Headless Audio-Plugins

Integrating CLAP and Qt for Enhanced Development Solutions

Dennis Oberst

04 December 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, an open-source and high performance Remote Procedure Call framework, the research explores new possibilities for remote 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 libraries not only provide 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

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 workflow. 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 subject solely due to the limitations of their chosen toolkit for the next project.

Therefore, Qt, a cross-platform framework for crafting GUIs, comes to mind when considering the development of an audio plugin UI[1] 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 certainly suggest the relevance and potential of Qt as a feasible option 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:

```cpp
// 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;
}
```

[1] User Interface

```cpp
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 QGuiApplication. These classes, which manage Qt's event loop through `Q*Application::exec()`, are static:

```cpp
// qtbase/src/corelib/kernel/qcoreapplication.h
static QCoreApplication *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 QGuiApplication(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.

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* [9], or system-specific extensions found in frameworks like Qt.

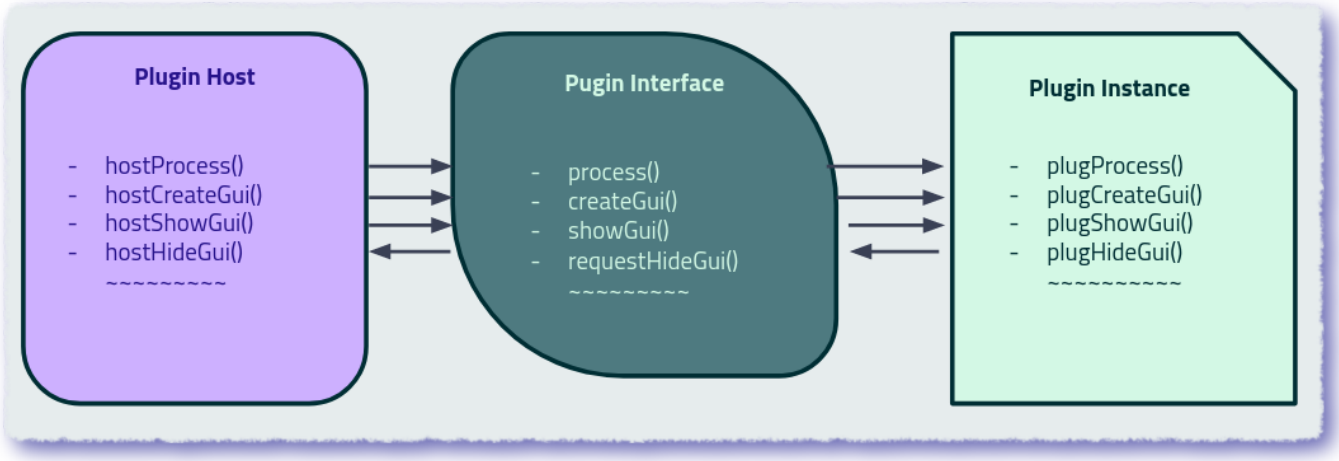

Figure 1: basic plugin architecture

fig. 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 **CL**ever **A**udio **P**lugin (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[2], is a reusable object that exports a table of symbols

[2]Dynamic Shared Object

(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:

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

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

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

This code defines a minimal shared library. After including the required standard-headers, we define a compile time directive that is used to signal the visibillity of the exported symbols. Windows and Unix based system differ here. On Windows with MSVC the symbols are *not* exported by default, and require explicit marking with `__declspec(dllexport)` . On Unix based system we use the visibillity attribute. Since by default all symbols are exported on these platforms, rendering this attribute seemingly redundant, it remains advantageous to maintain clarity. This would also allows us to control the visibillity in the linking step by simply using the linker flag `-fvisibility=hidden` to hide all symbols. 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, lets write a minimal build script and run our `simplehost` executable.

```
#!/bin/bash
# simplelib.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:

```
./simplelib
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 **I**nterface, 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.

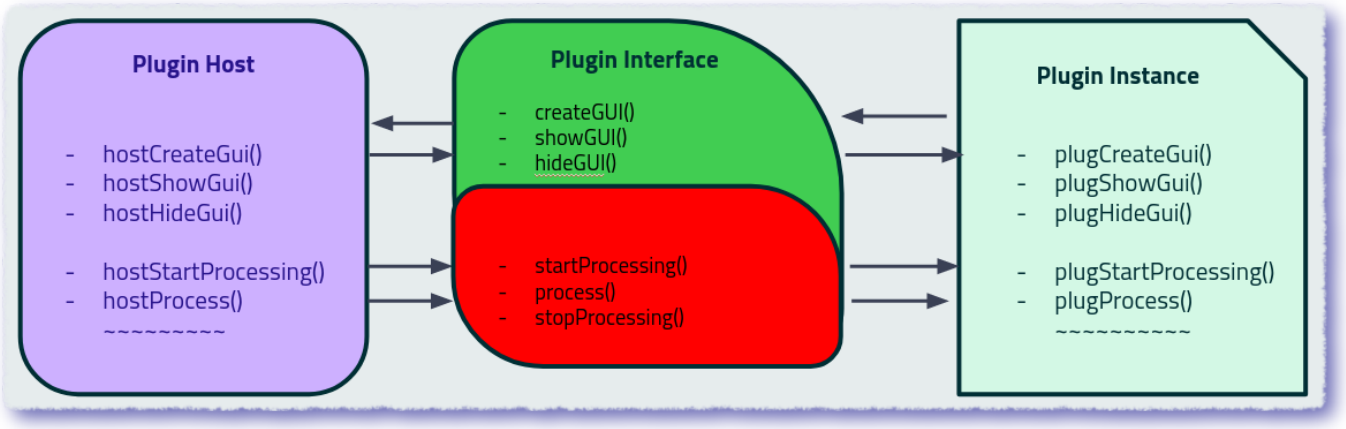

Figure 2: basic audio plugin architecture

fig. 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.[3]

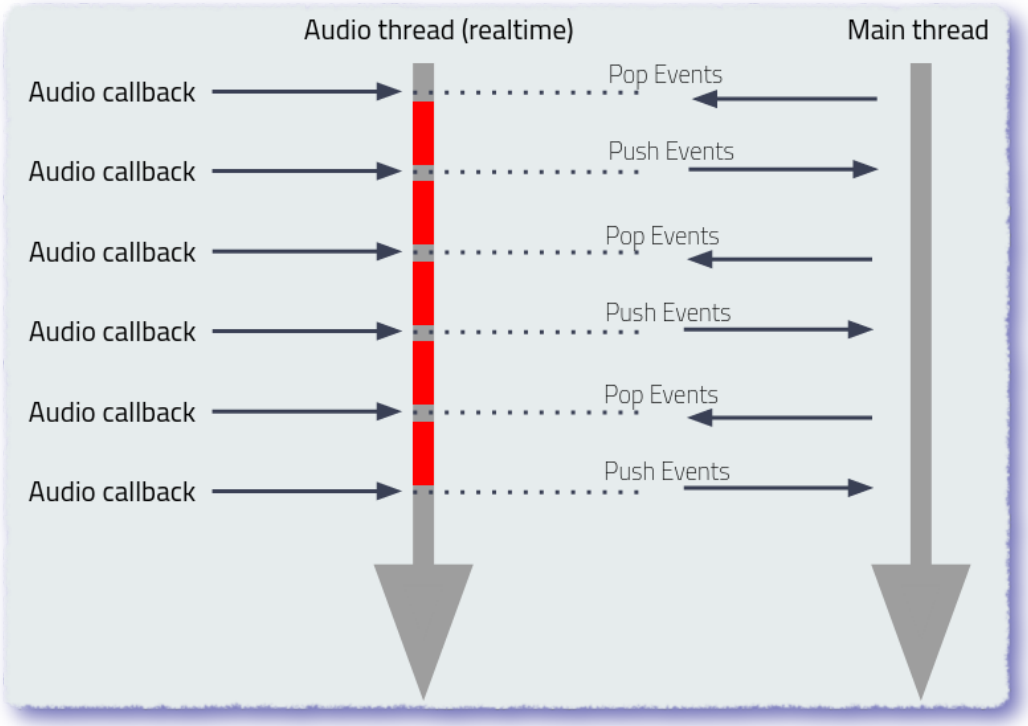

Figure 3: Realtime overview

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

[3]Inspired by: NppNow 2023, What is Low Latency C++? (Part 2) - Timur Doumler

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[7]. Keeping in mind the simple but crucial reminder:

> Don't miss your Deadline!

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

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[4].

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. [4]

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[5] queues or ring-buffers to complement this.

[4] Taken from: ADC 2019, Fabian Renn-Giles & Dave Rowland - Real-time 101
[5] First In First Out

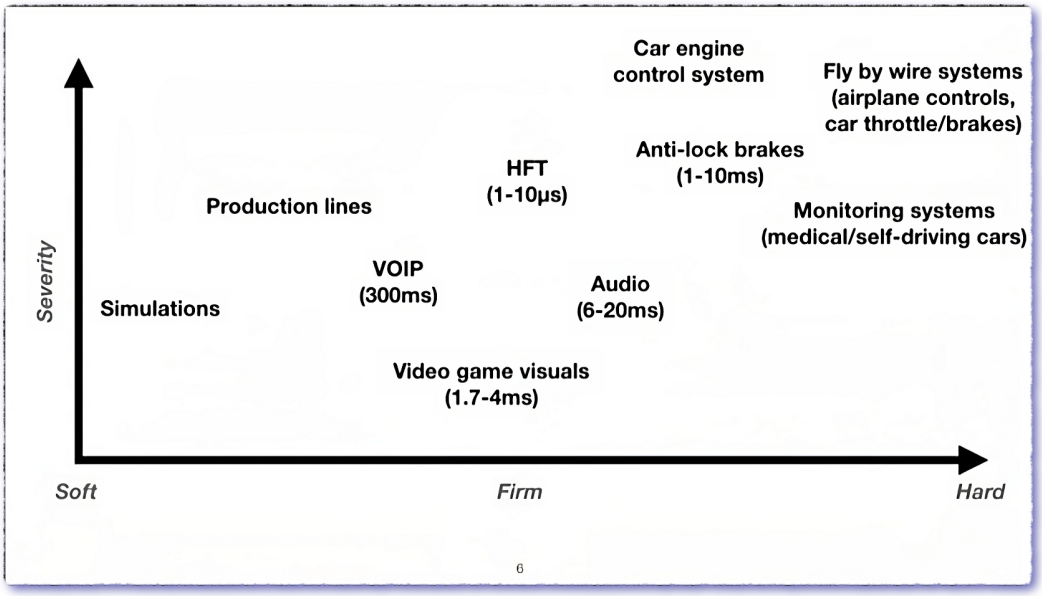

Figure 4: Realtime ranking

fig. 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 **dead**line. Audio sits in the middle, where a missed deadline compromises professional utility but doesn't pose dire threats. Video game rendering bears even lesser severity, as occasional frame drops don't render the product ineffectual and are relatively commonplace.

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
CLAP	Clever Audio Plugin	Bitwig & U-he	.clap	Windows, MacOS & Linux	2022	MIT
VST/VST3	Virtual Studio Technology	Steinberg	.dll, .vst, .vst3	Windows, MacOS & Linux	2017	GPLv3, Steinberg Dual License
AAX	Avid Audio Extension	Pro Tools (Avid)	.aax	Windows & MacOS	2011	Approved Partner
AU	Audio Units	Apple macOS & iOS	.AU	MacOS	"Cheetah" 2001	Custom License Agreement

Certain standards cater specifically to particular platforms or programs. For instance, Apple's **AU** is seamlessly integrated with their core audio SDK[6] [1]. Similarly, Avid's **AAX** is designed exclusively for plugin compatibility with the Pro Tools DAW[7]. On the other hand, standards like the **VST3** SDK are both platform and

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

program independent, and it's currently among the most popular plugin standards. Additionally, the newly introduced **CLAP** standard is also gaining traction.

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

Figure 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[8] 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.

[8] Midi Polyphonic Expression

```c
// <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.

```c
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.

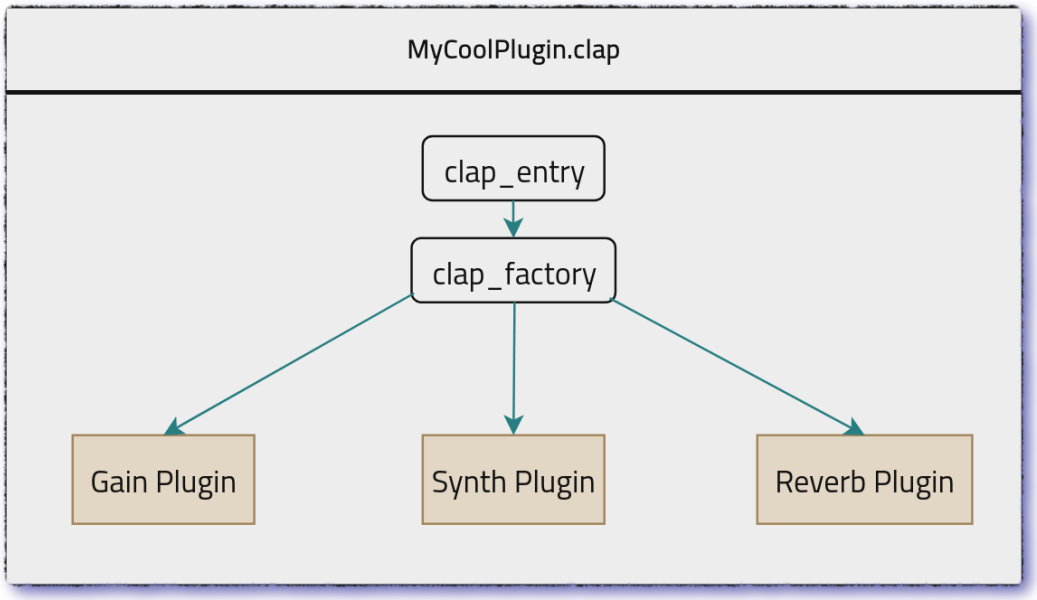

Figure 6: CLAP factory

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* routines. 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.

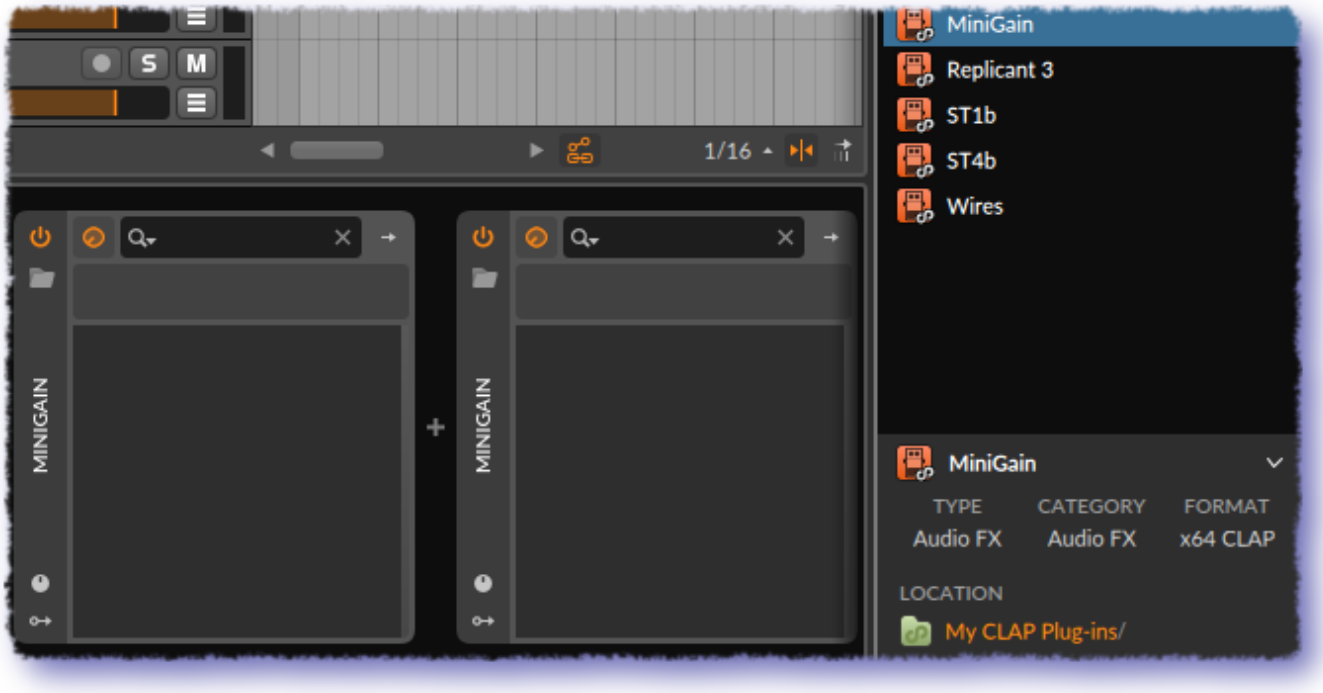

Figure 7: MiniGain hosted

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 stand-alone 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**

