Proc. of the 5th Int. Conference on Digital Audio Effects (DAFx-02), Hamburg, Germany, September 26-28, 2002

A C++ DEVELOPMENT PLATFORM FOR REAL TIME AUDIO PROCESSING AND SYNTHESIS APPLICATIONS

Enrique Robledo Arnuncio, Rubén Hinojosa, Maarten de Boer

Music Technology Group
Universidad Pompeu Fabra
{erobledo,rhinojos,mdeboer}@iua.upf.es

ABSTRACT

The computational power provided by current general purpose computers allows to undertake the implementation of low cost software-only real time audio processors. Unfortunately, computational power is not the only requirement for high demand applications. There are still important difficulties to overcome in other areas such as robustness and low latency. These difficulties have lead us to the development of *Rappid*, a development framework for C++ real-time high-demand audio processing applications.

This paper describes the *Rappid* development framework. First of all we discuss the objectives we pursue with its development, and we give an overview to some other existent solutions before starting the actual description of the framework. We finally describe a first sound processing application which has been successfully developed with it.

1. OBJECTIVES

In the computer music research field it is not possible to evaluate a given sound synthesis or processing algorithm without listening to its output. If the algorithm has real time control parameters, proper experimentation can only be done with a real time implementation and a human interpreter involved. This is necessary from the earliest stages of research.

This is the reason why computer music researchers very often become computer music application developers. It is thus important for them to have a good application development framework, so that the loop time between the algorithm idea and its evaluation is reduced as much as possible.

Converting a research application prototype into an application wich can be used as a musical instrument in other scenarios should not be hard work if both the framework and the developer take enough care of requirements such as efficiency, latency and robustness. This is what our framework aims to achieve. We state this in more detail in the following list of objectives.

1.1. Easy development of real time sound processing applications

When writing a sound synthesis or processing application, a big part of the time is usually spent on dealing with technical details such as file and sound input/output, separating the real-time and the non real-time code, setting up multi-threading, configuring thread priorities, etc. We try to reduce this burden by providing a high level platform independent interface for such tasks.

The other, more important task of the sound application developer is the implementation of the algorithm. *Rappid* is not a library of sound processing algorithms, but is meant to be used in conjunction with CLAM [1], wich is one of such libraries.

In other words, the framework we present can be seen as an extension to the CLAM library for the development of real time sound processing applications.

1.2. Live musical performances

The main motivation for the development of *Rappid* is the need to develop a software audio processor for live musical performances. The general idea is feeding the sound of several real instruments to the processor, and achieving extra *synthetic* instruments by transformation and combination of the real ones in real time, all of this with high quality audio inputs and outputs and with no sound artifacts.

1.3. Using CLAM

One of our main goals of *Rappid* is to beta test *CLAM*, a library of C++ classes for Audio and Music processing which is also being developed at the *Music Technology Group* [1].

As it has already been mentioned, *Rappid* is meant to be used with CLAM, but in addition to this, *Rappid* itself makes extensive use of this library, for mid-level input/output operations, configuration storage, etc.

1.4. General availability

Another important goal for us is to include the result of our work as a deliverable of the *AGNULA* project [2], which aims to provide reference distributions for the GNU/Linux operating system completely based on Free Software and completely devoted to professional and consumer audio and multimedia applications.

This objective made it impossible to base our work on closed-source solutions, and unfortunately most of the currently available commercial frameworks, such as the most popular audio plug-in systems, do not satisfy this requirement.

2. PC-BASED AUDIO PROCESSING

Nowadays a wide range of different PC-Based sound processing solutions exists. We can differentiate three main categories, depending on the target users:

Hobbyist tools: These tools usually satisfy the low-cost requirement, but are usually not flexible or powerful enough. Example of these: some music creation software with basic sound processing capabilities bundled in it.

Research tools: Sometimes cheap, coming from the academia or free software world, sometimes expensive, coming from companies. The most often lack real time capabilities, or features to ease the creation of simple user interfaces. Examples of these are Matlab, Octave, Ptolemy, etc.

Professional tools: They can be very powerful, especially if they involve the usage of specialized hardware, such as the Protools systems by Digidesign, but they are, until now, very expensive commercial products.

These kind of tools are most often based on the VST framework, the industry de-facto standard for sound applications in the Windows and Macintosh environments.

Our functional requirements would lead to choose some framework in the third category, but unfortunately, as we have already seen, the other requirements prevent that.

