
    }

    x->totalZones = 36;

    // allocate max char array size
    x->targets = getbytes((2*x->totalZones) * sizeof(char));

    strcpy(x->targets, "/target 0");

    // assign message outlet
    x->message_out = outlet_new(&x->x_obj, gensym("anything"));

    return (void *) x;
}

// object destructor
void Abel_dataOut_die(t_Abel_dataOut *x){
    // free memory allocated to outlet and targets message
    outlet_free (x-> message_out );
    // reset global variable to allow the creation of a new object
    abel_dataout = 0;
}

void Abel_dataOut_setup(void) {
    Abel_dataOut_class = class_new(gensym("Abel_dataOut"),
            (t_newmethod)Abel_dataOut_new,
            (t_method)Abel_dataOut_die,
            sizeof(t_Abel_dataOut),
            CLASS_DEFAULT,
            A_DEFFLOAT,
            0);

    class_sethelpsymbol(Abel_dataOut_class, gensym("Abel_data-help"));

    // method to run on target selection
    class_addmethod(Abel_dataOut_class, (t_method) Abel_dataOut_targetting, gensym("target"),
A_GIMME, 0);
    // method to run on message triggering
    class_addbang(Abel_dataOut_class, (t_method) Abel_dataOut_trigger);

}
```

7.3 Abel_accIn

```
#include "m_pd.h"
#include <math.h>
#include <time.h>
#include <string.h>

#define PI 3.14159265
typedef long int __time_t;

// placeholder for boolean type (non-C9 C has no bool type)
typedef enum { false, true } bool;

static t_class *Abel_accIn_class;

typedef struct Abel_accIn {
    t_object x_ob;

    // acceleration variables
    t_float old_x, old_y, old_z;
    // sensor specific variables
    t_float gravity, maxRange;
    // remapping scale variables
    t_float ratio;
    t_float target_min, target_max;
    // shake time variable
    time_t last_shake;
    // outlets
    t_outlet *pitch_out, *roll_out, *accel_out, *shake_out;
    t_outlet *raw_x, *raw_y, *raw_z;

} t_Abel_accIn;

// this is called when Abel_accIn gets the dump message
void Abel_accIn_dump (t_Abel_accIn *x){
    post("Gravity: %f", x->gravity);
    post("Max sensor reading: %f", x->maxRange);
    post("Last shake time: %d", x->last_shake);
    post("Current time: %d", time(NULL));
}

// this just sets the device's max accelerometer value to allow scale remapping
void Abel_accIn_defaults (t_Abel_accIn *x, t_floatarg max){
        // set max sensor range
        x->maxRange = max;
        // recalculate ratio
        x->ratio = (x->target_max - x->target_min) / (x->maxRange);
}

/* this is called when Abel_accIn gets the list. */
void Abel_accIn_values(t_Abel_accIn *x, t_symbol *s, int argc, t_atom *argv) {

    float pitch, roll, acceleration;

    int shake, i;
    float coord_x, coord_y, coord_z;

    // check if arguments are 3 (x y z) and if all are floats
    // do nothing if arguments are of different number or type
    if (argc != 3) {
        return;
    }
    for (i = 0; i < argc; i++) {
        if (argv[i].a_type != A_FLOAT) {
            return;
        }
    }

    // set received acceleration values
    coord_x = atom_getfloat(&argv[0]);
```

```
    coord_y = atom_getfloat(&argv[1]);
    coord_z = atom_getfloat(&argv[2]);


    //  atan (x / sqrt(y^2 + z^2))
    pitch = fabs(atan(coord_x / sqrt(coord_y * coord_y + coord_z * coord_z)));
    pitch *= 360 / PI;

    //  atan (y / sqrt(x^2 + z^2))
    roll = fabs(atan(coord_y / sqrt(coord_x * coord_x + coord_z * coord_z)));
    roll *= 360 / PI;

    // sqrt (x^2 + y^2 + z^2) - earth_gravity_acceleration (aprox 9.8m/s)
    acceleration = sqrt(coord_x*coord_x + coord_y*coord_y + coord_z*coord_z);
    acceleration -= x->gravity;

    shake = 0;
    // if more than 1s has gone by since last shake
    if( (time(NULL)-x->last_shake) > 1){
        shake = (coord_x != x->old_x) || (coord_y != x->old_y) || (coord_z != x->old_z);

        if(shake) {
            x->old_x = coord_x;
            x->old_y = coord_y;
            x->old_z = coord_z;
            x->last_shake = time(NULL);
            outlet_bang(x->shake_out);
        }
    }

    outlet_float(x->pitch_out, pitch);
    outlet_float(x->roll_out, roll);
    outlet_float(x->accel_out, acceleration);

    // output the remapped acceleration values
    outlet_float(x->raw_x, coord_x*x->ratio);
    outlet_float(x->raw_y, coord_y*x->ratio);
    outlet_float(x->raw_z, coord_z*x->ratio);
}


/* this is called when a new "Abel_accIn" object is created. */
void *Abel_accIn_new(t_symbol *s, int argc, t_atom *argv) {
    t_Abel_accIn *x = (t_Abel_accIn *) pd_new(Abel_accIn_class);

    x->gravity = 9.8;
    x->maxRange = 9.8*2;
    x->old_x = x->old_y = x->old_z = 0;
    x->last_shake = time(NULL);

    // target scale is not defined or wrongly defined, set ratio to
    // default 0-127 scale
    if ((argc < 2) ||
                        (argv[0].a_type != A_FLOAT) ||
                        (argv[1].a_type != A_FLOAT))
        {
        // assign default remapping scale
                x->target_min = 0;
                x->target_max = 127;
        }
    else {
        x->target_min = atom_getfloat(&argv[0]);
                x->target_max = atom_getfloat(&argv[1]);
    }

    // target scale range  / original scale range (0 - default max)
    x->ratio = (x->target_max - x->target_min) / x->maxRange;


    // calculated value outlets
    x->pitch_out = outlet_new(&x->x_ob, gensym("float"));
```

```
    x->roll_out = outlet_new(&x->x_ob, gensym("float"));
    x->accel_out = outlet_new(&x->x_ob, gensym("float"));

    // remapped value outlets
    x->raw_x = outlet_new(&x->x_ob, gensym("float"));
    x->raw_y = outlet_new(&x->x_ob, gensym("float"));
    x->raw_z = outlet_new(&x->x_ob, gensym("float"));

    // bang on shake outlet
    x->shake_out = outlet_new(&x->x_ob, gensym("bang"));

    // bind to receiver
    pd_bind(&x->x_ob.ob_pd, gensym("ab_accel"));

    return (void *) x;
}


/* this is called once at setup time, when this code is loaded into Pd. */
void Abel_accIn_setup(void) {
    Abel_accIn_class = class_new(gensym("Abel_accIn"),
            (t_newmethod) Abel_accIn_new,
                        0,
                        sizeof(t_Abel_accIn),
                        CLASS_NOINLET,
                        A_GIMME,
                        0);

    class_sethelpsymbol(Abel_accIn_class, gensym("Abel_accIn-help"));

    // method to run when a list is received (acceleration values from app)
    class_addlist(Abel_accIn_class, Abel_accIn_values);
    // method to run on debug message "dump" reception
    class_addmethod(Abel_accIn_class, (t_method) Abel_accIn_dump, gensym("dump"), 0);
    // method to run on "setdefaults" message reception (from app)
    class_addmethod(Abel_accIn_class, (t_method) Abel_accIn_defaults, gensym("setdefault"),
A_FLOAT);
}
```

7.4 Abel_proximityIn

```
#include "m_pd.h"

static t_class *Abel_proximityIn_class;

typedef struct _Abel_proximityIn {
        t_object x_obj;

        t_outlet *value_out;
}t_Abel_proximityIn;

// method for proximity sensor value reception from app
void Abel_proximityIn_float(t_Abel_proximityIn *x, t_floatarg value) {

        // some sensors output an actual distance from the proximity sensor
        // but most only do 0=near 1=far. For simplicity we assume that
        // Boolean logic and say that anything received that isn't 0
        // will correspond to the "far" state
        value = value > 0 ? 1 : 0;

        // output the proximity value
        outlet_float(x->value_out, (int)value);
}

void *Abel_proximityIn_new(void) {

        t_Abel_proximityIn *x = (t_Abel_proximityIn *)pd_new(Abel_proximityIn_class);

        // bind object to "ab_proxIn" send identifier (received from app)
        pd_bind(&x->x_obj.ob_pd, gensym("ab_proxIn"));

        // create outlet for proximity value output
        x->value_out = outlet_new(&x->x_obj, gensym("float"));

        return (void *) x;
}

// object destructor
void Abel_proximityIn_die(t_Abel_proximityIn *x){
        // free memory allocated to outlet
        outlet_free (x-> value_out );
}

void Abel_proximityIn_setup(void) {
        Abel_proximityIn_class = class_new(gensym("Abel_proximityIn"),
                        (t_newmethod)Abel_proximityIn_new,
                        (t_method)Abel_proximityIn_die,
                        sizeof(t_Abel_proximityIn),
                        CLASS_NOINLET,
                        0,
                        0);

        class_sethelpsymbol(Abel_proximityIn_class, gensym("Abel_proximityIn-help"));

        // method to run on proximity sensor value received
        class_addfloat(Abel_proximityIn_class, Abel_proximityIn_float);
}
```

7.5 Abel_touchIn

```
#include "m_pd.h"

static t_class *Abel_touchIn_class;

typedef struct _Abel_touchIn {
        t_object x_obj;

        // number of implemented gestures
        t_float implemented_gestures;

        // gesture bang outlets
        t_outlet ** gesture_slots;
        // touch/tap event coordinates outlets
        t_outlet * x_out, *y_out;

} t_Abel_touchIn;

// method for individual touch/tap event coordinate reception from app
void Abel_touchIn_list(t_Abel_touchIn *x, t_symbol *s, int argc, t_atom *argv) {
        t_float coord_x, coord_y;

        // if the received list isn't of expected type (2 number list)
        if ((argc != 2) || argv[0].a_type != A_FLOAT || argv[1].a_type != A_FLOAT)
                return;

        // get coordinate values
        coord_x = atom_getfloat(&argv[0]);
        coord_y = atom_getfloat(&argv[1]);

        // output position values
        outlet_float(x->x_out, coord_x);
        outlet_float(x->y_out, coord_y);
}

// method for gesture identifier code reception from app
void Abel_touchIn_float(t_Abel_touchIn *x, t_floatarg gesture) {
        switch ((int) gesture) {
        case 0:
                // swipe up
                outlet_bang(x->gesture_slots[0]);
                break;
        case 1:
                // swipe down
                outlet_bang(x->gesture_slots[1]);
                break;
        case 2:
                // swipe left
                outlet_bang(x->gesture_slots[2]);
                break;
        case 3:
                // swipe right
                outlet_bang(x->gesture_slots[3]);
                break;
        default:
                break;
        }
}

void *Abel_touchIn_new(void) {

        t_Abel_touchIn *x = (t_Abel_touchIn *) pd_new(Abel_touchIn_class);

        int i;

        // bind object to "ab_touchIn" send identifier (received from app)
        pd_bind(&x->x_obj.ob_pd, gensym("ab_touchIn"));

        // number of implemented gestures
```

```
        x->implemented_gestures = 4;

        // allocate space for gesture bang outputs
        x->gesture_slots = getbytes(x->implemented_gestures * sizeof(t_outlet*));

        // create outlets for implemented gestures
        for (i = 0; i < x->implemented_gestures; i++) {
                x->gesture_slots[i] = outlet_new(&x->x_obj, gensym("bang"));
        }

        // create outlets for event coordinates
        x->x_out = outlet_new(&x->x_obj, gensym("float"));
        x->y_out = outlet_new(&x->x_obj, gensym("float"));

        return (void *) x;
}

// object destructor
void Abel_touchIn_die(t_Abel_touchIn *x) {
        int i;
        // free memory allocated to outlets
        for (i = 0; i < x->implemented_gestures; i++) {
                outlet_free(x->gesture_slots[i]);
        }
        outlet_free(x->x_out);
        outlet_free(x->y_out);
}

void Abel_touchIn_setup(void) {
        Abel_touchIn_class = class_new(gensym("Abel_touchIn"),
                        (t_newmethod) Abel_touchIn_new, (t_method) Abel_touchIn_die,
                        sizeof(t_Abel_touchIn),
                        CLASS_NOINLET, 0, 0);

        class_sethelpsymbol(Abel_touchIn_class, gensym("Abel_touchIn-help"));

        // method to run on gesture value received
        class_addfloat(Abel_touchIn_class, Abel_touchIn_float);
        // method to run on touch/tap coordinates received
        class_addlist(Abel_touchIn_class, Abel_touchIn_list);
}
```

7.6 Abel_colorOut

```
#include "m_pd.h"

static t_class *Abel_colorOut_class;

typedef struct _Abel_colorOut {
        t_object x_obj;

        t_symbol *identifier;
}t_Abel_colorOut;

