Audio Plugin Development with QtGrpc

A Journey Into Remote GUIs

Dennis Oberst

19 October 2023

Abstract

The Qt project is a powerful and versatile C++ framework widely used for developing cross-platform applications. Additionally, Qt is no stranger to the audio-industry, with a strong presence within professional audio companies. If we look into the expanding world of audio-plugins however, we would steer into the void when looking for Qt user-interfaces. With this work I will explain the reasons and problems that led to this state and show a new and innovative approach in solving the problems. A user should be able to find plugins that fits perfectly their needs to customize their environment. However developers often don't have that freedom when it comes to developing such plugins. The goal is to bridge the gap and extend the toolset developers have in developing audio plugins and their graphical user interface specifically.

Special thanks to:

- Dr. Cristián Maureira-Fredes (Qt) for making all of this possible and his …
- Alexandre Bique (Bitwig & CLAP) for listening to my idea and providing deep insights into CLAP and extended techniques for communication

Contents

Chapter 1: Introduction

Throughout this work we will talk about plugins and how they affect the usabillity of an application. We will specifically explore the development of graphical user interfaces for audio plugins. We will inspect how it affects user experiences and explore their correlations with development experiences. When thinking about GUIs[1] in general it comes natural to spend some thoughts about their flexibillity and stability. There are more operating systems, graphic-backends, and platform specific details out there, than a single developer could ever handle to implement.

When we spent time learning and potentially mastering a skill, we often want it to be applicable to a variety of use-cases. Choosing a library that *has stood the test of time* would complement the stability, but we often don't want to re-learn a topic just because the API[2] of our skill set has changed.

Thus, it comes natural to think about Qt, a cross-platform framework for creating graphical user interfaces (GUIs) when considering the implementation of an audio plugin UI[3], that should be available on all the major platforms. The skill set we acquire in using the Qt framework is versatile and can be used to develop mobile, desktop or even embedded applications without the need to re-learn syntax or logic. One of the slogans of Qt applies here:

> Code once, deploy everywhere.

We can see the demand on this topic by simply searching for references on the forum "kvraudio.com", which is a well-known website to discuss audio related topics.

A short query of:

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

reveals 57.800 entries found. From which 580 are in the timeframe of the past year between 10/19/2022 - 10/19/2023. Although the meaningfullness of such numbers is questionable, it still shows the relevance and need of seeing Qt as an option for developing audio plugins.

1.1 Background

1.1.1 Plugins: Shared Libraries

When we talk about plugins written in a compiled language, we most often refer to them as shared libraries. A shared library (also known as dynamic library or DSO[4]) are reusable objects that export a table of symbols (functions, variables, global data etc.). These libraries are then loaded into shared memory once, and made available to all the instances that potentially use them. This technique allows for efficient memory and resources management. Frequently used libraries can benefit from that. On the other hand it affects portabillity since those libraries need to be present on the target platform or have to be shipped together with the application.

A cannonical example would be the *standard C library* (libc), which we can find in almost every application, hence the effectiveness of using shared libraries can be fully explored.