Recently good sound support is becoming available in the Free Software world, with the ALSA [3] system getting close to its first release, and more and more sound cards supporting it. This, added to the good reliability and latency capabilities that GNU/Linux systems provide [4], makes these platforms very appealing for real time sound processing applications.

Many such applications have started to emerge in the Free Software world, as can be seen in Dave Phillips popular sound software index [5]. In contrast to the commercial world, many of these are small and very specific standalone sound processing tools, such as Tapiir [6].

Until now, though, there is not a clear standard framework for sound applications development on GNU/Linux environments. Some very promising solutions, such as Jack and LADSPA [7], are being developed, and we will probably see improvements in this area in the next months. They will probably allow, for example, to use many of the small applications mentioned in the previous paragraph in conjunction with each other.

But all of these frameworks provide only low level C APIs for sound input and output; a higher level of abstraction is convenient to ease the development of complex applications with involve interactive graphical interfaces.

Also, in the sound processing algorithms domain, some very promising libraries are starting to be available. Examples of these include CLAM, on which *Rappid* is based, or the SndObj library [8], which provides a more compact set of classes for time domain processing.

3. SYSTEM OVERVIEW

Rappid is designed to fit a live performance use case, for which the application requires:

- A set of audio inputs, corresponding to some external audio sources in the performance, such as natural instruments.
- A set of audio outputs, corresponding to synthetic instruments generated by the application.
- A set of control parameters, which define which aspects of the processing algorithm can be modified from the user interface.

These elements are implemented as a set of C++ classes and services, which are organized in an architecture as described in figure 1. Actually, there is a *Rappid* class which contains all the processing modules. The graphical user interface can use the public methods of this class to control it.

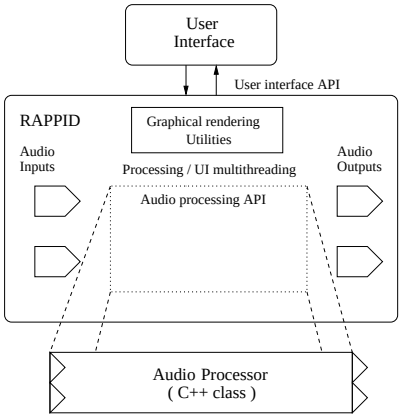

Figure 1: *Rappid architecture.*

The figure also shows the components that the application developer needs to implement: the audio processor and the user interface. The system provides two different programming interfaces for both parts of the application. The current version of the framework does not yet provide a final version of these APIs. It is still a proof of concept to check that the general approach is valid.

3.1. API for the audio processors

Developers of new real time audio processing algorithms in C++ can take advantage of several services from *Rappid*:

3.1.1. Frame based real time audio input and output

Following the CLAM way of doing things, the programmer has to provide an *execution* C++ method in his processing class, which will be called each time a new audio frame is available at the input.

Each input and output can be connected to a real-time audio device or to a file on disk once the application is run, depending on user choice. The audio processor does not have to worry about this.

3.1.2. Multi-threading and process priority management

The developer does not need to care about this issues. *Rappid* creates an audio processing thread, sets it to an adequate operating system scheduling policy, and handles synchronization in the communication with the user interface thread. Also, a watchdog mechanism avoids system locks caused by errors when the processing thread priority is high.

3.1.3. Basic graphical capabilities for debugging

This capability is still in an experimental stage. A simple graphical application will be available for developers of processing modules, which should allow the visualization of input and output signals, as well as internal ones.

3.1.4. Spectral analysis an synthesis

This is also still in an experimental stage. For spectral domain processing, *Rappid* will provide out of the box STFT data blocks calculated from an audio input. Thus, a *Rappid* based processor will be able to directly take spectral frames as inputs, and write spectral frames as outputs.

3.2. API for the user interface application

Of course, many applications won't have enough with the small graphical display and control capabilities *Rappid* aims to provide. These applications can implement their own graphical interface, and use a whole *Rappid* system as a C++ library class, through a simple control and configuration interface.

One of the main advantages of *Rappid* for these applications is isolating the complexity of multi-threading. The application does not need to know that the processing code is running in a different thread. It just need to use the control interface to start/stop the system, change processing parameters in run time, or ask the visualization interface to provide it with the data to visualize. *Rappid* takes care of the initialization of multi-threading, and of the synchronization in these data transactions.

