Audio Plugin Development with QtGrpc

A Journey Into Remote GUIs

Dennis Oberst

19 October 2023

Abstract

The Qt project is a robust C++ framework, prominently recognized for enabling the development of cross-platform applications. Within the realm of the audio industry, Qt maintains a significant presence amongst professional entities. However, when examining the domain of audio-plugins, there's a notable absence of Qt-based user interfaces. This document elucidates the reasons and challenges contributing to this absence. Furthermore, it introduces a structured approach to address these challenges. While users often navigate the market in search of plugins that cater to their requirements, developers face constraints in crafting such plugins. The objective of this research is to bridge this disparity, enhancing the resources available to developers, especially in the creation of graphical user interfaces for audio plugins.

Contents

Chapter 1: Introduction

Throughout this work, we will discuss plugins and their impact on the usability of an application. Specifically, we will delve into the development of graphical user interfaces (GUIs) for audio plugins, examining their influence on user experiences and their relationship with development experiences. When considering GUIs[1] broadly, it's natural to contemplate their flexibility and stability. The sheer number of operating systems, graphic backends, and platform-specific details is more than any single developer could realistically address.

Investing time in learning and potentially mastering a skill naturally leads to the desire to apply it across various use-cases. Opting for a library that has *withstood the test of time* enhances stability, yet professionals often seek continuity, preferring not to re-acquaint themselves with a topic merely because API[2] of our chosen toolkit doesn't support the targeted platform.

Therefore, Qt, a cross-platform framework for crafting GUIs, comes to mind when contemplating the development of an audio plugin UI[3] intended for widespread platform compatibility. The expertise gained from utilizing the Qt framework is versatile, suitable for crafting mobile, desktop, or even embedded applications without relearning syntax or structure. As one of Qt's mottos aptly states:

> Code once, deploy everywhere.

The significance of this subject becomes evident when browsing the forum "kvraudio.com", a renowned platform for audio-related discussions.

A brief search of:

> "Qt" "Plugin" :site www.kvraudio.com

uncovers 57'800 results, with 580 from the span between 10/19/2022 and 10/19/2023. While the weight of such figures may be debated, they undeniably highlight the relevance and potential of Qt as a viable choice for audio plugin development.

1.1 Background

1.1.1 Plugins: Shared Libraries

When discussing plugins written in a compiled language, we typically refer to them as shared libraries. A shared library, also known as a dynamic library or DSO[4], is a reusable object that exports a table of symbols (e.g., functions, variables, global data). These libraries are loaded into shared memory once and made accessible to all instances that might utilize them. This approach ensures efficient memory and resource management. Commonly used libraries can greatly benefit from this. However, this efficiency can compromise portability, as these libraries must either be present on the target platform or packaged with the application.

A canonical example is the *standard C library* (libc). Given its presence in nearly every application, the efficiency of shared libraries becomes evident.

[1] Graphical User Interfaces
[2] Application Programming Interface
[3] User Interface
[4] Dynamic Shared Object

To demonstrate, we'll examine the shared object dependencies of some standard applications using the Linux utility **ldd**:

```
ldd /usr/bin/git
  linux-vdso.so.1 (0x00007ffcf7b98000)
  libpcre2-8.so.0 => /usr/lib/libpcre2-8.so.0 (0x00007f5c0f286000)
  libz.so.1 => /usr/lib/libz.so.1 (0x00007f5c0f26c000)
  libc.so.6 => /usr/lib/libc.so.6 (0x00007f5c0ec1e000)

ldd /usr/bin/gcc
  linux-vdso.so.1 (0x00007ffef8dfd000)
  libc.so.6 => /usr/lib/libc.so.6 (0x00007fcf68af9000)
```

Shared libraries can be further categorized into:

- *Dynamically linked libraries* - The library is linked against the application after the compilation. The kernel then loads the library, if not already in shared memory, automatically upon execution.
- *Dynamically loaded libraries* - The application takes full control by loading libraries manually with tools like dlopen or QLibrary.

For audio plugins, the latter method is employed to load plugin instances. The interface defined by a plugin standard can be viewed more as a communication layer, resonating more with a request-response paradigm than the conventional utility functions of linked libraries.

Considering the foundational role of shared libraries in every plugin, it's beneficial to delve deeper into their intricacies. Let's explore a basic example:

```cpp
// simplelib.cpp
#include <format>
#include <iostream>
#include <string_view>
#include <source_location>

#ifdef _WIN32
#   define EXPORT __declspec(dllexport)
#else
#   define EXPORT __attribute__((visibility("default")))
#endif

extern "C" EXPORT void lib_hello() {
    constexpr std::string_view LibName = "simplelib";
    std::cout << std::format(
        "{}: called from {}:{}\n", LibName,
        std::source_location::current().file_name(),
        std::source_location::current().line()
    );
}
```

This code defines a minimal shared library. After including the required standard-headers, we define a

compile time directive that is used to signal the visibillity of the exported symbols. Windows and Unix based system differ here. On Windows with MSVC the symbols are *not* exported by default, and require explicit marking with `__declspec(dllexport)`. On Unix based system we use the visibillity attribute. Since by default all symbols are exported on these platforms, rendering this attribute seemingly redundant, it remains advantageous to maintain clarity. This would also allows us to control the visibillity in the linking step by simply using the linker flag `-fvisibility=hidden` to hide all symbols.