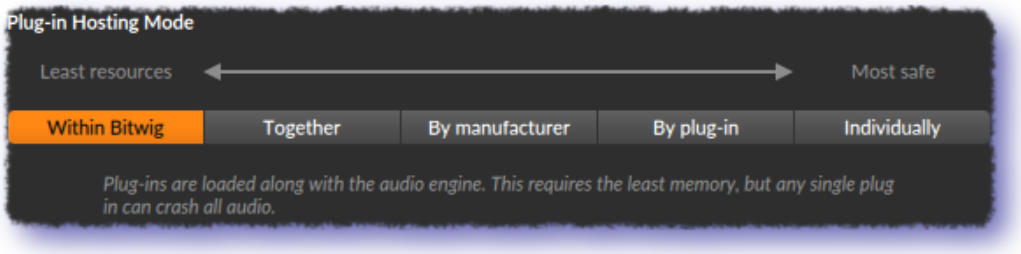

Figure 8: Bitwig Hosting

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 Bitwigs 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 metadata about them. The initialization function for the extension is defined as follows:

```
void MiniGain::initializeExtParams() {
    mExtParams = {
        .count = [](const clap_plugin*) -> 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.

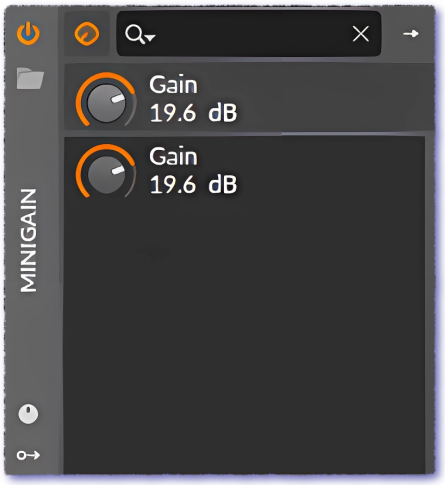

Figure 9: MiniGain hosted

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.

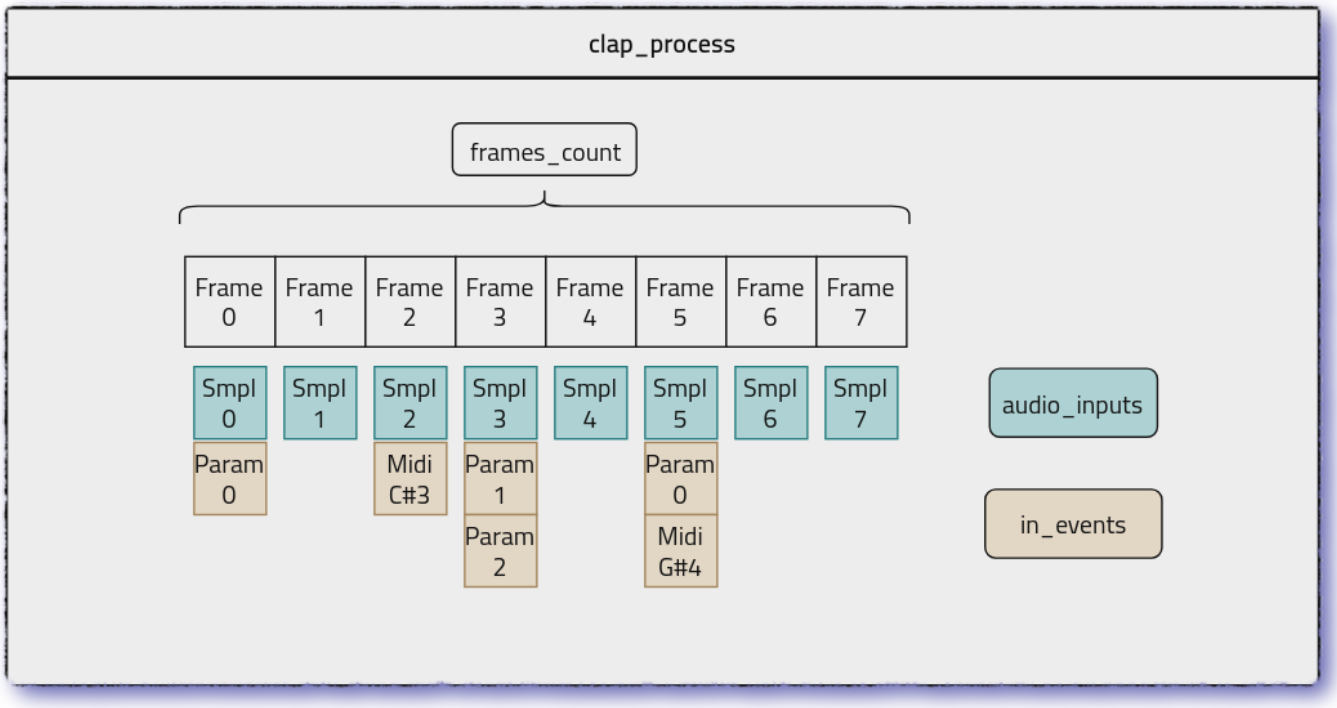

Figure 11: CLAP process

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) // Consumned 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[8].

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[5], 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[8].

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.

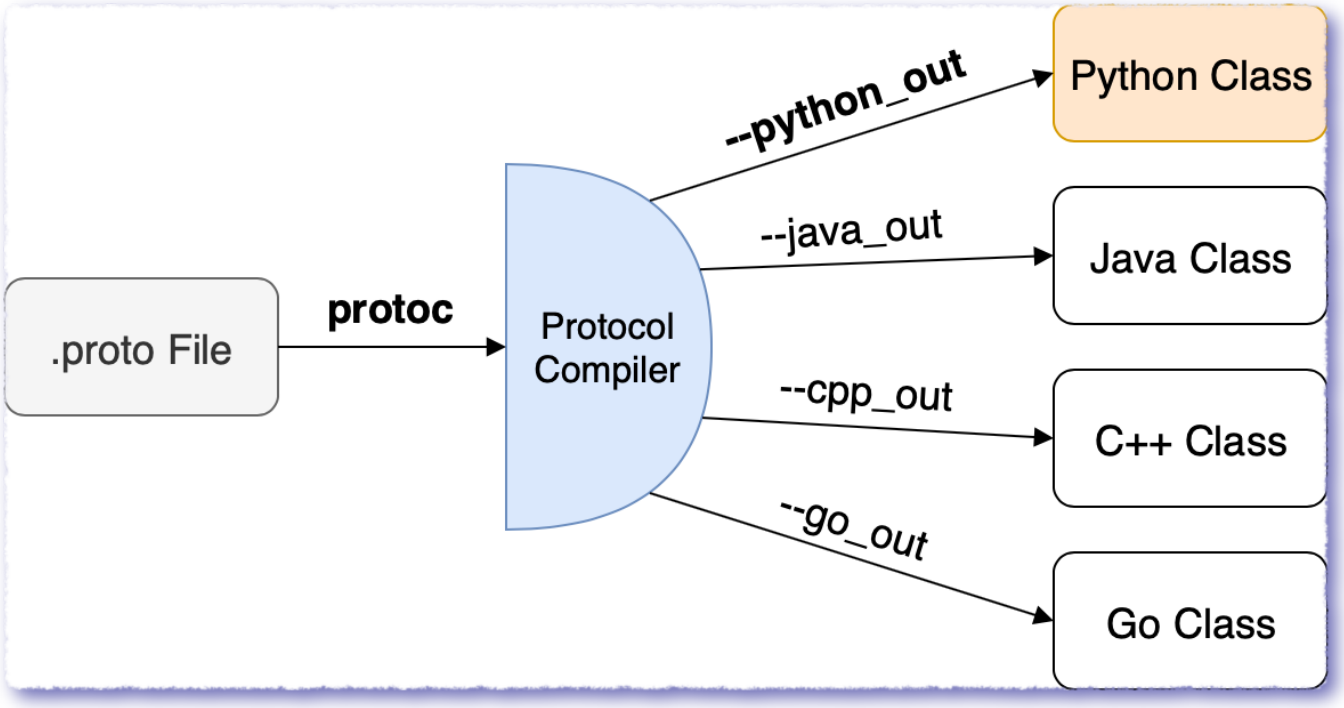

Figure 13: protoc extensions

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.

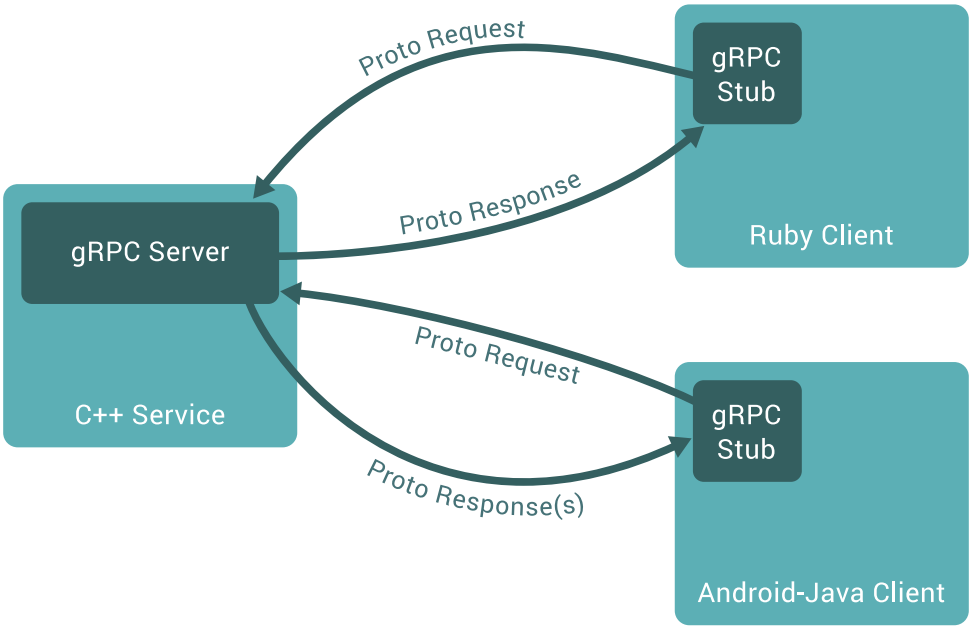

Figure 14: gRPC high-level overview

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.

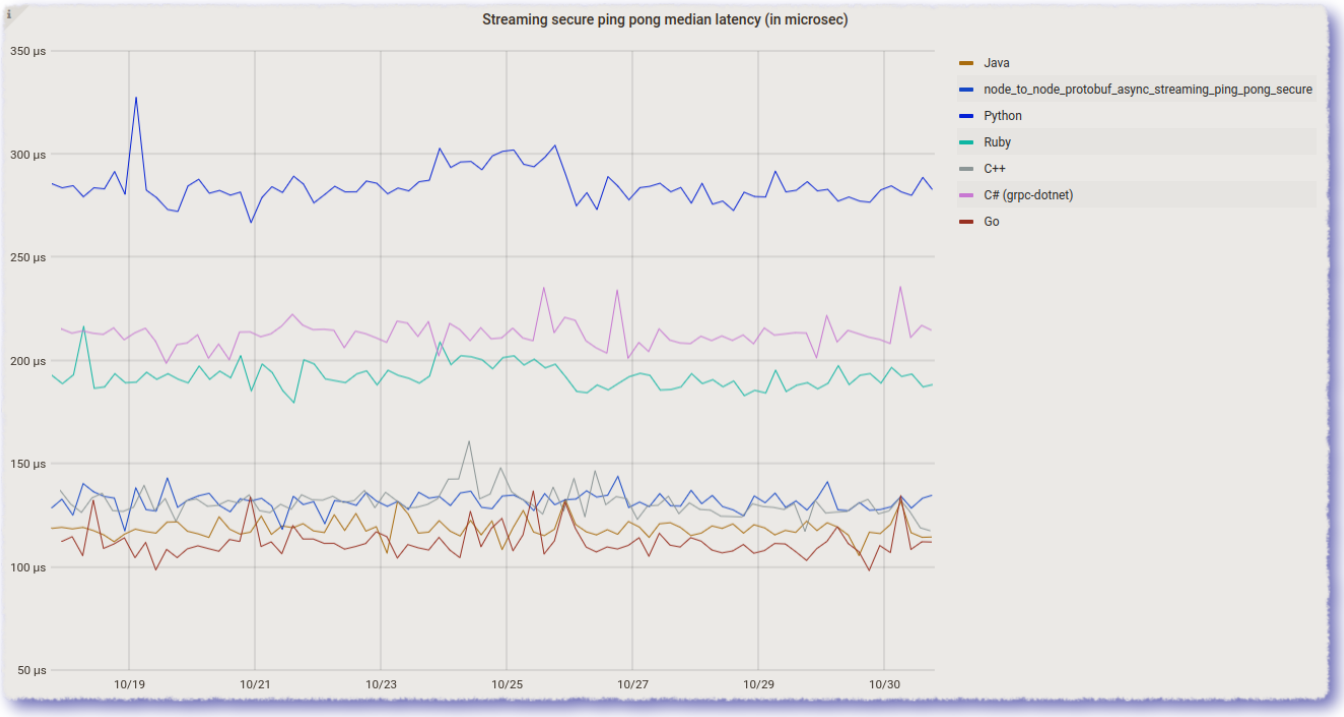

Figure 15: gRPC streaming benchmarks

As depicted in *fig.* 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[])
{
    QCoreApplication 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 QCoreApplication::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/ccbsQpDO.o: in function `main':
signals_and_moc.cpp:(.text+0x2e4): undefined reference to `MyObject::dataChanged(long long)'
/usr/bin/ld: /tmp/ccbsQpDO.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.

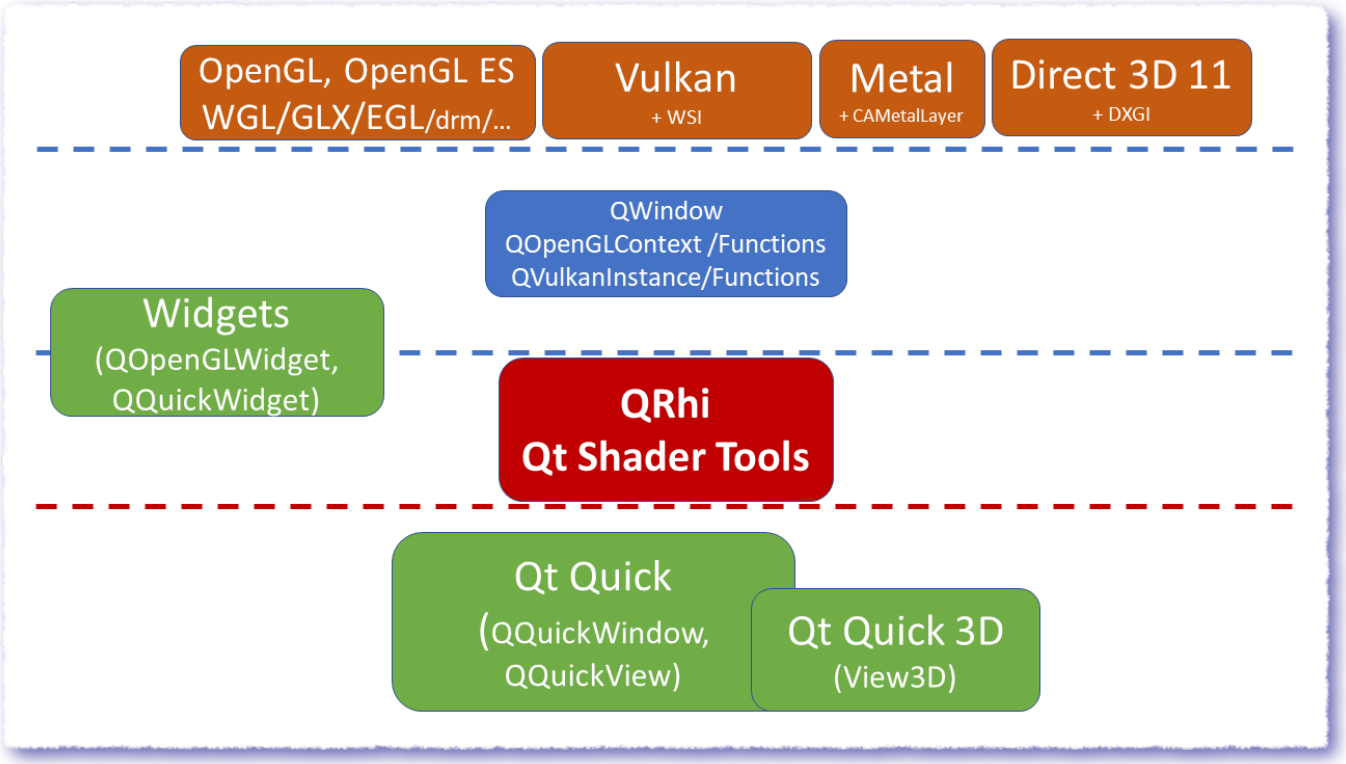

Figure 16: Qt Rendering Hardware Interface RHI

fig. 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.

Chapter 3: Related Work

This section examines existing research and current advancements in the field of audio plugin development. Given the focused scope of this thesis, attention is centered on the most relevant aspects, particularly the use of inter-process communication in the remote control of plugins.

Clap-Plugins

The project most directly related to this thesis comes from the CLAP team. In the official repository of the *free-audio* organization, which is the home of the CLAP standard, there is a project named clap-plugins. This project serves as a hands-on example of how to implement a CLAP plugin. The plugins in this project use Qt for their GUIs. Alexandre Bique, the project's maintainer and the lead developer of CLAP, has recognized the challenges of integrating Qt GUIs with audio plugins and has offered solutions to these challenges. The first suggestion is to avoid dynamically linking any dependencies to the plugin, to keep the plugin self-contained. Following this, the project presents two different strategies for integrating Qt with CLAP:

> We can use two different strategies:
>
> `local: statically link everything`
>
> `remote: launch the GUI in a separate child process`

The *local* method involves building a custom version of Qt in a separate namespace, which is achieved by using the compile flag `QT_NAMESPACE=<uniqueId>` . This step is crucial to prevent conflicts when multiple Qt applications run in the same process and namespace. The *remote* method involves starting the GUI in a separate process and communicating with it via inter-process communication techniques.

The initial approach of using a separate namespace for Qt still presents challenges, particularly on Linux systems. Even with Qt compiled under a unique namespace, issues arise when multiple instances of the event loop are running with the glib event loop, which is the default on Linux desktops. Loading a plugin in this manner, even with a static Qt build under a unique namespace, into an environment where Qt is already running a glib event loop, might result in an error like this:

