

# Audio Plugin Development with QtGrpc

## A Journey Into Remote GUIs

Dennis Oberst

19 October 2023

## **Abstract**

The Qt framework is well-established in the domain of cross-platform application development, yet its application within audio plugin development has remained largely unexplored. This thesis explores the integration of Qt in audio plugin development, examining the synergy between Qt's robust GUI capabilities and the emerging CLAP (Clever Audio Plugin) standard. CLAP, with its emphasis on simplicity, clarity, and robustness, offers a streamlined and intuitive API that aligns with Qt's design principles.

By integrating gRPC (g Remote Procedure Calls), an open-source communication protocol, the research highlights a method for enhancing interactions between audio plugins and host applications. This integration points to a flexible, scalable approach, suitable for modern software design.

This work presents a novel approach to audio plugin development that leverages the combined strengths of Qt, CLAP, and gRPC. The resulting library not only provides a consistent and adaptable user experience across various operating systems but also simplifies the plugin creation process. This work stands as a testament to the untapped potential of Qt in the audio-plugin industry, paving the way for advancements in audio plugin creation that enhance both user engagement and developer workflow.

# Contents

<b>Chapter 1: Introduction</b>	<b>2</b>
1.1 Motivation . . . . .	2
1.2 Objective . . . . .	2
1.3 Structure . . . . .	5
<b>Chapter 2: Foundation</b>	<b>6</b>
2.1 Plugins . . . . .	6
2.1.1 Shared Libraries and Plugins . . . . .	7
2.1.2 Audio Plugins . . . . .	10
2.1.3 Standards and Hosts . . . . .	13
2.2 The CLAP Audio Plugin Standard . . . . .	14
2.2.1 CLAP basics . . . . .	14
2.2.2 Creating a CLAP Plugin . . . . .	16
2.2.3 Debugging . . . . .	21
2.2.4 Extensions . . . . .	22
2.2.5 Processing . . . . .	25
2.3 gRPC - Remote Procedure Calls . . . . .	27
2.3.1 Protobuf . . . . .	28
2.3.2 gRPC Core Concepts . . . . .	30
2.3.3 Performance . . . . .	31
2.3 The Qt Framework . . . . .	32
2.3.1 Core Techniques . . . . .	32
2.3.2 Graphics . . . . .	35
2.3.3 QtGrpc and QtProtobuf . . . . .	37
<b>3 Remote Plugins</b>	<b>39</b>
3.1 Introduction . . . . .	39
3.2 Remote Control Interface . . . . .	40
3.3 QtGrpc Client Implementation . . . . .	40
<b>4 Conclusions</b>	<b>40</b>
Integrity . . . . .	40
Performance . . . . .	40
Future Work . . . . .	40
<b>Bibliography</b>	<b>40</b>

# Chapter 1: Introduction

## 1.1 Motivation

Throughout this work, we will discuss plugins and their impact on the usability of an application. Specifically, we will focus on 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 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 the API<sup>1</sup> of the chosen toolkit doesn't support the targeted platform.

Therefore, Qt, a cross-platform framework for crafting GUIs, comes to mind when considering the development of an audio plugin UI<sup>2</sup> 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.

## 1.2 Objective

A primary challenge in integrating Qt user interfaces within audio plugin environments centers 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 within itself. To illustrate this challenge, consider the following example:

```
// 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;
```

---

<sup>1</sup>Application Programming Interface

<sup>2</sup>User Interface

```

}

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 problem here is the single 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 of both functions is successful, re-entering the function during the second call results in no output from the non-reentrant function. This outcome stems from the use of the static specifier in the for loop counter. Due to this, the counter doesn't reset between calls, failing to meet the `i < 3` condition in subsequent invocations, as it retains its initialized value from the first call.

Static objects offer global accessibility and can enhance application design by allowing for the initialization of crucial objects just once, with the capability to share them throughout the entire codebase. However, this approach has its trade-offs, especially in terms of integration and operation in multi-threaded environments. This is evident in the case of QApplication variants like QCoreApplication and QApplication. These classes, which manage Qt's 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`, already utilizes a QApplication object or when multiple plugin instances operate within a single 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 QApplication(argc, argv);
}
~~~

// Set our 'QWindow *window' to the received window from the host.
window->setParent(QWindow::fromWinId(WId(hostWindow)));
```

Attempting to reuse the event system of a parent window, while occasionally effective, is fraught with uncertainties. A primary limitation is that this approach is not consistently supported across different platforms. For instance on Linux, where event systems such as **xcb** and **glib** are not standardized would render this method impractical. Additionally, there are risks associated with connecting to an event loop from an outdated version, which could lead to compatibility issues and impaired functionality.

Another concern is the management of multiple instances attempting to connect to the same event loop, raising doubts about the reliability and efficiency. Consequently, this approach is not a dependable solution for the challenges faced in integrating Qt user interfaces with audio plugin environments.

This research is focused on developing innovative methods to integrate Qt-based graphical user interfaces seamlessly into the audio plugin landscape. The main objective is to provide seamless integration with audio plugin standards, ensuring effective and responsible communication. A crucial aspect of this integration is to enhance user experience by providing a stable method that works across all major operating systems. An important part of achieving this is the seamless integration of events generated by the plugin into Qt's event system. This strategy is designed to offer a development process that is inherently aligned with Qt's principles, making it more intuitive for developers and ensuring a robust, platform-independent solution.

The research primarily focuses on the CLAP plugin standard. CLAP is chosen for its innovative capabilities and its relevance in the context of current technology trends. This standard is viewed as the most suitable for the intended integration tasks. The usage of CLAP emphasizes the study's aim to tackle the intricate challenges of integrating Qt's comprehensive GUI framework with the dynamic field of audio plugins.

### **1.3 Structure**

TODO: Structure

## Chapter 2: Foundation

### 2.1 Plugins

**Plug-ins**, at their essence, serve as dynamic extensions, enhancing the capabilities of a plugin-loading host. One can perceive them as *on-demand* features 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*.

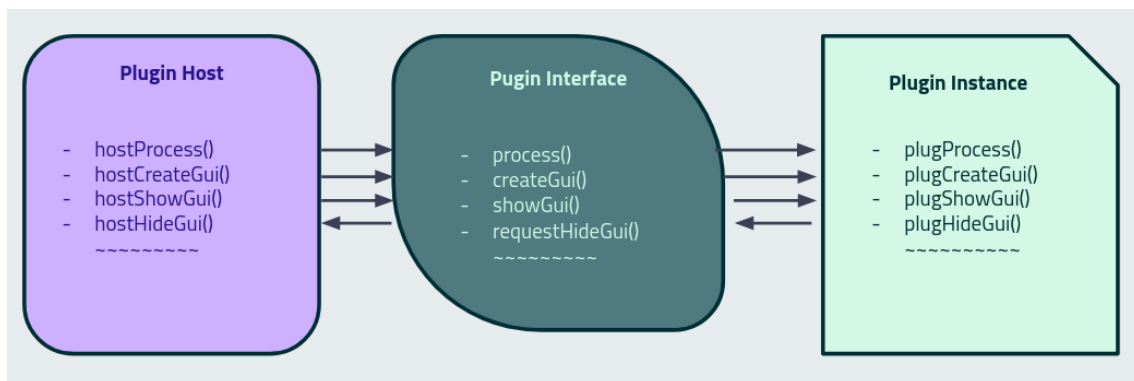


Figure 1: basic plugin architecture

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 correlates with the actual implementation of the plugin. Once the plugin is successfully loaded 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 their 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 practical illustration of such an interface is seen in the **CLever Audio Plugin (CLAP)** format. This standard establishes the communication protocols between a Digital Audio Workstation (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 function returns a `bool` and receives a constant pointer to the struct `clap_plugin` as an argument. The preprocessor macro `CLAP_ABI` is an implementation detail without significant importance. In a practical scenario, a plugin can assign a specific function to this pointer. This allows the plugin to respond to calls made by the host, which controls the timing of these calls



### 2.1.1 Shared Libraries and Plugins

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<sup>3</sup>, 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.

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 the DSO manually with libraries like [dlopen](#) or [QLibrary](#).

In the context of audio plugins, the approach of dynamically loaded libraries is commonly used for loading plugin instances. Unlike dynamically linked libraries, where the kernel handles the loading post-compilation, dynamically loaded libraries give the application the autonomy to load the Dynamic Shared Object (DSO) manually. 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:

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

#ifdef _WIN32
#   define EXPORT __declspec(dllexport)
```

---

<sup>3</sup>Dynamic Shared Object

```

#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 visibility 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 visibility 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 visibility in the linking step by simply using the linker flag `-fvisibility=hidden` to hide all symbols. You might noticed the absence of `__declspec(dllimport)` here. We actually don't need it in this example because we're going to load the library manually, like a plugin.

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, let's 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

```

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 Interface, which corresponds to the actual binary that gets executed.

In software development, as we move from version 1.1 to 1.2, then to 1.3, and so on, keeping a consistent interface is crucial. This is where **Binary Compatibility** comes into play. It's essential for ensuring that various versions of software libraries can work together smoothly. Ideally, software built with version 1.1 of an interface should seamlessly operate with version 1.3, without recompilation or other adjustments.

Take this example:

```
// Version 1.0 of 'my_library'
typedef uint32_t MyType;
~~~
void my_library_api(MyType type);
```

Here, `MyType` is a 32-bit unsigned integer. But in the next version:

```
// Version 1.1 of 'my_library'
typedef uint64_t MyType;
~~~
void my_library_api(MyType type);
```

`MyType` is updated to a 64-bit integer. This change maintains **Source Compatibility** (the code still compiles) but breaks **Binary Compatibility**. Binary compatibility is lost because the binary interface of the library, has changed due to the increased size of `MyType`.