// this function converts the r, g, b components to an hexadecimal code
void setHex(char * buffer, int r, int g, int b) {
        // hex characters. index corresponds to decimal value
        char hex[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e',
'f'};

        // truncate values if under or over limits (0-255)
        r = r < 0 ? 0 : r;
        g = g < 0 ? 0 : g;
        b = b < 0 ? 0 : b;
        r = r > 255 ? 255 : r;
        g = g > 255 ? 255 : g;
        b = b > 255 ? 255 : b;

        // change "red" second digit to corresponding hex character
        buffer[1] = hex[r % 16];
        // if number is greater than 16, first "red" digit also changes
        if(r>15)
                buffer[0] = hex[(r / 16) % 16];

        // "green"
        buffer[3] = hex[g % 16];
        if(g > 15)
                buffer[2] = hex[(g / 16) % 16];

        // "blue"
        buffer[5] = hex[b % 16];
        if(b>15)
                buffer[4] = hex[(b / 16) % 16];

}

void Abel_colorOut_rgb(t_Abel_colorOut *x, t_symbol *s, int argc, t_atom *argv) {
        // initialize char array with 0 char in all indexes
        // and /0 on last one to make it a string
        char rgb[7] = {'0', '0', '0', '0', '0', '0', '\0'};

        // verify if received data is the expected (3 floats)
        if(argc != 3 ||
                (argv[0].a_type != A_FLOAT) ||
                (argv[1].a_type != A_FLOAT) ||
                (argv[2].a_type != A_FLOAT)){
                post("Non-rgb numeric values.");
                return;
        }

        // convert received values and store into rgb string
        setHex(rgb,
                        (int) atom_getfloat(&argv[0]),
                        (int) atom_getfloat(&argv[1]),
                        (int) atom_getfloat(&argv[2]));

        // If any object has been bound to this s_thing
        // it means there is a receiver for this message type.
        // Without this verification, PD would crash attempting to send the message
        if (x->identifier->s_thing)
                pd_symbol(x->identifier->s_thing, gensym(rgb));
```

```
        // if this receiver exists, then this means the AbelSim_deviceOut
        // abstractions is open. So the object should send a special
        // color code to change the canvas object's background color
        if (gensym("ab_devRgb")->s_thing) {
                float colcode = (((((int) atom_getfloat(&argv[0]) * 256) + (int)
atom_getfloat(&argv[1])) * 256) + (int) atom_getfloat(&argv[2]))*(-1) -1;
                pd_float(gensym("ab_devRgb")->s_thing, colcode);
        }
}

void *Abel_colorOut_new(t_symbol *s, int argc, t_atom *argv) {

        t_Abel_colorOut *x = (t_Abel_colorOut *)pd_new(Abel_colorOut_class);

        // application "send" message identifier code
        x->identifier = gensym("ab_rgb");

        return (void *) x;
}

void Abel_colorOut_setup(void) {
        Abel_colorOut_class = class_new(gensym("Abel_colorOut"),
                        (t_newmethod)Abel_colorOut_new,
                        0,
                        sizeof(t_Abel_colorOut),
                        CLASS_DEFAULT,
                        0,
                        0);

        class_sethelpsymbol(Abel_colorOut_class, gensym("Abel_colorOut-help"));

        // method for any input (gets validated inside)
        class_addanything(Abel_colorOut_class, (t_method) Abel_colorOut_rgb);
}
```

7.7 Abel_msgOut

```
#include "m_pd.h"
#include <stdlib.h>
#include <string.h>

static t_class *Abel_msgOut_class;

typedef struct _Abel_msgOut {
        t_object x_obj;

        t_symbol *identifier;
}t_Abel_msgOut;

void Abel_msgOut_msg(t_Abel_msgOut *x, t_symbol *s, int argc, t_atom *argv) {

        int i;
        int start = 0;

        t_symbol *temp;
        // max length of message is 255 chars
        char totalMessage[256] = "";

        /*
        * When a message starts with a float, PD interprets it as a float if it has
        * only one number, or as a list if it has more elements.
        * If the received message is of any of these PD types, then all elements
        * are correctly inserted in the arguments array and the type word is correctly ignored.
        * Otherwise it interprets it as an "custom type" and ignores first element
        * ex: 2 is a number -> is considered a list with arguments "2", "is", "a", "number"
        * two is a number -> is considered an "instruction" of custom type "two",
        * with arguments "is", "a", "number"
        */
        if(strcmp(s->s_name, "list") != 0 &&
                        strcmp(s->s_name, "float") != 0 &&
                        strcmp(s->s_name, "symbol") != 0 &&
                        strcmp(s->s_name, "array") != 0 &&
                        strcmp(s->s_name, "pointer") != 0) {
                // get first element (type) into the final message string
                strcat(totalMessage, s->s_name);
                // add a trailing space
                strcat(totalMessage, " ");
        }

        for(i=start; i < argc; i++) {
                // convert argument atom to symbol (needed to use floats as text)
                temp = atom_gensym(&argv[i]);

                // add current word to total message with trailing space
                strcat(totalMessage, temp->s_name);
                strcat(totalMessage, " ");
        }

        /* If any object has been bound to this s_thing
         * it means there is a receiver for this message type.
         * Without this verification, PD would crash attempting to send the message
         */
        if (x->identifier->s_thing)
                pd_symbol(x->identifier->s_thing, gensym(totalMessage));

        freebytes(totalMessage, sizeof(totalMessage));
}

void *Abel_msgOut_new(t_symbol *s, int argc, t_atom *argv) {

        t_Abel_msgOut *x = (t_Abel_msgOut *)pd_new(Abel_msgOut_class);

        // "send" message identifier code
        x->identifier = gensym("ab_msg");
```

```
        return (void *) x;
}

void Abel_msgOut_setup(void) {
        Abel_msgOut_class = class_new(gensym("Abel_msgOut"),
                        (t_newmethod)Abel_msgOut_new,
                        0,
                        sizeof(t_Abel_msgOut),
                        CLASS_DEFAULT,
                        0,
                        0);

        class_sethelpsymbol(Abel_msgOut_class, gensym("Abel_msgOut-help"));

        class_addanything(Abel_msgOut_class, (t_method) Abel_msgOut_msg);
}
```

7.8 Abel_scale

```
#include "m_pd.h"

// placeholder for boolean type (non-C9 C has no bool type)
typedef enum { false, true } bool;

static t_class *Abel_scale_class;

typedef struct _Abel_scale {
        t_object x_obj;

        bool orig_defined;
        // scale values
        t_float orig_min, orig_max, target_min, target_max;

        t_outlet *value_out;

}t_Abel_scale;

// method for value remapping
void Abel_scale_float(t_Abel_scale *x, t_floatarg value) {
        t_float new_value;
        float ratio;

        // if origin scale was not set correctly ratio should be 1
        // original value will be the same as target value
        ratio = 1;

        // if origin scale was correctly defined, calculate ratio
        // between origin and target scales
        if(x->orig_defined == true) {
                ratio = (x->target_max - x->target_min) / (x->orig_max - x->orig_min);
        }

        // recalculate value on target scale
        new_value = (value - x->orig_min) * ratio;
        new_value += x->target_min;

        // output value
        outlet_float(x->value_out, new_value);
}

void *Abel_scale_new(t_symbol *s, int argc, t_atom *argv) {

        t_Abel_scale *x = (t_Abel_scale *)pd_new(Abel_scale_class);

        // if origin scale is not correctly set
        // 2 float arguments
        if ((argc < 2) ||
                        (argv[0].a_type != A_FLOAT) ||
                        (argv[1].a_type != A_FLOAT))
        {
                post("Invalid scale parameters provided, object will do nothing.");
                x->orig_defined = false;
        }
        else {
                // origin scale is correctly set
                x->orig_defined = true;

                // define origin scale
                x->orig_min = atom_getfloat(&argv[0]);
                x->orig_max = atom_getfloat(&argv[1]);


                // if target scale is set and arguments are floats
                if((argc == 4) &&
                                (argv[2].a_type == A_FLOAT) &&
                                (argv[3].a_type == A_FLOAT)) {
```

```
                        // set target scale
                        x->target_min = atom_getfloat(&argv[2]);
                        x->target_max = atom_getfloat(&argv[3]);
                }
                else {
                        // target scale not set, default target scale to 0-127
                        post("Target scale not set or incorrectly set. Defaulting to 0-127.");
                        x->target_min = 0;
                        x->target_max = 127;
                }
        }

        // create "cold" inlets to dynamically change scales
        floatinlet_new (&x->x_obj , &x-> orig_min );
        floatinlet_new (&x->x_obj , &x-> orig_max );
        floatinlet_new (&x->x_obj , &x-> target_min );
        floatinlet_new (&x->x_obj , &x-> target_max );

        // create outlet for converted value
        x->value_out = outlet_new(&x->x_obj, gensym("float"));

        return (void *) x;
}

// object destructor
void Abel_scale_die(t_Abel_scale *x){
        // free memory allocated to outlet
        outlet_free (x-> value_out );
}

void Abel_scale_setup(void) {
        Abel_scale_class = class_new(gensym("Abel_scale"),
                        (t_newmethod)Abel_scale_new,
                        (t_method)Abel_scale_die,
                        sizeof(t_Abel_scale),
                        CLASS_DEFAULT,
                        A_GIMME,
                        0);

        class_sethelpsymbol(Abel_scale_class, gensym("Abel_scale-help"));

        // method to run on float input
        class_addfloat(Abel_scale_class, (t_method)Abel_scale_float);
}
```

7.9 Abel_seqTarget

```
#include "m_pd.h"

static t_class *Abel_seqTarget_class;

typedef struct _Abel_seqTarget {
        t_object x_obj;

        // defaults for reset
        t_float defaultStart;
        t_float defaultTargets;

        // current settings
        t_float numTargets;
        t_float startZone;

        t_float currentZone;

        // output
        t_outlet * targetOut;

}t_Abel_seqTarget;

void Abel_seqTarget_dump(t_Abel_seqTarget *x)
{
        post("Default number of zones to cycle: %d", (int)x->defaultTargets);
        post("Current number of zones to cycle: %d", (int)x->numTargets);
        post("Default starting zone: %d", (int)x->defaultStart);
        post("Current starting zone: %d", (int)x->startZone);
        post("Current zone: %d", (int)x->currentZone);
}

void Abel_seqTarget_reset(t_Abel_seqTarget *x)
{
        // reset current values to defaults
        x->startZone = x->defaultStart;
        x->numTargets = x->defaultTargets;
}

void Abel_seqTarget_advance(t_Abel_seqTarget *x)
{
        t_atom *tZone = getbytes(sizeof(t_atom*));

        // check if current zone is higher than max zone in cycle or lower than start zone
        // and change value accordingly (cycle to start or set it to lowest possible)
        x->currentZone = x->currentZone > (x->startZone + x->numTargets - 1) ? (int)x->startZone
: (int)x->currentZone;
        x->currentZone = x->currentZone < x->startZone ? (int)x->startZone : (int)x-
>currentZone;

        // output target message
        SETFLOAT(tZone, x->currentZone);
        outlet_anything(x->targetOut, gensym("target"), 1, tZone);

        // increment current zone
        x->currentZone++;
}

void *Abel_seqTarget_new(t_symbol *s, int argc, t_atom *argv) {
        t_Abel_seqTarget *x = (t_Abel_seqTarget *)pd_new(Abel_seqTarget_class);

        // if first parameter is invalid cancel object creation
        if ((argc < 1) ||
                        (argv[0].a_type != A_FLOAT))
        {
                error("Invalid parameters provided.");
                return 0;
        }
```

```
        // if it is, assign its value to the target variables
        x->numTargets = x->defaultTargets = (int)atom_getfloat(&argv[0]);

        // if second argument exists and is a float set start zone to its value
        if(argc >=2 && (argv[1].a_type == A_FLOAT)){
                // on object creation current zone and start zone are equal to the default start
zone
                x->defaultStart = x->startZone = x->currentZone = (int)atom_getfloat(&argv[1]);
        }
        // if it doesn't, default to start at zone 1
        else {
                x->defaultStart = x->startZone = x->currentZone = 1;
        }

        // create passive inlets for current value changing
        floatinlet_new(&x->x_obj, &x->numTargets);
        floatinlet_new(&x->x_obj, &x->startZone);
        floatinlet_new(&x->x_obj, &x->currentZone);

        // create outlet
        x->targetOut = outlet_new(&x->x_obj, gensym("anything"));

        return (void *) x;
}

void Abel_seqTarget_die(t_Abel_seqTarget *x){
        // unallocate outlet
        outlet_free(x->targetOut);
}

void Abel_seqTarget_setup(void) {
        Abel_seqTarget_class = class_new(gensym("Abel_seqTarget"),
                        (t_newmethod)Abel_seqTarget_new,
                        (t_method)Abel_seqTarget_die,
                        sizeof(t_Abel_seqTarget),
                        CLASS_DEFAULT,
                        A_GIMME,
                        0);

        class_sethelpsymbol(Abel_seqTarget_class, gensym("Abel_seqTarget-help"));