```
GLib-CRITICAL **: 00:25:07.633: g_main_context_push_thread_default: assertion 'acquired_context' failed
```

The definitive explanation for this error remains unclear. The problem might stem from glib retaining some internal state, or perhaps there's a namespace mismatch occurring. Whether intentionally or not, the *clap-plugins* implementation explicitly employs the *xcb* backend:

```
// plugins/gui/local-gui/factory.cc
#ifdef Q_OS_LINUX
   ::setenv("QT_QPA_PLATFORM", "xcb", 1);
#endif
```

This solution bypasses the problem, but at the expense of not using the default event-system, which is generally more robust and well-tested. Additionally, this method necessitates manually building Qt in a separate and unique namespace for each plugin, adding complexity to the development process.

For the *remote* strategy, the implementation utilizes a custom-built IPC mechanism. On POSIX systems, it employs a unix domain sockets approach, and on Windows, it uses I/O completion ports (IOCP). Events initiated by the host are eventually transmitted to the GUI process:

```
// plugins/gui/remote-gui-proxy.cc

   bool RemoteGuiProxy::show() {
      bool sent = false;
      messages::ShowRequest request;
```

```
    _clientFactory.exec(
        [&] { sent = _clientFactory._channel->sendRequestAsync(_clientId, request); });

    return sent;
  }
```

This function is responsible for dispatching the event that triggers the GUI window's visibillity. The events, initiated by the host application, are received and processed by the GUI process through a switch statement that handles all the various events:

```
// plugins/gui/gui.cc
~~~
      case messages::kShowRequest: {
         messages::ShowRequest rq;
         messages::ShowResponse rp;
         msg.get(rq);
         if (c)
            rp.succeed = c->show();
         _channel->sendResponseAsync(msg, rp);
         break;
      }
~~~
```

In this code, *line* 5 sees the incoming message extracted from the communication channel and stored in `rp`. If the GUI component is operational, it triggers the `show()` function, which in turn interfaces with the QQuickView of the user interface. This process effectively ensures the display of the GUI window in response to the host application's event.

In this segment of code, the message is retrieved from the communication channel on *line* 5 and copied into `rp`. If the GUI is available, the `show()` method is executed. This call links to the QQuickView of the user interface, ensuring that the GUI window is displayed in response to the event from the host application.

Sushi

Another project employing a technical specification akin to the one detailed in this research is Sushi. Its description reads:

> Headless plugin host for ELK Audio OS.

Elk Audio OS is a Linux-based operating system specifically developed for real-time audio applications. Its primary focus is on embedded hardware within the "Internet of Musical Things" (IoMusT) sector. This OS is available as an open-source version for the *Raspberry Pi 4 Single Board Computer*, and it is also commercially offered for additional SOCs[9] like the STM32MP1, or Intel x86 platforms.[10]

Sushi serves as a headless *Digital Audio Workstation* that establishes the fundamental communication with the Elk Audio OS. It functions as headless software, characterized by its design to operate without a GUI frontend - the 'missing head'.

Sushi is equipped with multiple protocols and technologies for interaction and control, including MIDI, OSC[11], and gRPC[12].

[9] System-On-Chip
[10] https://elk-audio.github.io/elk-docs/html/documents/supported_hw.html
[11] Open Sound Control
[12] https://www.elk.audio/articles/controlling-plug-ins-in-elk-part-ii

> As SUSHI is a headless host, intended for use in an embedded device, it does not feature a graphical user interface. In its place is a gRPC interface that can be used for controlling all aspects of SUSHI and hosted plugins[6]

Additionally, Sushi integrates with the VST audio plugin standard[13]. This allows for the remote control of such plugins through its gRPC interface. A segment of the protobuf interface can be seen below:

```
service NotificationController
{
    rpc SubscribeToTransportChanges (GenericVoidValue) returns (stream TransportUpdate) {}
    rpc SubscribeToParameterUpdates (ParameterNotificationBlocklist) returns (stream ParameterUpdate) {}
    rpc SubscribeToPropertyUpdates (PropertyNotificationBlocklist) returns (stream PropertyValue) {}
    ~~~
}
```

Sushi provides various services to remotely manage the application or to monitor changes in the SUSHI application. Leveraging gRPC, this interface can be implemented in various languages supported by gRPC and protobuf. Elk Audio also offers two ready-made client implementations, one in C++[14] and another in Python[15].

[13] https://elk-audio.github.io/elk-docs/html/documents/building_plugins_for_elk.html#vst-2-x-plugins-using-steinberg-sdk
[14] https://github.com/elk-audio/elkcpp
[15] https://github.com/elk-audio/elkpy

Chapter 4: Headless Audio Plugins

TODO: clarify headless

4.1 Overview

This section transitions from defining the problem and introducing the necessary foundation to detailing our primary objective: enabling a seamless native development experience. This is achieved by moving away from rigid methods, such as the requirement for users to compile Qt in a separate namespace. Additionally, ensuring stabillity across all desktop platforms is a critical requirement.

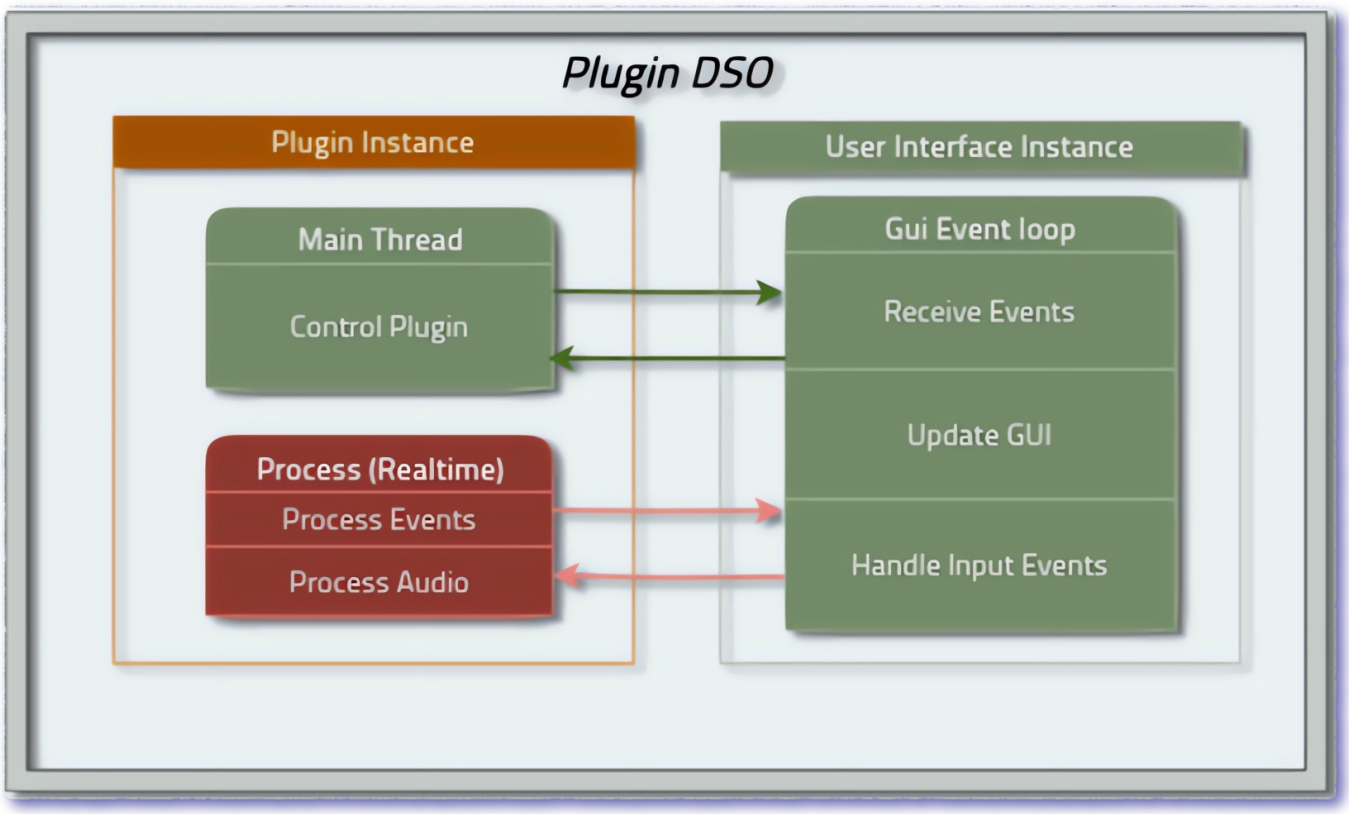

Figure 17: Overview of Traditional Plugin Architecture

fig. 17 outlines the conventional architecture of audio plugins when integrated with graphical user interfaces. In this context, multiple instances are generated from our Dynamic Shared Object (DSO), hence enforcing reentrancy. The traditional approach involves initiating the user interface's event loop within a distinct thread, typically resulting in at least two threads operating within the DSO's process. This method is evident in JUCE, a renowned library for creating agnostic plugins, and is also employed in various open-source projects.

Starting Qt's event loop within a dedicated thread presents significant challenges. To address these, we opted to run the plugin's GUI in an entirely separate process. While this might initially seem resource-intensive or complex, especially given the intricacies of IPC[16], the experimental findings have confirmed the effectiveness and feasibility of this approach.

With modern computers capable of efficiently handling an additional process, concerns regarding increased resource usage are mitigated. Operating the plugin's GUI in a separate process circumvents the re-entrancy issue with Qt and contributes to the overall stability and security of the implementation.

Isolating the GUI in a separate process strengthens the architecture. This ensures that GUI-related issues do

[16] Inter Process Communication

not affect the plugin's core functionality. It also increases the plugin's resilience to crashes or glitches in the GUI, safeguarding the main application's integrity and functionality. This method, though initially more complex, ultimately enhances reliability and user experience.

Consider a scenario where both the GUI and the real-time thread coexist within the same process. In such a case, a crash in the GUI could lead to the failure of the real-time thread. However, if the GUI operates as a separate process and encounters a crash, the real-time thread remains unaffected. Implementing Inter-Process Communication (IPC) or Remote Procedure Call (RPC) techniques, while more complex, does not significantly impact user experience. This is because the data exchanged between the processing and GUI unit, such as parameter adjustments or note events, is minimal, often just a few bytes. For sharing larger sets of data like audio, strategies like using shared memory pools can be effective, ensuring speed comparable to direct thread communication. However, this approach does introduce complexity in implementation. It is also worth noting that the operational speed of GUIs does not need to match that of real-time threads.

Human perception plays a crucial role when interacting with GUIs. The human eye's ability to perceive smooth and fluid visuals is limited, a concept encapsulated in the critical flicker fusion rate:

> The critical flicker fusion rate is defined as the rate at which human perception cannot distinguish modulated light from a stable field. This rate varies with intensity and contrast, with the fastest variation in luminance one can detect at 50–90 Hz

Recent research suggests that humans can detect flickering up to 500 Hz [3]. Standard desktop monitor refresh rates vary from 60 to 240 Hz, corresponding to 16 to $4.1\,ms$. This range sets the minimum communication speed an application should aim for to ensure a seamless user experience.

To integrate Qt GUIs into the CLAP standard, we will employ the following techniques:

1. Develop a native gRPC server to manage client connections and handle bidirectional event communication.
2. Implement a client-library that supports the server's API.
3. Launch the GUI in a separate process; establish a connection to the server upon startup.
4. Remotely control the GUI from the host for actions like showing and hiding.
5. Optionally, enable the client to request specific functionalities from the host.

Utilizing these methods in conjunction with QtGrpc has proven successful, delivering a reliable, fast, and native experience on desktop platforms.

4.2 Remote Control Interface

In this section, we explore the development of the Remote Control Interface (RCI) for CLAP. This interface is designed as a thin overlay on top of the CLAP API, providing a streamlined plugin layer abstraction. Its primary function is to abstract the interface in a manner that ensures communication is not only automatically configured but also consistently delivered. Users of this library are encouraged to engage with the features they wish to tailor to their needs.

> Note: The techniques and code examples presented in the subsequent sections represent the current stage of development and may not reflect the most optimized approaches. While several aspects are still under refinement, the fundamental concept has already demonstrated its effectiveness.

To ensure our server efficiently manages multiple clients, we will employ techniques similar to those used in Q*Application* objects. These were the same techniques that initially presented challenges in integration, leading to the extensive research detailed here. As highlighted in the objective section, static objects can be problematic but also have the potential to significantly enhance the architecture and, in this case, performance. As briefly touched upon in the clap-debugging section, the host may implement various plugin hosting modes. These modes impose additional constraints on our project, as the server must operate smoothly across all of them. The specific requirements for our server implementation are as follows:

- In scenarios where all plugins within a CLAP reside in the same process address space, we can optimize resource usage by sharing the server across multiple clients.
- In contrast, loading every plugin into a separate process, while being the safest approach, is also the most resource-intensive. In this case, reusing the server is not possible, demanding a separate server for each plugin, which leads to underutilized potential.

By avoiding the use of Qt's event system, we gain greater flexibility and overcome various integration challenges. At this stage, this approach appears to be the most flexible, performant, and stable solution. The adoption of gRPC leverages the language-independent features of protobufs IDL, allowing clients the freedom to implement the server-provided API in any desired manner. The server's primary role is to efficiently manage event traffic to and from its clients. This opens up possibilities such as controlling plugins through the CLI[17] or embedded devices, as long as they support gRPC and have a client implementation for the API. Additionally, we can statically link all dependencies against our library target, enhancing portability. This aspect of portability is vital, particularly in the realm of plugins, which often require extra attention and care.

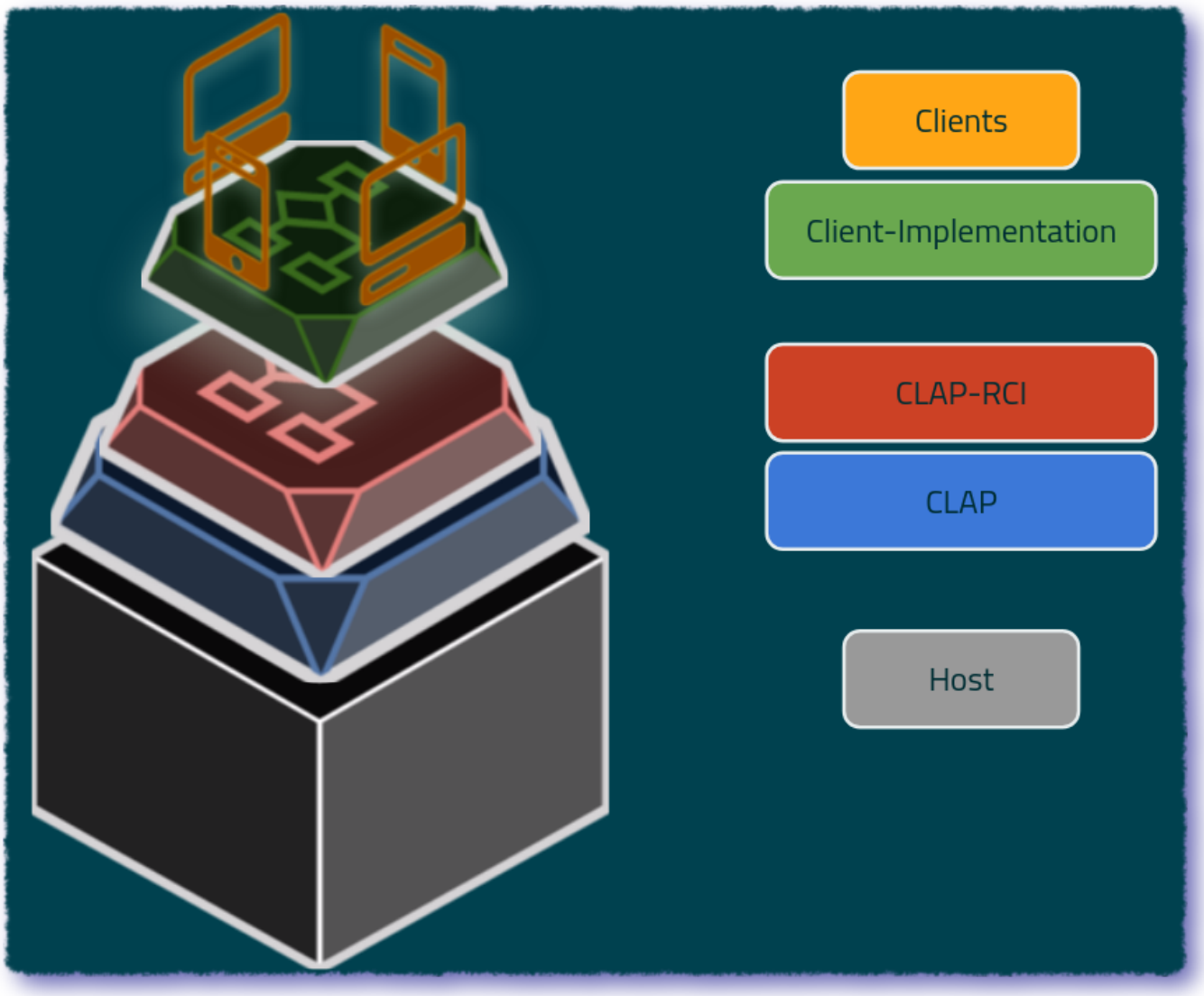

Figure 18: CLAP-RCI Architecture

fig. 18, represents the architecture of the developed server library. At the foundational level, there is a host that is compatible with the CLAP interface. Positioned just above is the CLAP API, which the host employs to manage its plugins. The client also interacts with this API, but with the addition of the CLAP-RCI wrapper library, which seamlessly integrates all necessary communication mechanisms. Above this layer is the client

[17] Command Line Interface

implementation of CLAP-RCI. The communication between these two layers is facilitated through the HTTP/2 protocol of gRPC. This structure enables the client implementation to support a diverse range of languages while providing independent and robust interference with the plugin's backend.

4.2.1 Server Implementation

When designing IPC mechanisms, there are two primary interaction styles: synchronous and asynchronous. Synchronous communication is characterized as a request/response interaction style, where a response to the RPC is awaited actively. In contrast, asynchronous communication adopts an event-driven approach, offering the flexibility for customized processing, which can potentially lead to more efficient hardware utilization.[2]

At the core of the gRPC C++ asynchronous API is the *Completion Queue*. As described in the documentation:

```
/// grpcpp/completion_queue.h
/// A completion queue implements a concurrent producer-consumer queue, with
/// two main API-exposed methods: \a Next and \a AsyncNext. These
/// methods are the essential component of the gRPC C++ asynchronous API
```

The gRPC C++ library offers three distinct approaches for server creation:

1. **Synchronous API**: This is the most straightforward method, where event handling and synchronization are abstracted away. It simplifies server implementation, with the client call waiting for the server's response.
2. **Callback API**: This serves as an abstraction over the asynchronous API, offering a more accessible solution to its complexities. This approach was introduced in the proposal L67-cpppcallback-api.
3. **Asynchronous API**: The most complex yet highly flexible method for creating and managing RPC calls. It allows complete control over the threading model but at the cost of a more intricate implementation. For example, explicit handling of RPC polling and complex lifetime management are required.

Since our application of gRPC deviates from traditional microservice architectures and operates within a constrained environment, we have opted for the asynchronous API. This choice allows for a more adaptable approach to our library's design. The following illustrates the code flow we will employ:

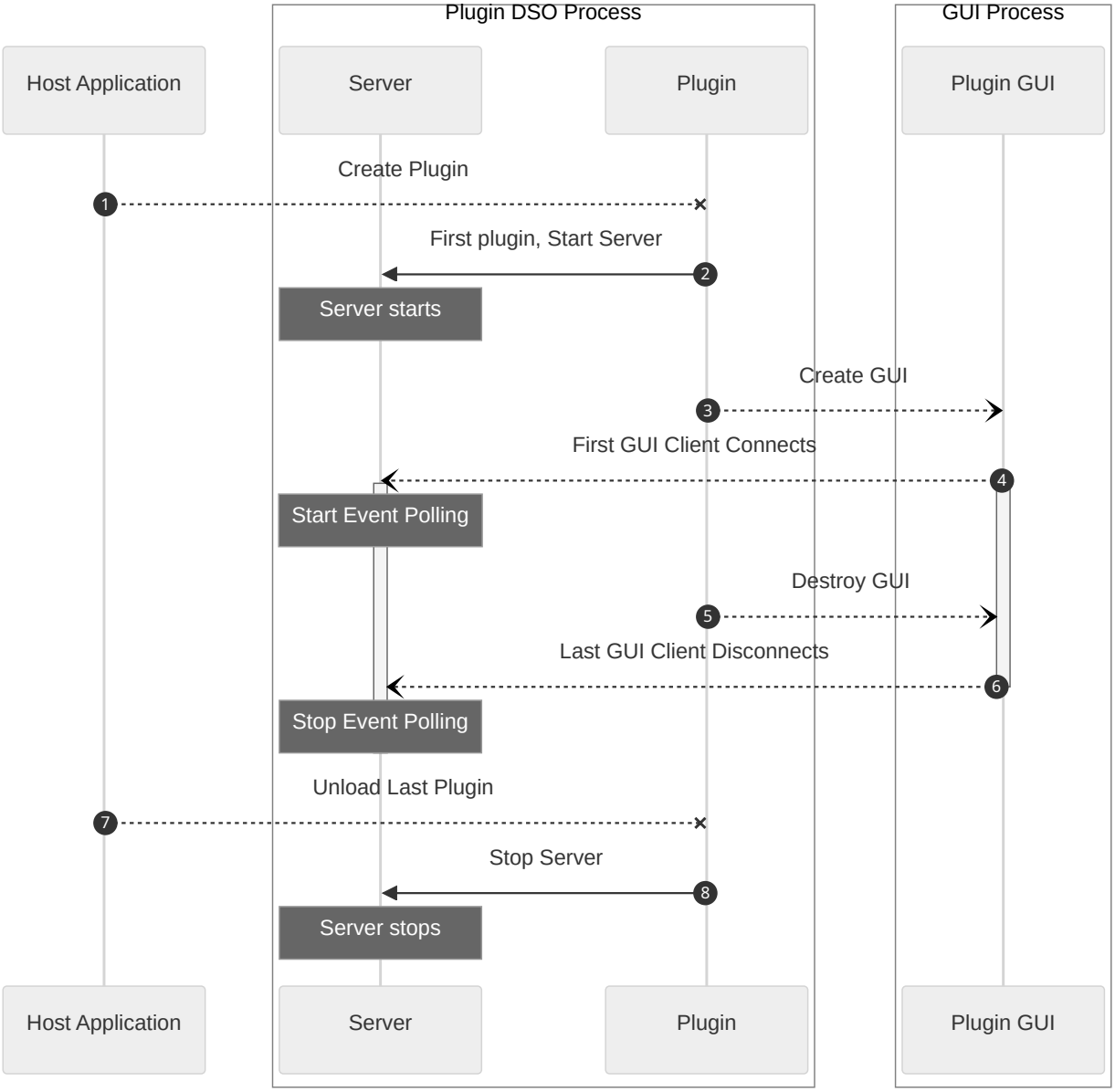

Figure 19: Lifetime of CLAP-RCI

fig. 19, outlines the server's lifecycle. The process begins with the creation of the first plugin, triggering the static server controller to initiate and start the server. When a plugin instance launches its GUI, it is provided with necessary command line arguments, such as the server's address and the plugin identifier. The GUI process then attempts to connect to the server. On the server side, the connection from the GUI client is managed, and an internal polling callback mechanism is initiated. This mechanism is responsible for polling and dispatching events to all active clients of every plugin. It is crucial to note, although not depicted in the diagram, that the server is capable of managing multiple plugin instances, each potentially having several clients. The event polling stops when the last client disconnects and all server-side streams are closed, a strategy implemented to enhance performance and minimize resource consumption in the absence of connected clients. Finally, the server is terminated when the last plugin is unloaded by the host, and the Dynamic Shared Object (DSO) is about to be unloaded.

To accurately identify or 'register' each plugin instance within the context of the static server handler, we employ a hashing algorithm known as MurmurHash. This step is crucial for ensuring that each GUI connecting to the server can be correctly mapped to its corresponding plugin instance. The generated hash value is used both as a command line argument when launching the GUI and as the key for inserting the plugin into a map.The following C++ code snippet demonstrates the implementation of MurmurHash3:

```
// core/global.h

// MurmurHash3
// Since (*this) is already unique in this process, all we really want to do is
// propagate that uniqueness evenly across all the bits, so that we can use
// a subset of the bits while reducing collisions significantly.
[[maybe_unused]] inline std::uint64_t toHash(void *ptr)
{
    uint64_t h = reinterpret_cast<uint64_t>(ptr); // NOLINT
    h ^= h >> 33;
    h *= 0xff51afd7ed558ccd;
    h ^= h >> 33;
    h *= 0xc4ceb9fe1a85ec53;
    return h ^ (h >> 33);
}
```

To facilitate communication with plugin instances, we have encapsulated all communication-related logic within the `SharedData` class. It serves as the hub for communication between the plugin and the server. This class is responsible for managing the polling mechanism and operating non-blocking, wait-free queues for event exchange with the plugin. A separate instance of *SharedData* is created for each plugin. Below is a segment of the public API for this class:

```
// server/shareddata.h
class SharedData
{
public:
    explicit SharedData(CorePlugin *plugin);

    // Methods for managing core plugins and event streams
    bool addCorePlugin(CorePlugin *plugin);
    bool addStream(ServerEventStream *stream);
    bool removeStream(ServerEventStream *stream);

    // Methods for handling events
    bool blockingVerifyEvent(Event e);
    bool blockingPushClientEvent(Event e);
    void pushClientParam(const ClientParams &ev);

    // Queue accessors
    auto &pluginToClientsQueue() { return mPluginProcessToClientsQueue; }
    auto &pluginMainToClientsQueue() { return mPluginMainToClientsQueue; }
    auto &clientsToPluginQueue() { return mClientsToPluginQueue; }
    // ...
}
```

In the upcoming code snippet, we will explore the constructor of the `CorePlugin` class. This constructor is fundamental, establishing the essential framework upon which the class operates. As a cornerstone of our architecture, the *CorePlugin* class stands at the forefront, acting as the primary interface for client interactions with this library:

```
// plugin/coreplugin.cpp
// CorePlugin::CorePlugin(~) {
~~~
    auto hc = ServerCtrl::instance().addPlugin(this);
    assert(hc);
    dPtr->hashCore = *hc;
    logInfo();
    // The shared data is used as a pipeline for this plugin instance
    // to communicate with the server and its clients
    dPtr->sharedData = ServerCtrl::instance().getSharedData(dPtr->hashCore);

    if (ServerCtrl::instance().start())
        SPDLOG_INFO("Server Instance started");
    assert(ServerCtrl::instance().isRunning());
~~~
```

Here we first add the plugin to the static `ServerCtrl`, which registers it with the hashing algorithm. We then obtain the `SharedData` instance for further communication and start the server if it hasn't been started yet.

Next, we explore the internal workings of the gRPC server. We employ the asynchronous API, chosen for its supreme flexibility. Despite the extensive coding required for the server implementation, this method provides considerable control and adaptability in handling the threading model.

The code segment below illustrates the construction of the server and the creation of the completion queues:

```
// server/server.cpp
    // Specify the amount of cqs and threads to use
    cqHandlers.reserve(2);
    threads.reserve(cqHandlers.capacity());

    // Create the server and completion queues; launch server
    { ~~~ } // Condensed for brevity

    // Create handlers to manage the RPCs and distribute them across
    // the completion queues.
    cqHandlers[0]->create<ServerEventStream>();
    cqHandlers[1]->create<ClientEventCallHandler>();
    cqHandlers[1]->create<ClientParamCall>();

    // Distribute completion queues across threads
    threads.emplace_back(&CqEventHandler::run, cqHandlers[0].get());
    threads.emplace_back(&CqEventHandler::run, cqHandlers[1].get());
```

The above code snippet already illustrates the advantages of utilizing the asynchronous API. By creating multiple completion queues and distributing them across various threads, we gain complete control over the threading model. The first completion queue, positioned at `0`, is dedicated to managing all server communications to clients, while the second queue, at position `1`, handles all client communications.

Three distinct *rpc-tag* handlers are employed:

1. `ServerEventStream`: Manages the outgoing stream from the plugin instance to its clients.
2. `ClientEventCallHandler`: Addresses significant gRPC unary calls originating from the client.
3. `ClientParamCall`: Specifically handles gRPC unary calls related to parameter adjustments.

This distinction between call types is crucial. Parameter calls from the client are directly forwarded to the real-time audio thread. In contrast, event calls, which can be blocking, do not require real-time handling. Instead, they are used for critical events that necessitate client verification during the GUI creation process.

The completion queues, as previously noted, function as FIFO (First-In, First-Out) structures, utilizing a tag system to identify the nature of the events being processed. Tags, representing specific instructions, are enqueued and later retrieved when the gRPC framework is ready to handle them.

The following C++ code snippet provides a glimpse into this mechanism:

```
// server/cqeventhandler.cpp
void CqEventHandler::run()
{
    state = RUNNING;

    void *rawTag = nullptr;
    bool ok = false;
    // Repeatedly block until the next event is ready in the completion queue.
    while (cq->Next(&rawTag, &ok)) {
        auto *tag = static_cast<EventTag*>(rawTag);
        tag->process(ok);
    }
~~~
```

At its very core, the completion queue is a blocking call that waits for the the next operation. We start them in a seperate thread inside the constructor of the server and they will run until we decide to stop them. The `Next` function returns a generic `void` pointer, which correlates to the tag we have enqueued at a earlier point in time. We then cast it back to the original type and call the process function. The `EventTag` is the base class for all other tags inside a completion queue. It provides a virtual `process` function which will be handled by either the derived classes or the base class itself. Extra care has been taken to keep those *tag* implementations slim to avoid potential bottlenecks. For instance, a segment from the `ServerEventStream` implementation looks like the following:

```
void ServerEventStream::process(bool ok)
{
    switch (state) {
        case CONNECT: {
            if (!ok)
                return kill();
            // Create a new instance to serve new clients while we're processing this one.
            parent->create<ServerEventStream>();
            // Try to connect the client. The client must provide a valid hash-id
            // of a plugin instance in the metadata to successfully connect.
            if (!connectClient()) {
                state = FINISH;
                stream.Finish({ grpc::StatusCode::UNAUTHENTICATED,
                    "Couldn't authenticate client" }, toTag(this));
                return;
            }
            sharedData->tryStartPolling()) {
                SPDLOG_INFO("Already polling events!");
            }
            state = WRITE;
```

```
            SPDLOG_INFO("ServerEventStream new client @ {} connected", toTag(this));
        } break;
~~~
```

This code segment illustrates the state machine within the `ServerEventStream`, triggered during the connection phase of new clients. The `connectClient()` function, as seen here, encapsulates the client authentication process.

A client is required to submit a valid hash-ID of a plugin instance for a successful connection. If this authentication step fails, the stream will be closed. This scenario also highlights how the tag-based system operates. The `stream.Finish` call involves the `toTag` function, which converts the `this` pointer into a `void` pointer using `reinterpret_cast`. With the state now set to `FINISH`, the subsequent processing of this tag by the completion queue will result in another call of the `process` function, but this time in a different state. Once the client connection is established, we initiate the polling process to facilitate the delivery of events.