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

Here are some examples where we explore some common applications with the tool **ldd**, which is a standard linux utillity to print shared object dependencies:

```
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.

In the case of audio plugins, the latter technique will be used to load plugin instances. The interface, that a plugin standard defines can hence be seen as a communication layer, more in the sence of a *request - response* mechanism then the traditional utillity functionality of linked libraries.

Since shared libraries are at the foundation of every plugin, it comes beneficial to explore them a bit deeper by going through a minimal 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 platform this attribute could be seen as superfluous, it is still nice to be explicit here. 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* in their most basic form extend the functionality of a plugin-loading-host dynamically. We could think of them as functionality that is made available *on-demand*. They are used all around the software and hardware world and can be found in a multitude of areas. Be it the extension of specialized filters for image processing applications like *Adobe Photoshop*, dynamically loadable drivers for operating systems like *GNU/Linux* (Vandevoorde, 2006) or operating system dependent features as used in the Qt framework.

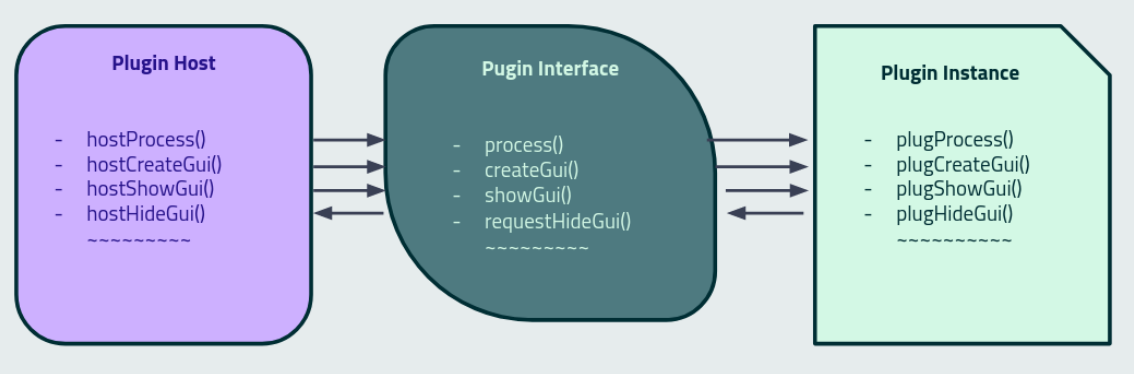


Figure 1: basic plugin architecture

Figure 1 describes this process. On the left side is the host that has implemented the hosting functionality of the plugin interface. In the middle is the plugin interface that serves as a means of communication between the two instances and to the right is the plugin implementation. After successfully loading a shared object, the host then *requests* the required functionality from the plugin on demand. It calls specific functions and acts upon the results it gets. The plugin often also has access to request some functionality from the host, although most often the plugin just reacts to requests from the host. The communication follows the requirements of the well defined plugin interface that sits in-between those two instances.

An example interface can be seen in the **CL**ever **A**udio **P**lugin format, which defines a standard for communication between a **D**igital **A**udio **W**orkstation and plugins (synthesizers, audio effects, ...) to work together. Here is a function of the C-API:

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

Lets deconstruct this function to get a deeper understanding. We define the calling convention for a function pointer `start_processing`, which returns a `bool` and accepts a const pointer to the struct `clap_plugin`. Additionally there is the preprocessor macro CLAP_ABI, which has following definition:

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

So on Windows like platform we use the attribute `__cdecl`, which enforces function calling convention to C-style. On all other platforms this macro is empty and does nothing. This is an implementation detail and not important to know unless you want to build your own plugin interface.