        // method for counting forward in targeting sequence
        class_addbang(Abel_seqTarget_class, Abel_seqTarget_advance);
        // method for resetting to initial values on "reset" message reception
        class_addmethod(Abel_seqTarget_class, (t_method) Abel_seqTarget_reset, gensym("reset"),
0);

        // debug method to check current values
        class_addmethod(Abel_seqTarget_class, (t_method) Abel_seqTarget_dump, gensym("dump"),
0);
}
```

7.10 Abel_movTarget

```
#include "m_pd.h"

static t_class *Abel_movTarget_class;

typedef struct _Abel_movTarget {
        t_object x_obj;

        // defaults for reset
        t_float *defaultTargets;
        t_float defaultNum;

        t_float numTargets;
        t_float currentIndex;

        // current settings
        t_float *targetList;

        // output
        t_outlet * targetOut;

}t_Abel_movTarget;

void Abel_movTarget_dump(t_Abel_movTarget *x) {
        int i;

        post("#%d defaults:", (int)x->defaultNum);
        for(i=0; i<x->defaultNum; i++){
                post("#%d = %d", i, (int)x->defaultTargets[i]);
        }

        post("#%d currents:", (int)x->numTargets);
        for(i=0; i<x->numTargets; i++){
                post("#%d = %d", i, (int)x->targetList[i]);
        }
}

void Abel_movTarget_advance(t_Abel_movTarget *x) {
        t_atom *tZone = getbytes(sizeof(t_atom*));

        // check if current zone is higher than max zone in cycle (total number of targets)
        // and change value accordingly (reset to 0 if true)
        x->currentIndex = x->currentIndex > x->numTargets - 1 ? 0 : (int)x->currentIndex;

        // output target message
        SETFLOAT(tZone, x->targetList[(int)x->currentIndex]);
        outlet_anything(x->targetOut, gensym("target"), 1, tZone);

        // increment current zone
        x->currentIndex++;
}

void Abel_movTarget_reset(t_Abel_movTarget *x) {
        int i;

        // reset old targetlist and allocate space for new one
        freebytes(x->targetList, x->numTargets * sizeof(t_float));

        // reallocate space for new list
        x->targetList = getbytes((int)x->defaultNum * sizeof(t_float));

        // set new number of targets
        x->numTargets = x->defaultNum;
        // set current target list to the one provided at object creation
        for(i=0; i<(int)x->numTargets; i++){
                x->targetList [i] = x->defaultTargets[i];
        }

        // reset curret zone to first element in list
```

```
        x->currentIndex = 0;
}

void Abel_movTarget_newTargets(t_Abel_movTarget *x, t_symbol *s, int argc, t_atom *argv) {
        int i;

        // cancel new target list creation
        // if there are no parameters
        if (argc < 1)
        {
                error("Invalid parameters provided.");
                return;
        }
        // or any of them are non-floats
        for(i=0; i<argc; i++){
                if(argv[i].a_type != A_FLOAT) {
                        error("Invalid parameters provided.");
                        return;
                }
        }

        // reset old targetlist and allocate space for new one
        freebytes(x->targetList, x->numTargets * sizeof(t_float));
        x->targetList = getbytes ((int) argc * sizeof(t_float));

        // set new number of targets
        x->numTargets = argc;
        // assign new list elements to current target list
        for(i=0; i<argc; i++){
                x->targetList [i] = (int) atom_getfloat(&argv[i]);
        }

        // reset curret zone to first element in list
        x->currentIndex = 0;
}


void *Abel_movTarget_new(t_symbol *s, int argc, t_atom *argv) {
        int i;

        t_Abel_movTarget *x = (t_Abel_movTarget *)pd_new(Abel_movTarget_class);

        // cancel object creation
        // if there isn't at least one zone defined
        if (argc < 1)
        {
                error("Invalid parameters provided.");
                return 0;
        }
        // or parameters are non-floats
        for(i=0; i<argc; i++){
                if(argv[i].a_type != A_FLOAT) {
                        error("Invalid parameters provided.");
                        return 0;
                }
        }

        // allocate bytes for default and current zone lists
        x->defaultTargets = getbytes(argc*sizeof(t_float));
        x->targetList = getbytes(argc*sizeof(t_float));

        // set list elements number
        x->defaultNum = x->numTargets = argc;
        // set zones to cycle through (at creation default and current are the same)
        for(i=0; i<argc; i++){
                x->defaultTargets [i] = x->targetList [i] = (int) atom_getfloat(&argv[i]);
        }

        // set curret zone to first element in list
        x->currentIndex = 0;
```

```
        // this creates a second active inlet associated to the class_addmethod method
        // defined to react to the "zonelist" code created in the setup function
        inlet_new(&x->x_obj, &x->x_obj.ob_pd, gensym("list"), gensym("zonelist"));

        // create outlet
        x->targetOut = outlet_new(&x->x_obj, gensym("anything"));

        return (void *) x;
}

void Abel_movTarget_die(t_Abel_movTarget *x){
        // unallocate outlet
        outlet_free(x->targetOut);
        // unallocate arrays for lists
        freebytes(x->defaultTargets, sizeof(x->defaultTargets));
        freebytes(x->targetList, sizeof(x->targetList));
}

void Abel_movTarget_setup(void) {
        Abel_movTarget_class = class_new(gensym("Abel_movTarget"),
                        (t_newmethod)Abel_movTarget_new,
                        (t_method)Abel_movTarget_die,
                        sizeof(t_Abel_movTarget),
                        CLASS_DEFAULT,
                        A_GIMME,
                        0);

        class_sethelpsymbol(Abel_movTarget_class, gensym("Abel_movTarget-help"));

        // method for counting forward in targeting sequence
        class_addbang(Abel_movTarget_class, Abel_movTarget_advance);
        // method for defining new target list
        class_addmethod(Abel_movTarget_class, (t_method) Abel_movTarget_newTargets,
gensym("zonelist"), A_GIMME, 0);
        // method for resetting to initial values on "reset" message reception
        class_addmethod(Abel_movTarget_class, (t_method) Abel_movTarget_reset, gensym("reset"),
0);


        // debug method to check current values
        class_addmethod(Abel_movTarget_class, (t_method) Abel_movTarget_dump, gensym("dump"),
0);
}
```

Appendix B

Test applications source code

```
package com.abel.abeltester;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.util.List;

import org.puredata.android.io.AudioParameters;
import org.puredata.android.io.PdAudio;
import org.puredata.core.PdBase;
import org.puredata.core.PdListener;
import org.puredata.core.utils.IoUtils;
import org.puredata.core.utils.PdDispatcher;

import android.content.Context;
import android.content.pm.PackageManager;
import android.content.res.Resources;
import android.graphics.Color;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.support.v7.app.ActionBarActivity;
import android.util.DisplayMetrics;
import android.util.Log;
import android.view.MotionEvent;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;

public class MainActivity extends ActionBarActivity implements
                SensorEventListener {

        // app tag
        private static final String TAG = "LibPDTest";

        // GUI variables
        private TextView raw_x, raw_y, raw_z, acc_max;
        private TextView lbl_pitch, lbl_roll, lbl_accel;
        private DisplayMetrics screen = new DisplayMetrics();

        // for debug purposes
        private SensorManager sMgr;

        // sensor variables
        private SensorManager senSensorManager;
        private Sensor senAccelerometer;
        private Sensor senProximity;
        private boolean hasAccelerometer, hasProximity;

        // accelerometer reading variables
        private long lastUpdate = 0;

        // touchscreen reading variables
        private float initial_x, initial_y;
```

```
// for debug printouts
private boolean debug = false;

// used to communicate with PD
private final PdDispatcher dispatcher = new PdDispatcher() {
        // prints out PD "post" console messages
        @Override
        public void print(String s) {
                Log.d(TAG, s);
        }
};

// print out messages to app GUI
private Toast toast = null;

private void toast(final String msg) {
        runOnUiThread(new Runnable() {
                @Override
                public void run() {
                        if (toast == null) {
                                toast = Toast.makeText(getApplicationContext(), "",
                                                Toast.LENGTH_LONG);
                        }
                        toast.setText(msg);
                        toast.show();
                }
        });
}

@Override
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // sensor checks and initialization
        initSensors();

        // find GUI elements and assign related variables
        initGui();

        // initialize PD engine and output errors
        try {
                initPd();
        } catch (IOException e) {
                Log.d(TAG, "Create error:" + e.toString());
                finish();
        }

}

@Override
protected void onResume() {
        super.onResume();

        // re-register sensor listeners
        if (hasAccelerometer) {
                senSensorManager.registerListener(this, senAccelerometer,
                                SensorManager.SENSOR_DELAY_NORMAL);
        }
        if (hasProximity) {
                senSensorManager.registerListener(this, senProximity,
                                SensorManager.SENSOR_DELAY_NORMAL);
        }

        // restart PD audio
        PdAudio.startAudio(this);
}

@Override
protected void onPause() {
        super.onPause();
```

```
        // unregister sensor listeners
        senSensorManager.unregisterListener(this);

        // stop PD audio
        PdAudio.stopAudio();
}

@Override
public void onDestroy() {
        super.onDestroy();

        // disconnect PD engine and release service
        PdAudio.release();
        PdBase.release();
}

@Override
public void onSensorChanged(SensorEvent sensorEvent) {
        Sensor mySensor = sensorEvent.sensor;

        if (debug) {
                Log.d(TAG, "Sensor changed...");
                Log.d(TAG, String.valueOf(mySensor.getType()));
        }

        // specific sensor handlers
        switch (mySensor.getType()) {
        // if the accelerometer has changed
        case (Sensor.TYPE_ACCELEROMETER):
                long curTime = System.currentTimeMillis();

                // get accelerometer values
                float x = (float) sensorEvent.values[0];
                float y = (float) sensorEvent.values[1];
                float z = (float) sensorEvent.values[2];
                z = 9.8f;

                // only send accelerometer values to PD once each 100ms
                if ((curTime - lastUpdate) > 100) {
                        lastUpdate = curTime;

                        // if accelerometer is working
                        if (raw_x != null) {
                                raw_x.setText(Float.toString(x));
                                raw_y.setText(Float.toString(y));
                                raw_z.setText(Float.toString(z));
                        }

                        // send value list to Abel_accIn object receiver
                        PdBase.sendList("ab_accel", Math.abs(x), Math.abs(y),
                                        Math.abs(z));
                }
                break;
        case (Sensor.TYPE_PROXIMITY):
                // send sensor value to Abel_proximityIn object receiver
                PdBase.sendFloat("ab_proximityIn", sensorEvent.values[0]);
                break;
        }
}

// unused but necessary
@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {

}

// touchscree event handlers
@Override
public boolean onTouchEvent(MotionEvent event) {
        int action = event.getActionMasked();
```

```
        switch (action) {
        case MotionEvent.ACTION_DOWN:
                // get touch press coordinates and convert to screen percentage
                initial_x = event.getX() * 100 / screen.widthPixels;
                initial_y = event.getY() * 100 / screen.heightPixels;
                break;
        case MotionEvent.ACTION_UP:
                // get touch release coordinates and convert to screen percentage
                float finalX = event.getX() * 100 / screen.widthPixels;
                float finalY = event.getY() * 100 / screen.heightPixels;

                // swipe right
                if (initial_x - finalX < -2) {
                        PdBase.sendFloat("ab_touchIn", 3);
                }
                // swipe left
                if (initial_x - finalX > 2) {
                        PdBase.sendFloat("ab_touchIn", 2);
                }
                // swipe down
                if (initial_y - finalY < -2) {
                        PdBase.sendFloat("ab_touchIn", 1);
                }
                // swipe up
                if (initial_y - finalY > 2) {
                        PdBase.sendFloat("ab_touchIn", 0);
                }
                // touch/tap
                if (initial_x == finalX && initial_y == finalY) {
                        PdBase.sendList("ab_touchIn", initial_x, initial_y);
                }

                break;
        }

        return super.onTouchEvent(event);
}

private void initSensors() {

        // build device's sensor list and show in logcat
        if (debug) {
                sMgr = (SensorManager) this.getSystemService(SENSOR_SERVICE);
                List<Sensor> list = sMgr.getSensorList(Sensor.TYPE_ALL);

                String data = new String();

                for (Sensor sensor : list) {
                        data += (sensor.getName() + "\n");
                        data += (sensor.getVendor() + "\n");
                        data += (sensor.getVersion() + "\n");
                }

                Log.d(TAG, data);
        }

        // sensor availability checks
        PackageManager manager = getPackageManager();
        hasAccelerometer = manager
                        .hasSystemFeature(PackageManager.FEATURE_SENSOR_ACCELEROMETER);
        hasProximity = manager
                        .hasSystemFeature(PackageManager.FEATURE_SENSOR_PROXIMITY);

        senSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);

        if (hasAccelerometer) {
                senAccelerometer = senSensorManager
                                .getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
                senSensorManager.registerListener(this, senAccelerometer,
                                SensorManager.SENSOR_DELAY_NORMAL);
```