Maintaining binary compatibility is vital for the stability of systems, especially when updating shared libraries. It allows users to upgrade software without disrupting existing functionalities.

### 2.1.2 Audio Plugins

In the realm of audio plugins, two primary components define their structure: the realtime audio section and the control section.

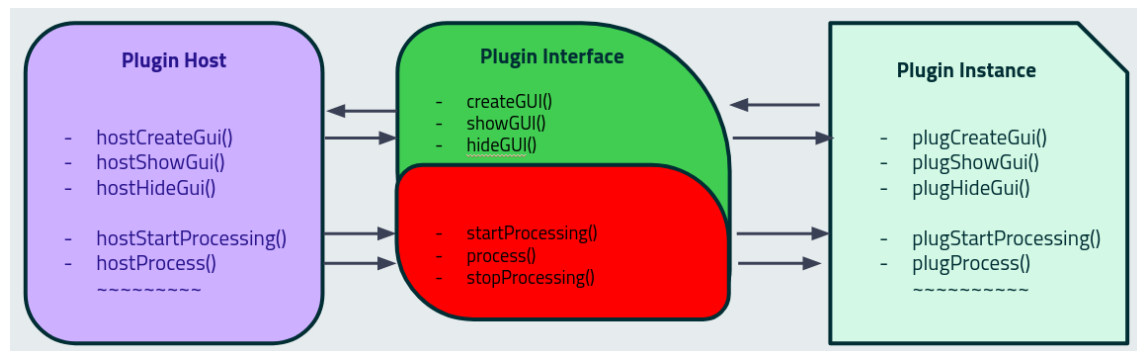


Figure 2: basic audio plugin architecture

Figure 2, shows a typical plugin interface with two distinct colors for clarity. The green section represents the plugin's interaction with the host through a low-priority main thread. This part primarily manages tasks like GUI setup and other control functions. "Low priority" means that it can handle non-deterministic operations, which may briefly pause execution.

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. For instance, 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.<sup>4</sup>

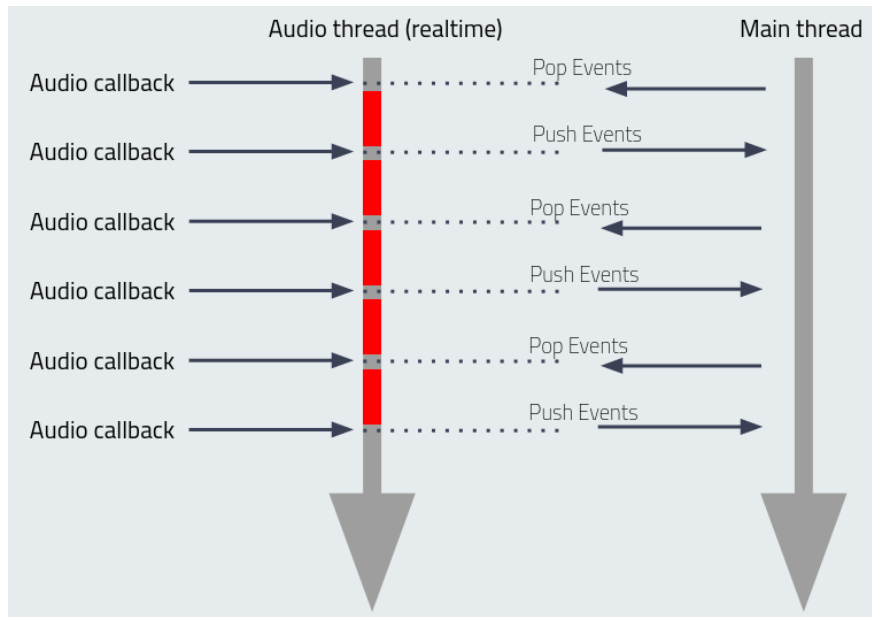


Figure 3: Realtime overview

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 audio buffers, along with any events related to the audio plugin (such as a user adjusting a dial within the host).
2. The plugin must swiftly act on these values and compute its functionality based on these input parameters. Given that a GUI is often integrated, synchronization with the audio engine's events is vital. Once all 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. Therefore, it's essential to prioritize punctuality and precision in all aspects of audio work(Scavone 2002). Keeping in mind the simple but crucial reminder:

Don't miss your Deadline!

$$\frac{\text{AudioBufferSize}}{\text{SamplingFrequency}} = \text{CallbackFrequency} \quad , \text{ e.g. } \frac{512}{48000\text{Hz}} = 10.67\text{ms} \quad (1)$$

The equation above 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 block-size generally fluctuates between 128 - 2048 samples, with sampling rates ranging from 48'000 - 192'000 Hz. Consequently, our callback algorithm must operate within approximately **2.9ms - 46.4ms** for optimal functionality(Joseph Timoney 2015).

<sup>4</sup>Inspired by: NppNow 2023, What is Low Latency C++? (Part 2) - Timur Doumler

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 normal thread brings this into sharper focus: In a realtime thread, responses must meet specific deadlines, ensuring immediate and predictable behavior. In contrast, a normal thread has more flexibility in its operation, allowing for variable response times without the demanding need for timely execution.

Table 1: Realtime limitations.<sup>5</sup>

Problems to Real-time	Real-time	Non-real-time
CPU work	+	+
Context switches	+ (avoid)	+
Memory access	+ (non-paged)	+
System calls	x	+
Allocations	x	+
Deallocations	x	+
Exceptions	x	+
Priority Inversion	x	+

The above table 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 often use non-blocking and wait free data structures as FIFO<sup>6</sup> queues or ring-buffers to complement this.

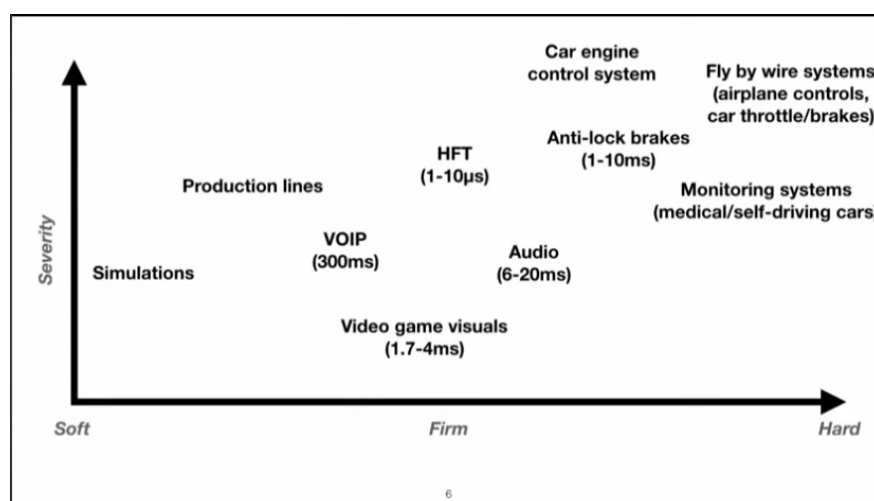


Figure 4: Realtime ranking

<sup>5</sup>Taken from: [ADC 2019, Fabian Renn-Giles & Dave Rowland - Real-time 101](#)

<sup>6</sup>First In First Out

Figure 4 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 **deadline**. 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.

### 2.1.3 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
<b>CLAP</b>	Clever Audio Plugin	Bitwig & U-he	.clap	Windows, MacOS & Linux	2022	MIT
<b>VST/VST3</b>	Virtual Studio Technology	Steinberg	.dll, .vst, .vst3	Windows, MacOS & Linux	2017	GPLv3, Steinberg Dual License
<b>AAX</b>	Avid Audio Extension	Pro Tools (Avid)	.aax	Windows & MacOS	2011	Approved Partner
<b>AU</b>	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<sup>7</sup> (Apple 2023). Similarly, Avid's **AAX** is designed exclusively for plugin compatibility with the **Pro Tools** DAW<sup>8</sup>. 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:

<sup>7</sup>Software Development Kit

<sup>8</sup>Digital Audio Workstation

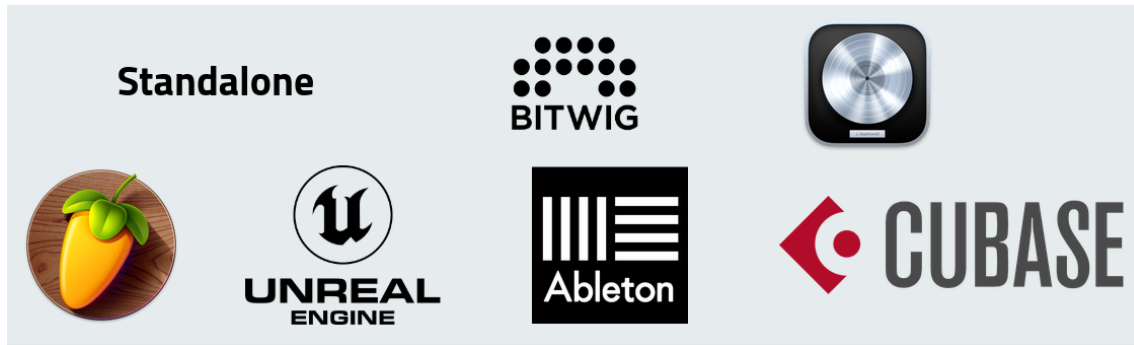


Figure 5: plugin hosts

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](#).

## 2.2 The CLAP Audio Plugin Standard

Originating from the collaboration between Berlin-based companies Bitwig and u-he, CLAP (**CL**ever **A**udio **P**lugin) emerged as a new plugin standard. Born from developer Alexandre Bique’s vision in 2014, it was revitalized in 2021 with a focus on three core concepts: Simplicity, Clarity, and Robustness.