Over the time we will encounter the terms *ABI* and *API* quite regularly, so let us resolve those. API, or **A**pplication **P**rogramming **I**nterface is a set of rules and protocols that allow different software components to communicate with each other. It serves as a contract between the software components, allowing developers to create applications that interact with these components. It can be thought of as the *front-end* to the ABI, the **A**pplication **B**inary **I**nterface, which is the actual binary that is being executed by the callee. Since software evolves sequentially, that is, version 1.1 is followed by version 1.2, is followed by version 1.3 and so on, providing a stable interface is crucial for the development and growth of the interface. We often want one instance, that uses version 1.3 of the interface to be compatible with version 1.1, without the need of recompilation or extra burdens for the clients, who already implemented existing API. This is called ***Binary Compatibillity***, it defines that a new version of a library is compatible with an older version without causing issues for applications that depend on it. Since binary compatibillity is, as the name suggest, on the binary level, this doesn't necessarily mean that the public API, where clients interact with can't be changed. It simply means that the binary parts of the existing API *have* to be present in the compiled binary. If we additionally want the API to be stable we speak of ***Source Compatibillity***. Achieving this requires careful design decisions and involves maintaining a consistent API without breaking changes, such as renaming functions or altering their behavior.

1.1.3 Audio Plugins: Structure and Realtime

If we talk about audio plugins specifically and look at their structure, we will see that their interface consists of two seperated parts. A realtime audio sections and a controlling section.

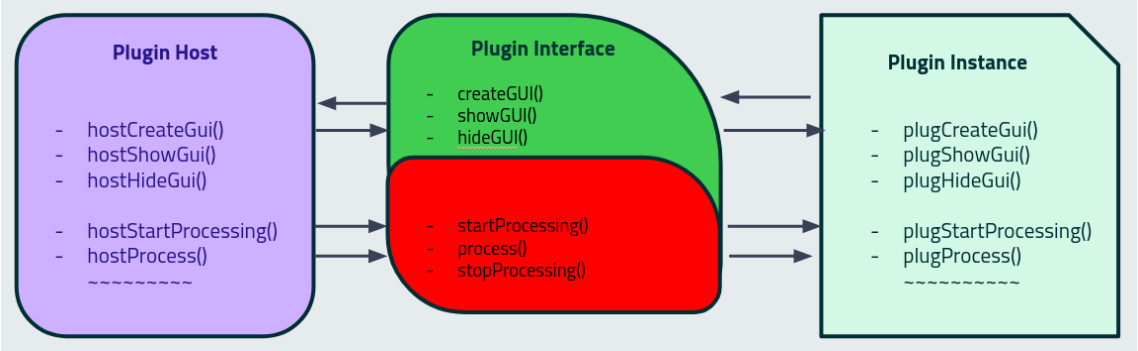


Figure 2: basic audio plugin architecture

Figure 2 has a similar structure to the one seen in *Figure 1*, however the plugin interface is now seperated into two colors. The green color means that the host will interact with the API from a low priority main thread. It is used to control the plugin behavior as to initiate the creation of a GUI or to request other controlling behavior. Low priority here means that it is safe to perform operations that are non-deterministic and can potentially block any further progress for an unknown amount of time.

The red color here means that the API is beeing called from a high priority realtime thread. Those functions are frequently called and require a low latency to quickly respond to the host. They process all incoming events from the host, as the change of parameters and various other important events that are required to further process the audio. When processing the audio, the plugin will take all changes that it requires and perform the operation it promises. One example would be a gain plugin. A gain typically changes the output volume and has one parameter: the gain in decibels that we want to increase or reduce the output audio.

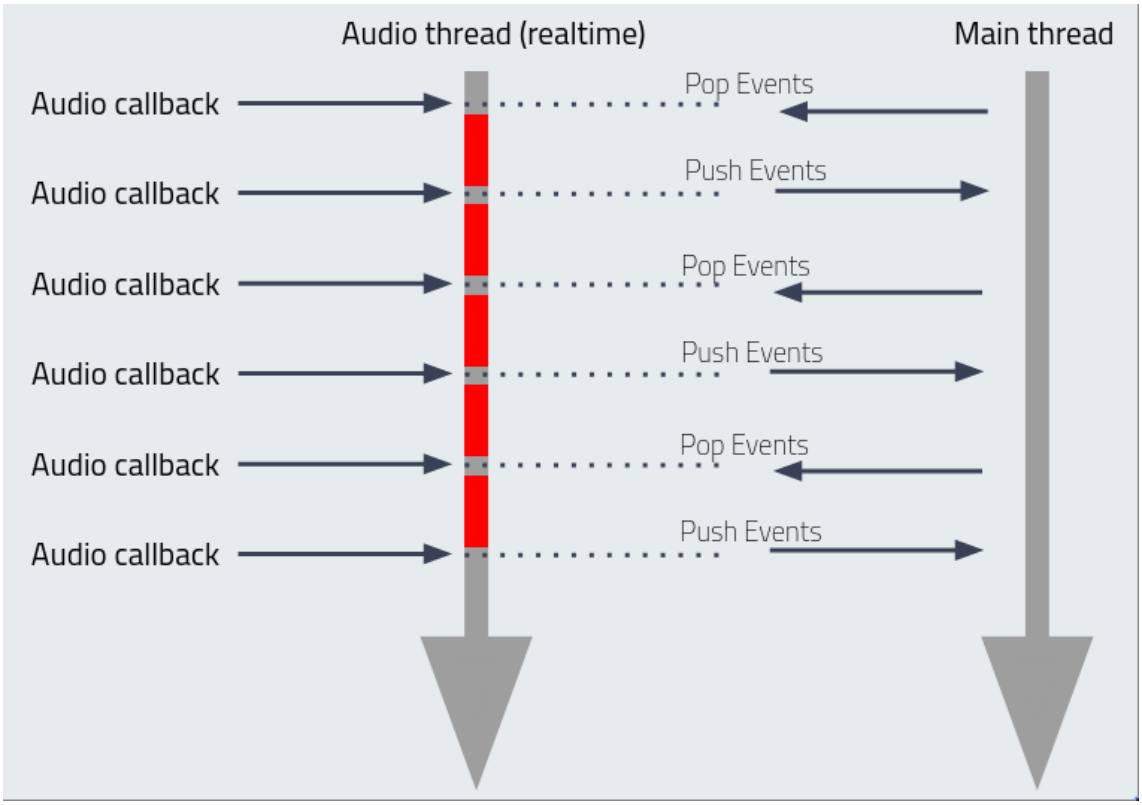