On the other hand, the application developer has full control on the graphical user interface implementation. This is the main difference with existent plug-in frameworks, which provide the developer with a very limited graphics API.

4. SYSTEM ARCHITECTURE

Figure 2 shows the dataflow diagram of *Rappid*. Each block in the figure corresponds to a CLAM processing object. It can be seen there that the whole audio processor is composed of a set of processing modules, each of which can take all the system inputs as its inputs, and generate an output.

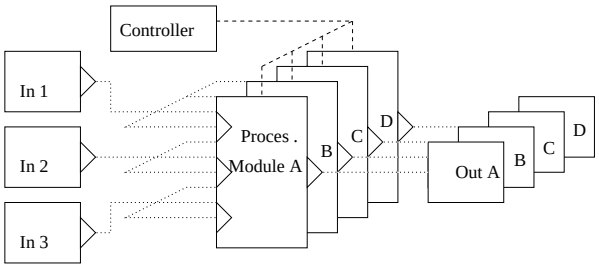

Figure 2: Dataflow diagram of a possible *Rappid* application.

The figure also shows how the user interface part has no relation with the data flow, but for the (asynchronous) control mechanism.

4.1. Current implementation

The current *Rappid* implementation still forces a very high interdependency between the framework services and the processing modules. Adding or modifying processing modules requires extending the *Rappid* class and compiling the whole framework.

The class diagram of an example application is shown in figure 3. This figure represents the same example as in figure 2.

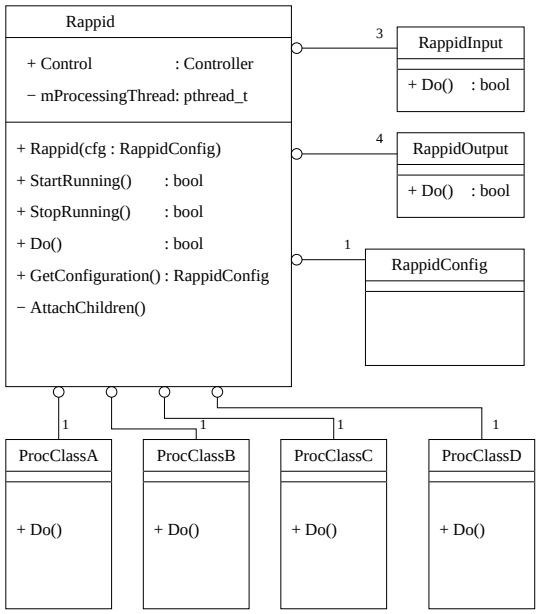

Figure 3: *Rappid* class diagram.

4.2. Current work

We are now working on a more flexible implementation, allowing dynamic linking of processing modules into the application, and dynamic configuration of all the *Rappid* parameters, so that it is not necessary to recompile it when the processing module or the graphical interface change.

The structure of the application will be like the one shown in figure 4. This will allow greater flexibility, but will probably cause a small performance penalty.

5. IMPLEMENTATION DETAILS

5.1. Development environment

Rappid will probably be a multi-platform framework in future, but the current version is being developed with the GNU development tool-set, and uses several POSIX tools and Linux kernel services.

5.1.1. The CLAM library

CLAM is a very rich collection of C++ classes. It provides audio processing tools, data structures, XML support, input and output abstraction, etc. *Rappid* processors will typically be implemented using CLAM Processing objects as building blocks.

C++ is the language of choice of CLAM because of its balanced combination of high level language support for object oriented and generative programming, and the possibility of obtaining very efficient object code.

No public versions of CLAM have been yet released, but a first beta release is available [1].

5.1.2. Multi-threading and real time operation

Multi-threading is achieved using POSIX threads. On Linux systems, audio processing can be configured to run as a real time ap-