CLAP stands out for its:

- *Consistent API* and *flat architecture*, making plugin creation more intuitive.
- *Adaptability*, allowing flexible integration of unique features like per-voice modulation (MPE<sup>9</sup> on steroids) or a host thread-pool extension.
- Open-source ethos, making the standard accessible under the [MIT License](#).

CLAP establishes a stable and backward compatible Application Binary Interface. This ABI forms a bridge for communication between digital audio workstations and audio plugins such as synthesizers and effect units.

### 2.2.1 CLAP basics

The principal advantage of CLAP is its simplicity. In contrast to other plugin standards, such as VST, which often involve complex structures including deep inheritance hierarchies that complicate both debugging and understanding the code, CLAP offers a refreshingly straightforward and flat architecture. With CLAP, a single exported symbol is the foundation from which the entire plugin functionality extends: the `clap_plugin_entry` type. This must be made available by the DSO and is the only exported symbol.

Note: We will use the term *CLAP* to refer to the DSO that houses the plugin implementation of the clap interface. This is a common phrase in the CLAP community.

<sup>9</sup>Midi Polyphonic Expression



```
// <clap/entry.h>
typedef struct clap_plugin_entry {
    clap_version_t clap_version;
    bool(CLAP_ABI *init)(const char *plugin_path);
    void(CLAP_ABI *deinit)(void);
    const void *(CLAP_ABI *get_factory)(const char *factory_id);
} clap_plugin_entry_t;
```

The `clap_version_t` type specifies the version the plugin is created with. Following this, there are three function pointers declared:

1. The initialization function `bool init(const char*)` is the first function called by the host. It is primarily used for plugin scanning and is designed to execute quickly.
2. The de-initialization function `void deinit(void)` is invoked when the DSO is unloaded, which typically occurs after the final plugin instance is closed.
3. The `const void* get_factory(const char*)` serves as the “constructor” for the plugin, tasked with creating new plugin instances. A notable aspect of CLAP plugins is their containerized nature, allowing a single DSO to encapsulate multiple distinct plugins.

```
typedef struct clap_plugin_factory {
    uint32_t(CLAP_ABI *get_plugin_count)(const struct clap_plugin_factory *factory);

    const clap_plugin_descriptor_t *(CLAP_ABI *get_plugin_descriptor)(
        const struct clap_plugin_factory *factory, uint32_t index);

    const clap_plugin_t *(CLAP_ABI *create_plugin)(
        const struct clap_plugin_factory *factory, const clap_host_t *host,
        const char *plugin_id);
} clap_plugin_factory_t;
```

The plugin factory acts as a hub for creating plugin instances within a given CLAP object. The structure requires three function pointers:

1. `get_plugin_count(~)` determines the number of distinct plugins available within the CLAP.
2. `get_plugin_descriptor(~)` retrieves a description of each plugin, which is used by the host to present information about the plugin to users.
3. `create_plugin(~)` is responsible for instantiating the plugin(s).

At the heart of the communication between the host and a plugin is the `clap_plugin_t` type. It defines the essential functions that a host will invoke to control the plugin, such as initializing, processing audio, and handling events. It is through this interface that the plugin exposes its capabilities and responds to the host, thereby allowing for the dynamic and interactive processes required for audio manipulation and creation.

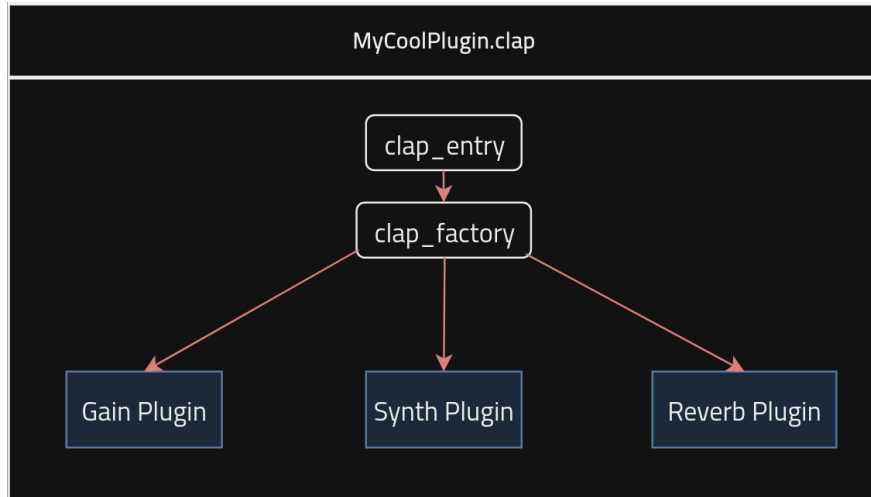


Figure 6: CLAP factory

### 2.2.2 Creating a CLAP Plugin

To illustrate the ease with which one can get started with CLAP, we will develop a simple gain plugin, a basic yet foundational tool in audio processing. A gain plugin's role is to control the output volume by adjusting the input signal's level in decibels. Accordingly, our plugin will manage the audio inputs and outputs and feature a single adjustable parameter: gain.

Setting up our project is straightforward. We'll create a directory for our plugin, obtain the CLAP library, and prepare the initial files:

```
mkdir mini_clap && cd mini_clap
git clone https://github.com/free-audio/clap.git
touch mini_gain.cpp
touch CMakeLists.txt
```

Next, we'll configure the build system:

```
cmake_minimum_required(VERSION 3.2)
project(MiniGain LANGUAGES C CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

set(target mini_gain)
add_library(${target} SHARED mini_gain.cpp)

# CLAP is a header-only library, so we just need to include it
target_include_directories(${target} PRIVATE clap/include)
```

This CMake snippet sets up the build environment for our *MiniGain* project. We enable C++20 to use the latest language features and declare a shared library named `mini_gain` which will be built from

mini\_gain.cpp.

To meet the CLAP naming conventions, we need to modify the library's output name:

```
# A CLAP is just a renamed shared library
set_target_properties(${target} PROPERTIES PREFIX "")
set_target_properties(${target} PROPERTIES SUFFIX ".clap")
```

This configuration tells CMake to output our library with the name `mini_gain.clap`.

While the output is correctly named, it remains within the build directory, which isn't automatically recognized by CLAP hosts. For practical development, creating a symlink to the expected location is beneficial. We'll append a post-build command to do just that:

```
# Default search path for CLAP plugins. See also <clap/entry.h>
if(UNIX)
    if(APPLE)
        set(CLAP_USER_PATH "$ENV{HOME}/Library/Audio/Plug-Ins/CLAP")
    else()
        set(CLAP_USER_PATH "$ENV{HOME}/.clap")
    endif()
elseif(WIN32)
    set(CLAP_USER_PATH "$ENV{LOCALAPPDATA}\\Programs\\Common\\CLAP")
endif()

# Create a symlink post-build to make development easier
add_custom_command(
    TARGET ${target} POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E create_symlink
    "$<TARGET_FILE:${target}>"
    "${CLAP_USER_PATH}/${<TARGET_FILE_NAME:${target}>}"
)
```

We start by defining the entry point for the plugin. Here's how you might proceed:

```
// mini_gain.cpp
#include <clap/clap.h>

#include <set>
#include <format>
#include <cstring>
#include <iostream>

// 'clap_entry' is the only symbol exported by the plugin.
extern "C" CLAP_EXPORT const clap_plugin_entry clap_entry = {
    .clap_version = CLAP_VERSION,
```

```

// Called after the DSO is loaded and before it is unloaded.
.init = [](const char* path) → bool {
    std::cout << std::format("MiniGain -- initialized: {}\n", path);
    return true;
},
.deinit = []() → void {
    std::cout << "MiniGain -- deinitialized\n";
},

// Provide a factory for creating 'MiniGain' instances.
.get_factory = [](const char* factoryId) → const void* {
    if (strcmp(factoryId, CLAP_PLUGIN_FACTORY_ID) == 0) // sanity check
        return &pluginFactory;
    return nullptr;
}
};

```

First, we include the `clap.h` header file, which aggregates all components from the CLAP API. We also import some standard headers for later use and initialize the fields of the `clap_plugin_entry` type with simple lambda functions. Our plugin factory is designed to offer a single plugin within this DSO:

```

// The factory is responsible for creating plugin instances.
const clap_plugin_factory pluginFactory = {
    // This CLAP has only one plugin to offer
    .get_plugin_count = [](auto*) → uint32_t {
        return 1;
    },
    // Return the metadata for 'MiniGain'
    .get_plugin_descriptor = [](auto*, uint32_t idx) → const auto* {
        return (idx == 0) ? &MiniGain::Descriptor : nullptr;
    },
    // Create a plugin if the IDs match.
    .create_plugin = [](auto*, const clap_host* host, const char* id) → const auto* {
        if (strcmp(id, MiniGain::Descriptor.id) == 0)
            return MiniGain::create(host);
        return static_cast<clap_plugin*>(nullptr);
    }
};

```

The `host` pointer serves as a means of communication from the plugin towards the host, requesting necessary services and functionality.