Figure 3: realtime overview

What makes audio programming so difficult at some point is the frequency in what we have to respond to the audio callback. The general structure is as followed:

1. The host frequently calls a process function and passes the audio input buffer as an argument, as well as all the events that occurred relative to the audio samples (as e.g. a user rotating a dial inside the host).
2. The plugin then has to perform its functionality on this buffer as fast as possible and return the updated audio buffer to the host. Since we most often include a GUI we also have to synchronize all processing with the events from the audio engine. After finishing our processing of all audio samples and events we push those changes to our GUI again.

If we miss a deadline of the callback however, because our previous processing took too long, this can result in audio drop outs and glitches. This is an unacceptable behavior for professional audio and should be avoided at any cost. So the saying is:

> Don't miss your Deadline!

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

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

Figure 4: realtime callback frequency

Figure 4 shows the formula to calculate the minimum frequency in what the processing function has to serve audio samples, in order to avoid glitches and drop-outs. So for 512 individual sampling points, with a sampling frequency of 48'000 samples per second, we result in callback frequency of **10.67ms**. The block-size of the audio typically ranges between 128 - 2048 samples, and the sampling frequency is usually between 48'000 - 192'000 Hz. That means that our callback frequency has to work with a range of roughly **2.9ms - 46.4ms** in order to function properly.

Now this is only the tip of the iceberg. The tricky part lays in writing algorithms and data structures that are compatible with those requirements. Lets compare the restrictions of a realtime thread and a normal thread that doesn't have those requirements:

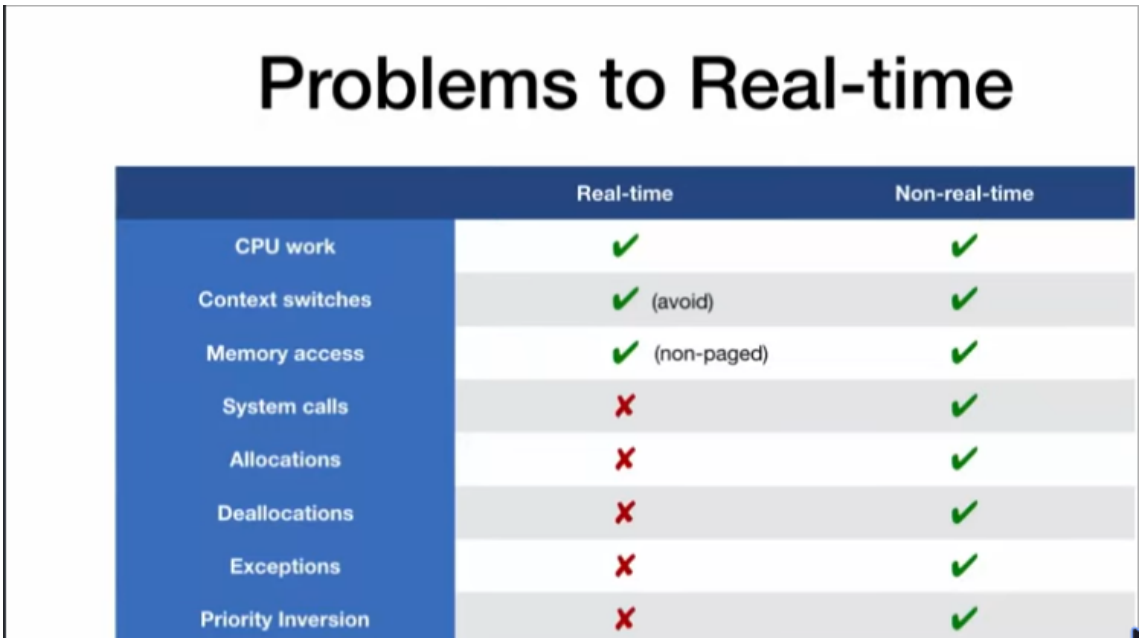


Figure 5: realtime limitations

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 the application we use non-blocking and wait free data structures as FIFO's[5] or ring-buffers to complement this.

Realtime requirements can then be ranked into different aspects again:

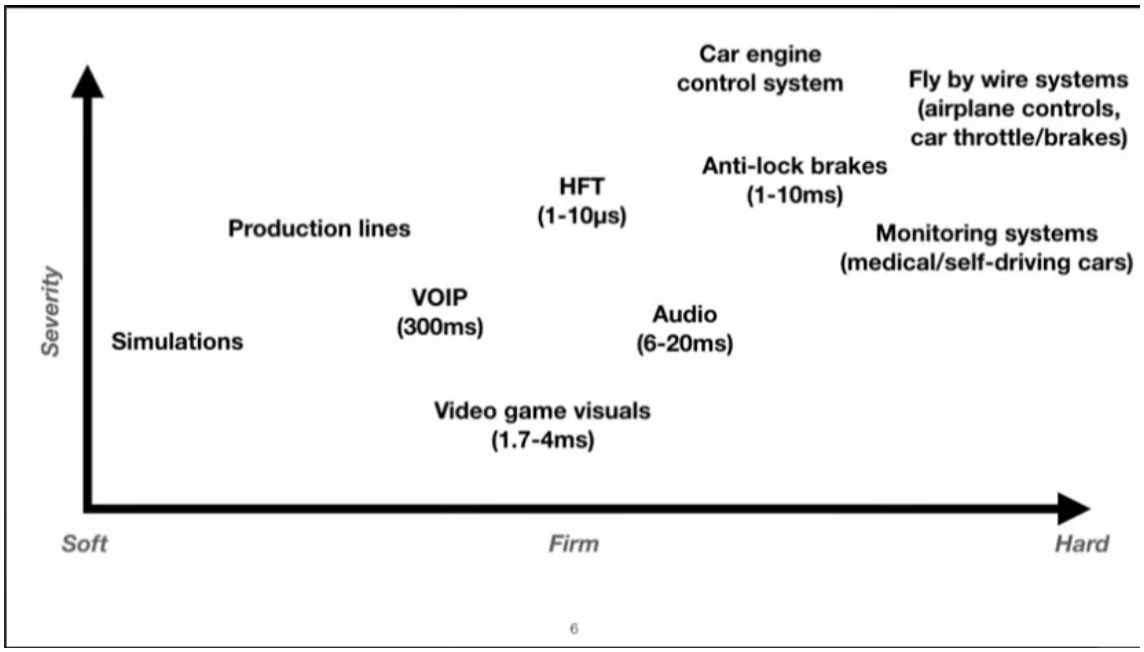


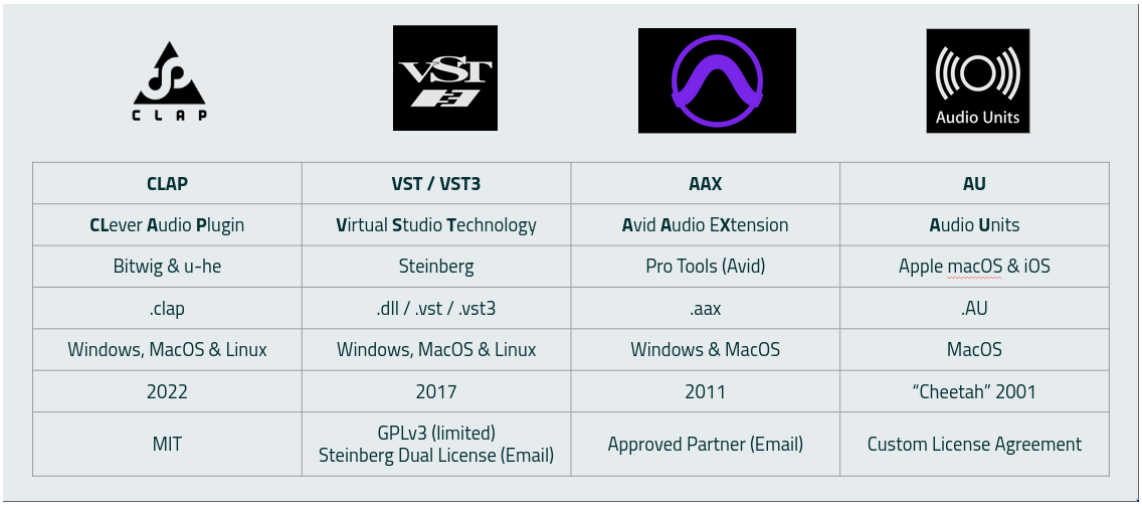
Figure 6: realtime ranking

Figure 6 categorizes different realtime systems based on their severity (Y-axis) and their resulting importance on the general design structure (X-axis). Realtime can be splitted into three different sub-groups: Soft-Realtime, Firm-Realtime and Hard-Realtime requirements. That means missing something with a high severity can result in fatal outcome. As for instance medical monitoring systems or the breaks of a car engine - missing a deadline here could potentially result in death, resulting in a literally **dead**line. Audio is placed in the middle, as missing a deadline here would render the system un-usable for professional use, but at least nothing critical is happening. Video game rendering has an even lower severity since a drop of a frame would not render the product useless and is a quite regular occurence when using such systems.