```
########################################################################
        ******** 2023-10-17 01:43:34 ********
Host:               Clap Test Host, 0.1.0
Clap:               /home/wayn/.clap/qtclap-plugins.clap
Plugin:             QGain
CorePlugin hash:    18112701654865912731
Root Module:        GainModule
Plugin directory:   /home/wayn/.clap/qtclap-plugins
Log file:           /home/wayn/.clap/qtclap-plugins/logs/plugin.log
Executable:         /home/wayn/.clap/qtclap-plugins/Gain
########################################################################

[  info  ][7363#7363] Server listening on URI: 0.0.0.0, Port: 43763
[ trace  ][7363#7363] ServerEventStream created 0x555555a613d0
[ trace  ][7363#7363] Server-Stream completion queue served @ 0x5555558b55a0
```

Figure 20: Server Trace

fig. 20 showcases a section of the tracing when the project is compiled in debug mode. Debugging can be challenging with multiple threads operating in the completion queues and the potential simultaneous connection of several plugins and clients. This necessitates a development and debugging approach that is both structured and effective.

4.3.2 Event Handling and Communication

Handling and distributing events is a critical and complex part of this library. It's essential to process events quickly while adhering to real-time constraints, ensuring events are served in a deterministic and predictable manner. Additionally, managing the lifespan of incoming RPC calls, synchronizing access to shared resources, and developing a method to correctly invoke gRPC calls are significant challenges.

To expand on the last point, the gRPC library offers several methods for interacting with underlying channels, as seen with the `Finish` method for closing a channel, along with `Read` and `Write` methods. These are the essential vehicles to receive and transmit messages. However, the complexity arises with the asynchronous API, where these calls must be manually enqueued into the completion queues. This restricts the ability to call them directly from the plugin multiple times for client communication. Instead, they must be enqueued with a tag and await processing by the completion queue before another can be queued.

> Note: Completion Queues can only have one pending operation per tag.

This limitation is manageable when a single plugin instance communicates with one client. However, in scenarios where the server is designed to support **N** plugins, each with **M** clients, the method not only proves to be unscalable but, in fact, it doesn't scale at all. In such scenarios, the system's ability to scale effectively is significantly hindered.

An effective and efficient method is to centralize event distribution. This approach involves scanning through all messages queued from the plugin instances and dispatching them to the clients simultaneously. Such a technique significantly improves our operational efficiency. On the plugin side, concerns about synchronization and threading are softened. Plugins are registered with the server, obtaining a handle to the `SharedData` object. Events are then pushed into FIFO queues within SharedData, and the server automatically manages everything thereafter.

To enable this, it's necessary to implement an event polling mechanism. I opted not to introduce a separate thread for this task, aiming to avoid the complexities of thread switching and synchronization issues. Instead, I repurposed the existing completion queues. These queues are already operational and maintain internal thread pools, additionally offering capabilities for client channel communication. Our task involves enqueuing custom operations into the completion queue, which then manages their lifecycle and scheduling. Fortuitously, the gRPC library provides a suitable, albeit lesser-known, feature for this purpose: the `Alarm` class.

Implementing the event polling mechanism is crucial for this system to function effectively. To avoid creating another independent thread for this purpose, and to sidestep the complexities associated with thread switching and synchronization, I chose to utilize the existing completion queues. These queues are already active and efficiently managed by internal thread pools. They are also crucial in facilitating communication with client channels. Our approach requires enqueuing custom operations into the completion queue. This queue then takes responsibility for managing the lifecycle of these operations and coordinating their subsequent invocations. Fortunately, the gRPC library offers a valuable, though often underutilized, feature for this purpose: the `Alarm` class.

```
/// grpcpp/alarm.h
/// Trigger an alarm on completion queue \a cq at a specific time.
/// When the alarm expires (at \a deadline) or is cancelled (see \a Cancel),
/// an event with tag \a tag will be added to \a cq. If the alarm has expired,
/// the event's success bit will be set to true, otherwise false (i.e., upon cancellation).
//
// USAGE NOTE: This is often used to inject arbitrary tags into \a cq by
// setting an immediate deadline. This technique facilitates synchronization
// of external events with an application's \a grpc::CompletionQueue::Next loop.
template <typename T>
void Set(grpc::CompletionQueue* cq, const T& deadline, void* tag)
```

As highlighted in the *usage note*, this feature aligns perfectly with our requirements. To efficiently handle completion queues, we developed the `CqEventHandler` class. This class abstracts and manages the queues, storing all active (pending) tags:

```
    // server/cqeventhandler.h
~~~
    std::map<std::unique_ptr<eventtag>, grpc::alarm> pendingalarmtags;
    std::map<std::uint64_t, std::unique_ptr<eventtag>> pendingtags;
~~~
```

The `pendingtags` map tracks all currently active streams or calls for communication, while the `pendingalarmtags` map keeps track of all active alarms. Notably, our foundational `EventTag` class, while serving as a base for

other tag implementations, is also equipped to handle the `process(bool ok)` callback:

```
// server/tags/eventtag.h
~~~
class EventTag
{
public:
    using FnType = std::function<void(bool)>;
~~~
    virtual void process(bool ok);
    virtual void kill();

protected:
    CqEventHandler *parent = nullptr;
    ClapInterface::AsyncService *service = nullptr;

    std::unique_ptr<FnType> func{};
    Timestamp ts{};
};
~~~
```

Derived classes have the option to override the `process` function, as exemplified in the `ServerEventStream` implementation. However, by default, it executes the `func` member. This design allows for the integration of custom functions into our alarm tag-based subsystem and serves the foundation for the event polling.

When an alarm tag is processed by the completion queue, it triggers the `process` function, a standard procedure for all tags. In instances involving alarm tags, the function's default implementation is called into action:

```
// server/tags/eventtag.cpp
void EventTag::process(bool ok)
{
    (*func)(ok);
    kill();
}
```

In this implementation, the function is executed and the tag is immediately destroyed. Currently, this works with single-shot alarm tags, presenting an opportunity for future optimization. The key interface for interacting with this mechanism is within the `CqEventHandler` class:

```
// server/cqeventhandler.cpp
bool CqEventHandler::enqueueFn(EventTag::FnType &&f, std::uint64_t deferNs /*= 0*/)
{
    auto eventFn = pendingAlarmTags.try_emplace(
        std::make_unique<EventTag>(this, std::move(f)),
        grpc::Alarm()
    );
    { ~~~ }
    if (deferNs == 0) { // Execute immediately.
        eventFn.first->second.Set(
            cq.get(), gpr_now(gpr_clock_type::GPR_CLOCK_REALTIME), toTag(eventFn.first->first.get())
        );
    } else { // Defer the function.
        const auto tp = gpr_time_from_nanos(deferNs, GPR_TIMESPAN);
```

```
        eventFn.first->second.Set(cq.get(), tp, toTag(eventFn.first->first.get()));
    }

    return true;
}
```

The `enqueueFn` function takes a callable `f` and an optional defer time as parameters. First, it places the EventTag and Alarm instances into the map. This storage is crucial because destroying a `grpc::Alarm` triggers its `Cancel` function, which we want to avoid unless the server is shutting down. The completion queue should autonomously manage the timing of each tag's invocation, with manual cancellation reserved for server shutdown. Hence, the function is either deferred or executed immediately based on the provided parameters.

With this approach, we can seamlessly integrate events from plugin instances into their respective completion queue handlers. This not only offers a scalable approach but also ensures reliable interaction with the gRPC interface from external threads.

Let's delve into the essential mechanism operating behind the scenes, responsible for distributing events: the polling mechanism. As previously shown, a successful connection of a client triggers the `tryStartPolling` function. Here is how this function is defined:

```
bool SharedData::tryStartPolling()
{
    if (!isValid() || pollRunning)
        return false;

    { ~~~ } // Simplified for brevity

    pollRunning = true;
    mServerStreamCq->enqueueFn([this](bool ok){ this->pollCallback(ok); }, mPollFreqNs);
    return true;
}
```

This function returns false if the callback is already running. If not, it initiates the callback by enqueueing the `pollCallback` function, into the previously mentioned code segment. The standard polling frequency, `mPollFreqNs` , is set at 5,000 nanoseconds or 5 microseconds. This means that every 5 microseconds, the function is re-invoked to check for new events.

```
void SharedData::pollCallback(bool ok)
{
    ~~~
    if (streams.empty()) {
        SPDLOG_INFO("No streams connected. Stop polling.");
        return endCallback();
    }
    ~~~
```

To avoid unnecessary polling in the absence of connected clients, the polling mechanism is designed to stop automatically. When clients disconnect, either by themself or by the server, the `streams` container will be updated.

```
~~~
    // Processing events from queues
    const auto nProcessEvs = consumeEventToStream(mPluginProcessToClientsQueue);
```

```
    const auto nMainEvs = consumeEventToStream(mPluginMainToClientsQueue);

    if (nProcessEvs == 0 && nMainEvs == 0) {
        // No events to dispatch, increase the backoff for the next callback.
        mServerStreamCq->enqueueFn([this](bool ok){ this->pollCallback(ok); }, nextExpBackoff());
        return;
    }
~~~
```

Upon passing the initial checks, the system starts to process the events generated by each plugin instance. The `consumeEventToStream` function handles the translation of CLAP-specific events into gRPC messages. We have branched the event container into two queues: one for real-time processing and another for generic operations. This isolation is necessary to comply with the strict real-time requirements. Both queues are preallocated with a fixed size and are designed to be wait-free, overwriting old messages when reaching capacity. This approach is a necessary tradeoff, as blocking on the real-time thread cannot be afforded. If there are no events to be sent, the system schedules the next callback with a progressively increasing exponential backoff. This technique is employed to minimize the server's CPU consumption during its hiatus in event activity, thereby enhancing overall efficiency.

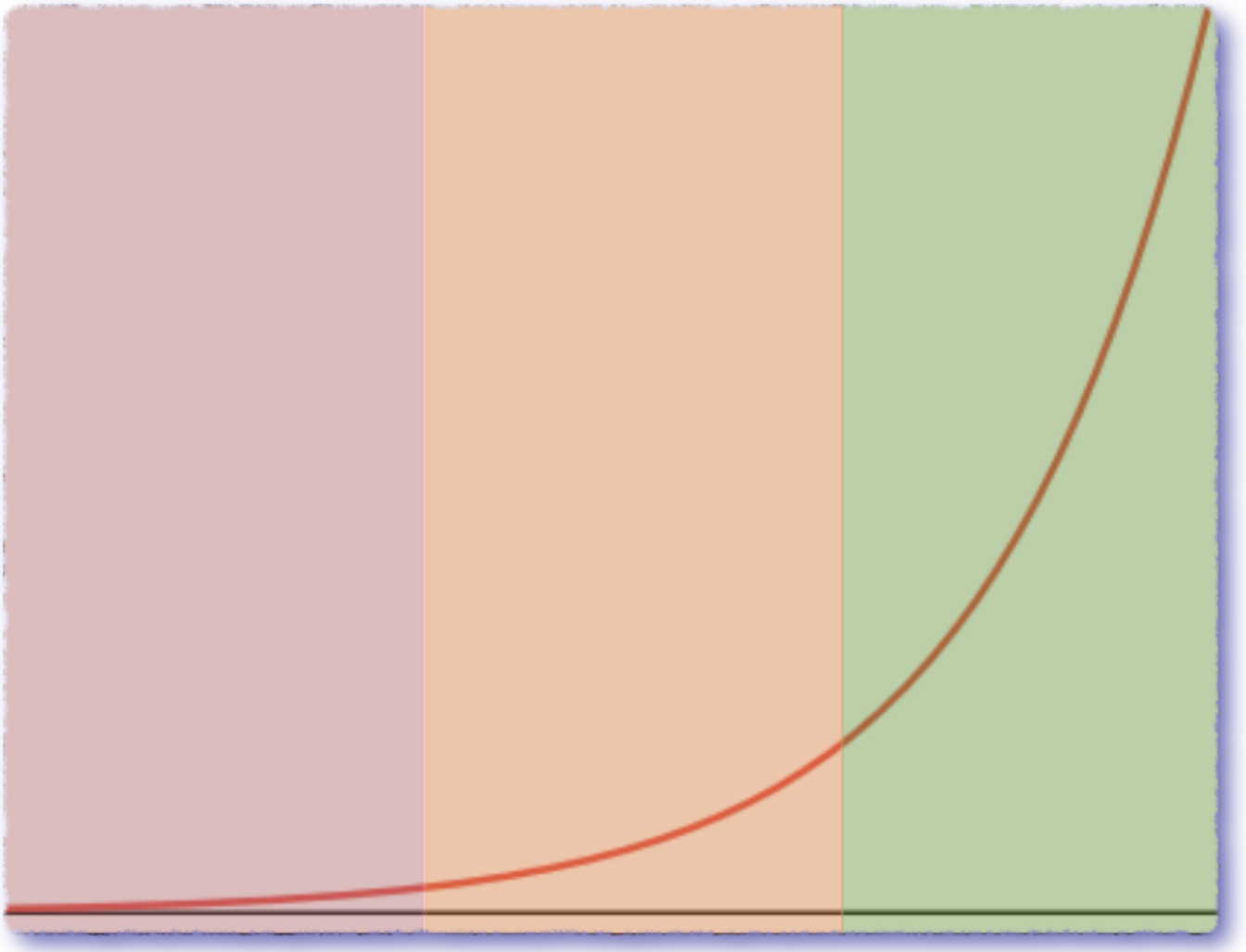

Figure 21: Exponentiall Backoff

fig. 21 illustrates the exponential backoff curve employed to determine the interval before the next event check. The curve is segmented into three color-coded sections, each representing different levels of server activity

and corresponding backoff strategies:

1. **High Event Occurrence (Red Zone)**: In this zone, the server immediately processes events as they occur, providing the quickest response during high-traffic periods.
2. **Cooldown (Orange Zone)**: This section marks a reduced event rate. Without new events, the server increases the backoff time, balancing readiness with resource saving.
3. **Standby (Green Zone)**: Representing periods where no events are detected. Here, the backoff time is increased progressively, placing the server in a standby state. The server operates with the least frequency of checks in this zone, optimizing the server's resource utilization until a new event triggers a return to the red zone.

Any event occurrence immediately resets the backoff to the shortest interval. This mirrors common scenarios where users are actively interacting with the system, triggering frequent events that require quick server responses. Once user interaction drops off and plugins are not in use, the system seamlessly shifts into **stand-by** mode to optimize efficiency.

```
trace  ][99023#99057] 0 0, time: 5000000
trace  ][99023#99057] 0 0, time: 10000000
trace  ][99023#99057] Consumed event: ParamValue
trace  ][99023#99057] 1 0, time: 20000000
trace  ][99023#99057] PollCallback has sent: 1 Process Events and 0 Main Events
trace  ][99023#99057] 0 0, time: 2500000
trace  ][99023#99057] 0 0, time: 5000000
trace  ][99023#99057] 0 0, time: 10000000
trace  ][99023#99057] 0 0, time: 20000000
trace  ][99023#99057] 0 0, time: 40000000
trace  ][99023#99057] 0 0, time: 80000000
trace  ][99023#99057] 0 0, time: 160000000
trace  ][99023#99057] 0 0, time: 320000000
trace  ][99023#99057] 0 0, time: 640000000
trace  ][99023#99057] 0 0, time: 1280000000
```

Figure 22: Exponentiall Backoff Server Trace

fig. 22 captures this dynamic behavior. An additional trace message in the code helps accentuate the exponential backoff's response to activity. The trace messages indicate that the backoff interval is reset upon encountering an event and progressively increasing as the server enters a period of inactivity.

```
~~~
    // If we reached this point, we have events to send.
    bool success = false;
    for (auto stream : streams) { // For all streams/clients
        if (stream→sendEvents(mPluginToClientsData)) // try to dispatch events.
            success = true;
    }

    { ~~~ } // Condensed for clarity

    // Successfully completed a cycle. Schedule the next callback with
    // the standard polling interval and reset the backoff.
    mPluginToClientsData.Clear();
    mCurrExpBackoff = mPollFreqNs;
    mServerStreamCq→enqueueFn([this](bool ok){ this→pollCallback(ok); }, mPollFreqNs);
```

```
~~
```

In this final segment, the process of event dispatching is detailed. It goes through every connected stream (the `ServerEventStream` s), and attempts to forward the events. Once this process is successfully executed, the next callback is arranged at the usual polling rate.

4.2.3 CLAP API Abstraction

The previous section has already illustrated the server's structure and the interaction among its core components. This section delves into the integration of the server with the CLAP API. Given our insights from the CLAP section, we're aware that the CLAP standard offers an extension-based interface that allows plugins to add their unique functionalities. The API abstraction utilizes the clap-helpers toolkit, provided by the CLAP team. This small library contains a range of utilities designed to ease the development processes. Among these is the `Plugin` class, which acts as an intermediary between the C-API and the plugin specific contexts. It functions as a slim abstraction layer, employing trampoline functions to relay calls to the individual c++ instances.

The `CorePlugin` class is employed to offer a standard default implementation, incorporating server-specific features. Users can then tailor this class by overriding certain functions in its virtual interface, allowing for customized functionality.

For client GUI interactions, two separate communication channels are established. The primary channel is non-blocking, designed for interactions where timing is critical. Here, events are sent using a UDP[18]-inspired, *fire and forget* approach. This approach is practical because it doesn't require immediate confirmation of the client receiving the message. The events, generated from the host's process callback, reflect current values. Due to the frequent updates and the rapid pace at which the process callback operates, waiting for client feedback is not practical.

Nevertheless, certain situations necessitate waiting for client responses. A notable instance is the implementation of the GUI extension. In this setup, the GUI operates as an independent process, leading to extra synchronization complexities. The approach to manage this includes:

1. Providing the hashed ID of the plugin instance and the server address to the GUI.
2. Launching the GUI executable using platform-specific APIs.
3. The GUI then connects to the server and verifies the hash ID.
4. For every subsequent GUI event initiated by the host, such as showing or hiding the GUI, it becomes essential to wait for the client's response.

> Note: The CLAP specification does not enforce a strict design and provides some freedom to the host application's implementation. Some hosts may choose to destroy the GUI after hiding it, while some other hosts may choose to conceal the UI, keeping it active in the background.

This variation in approach requires a versatile and robust GUI creation mechanism. The public interface for these GUI-related functions is outlined as follows in:

```
// coreplugin.h
    // #### GUI ####
    bool implementsGui() const noexcept override;
    bool guiIsApiSupported(const char *api, bool isFloating) noexcept override { return isFloating; };
    bool guiCreate(const char *api, bool isFloating) noexcept override;
    bool guiSetTransient(const clap_window *window) noexcept override;
    bool guiShow() noexcept override;
    bool guiHide() noexcept override;
    void guiDestroy() noexcept override;
```

[18] User Datagram Protocol

Every extension can be controlled with the `bool implements<ext>()` function. If a user overrides this function in their subclassed instance to return false, the corresponding extension is deactivated and subsequently disregarded by the host application.

The process of creating a GUI demands synchronization with the client. This step is crucial to confirm the successful creation and validation of the GUI before letting the host application proceed.