The implementation of our plugin in modern C++ requires a mechanism to interact with the C-API of the CLAP standard. Since the `clap_plugin_t` struct expects static function pointers and member functions are not static, we resolve this mismatch using *trampoline* functions, also known as *glue rou-*

tines. These functions connect the static world of the C-API with the instance-specific context of our C++ classes:

```
MiniGain* MiniGain::self(const clap_plugin *plugin) {
    // Cast plugin_data back to MiniGain* to retrieve the class instance.
    return static_cast<MiniGain*>(plugin->plugin_data);
}

// A glue layer between our C++ class and the C API.
void MiniGain::initializePlugin() {
    mPlugin.desc = &Descriptor;
    mPlugin.plugin_data = this; // Link this instance with the plugin data.

    mPlugin.destroy = [](const clap_plugin* p) {
        self(p)->destroy();
    };

    mPlugin.process = [](const clap_plugin* p, const clap_process* proc) {
        return self(p)->process(proc);
    };

    mPlugin.get_extension = [](const clap_plugin* p, const char* id) -> const void* {
        return nullptr; // TODO: Add extensions.
    };

    // Simplified for brevity.
    mPlugin.init = [](const auto*) { return true; };
    mPlugin.activate = [](const auto*, double, uint32_t, uint32_t) { return true; };
    mPlugin.deactivate = [](const auto*) {};
    mPlugin.start_processing = [](const auto*) { return true; };
    mPlugin.stop_processing = [](const auto*) {};
    mPlugin.reset = [](const auto*) {};
    mPlugin.on_main_thread = [](const auto*) {};
}
```

The code sets up glue routines to redirect calls from the static C API to our C++ class methods. The `self` function retrieves the class instance from `plugin_data`, facilitating the call to the relevant member functions. We focus on `destroy` and `process`, with other functions returning defaults to streamline our plugin's integration with the host.

Finally we create the `MiniGain` class which encapsulates the functionality of our plugin:

```
class MiniGain {
public:
    constexpr static const clap_plugin_descriptor Descriptor = {
        .clap_version = CLAP_VERSION,
        .id = "mini.gain",
        .name = "MiniGain",
        .vendor = "Example",
        .version = "1.0.0",
    };
};
```

```

        .description = "A Minimal CLAP plugin",
        .features = (const char*[]){ CLAP_PLUGIN_FEATURE_MIXING, nullptr }
    };

    static clap_plugin* create(const clap_host *host) {
        std::cout << std::format("{} -- Creating instance for host: <{}>, v{}>, {}>\n",
            Descriptor.name, host->name, host->version, pluginInstances.size()
        );
        auto [plug, success] = pluginInstances.emplace(new MiniGain(host));
        return success ? &(*plug)->mPlugin : nullptr;
    }

    void destroy() {
        pluginInstances.erase(this);
        std::cout << std::format("{} -- Destroying instance. {} plugins left\n",
            Descriptor.name, pluginInstances.size()
        );
    }

    clap_process_status process(const clap_process *process) {
        return {}; // TODO: Implement processing logic.
    }

private:
    explicit MiniGain(const clap_host *host) : mHost(host) { initializePlugin(); }
    static MiniGain *self(const clap_plugin *plugin);
    void initializePlugin();

    // private members:
    [[maybe_unused]] const clap_host *mHost = nullptr;
    clap_plugin mPlugin = {};
    inline static std::set<MiniGain*> pluginInstances{};
};

```

The `Descriptor` in the `MiniGain` class contains essential metadata for the plugin, including identifiers and versioning. The `create` function allocates a plugin instance and stores it inside a static `std::set`. Overall, this minimalistic plugin structure is achieved in approximately 100 lines of code.

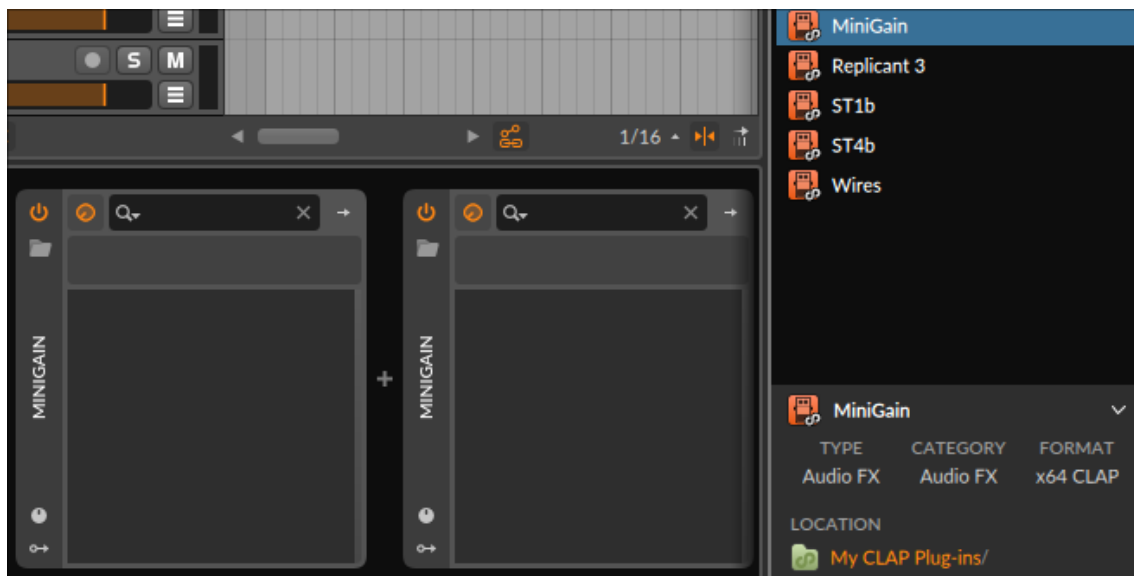


Figure 7: MiniGain hosted

This implementation of the MiniGain plugin is already sufficient for it to be recognized and loaded by CLAP-compatible hosts. At this stage, a developer might wonder about how to debug such a plugin, considering it operates as a shared library that has to be loaded and called.

### 2.2.3 Debugging

Debugging is a crucial step in software development. When the complexity of our program increases or even if we just want to verify correct behavior, it is great to step through the code with debuggers such as [gdb](#) or [lldb](#).

When developing shared objects, debugging introduces an additional layer of complexity. We rely upon a host that has the CLAP standard implemented and all features supported.

Fortunately, Bitwig offers a streamlined approach to this challenge. Since their audio engine runs as a standalone executable, we can simply start it with a debugger and latch onto the breakpoints of our plugin. To set up debugging in Bitwig, carry out the following steps:

1. Bitwig supports different hosting modes. For debugging purposes, we want the plugin to be loaded within the same thread as the audio engine, so we use **Within Bitwig**

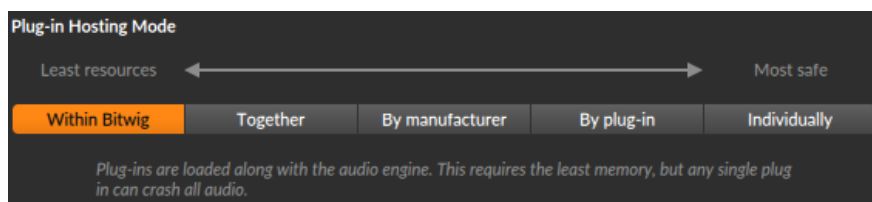


Figure 8: Bitwig Hosting

2. If the audio engine is running, we must first shut it down. Do this by right-clicking in the transport bar and choosing **terminate audio engine**.

Now we can execute the audio engine with the debugger of choice. The executable is located inside Bitwig's installation directory. On my linux machine this would be:

```
gdb /opt/bitwig-studio/bin/BitwigAudioEngine-X64-AVX2
(gdb) run
```

With the debugger running, you'll have access to debugging tools, and you can monitor the console output for any print statements. For instance, here's what you might see when creating and then removing three instances of the `MiniGain` plugin:

```
MiniGain -- initialized: /home/wayn/.clap/mini_gain.clap
MiniGain -- Creating instance for host: <Bitwig Studio, v5.0.7, 0>
MiniGain -- Creating instance for host: <Bitwig Studio, v5.0.7, 1>
MiniGain -- Creating instance for host: <Bitwig Studio, v5.0.7, 2>

PluginHost: Destroying plugin with id 3
MiniGain -- Destroying instance. 2 plugins left
PluginHost: Destroying plugin with id 2
MiniGain -- Destroying instance. 1 plugins left
PluginHost: Destroying plugin with id 1
MiniGain -- Destroying instance. 0 plugins left
```

## 2.2.4 Extensions

The MiniGain plugin, in its current state, is not yet operational due to the absence of needed extensions. While the fundamental framework of the plugin has been established, the extensions are key to unlocking its full potential. The CLAP repository's `ext/` directory is home to a range of extensions, including *gui* and *state*, among others. To fulfill our plugin's requirements of managing a parameter and processing an audio input and output, we focus on incorporating the **audio-ports** and **params** extensions:

```
~~~
void setParamGain(double value) noexcept { mParamGain = value; }
[[nodiscard]] double paramGain() const noexcept { return mParamGain; }
private:
    explicit MiniGain(const clap_host *host) : mHost(host) { initialize(); }
    void initialize() {
        initializePlugin();
        initializeExtAudioPorts();
        initializeExtParams();
    }
    void initializePlugin();
    void initializeExtAudioPorts();
    void initializeExtParams();

    clap_plugin_audio_ports mExtAudioPorts = {};
    clap_plugin_params mExtParams = {};
```



```
double mParamGain = 0.0;
~~~
```

We add an initialization function that bundles the setup processes for these extensions. We maintain the parameter value in a dedicated variable and provide access through getters and setters. Finally, to utilize these extensions, the plugin communicates the supported functionalities back to the host:

```
// initializePlugin() {
~~~

    .get_extension = [](const clap_plugin* p, const char* id) → const void* {
        if (!strcmp(id, CLAP_EXT_PARAMS))
            return &self(p)→mExtParams;
        else if (!strcmp(id, CLAP_EXT_AUDIO_PORTS))
            return &self(p)→mExtAudioPorts;
        return nullptr;
    },
~~~
```

The host determines the plugin's supported extensions by calling this function with all supported extension IDs. This ensures the host recognizes the plugin's capabilities. For MiniGain, the next step involves implementing the audio-ports extension as follows:

```
void MiniGain::initializeExtAudioPorts() {
    mExtAudioPorts = {
        .count = [](const clap_plugin*, bool) → uint32_t { return 1; },
        .get = [](const clap_plugin*, uint32_t index, bool isInput,
            ↪ clap_audio_port_info *info) {
            if (index ≠ 0)
                return false;
            info→id = 0;
            std::snprintf(info→name, sizeof(info→name), "%s %s", Descriptor.name,
                ↪ isInput ? "IN" : "OUT");
            info→channel_count = 2; // Stereo
            info→flags = CLAP_AUDIO_PORT_IS_MAIN;
            info→port_type = CLAP_PORT_STEREO;
            info→in_place_pair = CLAP_INVALID_ID;
            return true;
        }
    };
}
```

With the `count` function, the plugin notifies the host about the plugin's audio input and output capabilities. The `get` function further details the configuration, marking a stereo channel setup. This establishes the essential framework for the plugin to access audio input and output streams during the process callback.