1.1.4 Audio Plugins: Standards and Hosts

There are many audio plugin standards that evolved over time. However only some of them are still relevant today. Lets have a look at some of the more well known standards:

[5] First In First Out



CLAP	VST / VST3	AAX	AU
CLever **A**udio **P**lugin	**V**irtual **S**tudio **T**echnology	**A**vid **A**udio **EX**tension	**A**udio **U**nits
Bitwig & u-he	Steinberg	Pro Tools (Avid)	Apple macOS & iOS
.clap	.dll / .vst / .vst3	.aax	.AU
Windows, MacOS & Linux	Windows, MacOS & Linux	Windows & MacOS	MacOS
2022	2017	2011	"Cheetah" 2001
MIT	GPLv3 (limited) Steinberg Dual License (Email)	Approved Partner (Email)	Custom License Agreement

Figure 7: plugin standards

Figure 7 shows some of these plugin standards. They are sorted by their time of initial release from left to right. The second row on this table shows the extended name of the abbreviated standard name. After that we can see the companies that were behind the development of these standards. The fourth row shows the file extension those plugins use. However under the hood they are all the same - renamed shared libraries. The next row shows the supported operating systems for the standard, followed by their initial date of release. The last row shows the licensing model that the standard uses.

Some of those standards are platform specific, as Apple's **AU** which provides extended integration into their core audio SDK[6] (Apple, 2023), or they are program specific as Avid's **AAX**, which sole purpose is to provide plugin integration with the Pro Tools DAW[7]. Then there are platform and program independent standards as the **vst3** sdk which is currently one of the most used plugin standards out there, as well as the newest standard **CLAP**.

When it comes to hosting those plugins, it is most often used inside **D**igital **A**udio **W**orkstations (DAWs). Those programs are used to work with audio and midi to create music, record podcasts or use sound-design techniques for custom sounds in games. The usecase is very versatile and there are many different DAW manufacturer:

Figure 8: plugin hosts

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

But plugin hosts are not always DAWs, often the plugin manufacturer include a *Standalone* version of their plugin to be used without any extra programs. One of the notable recent announcements is withing the games industry, with Unreal Engine planning to support the **CLAP** standard in the near future, as announced on the *Unreal Fest 2022*(Engine, 2023).

1.2 Problem Statement

One of the major issues in the integration of Qt user interfaces with these restricted audio plugin environments lays in reentrancy. A program or a function is re-entrant if multiple invocations can safely run concurrently, in other words they can be "re-entered". This behavior is crucial since multiple instances of a plugin are usually created from within those loaded shared libraries. Lets have a look on a simple example which shows this problem:

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

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;
}
```