```
                        if (debug)
                                Log.d(TAG, Float.toString(senAccelerometer.getMaximumRange()));
                } else {
                        if (debug)
                                Log.d(TAG, "No acceleration sensor");
                }

                if (hasProximity) {
                        senProximity = senSensorManager
                                        .getDefaultSensor(Sensor.TYPE_PROXIMITY);
                        senSensorManager.registerListener(this, senProximity,
                                        SensorManager.SENSOR_DELAY_NORMAL);

                        if (debug)
                                Log.d(TAG,
                                                "Proximity max:"
                                                                +
Float.toString(senProximity.getMaximumRange()));
                } else {
                        if (debug)
                                Log.d(TAG, "No proximity sensor");
                }

        }

        private void initGui() {
                setContentView(R.layout.activity_main);

                // set gui variables to corresponding text fields
                raw_x = (TextView) findViewById(R.id.lbl_raw_x);
                raw_y = (TextView) findViewById(R.id.lbl_raw_y);
                raw_z = (TextView) findViewById(R.id.lbl_raw_z);

                acc_max = (TextView) findViewById(R.id.lbl_max);
                if (hasAccelerometer)
                        acc_max.setText(Float.toString(senAccelerometer.getMaximumRange()));

                // assign text fields for PD received values
                lbl_pitch = (TextView) findViewById(R.id.lbl_pitch);
                lbl_roll = (TextView) findViewById(R.id.lbl_roll);
                lbl_accel = (TextView) findViewById(R.id.lbl_accel);

                getWindowManager().getDefaultDisplay().getMetrics(screen);
        }

        private final PdListener pdListener = new PdListener.Adapter() {
                @Override
                public void receiveList(String source, Object... args) {
                        // received from the ab_accelerate sender in test.pd
                        if (source.equals("ab_accelerate")) {
                                if (args.length < 3 || !(args[0] instanceof Float)
                                                || !(args[1] instanceof Float)
                                                || !(args[2] instanceof Float))
                                        return;
                                float pitch = (Float) args[0];
                                float roll = (Float) args[1];
                                float accel = (Float) args[2];

                                // set text fields to received values
                                lbl_pitch.setText(Float.toString(pitch));
                                lbl_roll.setText(Float.toString(roll));
                                lbl_accel.setText(Float.toString(accel));
                        }
                }

                @Override
                public void receiveSymbol(String source, String symbol) {
                        // received from Abel_colorOut
                        if (source.equals("ab_rgb")) {
                                changeColor(symbol);
                        }
```

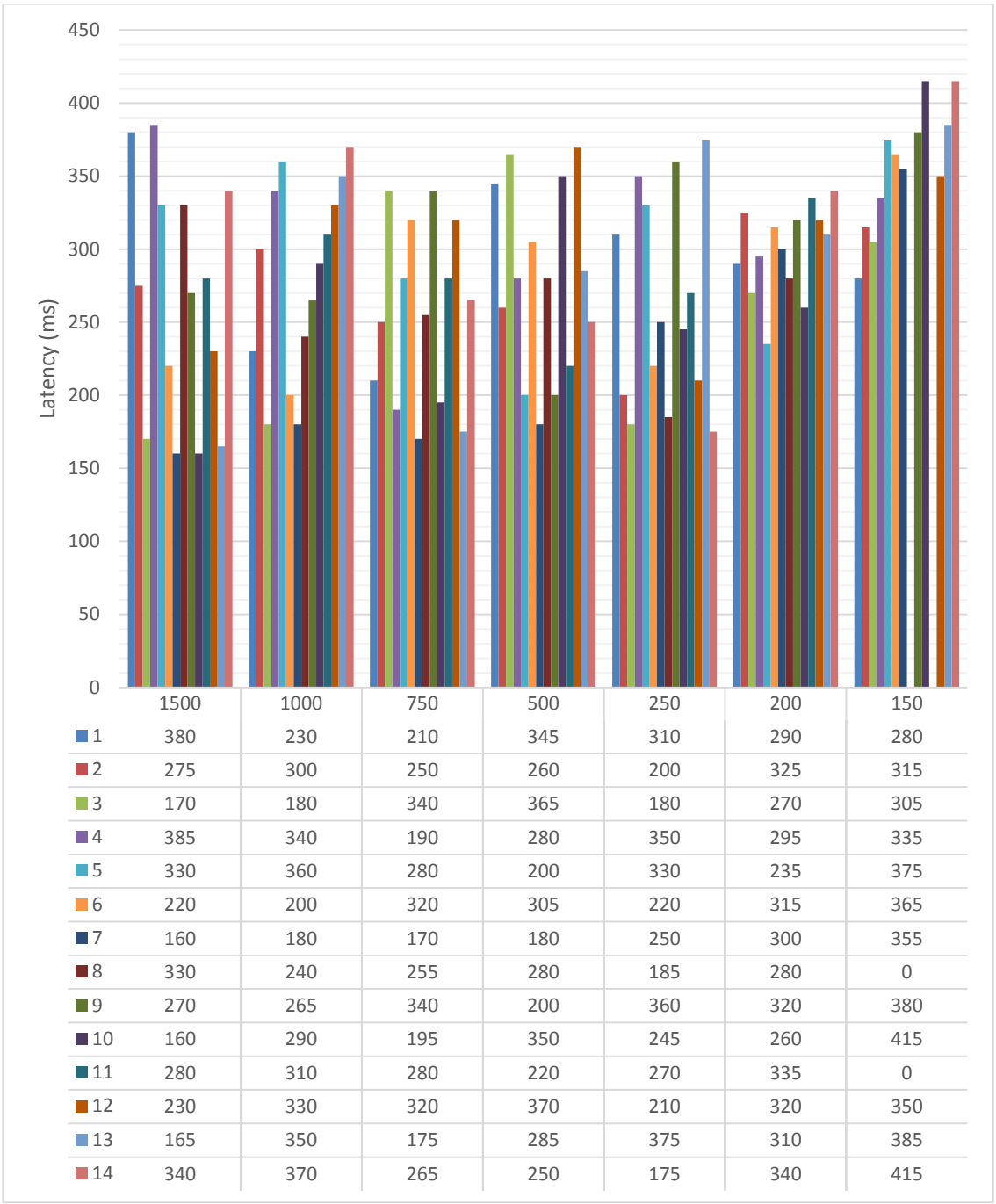
```
                        // received from Abel_msgOut
                        if (source.equals("ab_msg")) {
                                toast(symbol);
                        }
                }
        };

        private void initPd() throws IOException {
                int sampleRate = AudioParameters.suggestSampleRate();

                Resources res = getResources();
                File patchFile = null;

                // set path to find compiled externals
                // "/data/data/" + getPackageName() + "/lib"
                PdBase.addToSearchPath(getApplicationInfo().dataDir + "/lib");

                // init engine only AFTER setting externals directory
                PdAudio.initAudio(sampleRate, 0, 2, 8, true);

                // set dispatcher and listeners to receive data from PD
                PdBase.setReceiver(dispatcher);
                dispatcher.addListener("ab_accelerate", pdListener);
                dispatcher.addListener("ab_rgb", pdListener);
                dispatcher.addListener("ab_msg", pdListener);

                // extract and open test.pd patch
                InputStream in = res.openRawResource(R.raw.test);
                patchFile = IoUtils.extractResource(in, "test.pd", getCacheDir());
                PdBase.openPatch(patchFile);

                // if accelerometer sensor is present, set max range of Abel_accIn
                // for scale remapping
                if (hasAccelerometer) {
                        PdBase.sendMessage("ab_accel", "setdefault", 9.8f,
                                        senAccelerometer.getMaximumRange());
                }
        }