The function `void lib_hello()` is additionally marked with `extern "C"` to provide C linkage, which makes this function also available to clients loading this library from C-code. The function then simply prints the name and the source location of the current file.

Now lets have a look at the host, which is loading the shared library during its runtime. The implementation is Unix specific but would follow similar logic on Windows as well:

```
// simplehost.cpp
#include <cstdlib>
#include <iostream>
#include <dlfcn.h>

int main()
{
    // Load the shared library.
    void* handle = dlopen("./libsimplelib.so", RTLD_LAZY);
    if (!handle)
        return EXIT_FAILURE;
    // Resolve the exported symbol.
    auto *hello = reinterpret_cast<void (*)()>(dlsym(handle, "lib_hello"));
    if (!hello)
        return EXIT_FAILURE;
    // Call the function.
    hello();
    // Unload the shared library.
    dlclose(handle);
    return EXIT_SUCCESS;
}
```

For simplicity reasons the error handling has been kept to a minimum. The code seen above is basically all it takes to *dynamically load libraries*, and is what plugin-hosts are doing to interact with the plugin interface.

To finalize this example, lets write a minimal build script and run our `simplehost` executable.

```
#!/bin/bash
# build_and_run.sh

mkdir -p build
# Compile the shared library. PIC means position independent code and
# is required for shared libraries on Unix systems.
```

```
g++ -shared -o build/libsimplelib.so simplelib.cpp -fPIC -std=c++20 -Wall -Wextra -pedantic

# Compile the host program. The -ldl flag is required to link the
# dynamic loader on Unix systems.
g++ -o build/simplehost simplehost.cpp -ldl -Wall -Wextra -pedantic

# Run the host program. We change to the build directory, because
# the host expects the shared library to be in the same directory.
cd build/ || exit
./simplehost
```

And finally run our script:

```
./build_and_run
simplelib: called from simplelib.cpp:18
```

1.1.2 Plugins: Overview

Plug-*ins*, at their essence, serve as dynamic extensions, enhancing the capabilities of a plugin-loading host. One can perceive them as *on-demand* functionalities ready for deployment. Their prevalence is evident across both the software and hardware domains. For instance, they can be specialized filters added to image processing applications like *Adobe Photoshop*, dynamic driver modules integrated into operating systems such as *GNU/Linux* (Vandevoorde, 2006), or system-specific extensions found in frameworks like *Qt*.

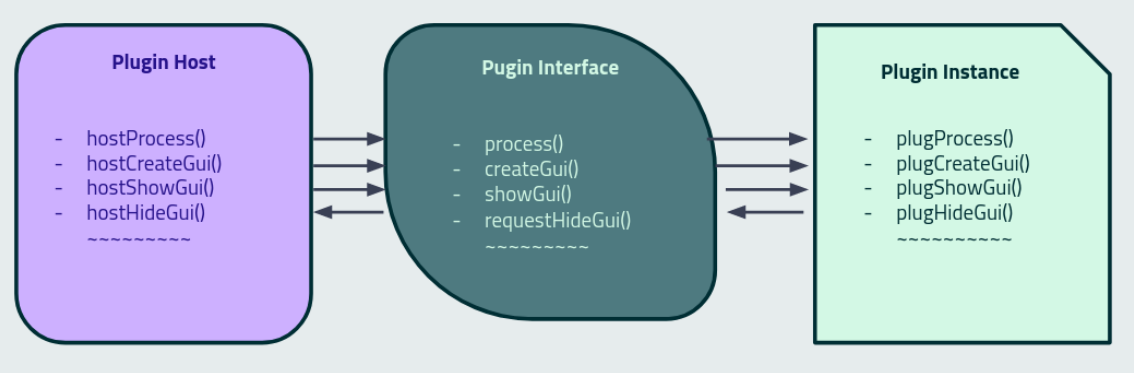

Figure 1: basic plugin architecture

Figure 1 offers a visual representation of this mechanism. On the left, we have the host, which is equipped with the capability to accommodate the plugin interface. Centrally located is the plugin interface itself, serving as the communication bridge between the host and the plugin. The right side showcases the actual implementation of the plugin. Once the shared object is loaded successfully by the host, it actively *requests* the necessary functionalities from the plugin as and when required. This interaction entails invoking specific functions and subsequently taking actions based on the results. While plugins can, on occasion, prompt functions from the host, their primary role is to respond to the host's requests. The foundation of this interaction is the meticulously designed plugin interface that facilitates this bilateral communication.