Static objects and variables in C++ follow the *static storage duration*:

> The storage for the object is allocated when the program begins and deallocated when the program ends. Only one instance of the object exists.

Having only one instance of the object is of key importance here. That means, in other words, that there is only one instance per application process. That also results in extra care that has to be taken when working with static types and reentrancy is required. We get the following result when executing the above program:

```
# 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
```

The first invocation of both functions works, but upon re-entering the function on the second call results in no output for the non-reentrant function. That is, because we use the static specifier inside the for loop. When entering the second time this variable doesn't fulfill the condition of `i < 3` anymore. This code is of course far of the reality and it's whole raison d'être is to highlight the correlation between *static storage duration* and reentrancy of functions.

- Clearly state the research problem and the challenges associated with the integration process.
- No exec, because of problems with multuple instances
- Explain Windows, Linux and Mac behavior, Glib static presence
- QApplication, static environment, can
- Shared Library - non blocking

1.3 Objectives

- Outline the main objectives of the thesis, focusing on the integration of Qt and Clap.

1.4 Scope and Limitations

- Define the boundaries and extent of the research.
- Cross platform: Linux (wayland + xorg), windows and mac
- Identify any limitations or constraints that may affect the integration process.

Chapter 2: The Qt Framework

- Provide a comprehensive introduction to the Qt framework, its history, and its key features.
- Discuss the advantages of using Qt for software development.

2.2 Qt Modules and Architecture, QtGRPC

- Explore the various modules and components of the Qt framework.
- Explain the architecture and design principles of Qt.

2.3 Qt APIs and Tools

- Discuss the different APIs and tools provided by Qt for application development.
- Highlight relevant APIs that will be utilized in the integration process.

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

- 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

Engine, U. (2023, October 23). Hear the future of UE audio | unreal fest 2022. Unreal Engine. retrieved 23.10.2023. Available at: https://www.youtube.com/watch?v=q9pFsl9Cq9c&t=621s

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