        // method to change GUI background color
        private void changeColor(String color) {
                View view = this.getWindow().getDecorView();
                view.setBackgroundColor(Color.parseColor("#" + color));
        }
}
```

Appendix C

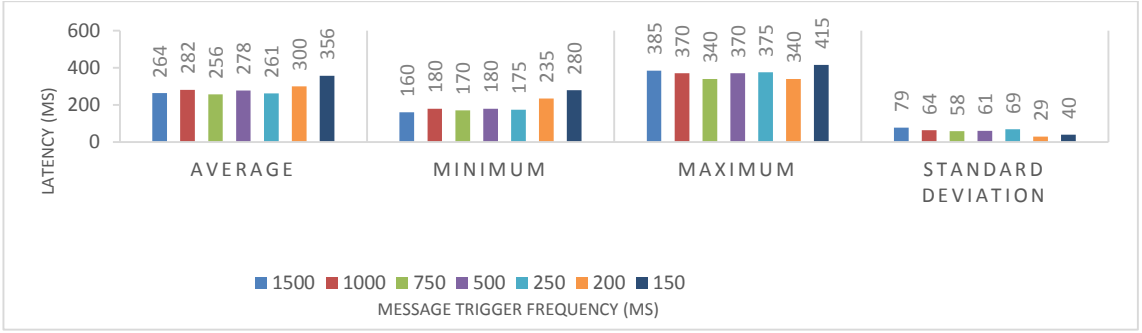
Testing result charts

Note: a 0 in the following result charts corresponds to a missed message, not to a 0 latency response.

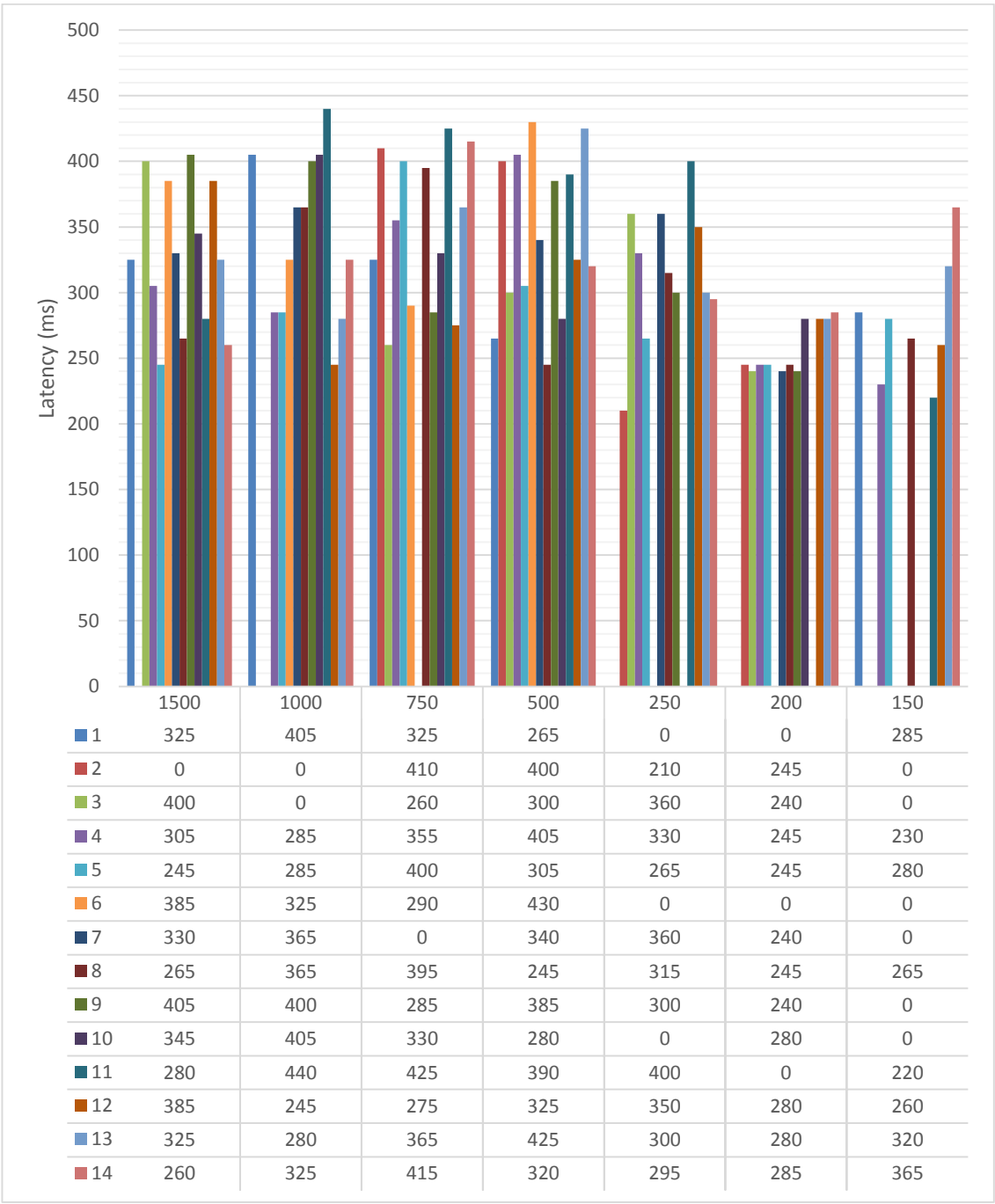
9.1 Device #1 – HTM



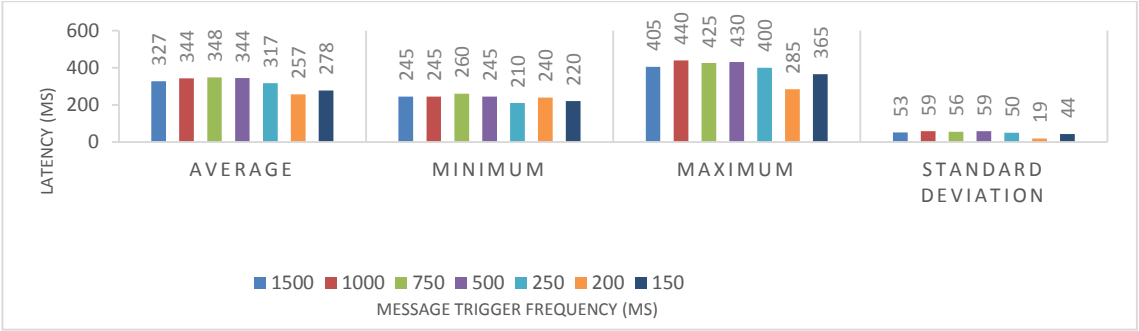
	1500	1000	750	500	250	200	150
1	380	230	210	345	310	290	280
2	275	300	250	260	200	325	315
3	170	180	340	365	180	270	305
4	385	340	190	280	350	295	335
5	330	360	280	200	330	235	375
6	220	200	320	305	220	315	365
7	160	180	170	180	250	300	355
8	330	240	255	280	185	280	0
9	270	265	340	200	360	320	380
10	160	290	195	350	245	260	415
11	280	310	280	220	270	335	0
12	230	330	320	370	210	320	350
13	165	350	175	285	375	310	385
14	340	370	265	250	175	340	415



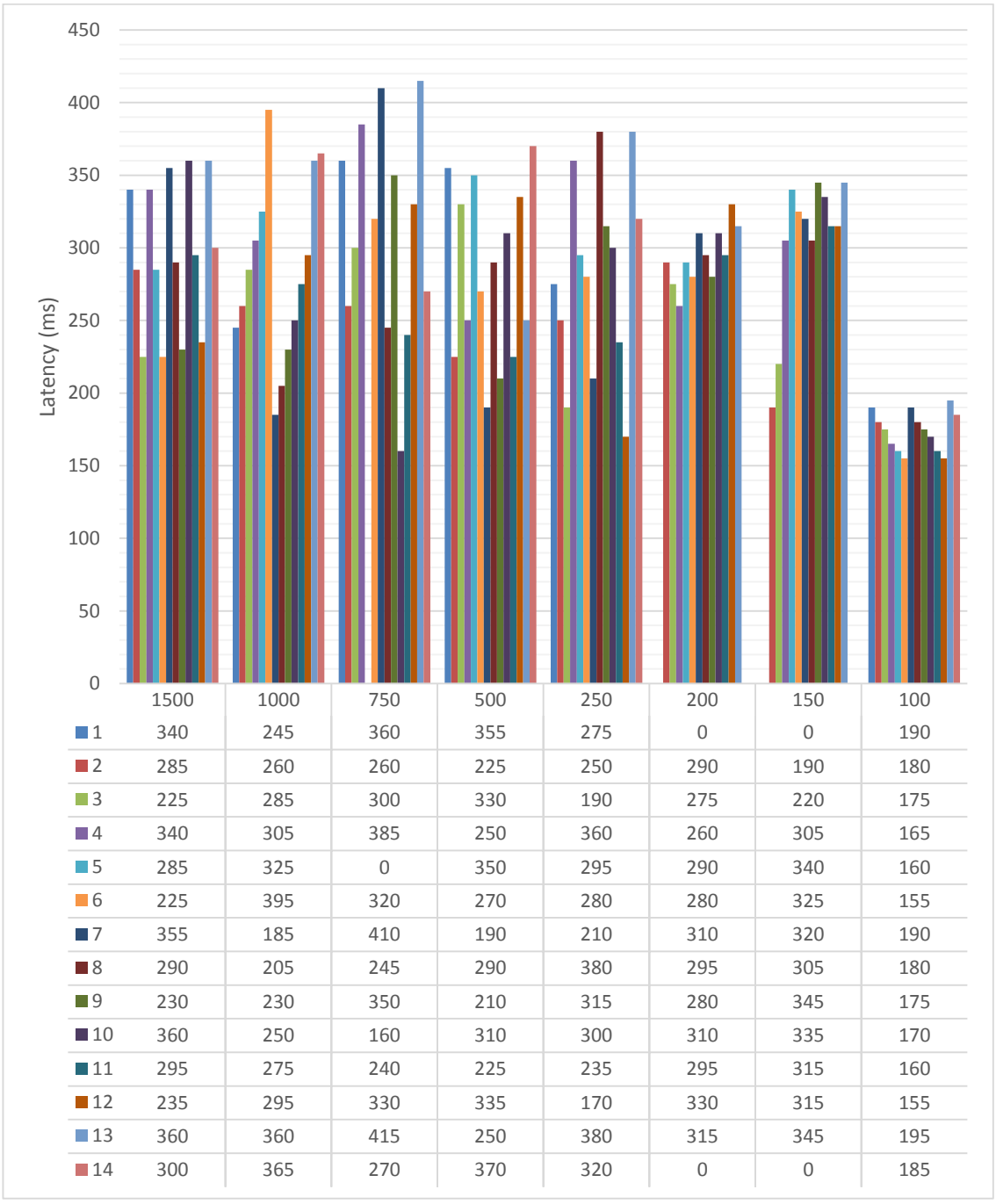
9.2 Device #2 – Galaxy Tab



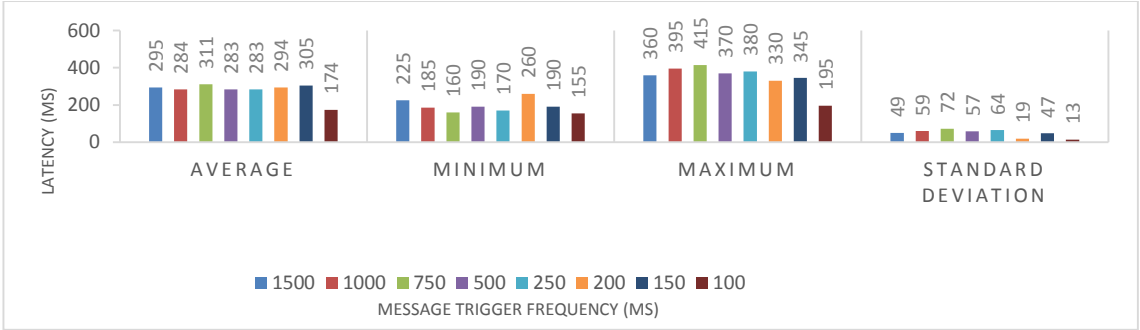
	1500	1000	750	500	250	200	150
1	325	405	325	265	0	0	285
2	0	0	410	400	210	245	0
3	400	0	260	300	360	240	0
4	305	285	355	405	330	245	230
5	245	285	400	305	265	245	280
6	385	325	290	430	0	0	0
7	330	365	0	340	360	240	0
8	265	365	395	245	315	245	265
9	405	400	285	385	300	240	0
10	345	405	330	280	0	280	0
11	280	440	425	390	400	0	220
12	385	245	275	325	350	280	260
13	325	280	365	425	300	280	320
14	260	325	415	320	295	285	365



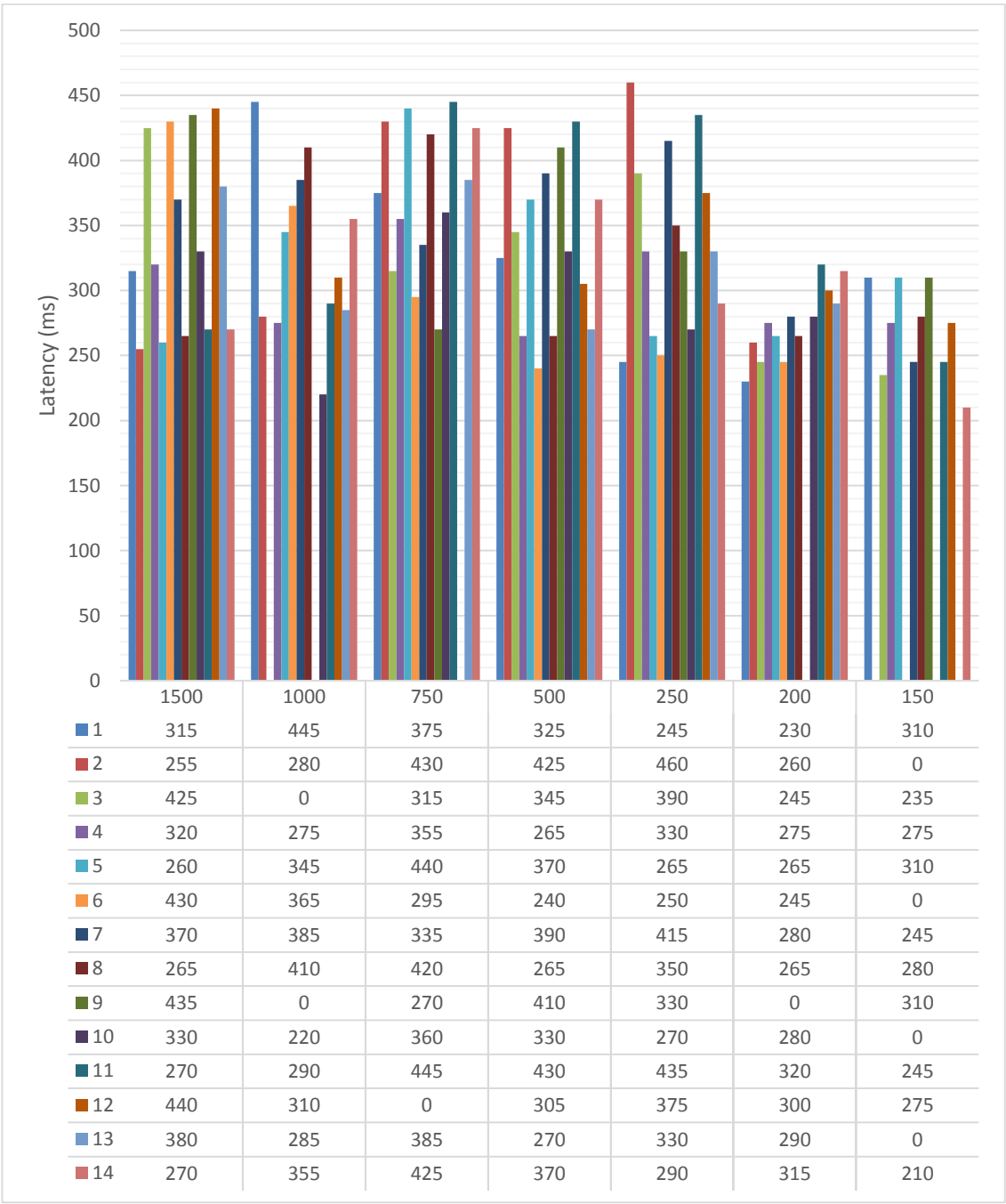