A concrete example of such an interface can be gleaned from the **CL**ever **A**udio **P**lugin (CLAP) format. This standard dictates the communication protocols between a **D**igital **A**udio **W**orkstation (DAW) and its various plugins, be it synthesizers, audio effects, or other elements. A segment from the C-API reads:

```
   // Call start processing before processing.
   // [audio-thread & active_state & !processing_state]
   bool(CLAP_ABI *start_processing)(const struct clap_plugin *plugin);
```

To unpack this, the function outlines the calling convention for a function pointer named `start_processing`. This pointer returns a `bool` and receives a constant pointer to the struct `clap_plugin` as an argument. Embedded within this is the preprocessor macro CLAP_ABI, defined as:

```
#if !defined(CLAP_ABI)
#   if defined _WIN32 || defined __CYGWIN__
#      define CLAP_ABI __cdecl
#   else
#      define CLAP_ABI
#   endif
#endif
```

For Windows and similar platforms, the attribute `__cdecl` is adopted, ensuring adherence to the C-style function calling convention. On other platforms, this macro is unassigned. This implementation detail only becomes pertinent for developers intent on designing theor own plugin interface.

Throughout our discussions, we'll frequently reference the terms *ABI* and *API*. To ensure clarity, let's demystify these terms. The **API**, an acronym for **A**pplication **P**rogramming **I**nterface, serves as a blueprint that dictates how different software components should interact. Essentially, it acts as an agreement, ensuring that software pieces work harmoniously together. If we think of software as a puzzle, the API helps ensure that the pieces fit together.

On the flip side, we have the **ABI** or **A**pplication **B**inary **I**nterface, which pertains to the actual executable binary. While the API sets the communication norms, the ABI ensures they're executed accurately.

As software undergoes iterative development—evolving from version 1.1 to 1.2, and then to 1.3, and so forth—maintaining a consistent interface becomes paramount. Ideally, software designed using version 1.3 of an interface should seamlessly operate with version 1.1. This seamless integration without necessitating recompilation or additional adjustments is known as ***Binary Compatibility***. It ensures that different versions of software libraries can coexist without conflict. Notably, while binary compatibility emphasizes the consistent presence of binary components, it doesn't imply rigidity in the API.

Beyond this, there's the concept of ***Source Compatibility***. This pertains to preserving the integrity of the API. Achieving this form of compatibility demands meticulous planning and entails upholding a uniform API, precluding drastic alterations like function renaming or behavioral modifications.

1.1.3 Audio Plugins: Structure and Realtime

In exploring the realm of audio plugins, two primary components emerge in their structure: a realtime audio section and a controlling section.

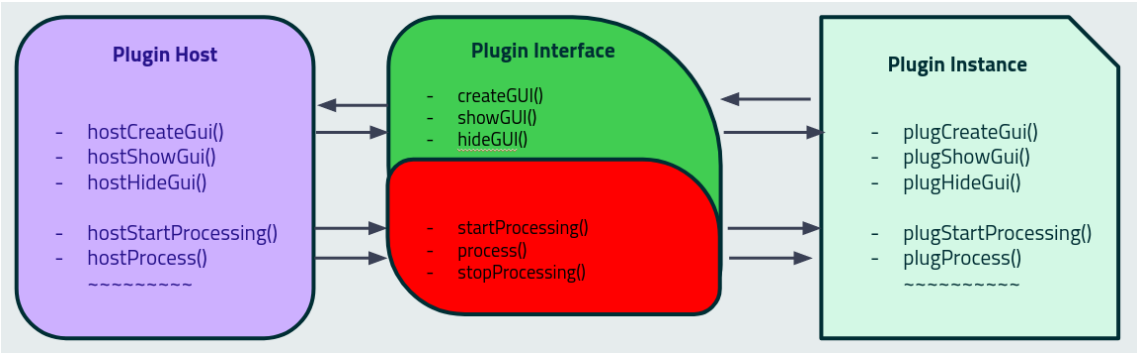

Figure 2: basic audio plugin architecture

Figure 2 illustrates a structure closely resembling *Figure 1*. The plugin interface now features two distinct colors to aid differentiation. The green section pertains to the host's interaction with the plugin through a low-priority main thread. This primarily handles tasks such as GUI setup and other control-related functions. "Low priority" implies that it's permissible to conduct non-deterministic operations which may momentarily halt progress.

The red segment signifies that the API is being invoked by a high-priority realtime thread. These functions are called frequently and demand minimal latency to promptly respond to the host. They process all incoming events from the host, including parameter changes and other vital events essential for audio processing. When processing audio, the plugin integrates all necessary alterations and executes the specified function, as demonstrated by a gain plugin; A typical gain modifies the output volume and possesses a single parameter: the desired gain in decibels to amplify or diminish the output audio.