```
bool CorePlugin::guiCreate(const char *api, bool isFloating) noexcept
{
    if (!ServerCtrl::instance().isRunning()) {
        SPDLOG_ERROR("Server is not running");
        return false;
    }

    if (dPtr->guiProc) {
        SPDLOG_ERROR("Gui Proc must be null upon gui creation");
        return false;
    }
    dPtr->guiProc = std::make_unique<ProcessHandle>();
    { ~~~ } // other checks simplified for brevity
```

Upon entering the `guiCreate` function, a series of critical checks are performed to ensure the GUI's safe operation. Following this, a new `ProcessHandle` instance is created, serving as an abstraction for platform-specific process APIs. This requirement arises from the lack of a standardized 'process' concept in C++. Therefore, this class provides a uniform interface for interacting with different platform-specific process APIs. For instance, the `CreateProcess` API from `Kernel32.dll` is used for Windows environments, while Linux and macOS employ a POSIX-compliant implementation. A section of the public interface looks like the following:

```
#if defined _WIN32 || defined _WIN64
#include <Windows.h>
using PidType = DWORD;
#elif defined __linux__ || defined __APPLE__
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
using PidType = pid_t;

{ ~~~ } // Only Windows and Unix supported for now

class ProcessHandle
{
public:
    ProcessHandle();
    { ~~~ }
    bool isChildRunning() const;
    std::optional<int> waitForChild();
    { ~~~ }
    bool startChild();
    std::optional<int> terminateChild();
    { ~~~ }
private:
    std::unique_ptr<ProcessHandlePrivate> dPtr;
```

```
};
```

This interface facilitates starting, terminating, or waiting for the completion of a GUI process. It's utilized in the `guiCreate` function, as demonstrated in the snippet below:

```
// coreplugin.cpp > guiCreate(~) {
~~~
    // Send the hash as argument to identify the plugin to this instance.
    const auto shash = std::to_string(dPtr->hashCore);
    if (!dPtr->guiProc->setArguments({ *addr, shash })) { // Prepare to launch the GUI
        SPDLOG_ERROR("Failed to set args for executable");
        return false;
    }

    if (!dPtr->guiProc->startChild()) {  // Start the GUI
        SPDLOG_ERROR("Failed to execute GUI");
        return false;
    }
~~~
```

As mentioned earlier in the server implementation section, a hashing algorithm is utilized to identify each plugin instance. This hash is provided as an argument to the executable, and it is essential that the client handles this hash and includes it when initiating communication.

```
// coreplugin.cpp > guiCreate(~) {
~~~
    pushToMainQueueBlocking({Event::GuiCreate, ClapEventMainSyncWrapper{}});
    if (!dPtr->sharedData->blockingVerifyEvent(Event::GuiCreate)) {
        SPDLOG_WARN("GUI has failed to verify in time. Killing Gui process.");
        if (!dPtr->guiProc->terminateChild())
            SPDLOG_CRITICAL("GUI proc failed to killChild");
        return false;
    }
    enqueueAuxiliaries();
    return true;
}
```

Here, the `Event::GuiCreate` event is dispatched to the main queue, triggering a waiting mechanism for the client's response. Should the client fail to respond in time, the process is terminated. Otherwise the host is allowed to proceed.

The next step requires the host to provide a window-handle to the plugin instance. The GUI plugin must incorporate this handle through a process known as "re-parenting," which embeds the GUI within the context of another external window:

```
bool CorePlugin::guiSetTransient(const clap_window *window) noexcept
{
    assert(dPtr->sharedData->isPolling());

    { ~~~ }

    pushToMainQueueBlocking( {Event::GuiSetTransient, ClapEventMainSyncWrapper{ wid } });
```

```
    if (!dPtr->sharedData->blockingVerifyEvent(Event::GuiSetTransient)) {
        SPDLOG_ERROR("Failed to get a verification from the client");
        return false;
    }
    return true;
}
```

The `guiSetTransient` implementation shows a similar pattern to the `guiCreate` function, but it's in the non-blocking event handling where we see a more streamlined approach:

```
void CorePlugin::processEvent(~~~) noexcept
{
    { ~~~ }
    switch (evHdr->type) {

    case CLAP_EVENT_PARAM_VALUE: {
        const auto *evParam = reinterpret_cast<const clap_event_param_value *>(evHdr);
        { ~~~ }
        param->setValue(evParam->value);
        pushToProcessQueue({ Event::Param, ClapEventParamWrapper(evParam) });
    } break;
    { ~~~ }
    case CLAP_EVENT_NOTE_ON: {
        const auto *evNote = reinterpret_cast<const clap_event_note *>(evHdr);
        pushToProcessQueue({ Event::Note, ClapEventNoteWrapper { evNote, CLAP_EVENT_NOTE_ON }});
    } break;
    { ~~~ }
```

Here, the focus is on efficiently enqueuing events as they are received from the host. Each event is move-constructed into its respective wrapper class. These specialized wrapper classes are essential in ensuring that the integrity of the events is maintained, particularly focusing on the realtime requirements under which they operate in.

```
void CorePlugin::deactivate() noexcept
{
    dPtr->rootModule->deactivate();
    pushToMainQueue({Event::PluginDeactivate, ClapEventMainSyncWrapper{}});
}

bool CorePlugin::startProcessing() noexcept
{
    dPtr->rootModule->startProcessing();
    pushToMainQueue({Event::PluginStartProcessing, ClapEventMainSyncWrapper{}});
    return true;
}
```

This design demonstrates the ease of integration with the server. The complexity of managing these events, including their timing and processing, is adeptly handled by the underlying server-library. This allows the plugin developer to focus on the plugin's functionality without delving into the intricate details of event management and server communication.

4.2.4 Server API

The library discussed thus far lays the groundwork for users to develop their projects. To fully leverage this library, users need a client that implements the public API provided by the server. This is crucial for integrating the full capabilities of gRPC and its language-independent protobuf interface into their plugin and GUI development workflows. The API is accessible in the `api/v0/api.proto` file, containing the essential interface for gRPC communication. This setup mirrors the event structuring approach of CLAP, where a header identifies the payload type:

```
enum Event {
  PluginActivate                = 0;
  // { ~~~ }
  GuiCreate                     = 26;
  GuiSetTransient               = 27;
  GuiShow                       = 28;
  // { ~~~ }
  Param                         = 31;
  ParamInfo                     = 32;
  // { ~~~ }
  EventFailed                   = 42;
  EventInvalid                  = 43;
}
```

The `Event` enum specifies the type of payload being processed. It is essential for the host to send the correct event type to ensure proper communication. The `ServerEvent` message, the fundamental construct for outgoing streams showcases this:

```
// Plugin → Clients
message ServerEvent {
  Event event = 1;
  oneof payload {
    ClapEventNote note = 2;
    ClapEventParam param = 3;
    ClapEventParamInfo param_info = 4;
    ClapEventMainSync main_sync = 5;
  }
}
```

Currently, this API only supports a subset of the CLAP specification, primarily note, parameter, and generic events. To further optimize the use the HTTP/2 channel with server streaming, we allow to batch multiple `ServerEvent` messages into a single message:

```
message ServerEvents {
  repeated ServerEvent events = 1;
}
```

Working in tandem with the previously discussed event polling and dispatching algorithm, which operates at a fixed maximum frequency, this allows for the aggregation of events. A specific event payload can be seen for the `ClapEventParam` message:

```
message ClapEventParam {
  enum Type {
    Value = 0;
```

```
    Modulation = 1;
    GestureBegin = 2;
    GestureEnd = 3;
  }
  Type type = 1;
  uint32 param_id = 2;
  double value = 3;
  double modulation = 4;
}
```

This message type wraps the CLAP specific parameter events. A parameter can have four different states:

- `Type::Value` signifies a standard parameter update, typically triggered by user actions like adjusting a knob or slider in the GUI.
- `Type::Modulation` occurs when a parameter is influenced by external modulation sources, such as Low-Frequency Oscillators (LFOs).
- `Type::GestureBegin` and `Type::GestureEnd` are generally sent by the plugin's GUI to mark the start and end of manual parameter manipulation. These events can be utilized to enhance the audio rendering process.

Exploring the service definition reveals how gRPC generates the actual API, building on the individual messages previously discussed. The current API is defined as follows:

```
service ClapInterface {
  rpc ServerEventStream(ClientRequest) returns (stream ServerEvents) {}
  rpc ClientEventCall(ClientEvent) returns (None) {}
  rpc ClientParamCall(ClientParams) returns (None) {}
}
```

In this definition, three distinct RPC methods are outlined:

1. `ServerEventStream` : Serves as the primary communication channel from the plugin to its clients. Clients must initiate this stream to start receiving events.
2. `ClientEventCall` and `ClientParamCall` : Function as unary calls from the client, meaning they are executed once per message.

Originally, due to QtGrpc supporting only unary calls and server streaming, these functionalities were adopted. While functional, this setup is not entirely optimal for the use case at hand. In the future, the service definition would benefit from supporting bidirectional streams and client-side streams. Such streams provide a more efficient, sustained communication channel compared to unary calls, which tend to be slower because they handle messages on a one-at-a-time basis.

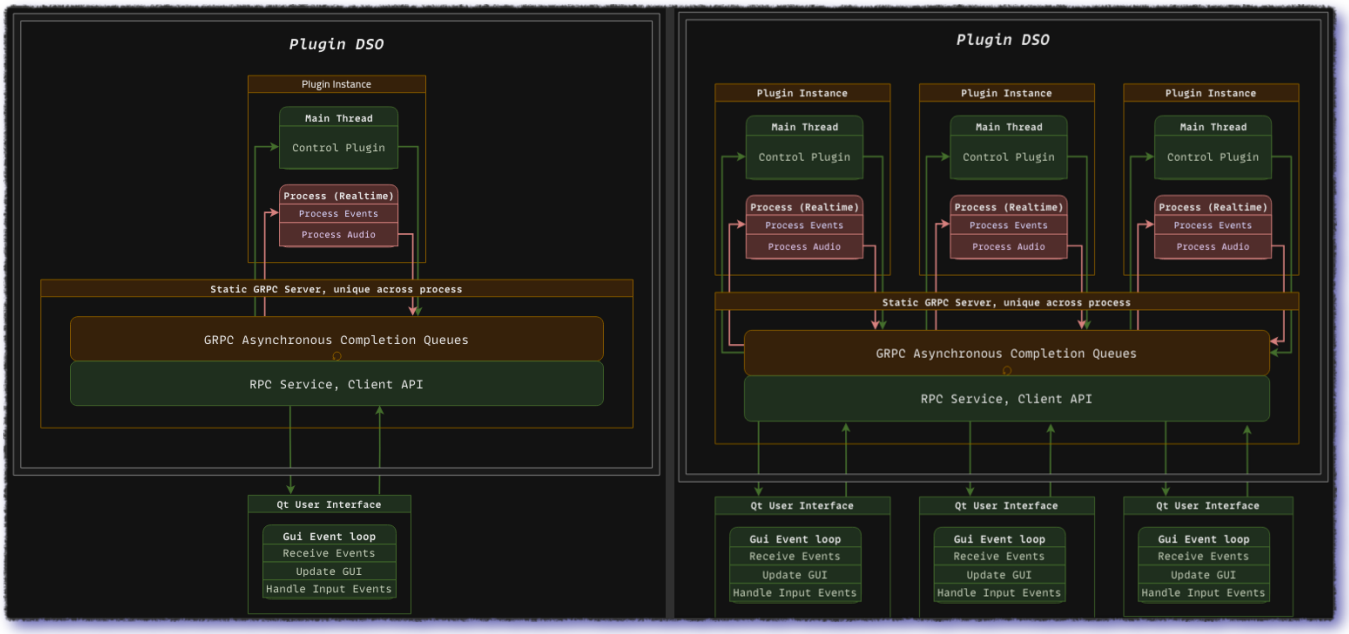

Figure 23: CLAP RCI mono and multi client

fig. 23 visualizes the architecture of the CLAP RCI library. On the left is the *mono-client* architecture, which means only a single client is connected, or the DSO is running fully sandboxed, which means only a single client will ever connect. On the right, the multi-client configuration unveils the library's full capabilities, where numerous clients share the same completion queues and threads, illustrating a more collaborative and dynamic environment.

4.3 Implementing the QtGrpc Client

This segment explores the use of QtGrpc to develop a client implementation for the `ClapInterface` service, as previously outlined in the server API section. The central aim of the client library is to aid in the development of graphical user interfaces for audio plugins. This tool leverages the CLAP-RCI library to allow for creating plugins, by focusing on the UI aspect. It efficiently handles the event stream emitted by the audio plugin and incorporates these events into Qt's event loop. This strategic integration pursues to utilize Qt's established signals and slots mechanism, empowering graphic components to autonomously and effectively manage their visual representation. The library's design and functionality are particularly geared towards simplifying and enhancing the GUI development process, while the creation of example plugins serves as an additional use case.

Currently, the development of this library is concentrated on integrating with Qt's QML and QtQuick modules. It's important to note that while the term "library" is used, the implementation is more akin to an extended sandbox designed for experimental purposes in implementing these features.

4.3.1 Core Concepts of the Client Library

The project's build configuration includes CLAP-RCI as a dependency. It employs the `api.proto` file to generate the necessary C++ client code. The implementation is divided into two primary sections:

1. **Clap.Interface:** This part functions in the background, handling the transmission of events to and from the server. It forms the core of the library, integrating the incoming messages with Qt.
2. **Clap.Controls:** This section is dedicated to the actual UI components, which are Qml components preset with essential properties and signals. These components are designed to effectively manage and display the events received from the server.

In the CMake configuration file, the server library is added as a dependency. The client code is generated by leveraging QtGrpc's protobuf plugin:

```
# src/CMakeLists.txt
add_subdirectory(3rdparty/clap-remote)
set(proto_file "${CMAKE_CURRENT_SOURCE_DIR}/3rdparty/clap-remote/api/v0/api.proto")
add_subdirectory(clapinterface)

~~~

# src/clapinterface/CMakeLists.txt
# Generate the protobuf and grpc files
qt_add_protobuf(ClapMessages QML
    QML_URI "Clap.Messages"
    PROTO_FILES ${proto_file}
)

qt_add_grpc(ClapMessages CLIENT
    PROTO_FILES ${proto_file}
)
```

The `qt_add_protobuf` and `qt_add_grpc` functions in CMake, provided by QtGrpc, streamline the process of integrating with the `protoc` plugin. These functions are specifically designed to simplify the generation of client code from protobuf definitions. Optionally, they can be configured to create QML metadata, enhancing the integration of protobuf messages within QML code. This feature allows client applications to seamlessly incorporate the generated modules into both their C++ and QML codebases, thereby facilitating a more efficient and cohesive development process.

We define the QML *module* for integration into both C++ and QML codebases of client applications as follows:

```
add_library(ClapInterface STATIC)
qt_add_qml_module(ClapInterface
    URI "Clap.Interface"
    VERSION ${CMAKE_PROJECT_VERSION}
    CLASS_NAME ClapInterfacePlugin
    PLUGIN_TARGET clapinterfaceplugin
    SOURCES
        "qclapinterface.h"
        "qclapinterface.cpp"
        "qnotehandler.h"
        "qnotehandler.cpp"
        "qnote.h"
        "qnote.cpp"
    OUTPUT_DIRECTORY ${QML_OUTPUT_DIR_INTERFACE}
    IMPORTS Clap.Messages
)
```

This configuration results in the creation of the `Clap.Interface` module, which is later registered with the QML engine. By declaring the `ClapInterface` as a static library, we minimize the number of dependencies on shared libraries. The compilation of this module produces the following files:

```
❱ tree -L 2 cmake-build/Clap/
Clap/
```

```
└── Interface
    ├── ClapInterface_qml_module_dir_map.qrc
    ├── ClapInterface.qmltypes
    ├── libclapinterfaceplugin.a
    └── qmldir
```

For their applications, users of this library need to link against the `libclapinterfaceplugin.a` file. This approach ensures that when the `Clap.Interface` module is imported in QML code, the QML engine will load the relevant directory. With this setup in place, the next step is to explore the functionalities and features of the `QClapInterface` class.

```
// src/clapinterface/qclapinterface.h
~~~
#include "api.qpb.h"
#include "api_client.grpc.qpb.h"
using namespace api::v0;
~~~
```

The code begins by including the generated protobuf and grpc files. These files are crucial for every QtGrpc client implementation as they provide a Qt-compatible representation of the server's proto API.

```
// src/clapinterface/qclapinterface.h
~~~
class QClapInterface : public QObject
{
    Q_OBJECT
    QML_SINGLETON
    QML_NAMED_ELEMENT(ClapInterface)
    QML_UNCREATABLE("QClapInterface is a singleton")

    Q_PROPERTY(PluginState state READ state NOTIFY stateChanged)
    Q_PROPERTY(bool visible READ visible WRITE setVisible NOTIFY visibleChanged)
    Q_PROPERTY(QWindow* transientParent READ transientParent NOTIFY transientParentChanged)
~~~
```

The `QClapInterface` class is uniquely identified as a `QML_SINGLETON`, ensuring only one instance exists within the QML engine. This singleton design is used as it centralizes the handling of all incoming and outgoing messages, eliminating the need for multiple instances. The class incorporates several properties using the `Q_PROPERTY` macro, which facilitates their accessibility from QML code. These properties are crucial for basic UI functionalities, such as toggling the UI's visibility or setting the parent window as per host requests.

To initiate communication with the server, the library requires the creation of a `QGrpcChannel`. This process is encapsulated in the `connect` function and is an essential step for users of this library to engage in server interactions.