9.3 Device #3 – Galaxy S3 LTE



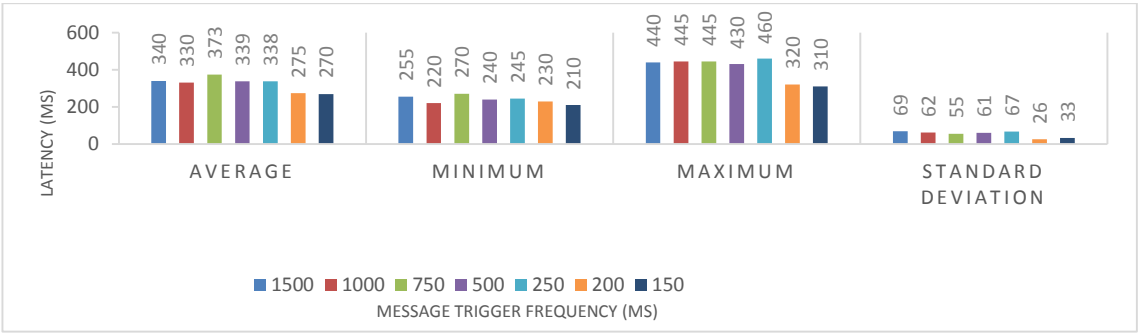
	1500	1000	750	500	250	200	150	100
1	340	245	360	355	275	0	0	190
2	285	260	260	225	250	290	190	180
3	225	285	300	330	190	275	220	175
4	340	305	385	250	360	260	305	165
5	285	325	0	350	295	290	340	160
6	225	395	320	270	280	280	325	155
7	355	185	410	190	210	310	320	190
8	290	205	245	290	380	295	305	180
9	230	230	350	210	315	280	345	175
10	360	250	160	310	300	310	335	170
11	295	275	240	225	235	295	315	160
12	235	295	330	335	170	330	315	155
13	360	360	415	250	380	315	345	195
14	300	365	270	370	320	0	0	185



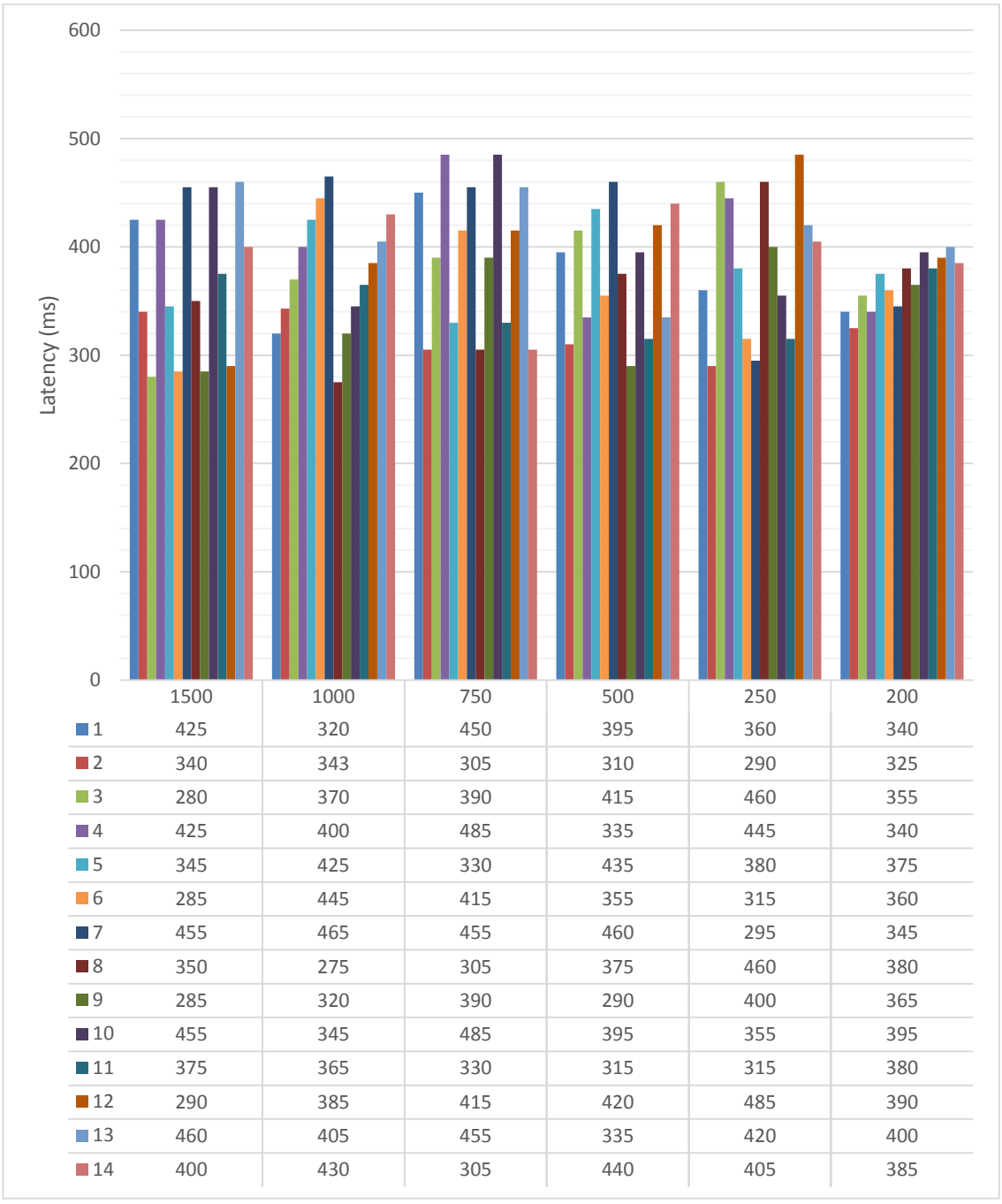
9.4 Device #4 – Galaxy S4 Mini



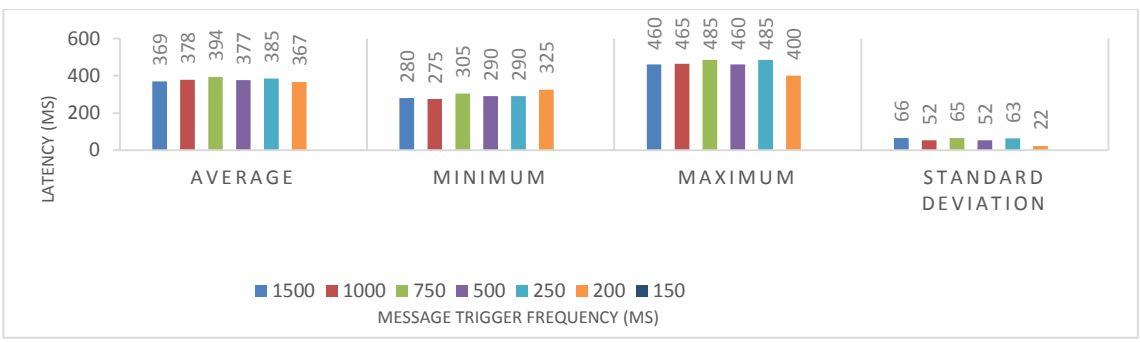
	1500	1000	750	500	250	200	150
1	315	445	375	325	245	230	310
2	255	280	430	425	460	260	0
3	425	0	315	345	390	245	235
4	320	275	355	265	330	275	275
5	260	345	440	370	265	265	310
6	430	365	295	240	250	245	0
7	370	385	335	390	415	280	245
8	265	410	420	265	350	265	280
9	435	0	270	410	330	0	310
10	330	220	360	330	270	280	0
11	270	290	445	430	435	320	245
12	440	310	0	305	375	300	275
13	380	285	385	270	330	290	0
14	270	355	425	370	290	315	210



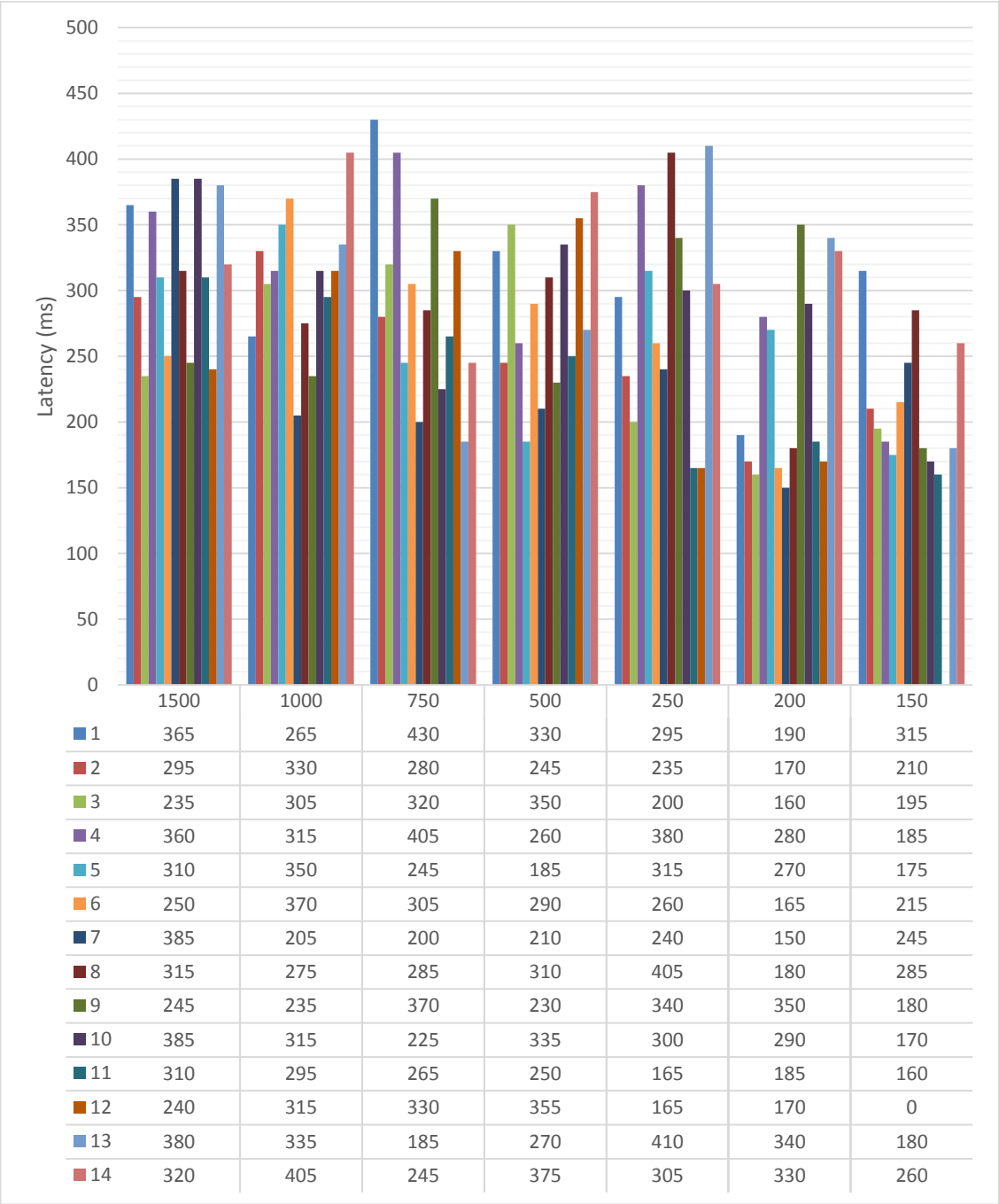
9.5 Device #5 – Ainol Novo 7 Venus



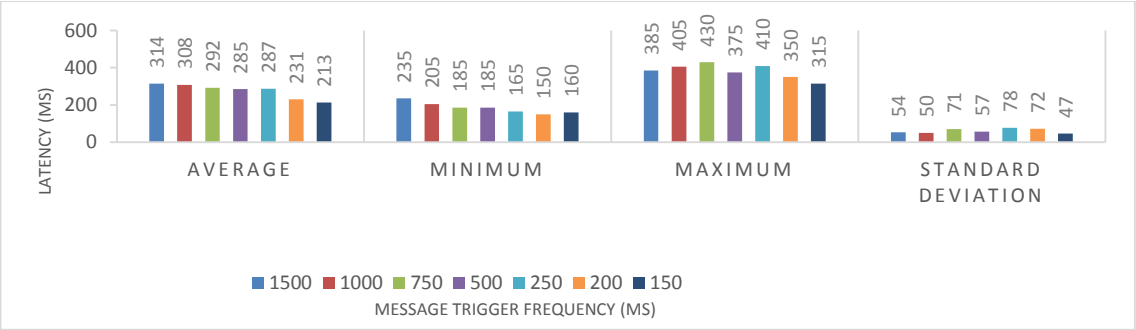
	1500	1000	750	500	250	200
1	425	320	450	395	360	340
2	340	343	305	310	290	325
3	280	370	390	415	460	355
4	425	400	485	335	445	340
5	345	425	330	435	380	375
6	285	445	415	355	315	360
7	455	465	455	460	295	345
8	350	275	305	375	460	380
9	285	320	390	290	400	365