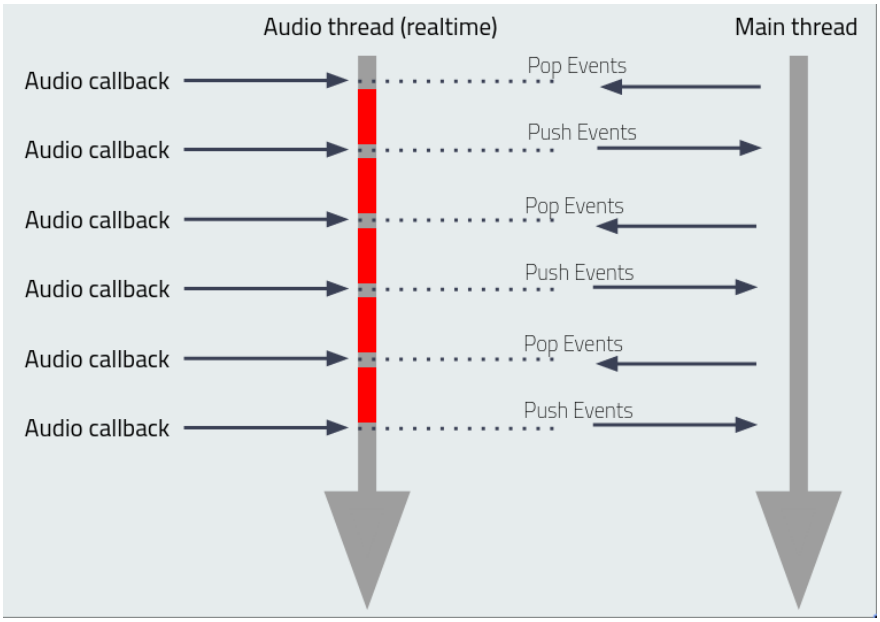

Figure 3: realtime overview(Doumler, 2023)

A challenge in audio programming stems from the rapidity with which the audio callback needs a response. The standard structure unfolds as:

1. The host routinely invokes a process function, forwarding the audio input buffer as an argument, along with any events related to the audio samples (such as a user adjusting a dial within the host).
2. The plugin must swiftly act on this buffer and relay the modified audio buffer back to the host. Given that a GUI is often integrated, synchronization with the audio engine's events is vital. Once all audio samples and events are processed, these changes are communicated back to the GUI.

Failure to meet the callback's deadline, perhaps due to extended previous processing, can lead to audio disruptions and glitches. Such inconsistencies are detrimental to professional audio and must be diligently avoided. Hence, a known saying is:

> Don't miss your Deadline!

$$\frac{AudioBufferSize}{SamplingFrequency} = CallbackFrequency$$

$$\frac{512}{48000Hz} = 10.67ms$$

Figure 4: realtime callback frequency

Figure 4 presents the formula for determining the minimum frequency at which the processing function must supply audio samples to prevent glitches and drop-outs. For 512 individual sampling points, at a sample rate of 48,000 samples per second, the callback frequency stands at **10.67ms**. Audio's block-size generally fluctuates between 128 - 2048 samples, with sampling rates ranging from 48,000 - 192,000 Hz. Consequently, our callback frequency must operate within approximately **2.9ms - 46.4ms** for optimal functionality.

However, this is just the tip of the iceberg. The real challenge lies in crafting algorithms and data structures that adhere to these specifications. Drawing a comparison between the constraints of a realtime thread and a standard thread without these stipulations brings this into sharper focus: <In a realtime thread, responses must meet specific deadlines, ensuring immediate and predictable behavior. In contrast, a standard thread has more flexibility in its operation, allowing for variable response times without the stringent need for timely execution.>

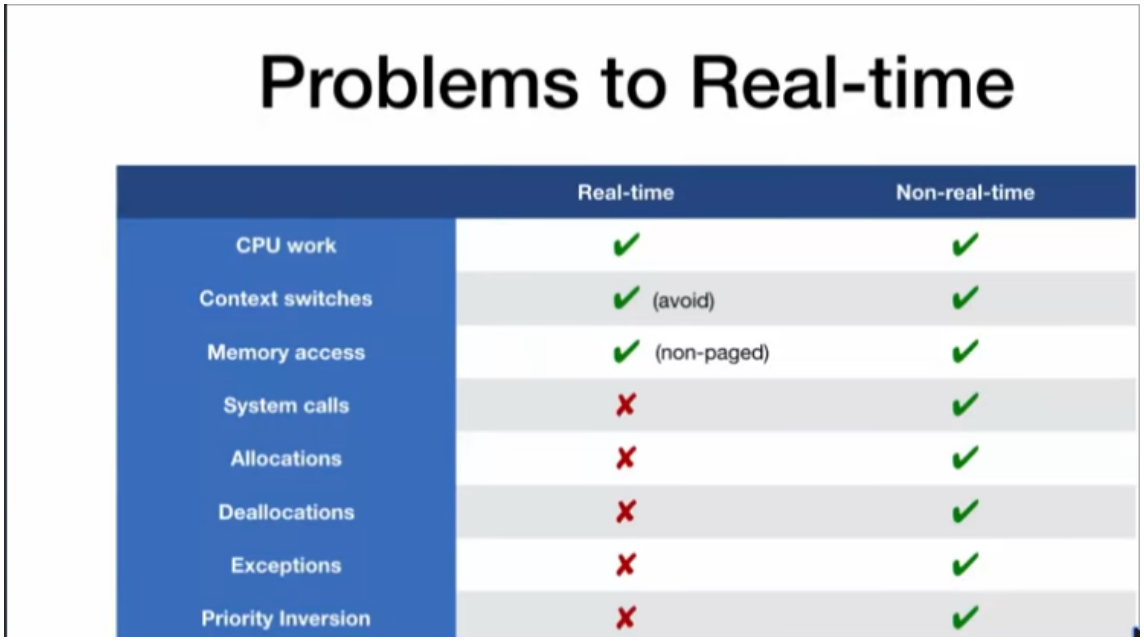

Figure 5: realtime limitations(Fabian Renn Giles, 2020)