The parameter extension in MiniGain outlines the parameters the plugin possesses and provides meta-data about them. The initialization function for the extension is defined as follows:

```
void MiniGain::initializeExtParams() {
    mExtParams = {
        .count = [](const clap_plugin* p) → uint32_t {
            return 1; // Single parameter for gain
        },
        .get_info = [](const clap_plugin*, uint32_t index, clap_param_info *info) →
        ↪ bool {
            if (index ≠ 0)
                return false;
            info→id = 0;
            info→flags = CLAP_PARAM_IS_AUTOMATABLE;
            info→cookie = nullptr;
            std::snprintf(info→name, sizeof(info→name), "%s", "Gain");
            std::snprintf(info→module, sizeof(info→module), "%s %s",
                ↪ MiniGain::Descriptor.name, "Module");
            info→min_value = -40.0;
            info→max_value = 40.0;
            info→default_value = 0.0;
            return true;
        },
    },
```

Here, `count` reveals that the plugin hosts a single gain parameter, and `get_info` provides crucial details such as the parameter's range and default setting.

```
.get_value = [](const clap_plugin* p, clap_id id, double *out) → bool {
    if (id ≠ 0)
        return false;
    *out = self(p)→paramGain();
    return true;
},
.value_to_text = [](const clap_plugin*, clap_id id, double value, char* out,
    ↪ uint32_t outSize) → bool {
    if (id ≠ 0)
        return false;
    std::snprintf(out, outSize, "%g %s", value, " dB");
    return true;
},
.text_to_value = [](const clap_plugin*, clap_id id, const char* text, double*
    ↪ out) → bool {
    if (id ≠ 0)
        return false;
    *out = std::strtod(text, nullptr);
    return true;
},
```

```

    },
    .flush = [](const clap_plugin*, const auto*, const auto*) {
        // noop
    },
};

```

The parameter's current value is fetched using `get_value`, while `value_to_text` and `text_to_value` manage the conversion between numerical values and user-readable text, appending 'dB' to indicate decibels. With these definitions, compiling and loading the plugin into a host will now allow us to interact with the parameter.

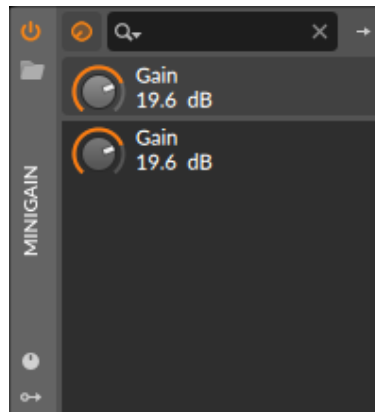


Figure 9: MiniGain hosted

## 2.2.5 Processing

To make the *MiniGain* plugin fully functional, we must finally tackle the `process` callback, which is the heart of all audio processing and event handling. This function is responsible for managing audio data and responding to events, such as parameter changes or MIDI note triggers.



Figure 10: CLAP parameter event

CLAP has a sophisticated method for coupling events with audio buffers. Events come with a header and payload; the header's `flags` denote the payload type, and the payload contains the event data.

In our case, it's the `clap_event_param_value`, carrying information about parameter changes.

The `process` function works with frames, which encapsulate both the audio samples and any associated event data in a time-ordered fashion. This approach ensures that the audio processing is accurate and responsive to real-time control changes.

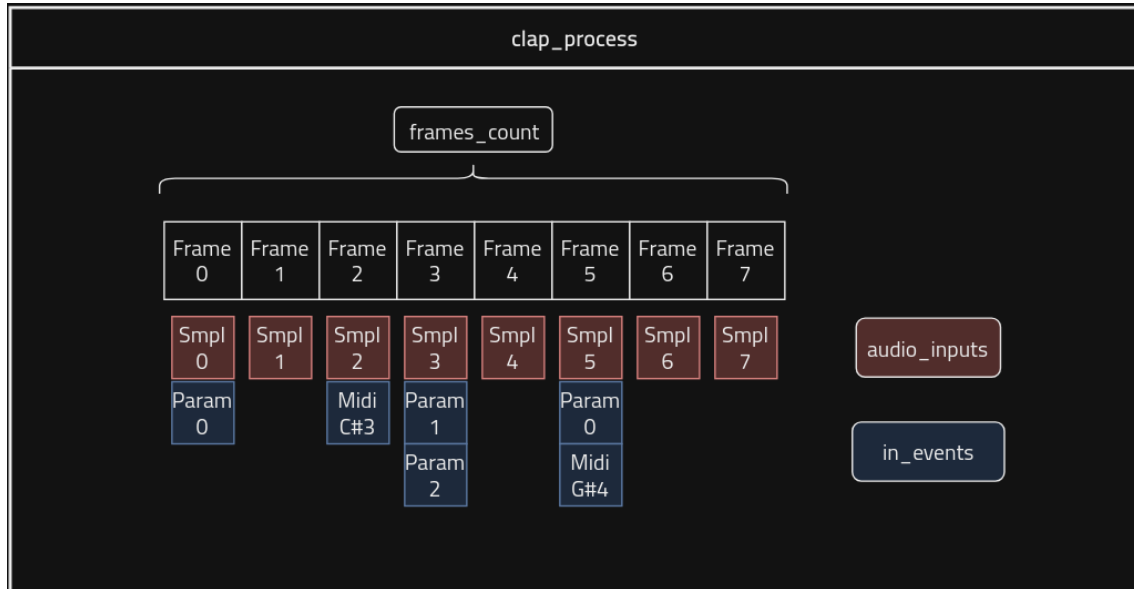


Figure 11: CLAP process

To implement the `process` function in *MiniGain*, we need to write code that iterates through these frames, applies the gain value to the audio samples, and handles any parameter change events. It is crucial to ensure that the gain adjustments occur precisely at the event's timestamp within the audio buffer to maintain tight synchronization between user actions and audio output.

By integrating this final piece, we'll have a working gain plugin that not only processes audio but also responds dynamically to user interactions. The code would look something like this:

```
clap_process_status process(const clap_process *process) {
    const auto *inEvents = process->in_events;
    const auto numEvents = process->in_events->size(inEvents);
    uint32_t eventIndex = 0;

    for (uint32_t frame = 0; frame < process->frames_count; ++frame) {
        // Process all events at given @frame, there may be more than one.
        while (eventIndex < numEvents) {
            const auto *eventHeader = inEvents->get(inEvents, eventIndex);
            if (eventHeader->time != frame) // Consumed all event for @frame
                break;
            switch(eventHeader->type) {
                case CLAP_EVENT_PARAM_VALUE: {
                    const auto *event = reinterpret_cast<const
                        ↪ clap_event_param_value*>(eventHeader);
```

```

        // Since we only interact with the parameter from this
        // thread we don't need to synchronize access to it.
        if (event→param_id == 0)
            setParamGain(event→value);
    };
}
++eventIndex;
}
// Process audio for given @frame.
const float gain = std::pow(10.0f, static_cast<float>(paramGain()) / 20.0f);
const float inputL = process→audio_inputs→data32[0][frame];
const float inputR = process→audio_inputs→data32[1][frame];
process→audio_outputs→data32[0][frame] = inputL * gain;
process→audio_outputs→data32[1][frame] = inputR * gain;
}
return CLAP_PROCESS_SLEEP;
}

```

The specific implementation will involve fetching the gain parameter, responding to parameter events, and manipulating the audio buffer accordingly. Once implemented, compiling and running the plugin should yield a controllable gain effect within the host application.

## 2.3 gRPC - Remote Procedure Calls

In today's fast-paced software development landscape, there is a growing need for efficient communication between various software systems. Addressing this demand, [gRPC](#), which stands for *g Remote Procedure Calls*, has risen as an open-source framework specifically tailored for this purpose. If you want to find out about the current meaning of [g](#) in gRPC, please consult the official documentation for clearance as it changes between versions. Its standout features include impressive speed, compatibility with a wide range of programming languages, and a steadily increasing adoption rate. These attributes have positioned gRPC as a leading choice for inter-service communication in contemporary software architecture.



Figure 12: gRPC supported languages

### 2.3.1 Protobuf

Central to gRPC's efficiency and flexibility is the *Protocol Buffer*, often abbreviated as protobuf. Developed by Google, protobuf is a serialization format that efficiently converts structured data into a format optimized for smooth transmission and reception.

**The Rationale Behind Protobuf** Traditionally, data interchange between systems utilized formats such as XML or JSON. While these formats are human-readable and widely accepted, they can be quite verbose. This increased verbosity can slow down transmission and demand more storage, leading to potential inefficiencies. In contrast, protobuf offers a concise binary format, resulting in faster transmission and reduced data overhead, positioning it as a preferred choice for many developers(Popić & Bojan Mrazovac & Dražen Pezer & Nikola Teslić" 2016).