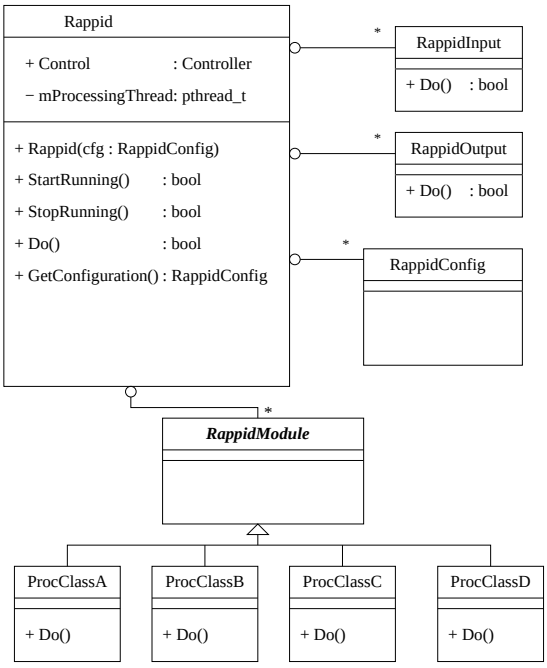

Figure 4: *Rappid* class diagram using dynamic linking

plication. A watchdog mechanism is used in such cases, to avoid system deadlocks due to a malfunctioning of the processing thread.

This approach has show to give really acceptable results. Of course it is not possible to guarantee hard real time constrains in a multitasking environment as Linux.

6. DEVELOPING APPLICATIONS WITH *RAPPID*

The development of an interactive sound processing or synthesis application based on *Rappid* consists of two main tasks: developing the processing modules and developing the user interface.

6.1. Development of processing modules

Processing modules can be seen as application specific plugins. In the current version of Rappid they are not actually plugins, as the whole application needs to be recompiled to add or modify modules, but things will change in future versions.

Most of the issues discussed in this section are common to the development of any kind of CLAM *Processing Class*.

6.1.1. Processing class

Rappid processing modules need to be derived from the `Processing` or `ProcessingComposite` class in the CLAM library. There are a few methods which need to be implemented in classes derived from those ones. The most important ones are:

Do(): This is the execution callback method. It will be called whenever new data frames are available at the module inputs, and it should write the corresponding output data frames during its execution.

ConcreteConfigure(...): This method will be called when any configuration parameter changes, so that the internal state of the module can be updated.

ConcreteStart(...): This method will be called whenever the system is restarted, and it should initialize the internal state of the module accordingly.

6.1.2. Configuration class

A configuration class, derived from the `ProcessingConfig` CLAM class, should also be defined, to be used as argument type for the `ConcreteConfigure` method.

Attributes in this class will be the initial values for processing parameters.

6.1.3. Modifications to Rappid class

In the current version of the framework, it is necessary to perform some modifications in the `Rappid` class, so that it becomes aware of the new processing modules added to it.

6.2. Development of User Interfaces

A graphical application can instantiate a *Rappid* object, configure it and start or stop it, using the public methods which appear in figure 3.

The next steps are very simple, as Rappid really does the hard work. Its `Control` attribute offers methods for changing the parameters values in real time so, the developer only needs to create a GUI and to link visual controls to these methods.

7. EXAMPLE APPLICATIONS

A prototype of a *Rappid* based audio processor was used in a composition by Gabriel Brncic which was performed on the 15 of June in the *Multiphonies 2002* cycle of concerts at the *GRM* in Paris[9]. This particular application was the main motivation for the development of *Rappid* in the first place.

7.1. Processing module

The processing module in this application is able to perform real time morphing between two instruments, a harp and a viola played during the performance.

The morphing mechanism is a time domain envelope cross-modulation designed by Gabriel Brncic for his composition, to achieve a mixture of the dynamics of the input signals.

We have chosen a simple linear interpolation of the amplitude average points as the envelope extraction algorithm. This mechanism allows implementing soft changes in the envelope *gain* in a very efficient way.

Two processing modules are instantiated in the system, so that each of the input signals modulates the other one, and both modulated results can be sent to separate output channels.

7.2. Graphical user interface

We used KDevelop [10], an awarded C/C++ Integrated Development Environment, for the development of the graphical interface. Based on Qt [11], KDevelop has an embedded version of Qt Designer, a very useful visual GUI design tool. Figure 5 shows the resulting interface, controlling the *Rappid* processor.

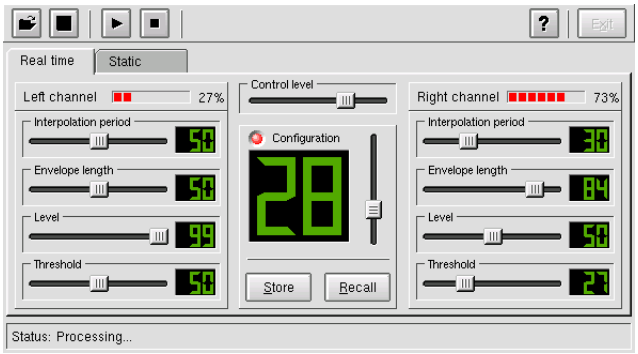

Figure 5: Graphical controller using *Rappid*.