Figure 5 compares realtime to non-realtime requirements. In short we can't use anything that has non-deterministic behavior. We want to know exactly how long a specific instruction takes to create an overall structure that provides a deterministic behavior. *System calls* or calls into the operating system kernel, are one of such non-deterministic behaviors. They also include allocations and deallocations. None of those mechanisms can be used when we deal with realtime requirements. This results in careful design decisions that have to be taken when designing such systems. We have to foresee many aspects of the architecture and use pre-allocated containers and structures to prevent a non-deterministic behavior. For example to communicate with the main thread of we use non-blocking and wait free data structures as FIFO's[5] or ring-buffers to complement this.

[5] First In First Out

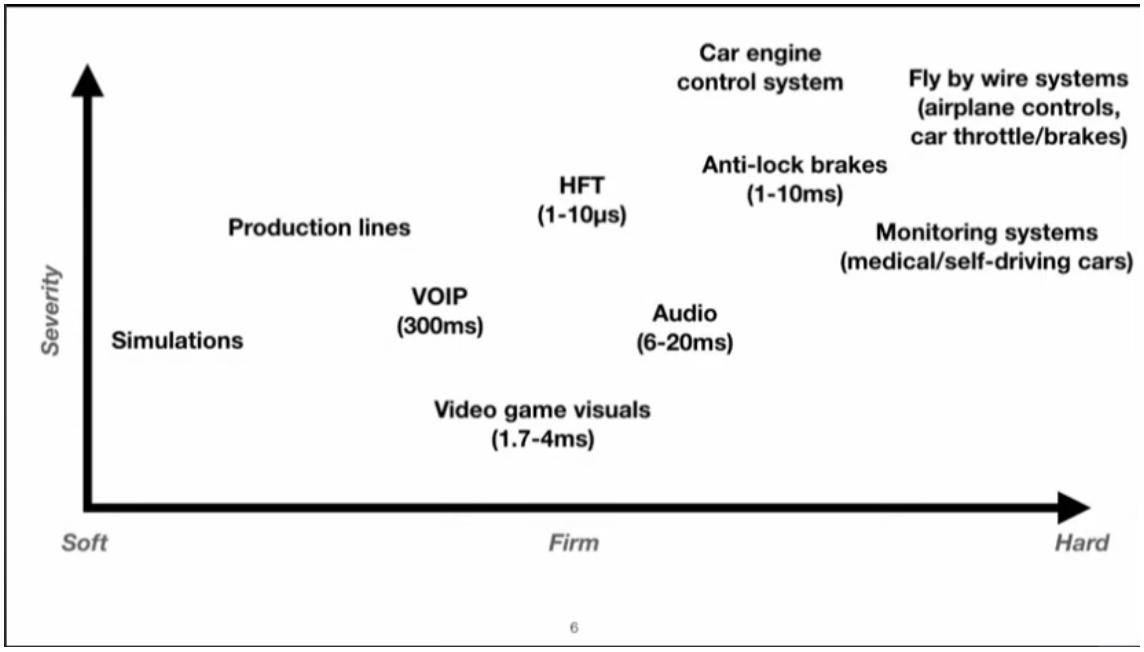

Figure 6: realtime ranking(Fabian Renn Giles, 2020)

Figure 6 classifies various realtime systems by their severity (Y-axis) and the impact on overall design (X-axis). Realtime can be partitioned into three categories: Soft-Realtime, Firm-Realtime, and Hard-Realtime. High-severity lapses might have catastrophic consequences. For instance, in medical monitoring systems or car braking systems, missing a deadline can be fatal, leading to a literal **dead**line. Audio sits in the middle, where a missed deadline compromises professional utility but doesn't pose dire threats. Video game rendering bears even lesser severity, as occasional frame drops don't render the product ineffectual and are relatively commonplace.

1.1.4 Audio Plugins: Standards and Hosts

Over time, various audio plugin standards have evolved, but only a select few remain significant today. The table below provides an overview of some of the most well-recognized standards:

Standard	Extended Name	Developer	File Extension	Supported OS	Initial Release	Licensing
CLAP	Clever Audio Plugin	Bitwig & U-he	.clap	Windows, MacOS & Linux	2022	MIT
VST/VST3	Virtual Studio Technology	Steinberg	.dll, .vst, .vst3	Windows, MacOS & Linux	2017	GPLv3, Steinberg Dual License (Email)
AAX	Avid Audio Extension	Pro Tools (Avid)	.aax	Windows & MacOS	2011	Approved Partner (Email)

Standard	Extended Name	Developer	File Extension	Supported OS	Initial Release	Licensing
AU	Audio Units	Apple macOS & iOS	.AU	MacOS	"Cheetah" 2001	Custom License Agreement

Certain standards cater specifically to particular platforms or programs. For instance, Apple's **AU** is seamlessly integrated with their core audio SDK[6] (Apple, 2023). Similarly, Avid's **AAX** is designed exclusively for plugin compatibility with the Pro Tools DAW[7]. On the other hand, standards like the **VST3** SDK are both platform and program independent, and it's currently among the most popular plugin standards. Additionally, the newly introduced **CLAP** standard is also gaining traction.