10	455	345	485	395	355	395
11	375	365	330	315	315	380
12	290	385	415	420	485	390
13	460	405	455	335	420	400
14	400	430	305	440	405	385



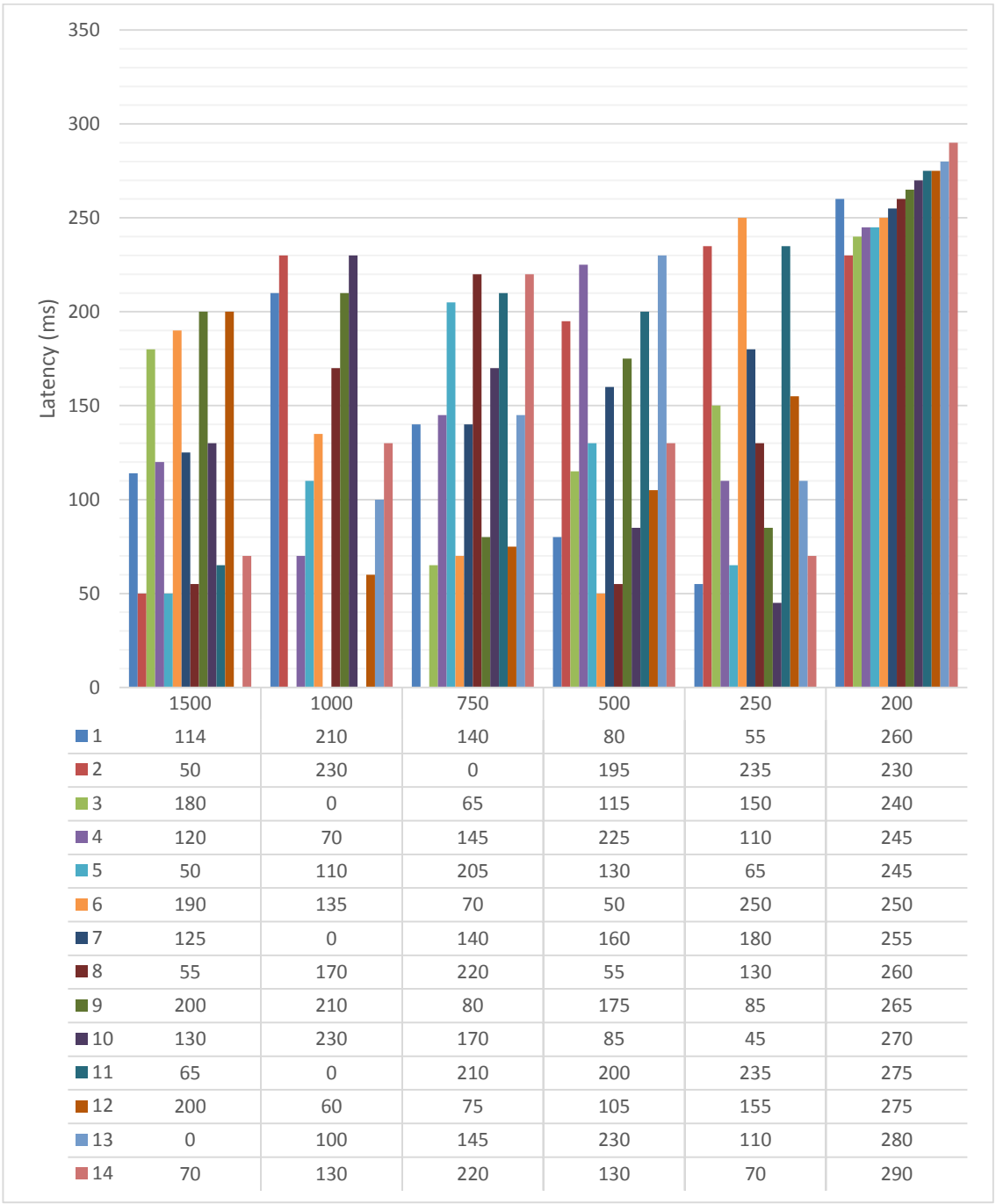
9.6 Device #6 – Jiayu G3



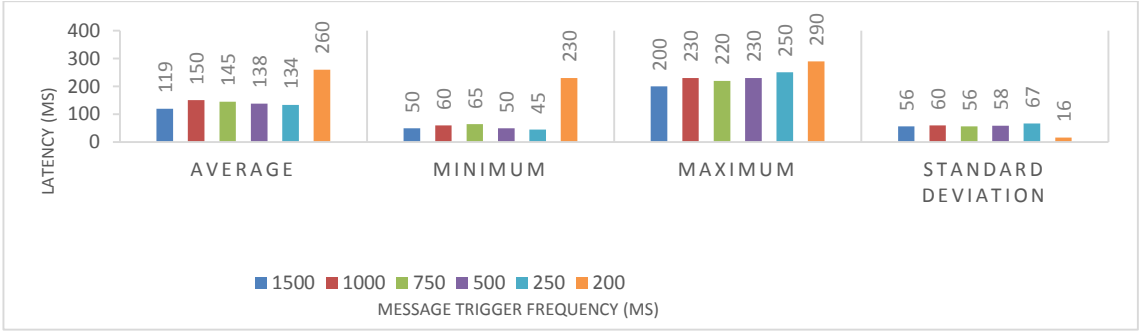
	1500	1000	750	500	250	200	150
1	365	265	430	330	295	190	315
2	295	330	280	245	235	170	210
3	235	305	320	350	200	160	195
4	360	315	405	260	380	280	185
5	310	350	245	185	315	270	175
6	250	370	305	290	260	165	215
7	385	205	200	210	240	150	245
8	315	275	285	310	405	180	285
9	245	235	370	230	340	350	180
10	385	315	225	335	300	290	170
11	310	295	265	250	165	185	160
12	240	315	330	355	165	170	0
13	380	335	185	270	410	340	180
14	320	405	245	375	305	330	260



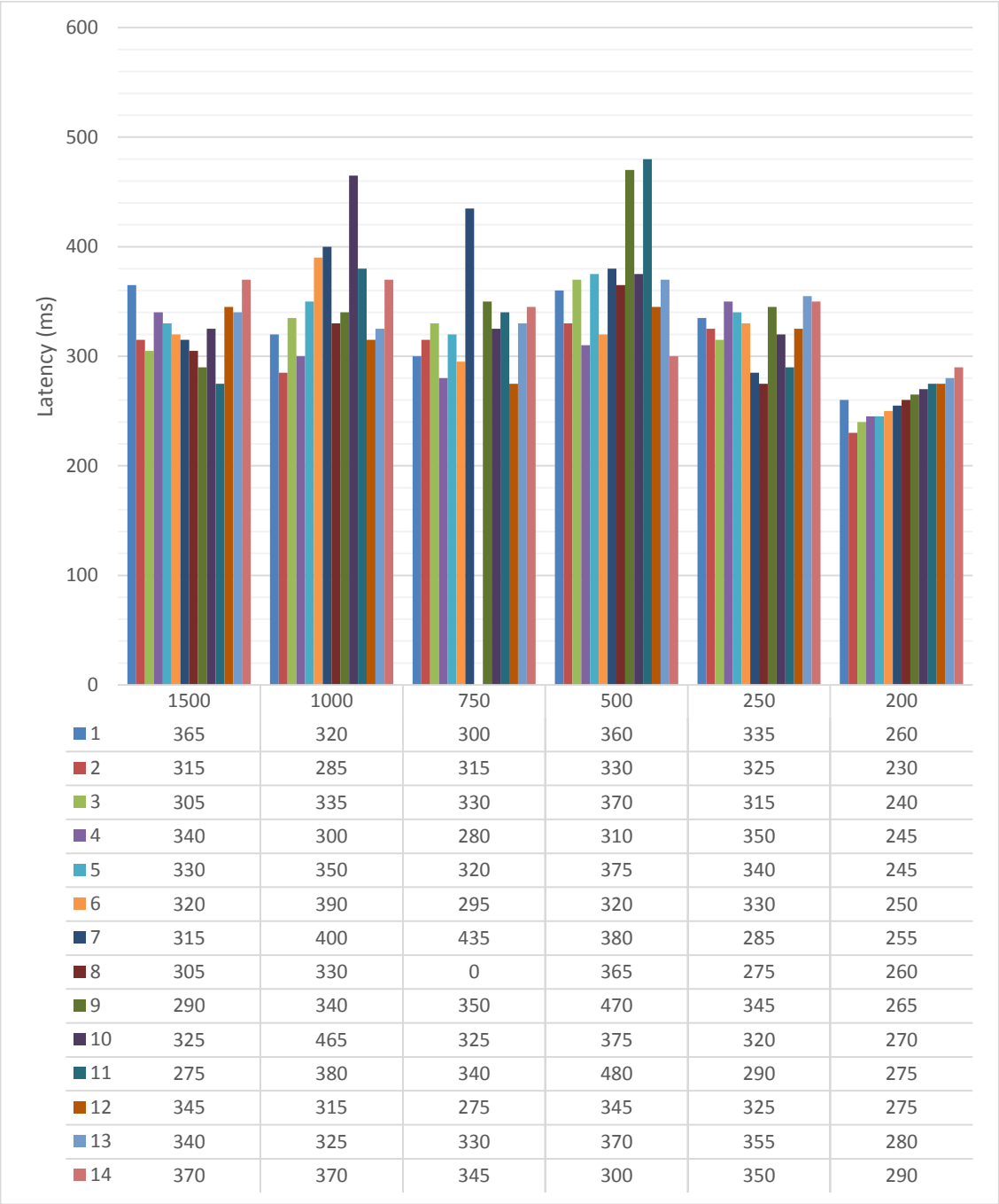
9.7 Device #7 – One plus one



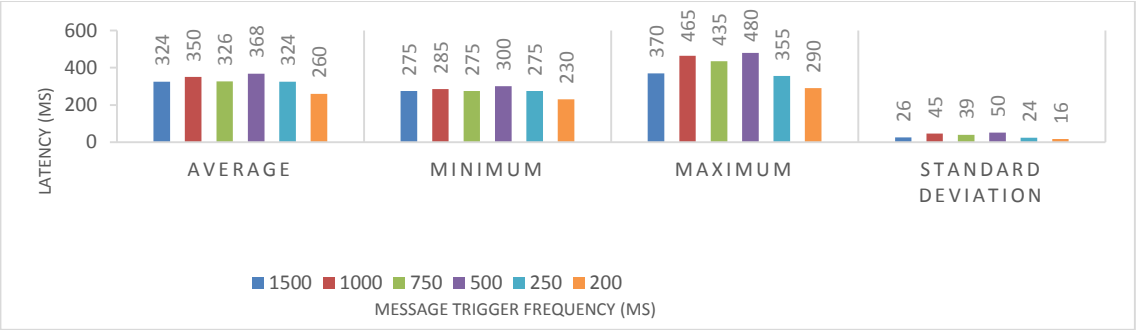
	1500	1000	750	500	250	200
1	114	210	140	80	55	260
2	50	230	0	195	235	230
3	180	0	65	115	150	240
4	120	70	145	225	110	245
5	50	110	205	130	65	245
6	190	135	70	50	250	250
7	125	0	140	160	180	255
8	55	170	220	55	130	260
9	200	210	80	175	85	265
10	130	230	170	85	45	270
11	65	0	210	200	235	275
12	200	60	75	105	155	275
13	0	100	145	230	110	280
14	70	130	220	130	70	290



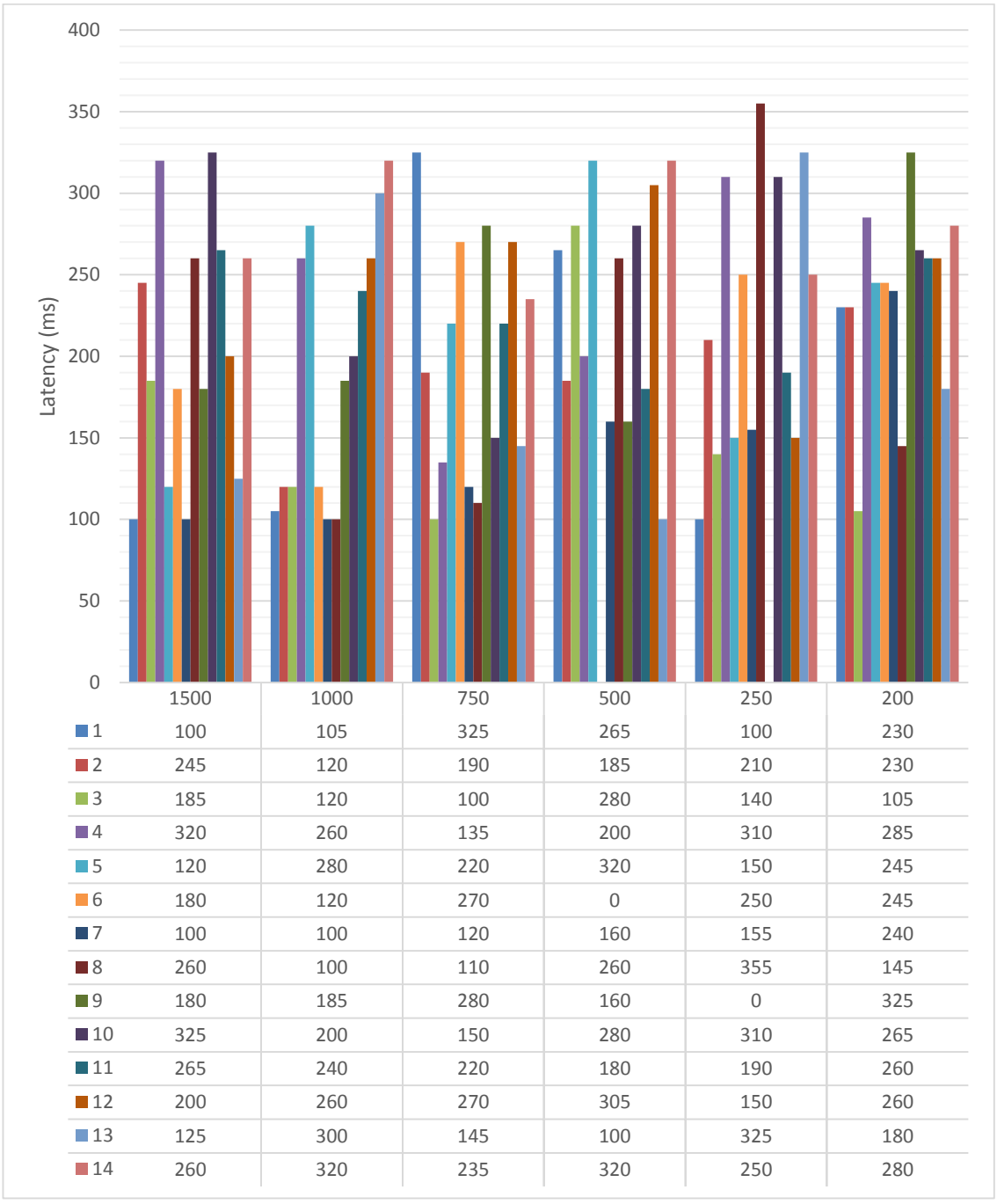
9.8 Device #8 – Lazer



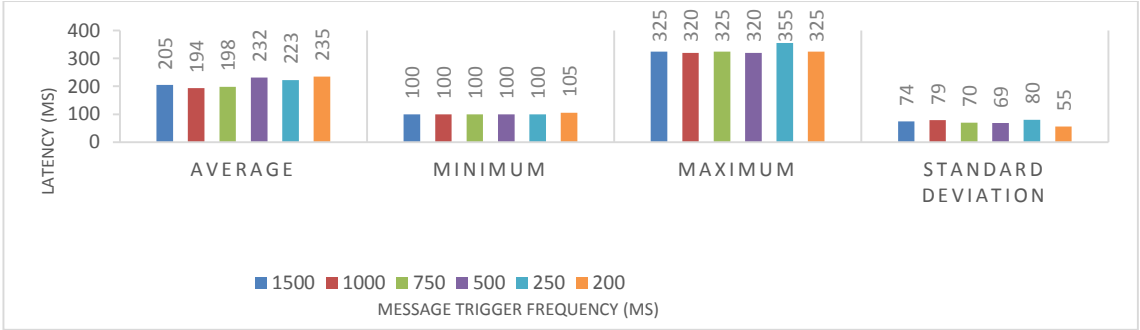
	1500	1000	750	500	250	200
1	365	320	300	360	335	260
2	315	285	315	330	325	230
3	305	335	330	370	315	240
4	340	300	280	310	350	245
5	330	350	320	375	340	245
6	320	390	295	320	330	250
7	315	400	435	380	285	255
8	305	330	0	365	275	260
9	290	340	350	470	345	265