**Flexibility in Design** A standout feature of protobuf is its universal approach. Developers can outline their data structures and services using an *Interface Definition Language* (IDL). IDLs are used to define data structures and interfaces in a language-neutral manner, ensuring they can be used across various platforms and languages. Once defined, this IDL can be compiled to produce libraries that are compatible with numerous programming languages(Lakhani" 2014), ensuring coherence even when different system components are developed in diverse languages.

**Seamless Evolution** As software services continually evolve, it's essential that changes do not disrupt existing functionalities. Protobuf's design flexibility allows for such evolutions without hindering compatibility. This adaptability ensures newer versions of a service can integrate seamlessly with older versions, ensuring consistent functionality(Popić & Bojan Mrazovac & Dražen Pezer & Nikola Teslić" 2016).

To encapsulate, protobuf's attributes include:

- A compact binary format for quick serialization and deserialization, ideal for performance-critical applications.
- Schema evolution capabilities, enabling developers to modify their data structures without affecting the integrity of existing serialized data.
- Strongly typed data structures, ensuring data exchanged between services adhere strictly to the specified schema, thus reducing runtime errors due to data discrepancies.

Consider the following illustrative protobuf definition:

```
// event.proto
syntax = "proto3";

// Namespace for this file
package example;

enum Type {
    CREATED = 0;
}

// Strongly-typed event. The numbered attributes of each field are used to
```

```
// encode the field's position in the serialized message.
message Event {
    Type id = 1;
    string name = 2;
    // Schema evolution allows for new fields to be
    // added, while maintaining backwards compatibility
    optional string description = 3; // New field added at a later revision
}
```

This definition specifies an event data-type with specific attributes. Central to processing this definition across different languages is the `protoc` compiler. One of its standout features is the extensible plugin architecture. As highlighted in **Chapter 1**, plugins are pivotal in enhancing the capabilities of software tools. With the aid of [protoc plugins](#), developers not only have the flexibility to generate code suitable for a wide array of programming languages, but they can also craft their own plugins. An example is seen in `QtGrpc`, where a custom plugin will translate the proto file to `c++` classes, which seamlessly integrate into the Qt ecosystem. Additionally, plugins like [protoc-gen-doc](#) extend the capability of `protoc` by offering the convenience of producing documentation directly from inline comments within the proto file.

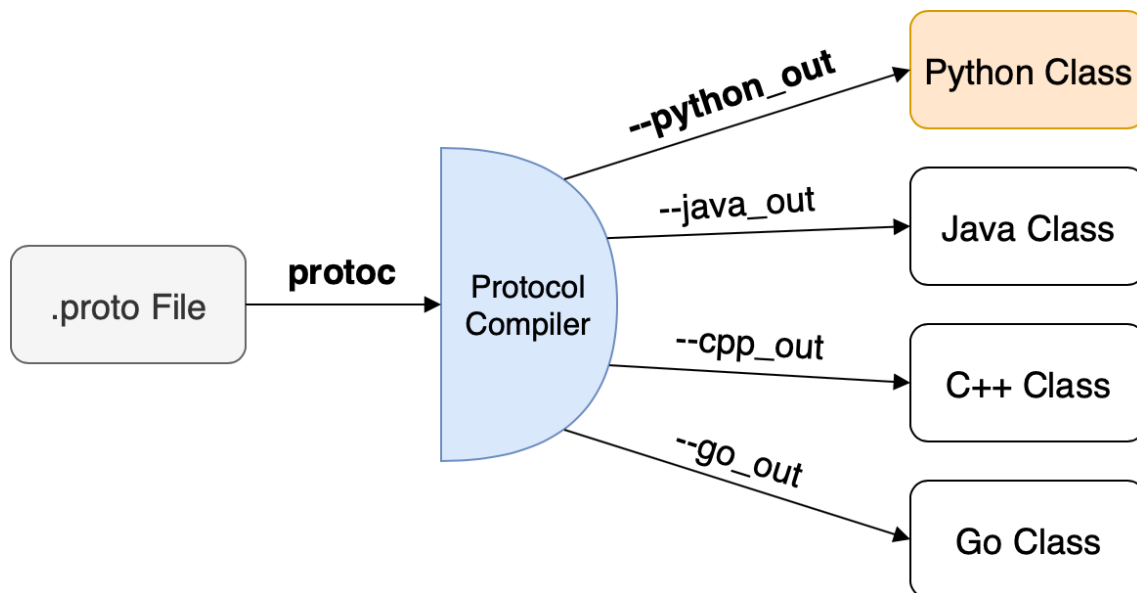


Figure 13: protoc extensions

For instance, to compile the protobuf file into a Python interface, use:

```
protoc --python_out=./build event.proto
```

The code outputted by protobuf may appear abstract as it doesn't directly provide methods for data access. Instead, it utilizes metaclasses and descriptors. Think of descriptors as guiding the overall behavior of a class with its attributes.

To further illustrate, here's an integration with our generated class:

```
# event.py
import build.event_pb2 as E
from google.protobuf.json_format import MessageToJson

event = E.Event(id=E.Type.CREATED, name='test', description='created event!')

def print_section(title, content):
    print(f"{title}:\n{'-'*len(title)}\n{content}\n")

print_section("Python structure", event)
print_section("Serialized structure", event.SerializeToString())
print_section("JSON structure", MessageToJson(event))
```

Executing this yields:

```
Python structure:
-----
id: CREATED
name: "test"
description: "created event!"

Serialized structure:
-----
b'\x08\x01\x12\x04test"\x0ecreated event!'
```

```
JSON structure:
-----
{
  "id": "CREATED",
  "name": "test",
  "description": "created event!"
}
```

This demonstration highlights protobuf's capabilities. It illustrates the simplicity with which a type can be created, serialized into a compact binary format, and its contents used by the application. The benefits of protobuf's efficiency become even more pronounced when contrasted with heftier formats like JSON or XML.

### 2.3.2 gRPC Core Concepts

**Channels** in gRPC act as a conduit for client-side communication with a gRPC service. A channel represents a session which, unlike HTTP/1.1, remains open for multiple requests and responses.

**Services** in gRPC define the methods available for remote calls. They're specified using the protobuf language and serve as API between the server and its clients.



**Stubs** are the client-side representation of a gRPC service. They provide methods corresponding to the service methods defined in the .proto files.

**Calls and Streams** in gRPC provide the functionality for continuous data exchange between the client and the server. The four primary RPC types are:

1. **Unary Calls:** This is the most common type, similar to a regular function call where the client sends a single request and gets a single response.
2. **Server Streaming:** The client sends a single request and receives multiple responses. Useful when the server needs to push data continuously after processing a client request.
3. **Client Streaming:** The client sends multiple requests before it awaits the server's single response. Useful in scenarios like file uploads where the client has a stream of data to send.
4. **Bidirectional Streaming:** Both client and server send a sequence of messages to each other. They can read and write in any order. This is beneficial in real-time applications where both sides need to continuously update each other.

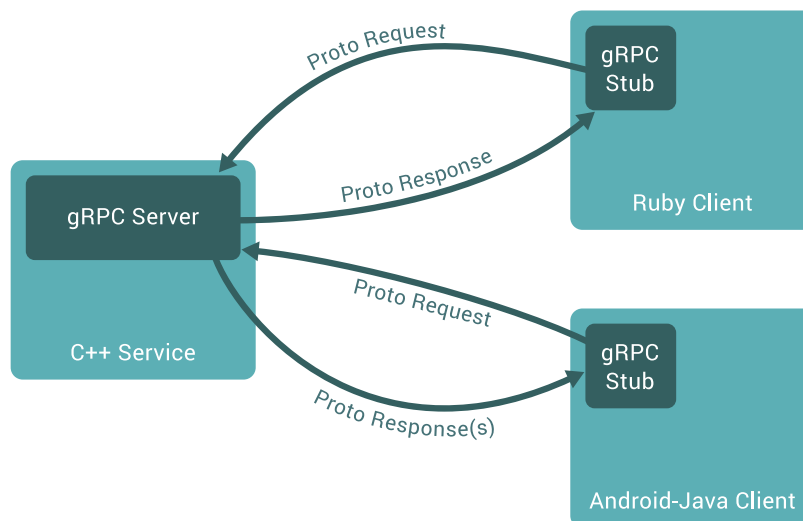


Figure 14: gRPC high-level overview

### 2.3.3 Performance

Traditional HTTP protocols don't support sending multiple requests or receiving multiple responses concurrently within a single connection. Each request or response would necessitate a fresh connection.

However, with the advent of HTTP/2, this constraint was addressed. The introduction of the binary framing layer in HTTP/2 enables such request/response multiplexing. This ability to handle streaming efficiently is a significant factor behind gRPC's enhanced performance.