Most commonly, these plugins are hosted within **Digital Audio Workstations (DAWs)**. These software applications facilitate tasks such as music production, podcast recording, and creating custom game sound designs. Their applicability spans a broad spectrum, and numerous DAW manufacturers exist:

Figure 7: plugin hosts

However, DAWs are not the only plugin hosts. Often, plugin developers include a *standalone* version that operates independently of other software. A recent noteworthy development in this domain is the game industry's move towards these plugins, exemplified by Unreal Engine's forthcoming support for the **CLAP** standard, as unveiled at Unreal Fest 2022.

1.2 Problem Statement

A primary challenge in integrating Qt user interfaces within restricted audio plugin environments hinges on reentrancy. A program or function is considered re-entrant if it can safely support concurrent invocations, meaning it can be "re-entered" within the same process. Such behavior is vital as audio plugins often instantiate multiple times from the loaded shared libraries. To illustrate this challenge, consider the following example:

```
// reentrancy.cpp
#include <iostream>
```

[6] Software Development Kit
[7] Digital Audio Workstation

```cpp
void nonReentrantFunction() {
    std::cout << "Non-Reentrant: ";
    for (static int i = 0; i < 3; ++i)
        std::cout << i << " ";
    std::cout << std::endl;
}

void reentrantFunction() {
    std::cout << "Reentrant: ";
    for (int i = 0; i < 3; ++i)
        std::cout << i << " ";
    std::cout << std::endl;
}

int main() {
    std::cout << "First call:" << std::endl;
    nonReentrantFunction();
    reentrantFunction();

    std::cout << "\nSecond call:" << std::endl;
    nonReentrantFunction();
    reentrantFunction();

    return 0;
}
```

In C++, static objects and variables adhere to the static storage duration:

> The storage for the object is allocated when the program starts and deallocated when it concludes. Only a single instance of the object exists.

The crux here is the singular instance of the object per application process. This necessitates caution when working with static types where reentrancy is essential. When the above program is executed, the outcome is:

```
# Compile and run the program
g++ -o build/reentrancy reentrancy.cpp -Wall -Wextra -pedantic-errors \
;./build/reentrancy

First call:
Non-Reentrant: 0 1 2
Reentrant: 0 1 2

Second call:
Non-Reentrant:
Reentrant: 0 1 2
```

While the initial invocation for both functions is successful, re-entering the function during the second call yields no output for the non-reentrant function. This outcome stems from the use of the static specifier in the for loop counter, causing the variable to not meet the `i < 3` condition during the second entry. This example serves to emphasize the interplay between *static storage duration* and function reentrancy.

While static objects offer global accessibility and potentially enhance application design, they also present certain challenges. For example, the QApplication variants, such as QCoreApplication and QGuiApplication, which manage the Qt event loop through `Q*Application::exec()`, are static:

```
// qtbase/src/corelib/kernel/qcoreapplication.h
static QCoreApplication *instance() noexcept { return self; }
```

This design choice means only one QApplication can exist within a process. Issues arise when a plugin-loading-host, as QTractor does, utilizes a QApplication object or when multiple plugin instances operate within a singular process. At first glance, one might assume the ability to verify the presence of a QApplication within the process and then conveniently reuse its event loop:

```
~~~
   if (!qGuiApp) {
       static int argc = 1; static char *argv[] = { const_cast<char*>("") };
       new QGuiApplication(argc, argv);
   }
~~~
```

However, while this approach occasionally proves successful, it's fraught with uncertainties. For instance, what if we inadvertently latch onto an event loop from an outdated version? And how does the system handle multiple instances simultaneously connecting to the event loop? Evidently, this method offers an *unreliable* remedy for addressing the issue at hand.

Furthermore, event loop's blocking nature means that `Q*Application::exec()` only returns post-execution. Yet, the plugin standards in discussion necessitate return capabilities.

Although there are various workarounds, like compiling Qt under a separate namespace and initiating the event loop in an isolated thread, these often fall short. Such methods either introduce undue complexity, as seen with namespace compilation for their users, or induce instability, as with separate thread initiation.

1.3 Objectives

This study endeavors to pave the way for innovative techniques that allow seamless integration of graphical user interfaces developed with the Qt framework into an audio plugin standard. Central to this goal is enabling the initiation of these GUIs directly from a host, while ensuring efficient and responsive communication between the two entities (host <-> GUI). The overall user experience is pivotal; hence, the development process of these Qt GUIs should not only feel native but also be intuitive. This implies that events triggered by the host should effortlessly weave into Qt's event system. Additionally, these events should be designed to offer signal & slot mechanisms, ensuring they are easily available for utilization within various UI components, streamlining development and promoting a more organic interaction between components.