10	325	465	325	375	320	270
11	275	380	340	480	290	275
12	345	315	275	345	325	275
13	340	325	330	370	355	280
14	370	370	345	300	350	290



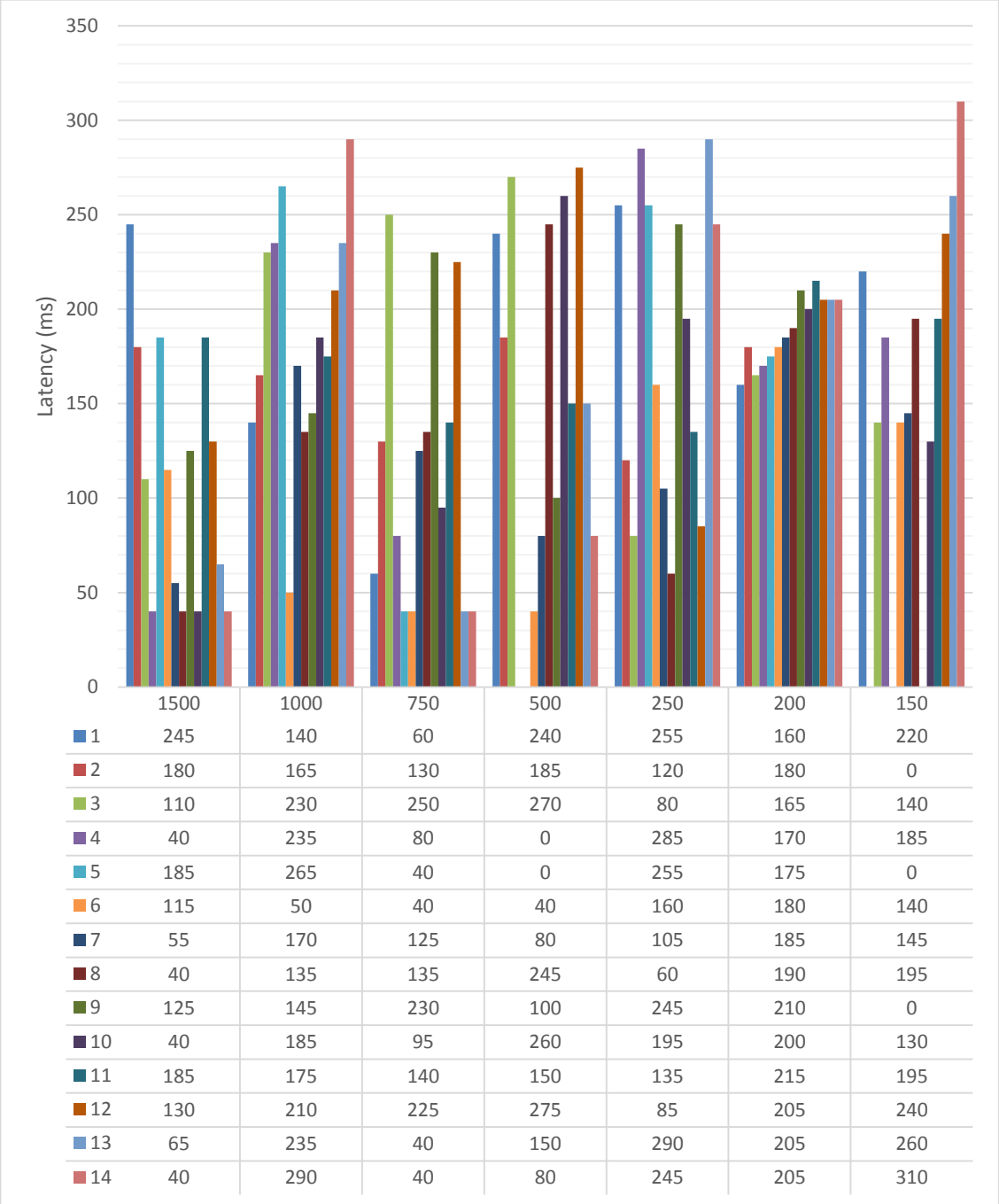
9.9 Device #9 – Moto G



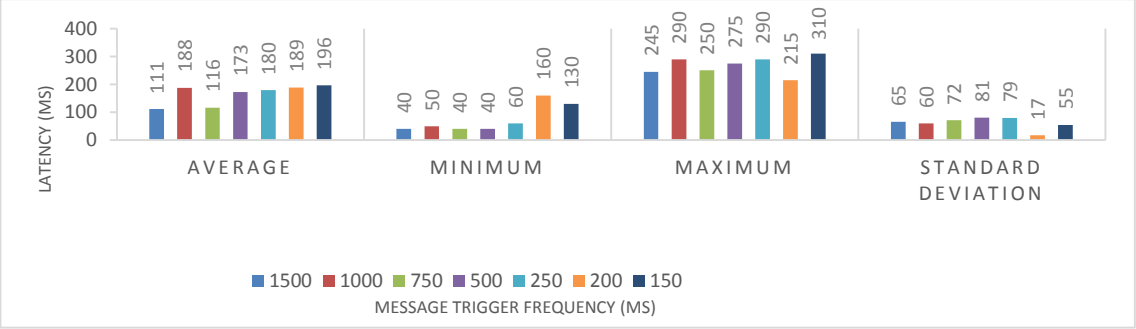
	1500	1000	750	500	250	200
1	100	105	325	265	100	230
2	245	120	190	185	210	230
3	185	120	100	280	140	105
4	320	260	135	200	310	285
5	120	280	220	320	150	245
6	180	120	270	0	250	245
7	100	100	120	160	155	240
8	260	100	110	260	355	145
9	180	185	280	160	0	325
10	325	200	150	280	310	265
11	265	240	220	180	190	260
12	200	260	270	305	150	260
13	125	300	145	100	325	180
14	260	320	235	320	250	280



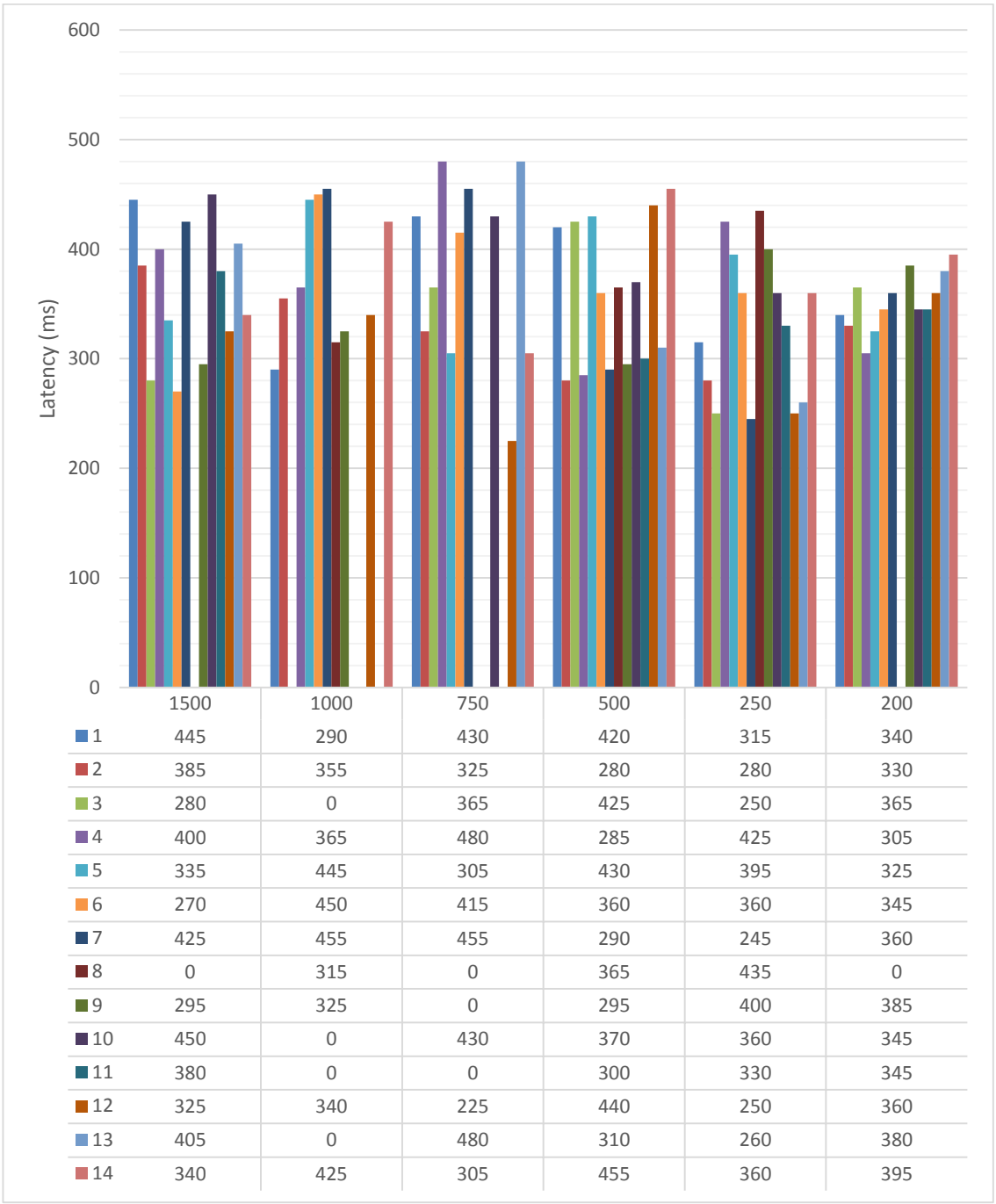
9.10 Device #10 – Nexus 7



	1500	1000	750	500	250	200	150
1	245	140	60	240	255	160	220
2	180	165	130	185	120	180	0
3	110	230	250	270	80	165	140
4	40	235	80	0	285	170	185
5	185	265	40	0	255	175	0
6	115	50	40	40	160	180	140
7	55	170	125	80	105	185	145
8	40	135	135	245	60	190	195
9	125	145	230	100	245	210	0
10	40	185	95	260	195	200	130
11	185	175	140	150	135	215	195
12	130	210	225	275	85	205	240
13	65	235	40	150	290	205	260
14	40	290	40	80	245	205	310



9.11 Device #11- Galaxy S3 i9300 (cyanogen)



	1500	1000	750	500	250	200
1	445	290	430	420	315	340
2	385	355	325	280	280	330
3	280	0	365	425	250	365
4	400	365	480	285	425	305
5	335	445	305	430	395	325
6	270	450	415	360	360	345
7	425	455	455	290	245	360
8	0	315	0	365	435	0
9	295	325	0	295	400	385
10	450	0	430	370	360	345
11	380	0	0	300	330	345
12	325	340	225	440	250	360
13	405	0	480	310	260	380
14	340	425	305	455	360	395