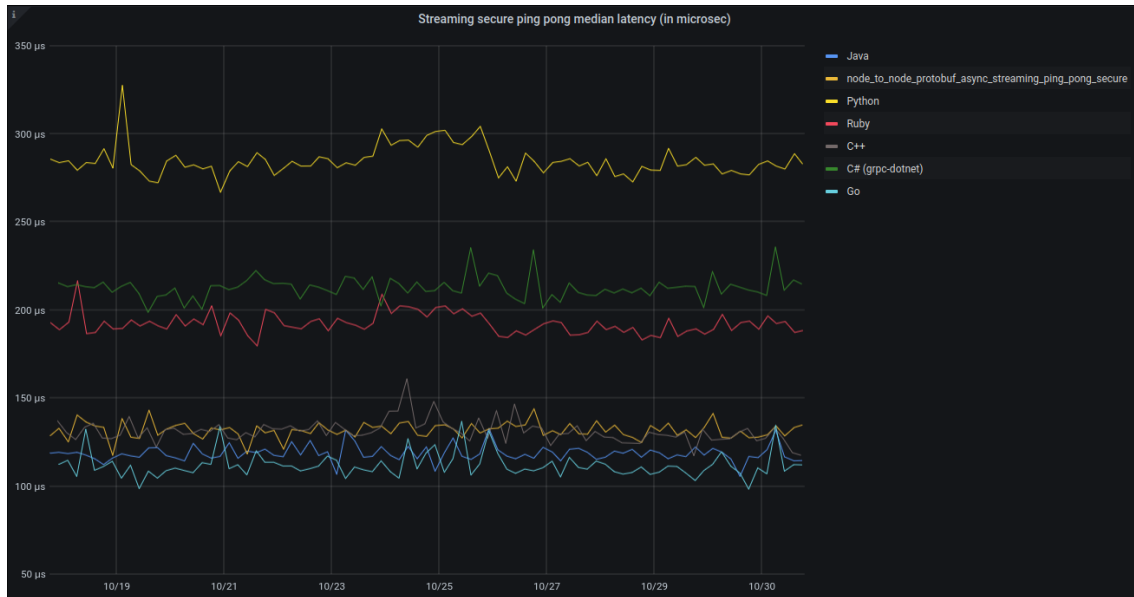


Figure 15: gRPC streaming [benchmarks](#)

As depicted in [Figure 15](#), the performance benchmarks highlight the latencies associated with various gRPC clients when interfaced with a C++ server implementation. The results reveal a spectrum of latencies, with some as low as 100 microseconds (us) and others peaking around 350 us. Importantly, these tests measure the Round Trip Time (RTT), representing the duration taken to send a message and subsequently receive a response.

## 2.3 The Qt Framework

[Qt](#), emerging in the early 90s from Trolltech in Norway, stands out in the landscape of software development for its robust toolkit that simplifies the creation of graphical user interfaces (GUIs) that are platform-agnostic. With Qt, the same codebase can be deployed on multiple operating systems such as Linux, Windows, macOS, Android or embedded systems with little need for modification, streamlining the development process significantly.

After its acquisition by Nokia in 2008, Qt has continued to thrive under the guidance of The Qt Company, which is responsible for its ongoing development and maintenance. Qt provides two licensing options: the GNU (L)GPL license, promoting a community-driven approach, and a commercial license for developers who prefer to maintain exclusive control over their software.

Qt's versatility extends beyond its C++ core, offering language bindings for additional programming languages, with Python being a notable example through [Qt for Python / PySide6](#). This extension facilitates rapid development by allowing the integration of Qt's powerful C++ modules within Python's flexible scripting environment.

### 2.3.1 Core Techniques

Central to Qt is its [Signals & Slots](#) mechanism, which can be thought of as a flexible implementation of the observer pattern. A 'signal' represents an observable event, while a 'slot' is comparable to an observer that can react to that event. This design allows for a many-to-many relationship meaning any

signal may be connected to any number of slots. This concept has been fundamental to Qt since its first release in 1994 and the concept of signals and slots is so compelling that it has become a part of the computer science landscape.

Essential for the reactive nature of graphical user interfaces, signals & slots enables a program to handle user-generated events such as clicks and selections, as well as system-generated events like incoming data transmissions. Qt employs the moc (Meta Object Compiler), a tool developed within the Qt framework, which seamlessly integrates these event notifications into Qt's event loop.

Let's examine a code example to understand the practicality and simplicity of signals & slots in Qt:

```
// signals_and_moc.cpp
#include <QCoreApplication>
#include <QRandomGenerator>
#include <QTimer>
#include <QDebug>

// To use signals and slots, we must inherit from QObject.
class MyObject : public QObject
{
    Q_OBJECT // Tell MOC to target this class
public:
    MyObject(QObject *parent = 0) : QObject(parent) {}
    void setData(qsizetype data)
    {
        if (data == mData) { // No change, don't emit signal
            qDebug() << "Rejected data: " << data;
            return;
        }
        mData = data;
        emit dataChanged(data);
    }
signals:
    void dataChanged(qsizetype data);
private:
    qsizetype mData;
};
```

In this code snippet, we create a class integrated with Qt's [Meta-Object System](#). To mesh with the Meta-Object System, three steps are critical:

1. Inherit from `QObject` to gain meta-object capabilities.
2. Use the `Q_OBJECT` macro to enable the **MOC's** code generation.
3. Utilize the `moc` tool to generate the necessary meta-object code, facilitating signal and slot functionality.

The `MyObject` class emits the `dataChanged` signal only when new data is set.

```

// signals_and_moc.cpp
void receiveOnFunction(qsizetype data)
{ qDebug() << "1. Received data on free function: " << data; }

class MyReceiver : public QObject
{
    Q_OBJECT
public slots:
    void receive(qsizetype data)
    { qDebug() << "2. Received data on member function: " << data; }
};

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    MyObject obj;
    MyReceiver receiver;

    QTimer timer; // A timer to periodically change the data
    timer.setInterval(1000);
    QObject::connect(&timer, &QTimer::timeout, [&obj]() {
        obj.setData(QRandomGenerator::global()→bounded(3));
    });
    timer.start();

    // Connect the signal to three different receivers.
    QObject::connect(&obj, &MyObject::dataChanged, &receiveOnFunction);
    QObject::connect(&obj, &MyObject::dataChanged, &receiver, &MyReceiver::receive);
    QObject::connect(&obj, &MyObject::dataChanged, [](qsizetype data) {
        qDebug() << "3. Received data on lambda function: " << data;
    });

    // Start the event loop, which handles all published events.
    return QApplication::exec();
}

```

The given code snippet demonstrates Qt's signal and slot mechanism by setting up a series of connections between a signal and various slot types using `QObject::connect`. In a typical compilation scenario for such a Qt application on a Linux system, you'd run a command like:

```

g++ -std=c++17 -I/usr/include/qt6 -I/usr/include/qt6/QtCore -lQt6Core -fPIC -o
↳ build/signals_and_moc signals_and_moc.cpp

```

However, this direct approach to compile the file will be met with linker errors, a sign that something is missing:

```

/usr/bin/ld: /tmp/ccbsQpD0.o: in function `main':
signals_and_moc.cpp:(.text+0x2e4): undefined reference to `MyObject::dataChanged(long
↳ long)'
/usr/bin/ld: /tmp/ccbsQpD0.o: in function `MyObject::MyObject(QObject*)':
signals_and_moc.cpp:(.text._ZN8MyObjectC2EP7QObject[_ZN8MyObjectC5EP7QObject]+0x26):
↳ undefined reference to `vtable for MyObject'

```

These errors indicate that we have a missing implementation for our `dataChanged` function, which handles the signaling mechanism of our object. This is where the meta-object-compiler comes into play and provides the needed (boilerplate) implementation to this function. The moc is integral to Qt's signal and slot system, and without it, the necessary meta-object code that facilitates these connections is not generated. To correct the compilation process, moc must be run on the source files containing the `Q_OBJECT` macro before the final compilation step.

```

# assuming moc is in $PATH
moc signals_and_moc.cpp > gen.moc
# include the generated file at the end of our source
echo '#include "gen.moc"' >> signals_and_moc.cpp
# re-compile

```

A closer look at the output from moc uncovers the previously missing pieces, providing the essential implementation for our `dataChanged` signal and clarifying how moc underpins the signal's integration.

```

// SIGNAL 0
void MyObject::dataChanged(qsizetype _t1)
{
    void *_a[] = { nullptr, const_cast<void*>(&reinterpret_cast<const
↳ void*>(std::addressof(_t1))) };
    QMetaObject::activate(this, &staticMetaObject, 0, _a);
}

```

After running the corrected compilation and executing the application, the output could look like this:

```

./build/signals_and_moc
1. Received data on free function: 0
2. Received data on member function: 0
3. Received data on lambda function: 0
Rejected data: 0

```

### 2.3.2 Graphics

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 components.

A significant distinction between Widgets and QML lies in their rendering approach. 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 major release in the Qt series, introduced a series 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 is to guarantee determined performance consistency across a diverse range of graphic backends. The introduction of **QRHI** underscores Qt's steadfast dedication to strengthen its robust cross-platform capabilities, aiming to create a unified experience across various graphic backends.

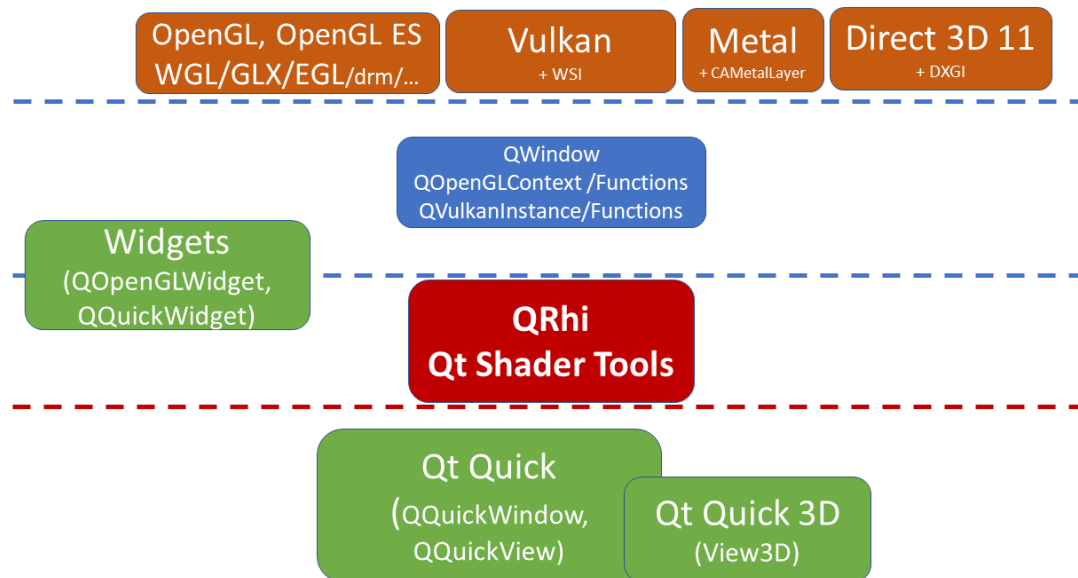


Figure 16: Qt Rendering Hardware Interface **QRHI**

Figure 16 presents the layered architecture of the rendering interface. At its base, it supports native graphic APIs such as OpenGL, Vulkan or Metal. 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.