1.4 Scope and Limitations

The cornerstone of this investigation is the **CLAP** plugin standard. Recognized for its innovative capabilities and being at the forefront of current technologies, it stands as the most promising contender for such integration tasks. The primary ambition of this work revolves around bridging the gap between the two distinct realms of audio plugins and Qt GUIs. However, it's imperative to note that this research doesn't aim to deliver a universal, cross-platform solution. The development focuses on Linux, ensuring compatibility and support for both X11 and Wayland protocols. By confining the research to this scope, the study seeks to delve deeper into the intricacies and nuances of the integration process, ensuring a robust and effective methodology.

Chapter 2: The Qt Framework

Introduction

Qt stands as a widely-recognized framework designed to facilitate the creation of cross-platform graphical user interfaces (GUIs). Its distinctiveness lies in its ability to function seamlessly across diverse platforms such as Linux, Windows, macOS, Android, and even embedded systems. A remarkable feature of Qt is that it achieves this cross-functionality with minimal modifications to its core codebase.

Emerging in 1992 from the innovation labs of Trolltech, a Norwegian software enterprise, Qt's journey has seen several pivotal moments. A notable one was its acquisition by Nokia in 2008. Today, the baton of its active maintenance and further development is in the hands of 'The Qt Company'.

Delving into its licensing, Qt presents two clear routes. The first aligns with the open-source ethos through the GNU (L)GPL license, underscoring a community-driven spirit. This path demands adherence to terms embedded within the license. On the other hand, for developers with an inclination towards building proprietary or commercial software without open-sourcing constraints, Qt offers a commercial license.

While its foundation is deeply rooted in C++, Qt showcases adaptability by providing language bindings, exemplified by Qt for Python / PySide6. This augmentation not only speeds up the development process but also capitalizes on the robustness of utilizing the power of the already present C++ modules by reusing them.

Within the vast expanse of the Qt universe, developers are presented with two primary paradigms for GUI crafting: Widgets and QML. Widgets offer the traditional approach in creating UI elements, making it the go-to for many classical desktop applications Their imprint is evident in widely-adopted applications like Telegram and Google Earth.

Conversely, **QML** (Qt Modeling Language) represents a contemporary, declarative approach to UI design. It employs a clear, JSON-like syntax, while utilizing inline JavaScript for imperative operations. Central to its design philosophy is dynamic object interconnection, leaning heavily on property bindings. One of its notable strengths is the seamless integration with C++, ensuring a clean separation between application logic and view, without any significant performance trade-offs.

Above the foundational QML module resides **QtQuick**, the de facto standard library for crafting QML applications. While Qt QML lays down the essential groundwork by offering the QML and JavaScript engines, overseeing the core mechanics, QtQuick comes equipped with fundamental types imperative for spawning user interfaces in QML. This library furnishes the visual canvas and encompasses a suite of types tailored for creating and animating visual entities.

A significant distinction between Widgets and QML lies in their rendering methodologies. Widgets, relying on software rendering, primarily lean on the CPU for graphical undertakings. This sometimes prevents them from harnessing the full graphical capabilities of a device. QML, however, pivots this paradigm by capitalizing on the hardware GPU, ensuring a more vibrant and efficient rendering experience. Its declarative nature streamlines design interpretation and animation implementation, ultimately enhancing the development velocity.

Qt6, the most recent landmark release in the Qt series, heralded a plethora of advancements. A standout among these is the QRHI (Qt Rendering Hardware Interface). Functioning subtly in the background,

QRHI adeptly handles the complexities associated with graphic hardware. Its primary mission? To guarantee unwavering performance consistency across a diverse range of graphic backends. The introduction of QRHI underscores Qt's steadfast dedication to fortifying its robust cross-platform capabilities, aiming to create a unified experience across various graphic backends.