```
// src/clapinterface/qclapinterface.cpp
void QClapInterface::connect(const QString &url, const QString &hash)
{
    // Provide the metadata to the server
    QGrpcChannelOptions channelOptions(url);
    metadata = {
        { QByteArray(Metadata::PluginHashId.data()), { hash.toUtf8() } },
```

```
    };
    channelOptions.withMetadata(metadata);

    // Create a Http2 channel for the communication
    auto channel = std::make_shared<QGrpcHttp2Channel>(channelOptions);
    client->attachChannel(channel);
    // Start the server side stream
    stream = client->streamServerEventStream(ClientRequest(), {});
    ~~~
}
```

This function is critical for establishing a connection between the client and the server. It is initiated by passing the `url` and `hash` arguments, typically supplied by CLAP-RCI during the executable's startup. These arguments are crucial in invoking this function. The `hash` is added to the channel, allowing the server to identify the specific plugin instance that the client aims to connect with. The `client` in this context utilizes the generated QtGrpc code to interact with the server-defined service. This setup ensures to establish a communication between the client application and the server.

The `client` object is declared as a unique pointer to the `ClapInterface::Client` type:

```
    // src/clapinterface/qclapinterface.h
    std::unique_ptr<ClapInterface::Client> client = {};
```

Delving into the generated code, we find that the `Client` class accurately mirrors the service defined by the server:

```
// build/api_client.grpc.qpb.h
class QPB_CLAPMESSAGES_EXPORT Client : public QAbstractGrpcClient
{
    Q_OBJECT
public:
    explicit Client(QObject *parent = nullptr);
    std::shared_ptr<QGrpcServerStream> streamServerEventStream(~~~);

    QGrpcStatus ClientEventCall(~~~);
    Q_INVOKABLE void ClientEventCall(~~~);

    QGrpcStatus ClientParamCall(~~~);
    Q_INVOKABLE void ClientParamCall(~~~);
~~~
```

The code above showcases the three service calls defined in the server's API. The generated code integrates seamlessly into Qt's ecosystem, utilizing the `Q_INVOKABLE` macro, which allows these methods to be invoked from QML. All other generated messages are also integrated into Qt's meta-object-system for convenience, enabling easy utilization and integration of these messages. This approach facilitates the incorporation of the external gRPC system, potentially written in any language supported by gRPC, into Qt's framework.

To complete the `connect` function, it's necessary to establish a connection to the server stream and manage the incoming messages effectively.

```
// src/clapinterface/qclapinterface.cpp
// void QClapInterface::connect(~~~) {
~~~
```

```
    QObject::connect(stream.get(), &QGrpcServerStream::errorOccurred, this,
        [](const QGrpcStatus &status) { QGuiApplication::quit(); }
    );

    QObject::connect(stream.get(), &QGrpcServerStream::finished, this,
         []() { QGuiApplication::quit(); }
    );

    QObject::connect(stream.get(), &QGrpcServerStream::messageReceived, this,
        [this]() { processEvents(stream->read<ServerEvents>()); }
    );
    callbackTimer.start();
~~~
```

In the current setup, both the `errorOccurred` and `finished` signals from the stream are simply connected to trigger the application's termination. The key aspect of this function, however, is the handling of incoming messages in the `processEvents` callback. This callback is responsible for distributing the received events as needed.

The function also initializes the `callbackTimer` , which operates at a fixed frequency. This design choice mirrors the approach used in CLAP-RCI and serves to consolidate all outgoing events into a single message. The rationale behind this method is to minimize the number of unary calls to the server, thereby enhancing performance. Ideally, future implementations would benefit from the use of bidirectional streaming to further reduce the number of independent RPC calls. This would offer a more efficient communication channel between the client and the server.

```
// src/clapinterface/qclapinterface.cpp
void QClapInterface::processEvents(const ServerEvents &events)
{
    for (const auto &event : events.events()) {
        switch (event.event()) {

            case EventGadget::PluginActivate: {
                mState = Active;
                emit stateChanged();
            } break;
            { ~~~ }
            case EventGadget::GuiCreate: {
                // Verify successful creation back to the host.
                auto call = client->ClientEventCall(create(EventGadget::GuiCreate));
                mState = Connected;
                emit stateChanged();
            } break;
            { ~~~ }
            case EventGadget::GuiShow: {
                client->ClientEventCall(create(EventGadget::GuiShow));
                setVisible(true);
            } break;
            { ~~~ }
        }
    }
```

```
}
```

The `processEvents` function is triggered every time a new message is received from the server. It iterates over all the events within the received message, handling each event based on its type, as defined in the mentioned `enum` of the API. The function utilizes a switch statement to process different events, such as `PluginActivate` , `GuiCreate` , and `GuiShow` . For each case, appropriate actions are taken, such as updating the state of the plugin or modifying visibility, and corresponding signals are emitted. These signals then enable subscribed components within the application to respond to these events effectively.

```
// src/clapinterface/qclapinterface.cpp
~~~
            case EventGadget::Note: {
                if (!event.hasNote())
                    return;
                emit noteReceived(event.note());
            } break;
~~~
```

When the `Note` event occurs, the function first checks if the event contains a note using the `hasNote()` method. If a note is present, the `noteReceived` signal is emitted with the note's information. This signal's emission triggers all QML components that are subscribed to the `noteReceived` signal. Consequently, these components receive notification of the event and can respond accordingly.

4.3.2 QML Components in the Client Library

This section focuses on the custom QML components included in the library. These components are designed to quickly construct user interfaces that dynamically respond to server-sent events. For instance, the `ClapWindow.qml` component enables the application window to automatically respond to events like `GuiShow` , `GuiHide` , and `GuiSetTransient` . This component essentially extends the functionality of Qt's native `ApplicationWindow` QML type. A snippet of its implementation is as follows:

```
// src/clapcontrols/ClapWindow.qml

import QtQuick
import QtQuick.Controls.Basic

import Clap.Interface
import Clap.Controls

ApplicationWindow {
    id: ctrl
    ~~~
    Connections {
        target: ClapInterface
        function onVisibleChanged() {
            ctrl.visible = ClapInterface.visible;
        }
        function onTransientParentChanged() {
            ctrl.transientParent = ClapInterface.transientParent;
        }
        function onStateChanged() {
```

```
            ctrl.pluginActive = QClapInterface.state == QClapInterface.Active;
        }
    }
    ~~~
}
```

In `ClapWindow.qml`, the `ClapInterface` is utilized to control the window's visibility, parent, and state in response to received events. The `Connections` block is set up to monitor changes in `ClapInterface`'s properties such as visibility and transient parent. As these properties change, corresponding functions within the `Connections` block update the `ApplicationWindow` accordingly. This setup ensures that the UI accurately reflects the current state and responses of the underlying plugin logic.

The component is straightforward to use in applications. By importing `Clap.Controls`, users can easily implement `ClapWindow` in their QML code:

```
import Clap.Controls

ClapWindow {
    id: root
    title: "My CLAP GUI!"
    width: 640; height: 480
}
```

This simple implementation is sufficient for a basic plugin where the window's visibility and behavior are managed by the host application.

The `Dial` component is an essential element in audio plugins, serving as a core interaction facility. It quickly responds to parameter-related events from the host, enabling automation and user interaction with the plugin's parameters. The component's design allows users to customize its visual representation to suit their specific needs. The functionality of `ClapDial` demonstrates the library's ability to seamlessly integrate UI components with the data received from their corresponding audio plugin. This integration ensures a synchronized relationship between user actions and the plugin's responses, highlighting the effectiveness in maintaining dynamic interaction within audio plugins.

In the `ClapDial` QML component, connectivity to the `ClapInterface` singleton is established to act on incoming signals:

```
// src/clapcontrols/ClapDial.qml
import Clap.Interface
import Clap.Controls

Dial {
    id: ctrl
    Connections {
        target: ClapInterface
        function onParamChanged() {
            ctrl.param = ClapInterface.param(paramId);
            ctrl.value = ctrl.param.value;
            ctrl.modulation = ctrl.param.modulation;
        }
        function onParamInfoChanged() {
            ctrl.paramInfo = ClapInterface.paramInfo(paramId);
            ctrl.value = ctrl.paramInfo.defaultValue;
```

```
            ctrl.from = ctrl.paramInfo.minValue;
            ctrl.to = ctrl.paramInfo.maxValue;
        }
~~~
    onValueChanged: {
        if (ctrl.pressed || ctrl.hovered) {
            ClapInterface.enqueueParam(ctrl.paramId, ctrl.value);
        }
    }
~~~
```

The component listens for the `onParamChanged` signal, which is emitted in response to updates from the host application. Upon receiving this signal, `ClapDial` updates its properties to reflect these changes. QML's robust property system ensures that these updates are seamlessly managed through bindings.

For interactions, such as a value change in the dial, the component checks whether it is currently pressed or hovered over. This check helps determine if the user is actively interacting with the dial. When an interaction is detected, the updated value is enqueued to be sent to the server-library. This process ensures that the updates are incorporated into the audio-processing section of the plugin.

This approach effectively abstracts the communication process, allowing users of the client library to focus primarily on the UI's visualization. It aligns with standard practices in other Qt-application development. Users are not burdened with the intricacies of server communication but have the option to engage with these events if they choose.This design enhances user experience by providing a robust default setup, which users can opt to customize according to their preferences. It streamlines the process of integrating interactive UI components while ensuring efficient communication with the server.

4.3.3 Example Plugins: Implementing a Gain Plugin

This final section demonstrates the practical application of the previously discussed libraries in constructing a simple audio plugin. A perfect example for this demonstration is a **Gain** plugin. The basic implementation of such a plugin was briefly touched upon in the clap-section, where we utilized the `clap` library for a rudimentary setup. This initial implementation, while covering the basics, lacked a user interface (UI), which is typically quite labor-intensive to develop without external libraries.

Below is the build file for the Gain plugin, illustrating how the proposed libraries are integrated into its construction:

```
cmake_minimum_required(VERSION 3.2)

project(QtClapPlugins
    VERSION 0.0.1
    LANGUAGES CXX
)

include(../cmake/autolink_clap.cmake)

set(target ${PROJECT_NAME})
add_library(${target} SHARED
    clap_entry.cpp
    reveal/revealprocessor.h
    reveal/revealprocessor.cpp
    gain/gainprocessor.cpp
```

```
    gain/gainprocessor.h
)

target_link_libraries(${target} PRIVATE clap-remote)
create_symlink_target_clap(${target})

# Add GUIs
add_subdirectory(reveal/gui)
add_subdirectory(gain/gui)
```

After setting up the project and including the `autolink_clap.cmake` script, the Gain plugin's build configuration is established. This script simplifies the process by automatically creating symlinks for the CLAP library and renaming the generated shared library to comply with CLAP specifications.

Essential to this setup is the `clap_entry.cpp` file, which includes the necessary CLAP entry point `clap_entry` and the `clap_plugin_factory` implementation.

The Gain plugin, along with another plugin called `Reveal`, is part of this CLAP suite. Both plugins are directly integrated into the plugin layer, handling the processing callback. The build file then specifies subdirectories for their GUIs, which are standalone executables capable of operating independently or in conjunction with the host application. When connected to the server, these GUIs respond to the incoming events.

The `Gain` class, derived from `CorePlugin`, is structured as follows:

```
#ifndef GAINPROCESSOR_H
#define GAINPROCESSOR_H

#include <plugin/coreplugin.h>

class Gain final : public CorePlugin
{
public:
    Gain(const std::string &pluginPath, const clap_host *host);
    static const clap_plugin_descriptor *descriptor();

private:
    bool init() noexcept override;
    void defineAudioPorts() noexcept;

private:
    uint32_t mChannelCount = 1;
};

#endif // GAINPROCESSOR_H
```

In this declaration, the `init` and `defineAudioPorts` methods are overridden, and a member variable for tracking the audio channel count is defined. The constructor passes necessary details to the `CorePlugin` base class:

```
Gain::Gain(const std::string &pluginPath, const clap_host *host)
    : CorePlugin(Settings(pluginPath).withPluginDirExecutablePath("Gain"),
        descriptor(), host, std::make_unique<GainModule>(*this))
{}
```

In the Gain plugin, a `Settings` object is instantiated to specify the path to the plugin's executable. If this path points to a valid executable, the base class activates the GUI feature. The plugin also passes its descriptor and the host details to the `CorePlugin` base class. Additionally, the root module for the plugin, `GainModule`, is created, focusing on the processing aspect.

The modular approach in the plugin separates processing logic from the plugin layer. This separation, though not the primary focus during the library's development, facilitates a more organized structure. The primary emphasis was on establishing smooth communication. The `GainModule`, responsible for the plugin's processing, also defines parameters for host exposure:

```
// The GainModule controls all the processing of the plugin.
class GainModule final : public Module
{
public:
    explicit GainModule(Gain &plugin) : Module(plugin, "GainModule", 0)
    {}

    void init() noexcept override
    {
        mParamGain = addParameter(
            0, "gain",
            CLAP_PARAM_IS_AUTOMATABLE | CLAP_PARAM_IS_MODULATABLE | CLAP_PARAM_REQUIRES_PROCESS,
            std::make_unique<DecibelValueType>(-40.0, 40.0, 0)
        );
    }

    clap_process_status process(const clap_process *process, uint32_t frame) noexcept override
    { /* Same as the 'mini_gain' example */ }

private:
    Parameter *mParamGain = nullptr;
};
```

In this implementation, the `gain` parameter is defined and made available to the host. The process callback's details, similar to those in the mini_gain example, are omitted for brevity. This setup provides a fundamental yet functional plugin structure that can be used in host applications.

The GUI for this plugin is developed as a separate project. The build configuration for the GUI project is detailed below, showcasing the modular nature of the development process. This structure not only simplifies the creation of plugins but also ensures that each component, whether DSP or UI, can be developed and maintained independently.

In the build file for the Gain plugin's GUI component, the setup is similar to that of regular Qt applications:

```
find_package(Qt6 REQUIRED COMPONENTS Core Qml Quick)
qt_standard_project_setup(REQUIRES 6.5)

set(target Gain)
qt_add_executable(${target} main.cpp)

set_target_properties(${target}
    PROPERTIES
```

```
        WIN32_EXECUTABLE TRUE
        MACOSX_BUNDLE TRUE
)

qt_add_qml_module(${target}
    URI "GainModule"
    VERSION 1.0
    QML_FILES "Main.qml"
)

target_link_libraries(${target}
    PRIVATE
        Qt6::Quick
        clapinterfaceplugin
        ClapControls
)

create_symlink_target_gui(${target})
```

Here the necessary Qt modules are first searched. The project then declares a regular (qt) executable, with the unique aspect of linking against the `clapinterfaceplugin` and `ClapControls` libraries. The primary entry point for the Gain plugin's GUI is defined in the `main.cpp` file:

```
int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine;
    engine.loadFromModule("GainModule", "Main");
    if (engine.rootObjects().isEmpty())
        return -1;
```

In this segment, a `QGuiApplication` object is instantiated, and the `QQmlApplicationEngine` is set up to load the primary QML file from the "GainModule". This structure is typical of Qt applications, where the Qt event loop is enabled by default, given that the application runs as a separate process.

The next part of the `main` function introduces specific functionalities for the plugin:

```
    QCommandLineParser parser;
    parser.process(app);
    const auto args = parser.positionalArguments();
    if (args.length() == 2) {
        auto *interface = engine.singletonInstance<QClapInterface*>("Clap.Interface","ClapInterface");
        if (interface == nullptr) {
            qFatal() << "Unable to find ClapInterface instance";
            return -1;
        }
        interface->connect(args[0], args[1]);
    }
    return QGuiApplication::exec(); // Start the event loop.
```

In this section, a `QCommandLineParser` is used to process the arguments passed to the application. The expectation is that these arguments will include the `url` and the `hash` of the plugin. If the arguments are provided,

the GUI connects to the server via the `QClapInterface`. This connection initializes the interaction between the GUI and the server. Finally the event loop is started.