Further up the hierarchy, the QtQuick and QtQuick3D modules showcase Qt's progression in graphics rendering.

In essence, Qt delivers a comprehensive framework for designing user interfaces that cater to the diverse array of existing platforms. It provides a broad spectrum of configuration and customization options and even enabling developers to tap into lower-level abstractions for crafting custom extensions tailored to specific project requirements—all while preserving *cross-platform functionality*. Beyond the foundational modules geared toward GUI development, Qt also offers a rich suite of additional modules. These include resources for localizing applications with [qttranslations](#), handling audio and video via [qtmultimedia](#), and various connectivity and networking options. Modules such as [qtconnectivity](#), [qtwebsockets](#), and the latest addition, [qtgrpc](#), facilitate the integration of Qt into an even wider range of systems.

### 2.3.3 QtGrpc and QtProtobuf

The [QtGrpc](#) module, which is in a technical preview stage as of Qt version 6.6, represents an innovative addition to Qt's suite. It provides plugins for the `protoc` compiler, which we touched upon in section 3.2. These plugins are designed to serialize and deserialize Protobuf messages into Qt-friendly classes, facilitating a smooth and integrated experience within the Qt framework. This integration significantly reduces the need for additional boilerplate code when working with Protobuf and gRPC.

To enhance our previous `event.proto` definition with Qt features, let's compile it using the `qtprotobufgen` plugin as follows:

```
# assuming qtprotobufgen is in $PATH
protoc --plugin=protoc-gen-qtprotobuf=qtprotobufgen --qtprotobuf_out=./build
↪ event.proto
```

This command will generate the `build/event.qpb.h` and `build/event.qpb.cpp` files. A review of the generated header demonstrates how the plugin eases the integration of Protobuf definitions within the Qt framework:

```
// build/event.qpb.h
~~~
class Event : public QProtobufMessage
{
    Q_GADGET
    Q_PROTOBUF_OBJECT
    Q_DECLARE_PROTOBUF_SERIALIZERS(Event)
    Q_PROPERTY(example::TypeGadget::Type id_proto READ id_proto WRITE setId_proto
↪    SCRIPTABLE true)
```

```

    Q_PROPERTY(QString name READ name WRITE setName SCRIPTABLE true)
    Q_PROPERTY(QString description READ description_p WRITE setDescription_p)
    Q_PROPERTY(bool hasDescription READ hasDescription)

public:
    using QtProtobufFieldEnum = Event_QtProtobufNested::QtProtobufFieldEnum;
    Event();
    ~Event();
    ~~~

```

In the provided C++ header file, the `Event` class inherits from `QProtobufMessage` to leverage Qt's meta-object system for seamless serialization and deserialization of Protocol Buffers. This class definition enables the automatic conversion of data types defined in .proto files to Qt-friendly types. Such integration allows developers to use these types with ease within both the Qt C++ environment and QML, facilitating rapid and adaptable integration with gRPC services, irrespective of the server's implementation language.

Furthermore, QtGrpc extends the functionality of Protocol Buffers within the Qt framework by providing essential classes and tools for gRPC communication. For example, `QGrpcHttp2Channel` offers an *HTTP/2* channel implementation for server communication, while `QGrpcCallReply` integrates incoming messages into the Qt event system. This synergy between QtGrpc and protocol buffers streamlines client-server interactions by embedding Protocol Buffer serialization within Qt's event-driven architecture.



## 3 Remote Plugins

### 3.1 Introduction

As outlined in the previous sections the problem should be clear, and the tools that we will use to solve it should be known. Since the goal is to enable native development experience, unflexible techniques as enforcing users to compile Qt in a seperate namespace are not applicable. Furthermore we want a stable user experience that works at least on all available desktop platforms.

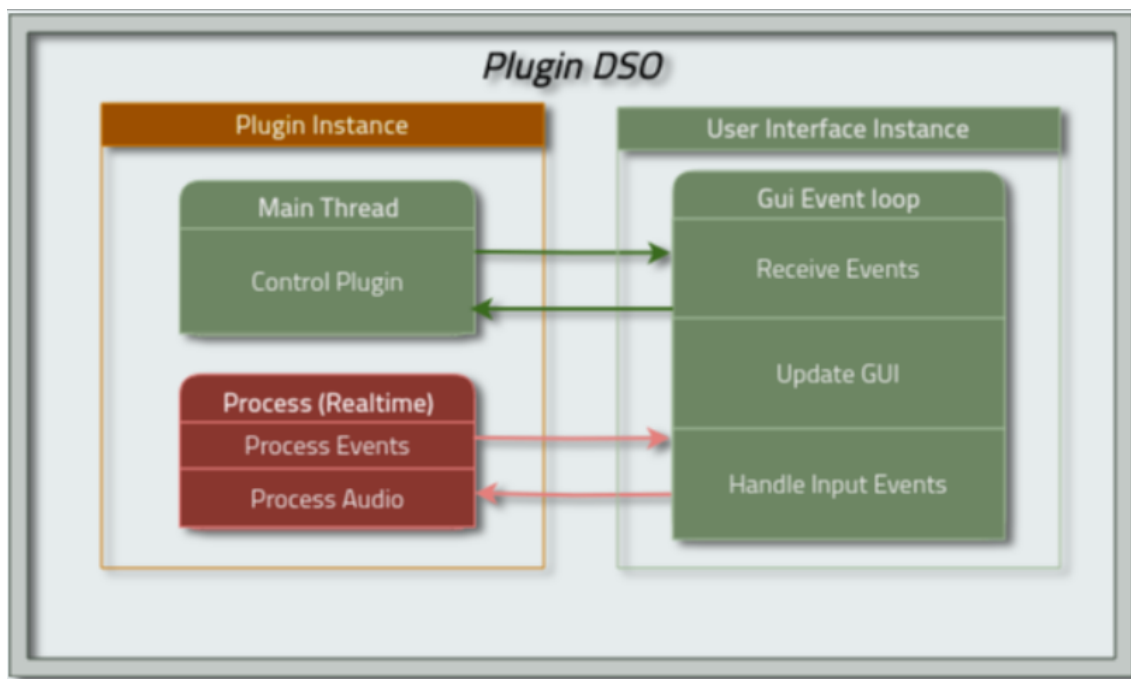


Figure 17: Traditional plugin architecture

Figure 17 outlines traditional audio plugin architectures, when used with graphical user interfaces. As we know, multiple instances will be created out of our DSO, hence enforcing reentrancy. The traditional approach is to start the event loop of the user interface inside a dedicated thread, resulting in a minimum of 2 threads that 'live' inside the DSO's process. This technique can be seen in [JUCE](#), which is a well-known library for creating plugin-standard agnostic plugins, as well as in various other open source implementation.

Since we can't start Qt's event loop inside a dedicated thread, we will simply execute the plugin GUI inside a separate process. Starting it inside a separate process could be seen as expensive or being slow and hard to incorporate because of the various issues with IPC<sup>10</sup>, but in my opinion it's equally as good. First of all modern computers should be able to handle another process with ease and secondly we will gain safety with this approach. To elaborate, let's look at the case where we have GUI and the realtime thread within the same process. A crash in the user interface would then also crash the realtime thread. However when the GUI crashes launched as separate process the realtime thread wouldn't even notice it at all. The implementation of IPC or RPC techniques is a bit more tricky and indeed result in some work. But the performance shouldn't be affected critically by it. The messages we have to

<sup>10</sup>Inter Process Communication

share between audio and gui thread are very small in size, as some parameter changes or note events. They are usually just a few bytes large. When thinking about sharing the audio data itself techniques such as opening shared memory pools could be employed which would make it equally as fast as it would be within the same thread. Although all at the cost of being way more 'expensive' to implement. Furthermore GUIs don't have to operate at the speed of the realtime thread. We humans

### **3.2 Remote Control Interface**

### **3.3 QtGrpc Client Implementation**

## **4 Conclusions**

### **Integrity**

### **Performance**

### **Future Work**

## **Bibliography**

- Apple. 2023. "Audio Unit V3 Plug-Ins." Apple. October 23, 2023. [https://developer.apple.com/documentation/audiotoolbox/audio\\_unit\\_properties](https://developer.apple.com/documentation/audiotoolbox/audio_unit_properties).
- Joseph Timoney, Rudi Villing, Victor Lazzarini. 2015. "Approaches for Constant Audio Latency on Android." December 15, 2015. <https://mural.maynoothuniversity.ie/6664/>.
- Lakhani, "Maharaja. 2014. "Protocol Buffers: An Overview (Case Study in c++)." "
- Popić & Bojan Mrazovac & Dražen Pezer & Nikola Teslić, "Srđan. 2016. "Performance Evaluation of Using Protocol Buffers in the Internet of Things Communication."
- Scavone, Gary P. 2002. "RtAudio: A Cross-Platform c++ Class for Realtime Audio Input/Output." October 1, 2002. <https://ccrma.stanford.edu/~gary/papers/icmc2002c.pdf>.
- Vandevoorde, Daveed. 2006. "Plugins in c++." September 7, 2006. <https://www.open-std.org/JTC1/sc22/wg21/docs/papers/2006/n2074.pdf>.