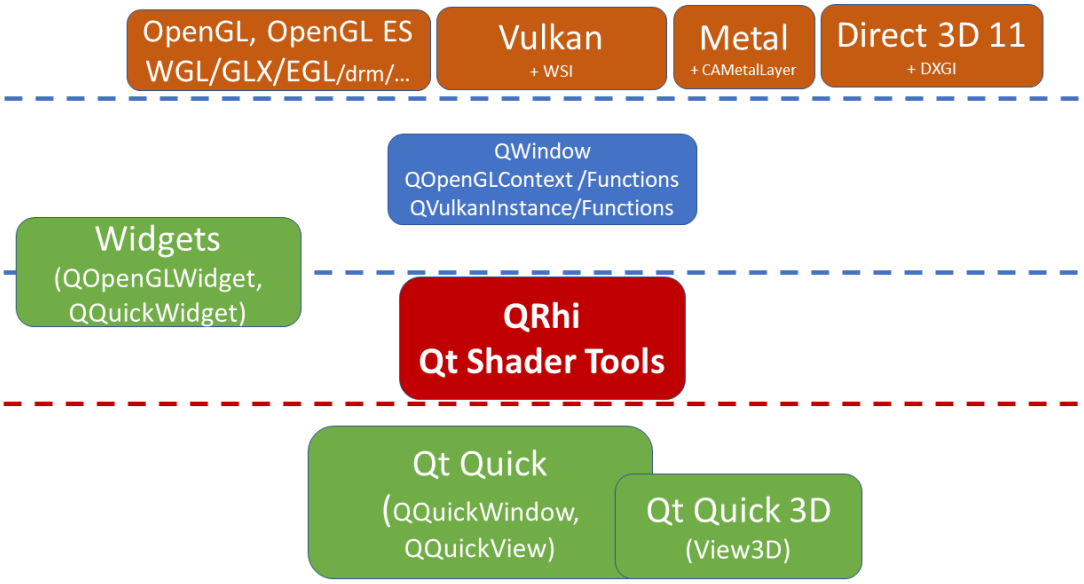

Figure 8: Qt Rendering Hardware Interface RHI

Figure 8 delineates the architecture of the prevailing rendering interface. At the foundational layer, we find the native graphic APIs, encompassing the likes of OpenGL, Vulkan, Metal, and Direct 3D 11. Positioned just above is the QWindows implementation, which is housed within the QtGui module of Qt. Notably, QtWidgets occupies a niche between these two levels. Given that Widgets emerged before the QRhi module, their integration with QRhi isn't as profound. While certain widgets do offer OpenGL capabilities, their primary reliance is on the QPainter API. Ascending to the subsequent tier, we are greeted by the Qt Rendering Hardware Interface, which serves as a crucial bridge, offering an abstraction layer over hardware-accelerated graphics APIs.

> **Note:** As of this writing, QRHI maintains a limited compatibility assurance, implying potential shifts in its API. However, such changes are anticipated to be minimal.

To integrate QRHI within personal projects, one needs to link against `Qt::GuiPrivate` followed by an inclusion directive, `#include <rhi/qrhi.h>`. Navigating higher up the abstraction ladder, we encounter the QtQuick and QtQuick3D modules, standing as testament to Qt's evolutionary journey in graphics rendering.

2.2 QtGrpc

QtGrpc is a relatively new module that is, at the time of writing at version Qt 6.6, currently in technical preview. It builds upon gRPC, a cross-platform high performance **R**emote **P**rocedural **C**all (RPC) framework, that generates client/server bindings for many supported languages:

Figure 9: gRPC supported languages

- Explain briefly, protobuf API, Server, Clients, API, streams and calls.
- Performance of native grpc.

2.3 Solving the Problem (Qt-Side)

- Approaches for integration
  - Picking up the event loop (only windows and mac) because of glib and event loop
    - unstable, launching in seperate thread.
  - Remote GUIs, launching executable
  - Using QtGrpc as a foundation
- Goal from Qt side (cross platform, easy development, integration into event loop).

Chapter 3: The Clap Audio Plugin Format

3.1 Introduction to Clap

- Explain the purpose and significance of the Clap audio plugin format.
- Discuss its role in the audio processing industry.

3.2 Clap Plugin Structure

- Describe the structure and organization of Clap audio plugins.
- Explain the required components and their functionalities.
- Events, Realtime

3.3 Clap API and Specifications

- Explore the Clap API and its usage in developing audio plugins.
- Discuss the specifications and guidelines for creating Clap-compatible plugins.

Chapter 3.5: Realtime, Events, Data Structures, Communications

Chapter 4: Integrating Qt with Clap

4.1 Design Considerations

- Discuss the design principles and considerations for integrating Qt with Clap.
- Address any challenges or conflicts that may arise during the integration process.

4.2 Architecture and Implementation

- clap-remote
- Propose an integration architecture that leverages the strengths of both Qt and Clap.
- Detail the implementation steps and techniques involved in the integration.

Chapter 5: Experimental Results and Evaluation

5.1 Test Setup and Methodology

- Explain the experimental setup used for evaluating the integrated solution.
- Describe the methodology employed to measure the performance and effectiveness.

5.2 Results and Analysis

- Mention all problems: GLib event loop / no auto attach on linux
- Present the empirical results obtained from the experiments.
- Analyze and interpret the results in relation to the integration objectives.

Chapter 6: Discussion and Conclusion

6.1 Summary of Findings

- Summarize the key findings and outcomes of the research.

6.2 Discussion of Results

- Discuss the implications and significance of the results obtained.
- Compare the integrated solution with existing approaches and discuss its advantages.

6.3 Contributions and Future Work

- Highlight the contributions of the thesis to the field of audio processing and software development.
- Identify potential areas for future research and improvement in the integration process.

Bibliography

Apple. (2023, October 23). Audio unit v3 plug-ins. Apple. retrieved 23.10.2023. Available at: https://developer.apple.com/documentation/audiotoolbox/audio_unit_properties

Doumler, T. (2023, August 18). What is low latency? CppNow. retrieved 23.10.2023. Available at: https://www.youtube.com/watch?v=5ulsadq-nyk

Fabian Renn Giles, D. R. &. (2020, April 30). Real time 101. Meeting Cpp. retrieved 23.10.2023. Available at: https://www.youtube.com/watch?v=ndeN983j_GQ&t=1370s

Vandevoorde, D. (2006, September 7). Plugins in c++. Wikibooks. retrieved 23.7.2023. Available at: https://www.open-std.org/JTC1/sc22/wg21/docs/papers/2006/n2074.pdf