The final piece of the implementation is the `Main.qml` file, which leverages the custom components developed for interaction with the server and host application. This setup illustrates a complete, functional plugin GUI, integrated into a standard Qt application framework, yet tailored to meet the specific demands of the CLAP standard.

```
~~~
import Clap.Controls

ClapWindow {
    id: root
    title: "QGain"
    width: 640; height: 480

    ColumnLayout {
        anchors.fill: parent
        anchors.margins: 10
        spacing: 10

        Text { ~~~ }
        ClapDial {
            id: dial
            paramId: 0
            Layout.alignment: Qt.AlignHCenter | Qt.AlignVCenter
            Layout.preferredHeight: parent.width / 3
            Layout.preferredWidth: parent.width / 3
        }
        Image { ~~~ }
    }
}
```

The Gain plugin UI starts with a `ClapWindow` that organizes its components in a `ColumnLayout`. This layout features a text label, a `ClapDial` with the parameter ID `0`, corresponding to the GainModule's parameter, and an image. The interface is straightforward but fully equipped for communication with host applications.

Upon completion, both the QGain and QReveal plugins are tested in Bitwig Studio, a professional digital audio workstation (DAW).

Figure 24: QGain Plugin Hosted

fig. 24 illustrates the QGain plugin in operation within Bitwig Studio. The plugin's user interface, centered around an interactive dial, effectively synchronizes with the audio processor. Changes made via the UI are reflected in the processor's values and vice versa, ensuring a cohesive link between the plugin's interface and its audio processing functionality. Managed by the `ClapWindow` component, the UI is seamlessly integrated into Bitwig Studio, providing responsiveness as requested by the host.

Figure 25: QReveal Plugin Hosted

fig. 25 depicts the QReveal plugin operating in Bitwig Studio. This plugin specializes in monitoring MIDI notes. It actively visualizes these notes along with relevant metadata, providing a dynamic and informative view of the MIDI activity on the track.

Chapter 5: Conclusions

Event System Performance Analysis

This section focuses on evaluating the performance of the CLAP-RCI event system. A series of tests were conducted to simulate a high volume of events and measure the system's efficiency in processing and delivering these events to the client. The key steps in the benchmarking process were:

1. **Client Initialization**: The client, running as a separate process, connects and initiates event polling, marking the event system as operational.
2. **Event Generation and Dispatch**: Events are generated and enqueued in the `pluginToClientsQueue`. These steps involve:
   - Wrapping events in a specific type for processing.
   - The polling callback aggregates these events into a gRPC streaming message.
   - The message is serialized and transmitted via the HTTP/2 channel.
3. **Event Reception and Processing**: The client receives the serialized messages, converts them back into gRPC format, and performs trivial computations with the data.

To accurately gauge the time taken for a complete cycle of event processing, from generation to client reception, timestamped messages were introduced at the beginning and end of the event stream. The `ServerEvents` message format was extended with an optional `TimestampMsg` field to facilitate this measurement:

```
message TimestampMsg {
  int64 seconds = 1;
  int64 nanos = 2;
}
~~~
message ServerEvents {
  repeated ServerEvent events = 1;
  optional TimestampMsg timestamp = 2;
}
```

The performance of the CLAP-RCI event system was assessed under various load conditions. The aim was to examine the system's behavior with different levels of queue occupancy, ranging from tightly packed message streams in high-load situations to streams with fewer messages.

To push the system's limits, the polling frequency was set to **0**. This setting means the system would operate without any imposed delays between callbacks. This setup allowed for a direct evaluation of the system's performance under continuous, high-speed operation.

```
// server/shareddata.h
static constexpr uint64_t mPollFreqNs = 0;
```

The benchmark then works with nested loops, where we try to occupy the event processing in a more or less controlled manner:

```
// tests/bench/bench_clap_rci.cpp
~~~
    child.startChild();
    while (sharedData->nStreams() <= 0) ; // Busy wait

    clockedEvent();
    for (uint64_t i = 0; i < iterations; ++i) {
        for (uint64_t k = 0; k < eventsPerIteration; ++k) {
```

```
            while (!sharedData->pluginToClientsQueue().push(ServerEventWrapper(TestEvent))) ;
        }
    }
    clockedEvent();
~~~
```

In this setup:

1. The client is first initiated and the system awaits its successful connection.
2. A time-stamped event marks the start of event generation.
3. The loop iterates `i` times, each iteration attempting to enqueue `k` events into the queue.
4. The events are wrapped in `ServerEventWrapper` and pushed to `pluginToClientsQueue`.

The experiments conducted with this structure indicated a degree of control over the utilization of the stream size. However, it was observed that it is not always guaranteed to enqueue exactly `k` events before the event callback intervenes and clears the queue. This aspect of the benchmark highlights the dynamic and somewhat unpredictable nature of event handling under different load scenarios.

In the client-side processing of the benchmark test for the CLAP-RCI event system, the focus shifts to reading the incoming stream and marking time-stamps similarly to the server. The client-side code proceeds as follows:

```
~~~
    Stamp tBegin = Timestamp::stamp();
    while (stream->Read(&serverEvents)) {
        bytesWritten += serverEvents.ByteSizeLong();
        messageCount += static_cast<uint64_t>(serverEvents.events_size());
        if (serverEvents.has_timestamp()) {
            stamps.emplace_back(
                serverEvents.timestamp().seconds(), serverEvents.timestamp().nanos()
            );
        }
    }
    Stamp tEnd = Timestamp::stamp();
~~~
```

In this sequence:

1. The client begins by recording the initial timestamp (`tBegin`).
2. It then continuously reads from the stream, counting the number of messages received (`messageCount`) and the total bytes (`bytesWritten`).
3. For each message with a timestamp, the client stores these in the container (`stamps`).
4. The reading process continues until the host terminates the connection.
5. Finally, the client marks the end timestamp (`tEnd`).

Following the stream reading and data accumulation, the client calculates several metrics:

```
~~~
    const auto serverRtt = tEnd.delta(stamps[0]);
    cout << "Server RTT: " << serverRtt.toDouble() << " s" << endl;
    cout << "Mbps: " << (static_cast<double>(bytesWritten * 8)) /
                            serverRtt.toDouble() / 1e6 << endl;
~~~
```

The benchmark test can be configured to run with varying values for `iterations` and `eventsPerIteration`, evaluating the performace under various load conditions.

In the benchmark results for the CLAP-RCI event system, two charts display the performance under different loads. Each benchmark invocation ran 1,000 iterations, with each iteration handling 1 to 64 messages. The key findings are presented in graphs showing average time per message and average throughput.

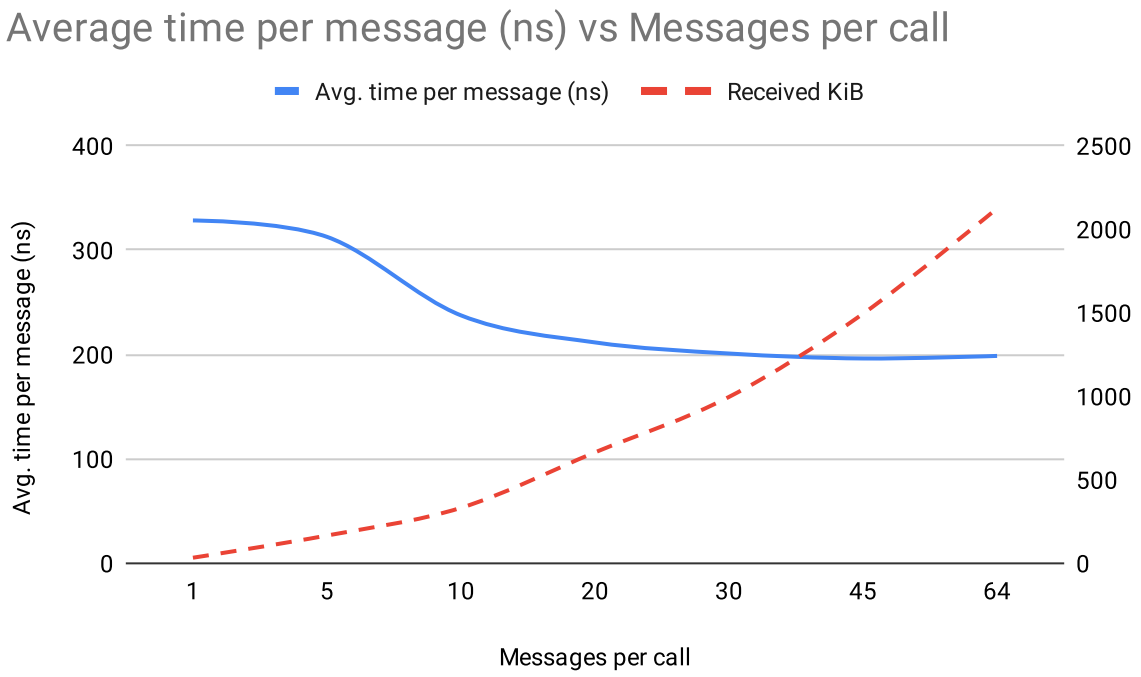

Figure 26: Benchmark results: Average time per message

The first graph shows the average time per message in nanoseconds. Performance initially drops to about 330 ns when the queue is underutilized with only one event. However, as more events are added to the queue, performance improves, with the best average time around 200 ns achieved at 30 events in the stream.

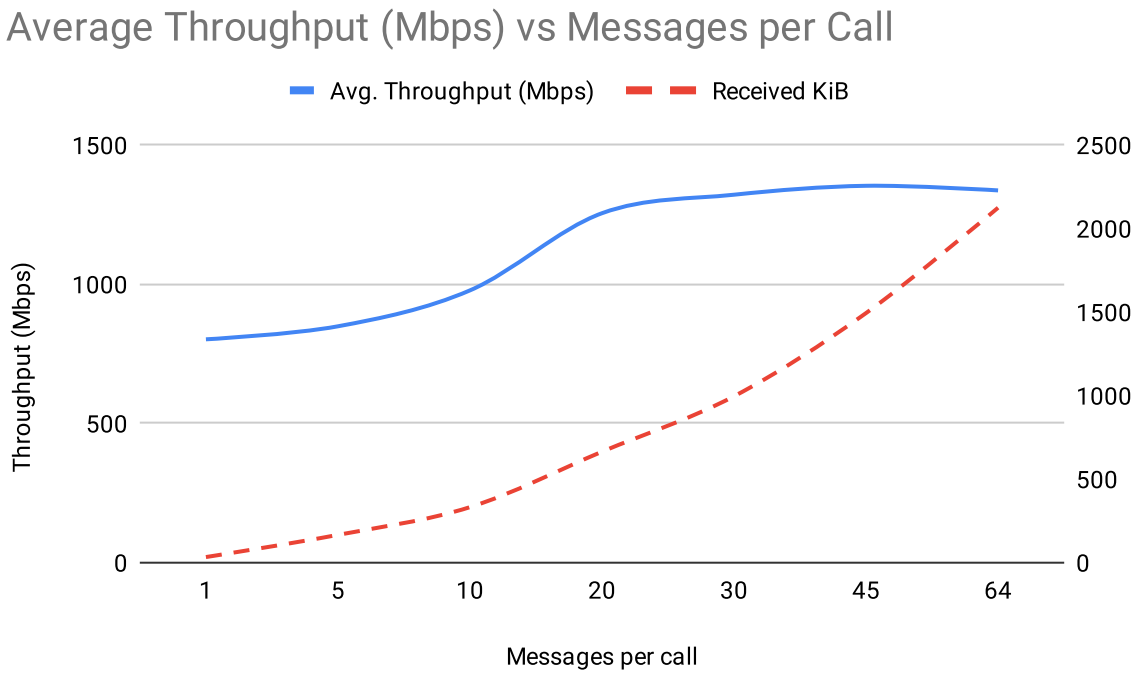

Figure 27: Benchmark results: Average throughput

The second graph illustrates the throughput measured in megabits per second (Mbps). It mirrors the previous graph's trend, peaking at around 45 messages per stream with a throughput of approximately 1300 Mbps or 1.3 Gbps. Beyond this point, the throughput stabilizes, suggesting an optimal message count for efficient processing.

Final Words on Development Experiences

The development of the CLAP-RCI library and its client-side Qt implementation has proven successful in facilitating efficient communication between the plugin side and user interfaces. By opting to launch the graphical user interface as a separate process, many of the challenges previously encountered were effectively resolved. The example plugins created using the Qt CLAP client interface demonstrate a seamless native development experience, similar to crafting a typical graphical user interface. This approach simplifies the complexity of the already streamlined CLAP interface and provides a robust default for users, allowing them to customize their interaction with the events they need to handle.

The decision to separate the problematic (Qt) component from the rest of the system has proved its worth, enabling the server implementation to leverage the language-agnostic capabilities provided by gRPC and protobuf. This separation of the GUI, or more accurately the clients, introduces a novel way to interact with audio plugins, where visualization responsibilities are entirely deferred to the client, freeing the server from these duties. Although currently limited, future development could enable multiple clients to connect to the same plugin simultaneously. This would allow for remote control of the plugin from various devices such as smartphones, tablets, and computers, expanding the range of interactive possibilities.

However, it's important to acknowledge that these libraries are still evolving and are not yet at an optimal stage of development. The API requires further refinement to offer a consistent and stable user experience, not even touching aspects like binary compatibility. Additionally, several features are yet to be implemented. Despite these areas for improvement, this research has validated the effectiveness of the methods presented, laying a solid foundation for future enhancements and investigations in this field.

Chapter 6: Acknowledgements

I would like to express my deepest gratitude to all those who have been a part of this project. The journey of this research, which started even before this thesis, initially emerged as a "Proof of Concept." Over time, it transformed into a remarkably intricate and surprisingly fulfilling endeavor.

> "In every small detail lies the possibility to exceed beyond what we imagine."

I am immensely grateful to *The Qt Company*. Their trust and the flexibility they offered were crucial for this project. The countless hours I devoted to tackling these challenges and exploring various topics during my work time were made possible because of their constant support. This project would not have been possible in its current form without their backing.

My sincere thanks also go to the *Cologne University of Applied Sciences*, particularly *Prof. Dr. Arnulph Fuhrmann*. His guidance allowed me to navigate my research path freely, adapting to the evolving directions of the study. His insightful feedback in our discussions greatly contributed to the progress of this research.

A special thanks goes to *Dr. Cristián Maureira-Fredes*. He played an instrumental role in introducing me to the Qt Company and mentored me throughout my internship, consistently cherishing a supportive environment. His patience on my various "ideas" (read: crazy thoughts) was invaluable. He helped mold these ideas, which otherwise might have remained mere thoughts. I recall one of our initial discussions at the Qt office in Berlin, revolving around this very topic of research:

> How come, that Qt user interfaces are not available for audio plugins? It would be such a good fit.

His response was encouraging: "Perhaps it's up to **you** to make it happen. It's all there for the taking." And here we are. I am truly grateful for his trust and the constant assistance he provided, whether through reviewing my documents or enlightening me with his knowledge. His mentorship has been extraordinary, and I am immensely thankful for it.

I also want to express special gratitude to *Alexandre Bique*. Our paths crossed at SUPERBOOTH23, where our conversation quickly evolved into in-depth discussions about plugins, GUIs, and how we imagined the future of them. Alexandre's work at Bitwig, particularly as the inventor and lead developer of the **C**Lever **A**udio **P**lugin standard, has been pivotal. He consistently provided a flow of new ideas, was receptive to mine, and provided significant guidance that helped shape this project. I am truly thankful for his support and collaboration in this journey.

I also wish to acknowledge several colleagues for their remarkable support. *Tor Arne Vestbø's* input during our initial exploration of this topic at a Qt hackathon laid the groundwork for this research. *Ivan Solovev's* assistance in navigating complex C++ concepts and his invaluable feedback on my ideas were pivotal in refining this project. *Marc Mutz's* extensive knowledge and insightful discussions greatly accelerated my learning and understanding of various technologies. *Fabian Kosmale's* guidance through the deep internals of Qt's event sytem, his help on the varios questions I had and ultimately suggesting QtGrpc as a foundation of this research.

Finally, I extend my heartfelt gratitude to the QtGrpc team, especially to *Tatiana Borisova* and *Alexey Edelev*. They are the inventors of QtGrpc and Qt Protobuf, and key figures in their ongoing development and maintenance. Throughout the research process, their guidance was pivotal in steering the project towards success. Their profound knowledge in these fields continuously inspired and supported me, proving to be an invaluable resource.

I consider myself extremely lucky to have worked alongside such an amazing and talented group of people. This project has thrived on the valuable feedback and guidance provided by the world's leading experts in this area of research, profoundly shaping and enriching its evolution.

Bibliography

1. Apple: Audio unit v3 plug-ins, https://developer.apple.com/documentation/audiotoolbox/audio_unit_properties, last accessed 2023/10/23.
2. Benyamin Shafabakhsh, S.H., Robert Lagerström: Evaluating the impact of inter process communication in microservice architectures. (2020).
3. James Davis, H.-C.L., Yi-Hsuan Hsieh: Humans perceive flicker artifacts at 500 hz. (2015).
4. Joseph Timoney, R.V., Victor Lazzarini: Approaches for constant audio latency on android, https://mural.maynoothuniversity.ie/6664/, last accessed 2023/11/14.
5. Lakhani, M.: Protocol buffers: An overview (case study in c++). (2014).
6. Luca Turchet, C.F.: Elk audio OS: An open source operating system for the internet of musical things. (2020).
7. Scavone, G.P.: RtAudio: A cross-platform c++ class for realtime audio input/output, https://ccrma.stanford.edu/~gary/papers/icmc2002c.pdf, last accessed 2023/11/14.
8. Srđan Popić, D.P., Bojan Mrazovac: Performance evaluation of using protocol buffers in the internet of things communication. (2016).
9. Vandevoorde, D.: Plugins in c++, https://www.open-std.org/JTC1/sc22/wg21/docs/papers/2006/n2074.pdf, last accessed 2023/07/23.