7.3. The concert

We used a desktop PC with an Intel Pentium III running at 800 MHz. The sound card was a RME DIGI96/8 sound card with an ADAT interface working with a sample rate of 48 Khz. It had Debian GNU/Linux (Woody) installed on it, and a recent version of the Linux Kernel (2.4.17), with low latency patches applied to it. We were able to keep the system running for hours with no dropouts.

8. CONCLUSIONS

We have described a first version of a development framework for sound processing applications, and how we have successfully used it to implement an interactive sound processor wich satisfied live performance requirements.

We have thus shown that a low-cost software-only approach for high demand sound processing applications is possible. But we have come across some problems wich still make the development of such applications quite time consuming.

One problem of using this framework for real time applications is the difficulty to predict the worst case execution time of interactive processing algorithms. The lack of time and latency profiling tools makes it hard to find the cause of some sporadic CPU-exhaustion dropouts for non-trivial processing algorithms.

Another drawback of using *Rappid* (or any other software solution) in high demand applications is the dependency on the hardware. Obtaining a low latency robust software processor is conditioned to the availability of a hardware platform which does not prevent the execution of the processing software for "long" intervals, and the availability of good drivers for this hardware. Also, these requirements do not only apply to sound hardware. Any bad device in the processing computer can degrade latency performance.

In order to overcome this difficulty, we had to use modifications to the Linux Kernel for better worst case latency[1], and we had to choose a mature hardware platform (about two years-old components) for which mature Linux drivers exist. We hope that, as more people start to use real-time applications, a wider knowledge base of adequate hardware for real-time applications will available in future.

[1]We used Andrew Morton's low latency patches, available at `http://www.zip.com.au/ akpm/linux/schedlat.html`

9. FURTHER DEVELOPMENTS

Rappid is still work in progress, and although its core is already being used for real time applications, many aspects have to be polished and improved.

We plan to add some important new functionality, such as mechanisms to allow latency analysis, and support for dynamic linking of processing modules from the application.

Finally, the Linux audio development scene is really lively these days. There are several promising frameworks and architectures which have to be closely studied and followed. The *Rappid* project will probably benefit from them, or maybe contribute to them.

10. ACKNOWLEDGMENTS

This work would not have been possible without the contribution of many people. We would specially like to thank Gabriel Brncic for boosting this work, and for his ideas for the sound processing algorithms. We would also like to thank Xavier Amatriain and all the CLAM development team for their great work, and for their support during the development of the framework.

We acknowledge the support of the European Commission, which funds this project through the contract IST-2001-34879; key action IV.3.3 (the AGNULA project). Also, we would like to thank the Spanish Fulbright Commission, which has recently started to fund one of the authors.

11. REFERENCES

[1] CLAM: C++ Library for Audio and Music processing, "http://iua.upf.es/mtg/clam," .

[2] A GNU/Linux Audio distribution. IST-2001-34879; key action IV.3.3 AGNULA, "http://www.agnula.org/," .

[3] ALSA. Advanced Linux Sound Architecture, "http://www.alsa-project.org," .

[4] Karl MacMillan, Michael Droettboom, and Ichiro Fujinaga, "Audio latency measurements of desktop operating systems," *Proceedings of the International Computer Music Conference*, 2001.

[5] Dave Phillips, "Sound & midi software for linux," *http://www.bright.net/˜dlphilp/linuxsound*.

[6] Maarten de Boer, "Tapiir, a software multitap delay," *Conference on Digital Audio Effects, Limerick, Ireland*, December 2001.

[7] Paul Davis et al., "Jack. low-latency audio server," *http://jackit.sourceforge.net/*.

[8] Victor Lazzarini, "Sound processing with the sndobj library: an overview," *Conference on Digital Audio Effects, Limerick, Ireland*, December 2001.

[9] GRM: Les concerts Multiphonies 2002. Live electronics, "http://www.ina.fr/grm/agenda/multiphonies.fr.html," .

[10] "Kdevelop: a c/c++ integrated development environment," *http://www.kdevelop.org*.

[11] Trolltech, "Qt the cross-platform c++ gui toolkit and related tools," *http://www.trolltech.com/products*.